

Wednesday, LV 1 (2016-08-31)

September 13, 2016

1 Info

“Problems” are problems in the sense of algorithmical problems, not exercises one should “do”.

Hand in exercises will be given weekly, these are voluntary but strongly recommended. (The first hand in exercise is due Wed. 2016-09-07.)

Exercise sessions will be given and they will be concentrated on exercises from the textbook. All three slots given are identical and you are asked to sign up on a doodle to spread the attendance.

2 Problem

An instance is a concrete input of logic.

Given: x, y (integers)
Compute: $x+y$ and $x*y$

Which is the easier operation?

Most people would probably say addition, but in what sense is that?

3 Algorithm

- An instruction: How to solve a problem (it must work for all instances of a problem).
- Unambiguous
- The work is split into small, simple, steps.

Myth One might think that developing algorithms is programming, but this is wrong. An algorithm is not a program.

4 Time Complexity

Time complexity is the limiting factor of algorithms. You can also think about the space required, but the time limit is harder, hence we will concentrate on time complexity.

The time complexity of an algorithm is its running time for every instance. $t: X \rightarrow \mathbb{R}^+$ (where X is a set of instances). However, this is far too complicated.

4.1 Worst case analysis: $\mathcal{O}(n)$

$t: \mathbb{N} \rightarrow \mathbb{R}^+$

maximum running time for instances of length n . The “time” is not counted in seconds, but in elementary operations. We also ignore constant factors.

Definition Let t and f be two functions, then t is $\mathcal{O}(f)$ if: $\exists c > 0, n_0 : \forall n > n_0 : t(n) \leq c * f(n)$

4.1.1 Examples

x, y : integers with n digits.

We can compute $x+y$ in $\mathcal{O}(n)$ time. (We need constant time to add two digits and we do this n times.) The reason is that the result depends on every digit in the input. Unlike a binary search of a sorted array, which we can search in $\mathcal{O}(\log(n))$ time. (More about this later I guess.)

We can add m integers with n digits in $\mathcal{O}(mn)$, but this is hard to prove! Note that the result of adding a column could be as bad as $9*m$. This carry over constant has $\mathcal{O}(\log(m))$ digits, so in the worst case scenario we get an addition of $\mathcal{O}(\log(m))$ operations per digit we want to calculate. This would yield: $\mathcal{O}(mn * \log(m))$.

We can compute $x*y$ in $\mathcal{O}(n^2)$ time. This is not optimal, due to the digits not being completely independent of each other when multiplying. (Take a look at Big-O Arithmetics in general and Fürers Algorithm in particular.)

4.2 Properties of \mathcal{O}

- $\mathcal{O}(f(n) + g(n))$ is the same as $\mathcal{O}(\max\{f(n), g(n)\})$.
- If: f monotone increasing function, $c > 0$ (constant), $a > 1$ (constant) $\Rightarrow f(n)^c$ is $\mathcal{O}(a^{f(n)})$
- $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$ because they differ only by a constant factor.

Monday, LV 2 (2016-09-05)

1 Interval Scheduling

Given a set of n intervals $[s_i, f_i]$, $i=1, \dots, n$, on the real axis. Select a subset X of these intervals, as many as possible, which are pairwise disjoint.

An exhaustive solution (check everything) would take $\mathcal{O}(n * 2^n)$ time.

However, if we put a certain interval $x=[s_i, f_i]$ in X , we can remove all intervals intersecting with x .

1.1 Attempt 1

Put $[s_i, f_i]$ with smallest s_i in X .

This might seem reasonable, but it is a bad attempt. Visualize a very long interval $[1, 8]$ and two shorter $[2, 3]$, $[4,5]$, we immediately see that it is better to choose the two shorter intervals, but our rule would fail to do this. Attempt 1 does not take into account, the length of the intervals.

1.2 Attempt 2

Put $[s_i, f_i]$ with smallest $f_i - s_i$ in X .

Visualize $[1,5]$, $[4,7]$, $[6, 10]$ and you will see that we should have picked the two longer intervals. However, we picked the shortest – and only the shortest – interval.

1.3 Attempt 3

Put $[s_i, f_i]$ in X which intersects the smallest number of intervals. (See P 117, Fig 4.1 in the book for counter example.)

1.4 Attempt 4 (Earliest End First, EEF)

Sort the intervals s.t. $f_1 < f_2 < \dots < f_n$. Put $[s_i, f_i]$ with smallest f_i in X . Then remove all intervals intersecting with x . Iterate.

1.4.1 Claim

There exists some optimal solution, Y , containing $[s_1, f_1]=x$.

Claim \Rightarrow EEF correct. This is because, after picking the first interval (and therefore removing its intersecting intervals), picking the next interval is simply a matter of applying the rule one more time, and so on. Proof by induction!

1.4.2 Proof of Claim

Assume: Y is optimal, $x \notin Y$.

x 's f_i is somewhere within the interval y 's $[s_i, f_i]$.

x intersects exactly one $y \in Y$.

$(Y \setminus \{y\}) \cup \{x\}$ is another optimal solution.

QED.

1.5 EEF Running time

EEF runs in $\mathcal{O}(n^2)$ time. We can however make it faster by:

- Represent each interval by two copies.
- Put them into two double linked lists.
 - One of the lists is sorted by ascending f_i .
 - The other list is sorted by ascending s_i .
- Connect copies by pointers. (If you find an interval in one list, you find it in the other too.)
- Find smallest f_i in constant time (heap?).
- Intersecting intervals $[s_j, f_j]$ are those with $s_j < f_i$.

This leads us to using $\mathcal{O}(n)$ time in total for the EEF. We do however also need to sort the list. This could be done in $\mathcal{O}(n * \log(n))$, but we would just say: $\mathcal{O}(n) + \text{Time}(\text{sorting})$.

2 Interval partitioning

Given a set of n intervals $[s_i, f_i]$, $i=1, \dots, n$, on the real axis. Partition the set of intervals into the smallest possible number d of subsets X_1, X_2, \dots, X_d , each consisting of pairwise disjoint intervals.

2.1 Algorithm

Sort the intervals such that $s_1 < s_2 < \dots < s_n$.

for all i : $X_i = \emptyset$

for all $j=1$ to n

put $[s_j, f_j]$ in X_i where i is the smallest index s.t. $[s_j, f_j]$ does not intersect other results in X_i .

2.2 Proof

$d :=$ max number of intervals that share a point.

Every solution requires at least d sets $X_i = \emptyset$.

The algorithm uses only d sets (graphical proof, see Page 123 in the book).

Wednesday, LV 2 (2016-09-07)

1 Weigthed Interval Scheduling

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \dots, n$, on the real axis. Every interval has also a positive weight v_i . The intervals are sorted such that the interval that *ends* first comes first. No intervals end, or start, at the same time.

Goal: Select a subset X of these intervals which are pairwise disjoint and have maximum total weight.

1.1 EEF Fails

The algorithm we showed in the previous lecture will not be able to solve this issue. AAMOF the interval scheduling problem is a special case of the weigthed interval scheduling problem. (I.e. where v_i is the same for all intervals.) Hence, the WIS is a harder problem to solve. In order to show that the EEF fails we can look at two intervals: $n_1=[1,3,1]$ and $n_2=[2,4,10]$ where $n_i=[\text{start, stop, weighth}]$. We see that it is not beneficial to pick n_1 , but EEF would do so since n_1 ends before n_2 .

1.2 Discard the worse solutions

$\text{OPT}(j) := \max$ weight achievable by a subset of disjoint intervals from $\{[s_1, f_1], [s_2, f_2], \dots, [s_j, f_j]\}$.

In the end we want only $\text{OPT}(n)$, but we will use $\text{OPT}(j)$ to get there.

We can conclude that: $\text{OPT}(1) \leq \text{OPT}(2) \leq \dots \leq \text{OPT}(n)$.

1.2.1 Computing $\text{OPT}(j)$

The key is to use induction on j .

Induction base: $\text{OPT}(1) = v_1$

Induction step: Consider any fixed $j > 1$. Suppose that all $OPT(i)$, $i < j$ are already computed.

$$OPT(j) = \max \begin{cases} OPT(j-1) & \text{don't take } [s_j, f_j] \\ v_j + OPT(p(j)) & \text{take } [s_j, f_j] \end{cases}$$

where $p(j) := \max\{i \mid f_i < s_j\}$ (auxiliary function)

1.2.2 Time Complexity

We can compute:

- all $p(j)$ in $\mathcal{O}(n)$ time (if all f_i, s_j are sorted)
- all $OPT(j)$ in $\mathcal{O}(n)$ time (arithmetic operations with real numbers are counted as elementary)

This table uses the schedule shown in the book. (P 253, Fig 6.2)

j	1	2	3	4	5	6
OPT(j)	2	4	6	7	8	8

1.2.3 Note about recursion

The formula used is recursive, however that can be dangerous to use (see P 255, Fig 6.3). The tree of subproblems gets really big... This is because we recalculate $OPT(j)$ for many js . **If we store $OPT(j)$ in an array at position $M[j]$ we can look there and see if there is already a value for $OPT(j)$ before we (re-)calculate it.**

1.2.4 What about the subset?

When we have obtained $OPT(n)$ we know what the optimal value will be. However, we have not kept track of which subset is optimal. Lets take that into account.

We can store all the intermediate solutions as shown below:

j	1	2	3	4	5	6
OPT(j)	2	4	6	7	8	8
Subset:	{1}	{2}	{1,3}	{4}	{1,3,5}	{1,3,5}

Note that it takes a lot of time. **Storing intermediate solutions $\Rightarrow \mathcal{O}(n^2)$ time. Oh, oh..**

We try another way and ask: "Where did the optimum come from?" The mistake in the previous way of calculating the subset was that we went from

left to right. If we go from right instead...

We ask: Where did 8 come from? 5 or p(6)? It came from 5. Hence, we can discard 6. We continue and get: (*this is hard to write, check the book*)

j	1	2	3	4	5	6
OPT(j)	2	4	6	7	8	8
Subset:	pick	discard	pick	discard	pick	discard

Backtracing in order to find the solutions $\Rightarrow \mathcal{O}(n)$ time!

1.3 Dynamic Programming

One scheme fits all!

1. Choose parameters that limit the sub-instances (to a polynomial number).
2. Define a recursive function which indicates the optimal value of the sub-instances. Be careful to define *what* the function is supposed to compute.
3. Compute the function without using recursive calls. (Instead, store the values in an array and check if you have already computed the value.)
4. Do a time analysis. (This should *usually* be $:= (\text{size of table}) \times (\text{time spent on table entry}) = (\text{size}) \times (\text{constant})$.)
5. Get the actual solution by backtracing, starting from the final value. Do *not* copy all the partial solutions!

Note: You only need to prove the correctness for the recursive function. The correctness of the method is given by the method itself.

Monday, LV 3 (2016-09-12)

1 The Knapsack Problem

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

weights(sizes): w_1, w_2, \dots

values: v_1, v_2, \dots

capacity W : (integers)

Find $S \subseteq \{1, \dots, n\}$ s.t. $\sum_{i \in S} w_i \leq W$ and $\max \sum_{i \in S} v_i$.

j

1.1 Possible Greedy Solution

Sort items s.t. $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$.

Take items in this order, put them in the solution as long as possible.

It turns out this is a pretty bad idea...

Let $v_1 = 10\varepsilon, v_2 = 90, w_1 = \varepsilon, w_2 = 10, W = 10$

1.2 Subset Sum (A partial Solution)

Given a set (or multiset) of integers, is there a non-empty subset whose sum is W ?

$P(j, w) := 1$ if some subset of $\{w_1, w_2, \dots, w_n\}$ has sum w .

$P(j, w) := 0$ otherwise.

Computing the $P(j, w)$

if $j = 0$, we take something from the empty set (w is therefore also 0):

$\forall w > 0: P(0, w) = 0$

$\forall j: P(j, 0) = 1$

Induction step

Suppose we have already computed $P(i, y)$ for all $i < j, y < w$.

$P(j, w) = P(j-1, w) \vee P(j-1, w-w_j)$ $P = (\text{don't take } \vee \text{ take } w_j)$

1.2.1 Time Complexity

The algorithm runs in: $\mathcal{O}(n * W)$, which is not polynomial. In the worst case it could be exponential in the input length. We say that the algorithm is pseudo-polynomial.

1.2.2 Example Run

$w_1 = 1, w_2 = 1, w_3 = 5, w_4 = 2, w_5 = 2$
 $W = 8$

$w \setminus j$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	0	1	1	1	1	.
2	0	0	1	1	1	.
3	0	0	0	0	1	.
4	0	0	0	0	1	.
5	0	0	0	1	1	.
6	0	0	0	1	1	.
7	0	0	0	1	1	.
8	0	0	0	0	1	.

Back tracing: (4,8) -> (3,6) -> (2,1)

1.3 Subset Sum 2.0

Find a subset of $\{w_1, w_2, \dots, w_n\}$ whose sum is as big as possible, but less than W . (If we have a knapsack problem where $v_i = w_i$, this is it.)

$OPT(j, w)$: The largest number $\leq w$, which is the sum of some subset of $\{w_1, \dots, w_j\}$.

$$OPT(j, w) = \max \begin{cases} OPT(j-1, w) \\ w_j + OPT(j-1, w-w_j) \end{cases}$$

1.4 Back To Knapsack

$OPT(j, w)$: The largest value of a subset of $\{w_1, \dots, w_j\}$ with weight $\leq W$.

$OPT(0, w) = OPT(j, 0) = 0$

$OPT(j-1, w-w_j) = 0$ if $w < w_j$

$$OPT(j, w) = \max \begin{cases} OPT(j-1, w) \\ v_j + OPT(j-1, w-w_j) \end{cases}$$

2.4 Recursive formula

$$\text{OPT}(i, j) = \min \begin{cases} \text{OPT}(i-1, j) + 1 \\ \text{OPT}(i-1, j-1) + \delta_{ij} \\ \text{OPT}(i, j-1) + 1 \end{cases} \quad \text{where } \delta = \begin{cases} 1 & a_i \neq b_j \\ 0 & \text{otherwise} \end{cases}$$

$$\min \begin{cases} \downarrow & +1 \\ \swarrow & +\delta_{ij} \\ \rightarrow & +1 \end{cases}$$

2.4.1 Example

		C	R	E	A	M	Y
	0	1	2	3	4	5	6
C	1	0	1	2	3	4	5
A	2	↑1	↖1	2	2	3	4
R	3	2	↖1	←↖2	3	3	4
A	4	3	2	2	↖ 2	3	4
M	5	4	3	3	3	↖ 2	3
E	6	5	4	3	4	↑3	↖3
L	7	6	5	4	4	4	↖↑4

2.5 Time Complexity

$\mathcal{O}(nm)$

Wednesday, LV 3 (2016-09-14)

1 Segmentation

Given: A sequence (x_1, \dots, x_n) of items.

Goal: Partition the sequence into segments (x_i, \dots, x_j) so that the sum of $f((x_i, \dots, x_j))$ of all these segments are maximized/minimized.

Let e_{ij} describe the penalty, or the quality score, for the sequence (x_i, \dots, x_j) . Find a segmentation that minimizes the sum of penalties or maximizes the quality score.

1.1 OPT

Assume that we have already decided a segmentation:

1				j					n
---	--	--	--	---	--	--	--	--	---

.

$\text{OPT}(j) :=$ minimum sum of penalties of all possible segmentations of (x_1, \dots, x_j) .

$\text{OPT}(j) = \min \{e_{ij} + \text{OPT}(i-1)\}$ (Note: $i \leq j$)

1.2 Time Complexity

$\mathcal{O}(n^2)$, if the e_{ij} are already given.

1.3 Alternative Algorithm (for maximize version only)

This is very similar to the weighted interval scheduling problem.

Consider (x_i, \dots, x_j) as an interval with weight e_{ij} . Solve this instance of weighted interval scheduling. The process of reducing a problem to another problem is called "reduction".

Recall that the running time of WIS is $\mathcal{O}(n)$, however n is the number of intervals - in our segmentation problem we have n^2 possible intervals, hence the running time is still $\mathcal{O}(n^2)$.

2 Searching - Divide and Conquer

The art of dividing a set into subsets, and solving them recursively.

Given: A sorted set $S = \{s_1, \dots, s_n\}$

Goal: Find x , if x is present in S .

Example, binary search:

```
search(i, j):
  if i == j:
    return i (x is located at S[i])
  else:
    m = (i+j)/2
    if x <= S[m]:
      search(i, m)
    else:
      search(m+1, j)
```

Comparison and arithmetic operations take constant time. Each iteration we half our n , hence: $\mathcal{O}(\log(n))$.

2.1 General running time of recursive DNC algorithms - Recurrence Equations

$T(n)$:= time for instances of length n . Even though we know nothing about $T(n)$ we can look at the algorithm above and see that:

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

$$T(1) = \mathcal{O}(1) \text{ (induction base)}$$

It is also enough to consider n as powers of 2. This is because the definition of \mathcal{O} . Consider: $(bn)^d = b^d n^d = \mathcal{O}(n^d)$

$$T(n) = \mathcal{O}(\log(n))$$

3 Skyline

Given: n rectangles, having their bottom lines on a fixed horizontal line.

Goal: Output the area covered by all these rectangles (in other words: their union), or just its upper contour.

Instance: Described by (l_i, r_i, h_i) , $i=1, \dots, n$ (l is the left x-value, r is the right x-value and h is the height of the rectangle)

Output: Described by the sequence of heights and points on the x-axis where the height changes.

Approach 1: Insert rectangles one by one and update the skyline. It takes us $\mathcal{O}(j)$ time to insert the j th rectangle leading us to a total runtime of $\mathcal{O}(n^2)$.
 $T(n) = T(n-1) + \mathcal{O}(n)$.

The first approach is bad because we spend a lot of time in order to insert only one new rectangle. We could use this time more efficiently using Divide and Conquer.

Approach 2:

1. Divide the instance arbitrarily in two instances with $\frac{n}{2}$ rectangles.
2. Compute their skylines recursively.
3. Merge these skylines in linear time.

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) = \mathcal{O}(n \log(n))$$

4 Solving recurrence equations (Master Theorem)

Let: a and b be integers, $a \geq 1, b \geq 2$.

Let: c and k be reals, $c > 0, k \geq 0$.

Consider $n = b^m$

$$T(1) = c$$

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

$$T(n) = a^{2*}T\left(\frac{n}{b^2}\right) + ca\left(\frac{n}{b}\right)^k + cn^k$$

$$T(n) = a^{3*}T\left(\frac{n}{b^3}\right) + ca^2\left(\frac{n}{b^2}\right)^k + ca\left(\frac{n}{b}\right)^k + cn^k$$

$$\dots$$

$$T(n) = c \sum_{i=0}^m a^{(m-i)} \left(\frac{n}{b^{(m-i)}}\right)^k$$

$$\text{Note: } a^{(m-i)} \left(\frac{n}{b^{(m-i)}}\right)^k = a^{(m-i)} b^{ik} = a^m \left(\frac{b^k}{a}\right)$$

$$T(n) = ca^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i$$

$$\underline{a > b^k}: T(n) = \mathcal{O}(a^m) = \mathcal{O}(a^{\log_b n}) = \mathcal{O}(n^{\log_b a})$$

The base of the log is important due to being in the exponent.

$$\underline{a = b^k}: T(n) = (ma^m) = \mathcal{O}(n^{\log_b a} * \log n) = \mathcal{O}(n^k \log n)$$

$$\underline{a < b^k}: T(n) = \mathcal{O}(a^m \left(\frac{b^k}{a}\right)^m) = \mathcal{O}(b^{km}) = \mathcal{O}(n^k)$$

Algorithms: Lecture 6

Chalmers University of Technology

Recap

- Greedy & Dynamic Programming

- extend solutions from smaller sub-instances incrementally to larger sub-instances, up to the full instance.

- Divide & Conquer

- follows the pattern of reducing a given problem to smaller instances of itself

BUT

- it makes jumps rather than incremental steps.

Recap

- **Divide-and-conquer**

- Split problem instance into a few significantly smaller sub-instances.
- Sub-instances are solved, independently, in the same way (recursion).
- Combine partial solutions to sub-instances into an overall solution.

- **Most common usage**

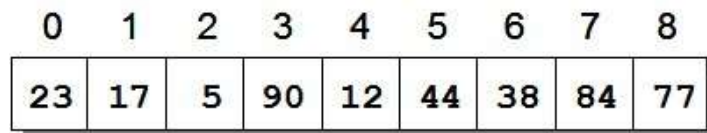
- Break up problem instance of size n into **two** equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in **linear time**.

Today's Lecture

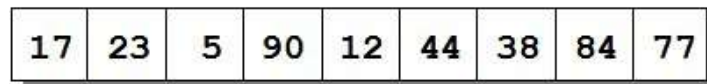
- **Important technique for Searching & Sorting**
 - Binary Search $O(\log n)$ (last lecture)
 - Brute force Sorting, e.g., Bubble sort : $O(n^2)$.
 - Divide-and-conquer: $O(n \log n)$.

Bubble Sort

- Scan the list of elements from left to right
 - whenever two neighbored elements are in the wrong order, swap them.



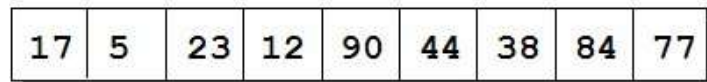
↑ exchange



↑ exchange



ok ↑ exchange



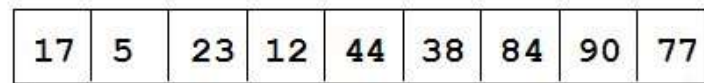
↑ exchange



exchange ↑



exchange ↑



exchange ↑



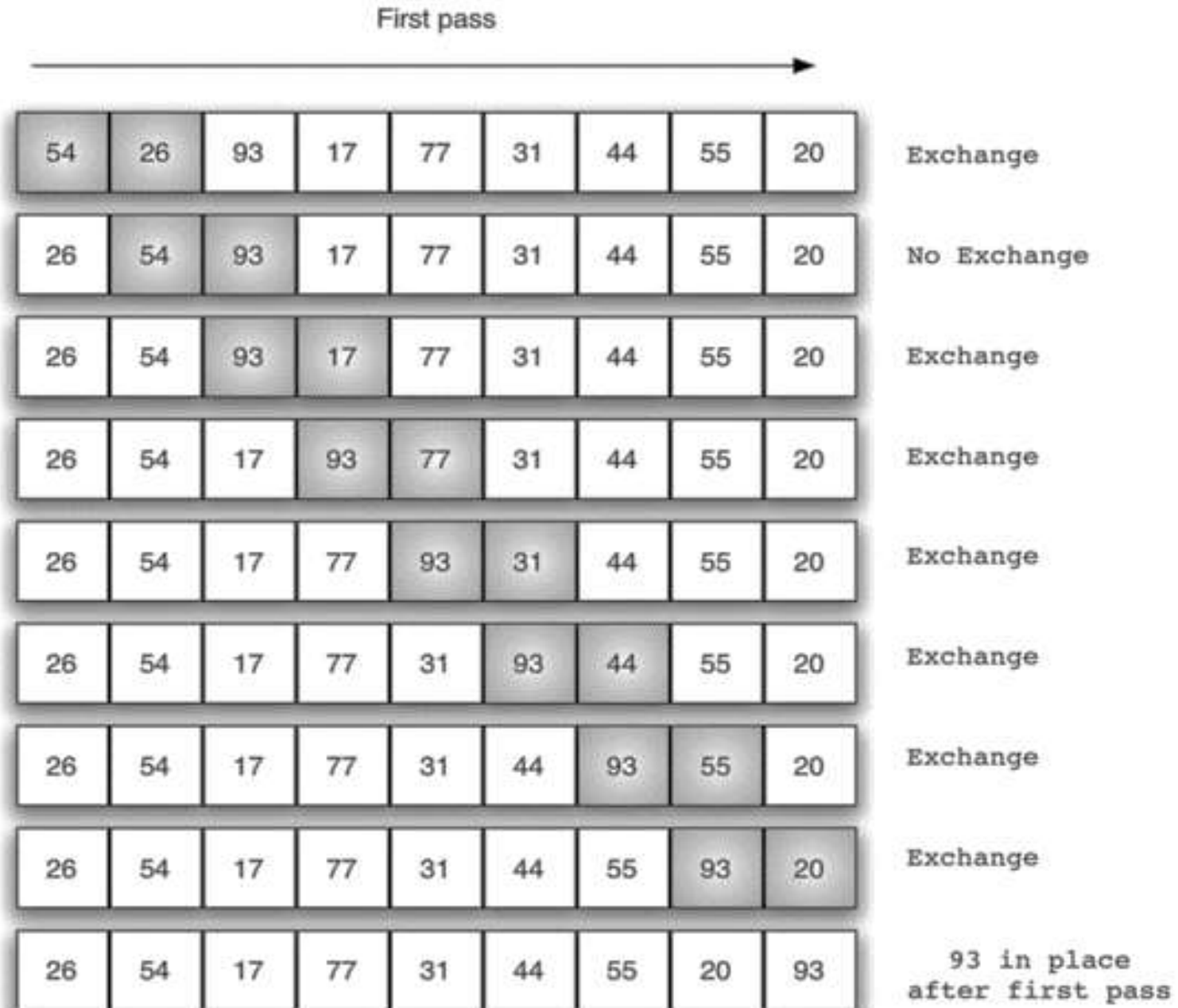
The largest value 90 is at the end of the list.

Bubble Sort

- Every pass puts one element to its proper place & reduces the instance size by 1

➤ $n(n-1)/2$

➤ $O(n^2)$

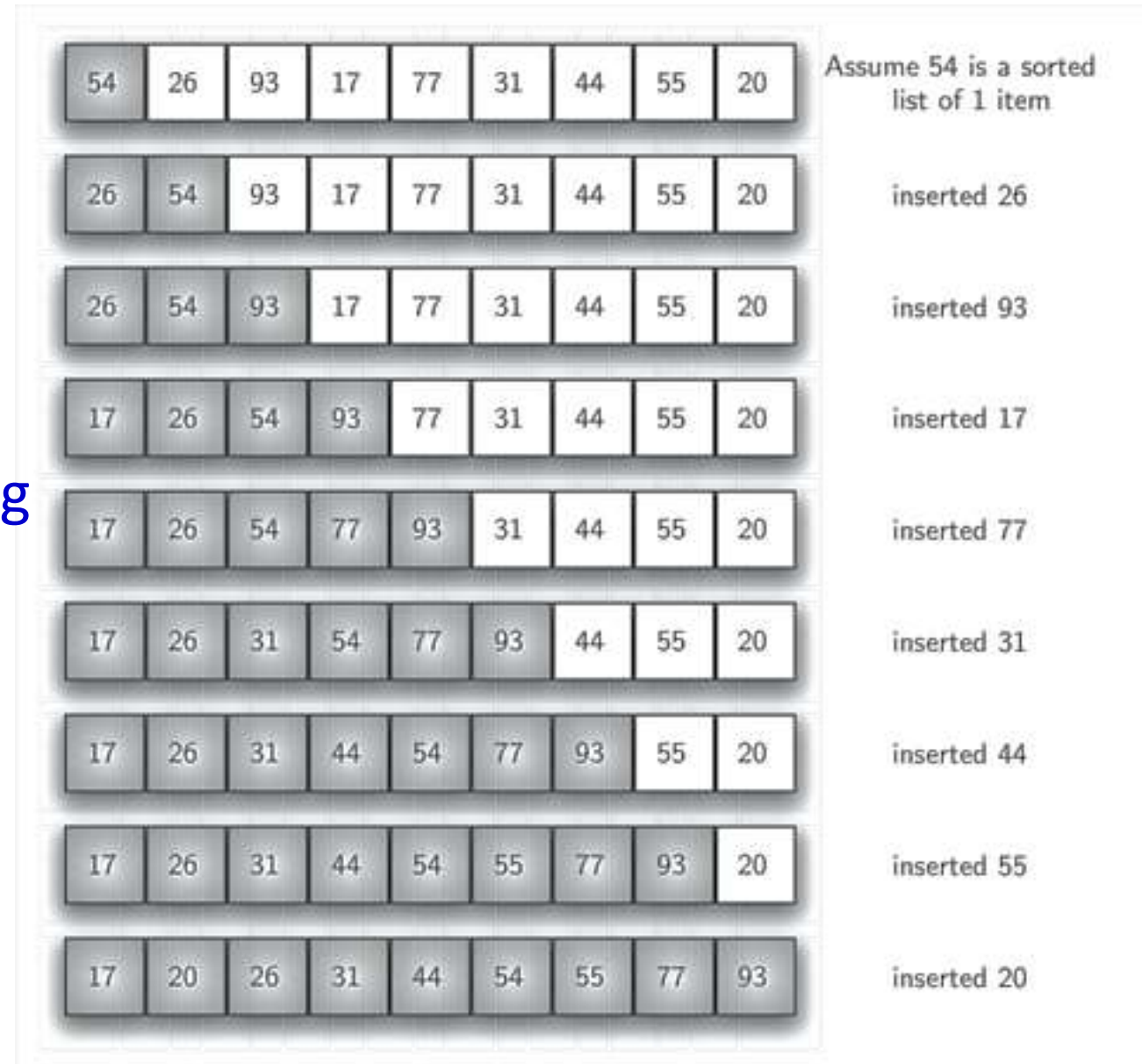


Bubble Sort

- **In place:** Needs only one array of size n for everything, except, possibly a few memory units.
- **Best:** In the first pass, if we don't have to make any swaps, that means that the array is sorted already.
- **Worst:** if many elements are far from their proper places(reverse order), because the algorithm moves them only step by step.
 - **Insertion Sort to overcome**

Insertion Sort

- After k rounds of Insertion Sort, the first k elements ($k = 1, \dots, n$) are sorted.
- To insert the $(k + 1)$ st element we search for the correct position, using binary search.
 - $O(n \log n)$?
 - we may be forced to move $O(k)$ elements in the k -th round, giving again an overall time complexity of $O(n^2)$.



Insertion Sort

- **Idea:** We can avoid moving the elements
 - Insert an element in $O(1)$ time at a desired position using **doubly linked list**.
- **But**, how do we apply **Binary Search without indices?**
 - We have to apply linear search, and once again: **$O(n^2)$** for all n rounds.
- However, **$O(n \log n)$ sorting algorithms** are known, as we already know
 - **Divide-and-conquer**

Mergesort

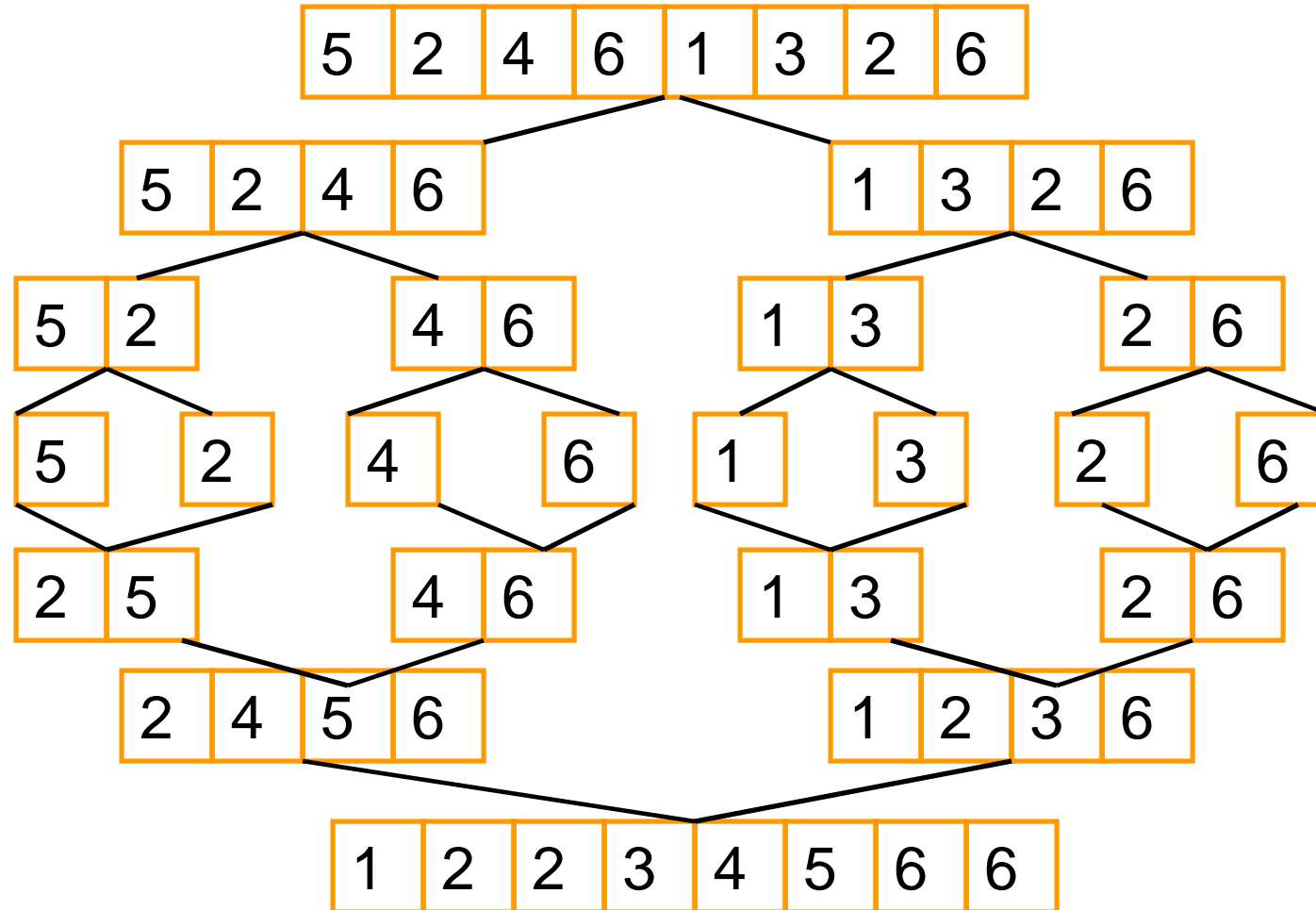
- **How it works:**

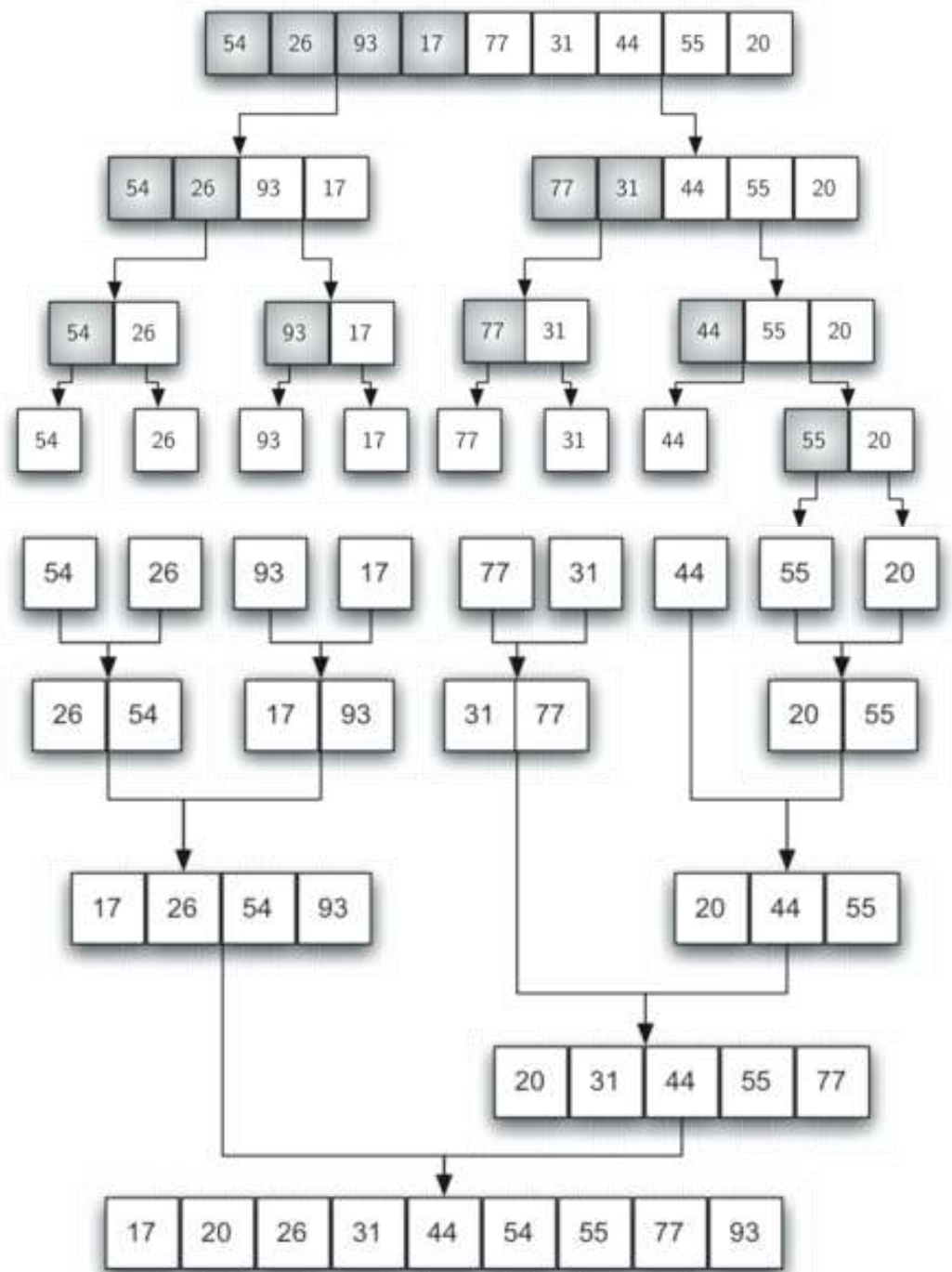
- arbitrary split the set into two halves
- **recursively sort the two halves separately**
- merge the two sorted halves

- **Merging the two sorted halves involves *comparing* the elements to each other**

- scan both ordered sequences simultaneously and always move the currently smallest element to the next position in the result sequence, **implies $O(n)$**

Mergesort Example





Time Complexity for Mergesort

Recurrence Relation:

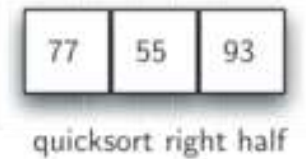
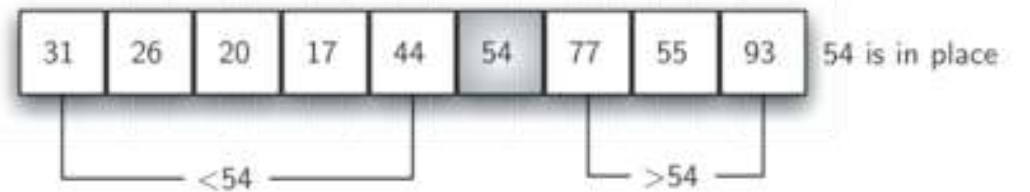
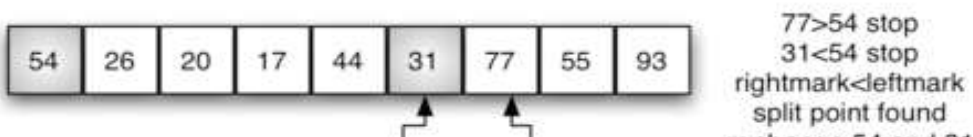
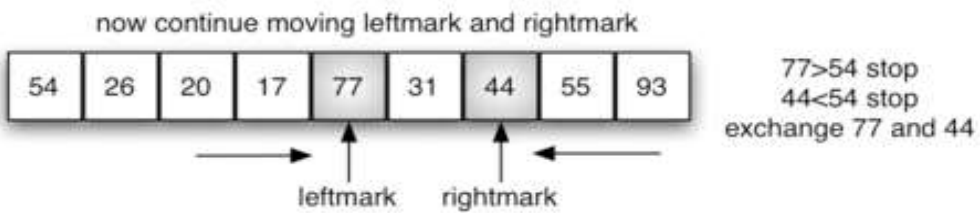
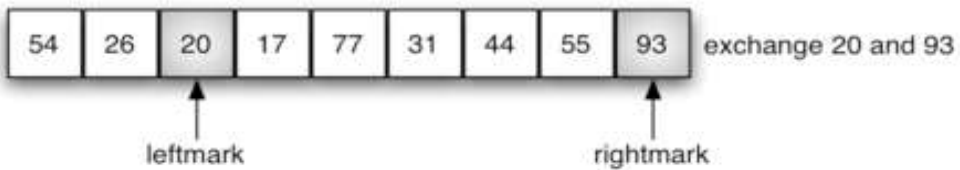
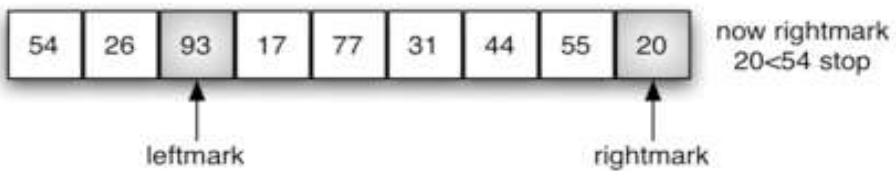
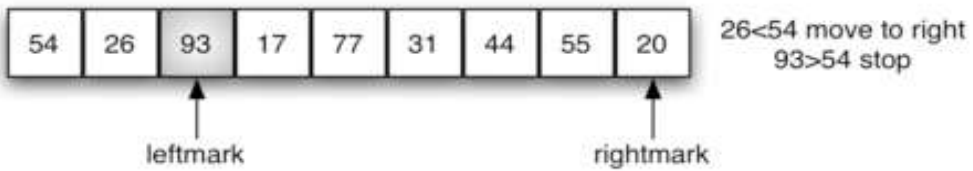
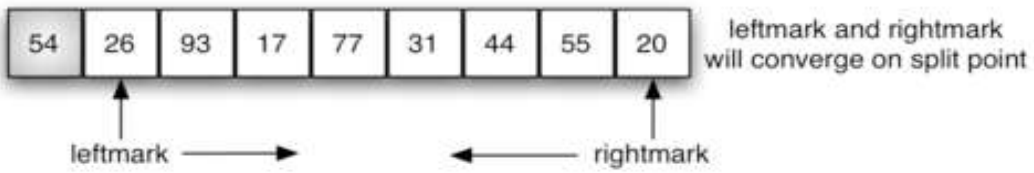
- Let $T(n)$ be worst case time on a sequence of n elements
- If $n = 1$, then $T(n) = O(1)$ (constant)
- If $n > 1$, then $T(n) = 2 T(n/2) + O(n)$
 - two sub-problems of size $n/2$ each that are solved recursively
 - $O(n)$ time to do the merge
- Solving the recurrence gives $T(n) = O(n \log n)$
- Remember general result from the Master Theorem
 - $T(n) = aT(n/b) + cn^k$, and for $a = b^k$ it gives $O(n^k \log n)$
 - For Mergesort, we have $a=2$, $b=2$ and $k=1$.

Caveat

- **Simple Structure but** not the fastest sorting algorithm in practice
 - Too many copy operations (In every merging phase on every recursion level we have to move all elements of the merged subsets into a new array.)
- **NOT *in place***
 - Additional memory required, while n could be very large in practice.
- **Other alternatives with $O(n \log n)$ time**
 - Different hidden constants factors
 - Hard to analyze theoretically
 - runtime experiments can figure out what is really faster.
- **Remark:** our Skyline algorithm from the previous lecture implicitly uses Mergesort to sort all endpoints of the rectangles.

Quicksort

- How it works:
 - choose one element to be the *pivot/ splitter*, called p
 - put all the elements $< p$, and those $> p$ in two different subsets
 - recursively sort the two subsets and concatenate putting p in between
- *In place*
- Conquer phase trivial
- Implementation of Divide makes quick sort - quick



Time Complexity for Quicksort

- **Worst case:** the splitter is always the minimum or maximum element of the set, $O(n^2)$ is needed.
- Only **careful selection** of the splitter can guarantee the better bound.
- If the splitters would **exactly halve the sets on every recursion level**, we have our standard recursion:
 - $T(n) = 2 T(n/2) + O(n)$
 - With solution: $T(n) = O(n \log n)$

Ideal Splitter for Quicksort

- **Rank of an element:** the position of this element if the set were already sorted.

- **Median:** Element with rank $n/2$



- **Computing Median?**

- Sort and read off the element of rank $n/2$
- **Stupid idea...** sorting is the actual problem for which we need to find out **Median**.

- **A splitter is selected at random!**

- the worst case (rank nearly 1 or n) is very unlikely.
- The splitters will mostly have ranks in the middle.
 - reasonably balanced partitions in two sets.
 - **$O(n \log n)$ time is needed on expectation.**

- **In practice, chose three random elements and take their median as the splitter.**

Center of a Point Set on the Line

Given: n points x_1, \dots, x_n on the real line.

Goal: Compute a point x so that the sum of distances to all given points $\sum_{i=1}^n |x - x_i|$ is minimized.

Distance: Walking or driving distance along the street, not the Euclidean distance.



Median of the given coordinates, not the **average**.



Selection and Median Finding

- **Given:** A set of n elements, where an order relation is defined, and an integer k .
- **Goal:** Output the element of rank k , that is, the k th smallest element.
- **Median:** Special case in Selection problem, $k := n/2$
 - often better suited as a “**typical**” value than the average, because it is robust against outliers.
- **Wealth in a population**
 - **Mean vs. Median**

Algorithm for Selection and Median Finding

- **Choose:** a random splitter s and compare all elements to s in $O(n)$ time to get *rank* r of s .
- **Decide:**
 - If $r > k$ then throw out s and all elements *larger than* s . REPEAT
 - If $r < k$ then throw out s and all elements *smaller than* s , and set $k := k-r$ REPEAT
 - If $r = k$ then return s . STOP
- **Time Complexity**
 - Given the splitters are always in the middle: $T(n) = T(n/2) + O(n)$
 - $T(n) = aT(n/b) + cn^k$, and for $a < b^k$ it gives $O(n^k)$
 - We have $a=1$, $b=2$ and $k=1$, therefore we get: $O(n)$
 - $O(n)$ is expected time, worst case could still be $O(n^2)$
 - **Fast Algorithm:** Intuition is that Selection needs much less information than Sorting.

Algorithm for Selection and Median Finding

- A deterministic divide-and-conquer, with $O(n)$ time exists
 - Complicated
 - More importantly, the hidden constant in $O(n)$ is large
 - Practically, random splitter algorithm is better

Information Flow and Optimal Time Bounds

- One of our primary goals is to make algorithms as fast as possible.
 - How good are our time bounds for sorting and searching algorithms?
- **Searching:**
 - Find a specific element in an ordered set of size n
 - Comparisons counted as the elementary operations
- **Binary Search:** $\log_2 n$ comparisons of elements
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
 - Claim holds due to the **information-theoretic** argument

Information Flow and Optimal Time Bounds

- **Binary Search:** $\log_2 n$ comparisons of elements
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
 - Claim holds due to the **information-theoretic** argument
 - **How much information do we gain from our elementary operation?**
 - Binary Answer (“smaller” or “larger”), splitting the set of possible results in two subsets for which either of the answers is true.
 - **worst case: the answer is true for the larger subset, always**
 - **candidate solutions are reduced by a factor at most 2**
 - **n possible solutions in the beginning, any algorithm needs at least $\log_2 n$ comparisons in the worst case.**
- such arguments are used to define the lower bound on the execution of a computation based on the rate at which information can be accumulated.

Information Flow and Optimal Time Bounds

- **Sorting:** We have $O(n \log_2 n)$ algorithms.
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
 - The n elements can be ordered in $n!$ possible ways, and only one of them is the correct order
 - Claim holds due to a similar reasoning as for Searching
 - Any sorting algorithm can be forced to use $\log_2 n!$ comparisons
 - Calculation shows that $\log_2 n!$ is $n \log_2 n$ subject to a constant factor

$$\log_2 n! = \sum_{k=1}^n \log_2 k \geq (n/2) \log_2(n/2).$$

Information Flow and Optimal Time Bounds

- **Selection Problem:** $O(n)$
- **Reasoning for Searching does not apply here**
 - $O(\log_2 n)$ would be a very poor lower bound
- **$O(n)$ is optimal**
 - **No order known before hand, ALL the n elements needs to be read**
 - Every change in the instance can change the result

Information Flow and Optimal Time Bounds

- **Faster Algorithms for special cases:**
- **Bucket Sort: $O(m + n)$**
 - n elements come from a fixed range of m different numbers.
- **$O(n)$ sorting in lexicographic order**
 - Words defined over a fixed alphabet
 - Total length of the given words: n
- **Do these two results contradict?**
 - **NO!**
 - **?**
 - **Because...**

Algorithms: Lecture 7

Chalmers University of Technology

Today's Lecture

Divide & Conquer

Counting Inversions

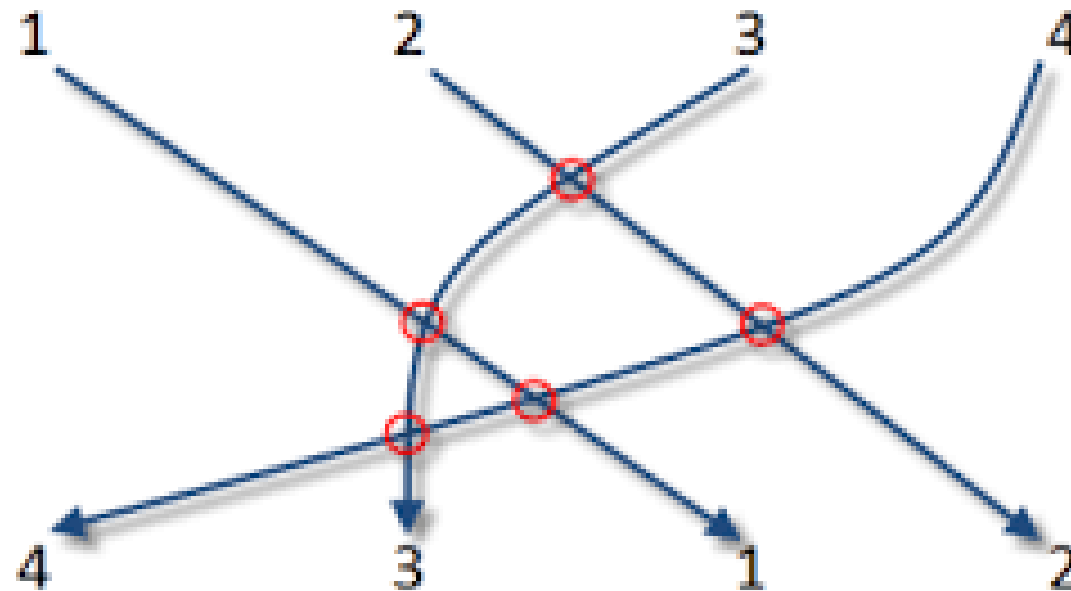
Closest Pair of Points

Multiplication of large integers

Intro to the forthcoming problems

Graphs: Basic Definitions, Applications and Interesting Problems

Counting Inversions



○ 5 inversions

Counting Inversions

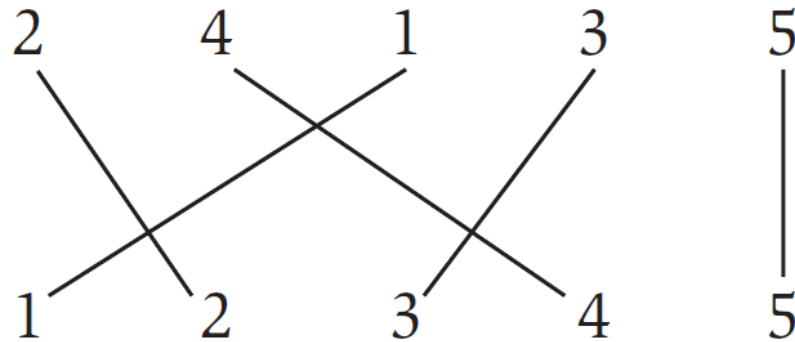
Given: a sequence (a_1, \dots, a_n) of elements where an order relation $<$ is defined.

Goal: Count the inversions in this sequence.

An inversion is a pair of elements where $i < j$ but $a_i > a_j$.

Find inversions in (2,4,1,3,5)

Assume there is a sequence $(1, 2, 3, \dots, n)$, thus the given sequence is a permutation of it.



What is it:

"degree of unsortedness" of a given sequence.

"Dissimilarity" of two sequences containing the same elements but in different order.

Counting Inversions

RANKING:

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with **similar** tastes.

Similarity metric: number of inversions between two rankings.

- **My rank:** $1, 2, \dots, n$.
- **Your rank:** a_1, a_2, \dots, a_n .
- Songs i and j **inverted** if $i < j$, but $a_i > a_j$.

	Songs				
	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

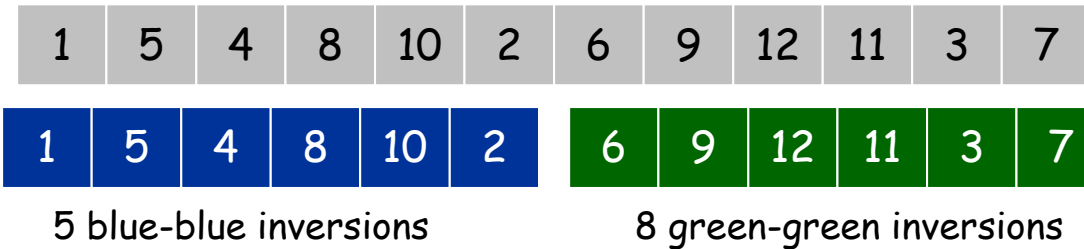
Inversions
3-2, 4-2

Brute force is Trivial: check all $O(n^2)$ pairs i and j .

Counting Inversions: Divide-and-Conquer

Due to the vague similarity with Sorting, Divide-and-conquer should be applicable

- split the sequence into two equal halves, **A** and **B**.
- **recursively count inversions in both A and B separately.**
- count inversions between **A** and **B**, and return sum of the **three** quantities.



9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Total = 5 + 8 + 9 = 22.

- $T(n) = 2T(n / 2) + ?$
- **Need to replace ? with $O(n)$ to make it better than the Brute Force.**
- **Sorting the sequence while counting inversions?**
 - **Can this help? OR Stupid idea as in Median Finding?**
- **Sorting AND Counting Inversions Simultaneously... Does it sound good?**
 - **Challenge:** merge two sorted sequences A and B, and simultaneously count the inversions between A and B, **still everything in $O(n)$ time**

Counting Inversions: Merge and Count

Counting A-B inversions

- Assume both **A** and **B** are sorted.
- Proceed as in the Mergesort
 - **While merging** whenever the next element copied into the merged sequence is from **B**, this element has inversions with exactly those **elements of A not visited yet**.



13 blue-green inversions: $6 + 3 + 2 + 2 + 0 + 0$

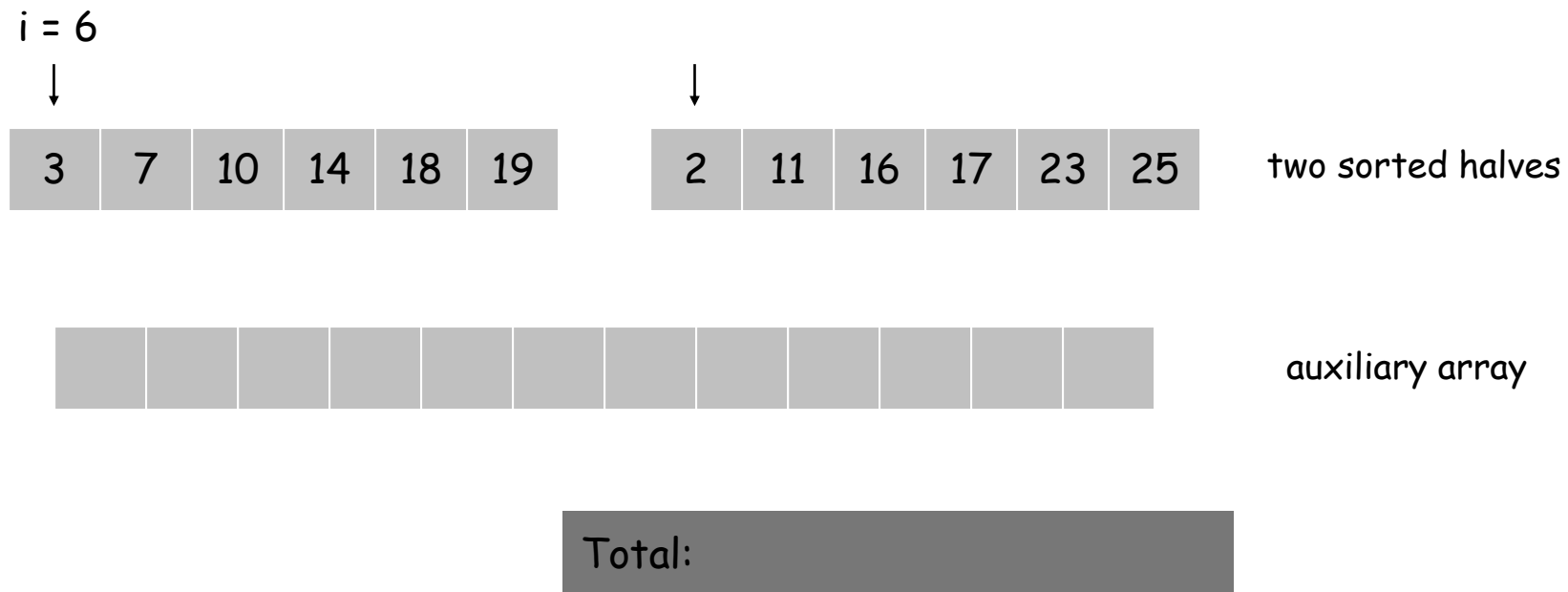


$$T(n) = 2T(n / 2) + O(n) \text{ implies } O(n \log n)$$

Merge and Count

Merge and count step.

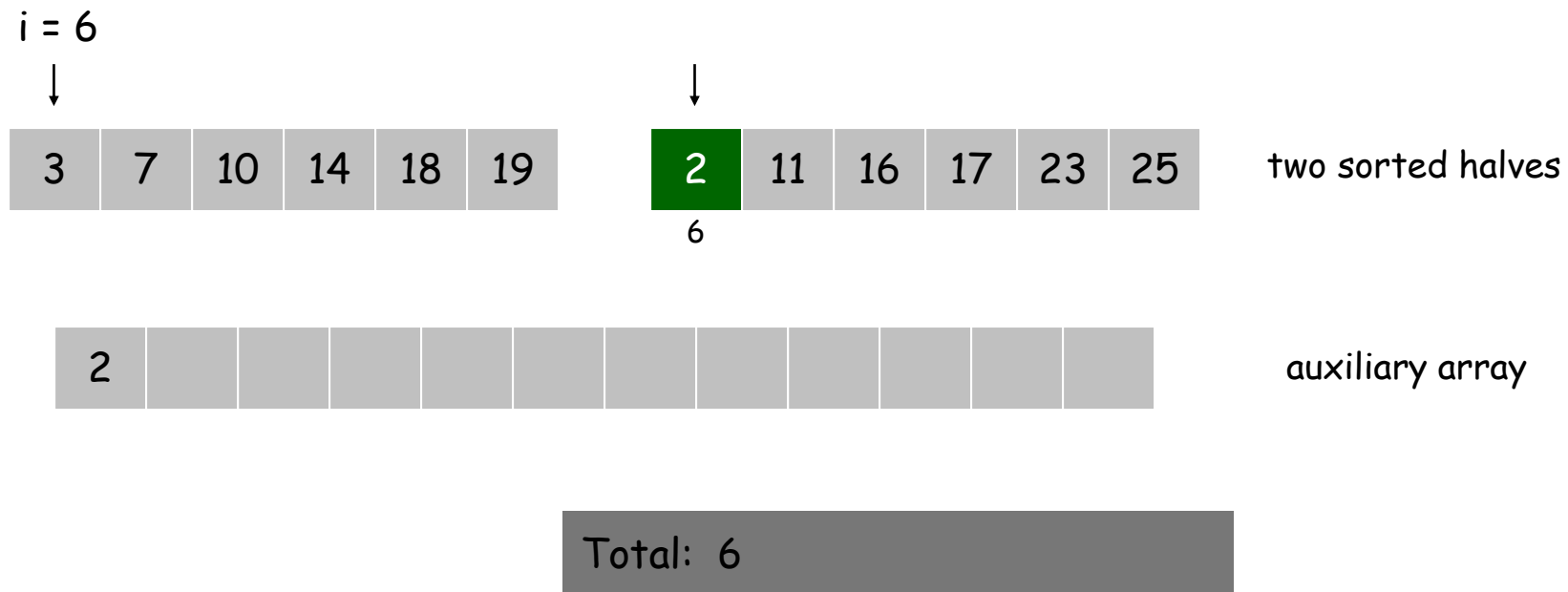
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

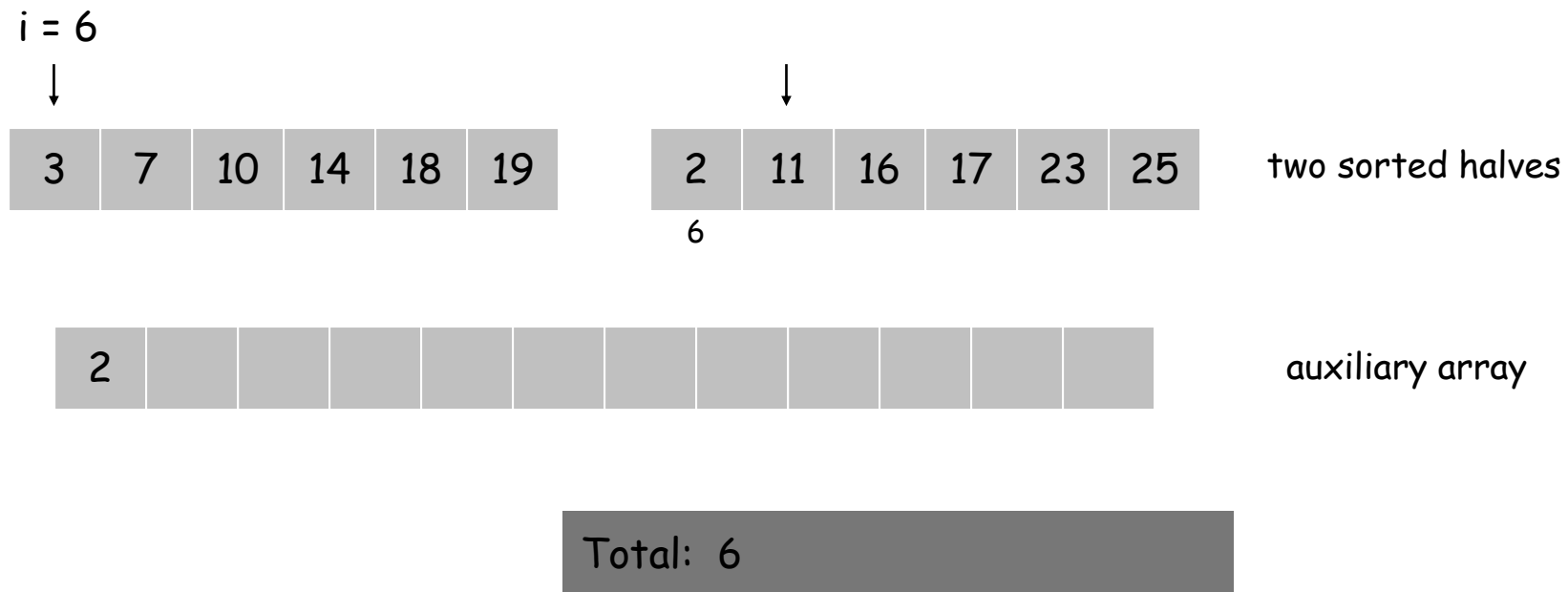
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

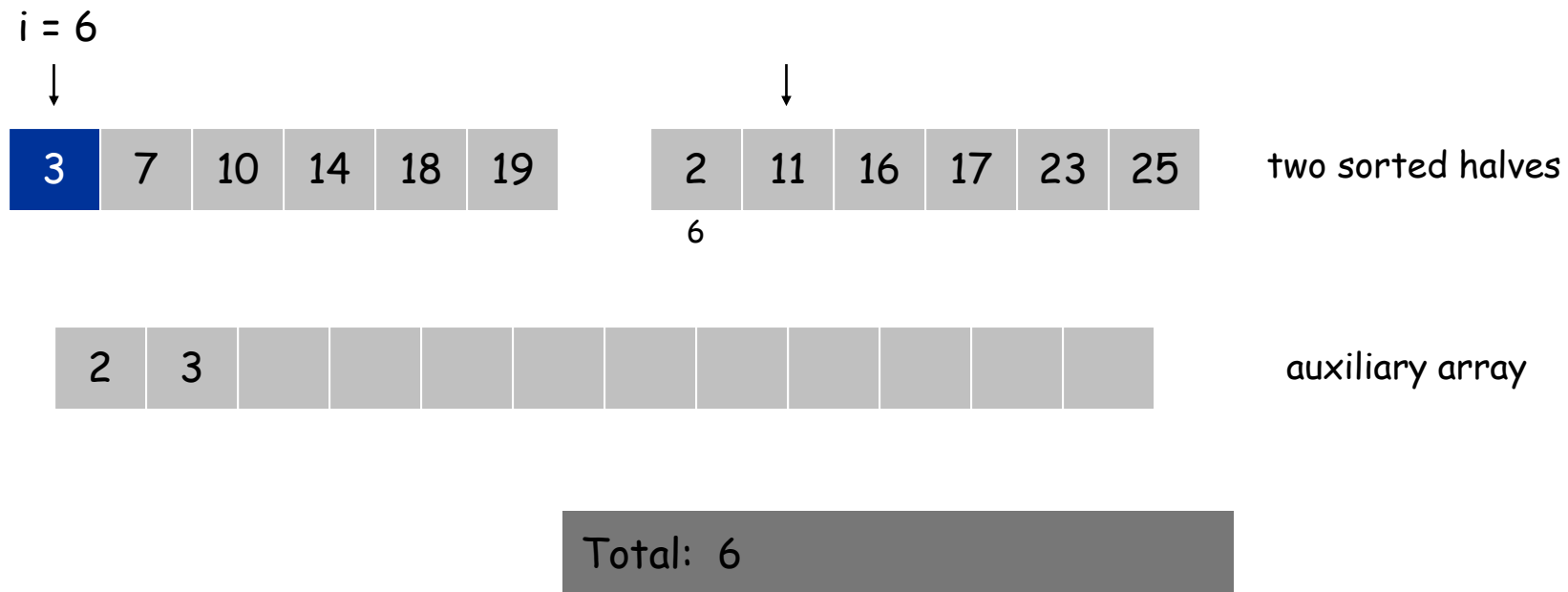
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

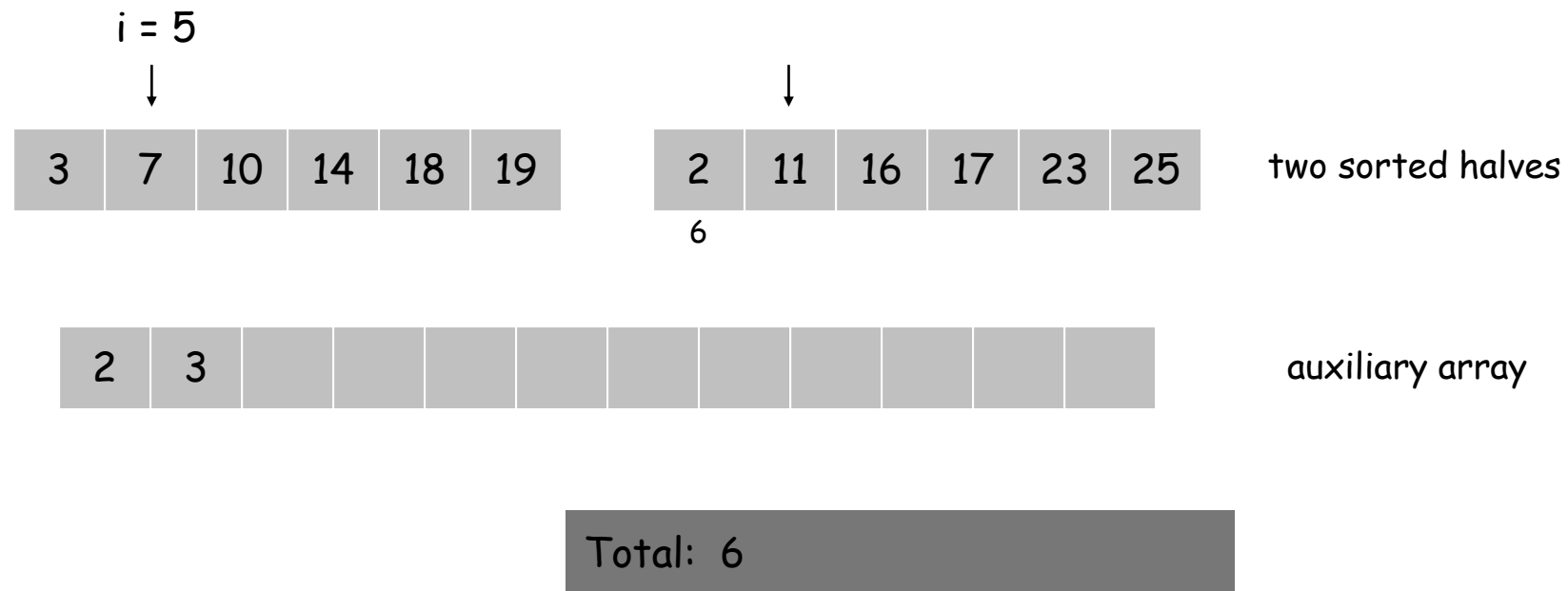
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

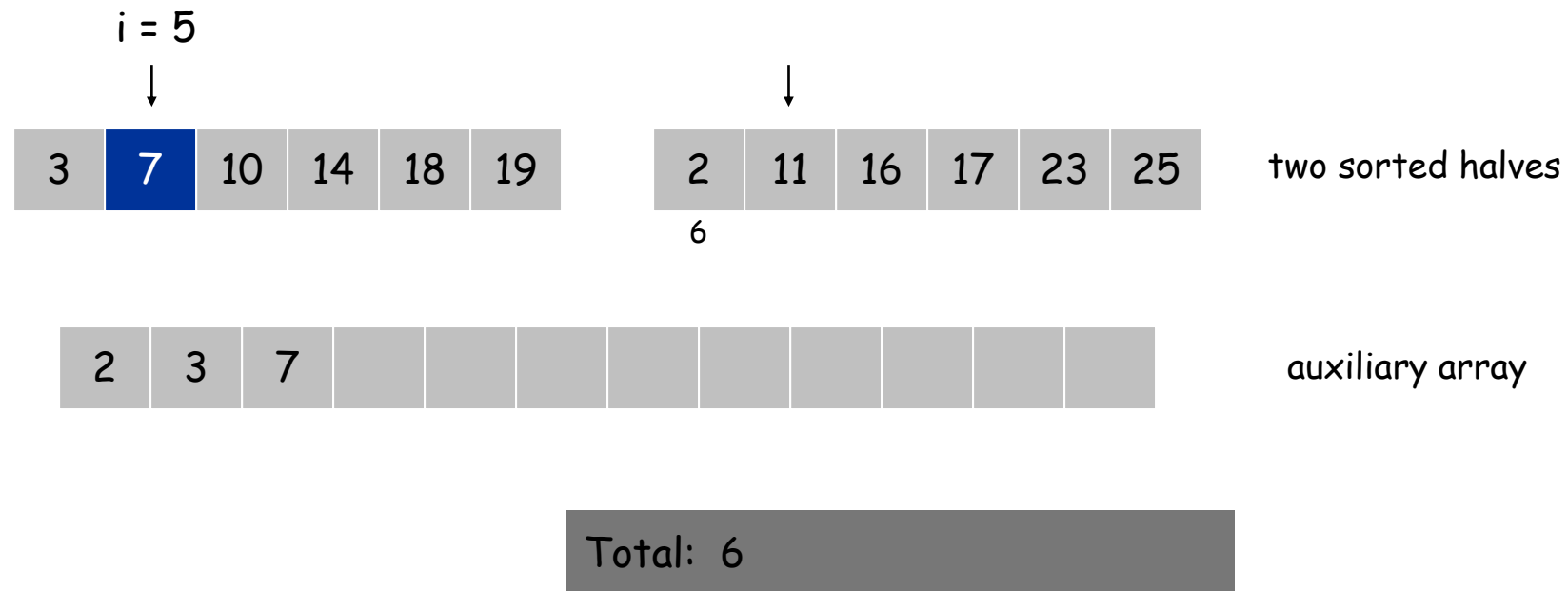
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

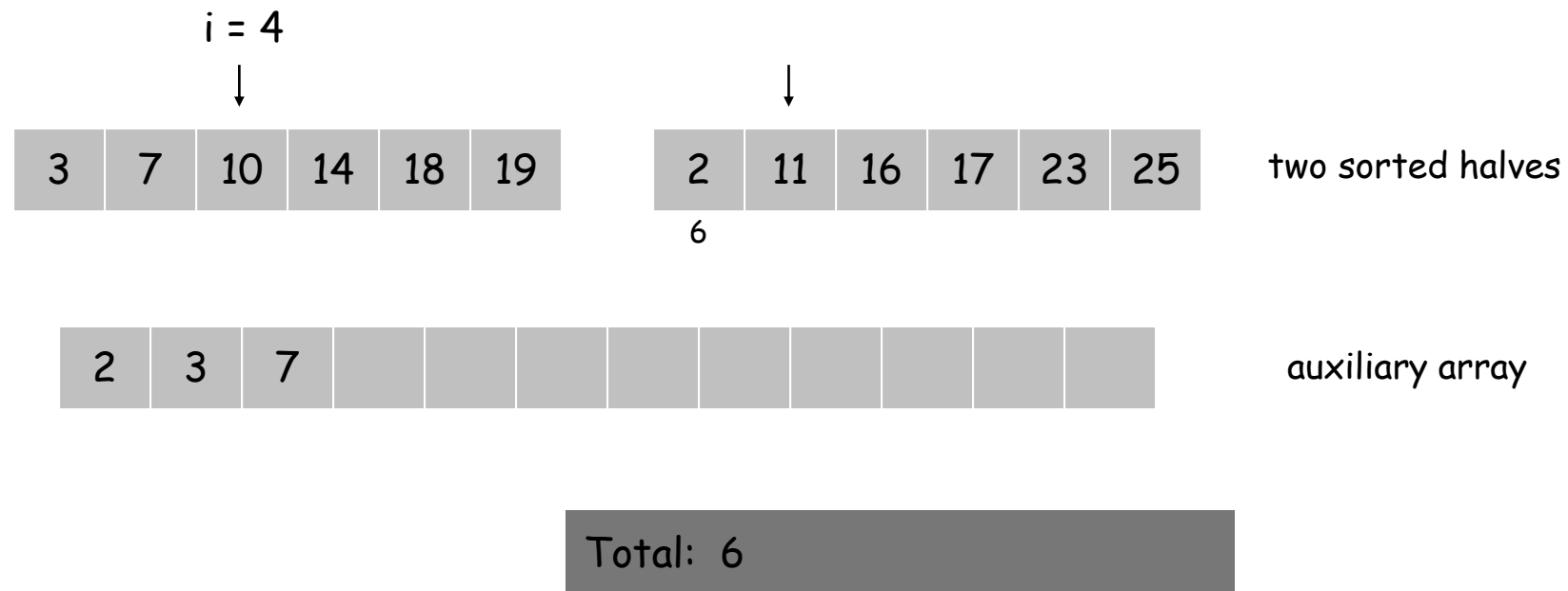
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

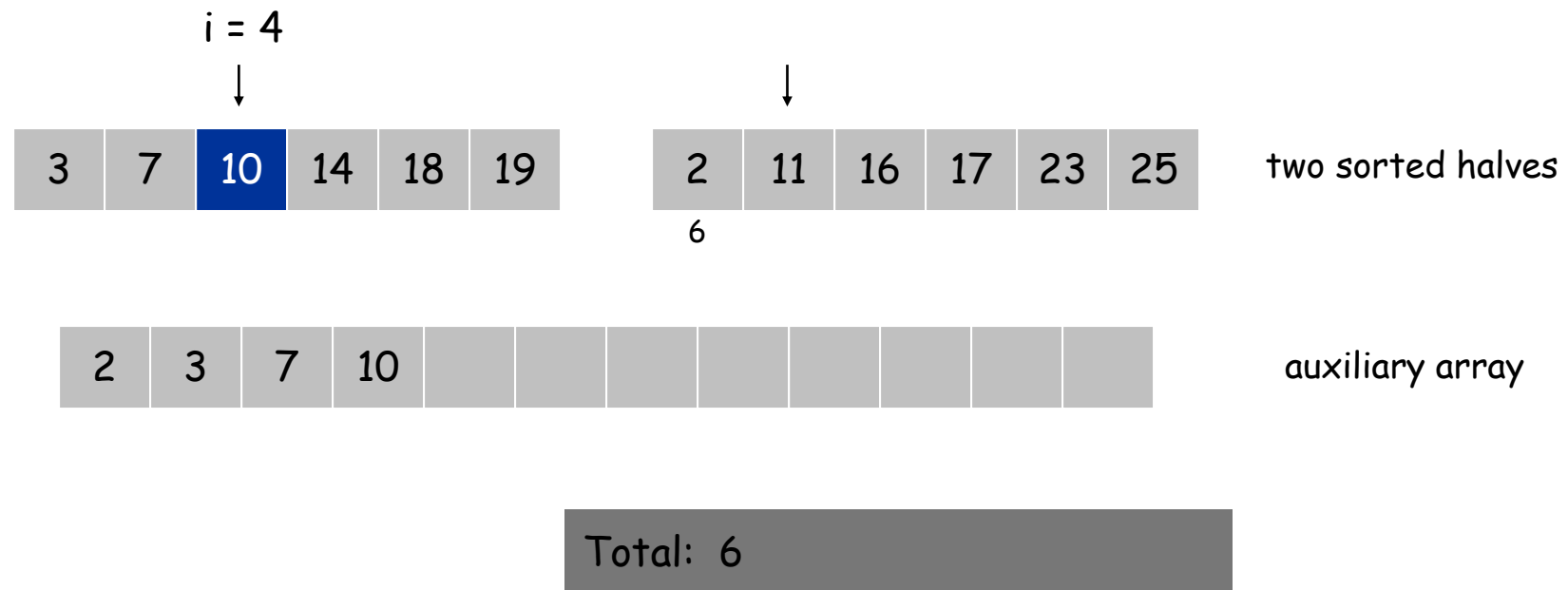
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

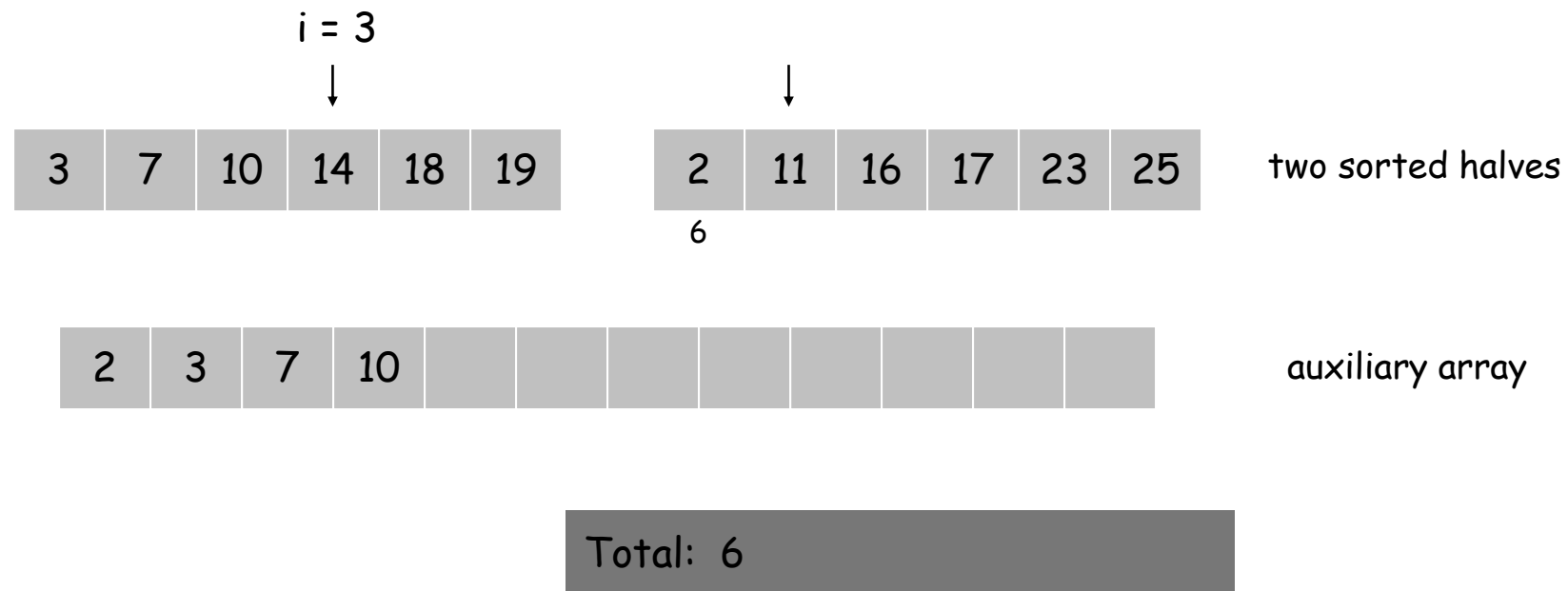
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

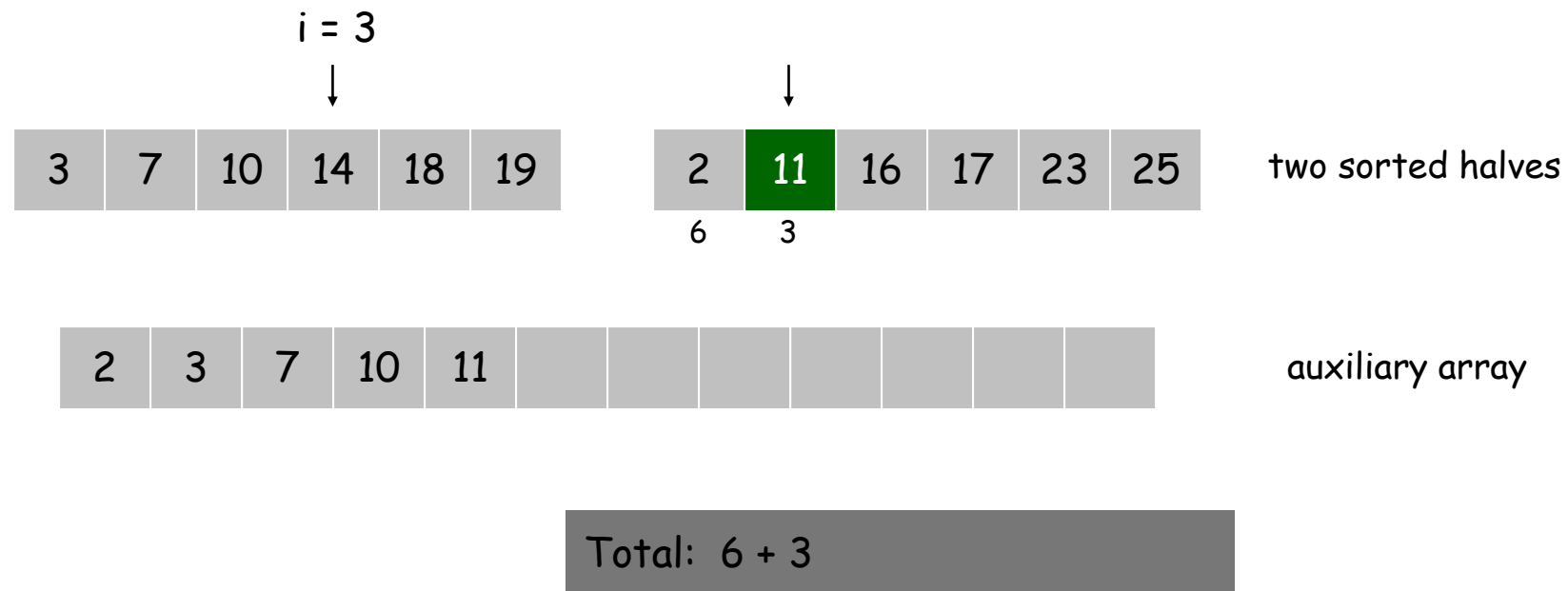
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

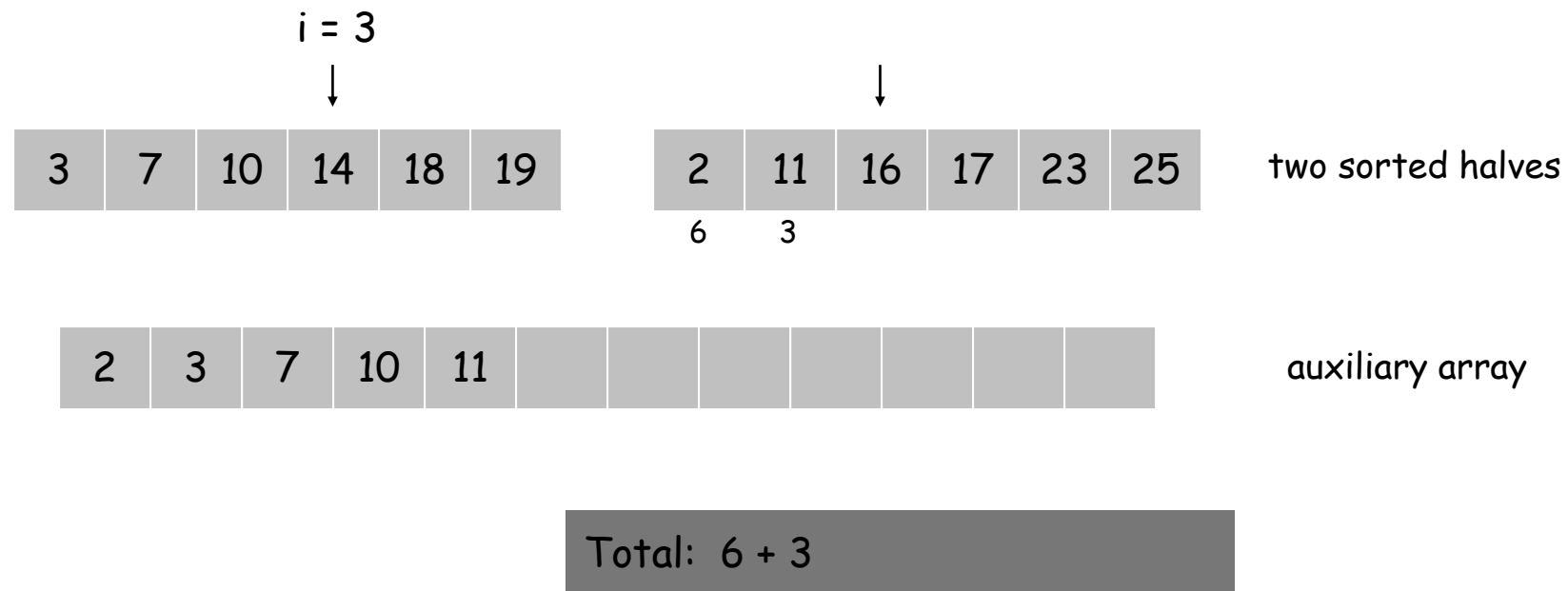
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

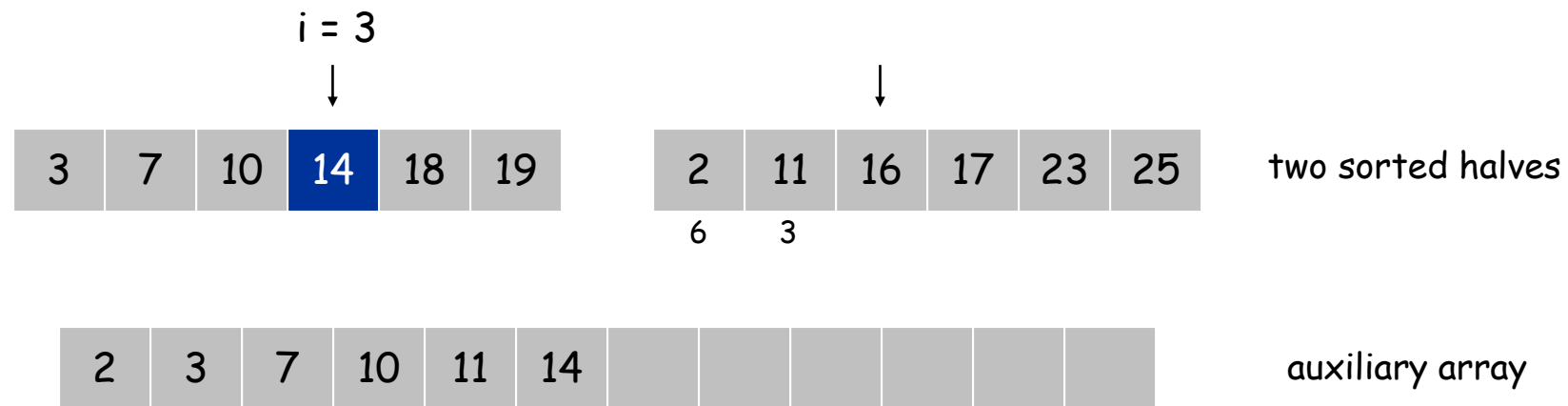
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

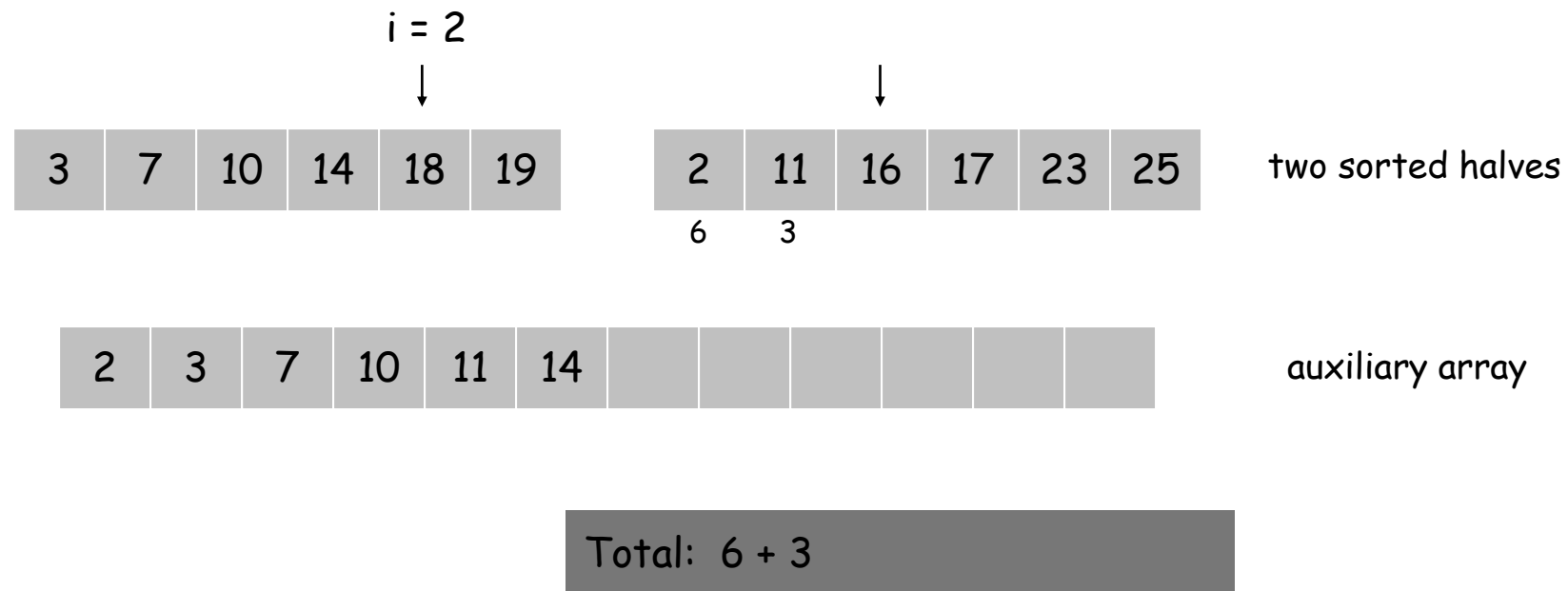


Total: 6 + 3

Merge and Count

Merge and count step.

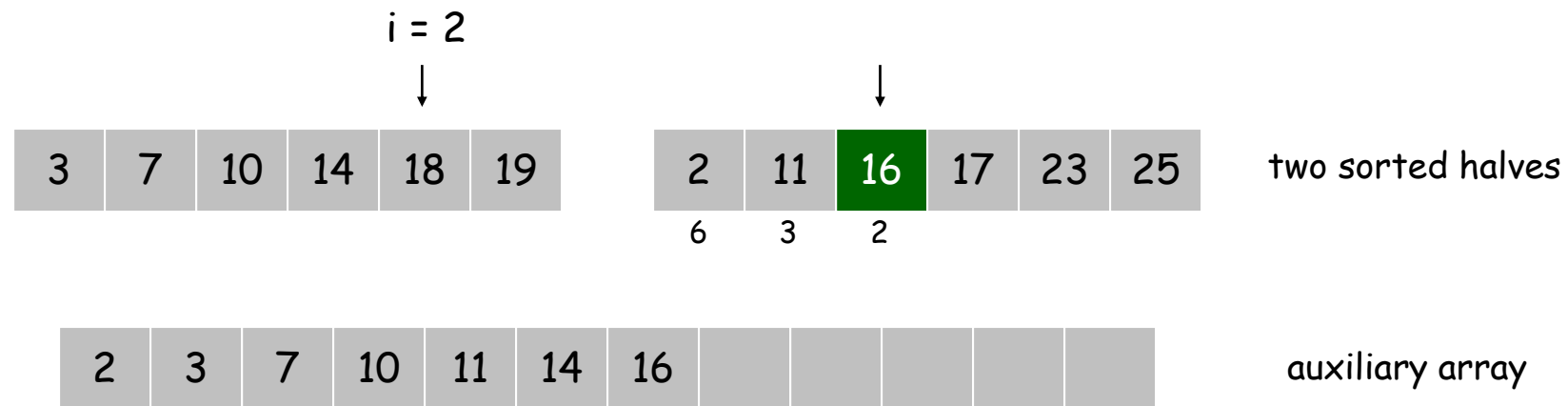
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

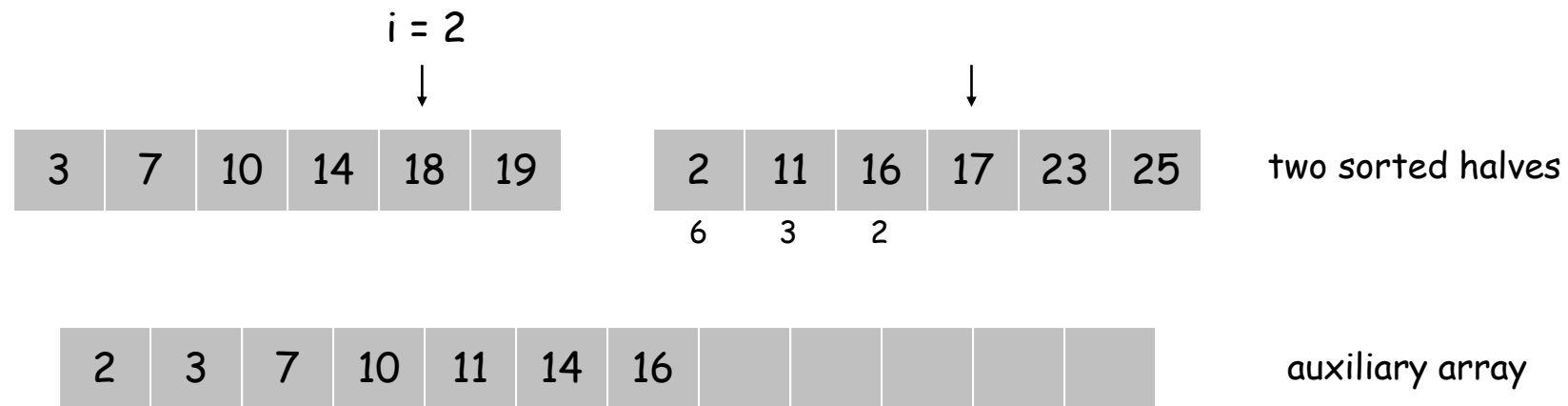


Total: $6 + 3 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

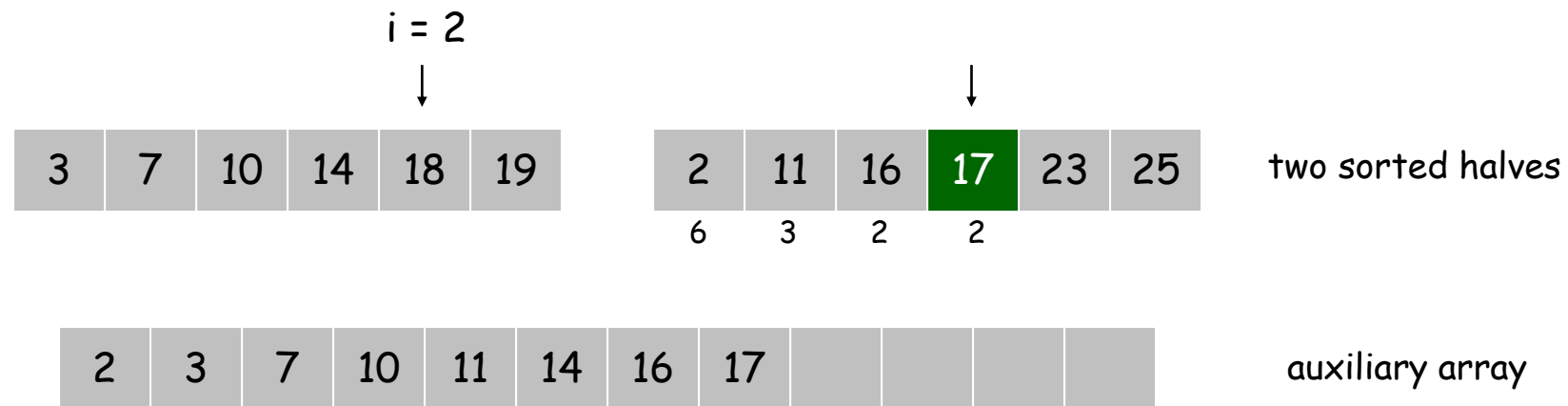


Total: $6 + 3 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

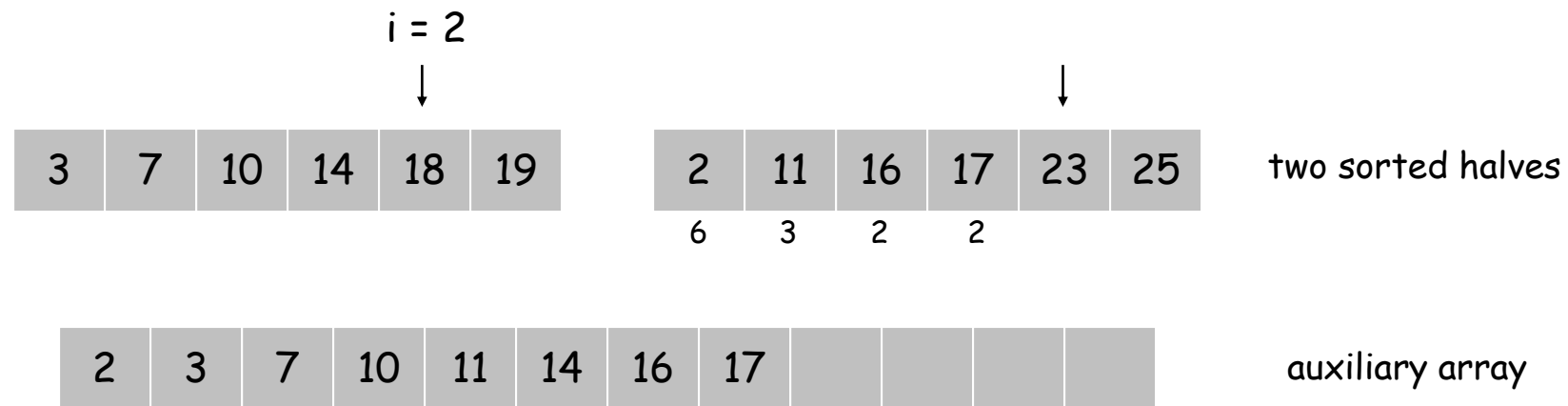


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

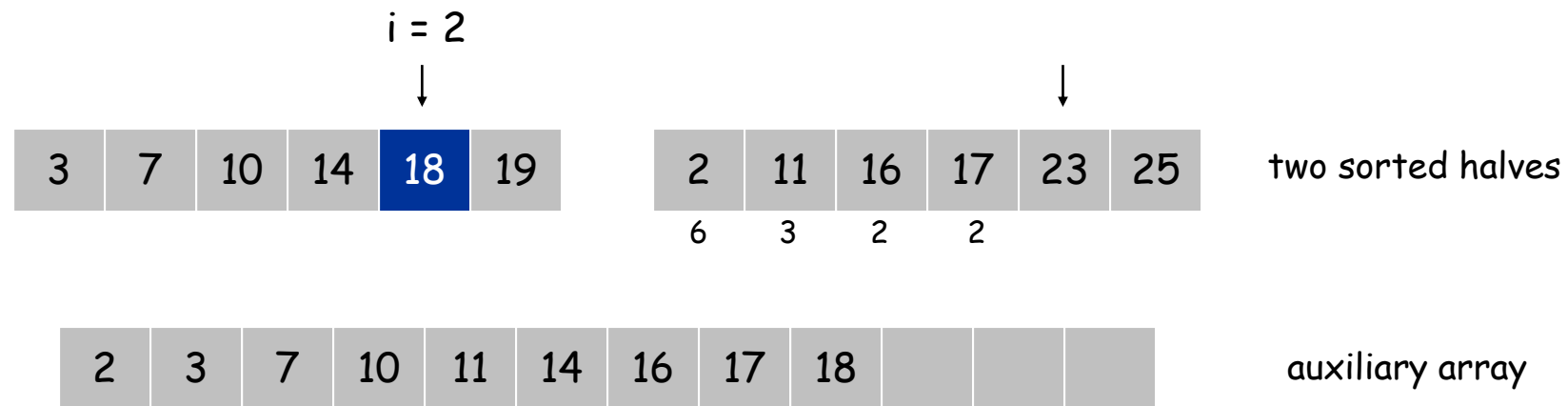


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

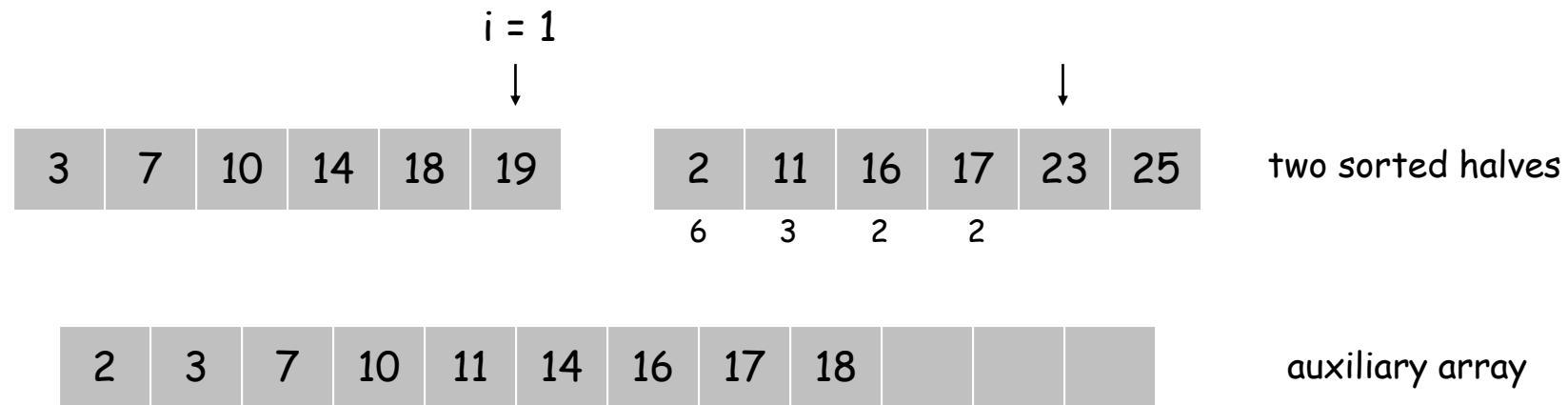


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

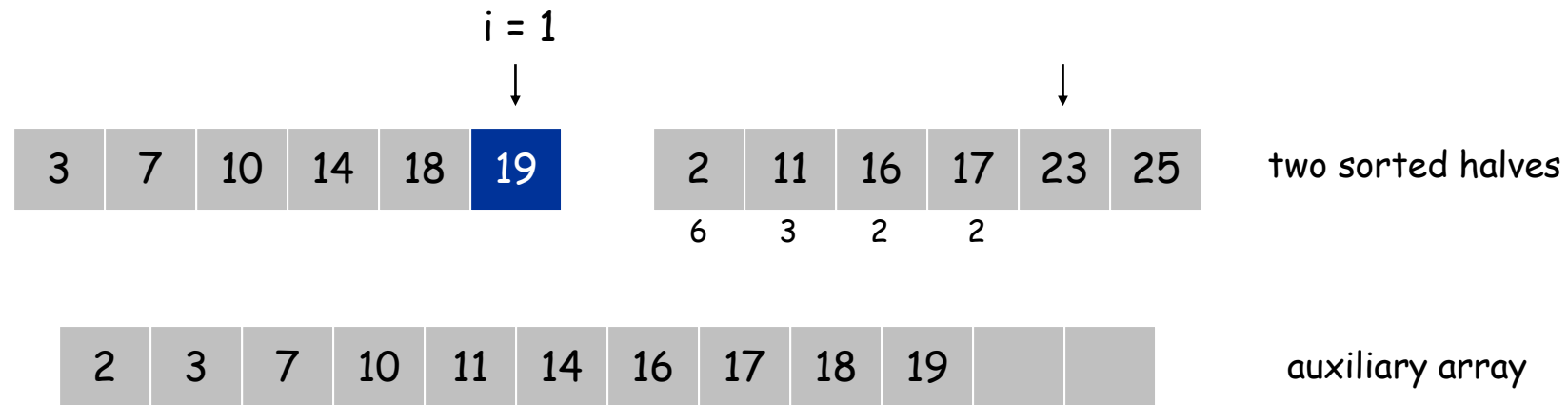


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

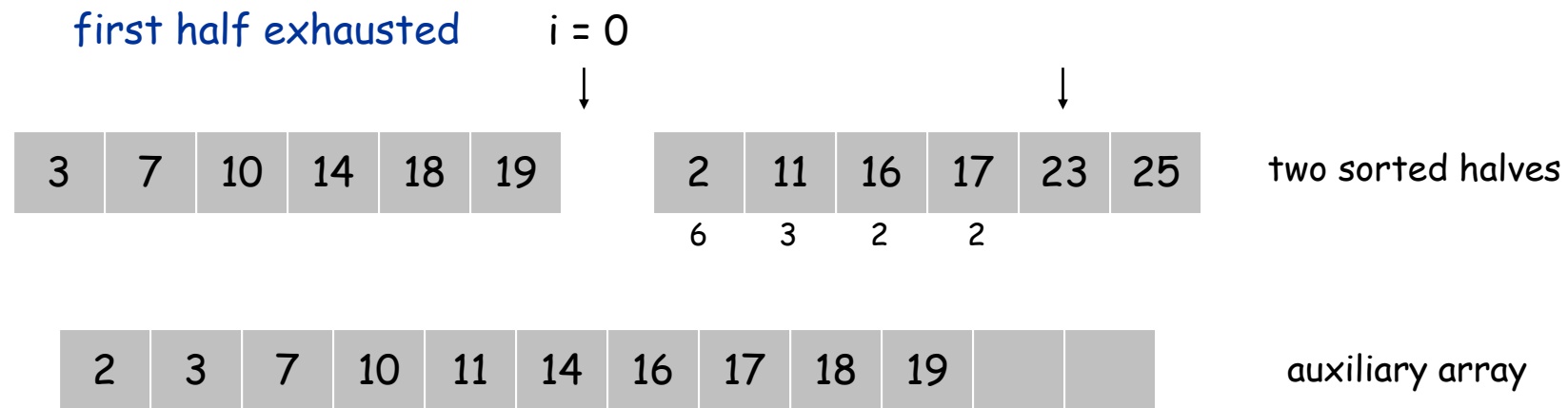


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

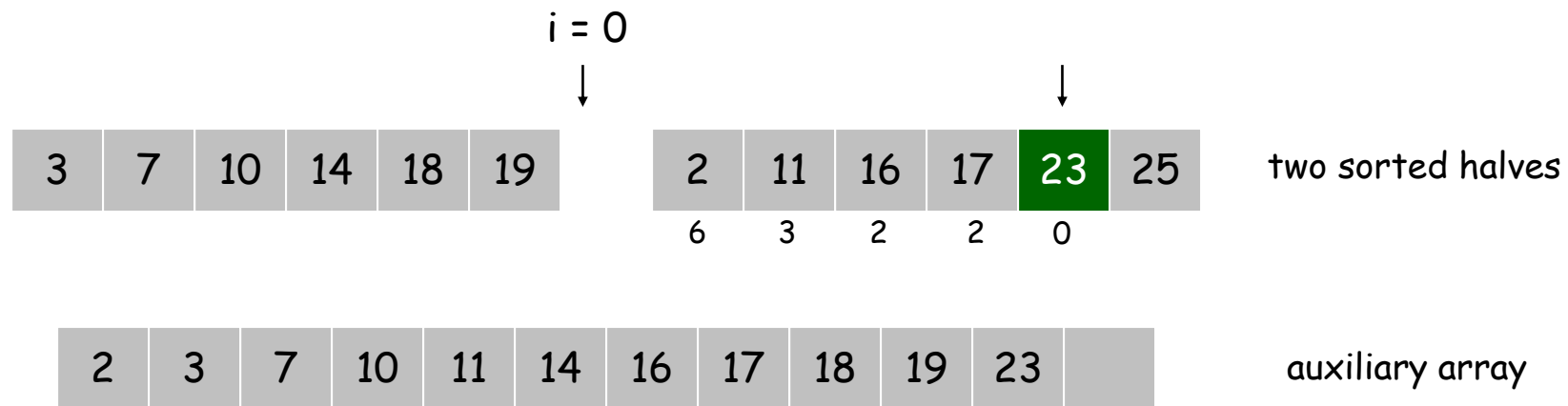


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

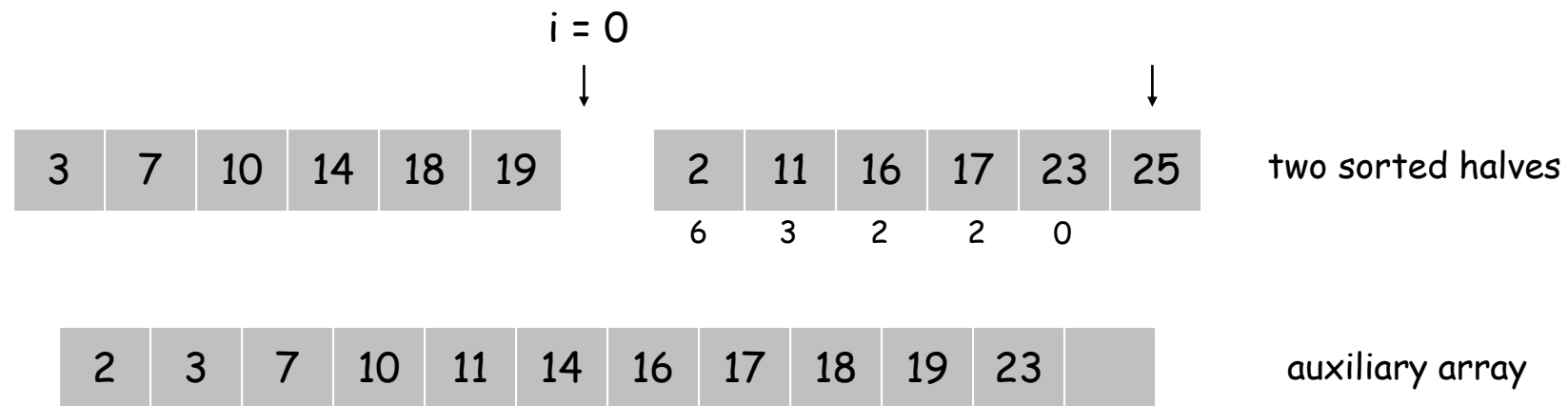


Total: $6 + 3 + 2 + 2 + 0$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

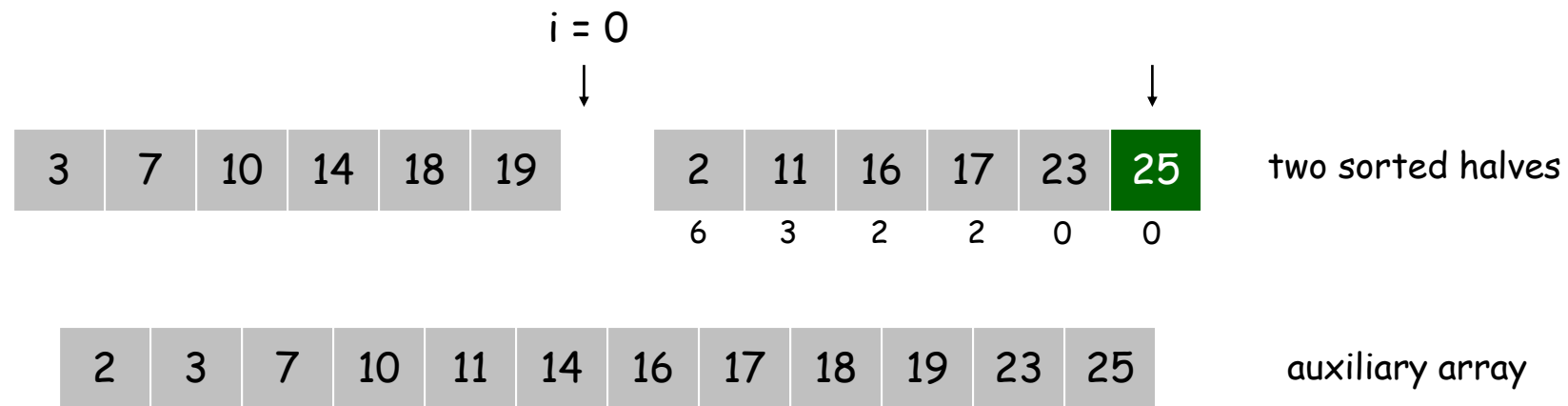


Total: $6 + 3 + 2 + 2 + 0$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

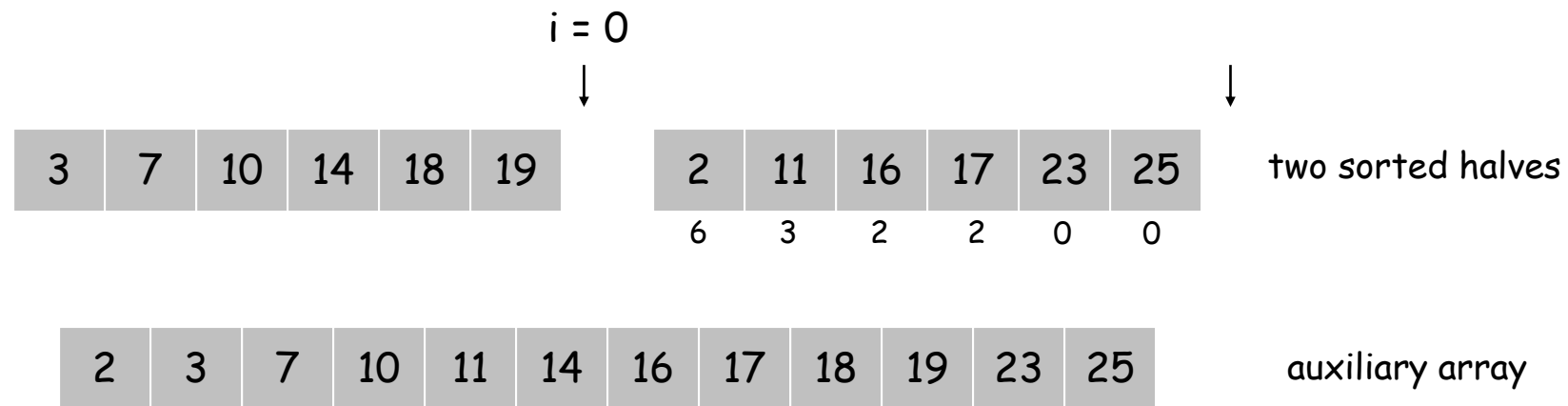


Total: $6 + 3 + 2 + 2 + 0 + 0$

Merge and Count

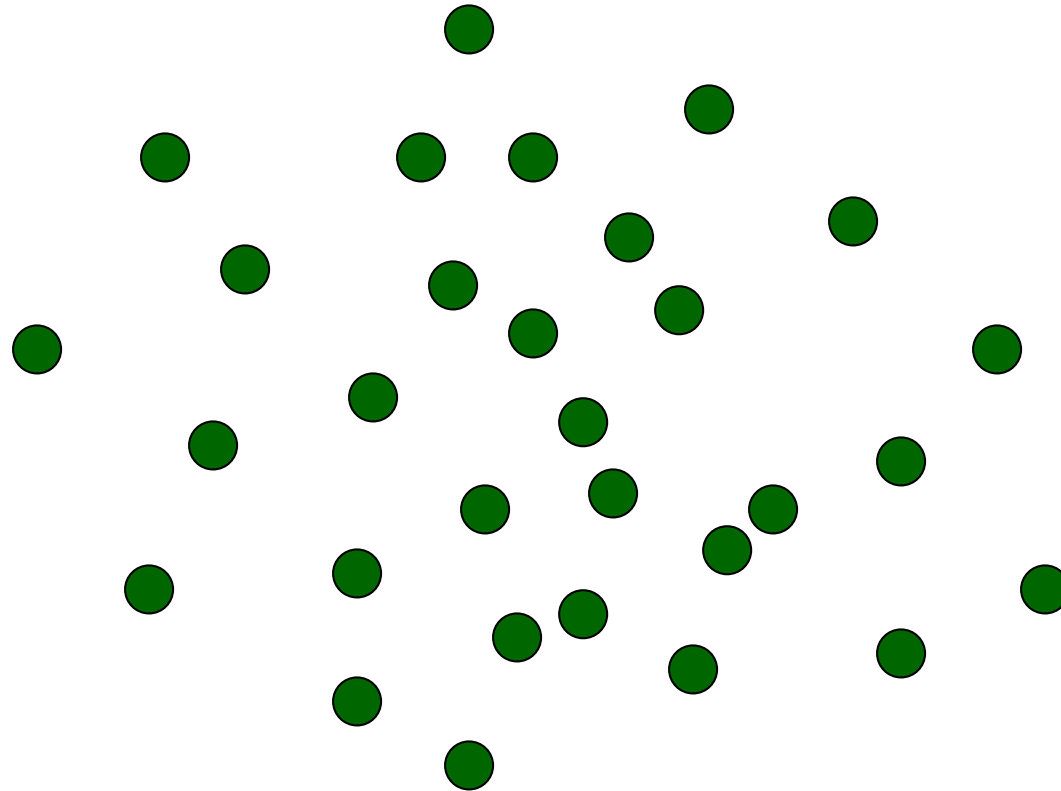
Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Total: $6 + 3 + 2 + 2 + 0 + 0 = 13$

Closest Pair of Points



Closest Pair of Points

Given: Set of n points given as Cartesian coordinates in the plane

Goal: Find a pair of points with minimum distance.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.

Brute force: Check all pairs of points p and q with $O(n^2)$ comparisons.

Our aim: Using Divide and Conquer, $T(n) = 2T(n / 2) + O(n)$ to get to $O(n \log n)$

Closest Pair of Points

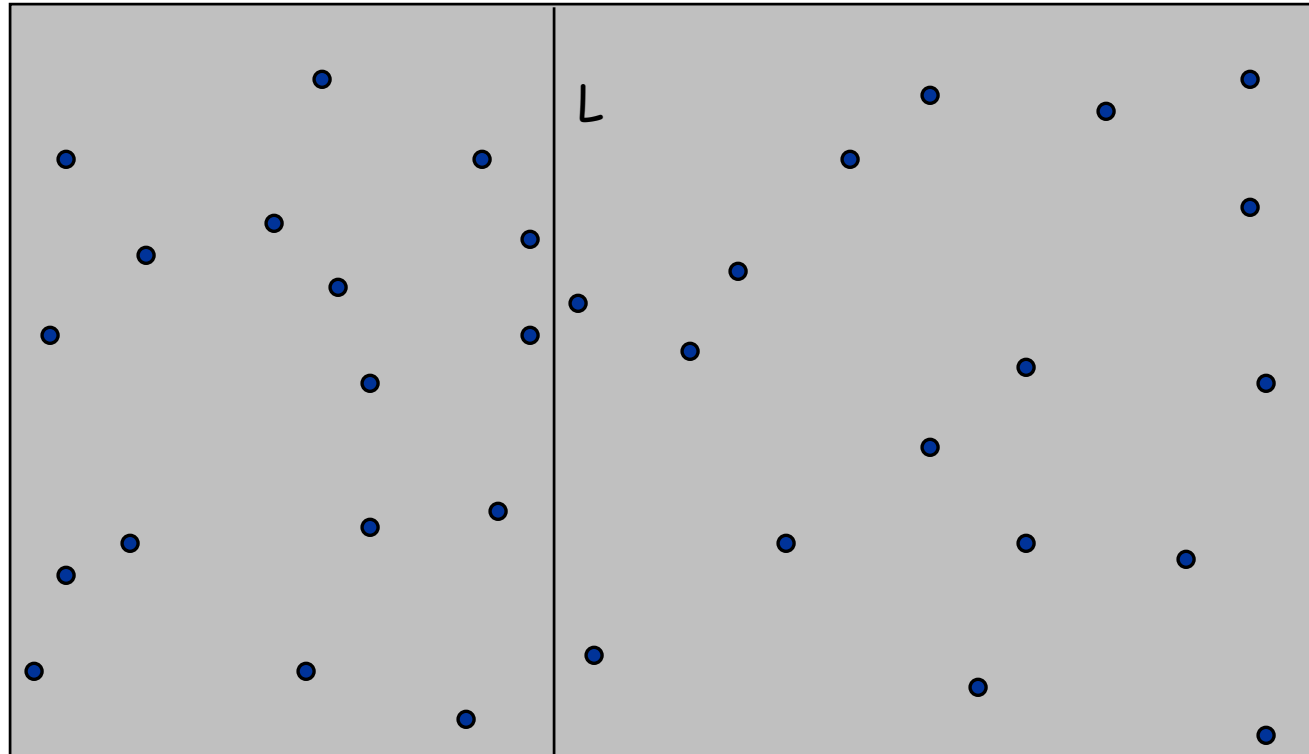
How it works:

Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x -coordinate ← Why sorting? find median directly in $O(n)$

Take the median z of all x -values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.



Closest Pair of Points

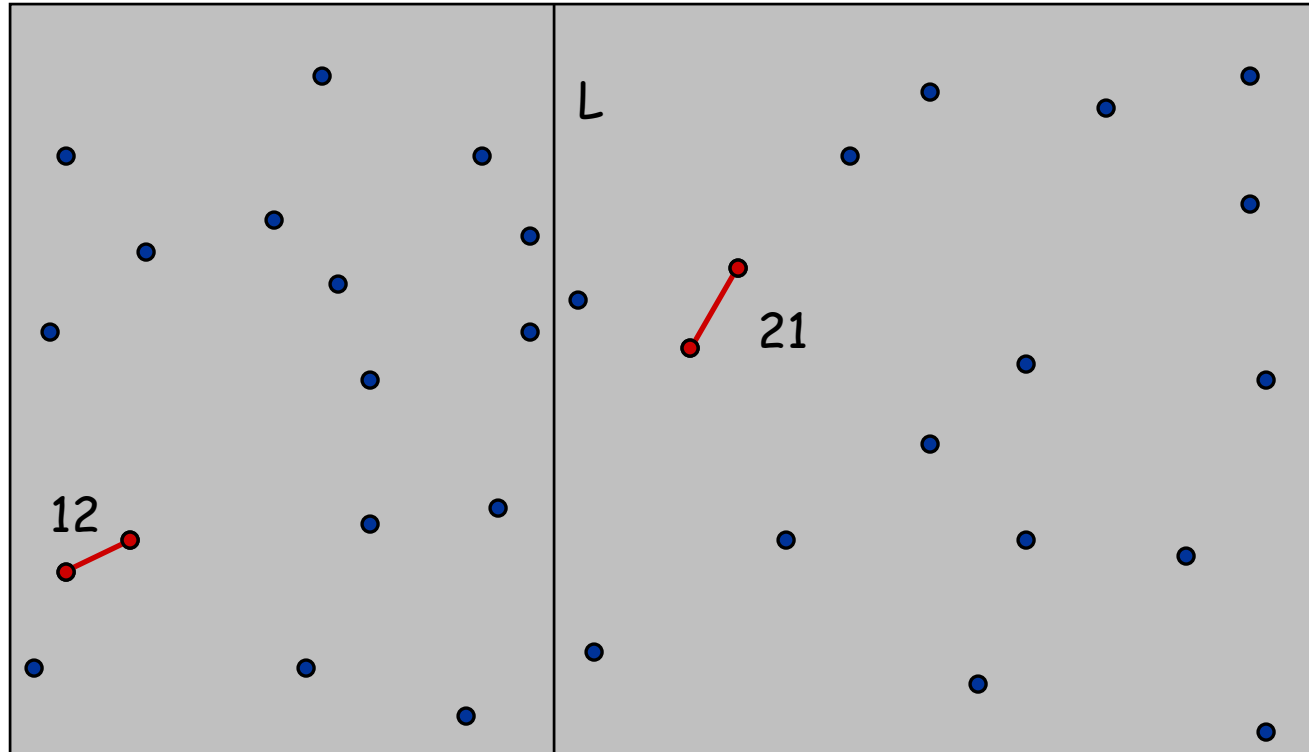
How it works:

Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x -coordinate ← Why sorting? find median directly in $O(n)$

Take the median z of all x -values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.



Closest Pair of Points

How it works:

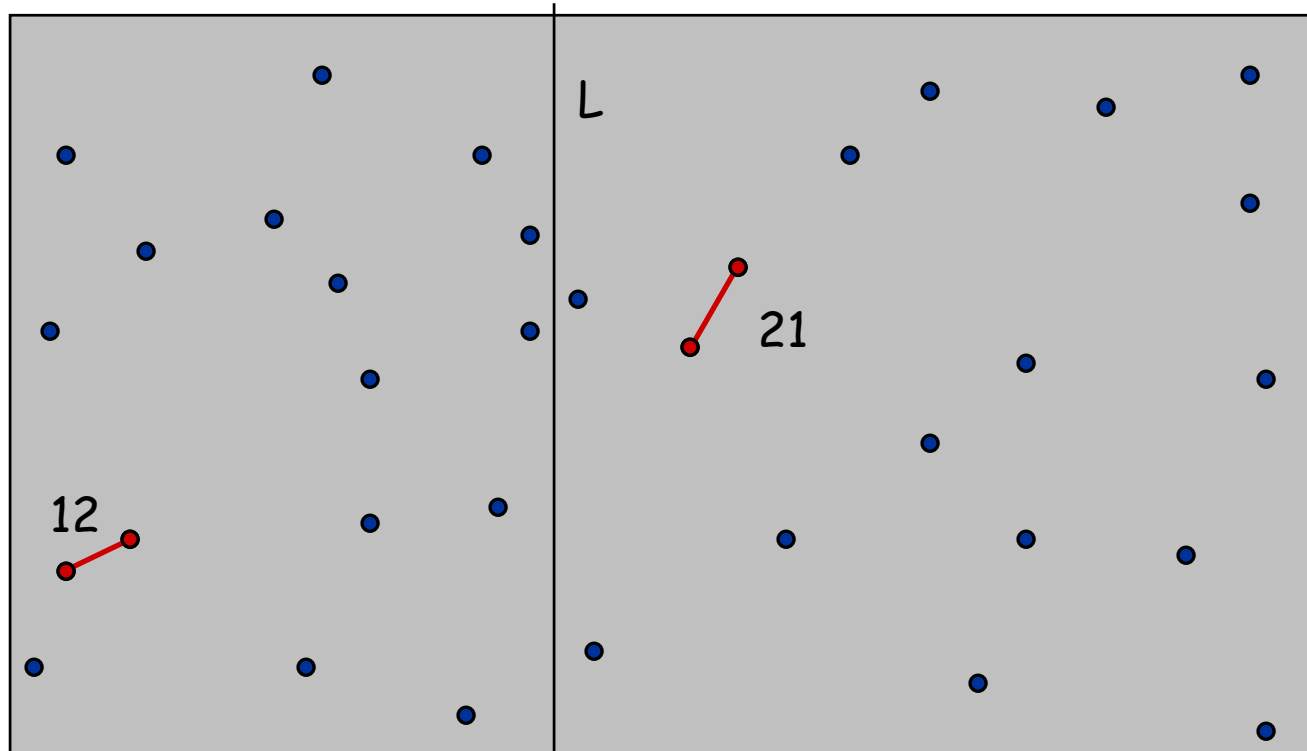
Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x -coordinate ← Why sorting? find median directly in $O(n)$

Take the median z of all x -values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.

Compute closest pairs where one point is in S_1 while the other in S_2 ???

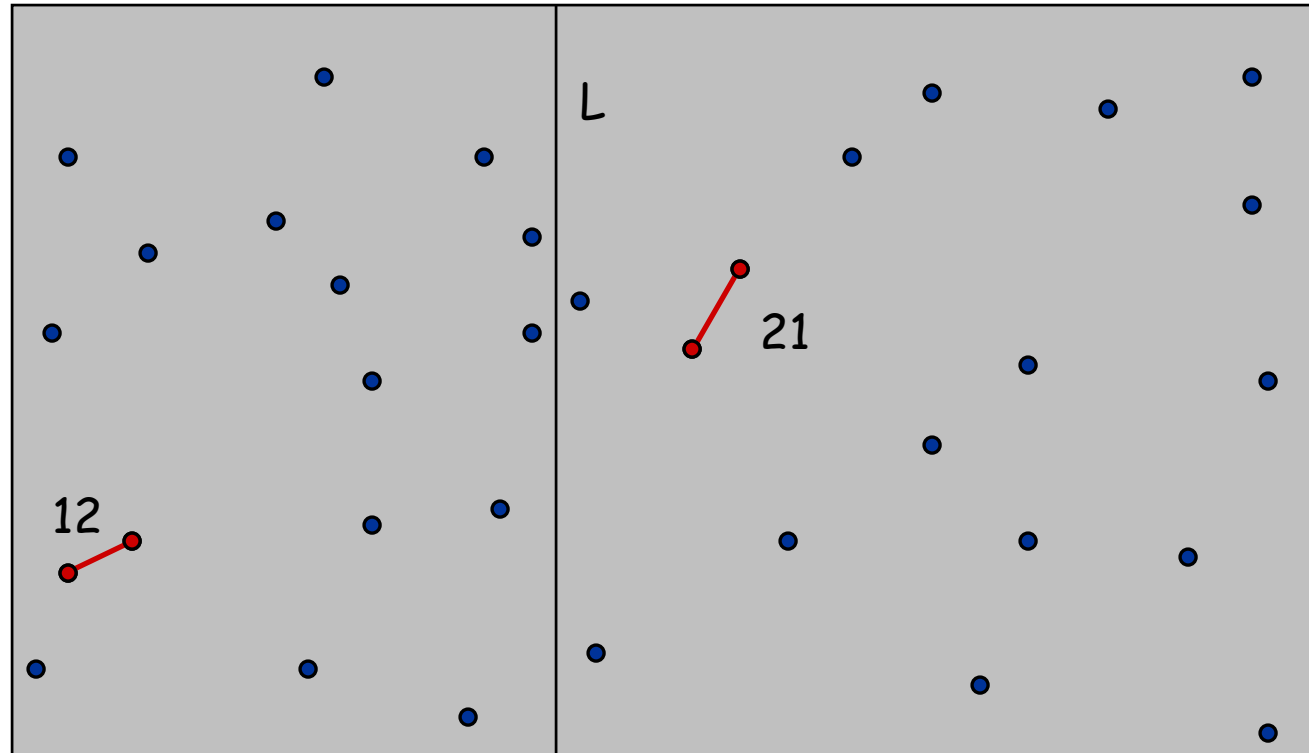


Closest Pair of Points

Let d_1 be the distance between the closest pair in S_1

Similarly d_2 for the set S_2

$d := \min(d_1, d_2)$



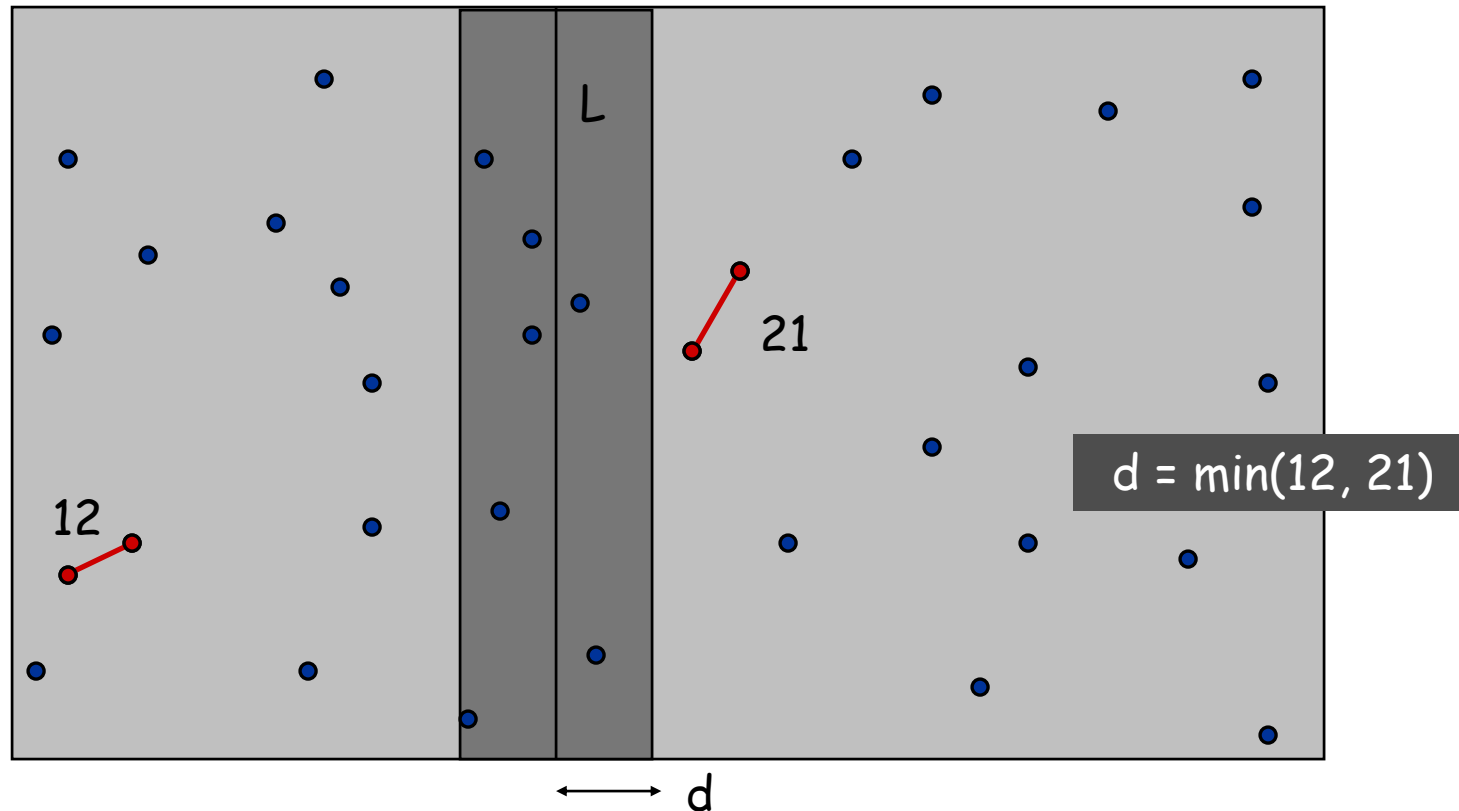
Closest Pair of Points

Find closest pair with one point in S_1 while the other in S_2

Because we already have closest pair with distance d

The candidates for such pairs of points are in a stripe of breadth d on both sides of the separating line L

Consider the pairs with one point in each set, assuming that distance $< d$.



Closest Pair of Points

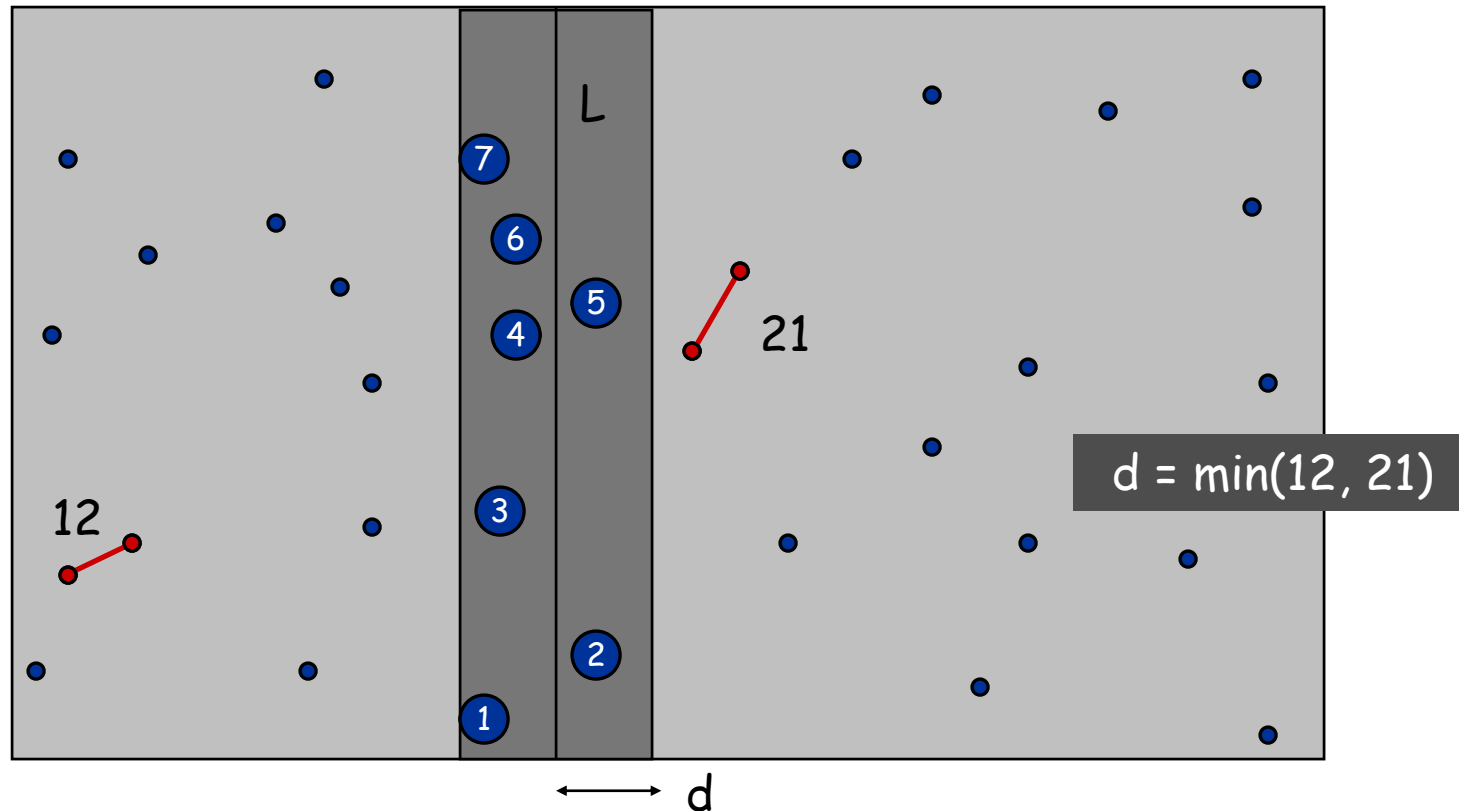
Find closest pair with one point in S_1 while the other in S_2

Sort points in 2d-strip by their y coordinate.

Only check distances of those within 11 positions in sorted list! (d is 12)

Managed everything in $O(n)$ for the conquer step.

Finally, standard recurrence $T(n) = 2T(n/2) + O(n)$ with solution $O(n \log n)$



Multiplication of large integers

2095067093034680994318596846868779409766717133476767930
X 5920175091777634709677679342929097012308956679993010921

9715480283945084383094856701043643845790217965702956767
+ 1242431098234099057329075097179898430928779579277597977

Multiplication of large integers

a , b are both n -digit integers

If we use the brute-force approach to compute $c = a * b$, what is the time efficiency?

Remember the "School Algorithm"

addition of n integers, each with $O(n)$ digits requires $O(n^2)$

Already discussed in Lecture 1.

Divide and Conquer: *Can we do better?*

Split the decimal representations of the factors a and b into two halves,
and to multiply, use the distributive law.

$$3 \times 6$$

$$= 3 \times (2 + 4)$$

$$= 3 \times 2 + 3 \times 4$$

$$\begin{array}{r} 5678 \cdot 4321 \\ \hline 22712000 \\ 01703400 \\ 00113560 \\ 00005678 \\ \hline 24534638 \end{array}$$

Multiplication of large integers

$$a = a_1a_0 \text{ and } b = b_1b_0$$

$$c = a * b$$

$$= (a_110^{n/2} + a_0) * (b_110^{n/2} + b_0)$$

$$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$

For instance: $a = 123456$, $b = 117933$:

$$\text{Then } c = a * b = (123*10^3+456)*(117*10^3+933)$$

$$=(123 * 117)10^6 + (123 * 933 + 456 * 117)10^3 + (456 * 933)$$

What we achieve:

multiplication of n -digit numbers is reduced to:

several multiplications of $n/2$ -digit numbers and some additions

In fact: we need 4 multiplications involving a_1, a_0, b_1 , and b_0

Multiplications with 10^n and $10^{n/2}$ are trivial: Append the required number of 0s.

Multiplication of large integers

$$a = a_1a_0 \text{ and } b = b_1b_0$$

$$c = a * b$$

$$= (a_110^{n/2} + a_0) * (b_110^{n/2} + b_0)$$

$$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$

Solve by recursive application of the above:

At every recursion step: problem reduces to 4 recursive calls AND addition

Clearly: $T(n) = 4T(n / 2) + O(n)$,

$a = 4, b = 2, k = 1$. Master Theorem implies: $T(n) = O(n^{\log_2 4}) = O(n^2)$

Still same like Brute Force... we need to avoid 4 recursive calls at each level

Multiplication of large integers

$$a = a_1a_0 \text{ and } b = b_1b_0$$

$$c = a * b$$

$$= (a_110^{n/2} + a_0) * (b_110^{n/2} + b_0)$$

$$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$

Can we achieve?

multiplication of n -digit numbers by **three** multiplications of $n/2$ -digit numbers + $O(n)$

<presenting the idea using the rectangle example>

Multiplication of large integers

$$a = a_1a_0 \text{ and } b = b_1b_0$$

$$c = a * b$$

We are aiming at:

$c = c_210^n + c_110^{n/2} + c_0$, where computing each of c_2 , c_1 and c_0 requires ONE multiplication

This is achievable as follows:

$$\begin{aligned} c &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_210^n + c_110^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0 .

Multiplication of large integers

Finally

$$c = c_2 10^n + c_1 10^{n/2} + c_0,$$

where

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

Clearly: $T(n) = 3T(n/2) + O(n)$,

$a = 3, b = 2, k = 1$. Master Theorem implies: $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$ far better than $O(n^2)$

Caution: Factors $(a_1 + a_0)$ and $(b_1 + b_0)$ may have $(n/2) + 1$ digits

Split off the first digit and have recursive calls of $(n/2)$

Can only cause $O(n)$ extra work, won't affect the time bound

Acceleration takes effect only for rather large n (more than some 100 digits)

Recursive calls being the overhead

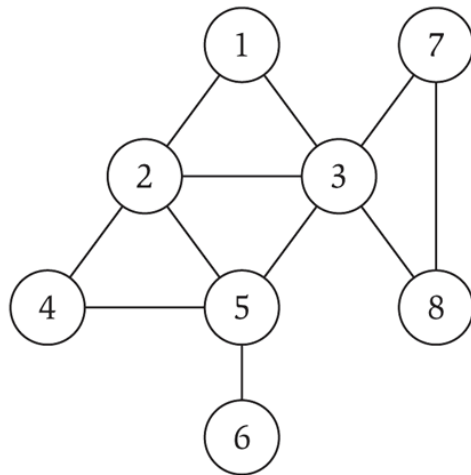
Graphs

Basic Definitions, Applications and Interesting Problems

Graphs

Graph. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures binary relationships between objects (nodes).
 - **Symmetric relationships**: G is undirected, all edges (u, v) are unordered: (u, v) and (v, u) are identical
 - **Asymmetric relationships**: G is directed, All edges (u, v) are ordered: (u, v) and (v, u) are different
- Graph size parameters: $n = |V|$, $m = |E|$.
- A node and an edge are **incident** if the edge contains this node.
- Two vertices(nodes) joined by an edge are called **adjacent**.



$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 \}$

$n = 8$

$m = 11$

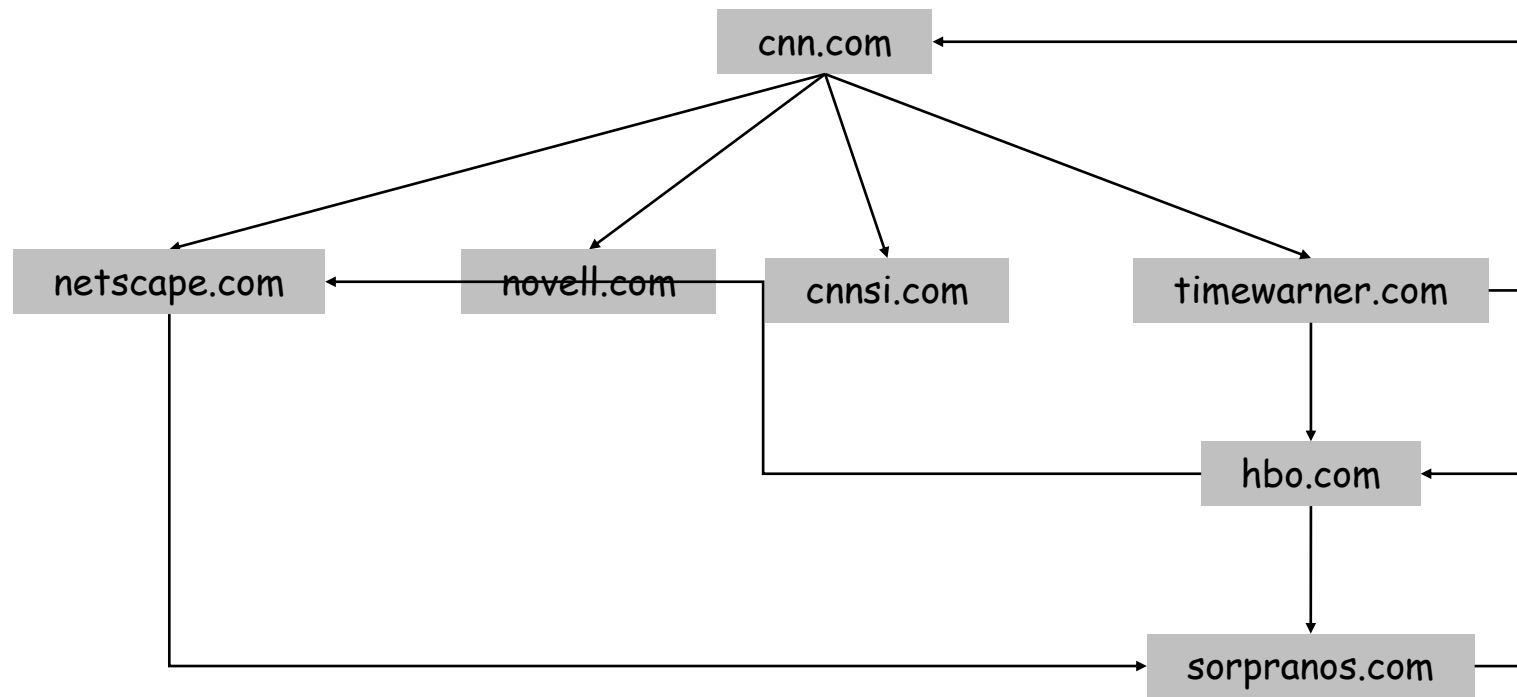
Some Graph Applications

<i>Graph</i>	<i>Nodes</i>	<i>Edges</i>
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

Web graph.

- Node: web page.
- Edge: hyperlink from one page to another.



Social network

Social network graph.

- Node: people.
- Edge: relationship between two people.

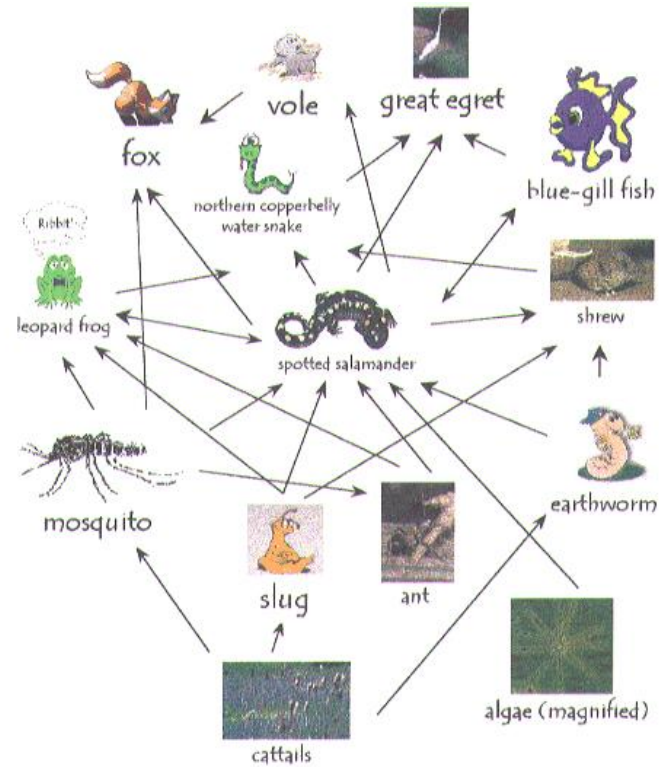


Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator. (victim to killer)

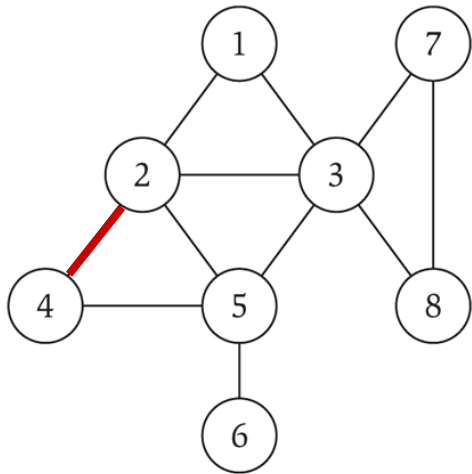


Reference: <http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif>

Graph Representation: Adjacency Matrix

Adjacency matrix. n -by- n matrix with $A_{uv} = 1$ if (u, v) is an edge, 0 otherwise

- Two representations of each edge.
- Space proportional to n^2 .
- Checking if (u, v) is an edge takes $O(1)$ time.
- Identifying all edges takes $O(n^2)$ time.



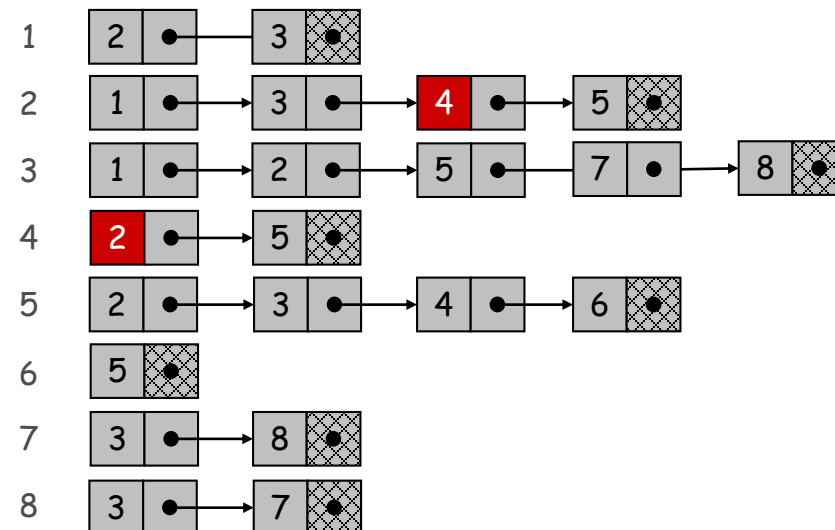
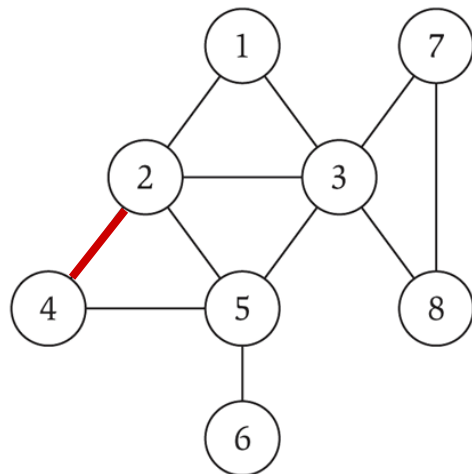
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if (u, v) is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $O(m + n)$ time.
 - In most graph algorithms, adjacency lists are preferable, as they do not waste space for non-edges.
- Nodes in Directed graphs:
 - ✎ **in-degree**: the number of incoming edges
 - ✎ **out-degree**: the number of outgoing edges.

degree = number of incident edges



Graph Problems: Clique

Given: An Undirected Graph G .

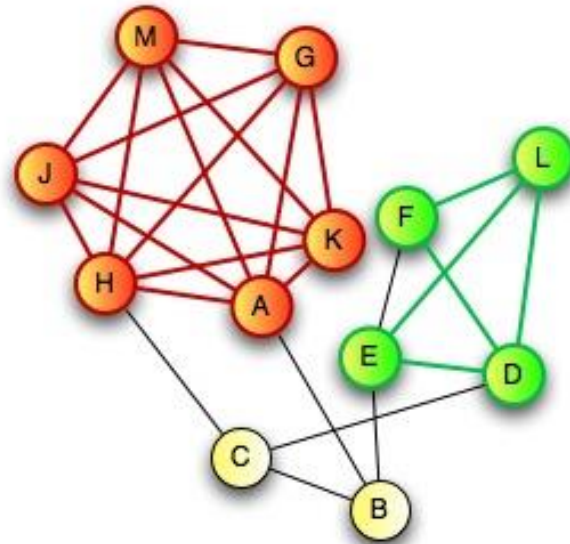
Goal: Find a clique of **maximum size**.



a subset K of nodes such that there is an edge between any two nodes in K

Motivation: The graph models an interaction network (persons in a social network, proteins in a living cell, etc.), where an edge means some close relation between two "nodes".

We may wish to **identify big groups of pairwise interacting "nodes"**.



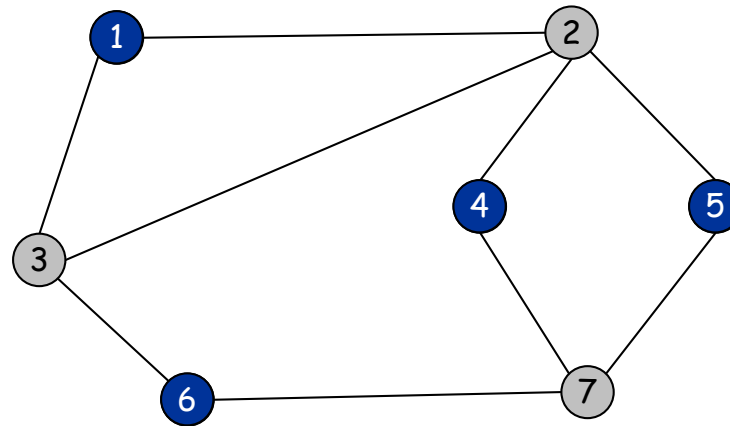
Graph Problems: Independent Set

Given: An Undirected Graph G .

Goal: Find an independent set of **maximum size**.

↑
subset of nodes such that no two joined by an edge

Motivation: The graph models conflicts between items, and we wish to select as many as possible items conflict-free.



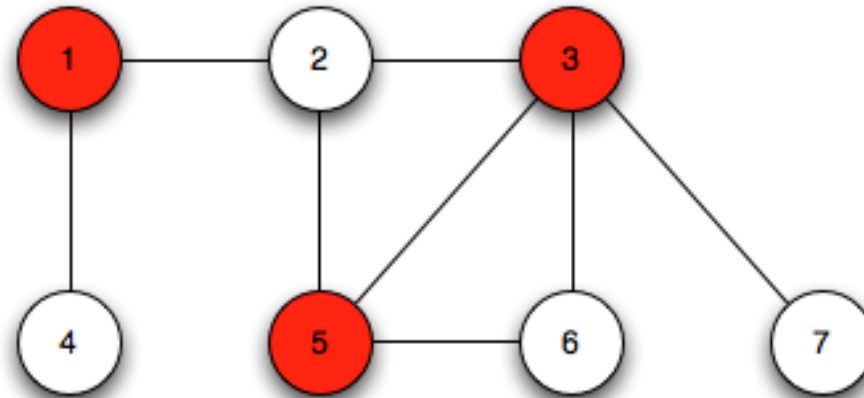
Graph Problems: Vertex Cover

Given: An Undirected Graph G .

Goal: Find a vertex cover of **minimum size**.

a subset of nodes such that every edge of G has at least one of its two nodes in it

Motivation: How can we place a minimum number of guards in a museum building so that they can watch all corridors?



Graph Problems: Difficulty

How difficult is to solve these Graph problems (some hints):

Maximal Clique: Find a clique of **maximum size**.



a subset K of nodes such that there is an edge between any two nodes in K

$k = 2$ is trivial (every pair of adjacent nodes)

But, we need to consider all possible k .

Furthermore, for every k , we need to **check all k -subsets of nodes**.

In other words, we need to check **all possible 2^n subsets of the node set**
 n could be very large...

This 2^n is not special here, same applies to e.g., Interval Scheduling...

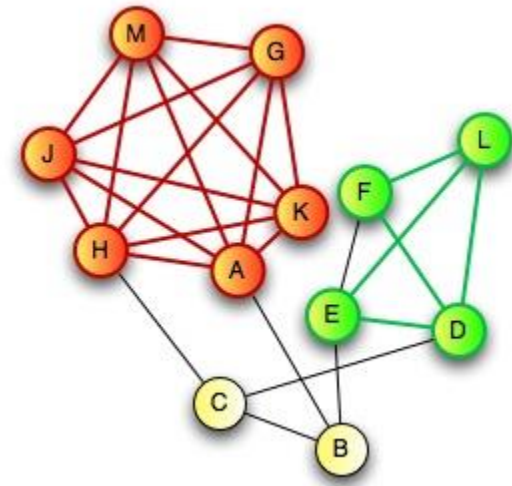
The difference is... Our Algorithm Design Approaches seem to fail here...

Next Week: Going to be hard...

for whom...?

Of course for these problems.

However, to fight against a hard opponent, **we need to work hard!**



Monday, LV 5 (2016-09-26)

1 Multiplying by Squaring

Given an integer a with n digits, we want to square a .

Given two integers a and b (each with n digits), we want to multiply $a*b$.

We can reduce one problem to the other.

Suppose we have an algorithm that can square integers in some time $\mathcal{O}(n)$. S = polynomial, $\mathcal{S}(n) \geq n$.

Using the squaring algorithm we compute: $ab = ((a + b)^2 - (a - b)^2) * \frac{1}{4}$. All operations except squaring runs in linear time (at worst, to divide by 4 in binary is constant).

Now we can multiply two arbitrary integers in: $\mathcal{O}(\mathcal{S}(n))$.

1.1 Is squaring easier than multiplication?

Squaring is obviously a special case of multiplication, hence is there a super good algorithm for squaring that we can use instead of using multiplication?

Nope.

2 Two uses of reduction

1. Solve a problem X with help of an existing algorithm for a problem Y .
2. Prove that problem X is at least as hard as problem Y .

3 Definition of Reduction

X, Y : problems

$|x|$: length of an instance x , for the problem X

If “ X is reducible to Y in $\mathcal{O}(t(n))$ time”, we can transform every instance of the first problem ($x, |x|=n$) into an instance of the second problem, $y=f(x)$ of Y .

We may then transform a solution of y back to a solution of x . All of this, in $\mathcal{O}(t(n))$ time.

3.1 Flowchart

Assume Y solvable in $\mathcal{O}(u(n))$ time.

- x (an instance of X) is transformed into an instance of the second problem (done by the reduction) $y=f(x)$. This takes $t(n)$ time.
- We now solve our second problem, and get the solution of y . This step takes $u(t(n))$ time.
- Transform the solution of y back into a solution of x . This step takes $t(n)$ time.
 - All in all, it takes $t(n) + u(t(n))$ time.

If Y is an easy problem, and the reduction is fast, X is also an easy problem. However, if X is hard, Y is hard.

4 Optimization Vs Decision

What's the difference?

4.1 Optimization

Given an instance x , find a solution with minimum/maximum value.

4.2 Decision

Given an instance x and a number k , does x possess a solution with value $\{\leq\}$ k ?

4.2.1 Reducibility for decision problems

X, Y : decision problems

$|x|$: length of instance x of X

X is reducible to Y in $\mathcal{O}(t(n))$ time. This means that we can transform every x into an instance $y=f(x)$ of Y s.t: x and y gives the same answer. I.e. x answers Yes $\Leftrightarrow y$ answers Yes. Note also that f is computable in $\mathcal{O}(t(n))$ time.

X is reducible to Y in polynomial time. This is as above, but t is polynomial. (Denoted by $X \leq_p Y$.)

5 Graph Problems

$G = (V,E), |V| = n$

Clique \leq_p Indep. Set

All cliques in a graph G is exactly the independent sets in the same graph. One can reduce clique to independent set as: Does G contain a clique of size k being equivalent to asking if \bar{G} has an independent set of k nodes. $f(\bar{G},k) = (G,k)$.

Indep Set \leq_p Clique

In the same way we can reduce independent set to clique: $f(G,k) = (\bar{G},k)$.

If we have vertex cover in some graph: $U \subseteq V$ vertex cover $\Leftrightarrow V \setminus U$ indep. set.

Vertex Cover \leq_p Indep Set

$f(G,k) = (G, n-k)$

We can also do: Indep Set \leq_p Vertex Cover.

6 Generic Reductions

Suppose X is a “special case” of Y , like squaring is a special of multiplication and interval scheduling of weighted interval scheduling. Then it is the case that $X \leq_p Y$ (via $f(x)=x$).

6.1 Interval scheduling \leq_p Independent Set

Given four intervals: $[1,5], [2,6], [3,8], [7,9]$ we want to see if we can find a subset of k pairwise disjoint intervals we can draw a graph where each interval is a vertex and each interval that intersects another interval has an edge between its nodes.

The reverse is however not trivial. Ptr cannot show this. :(Can it even be done? Think of a “circle graph”, the fifth interval must intersects only the first and last intervals – cannot be done.

6.2 Transitivity

If $X \leq_p Y \wedge Y \leq_p Z \Rightarrow X \leq_p Z$

Proof:

$X \xrightarrow{f} Y \xrightarrow{g} Z$
f is computable in time $p(n)$
g is computable in time $q(n)$
 p, q are polynomial in time

$X \leq_p Z$ via $h = g \circ f$:
 x is Yes $\Leftrightarrow f(x)$ is Yes $\Leftrightarrow g(f(x))$ is Yes

h is computable in $p(n) + q(p(n))$ which is polynomial.

7 NP or P?

7.1 Polynomial Time: \mathcal{P}

\mathcal{P} : class of decision problems that can be solved by an algorithm that runs in polynomial time.

If $X \leq_p Y \wedge Y \in \mathcal{P} \Rightarrow X \in \mathcal{P}$.

Proof:

$X \xrightarrow{f} Y$
f computable in $p(n)$ time
Y solvable in $q(n)$ time
 p, q are polynomial in time

To solve x , we compute $f(x)$ and solve this instance. The time required:
 $p(n) + q(p(n))$.

7.2 Non Deterministic Polynomial Time: \mathcal{NP}

\mathcal{NP} : class of all decision problems for which a polynomial-time algorithm exists that verifies solutions that are already given to Yes-instances.

7.2.1 Clique example

We do not know how to solve this problem efficiently. However, if someone gives us a solution we can simply count if the clique contains k nodes, then we check that each node has an edge to all other nodes. This can be done in polynomial time.

Wednesday, LV 5 (2016-09-28)

1 NP Completeness

The N in NP stands for “Non deterministic”.

A problem Y is called NP-complete if $Y \in \mathcal{NP}$ and $\forall x \in \mathcal{NP} : X \leq_p Y$. It contains the whole difficulty of the class \mathcal{NP} . There is no intersections between the \mathcal{P} and \mathcal{NP} -complete set. I.e., as long as $\mathcal{P} \neq \mathcal{NP}$, there is no \mathcal{NP} -complete problem in \mathcal{P} .

Proof

Assume $Y \in \mathcal{P}$ and Y is \mathcal{NP} -complete. It then follows that $\forall x \in \mathcal{NP} : X \leq_p Y$ and $\forall x \in \mathcal{NP} : X \in \mathcal{P}$. *This is not possible?*

Assume that (Y is \mathcal{NP} -complete) and ($Y \leq_p Z$) and ($Z \in \mathcal{NP}$), it then follows that Z is \mathcal{NP} -complete.

Proof

if $\forall x \in \mathcal{NP} : X \leq_p Y \leq_p Z$ then $\forall x \in \mathcal{NP} : X \leq_p Z$ due to transitivity.

Note: If you reduce X to Y it means that Y is at least as hard as X !!! If you are asked to prove a reduction $X \leq_p Y$ you shall show that you can solve X using Y .

Note 2: The \mathcal{NP} -problems are hard to solve, easy to check, but the \mathcal{NP} -complete are hard to check and to solve. Only the \mathcal{NP} -complete problems are non-polynomial.

1.1 Find an \mathcal{NP} -complete problem - The SAT Problem

Boolean variables: x_i

Literals: $x_i, \bar{x}_i (\neg x_i)$

Clause: \vee of literals

Conjunctive Normal Form (CNF): \wedge of clauses

1.1.1 Example of \mathcal{NP} -complete problem - CNF that describes vertex cover $\leq k$

Given a graph G we want to find a way to “cover all edges” (guard at the museum) using at most k edges. If we have a graph with vertices $(1, 2, 3, 4)$ and edges $((1,2), (1,3), (2,3), (3,4))$ we can achieve this with CNF:

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4)$$

SAT: Given a CNF, find a satisfying assignment of truth values. In our example we would set $x_1=x_3=1$ and $x_2=x_4=0$.

$\text{SAT} \in \mathcal{NP}$, if someone gives us a satisfied assignment we can check if the assignment is valid in polynomial time (linear in this case). The proof that $\text{SAT} \in \mathcal{NP}$ -complete is long...

1.2 K-SAT

A SAT-problem where the CNF contains at most k literals in each clause.

1.2.1 Proof that $\text{SAT} \leq_p \text{3-SAT}$

We are given an instance of the first problem (a CNF with unlimited length-clauses) ϕ . Now we have to construct an equivalent formula that has at most 3 literals in every clause.

Take any clause C in ϕ , that is too long, and try to split it into two sets A and B . This leads us to $C = A \vee B$. Replace $A \vee B$ with two shorter clauses $A \wedge B$ (this does not work since $\vee \neq \wedge$). However, by introducing a fresh variable u we can write: $A \vee B \Leftrightarrow (A \vee u) \wedge (B \vee \neg u)$.

Proof that $A \vee B \Leftrightarrow (A \vee u) \wedge (B \vee \neg u)$

If A is true, $A \vee u$ holds. We can choose u to be false, hence $\neg u$ is true and $(A \vee u) \wedge (B \vee \neg u)$ is true. P tr also showed the other way.

Iterate this until all clauses have at most three literals.

ψ : The generated formula
 ϕ is satisfiable $\Leftrightarrow \psi$ is satisfiable

Does this run in polynomial time? Yes, because of some sum of cube argument.

3-SAT is also \mathcal{NP} -complete!

1.2.2 Proof sketch that 3-SAT \leq_p Indep. Set

$(x_i \vee x_j \vee x_k) \wedge \dots$, where each literal could be negated.
m is the number of clauses
n is the number of variables

Every clause is represented by a triangle in a graph and each vertex is labeled by the name of their corresponding literal in the CNF. For every variable we also have an edge $(x_i - \neg x_i)$, ... We now connect each vertex in a triangle with its corresponding "edge vertex".

The given CNF is satisfiable \Leftrightarrow G has an independent set with $m+n$ nodes.

Proof

If the formula is satisfiable it means that we have one true literal in every clause (one true vertex in each triangle). We mark the true vertex in each triangle as well as the false literal vertex in the "edges".

If we have an independent set of size $m+n$ we know that we can have at most one marked vertex in each triangle and at most one node from each edge pair.

Indep. Set is \mathcal{NP} -complete

1.3 Subgraph Isomorphism

Given two graphs G and H we ask ourselves if H is a subgraph of G. We try to solve this by: Clique \leq_p Subgraph Isomorphism.

1.3.1 Reduction

Given a graph G and a number k we ask if the graph contains a clique of size k. Hence $f(G,k) = (G,H)$ where H is a clique of size k. H is exponentially larger than k (exponentially blow up). This is a polynomial time reduction, because we consider the instance as a whole (the graph and the number).

Algorithms: Lecture 10

Chalmers University of Technology

Today's Topics

Basic Definitions

Path, Cycle, Tree, Connectivity, etc.

Graph Traversal

Depth First Search

Breadth First Search

Testing Bipartateness (One Graph Two colors)

Cycles

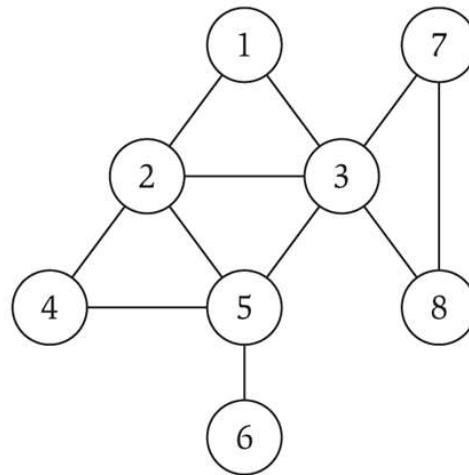
Def. A **path** is a sequence of $v_1, v_2, \dots, v_{k-1}, v_k$ nodes with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G .

Def. A **cycle** is a path $v_1, v_2, \dots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

Directed G : **directed path, cycle**

Every pair (v_i, v_{i+1}) in the **path** or **cycle** is joined by a **directed edge**

➤ must respect the directionality of edges.

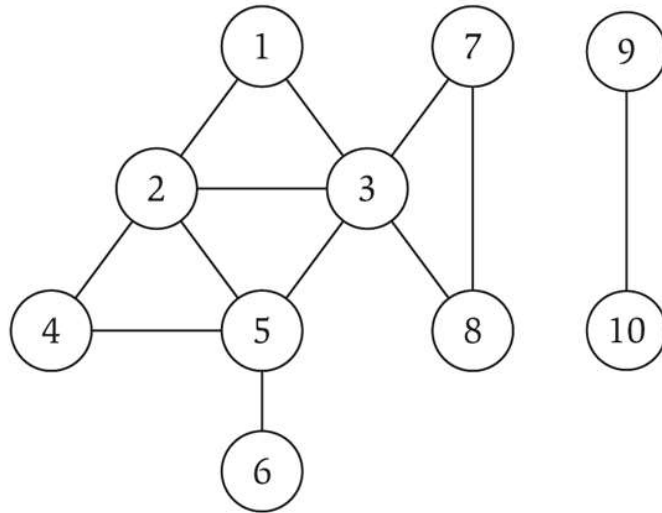


cycle $C = 1-2-4-5-3-1$

Not a cycle: $1-3-8-7-3-1$

Connected Graph

Def. An undirected graph is *connected* if, for *every pair* of nodes u and v , there is *a path from u to v* .

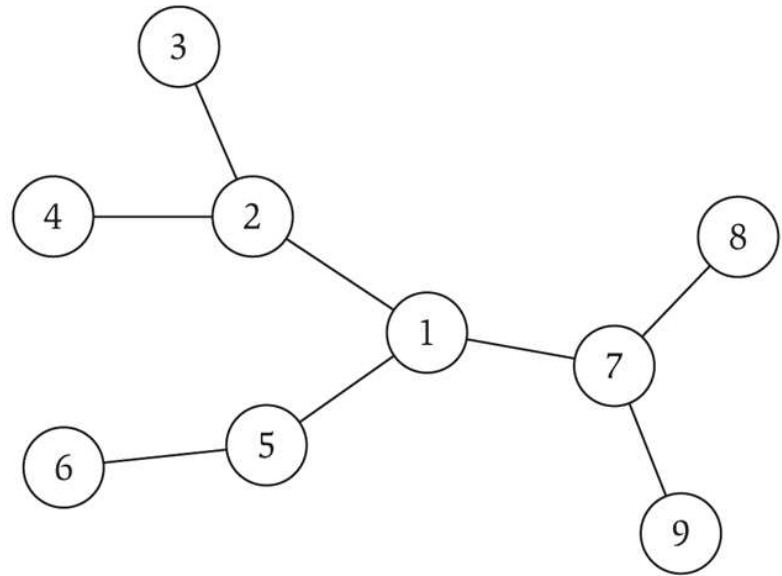


Trees

Def. An undirected graph is a **tree** if it is **connected** and **does not** contain a **cycle**.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

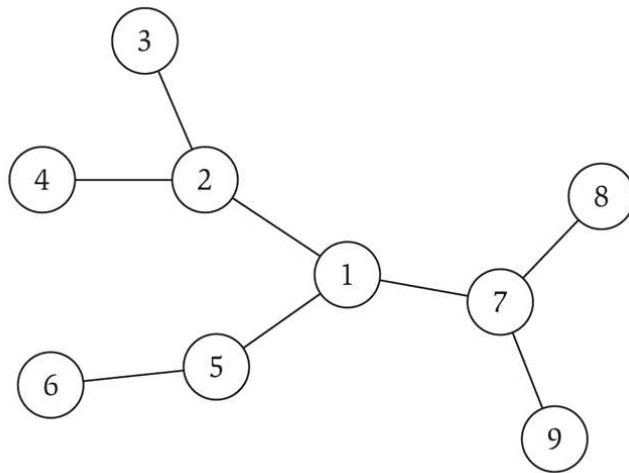
- G is connected.
- G does not contain a cycle.
- G has $n-1$ edges.



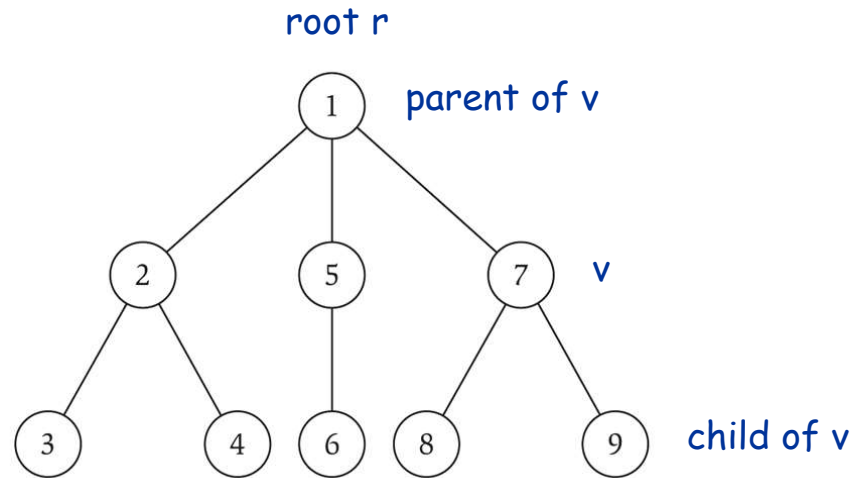
Rooted Trees

Rooted tree. Given a tree T , choose a **root node r** and **orient each edge away from r** .

Importance. Models hierarchical structure.



a tree

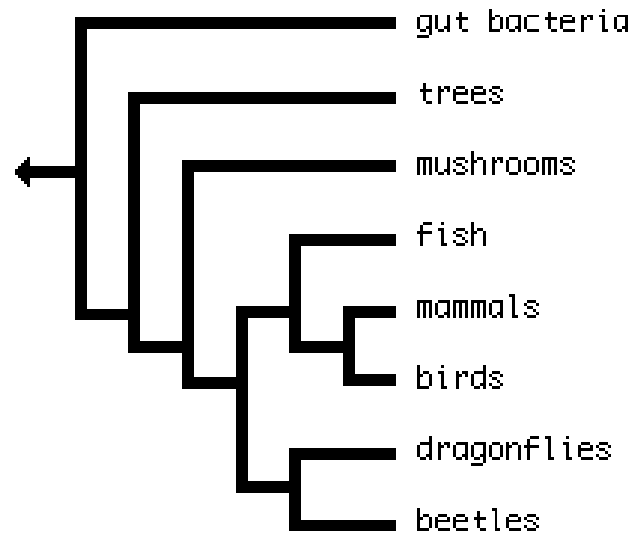


the same tree, rooted at 1

Phylogeny Trees

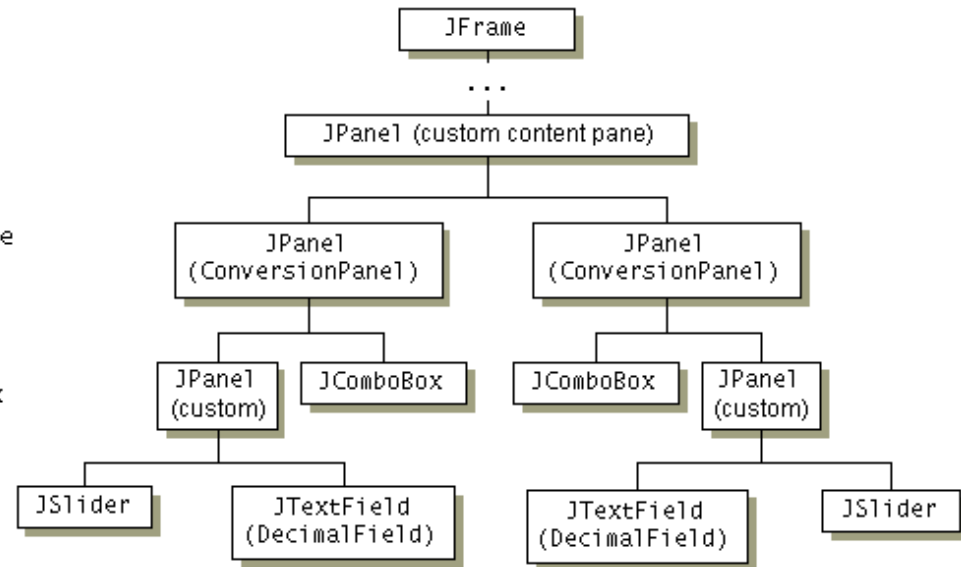
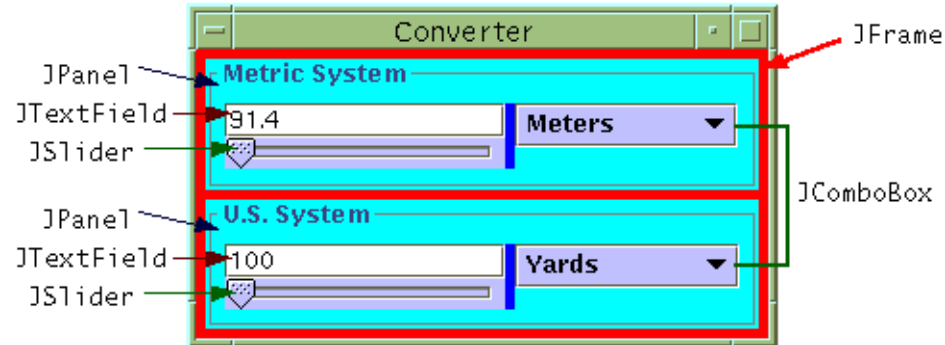
Phylogeny trees. Describe evolutionary history of species.

- biologists draw their tree from left to right



GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.



Graph Traversal

Connectivity

s-t connectivity problem. Given two node **s** and **t**, is there a **path** between **s** and **t**?

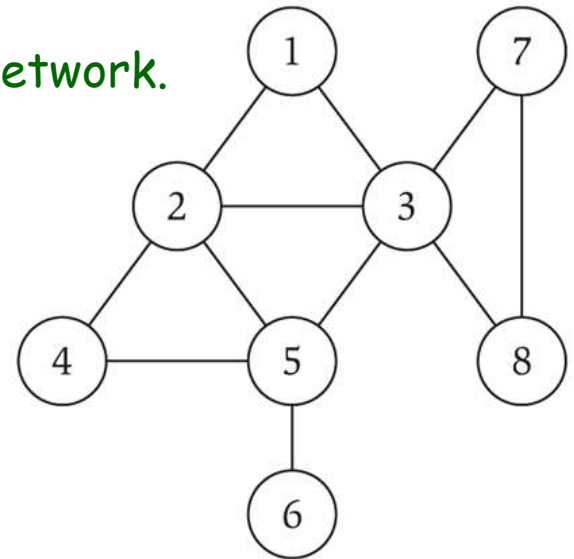
s-t shortest path problem. Given two node **s** and **t**, what is the length of the **shortest path** between **s** and **t**?

Applications.

- **Social Network:** deciding whether two persons are connected through a chain of friends.
- **Fewest number of hops in a communication network.**
- **Maze traversal.**

How to proceed to find 1-6 connectivity?

- **Need a systematic exploration of the graph**



Breadth First Search

Breadth First Search (BFS):

BFS(s) // Find a BFS tree rooted at s , which includes all nodes reachable from node s .

Create a Boolean array Discovered[1... n], Set Discovered[s] = true and Discovered[v] = false for all other v .

Create an empty FIFO queue Q , add node s to Q .

while Q is not empty

 dequeue a node u from Q

 for each node v adjacent to node u

 if Discovered[v] is false then

 add node v to Q , set Discovered[v] to true

 endif

 endfor

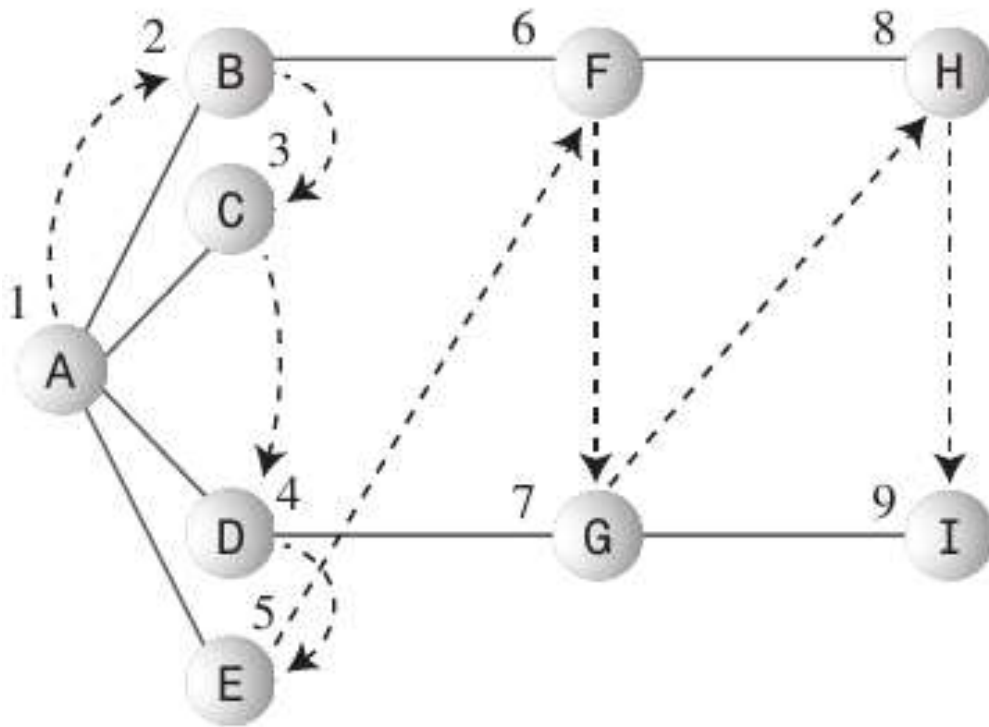
endwhile

Breadth First Search

BFS Tree:

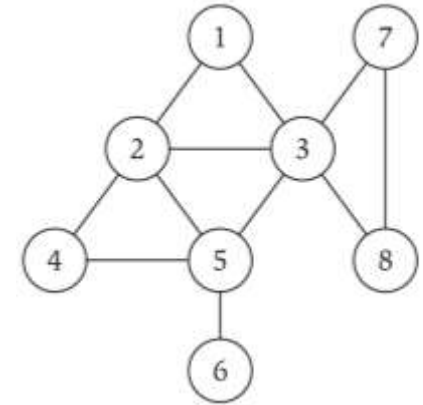
Create an empty tree T

Add edge (u, v) to the tree T , when v is discovered the first time



<http://i.stack.imgur.com/TjhfH.png>

Breadth First Search: Analysis



Analysis:

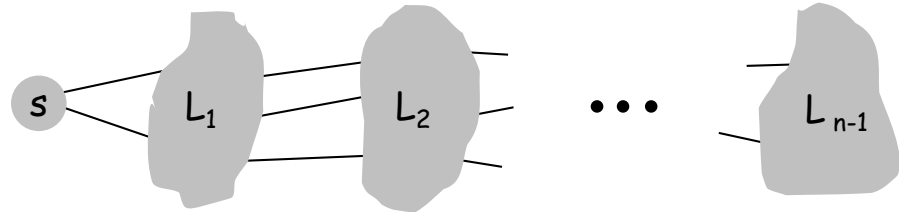
- A node u enters Q at most once, and the for loop needs nodes adjacent to every such u
- $O(1)$ to process an edge
- Finding all v adjacent to u :
 - Adjacency Matrix:
 - we have to check all matrix entries in u 's row: $O(n)$
 - total time required to process all rows of the Matrix: $O(n^2)$
 - Adjacency List:
 - when we consider node u , there are $\text{deg}(u)$ incident edges (u, v)
 - total time processing all the edges is $\sum_{u \in V} \text{deg}(u) = 2m \Rightarrow O(m)$
 - setup time for the array Discovered is $O(n)$, $\Rightarrow O(m + n)$
 - m is at least $n-1$ for connected graph, m dominates $\Rightarrow O(m)$

each edge (u, v) is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$

Breadth First Search: Properties

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

BFS algorithm partitions the nodes into layers:



- $L_0 = \{ s \}$.
 - $L_1 =$ all neighbors of L_0 .
 - $L_2 =$ all nodes that do not belong to L_0 or L_1 , and that have an edge to a node in L_1 .
 - $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i .
- Implementation using Queue processes the nodes exactly layer by layer
- explores in order of distance from s .

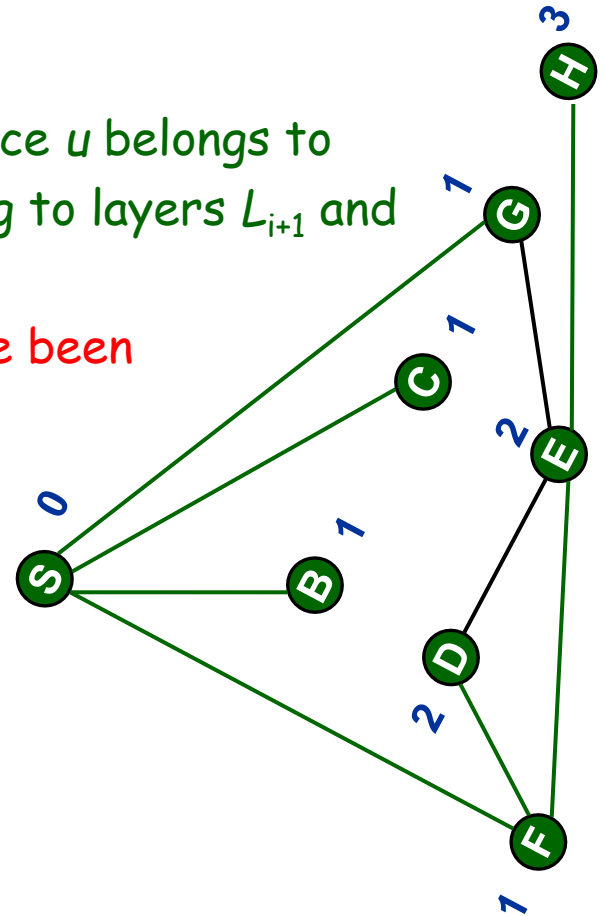
Breadth First Search: Properties

Property. Let T be a BFS tree of $G = (V, E)$, nodes u, v belong to T , and let (u, v) be an edge of G . Then the **level of u and v differ by at most 1**.

Let u, v belong to layers L_i and L_j respectively.

Suppose $i < j - 1$. (a contradiction)

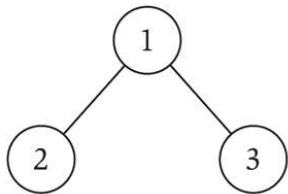
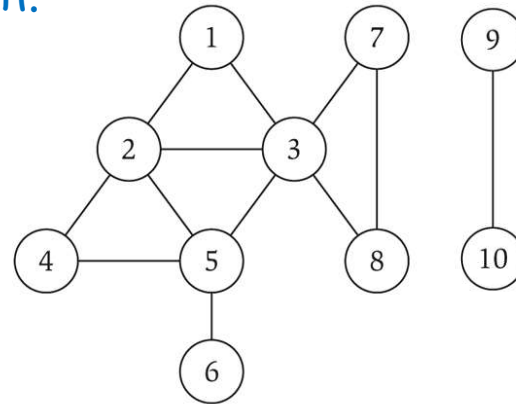
- When BFS examines the edges incident to u , since u belongs to layer L_i , the only nodes discovered from u belong to layers L_{i+1} and earlier;
- hence, if v is a neighbor of u , then it should have been discovered by this point at the latest, and
- should belong to layer L_{i+1} or earlier



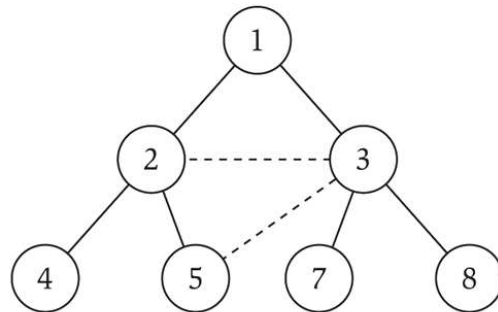
Breadth First Search: Properties

What follows:

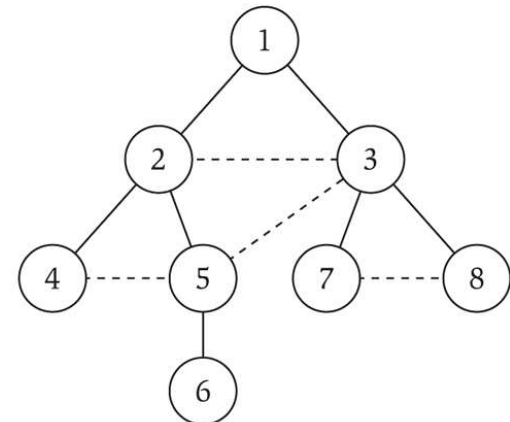
- For each i , L_i consists of all nodes at distance exactly i from s .
- There is a path from s to t iff t appears in some layer.
- Moreover: s - t is a shortest path.



(a)



(b)



(c)

L_0

L_1

L_2

L_3

Depth First Search

Depth First Search (DFS):

Create a Boolean array `Explored[1...n]`, initialized to false for all.

DFS(*u*)

set `Explored[u]` to true

for each node *v* adjacent to node *u*

 if `Explored[v]` is false then

 DFS(*v*)

 endif

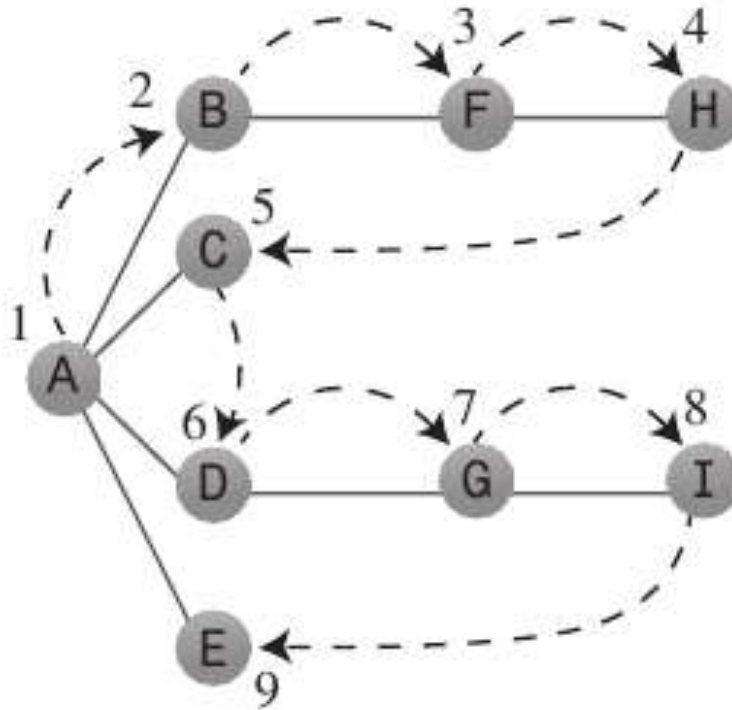
endfor

- Call DFS(*s*)
 - each recursive call is done only after termination of the previous call, this gives the desired depth first behavior.
- In iterative implementation, maintain a stack explicitly.

Depth First Search

DFS tree:

Take an array `parent`, set `parent[v] = u` when calling `DFS(v)` due to edge (u, v) .
While setting u ($u \neq s$) as Explored, add the edge $(u, \text{parent}[u])$ to the tree.

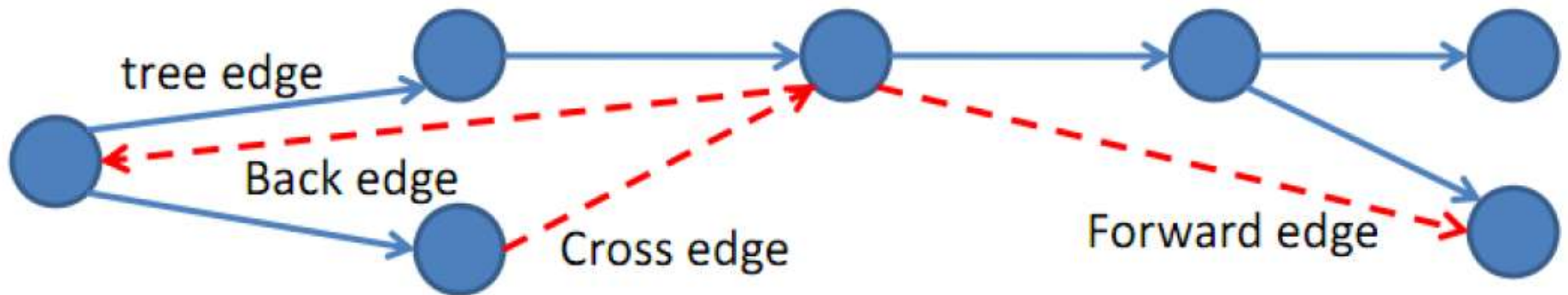


Depth First Search: Edge Classification

Edge Classification:

As we execute DFS, an edge (u, v) can be classified into **four** edge types.

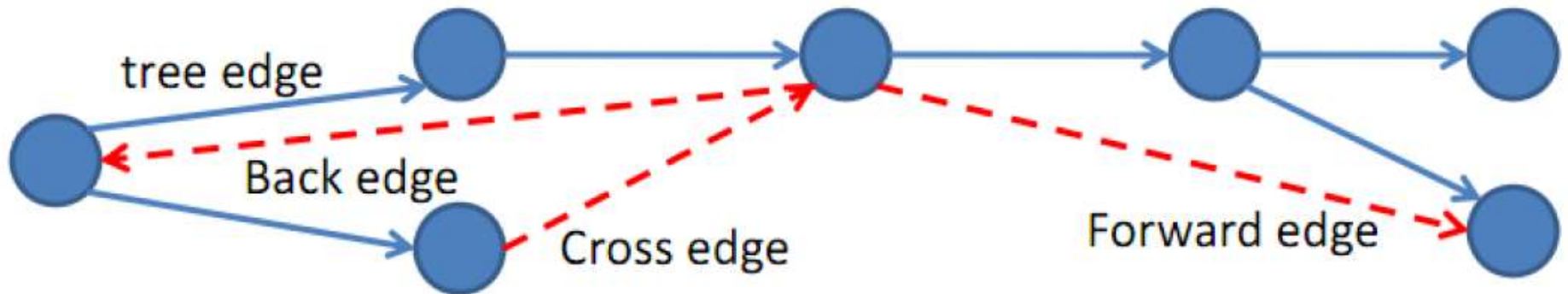
- IF v is visited for the first time as we traverse the edge (u, v) ,
 1. then the edge is a **tree edge**.
- ELSE, v has already been visited:
 2. If v is an ancestor of u , then (u, v) is a **back edge**.
 3. Else, if v is a descendant of u , then (u, v) is a **forward edge**.
 4. Else, if v is neither an ancestor or descendant of u , then (u, v) is a **cross edge**. (u and v belong to different paths from the root)



Depth First Search: Edge Classification

Important Properties:

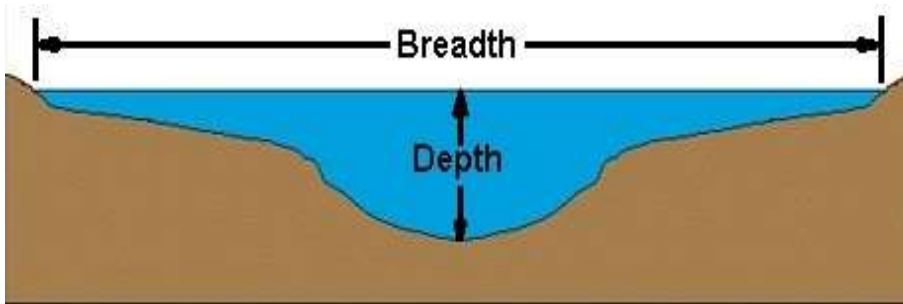
- tree edges form the DFS tree in G .
- G has a **cycle** if and only if DFS finds **at least one back edge**.
- G is undirected graph: **it cannot contain forward edges and cross edges**
 - the edge (v, u) would have already been traversed during DFS before we reach u and try to visit v via edge (u, v) .



BFS vs. DFS

BFS: Put unvisited vertices on a queue.

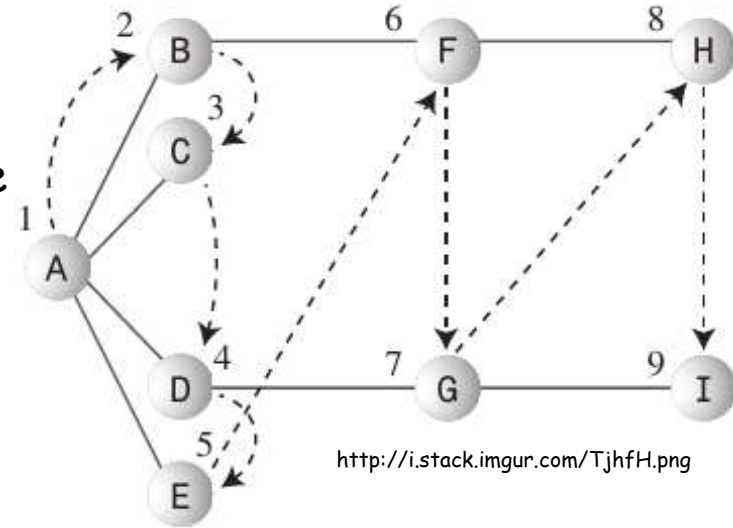
- Examines vertices in increasing distance from s .
- Using adjacency list requires $O(m)$.



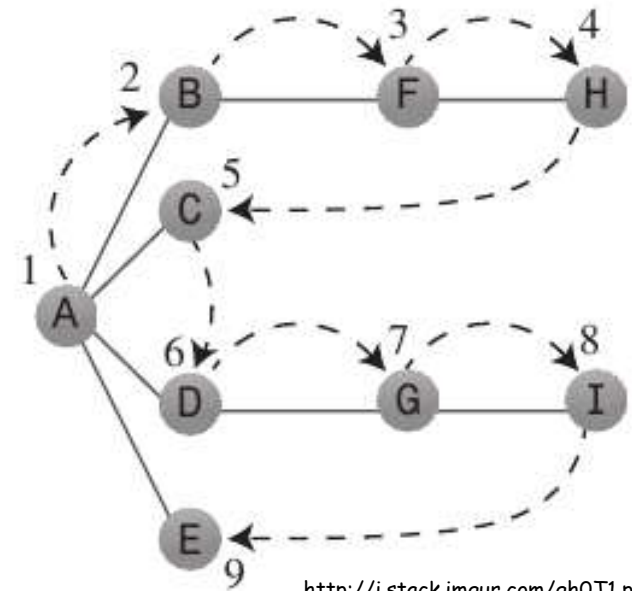
<http://i.stack.imgur.com/QtYo8.jpg>

DFS: Put unvisited vertices on a stack.

- tries to explore as deeply as possible
 - Mimics maze exploration.
- $O(m)$ due to similar reasoning.



<http://i.stack.imgur.com/TjhfH.png>



<http://i.stack.imgur.com/gh0T1.png>

Finding all Articulation Points

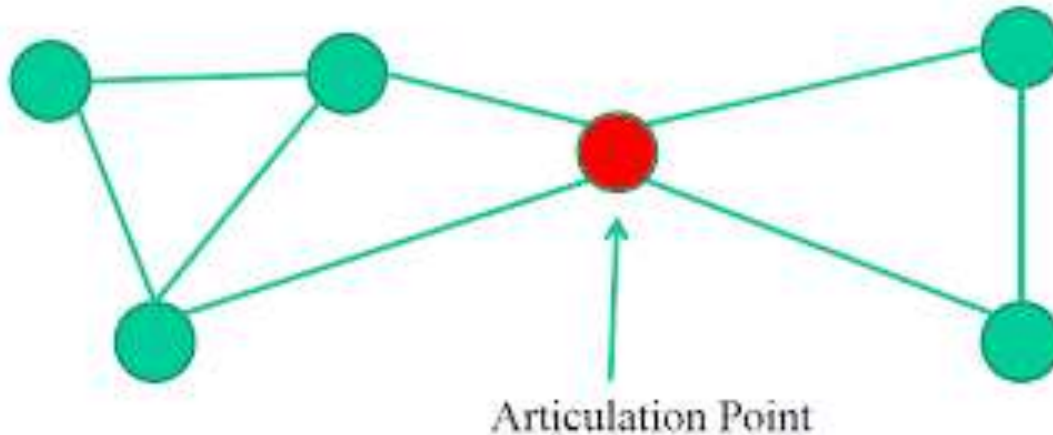
Def. An articulation point in a **connected, undirected graph** is a node v such that **removal of v** and of the edges incident to v makes the **graph disconnected**.

Given: an undirected graph $G = (V, E)$.

Goal: Find all articulation points.

Motivation:

reliability (failure tolerance) of communication networks



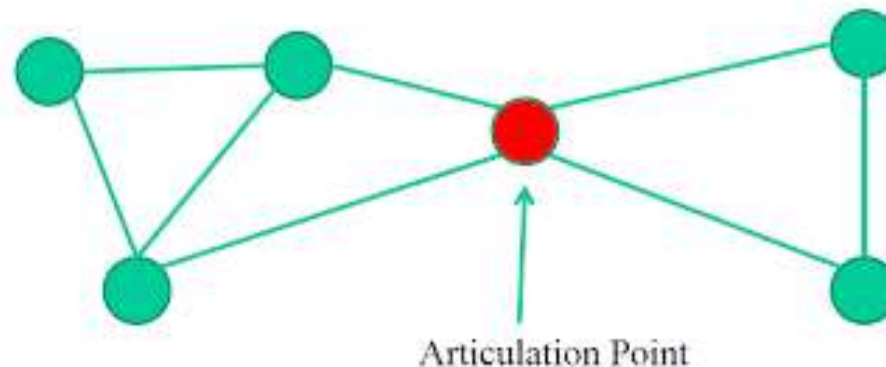
Finding all Articulation Points

Given: an undirected graph $G = (V, E)$.

Goal: Find all articulation points.

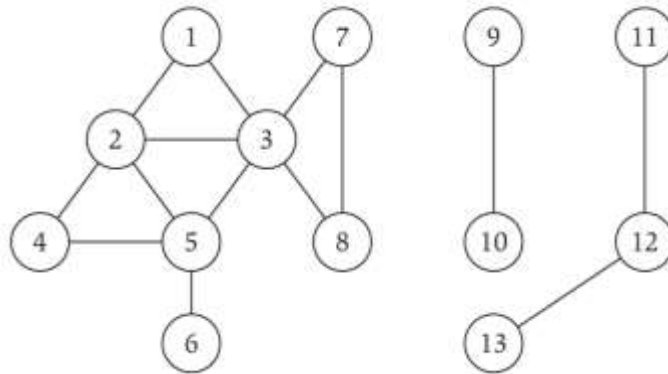
Observation: there no cross edges in DFS tree on undirected graphs.

- Chose arbitrary start node s and find a DFS tree rooted at s
 - s is an articulation point iff s has more than one child in the DFS tree
- run DFS n times, once from every start node, which costs $O(nm)$ time.
 - Amazingly, it is possible achieve $O(m)$ time, using only one DFS tree and a variant of dynamic programming



Connected Component

Connected component. Find all nodes reachable from s .



Connected component containing node 1 = $\{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

Connected Component

Connected component. In undirected graph G , Find all nodes reachable from s .

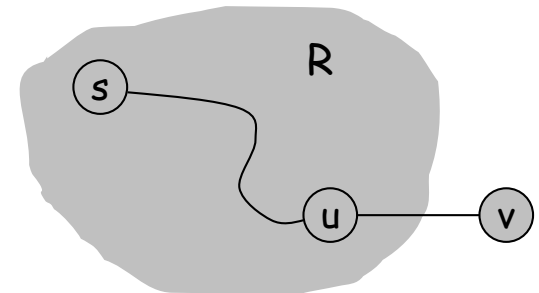
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

 Add v to R

Endwhile



it's safe to add v

Upon termination, R is the connected component containing s .

➤ **BFS, DFS $\Rightarrow O(m)$**

G is connected?

➤ iff all nodes are reachable from arbitrary node s

All connected components of G ?

➤ $O(m + n)$: If the search has aborted without finding all nodes, restart the search in a yet unmarked node, and so on.

Strong Connectivity

A directed graph G is strongly connected if, for every two nodes u and v , there is a path from u to v and a path from v to u .

- The street map of a city with one-way streets should be strongly connected, or the traffic planners made a mistake.

G is a directed graph, Is G Strongly Connected?

- G^{rev} : obtained from G simply by reversing the direction of every edge
- Run BFS (or DFS) with an arbitrary start node s , once on each G , and G^{rev}
 - both searches reach all nodes IFF G is Strongly Connected

Strong Connectivity

All Strongly connected components of G ?

- Find out the strongly connected component containing s (same as in Strong Connectivity check)
- Restart the search in a node, yet not part of any strong component,
- and so on.
- $O(nm)$: Worst case G has many small connected components, $O(m)$ time for each one using the above approach.
- Remark: $O(m)$ is achievable by some tricky use of DFS, we skip.

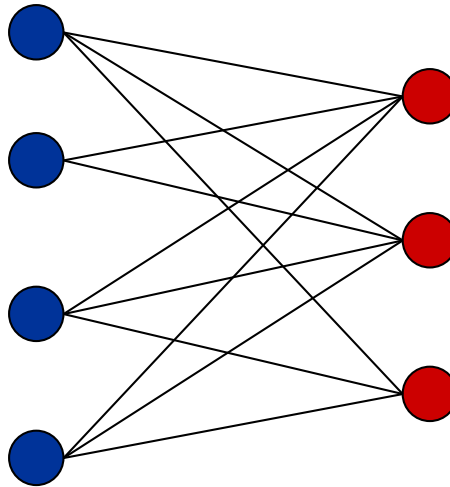
Testing Bipartiteness (One Graph Two colors)

Bipartite Graphs

Def. An **undirected** graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that **every edge** has **one red** and **one blue** end.

Applications.

- Scheduling: machines = red, jobs = blue.

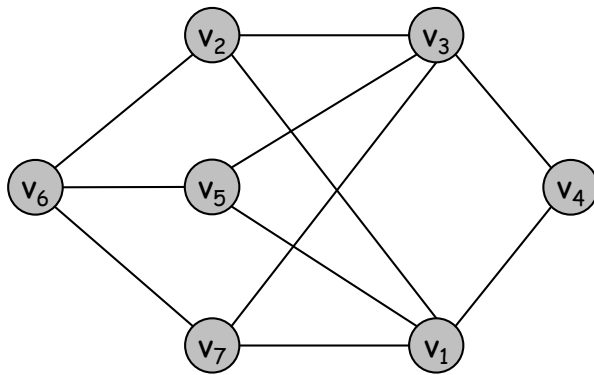


a bipartite graph

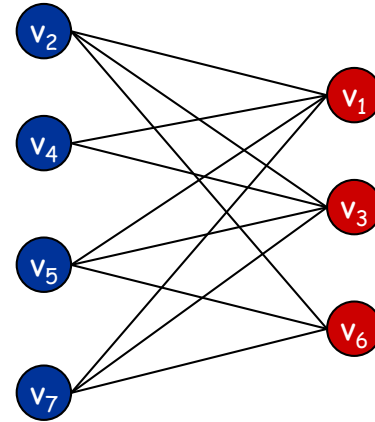
Testing Bipartiteness

Testing bipartiteness. Given a graph G , is it bipartite?

- Many graph problems become:
 - easier if the underlying graph is bipartite (matching)
 - tractable if the underlying graph is bipartite (independent set)

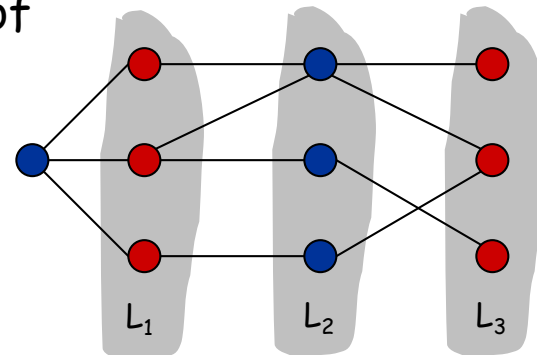


a bipartite graph G



another drawing of G

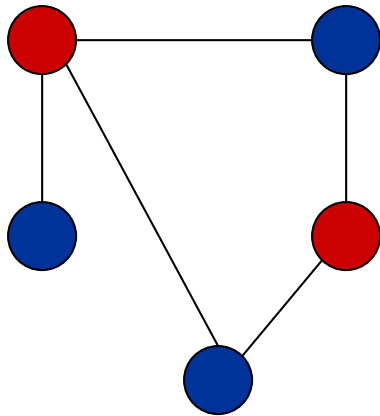
- we need to understand structure of bipartite graphs.



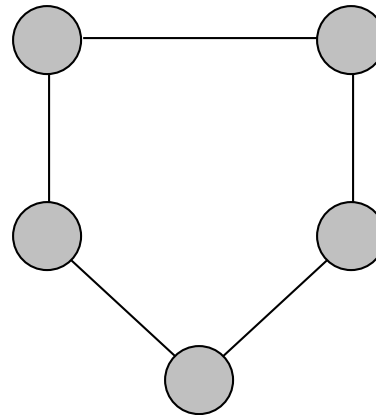
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an **odd length cycle**.

Pf. Not possible to 2-color the odd cycle, let alone G .



bipartite
(2-colorable)

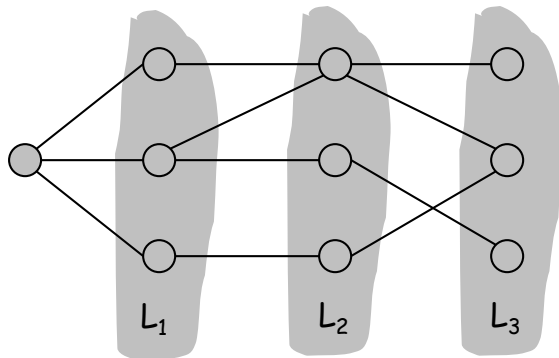


not bipartite
(not 2-colorable)

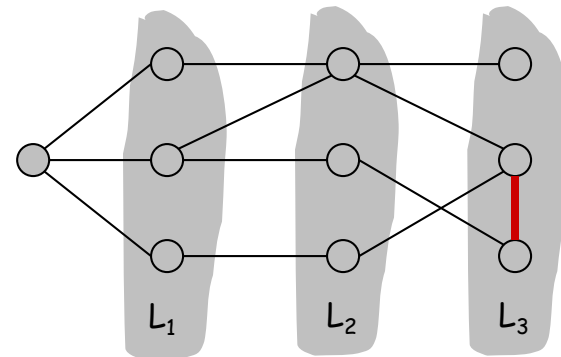
Bipartite Graphs

Lemma. Let G be a **connected** graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

- (i) No edge of G joins two nodes of the same layer, and G is bipartite.
- (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

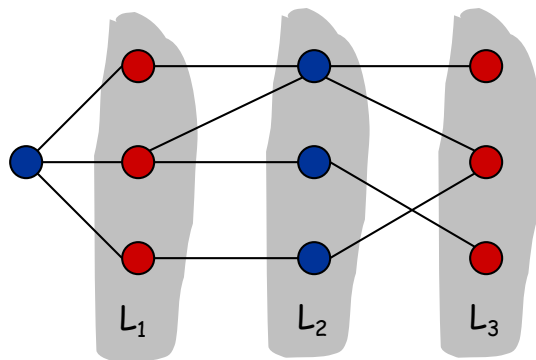
Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

Pf. (i)

- By assumption edges don't join nodes on same layer
 - Every edge joins two nodes in adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

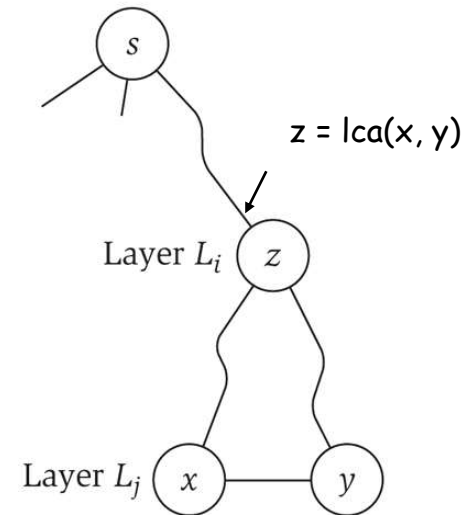
Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

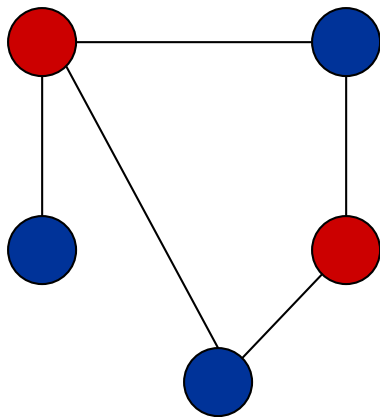
- Suppose (x, y) is an edge with x, y in same level L_j .
- Let $z = \text{lca}(x, y) =$ lowest common ancestor.
- Let L_i be level containing z .
- Consider cycle that takes edge from x to y , then path from y to z , then path from z to x .
- Its length is $1 + (j-i) + (j-i)$, which is $2(j-i) + 1$, an odd number. ▪

$$\underbrace{1}_{(x, y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$$

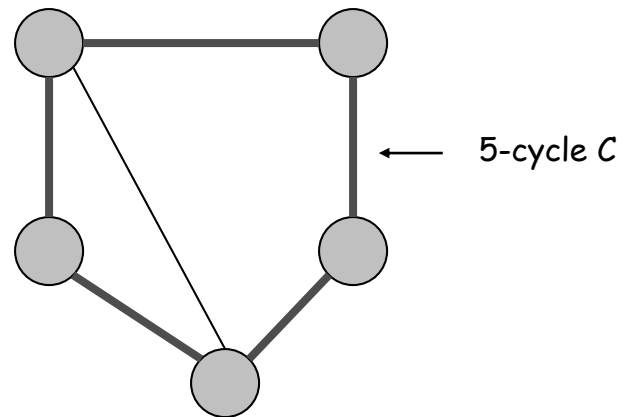


Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Algorithms: Lecture 11

Chalmers University of Technology

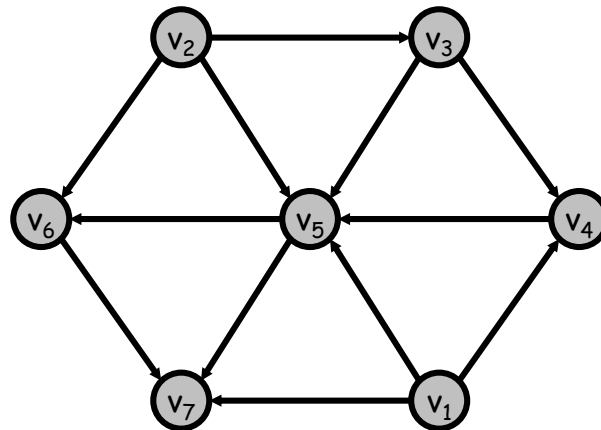
Today's Topics

Directed Acyclic Graphs and Topological Ordering

Minimum Spanning Trees

- Prim's Algorithm
- Krusal's Algorithm

DAGs and Topological Ordering



Directed Acyclic Graphs (DAG)

Def. A **Directed Cycle** in a directed graph is a cycle that can be traversed respecting the orientation of the edges: $v_1, v_2, v_3, \dots, v_n, v_1$, where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge.

Def. A **DAG** is a directed graph that contains **no directed cycles**.

Given: a directed graph $G = (V, E)$.

Goal: Find a directed cycle in G , or report that G is a DAG.

Motivation: Precedence constraints: edge (v_i, v_j) means v_i must precede v_j .

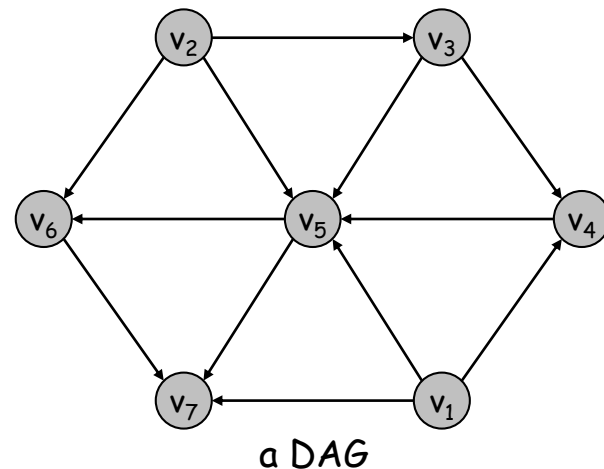
Task v_i must be done before v_j

tasks can be calculations in a program

or logic circuit, jobs in a project,

steps in a manufacturing process, etc.

A directed cycle would indicate error in the design of such models



Topological Order

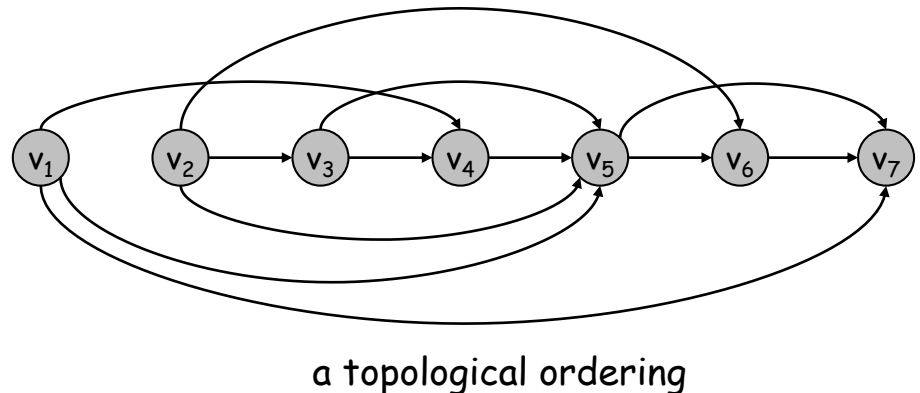
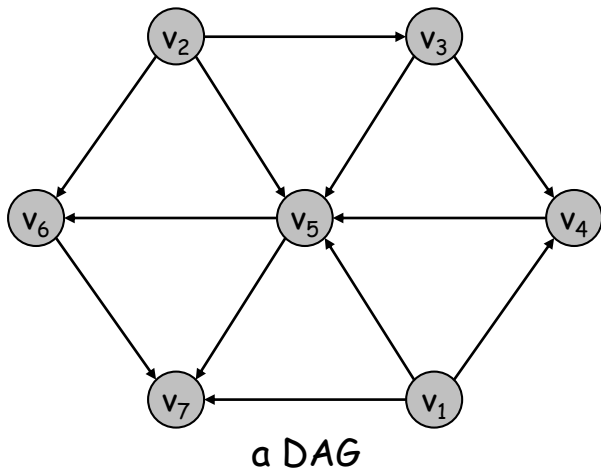
Def. A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as v_1, v_2, \dots, v_n so that **all directed edges go to the right**, that is for every edge (v_i, v_j) we have $i < j$.

Given: a directed graph $G = (V, E)$.

Goal: Construct a topological order of G , or report that G does not admit a topological order.

Motivation: Nodes are the tasks with pairwise dependency constraints.

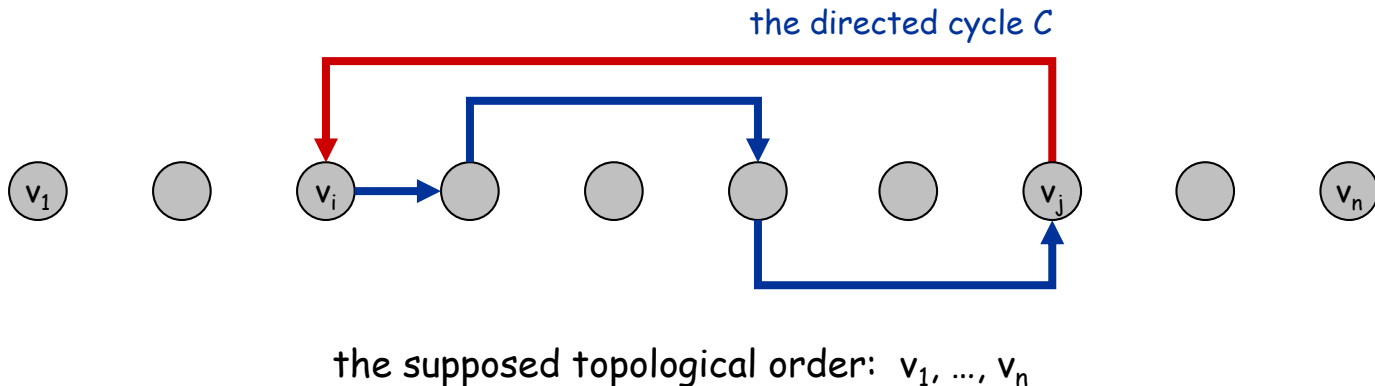
Existence of a topological order for G guarantees it is a DAG



Directed Acyclic Graphs

If G has a topological order, then G is a DAG.

- Topological order allows only edges from **left to right**
- **With this we can never close a cycle.**
- **Closing a cycle would require an edge from right to left which is against the definition of topological order.**



Directed Acyclic Graphs

Q. Does every DAG have a topological ordering?

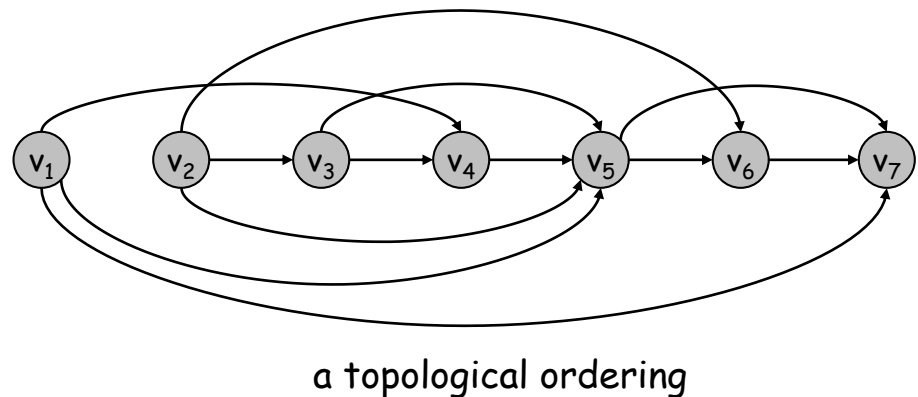
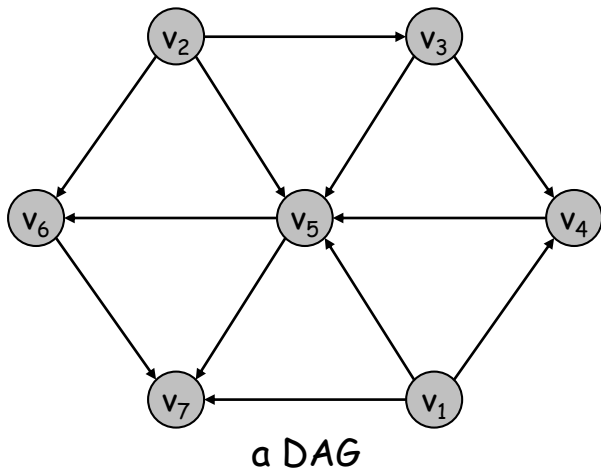
Q. If so, how do we compute one?

Stronger Version: G is a DAG if and only if G has a topological order

- If part (If G has a topological order, then G is a DAG)... done already
- Only if: If G is a DAG, then G has a topological ordering.

Observation:

A topological ordering must start with a node with **NO incoming edges**.

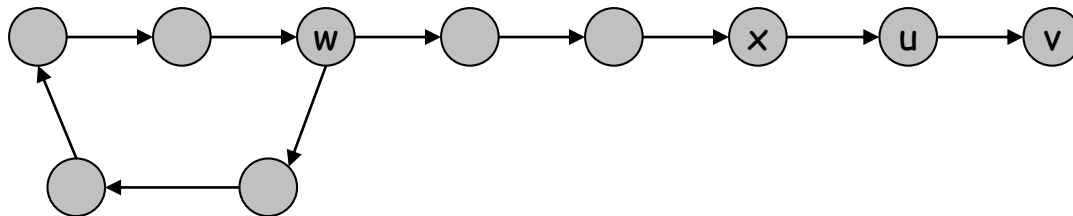


Directed Acyclic Graphs

If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v , and begin following edges backward from v . Since v has at least one incoming edge (u, v) we can walk backward to u .
- Then, since u has at least one incoming edge (x, u) , we can walk backward to x .
- Repeat until we visit a node, say w , twice.
- Let C denote the sequence of nodes encountered between successive visits to w . C is a cycle. ▪



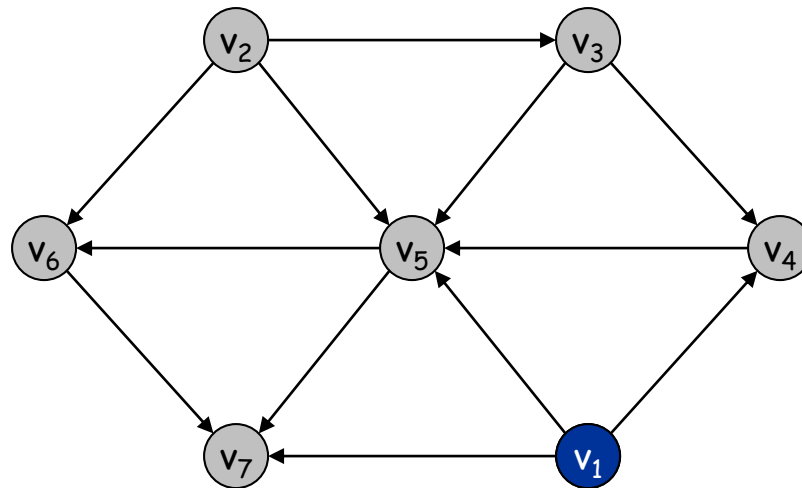
Directed Acyclic Graphs

If G is a DAG, then G has a topological ordering.

Pf.

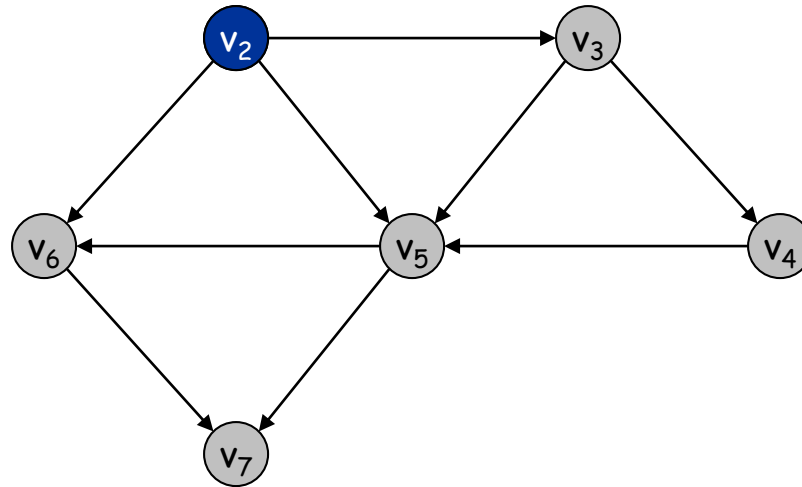
- Lets solve an example first: Given a DAG we find out topological ordering.
- Then, we proceed to the formal proof.

Topological Ordering Algorithm: Example



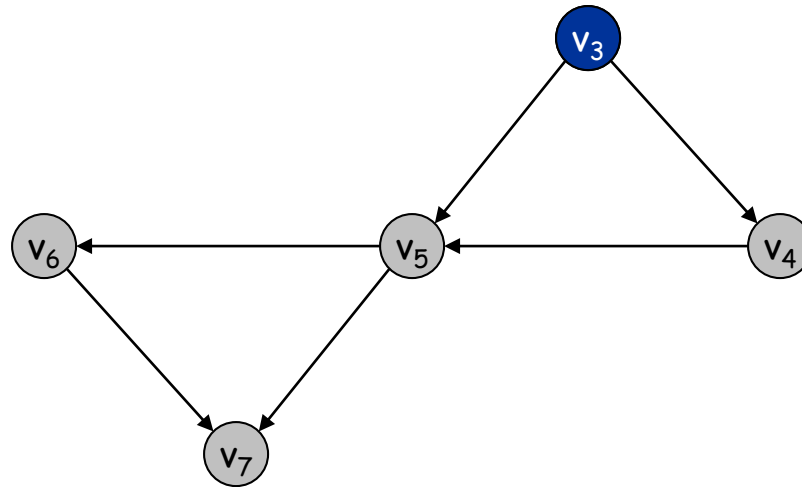
Topological order:

Topological Ordering Algorithm: Example



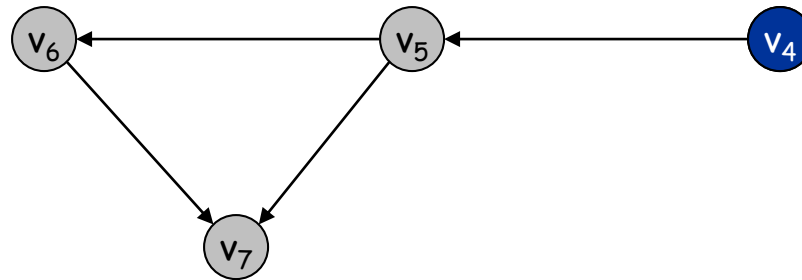
Topological order: v_1

Topological Ordering Algorithm: Example



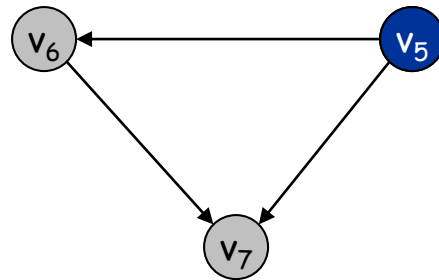
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



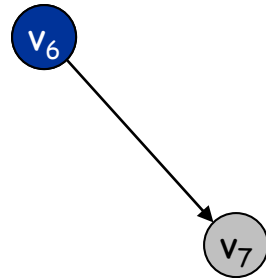
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example



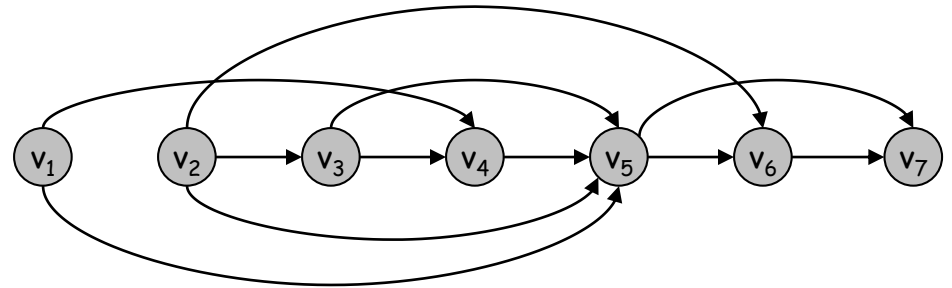
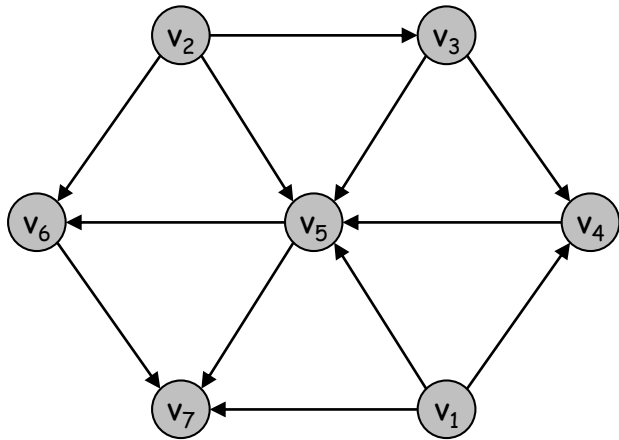
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Directed Acyclic Graphs

If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{v\}$ in topological order.
- This is valid since v has no incoming edges. ▪

Directed Acyclic Graphs

To compute a topological ordering of G : (already practiced on the example)

- Find a node v with no incoming edges and order it first
- Delete v from G
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after v

Seems like a Greedy Algorithm, however, nothing to optimize

- All we need to do is to choose an arbitrary node v with $\text{in-deg}(v) = 0$

How to efficiently maintain an updated degree for all current nodes in G ?

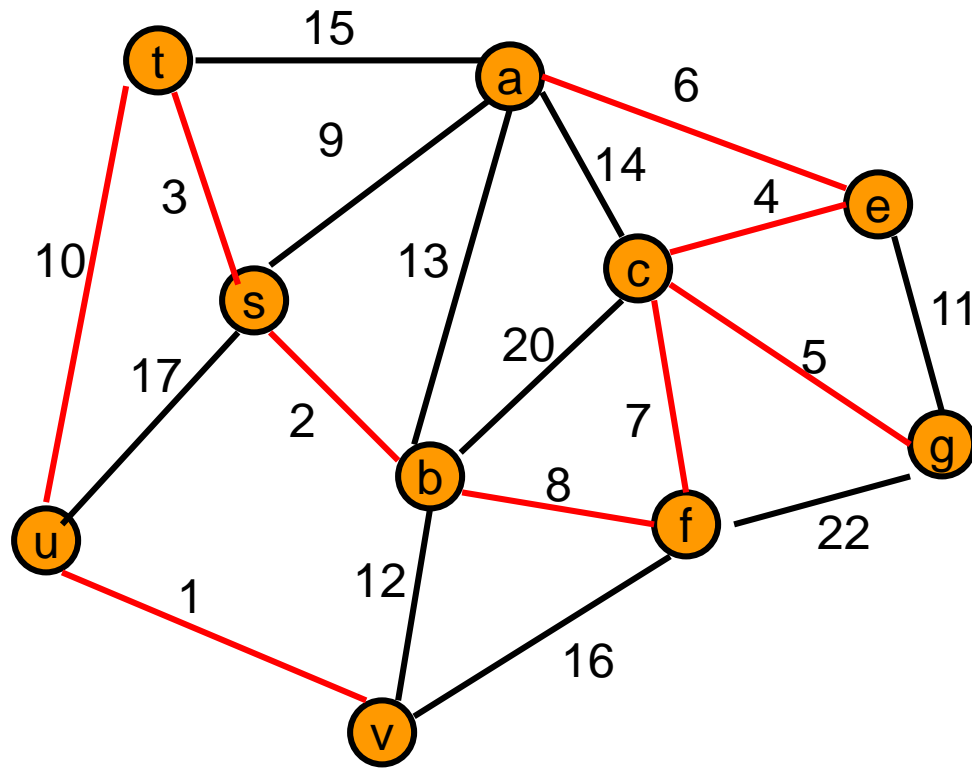
➤ Queueing & Counting

Topological Sorting Algorithm: Running Time

Maintain the following information:

- $\text{count}[v]$ = remaining number of incoming edges, $\text{in-degree}(v)$
- Q = queue of remaining nodes with no incoming edges, $\text{in-degree}(v)=0$
- Initialization: $O(m)$ via single scan through graph.
- Update: to delete u
 - remove u from Q
 - decrement $\text{count}[v]$ for all edges from u to v , and
 - add v to Q if $\text{count}[v]$ hits 0
 - this is $O(1)$ per edge ONCE
- Updating in-degrees costs $O(m)$ in total: G as Adjacency lists
 - The order of removal from Q gives the Topological Ordering

Minimum Spanning Trees

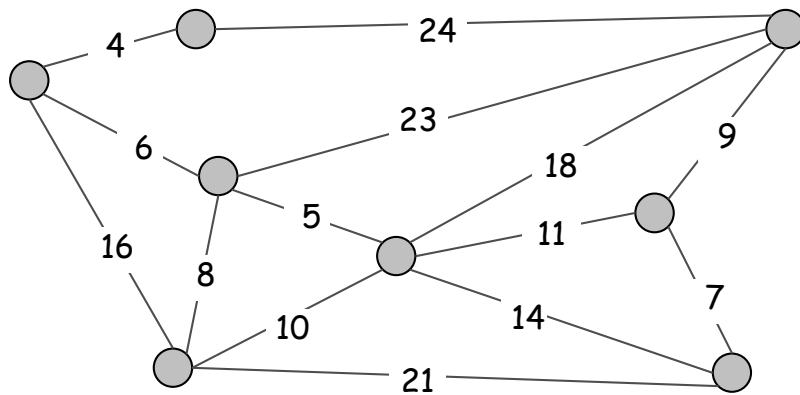


Minimum Spanning Tree (MST)

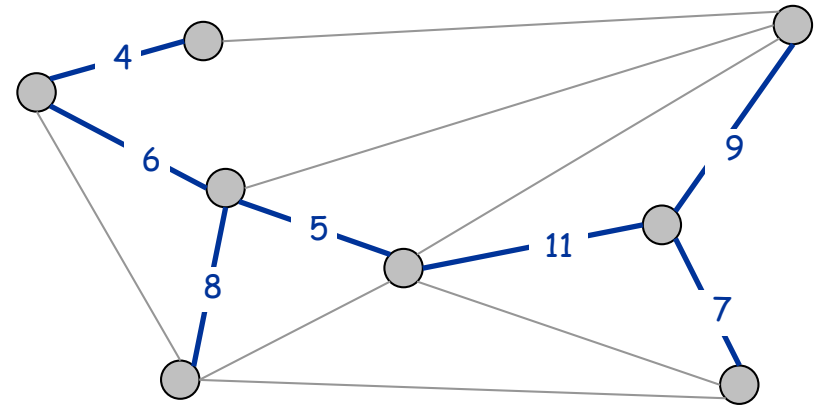
Given: a **connected undirected** graph $G = (V, E)$ where every edge has some **positive cost** c_e (also called weight)

Minimum spanning tree (MST): an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree (it "spans" all nodes in V) whose **sum of edge weights is minimized**.

Goal: Find an MST in G .



$G = (V, E)$



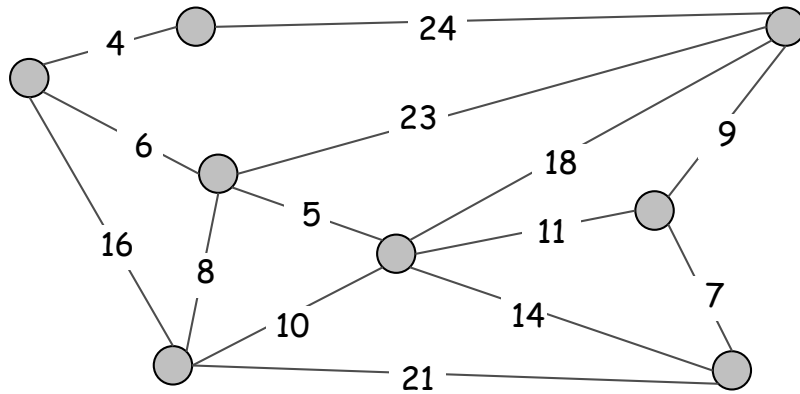
$T, \sum_{e \in T} c_e = 50$

Applications

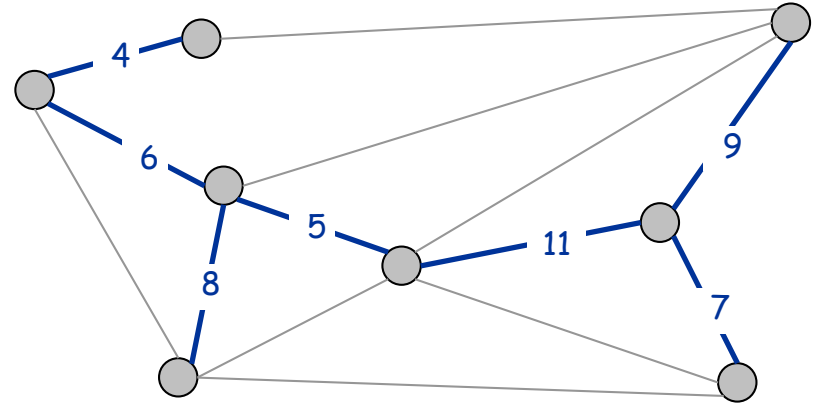
MST is fundamental problem with diverse applications.

- **Network design.**
 - telephone, electrical, hydraulic, TV cable, computer, road
- **Approximation algorithms for NP-hard problems.**
 - traveling salesperson problem, Steiner tree
- **Indirect applications.**
 - max bottleneck paths
 - LDPC codes for error correction
 - image registration with Renyi entropy
 - learning salient features for real-time face verification
 - reducing data storage in sequencing amino acids in a protein
 - model locality of particle interactions in turbulent fluid flows
 - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- **Cluster analysis.**

Minimum Spanning Tree (MST)



$G = (V, E)$



$T, \sum_{e \in T} c_e = 50$

Cayley's Theorem. There are n^{n-2} spanning trees in a complete graph of n nodes

can't solve by brute force

Problem Analysis:

Which edges should be chosen by an MST?

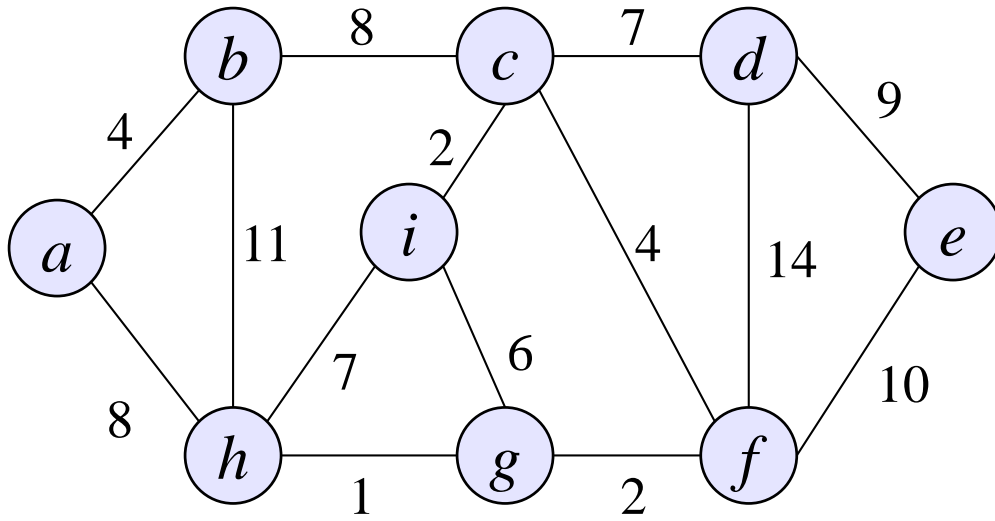
Intuitively: Those with min cost, to get an overall min.

Precisely: If e is a cheapest edge in G , then e belongs to some MST.

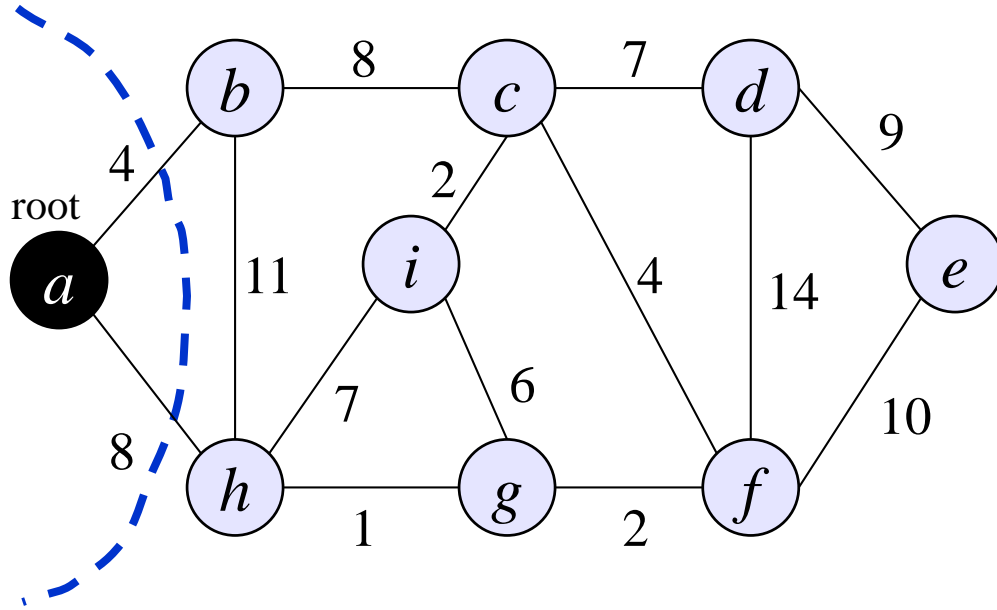
- Let T , an MST, does not include e .
- Add e to $T \Rightarrow T$ has a cycle C involving e .
- Remove some other edge from C , get a better T .

Greedy Algorithm?

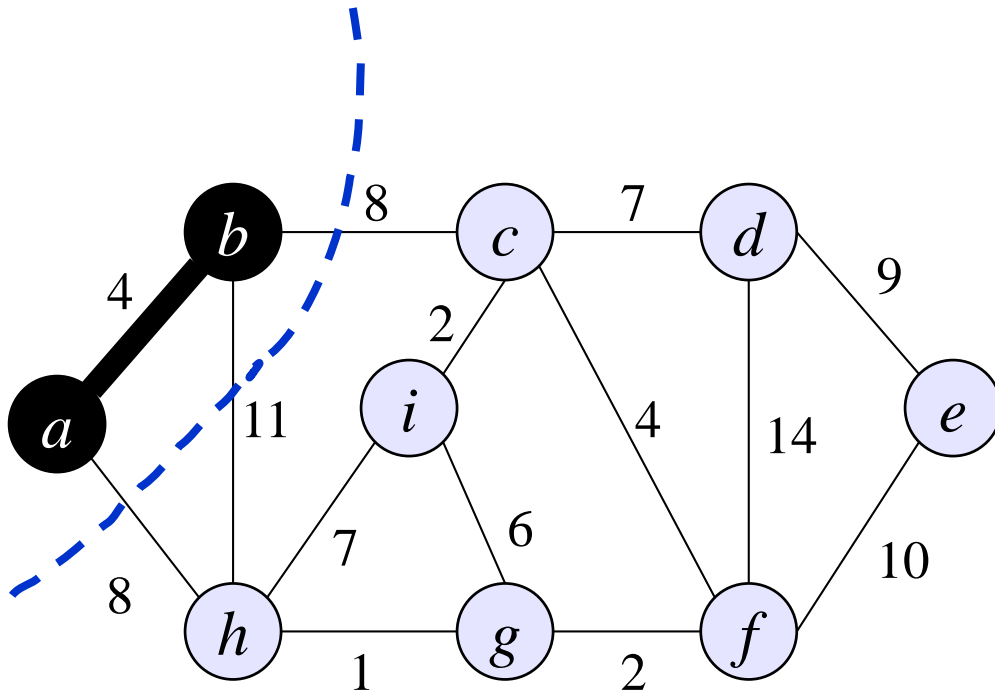
MST - Greedy Algorithms



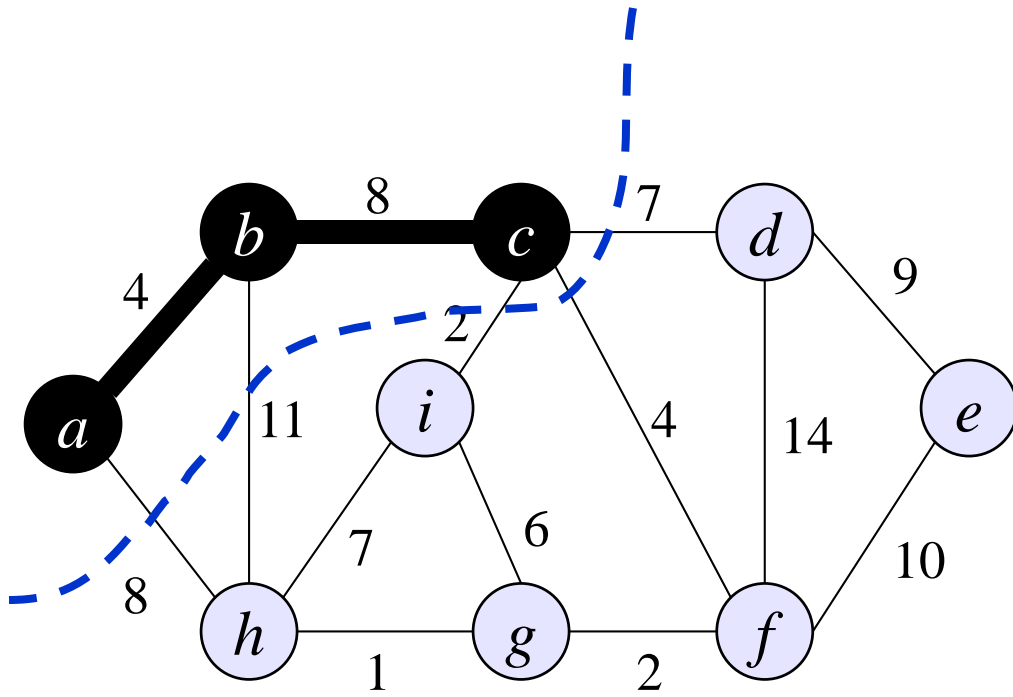
MST - Greedy Algorithms



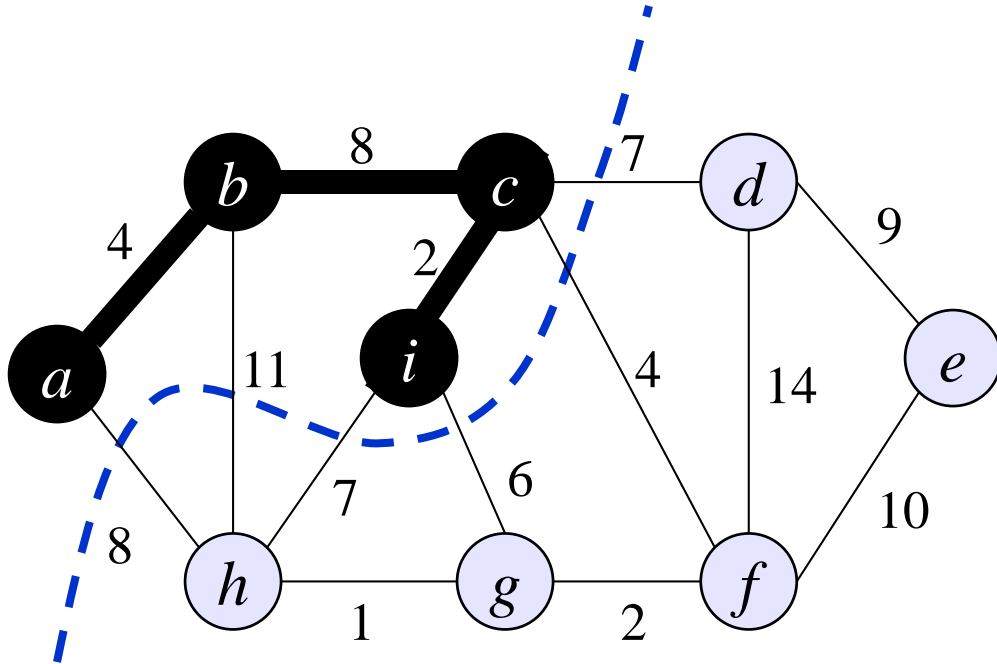
MST - Greedy Algorithms



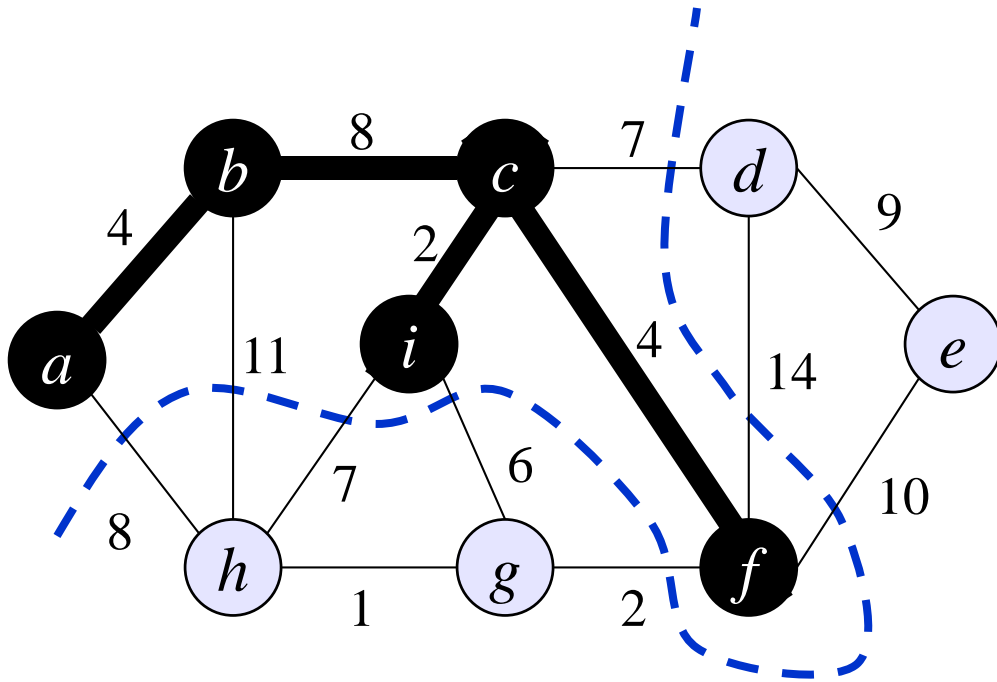
MST - Greedy Algorithms



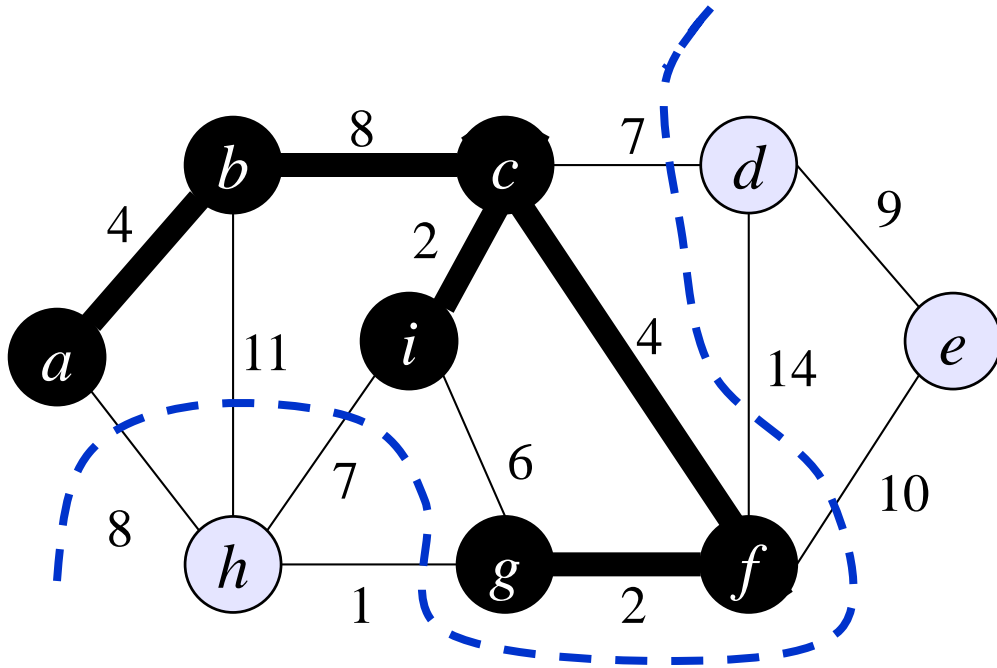
MST - Greedy Algorithms



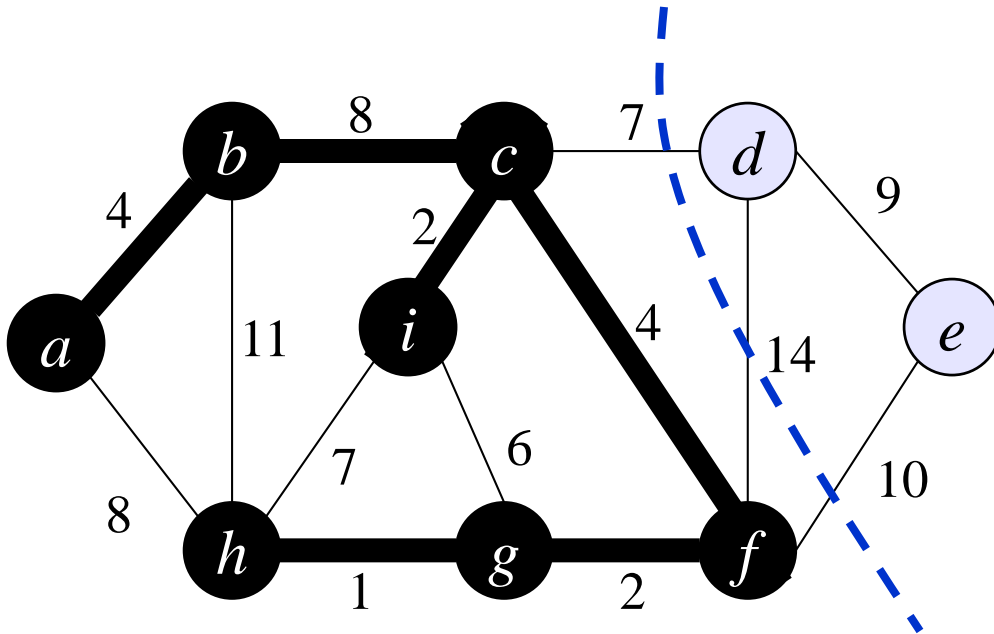
MST - Greedy Algorithms



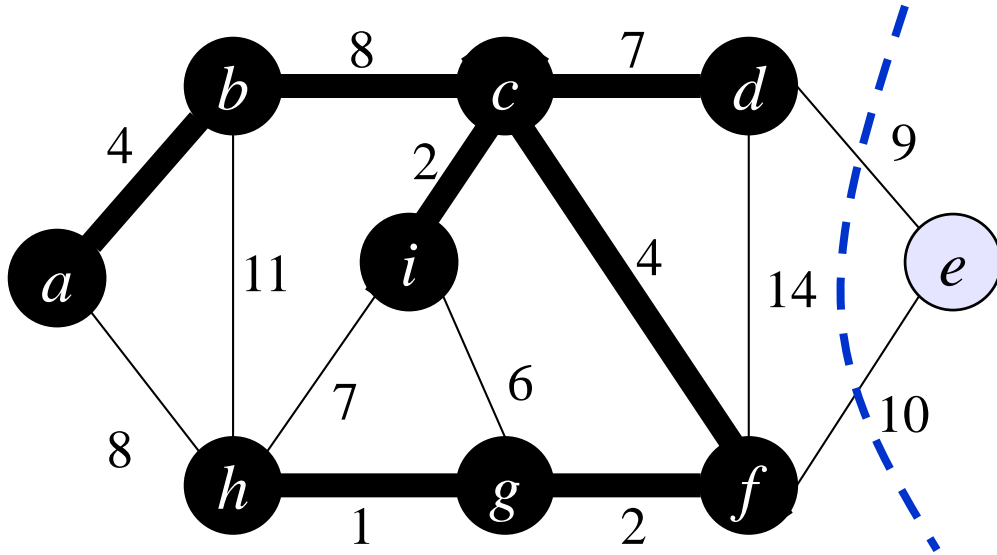
MST - Greedy Algorithms



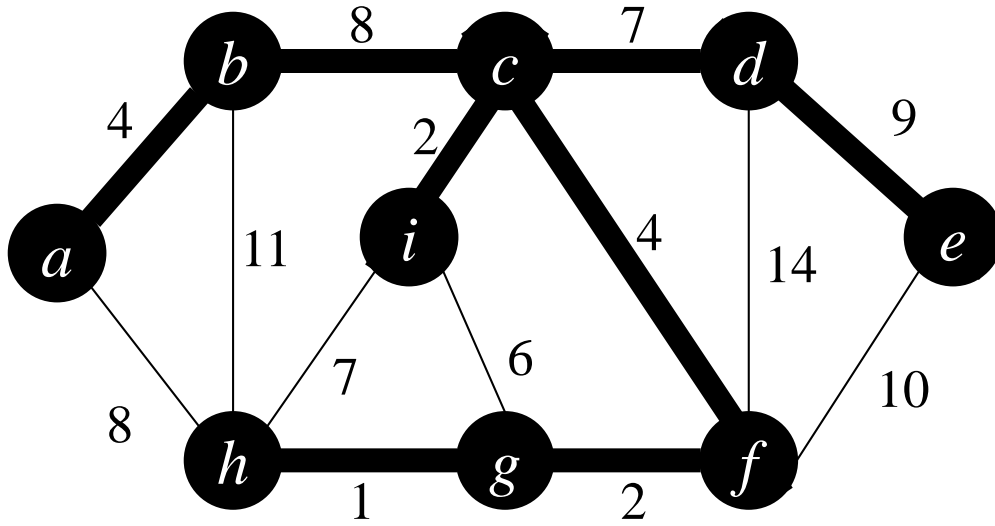
MST - Greedy Algorithms



MST - Greedy Algorithms



MST - Greedy Algorithms

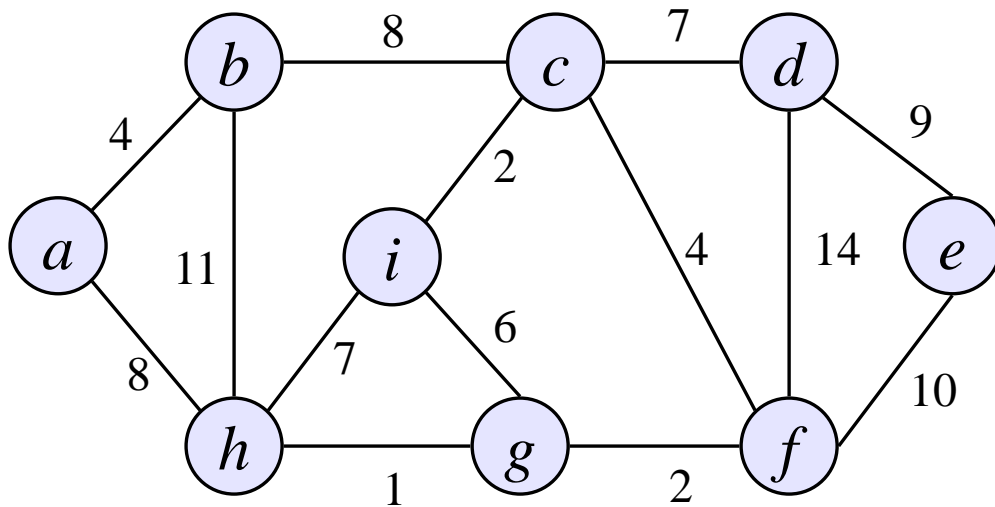


MST - Greedy Algorithms

The greedy rule that we followed in the demo on previous slides can be formulated as:

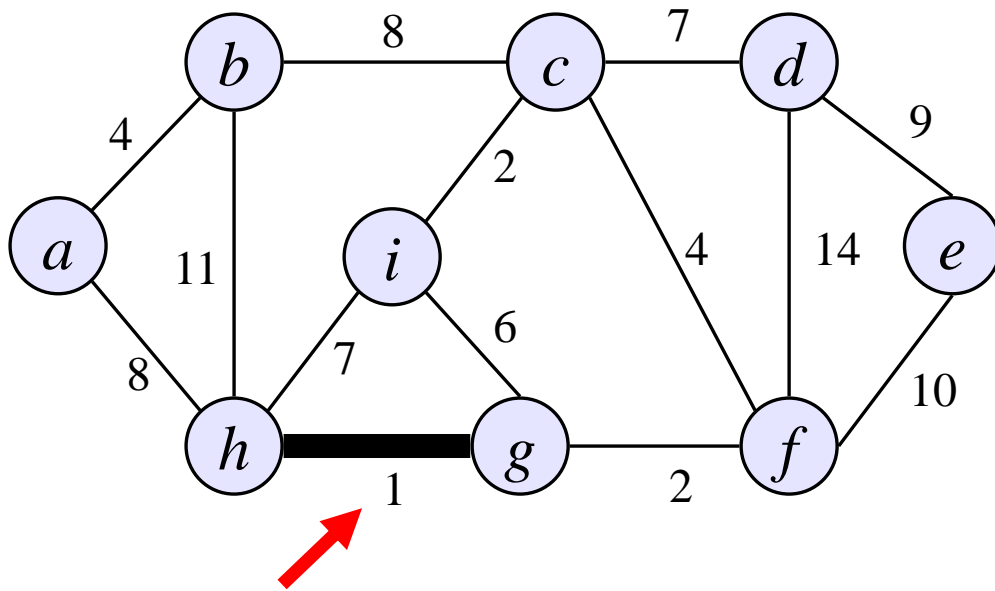
- **Start** at some root node **s** and **greedily grow** a tree **T** from **s** outward.
- At each step, add **the cheapest edge e** to **T** having **one endpoint in T**.
- Both edges in **T** would mean **T gets a cycle**.

Prim's Algorithm for finding an MST

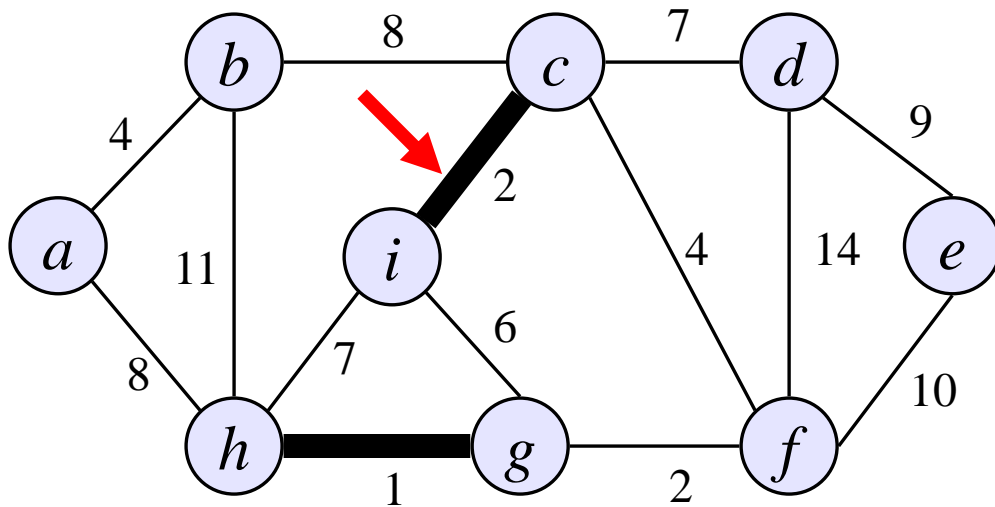


Initial sets = $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$

Edges	Weight
(g, h)	1
(c, i)	2
(f, g)	2
(a, b)	4
(c, f)	4
(g, i)	6
(c, d)	7
(h, i)	7
(a, h)	8
(b, c)	8
(d, e)	9
(e, f)	10
(b, h)	11
(d, f)	14

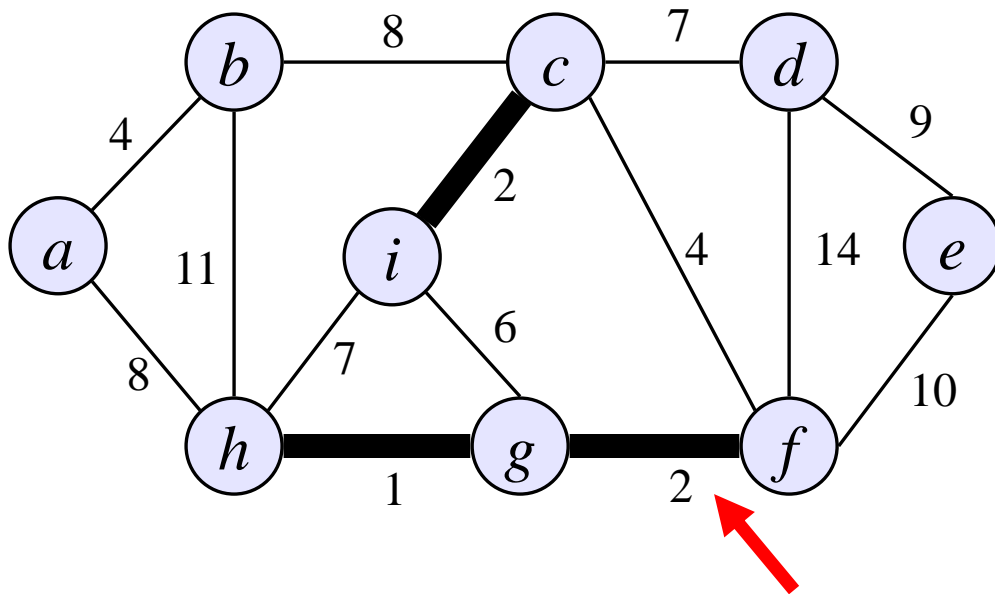


Initial sets = $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$
 Final sets = $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$



Initial sets = $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$

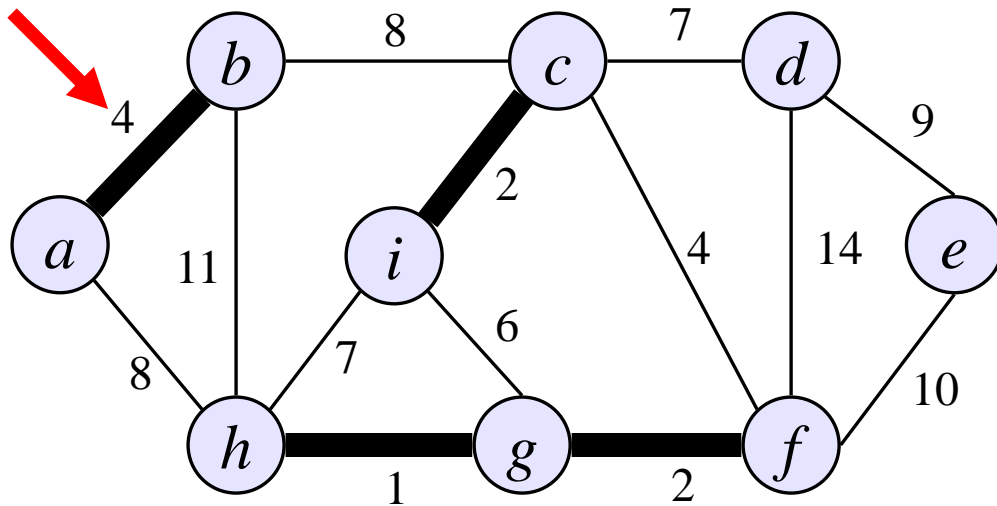
Final sets = $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$



Initial sets = $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$

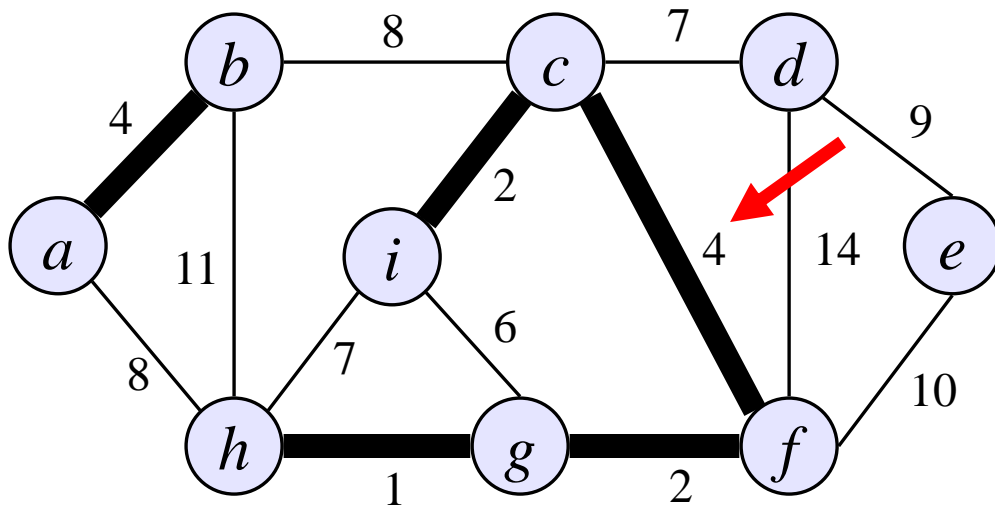
Final sets = $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

- In intermediate steps, we don't get a tree actually.
- A collection of connected components of edges, called a "Forest"



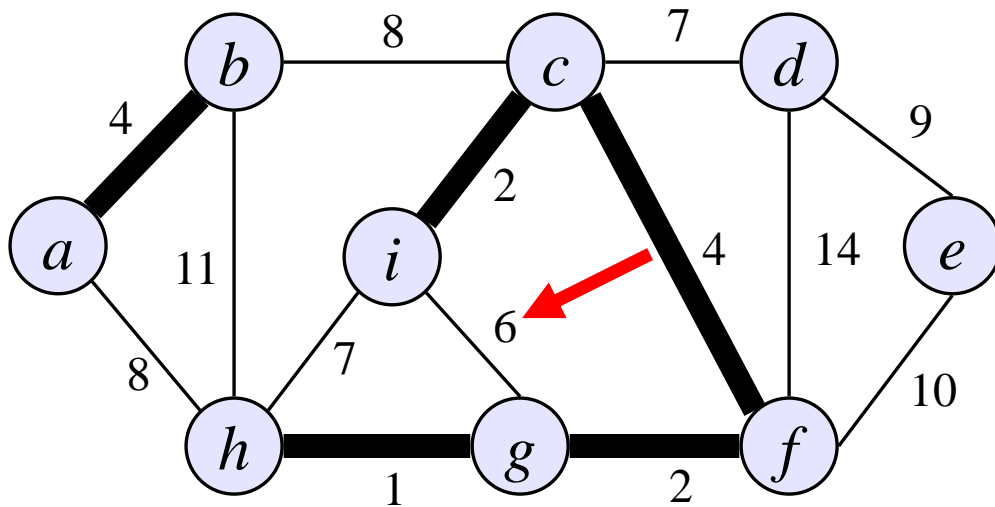
Initial sets = $\{\underline{a}\}, \{\underline{b}\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Final sets = $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$



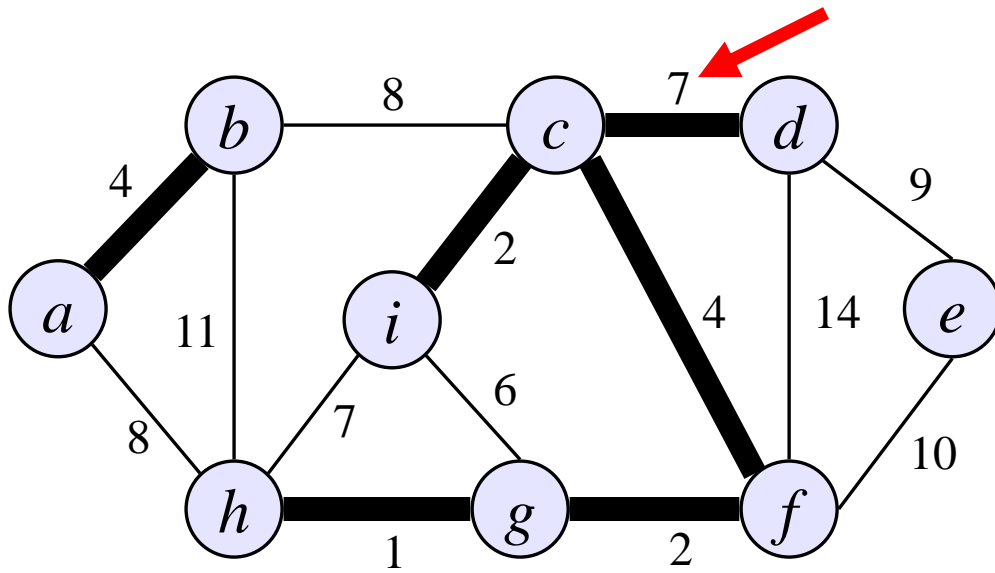
Initial sets = $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Final sets = $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$



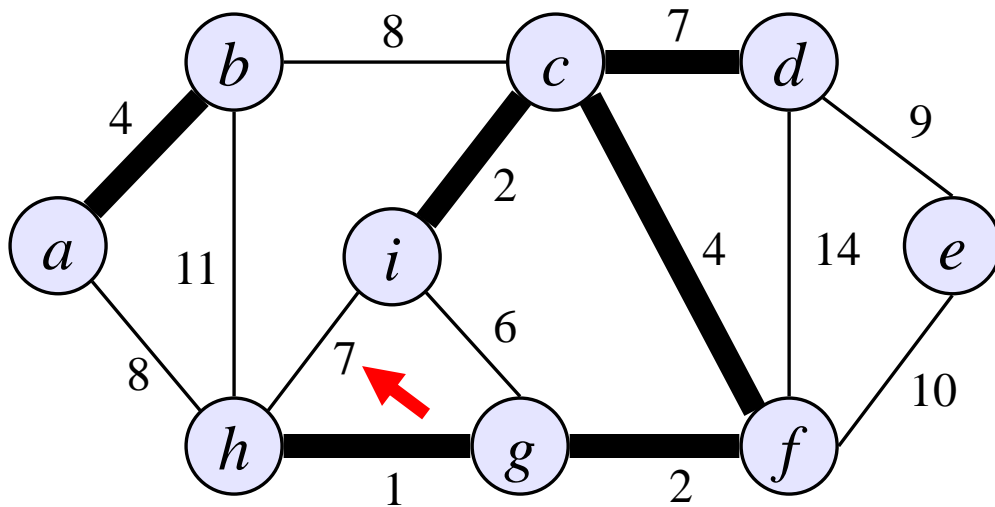
Initial sets = $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$

Final sets = $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$



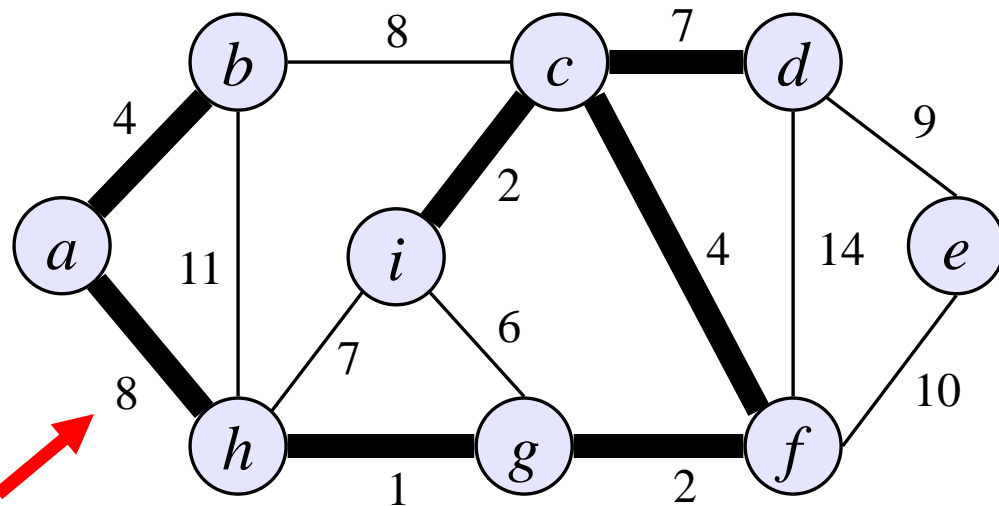
Initial sets = $\{a, b\}, \{\underline{c}, f, g, h, i\}, \{\underline{d}\}, \{e\}$

Final sets = $\{a, b\}, \{c, d, f, g, h, i\}, \{e\}$



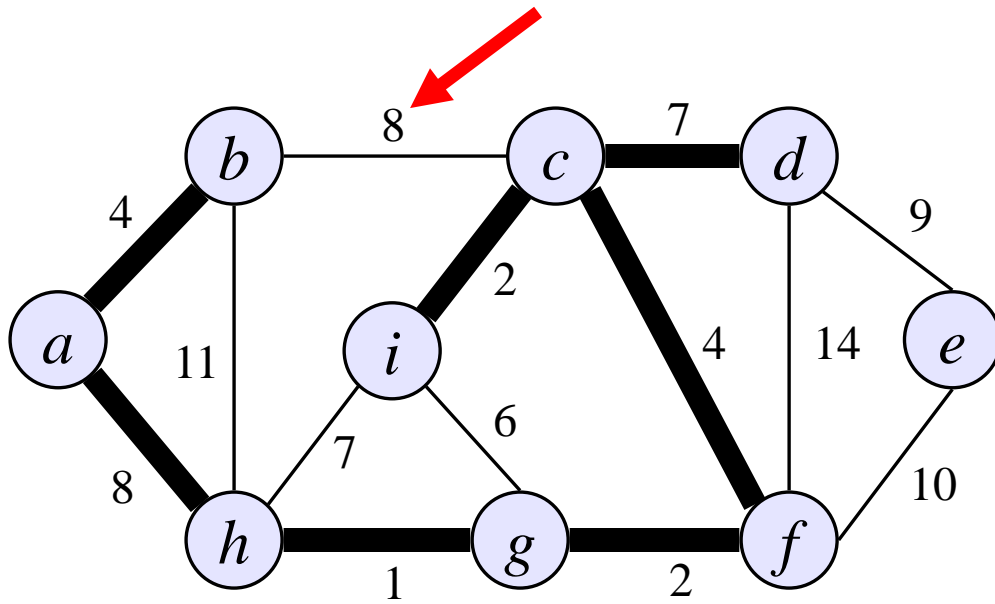
Initial sets = $\{a, b\}, \{c, d, f, g, \underline{h}, i\}, \{e\}$

Final sets = $\{a, b\}, \{c, f, d, g, h, i\}, \{e\}$



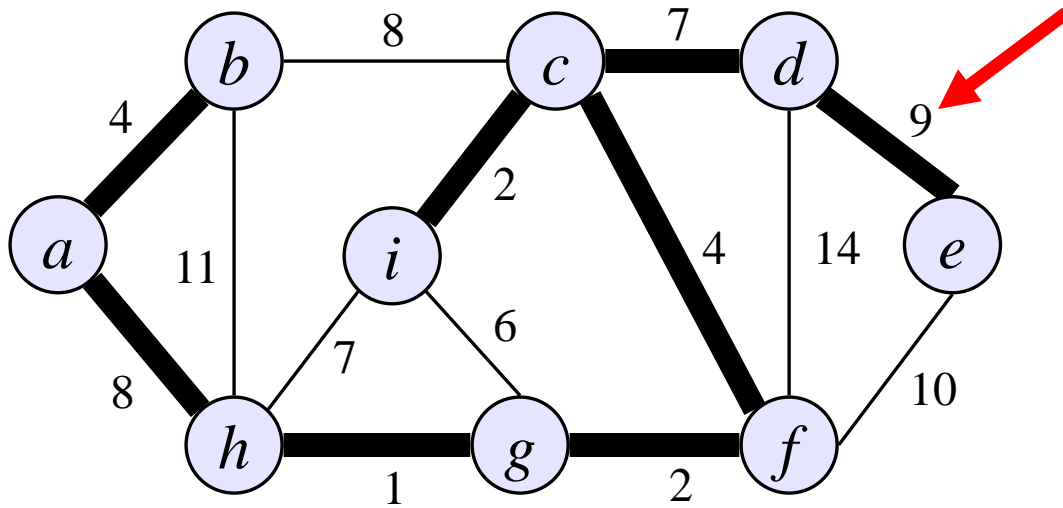
Initial sets = $\{\underline{a}, b\}, \{c, d, f, g, \underline{h}, i\}, \{e\}$

Final sets = $\{a, b, c, d, f, g, h, i\}, \{e\}$



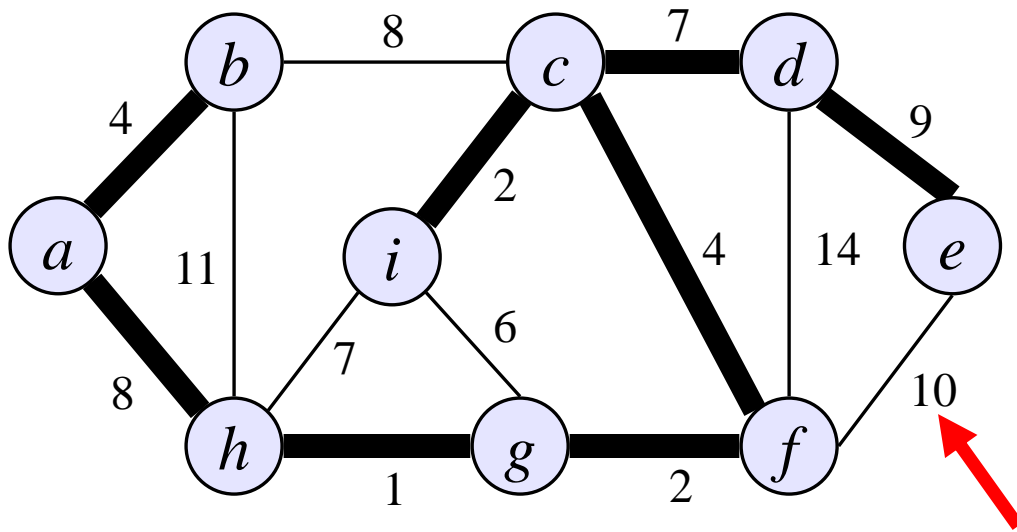
Initial sets = $\{a, \underline{b}, c, d, f, g, h, i\}, \{e\}$

Final sets = $\{a, b, c, d, f, g, h, i\}, \{e\}$



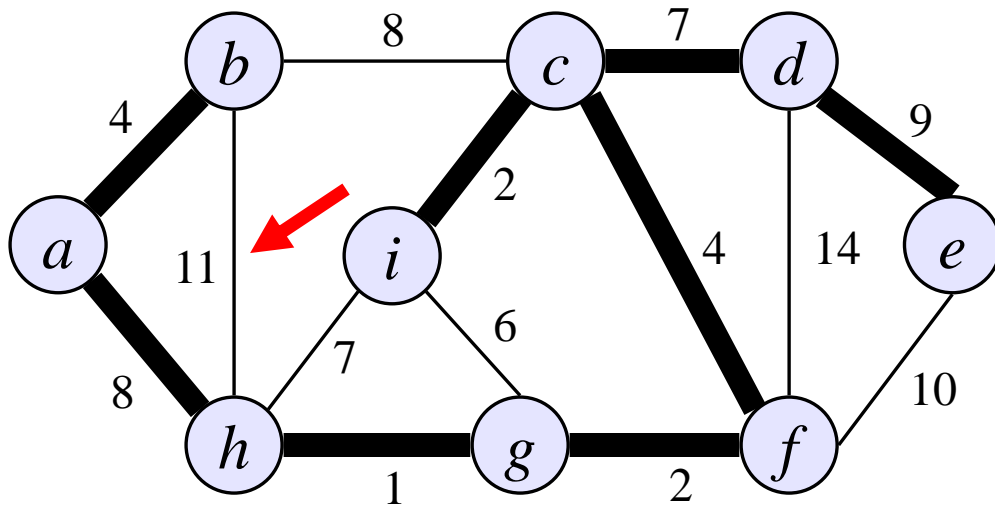
Initial sets = $\{a, b, c, \underline{d}, f, g, h, i\}, \{\underline{e}\}$

Final sets = $\{a, b, c, d, e, f, g, h, i\}$



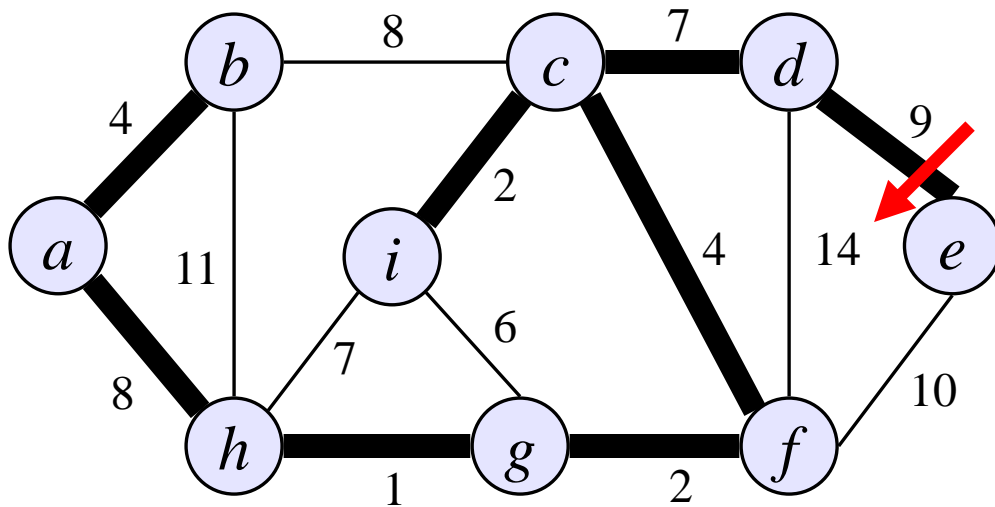
Initial sets = $\{a, b, c, d, \underline{e}, \underline{f}, g, h, i\}$

Final sets = $\{a, b, c, d, e, f, g, h, i\}$



Initial sets = $\{a, \underline{b}, c, d, e, f, g, \underline{h}, i\}$

Final sets = $\{a, b, c, d, e, f, g, h, i\}$



Initial sets = $\{a, b, c, \underline{d}, e, \underline{f}, g, h, i\}$

Final sets = $\{a, b, c, d, e, f, g, h, i\}$

MST - Greedy Algorithms

The previous demo follows what we call: **Kruskal's algorithm**.

- Start with $T = \phi$. Consider edges in ascending order of cost.
- **Insert edge e in T unless doing so would create a cycle.**

- **Like Prim's, this algorithm also produce an MST.**

MST - Greedy Algorithms

Correctness?

Our "the cheapest edge" argument does not hold for every step.

Let's extend it carefully

Simplification: All edge costs c_e are distinct.

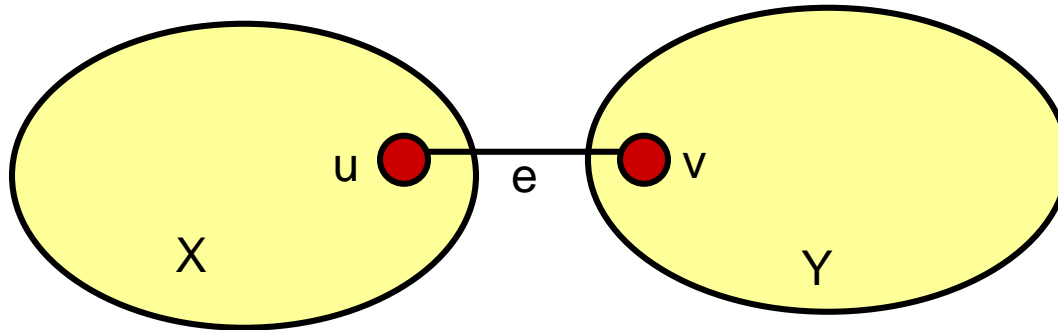
A generalized version of cheapest edge inclusion property of MSTs:

If we partition the node set V arbitrarily in two non-empty sets X and Y , then the cheapest edge among all edges connecting X and Y must belong to any MST.

MST - Greedy Algorithms

Edge inclusion lemma:

Let $X = S$ and $Y = V \setminus S$, and suppose $e = (u, v)$ is the minimum cost edge of E , between X and Y .



We want to show:

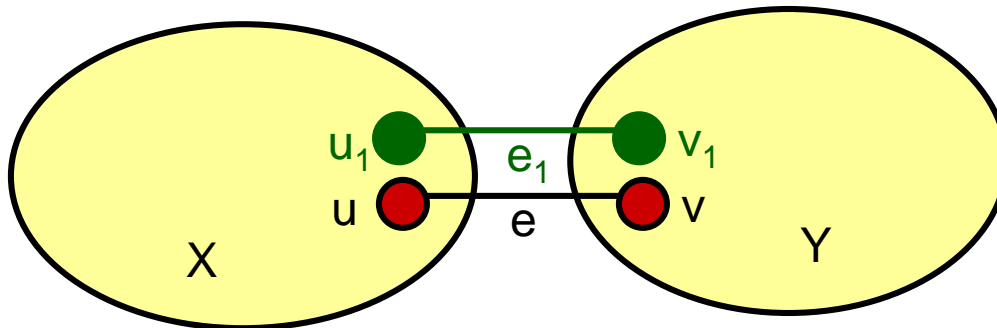
- e is in every minimum spanning tree of G
 - Or equivalently, if e is not in T , then T is not an MST.

Greedy Algorithms: Edge inclusion lemma

Edge inclusion lemma

Pf. Suppose T is an MST that does not contain e

- Add e to T , this creates a cycle
- The cycle must have some edge $e_1 = (u_1, v_1)$ with u_1 in X and v_1 in Y



$T^* = T \cup \{e\} - \{e_1\}$ is a spanning tree with **lower cost**

- by assumption, e is the minimum cost edge between X and Y

Hence, T is not a minimum spanning tree (a contradiction).

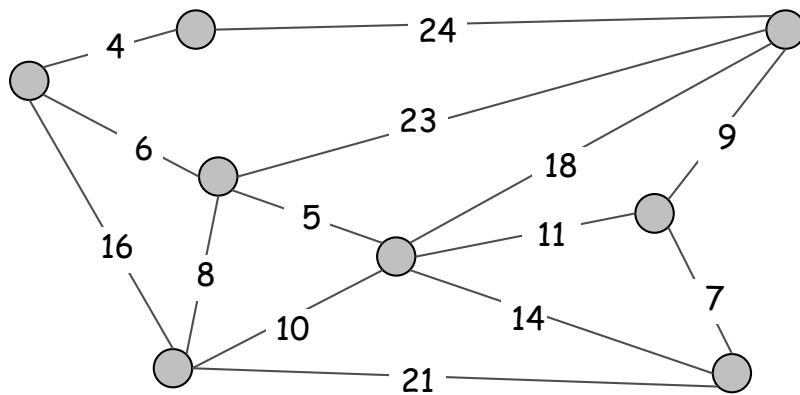
- In the presence of assumption (All edge costs c_e are distinct), above proof also says that: **the MST is uniquely determined.**

Optimality Proofs

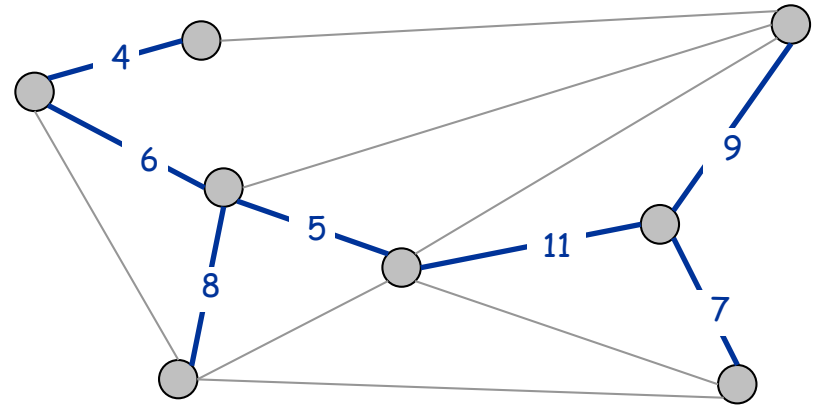
Prim's (Kruskal's) algorithm computes a Tree which is an MST

Show that:

- When an edge is added to the MST by Prim or Kruskal, the **edge** is the **minimum cost** edge between **S** and **V \ S** for some set S.
- **All selected edges together form a tree (NO Cycles).**
- **The tree spans the entire set V.**



$G = (V, E)$



$T, \sum_{e \in T} c_e = 50$

Optimality Proofs

Prim's Algorithm: Without losing any detail, we can write it as follows.

$S = \{\}; T = \{\};$

while $S \neq V$ ← This ensures all nodes are included in the tree

choose the minimum cost edge

$e = (u,v)$, with u in S , and v in $V \setminus S$ ← Ensures Cycle Free

add e to T

add v to S

endwhile

Edge inclusion lemma ensures every e chosen here actually belongs to an MST

Optimality Proofs

Kruskal's Algorithm:

Let $C = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$, $T = \{\}$

while $|C| > 1$

This ensures all nodes are included in the tree

Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C

Ensures Cycle Free

Replace C_i and C_j by $C_i \cup C_j$

Add e to T

endwhile

Edge inclusion lemma ensures every e chosen here actually belongs to an MST

Yet another Greedy Algorithm

Reverse-Delete Algorithm:

Delete the most expensive edge that does not disconnect the graph.

Relies on the following lemma:

- The most expensive edge on a cycle is never in a minimum spanning tree
- Correctness is proved along the same lines.

Remark: For dense graphs it is slower than the others, as it has to delete most edges.

Dealing with the assumption of no equal cost edges

Perturbation Argument:

Force the edge costs to be distinct to break the ties

- Add small distinct quantities to the edges having equal costs

Apply Prim's /Kruskal's algorithm

- the resulting edge set is an MST also for G with the original edge costs
 - the sum of perturbations should be small enough

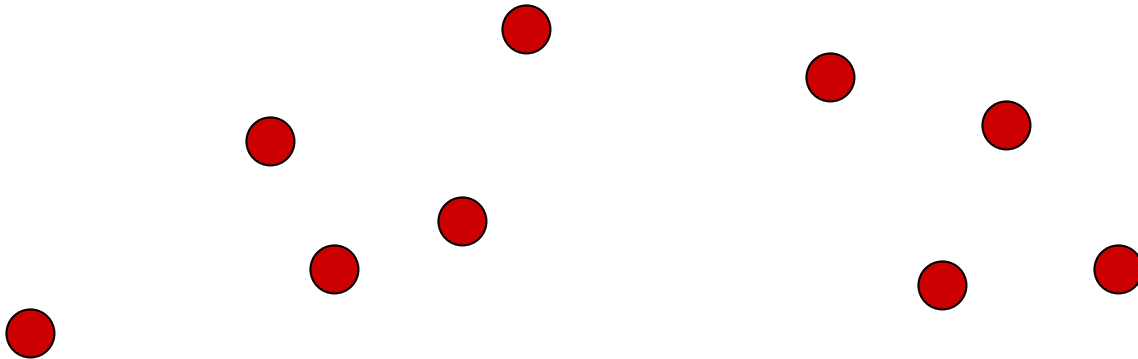
Application: Clustering

Clustering: a **partitioning** of a set of (data) points into **disjoint subsets of points**, called **clusters**.

How: Some **distance function** is defined between the points.

Spacing: the **minimum** distance of two clusters
(or equivalently, the minimum distance of **any two points** from different clusters).

Given : a collection of **n points in a geometrical space**, and an integer **$k < n$** .
The pairwise distances of points are known, or they can be easily computed.



Goal: Construct a clustering with **k clusters** and **maximum spacing**.

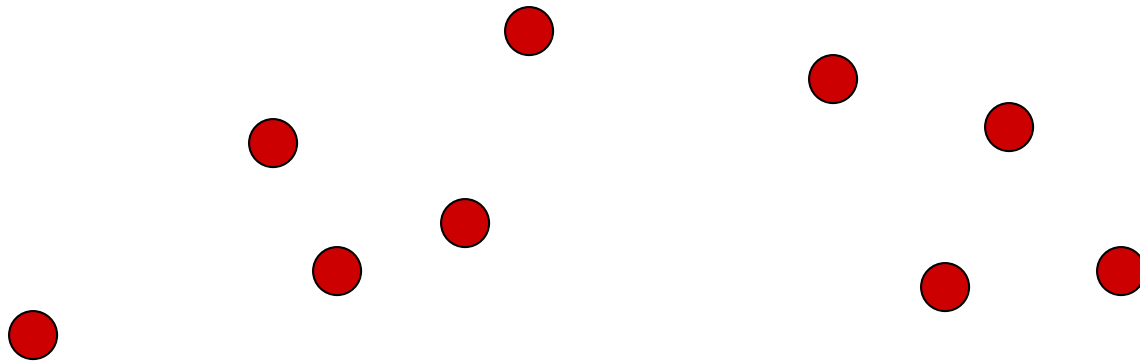
Distance clustering

Divide the data set into 2 subsets and maximum spacing

- $\text{dist}(S_1, S_2) = \min \{ \text{dist}(x, y) \mid x \text{ in } S_1, y \text{ in } S_2 \}$

Motivations:

Clustering in general has many applications in data reduction, pattern recognition, classification, data mining, and related fields.



Distance Clustering Algorithm

Kruskal's Algorithm:

Let $C = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$; $T = \{\}$

while $|C| > K$

 Let $e = (u, v)$ with u in C_i and v in C_j

 be the minimum cost edge joining distinct sets in C

 Replace C_i and C_j by $C_i \cup C_j$

endwhile

The K nodes sets so obtained are the K clusters.

To prove: **The obtained spacing is optimal.**

The proof follows from the fact that Kruskal's algorithm always joins two disjoint subsets (into a single cluster) via a min cost edge having endpoints in both.

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = {}; T = {};
```

```
while S != V
```

```
    choose the minimum cost edge
```

```
        e = (u,v), with u in S, and v in V \ S
```

```
    add e to T
```

```
    add v to S
```

```
endwhile
```

How we implement this step, is going to decide the running time.

Naïve implementation

- (n-1) iterations with $O(m)$ every time to choose min cost $e \Rightarrow O(nm)$

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

$S = \{\}; T = \{\};$

while $S \neq V$

 choose the minimum cost edge

$e = (u,v)$, with u in S , and v in $V \setminus S$

 add e to T

 add v to S

endwhile

How we implement this step, is going to decide the running time.

Improving the Naïve:

- For each v in $V \setminus S$, maintain attachment cost
 $a[v] = \text{cost of cheapest edge } e = (u,v), \text{ with } u \text{ in } S$
- Choosing the min cost edge is restricted to the n edges represented by a .
When "add v to S " happens, update array a as:
 - foreach (edge $e = (v, u)$ incident to v)
 - if $((u \notin S) \text{ and } (c_e < a[u]))$ $\Rightarrow O(n)$
 - decrease cost of $a[u]$ to c_e

Inside the while: $O(n + n) = O(n)$

Altogether: $(n-1)$ iterations of while with $O(n)$ every time $\Rightarrow O(n^2)$

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

$S = \{\}; T = \{\};$

while $S \neq V$

 choose the minimum cost edge

$e = (u,v)$, with u in S , and v in $V \setminus S$

 add e to T

 add v to S

endwhile

How we implement this step, is going to decide the running time.

Choosing a better data structure:

- Let F be the set of edges between S , and $V \setminus S$.
 - Choose the **minimum cost e** from F
- when "add v to S " happens, we update F :
 - every e enters and leaves F exactly once $\Rightarrow O(m)$
 - If F is a **priority Queue**, insert/remove(find min) is achievable using **$O(\log m)$**
 - Altogether $\Rightarrow O(m \log m) \Rightarrow O(m \log n)$
 - $m \leq n^2 \Rightarrow \log m$ is $O(\log n)$

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Prim's:

```
S = { }; T = { };  
while S != V  
    choose the minimum cost edge  
        e = (u,v), with u in S, and v in V\S  
    add e to T  
    add v to S  
endwhile
```

Both implementations of Prim's algorithm are justified:

- $O(n^2)$ is somewhat faster if the graph is dense (has a quadratic number of edges),
- but otherwise, $O(m \log n)$ is considerably faster.

Implementation

Which one is faster Prim's Algorithm or Kruskal's Algorithms?

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$, $T = \{\}$

while $|C| > 1$

Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C

Replace C_i and C_j by $C_i \cup C_j$

Add e to T

endwhile

The running time depends on the implementation of these steps.

We need efficient way to find a "global" min cost edge:

- sort the edges wrt cost $\Rightarrow O(m \log m) \Rightarrow O(m \log n)$
- if current e creates a cycle, move to the next in the sorted list.

Avoiding cycles efficiently depends how we:

- Make sure for $e = (u, v)$, u and v belong to disjoint sets
- Maintain C after every "Replace C_i and C_j by $C_i \cup C_j$ "
- A new data structure: Union-and-Find

Union-and-Find

- Maintains partitions of a set (here: V) into subsets
- Each subset has a label

Supports the following operations:

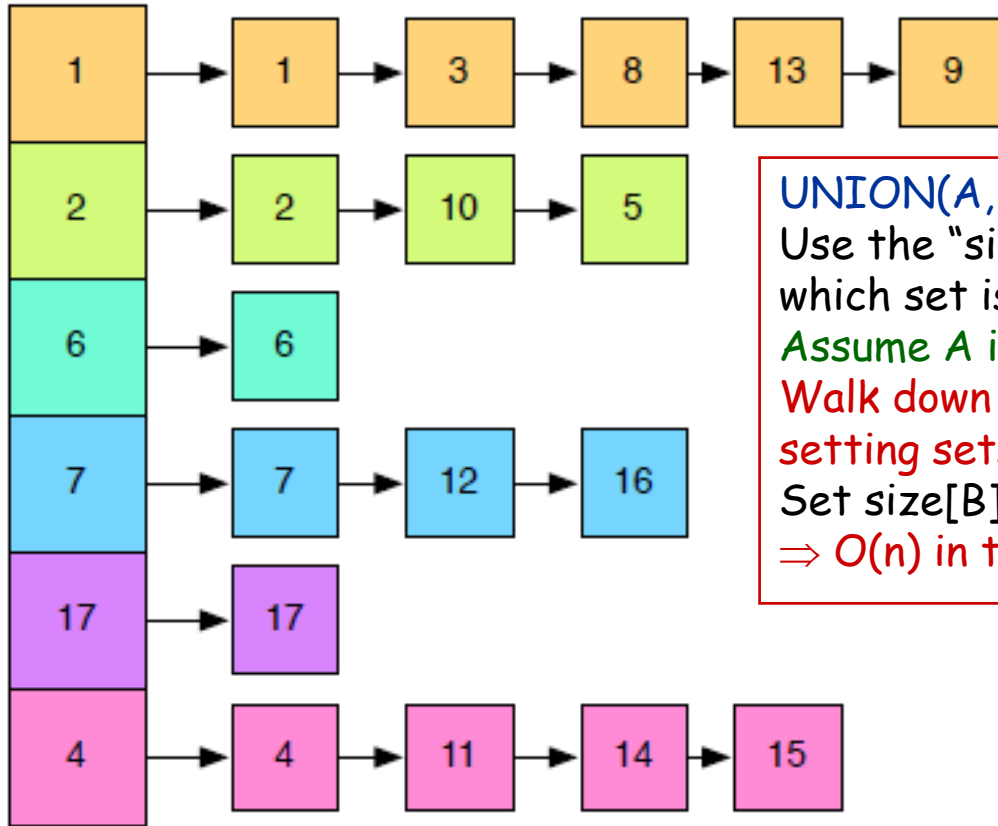
FIND(i): returns the label of the subset that contains element i .

UNION(A, B): merge the subsets with labels A and B , that is,
replace these sets with $A \cup B$ and give it a label.

UF Items:

Union-and-Find

UF Sizes:

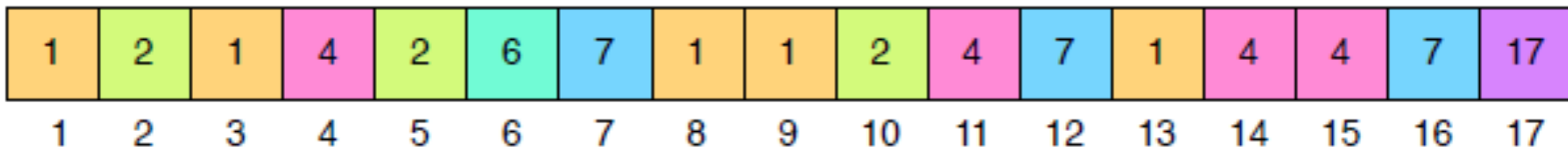


UNION(A,B):
 Use the "size" array to decide which set is smaller.
 Assume A is smaller.
 Walk down elements i in set A, setting $sets[i] = B$.
 Set $size[B] = size[B] + size[A]$
 $\Rightarrow O(n)$ in the worst case

1	5
2	3
6	1
7	3
17	1
4	4

FIND(i): return UF.sets[i].
 ➤ Takes a constant amount of time. $\Rightarrow O(1)$

UF Sets Array:



Union-and-Find

FIND(i): $O(1)$

UNION(A,B): $O(n)$ in the worst case

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$, $T = \{\}$
while $|C| > 1$

Let $e = (u, v)$ with u in C_i and v in C_j be the
minimum cost edge joining distinct sets in C

Replace C_i and C_j by $C_i \cup C_j$
Add e to T

endwhile

2 FIND calls per e
 $\Rightarrow O(m)$

$(n-1)$ UNION calls
 $\Rightarrow O(mn)$
using $O(n)$ per call

- Overall: $O(mn)$ dominates the sorting cost
- Why considering $O(n)$ which is worst case for ONE call to UNION?
- Can we do better by bounding the total for all UNION operations?
 - What is the time for $k < n$ call to UNION?

Union-and-Find

Rethinking "Time" for UNION(A,B):

Elementary operation:

- how often every element is relabeled and moved to other sets.

What invokes relabeling of an element?

- if it belongs to the smaller set in a UNION operation.

What happens then?

- the element now belongs to a new set of at least double the size.

What is the size of the larger set after k UNION operations?

- All sets are of size 1. After first UNION, the largest is 2
- After the second UNION, the largest can have at most 3 elements
- ..., after k UNION operation, the largest can have at most k + 1 size.

Every "elementary operation" moves the element from a set of size x to 2x

- every element is relabeled at most $\log_2(k+1)$ times.
- For all elements $\Rightarrow O(k \log k)$

Union-and-Find

➤ **Sorting:** $O(m \log m) \Rightarrow O(m \log n)$

$m \leq n^2 \Rightarrow \log m$ is $O(\log n)$

Kruskal's:

Let $C = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$, $T = \{\}$
while $|C| > 1$

Let $e = (u, v)$ with u in C_i and v in C_j be the
minimum cost edge joining distinct sets in C

Replace C_i and C_j by $C_i \cup C_j$
Add e to T

endwhile

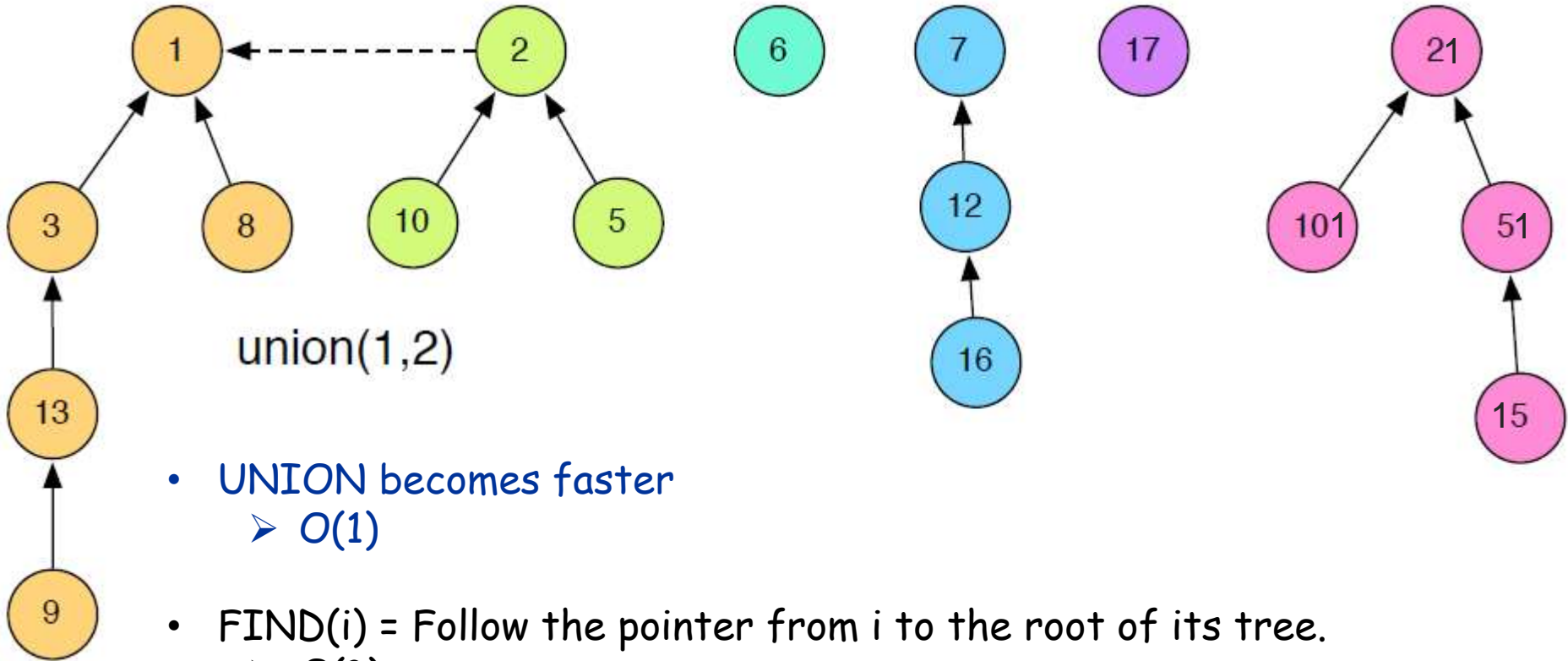
2 FIND calls per e
 $\Rightarrow O(m)$

$(n-1)$ UNION calls
 $\Rightarrow O(n \log n)$

• **Sorting dominates** $\Rightarrow O(m \log n)$

Union-and-Find

- Another Implementation of Union-and-Find
 - Simpler and Faster in practice



- UNION becomes faster
 - $O(1)$
- FIND(i) = Follow the pointer from i to the root of its tree.
 - $O(?)$

Union-and-Find

FIND(i) requires $O(\log n)$ in a tree-based union-and-find data structure containing n items.

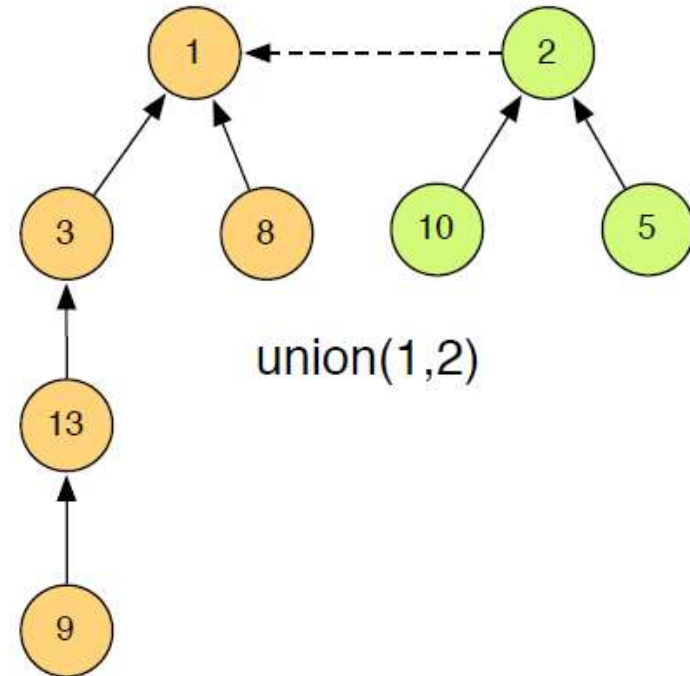
Pf.

What is the depth of an element v in its tree?

- the number of times the set containing it is renamed.

How many times a set can change its name?

- the set containing v at least doubles every time when its name is changed.



The largest number of times the size can double is $\log_2 n$.

Kruskal's Algorithm:

Sorting $O(m \log n)$, FIND: $O(m \log n)$, UNION: $O(n)$

- Same running time as using the array-based union-and-find
- Simpler and Faster in practice

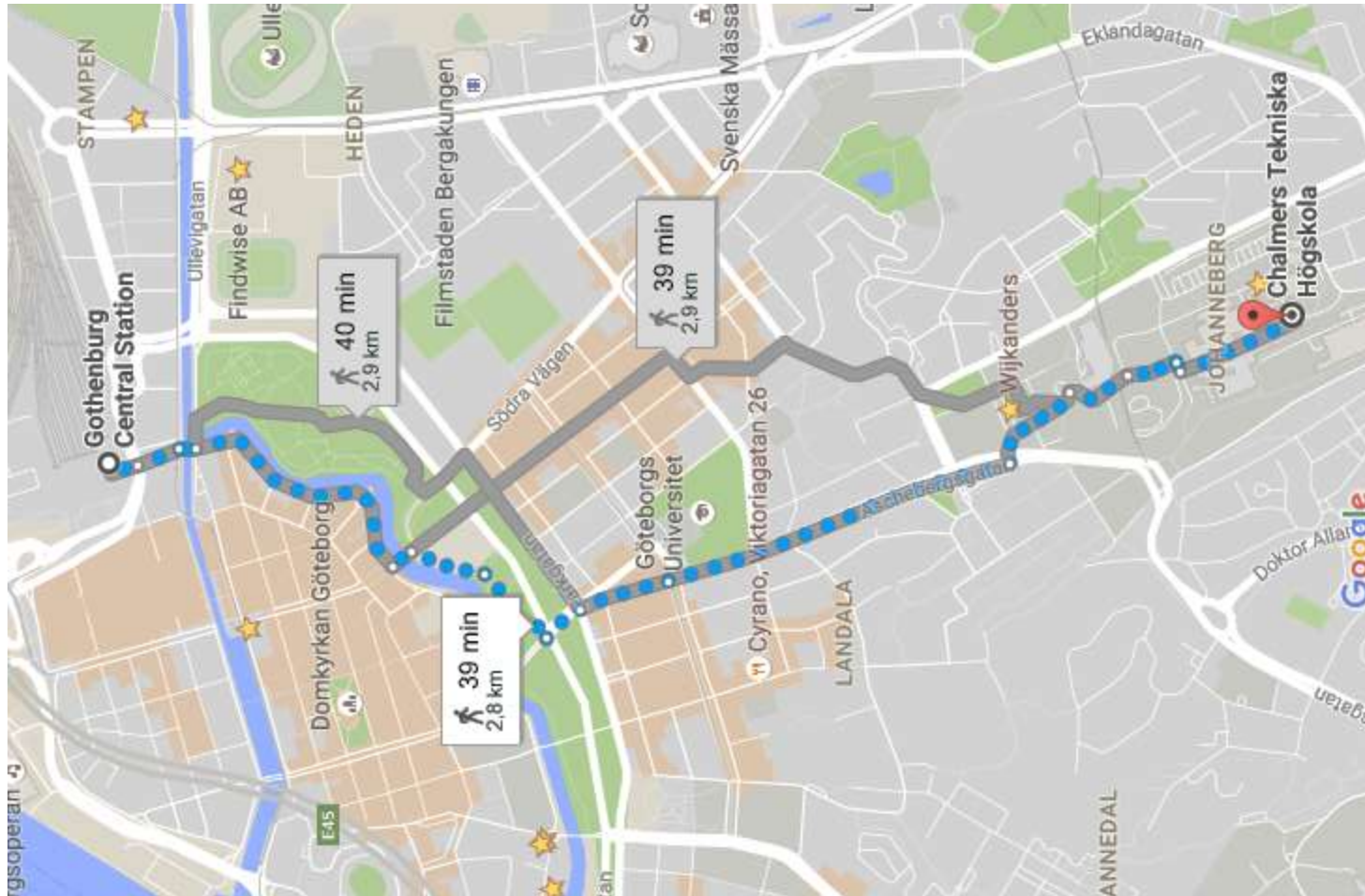
Algorithms: Lecture 12

Chalmers University of Technology

Today's Topics

- Shortest Paths
- Network Flow Algorithms

Shortest Path in a Graph



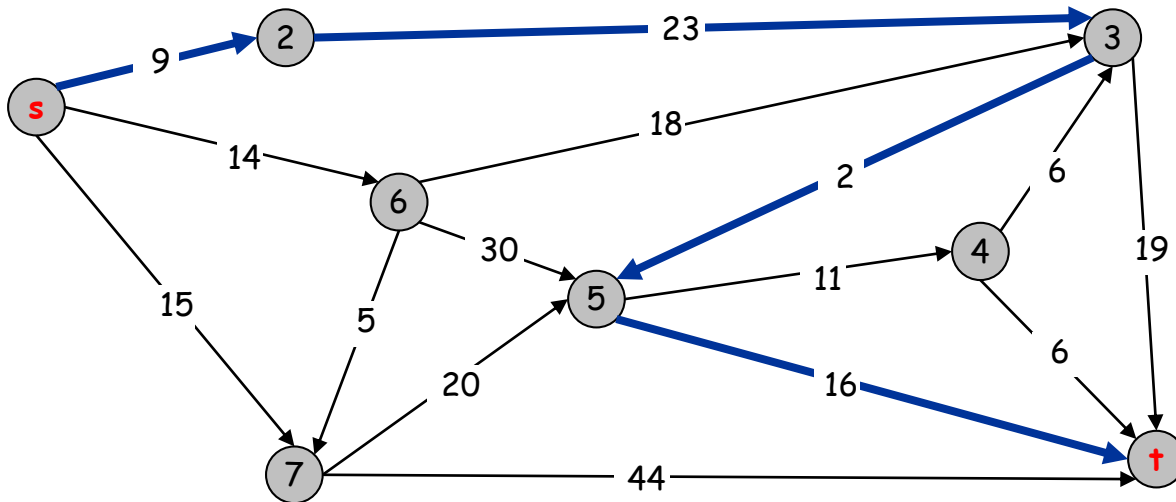
Shortest Path Problem

Shortest path network.

- **Directed** graph $G = (V, E)$.
- Source **s**, destination **t**.
- Length l_e = length of edge e , **non-negative**.

Single Source Shortest path problem: find shortest directed path from s to t.

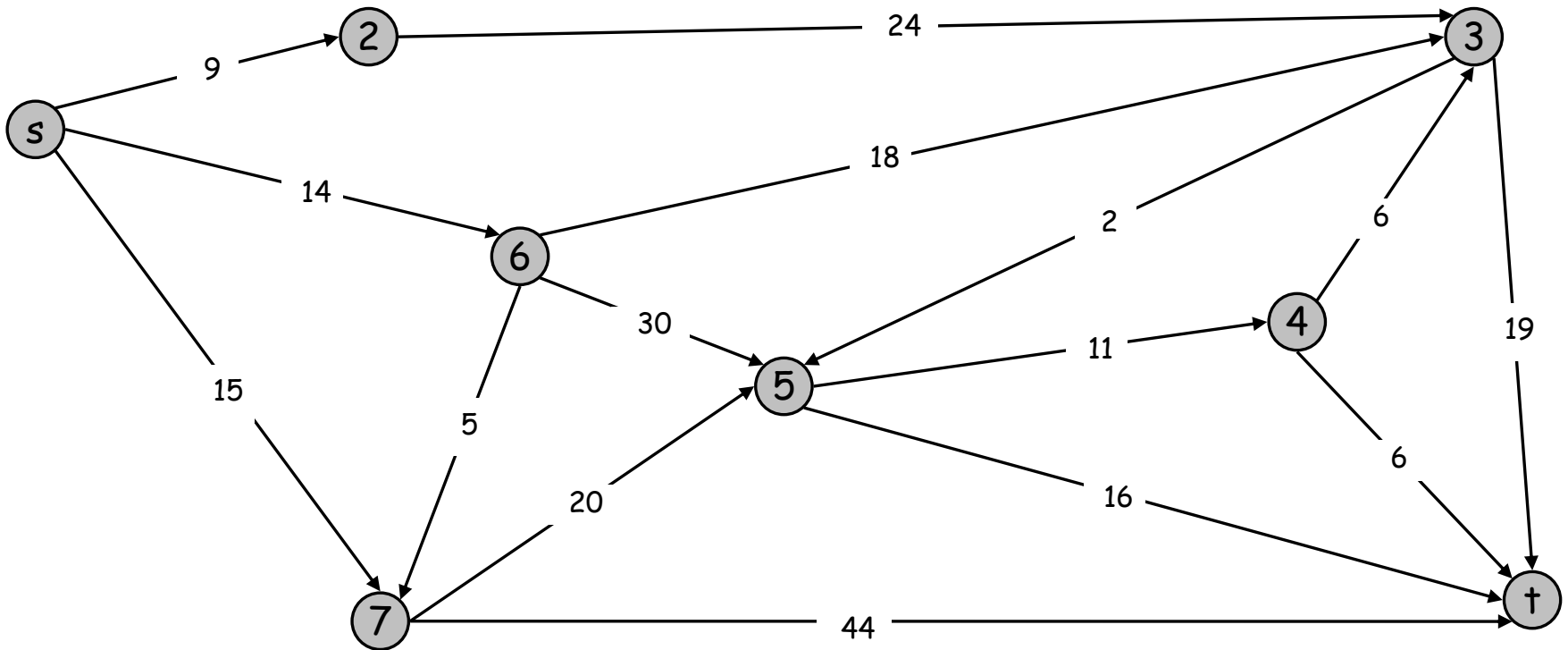
Length of path = sum of edge lengths in path



Length of path s-2-3-5-t:
= $9 + 23 + 2 + 16$
= 48.

Dijkstra's Shortest Path Algorithm

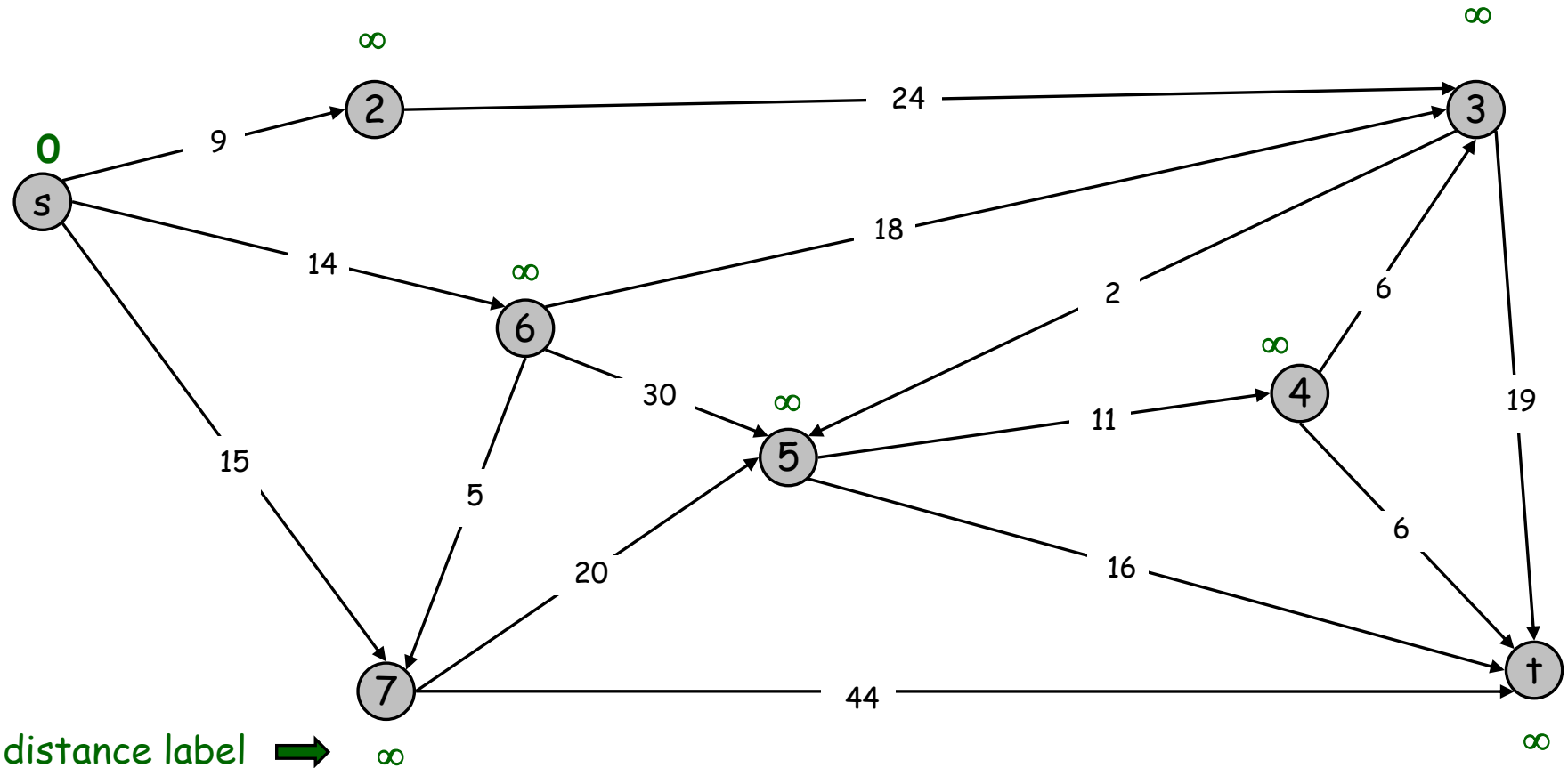
Find shortest path from s to t.



Dijkstra's Shortest Path Algorithm

$S = \{ \}$

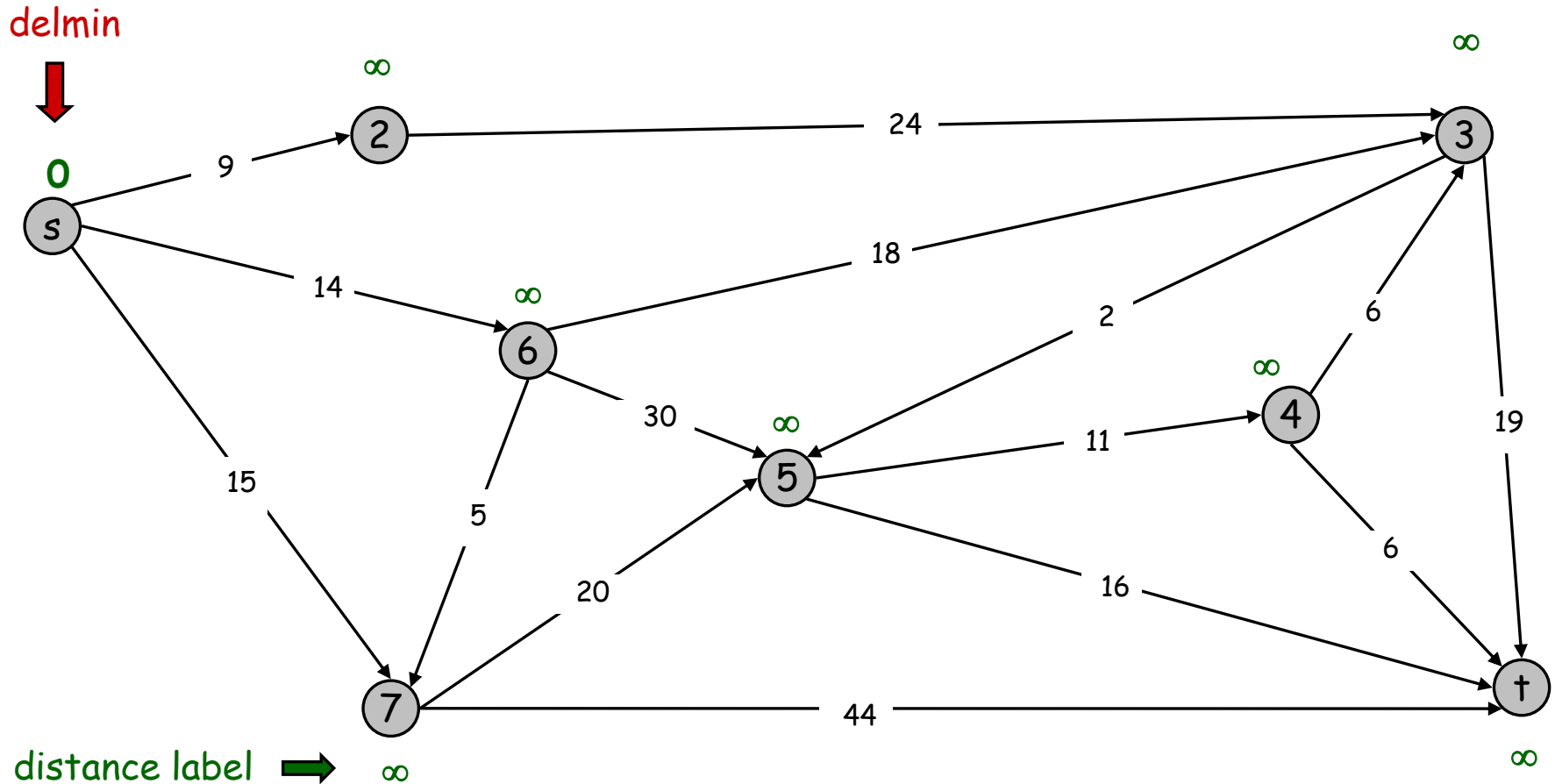
$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

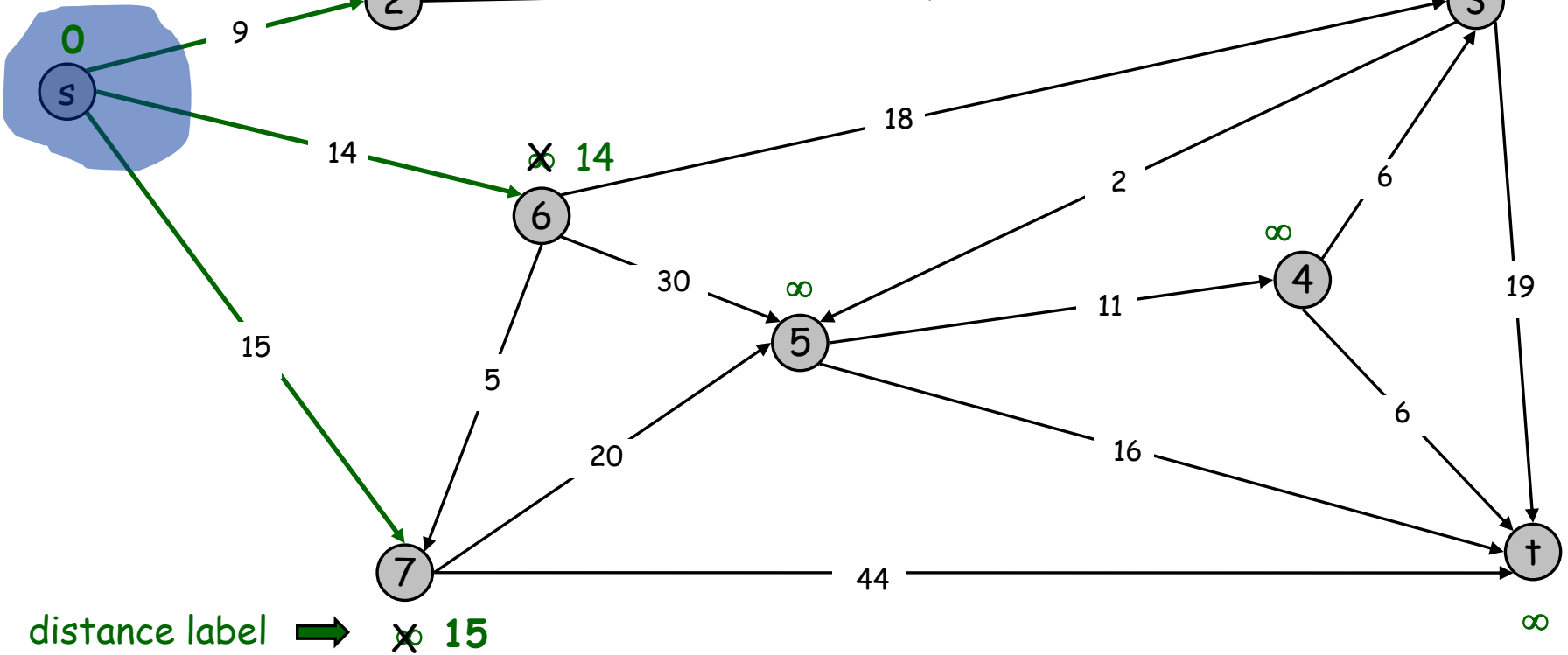
$S = \{s\}$

$PQ = \{2, 3, 4, 5, 6, 7, t\}$

decrease key



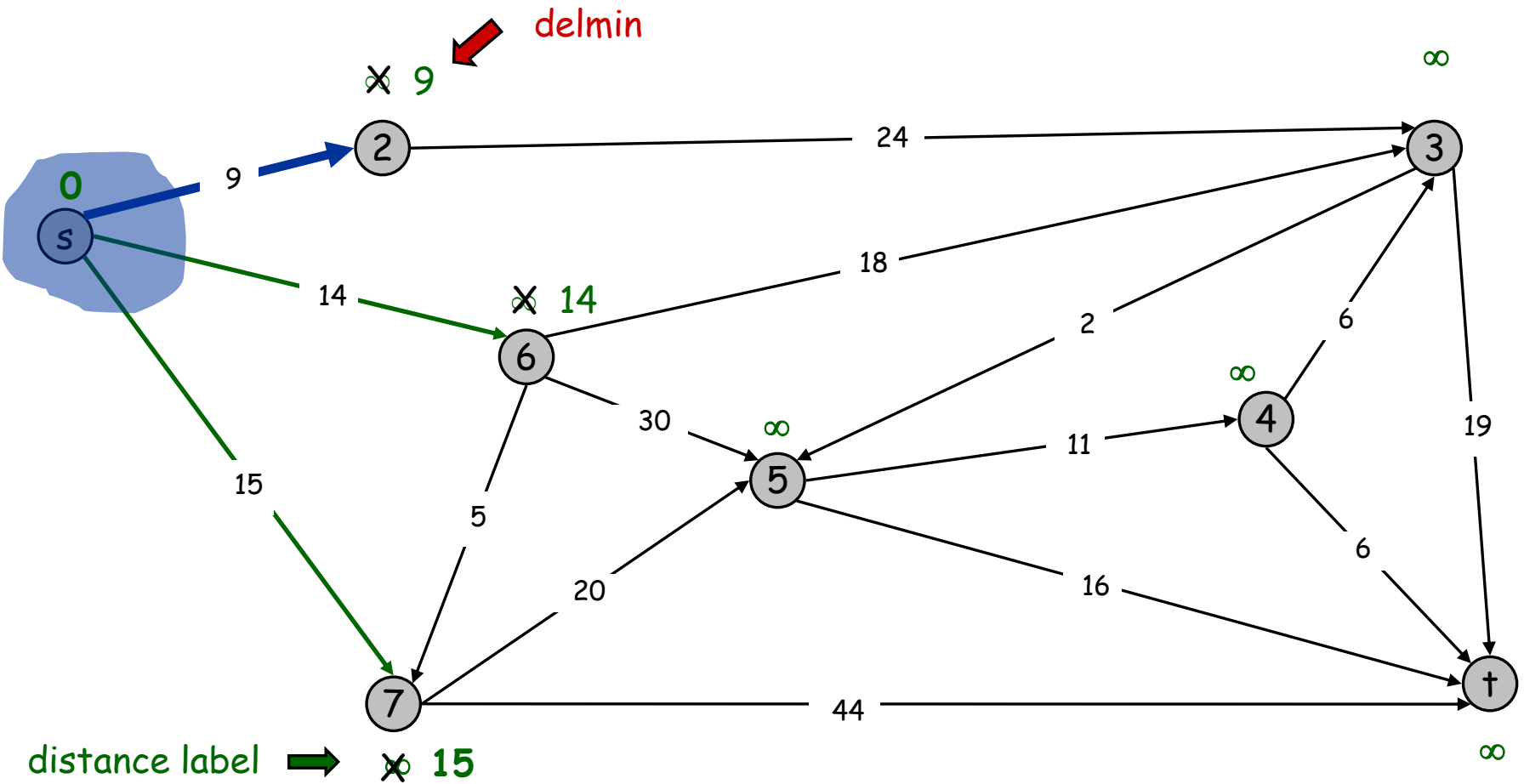
~~9~~



Dijkstra's Shortest Path Algorithm

$S = \{s\}$

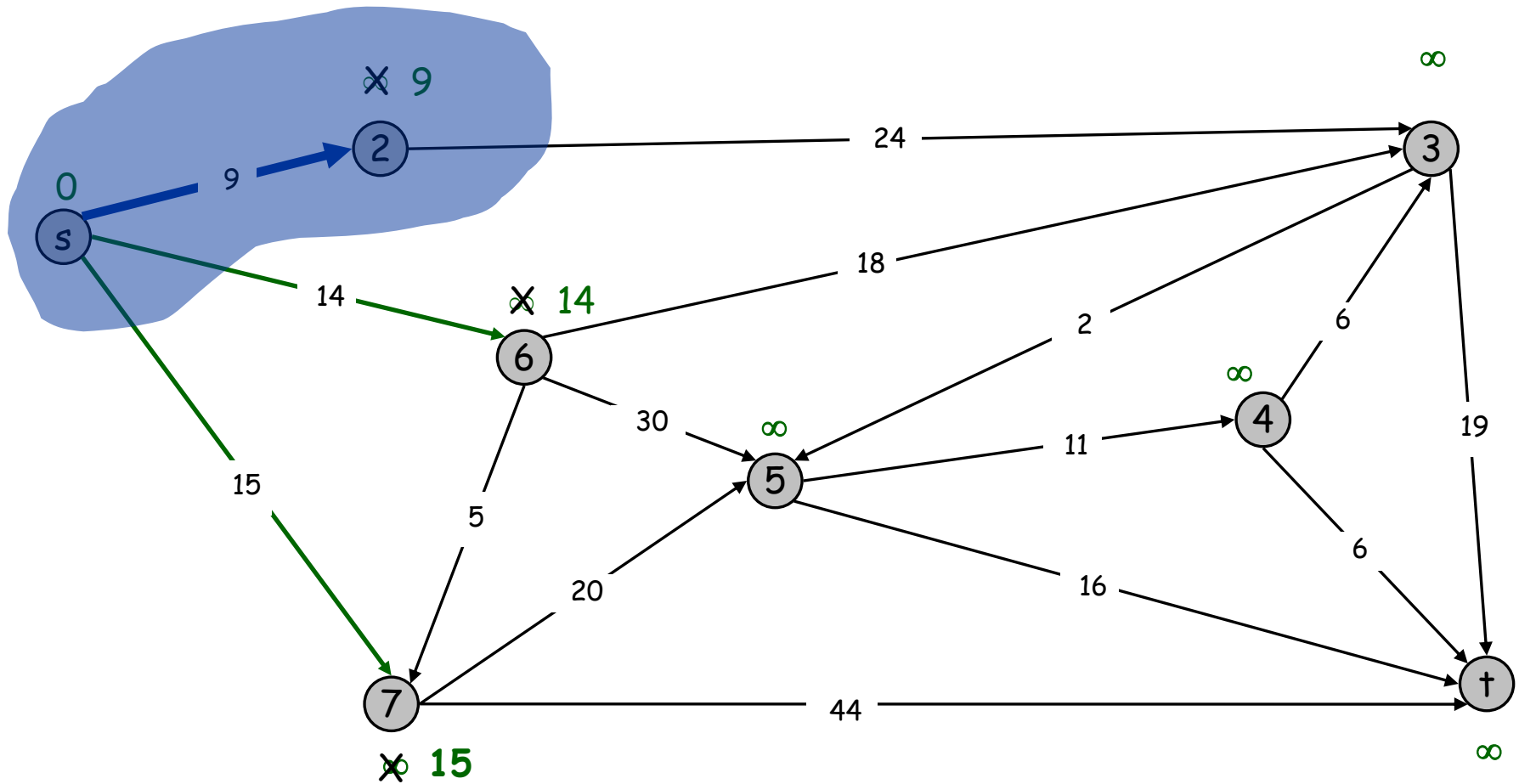
$PQ = \{2, 3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

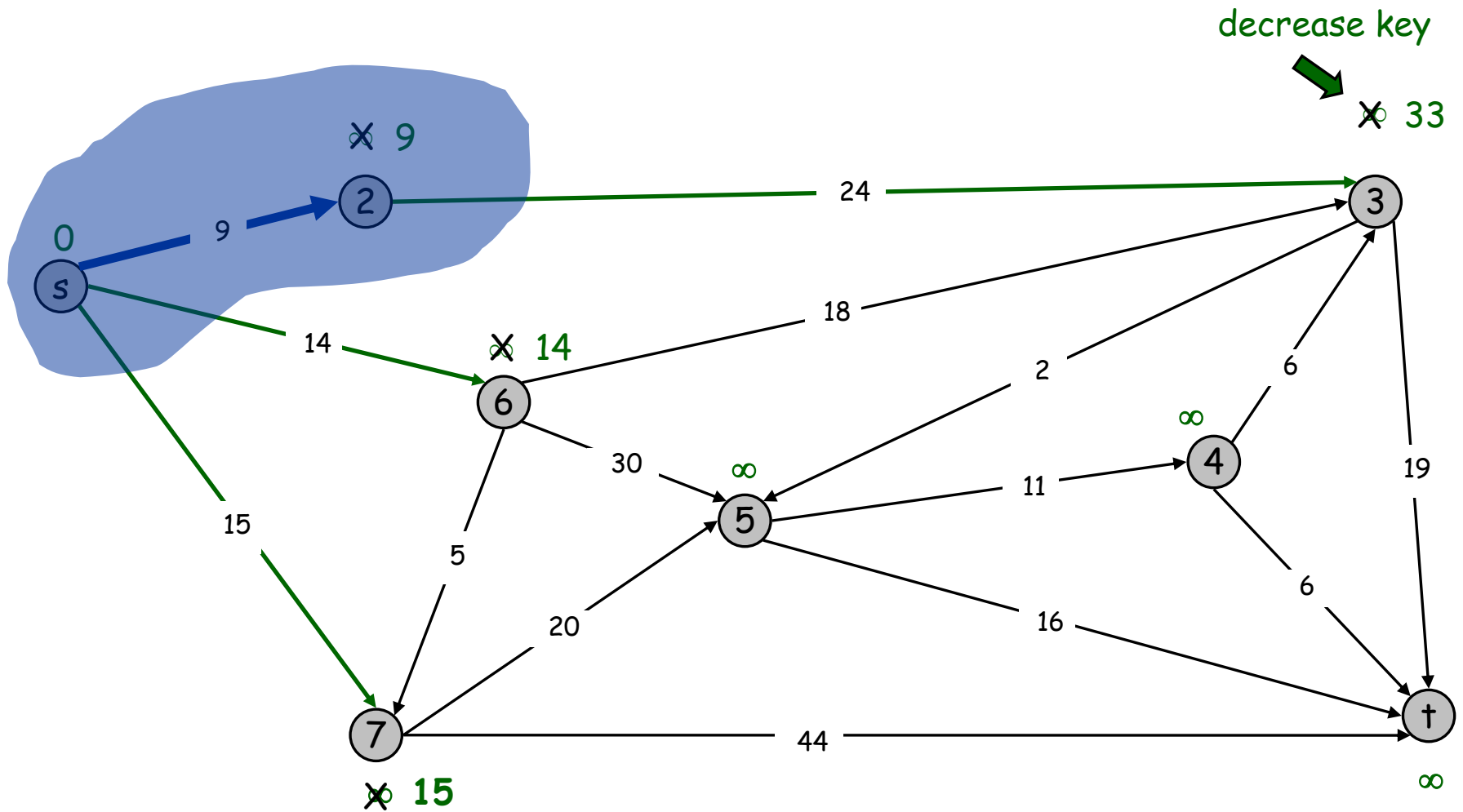
$PQ = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

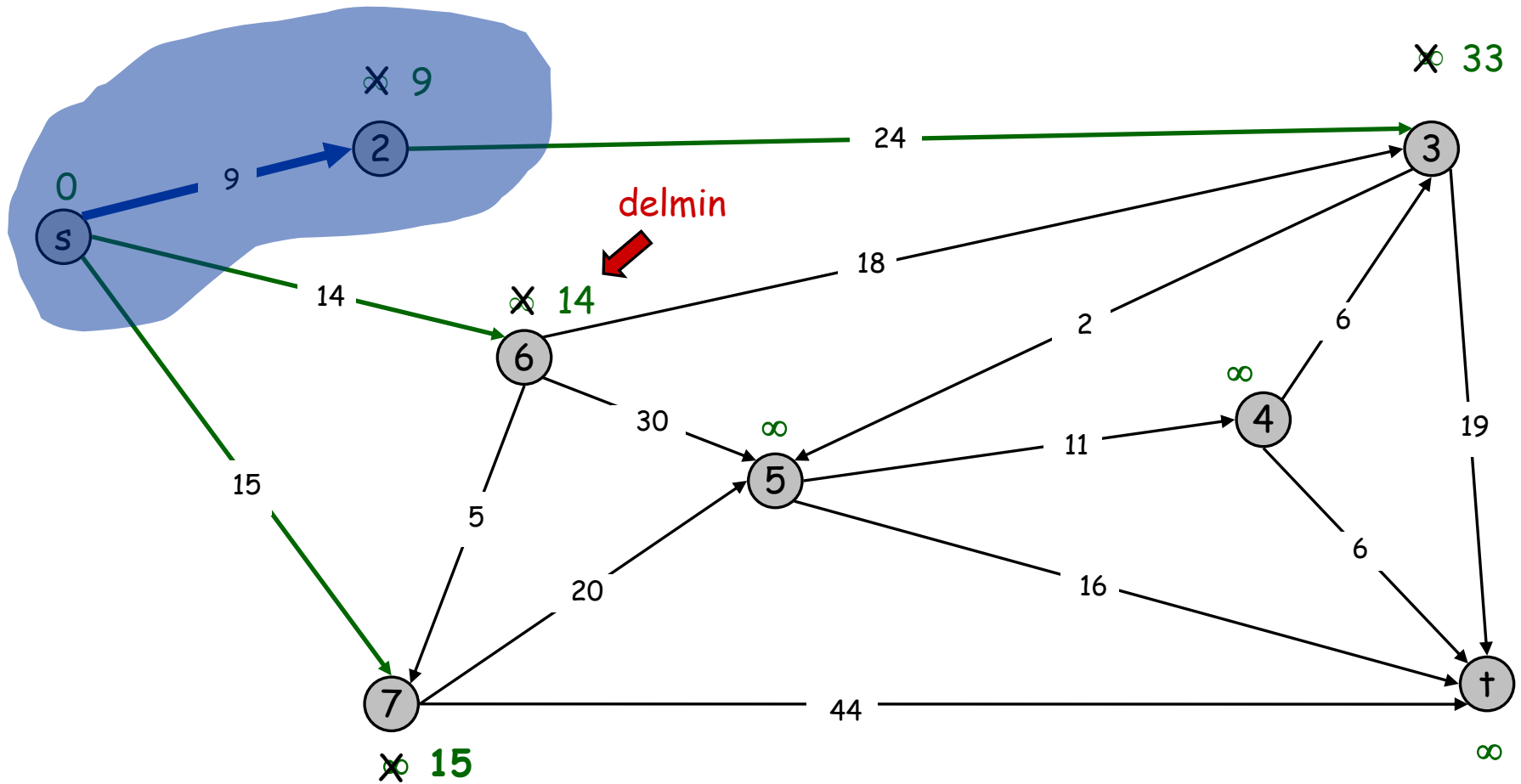
$PQ = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

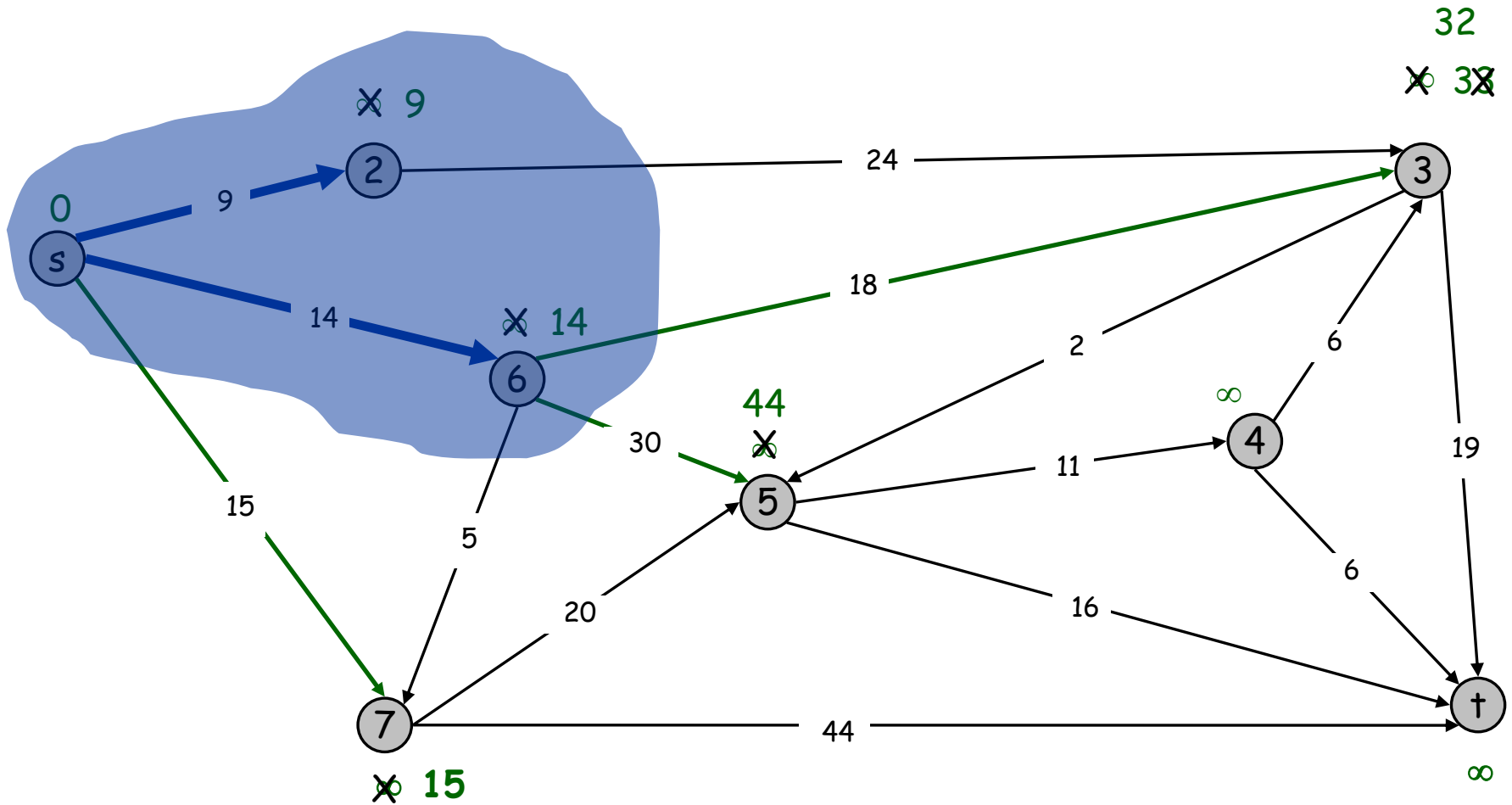
$PQ = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$

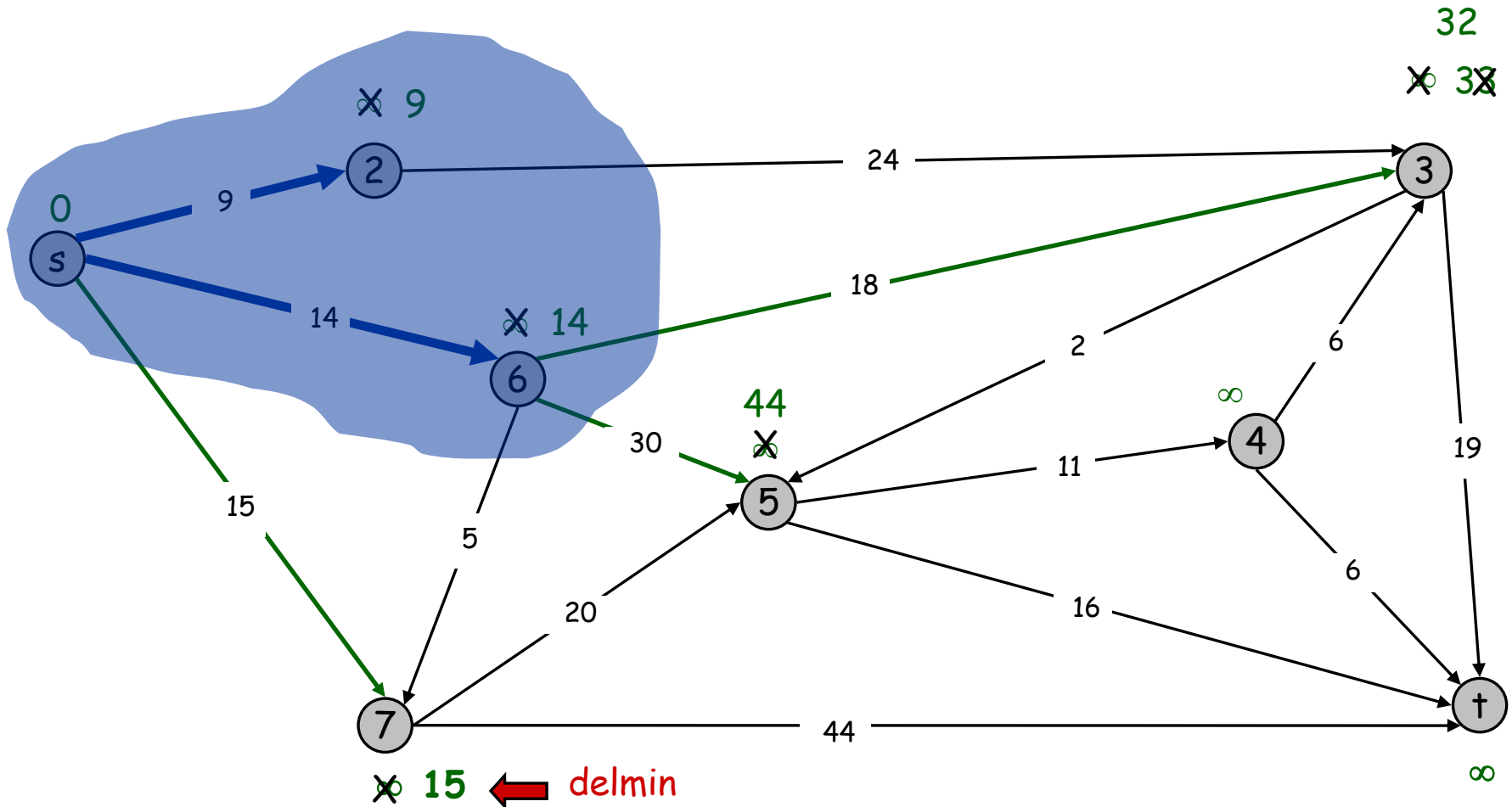
$PQ = \{3, 4, 5, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$

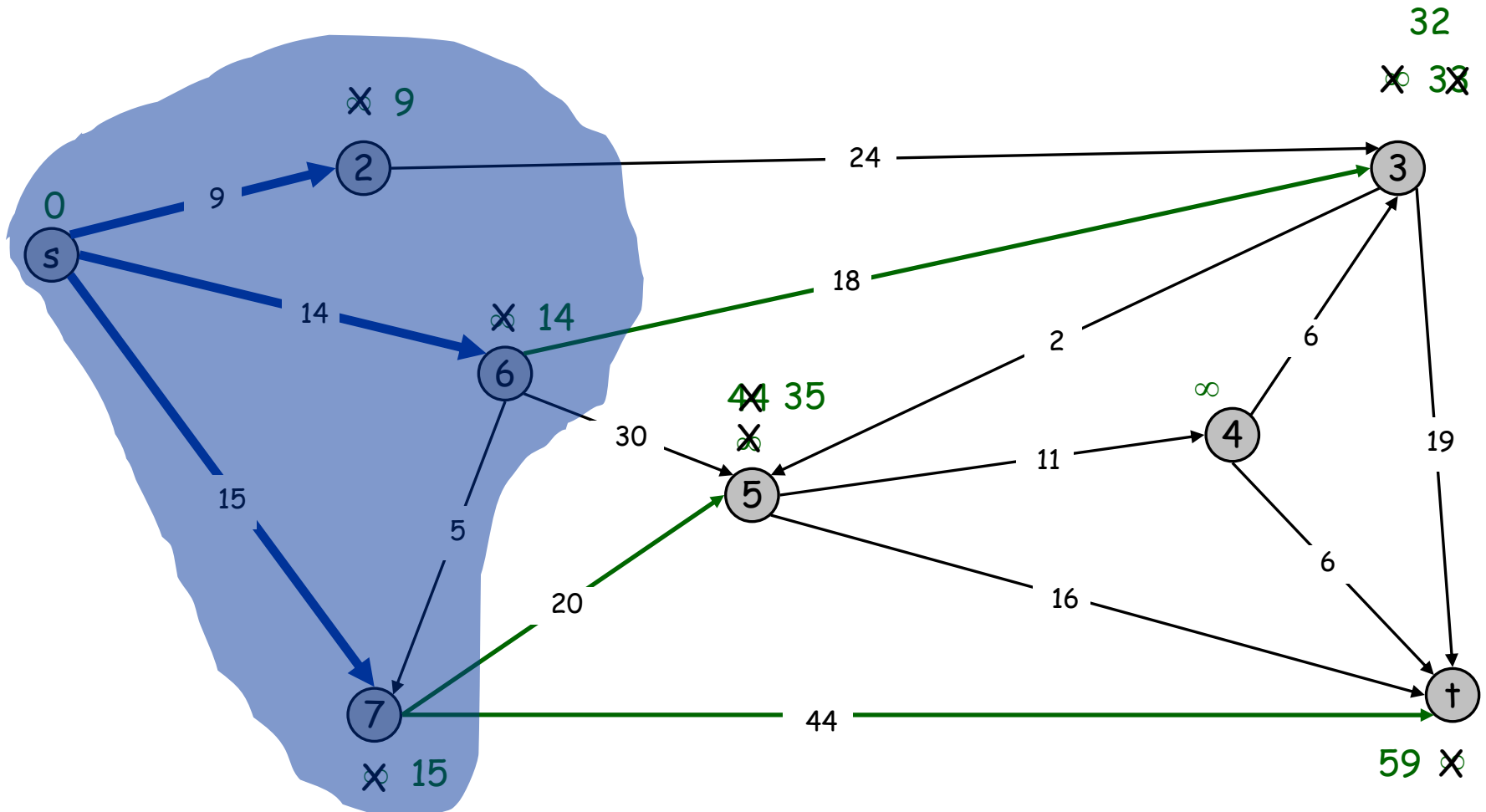
$PQ = \{3, 4, 5, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6, 7\}$

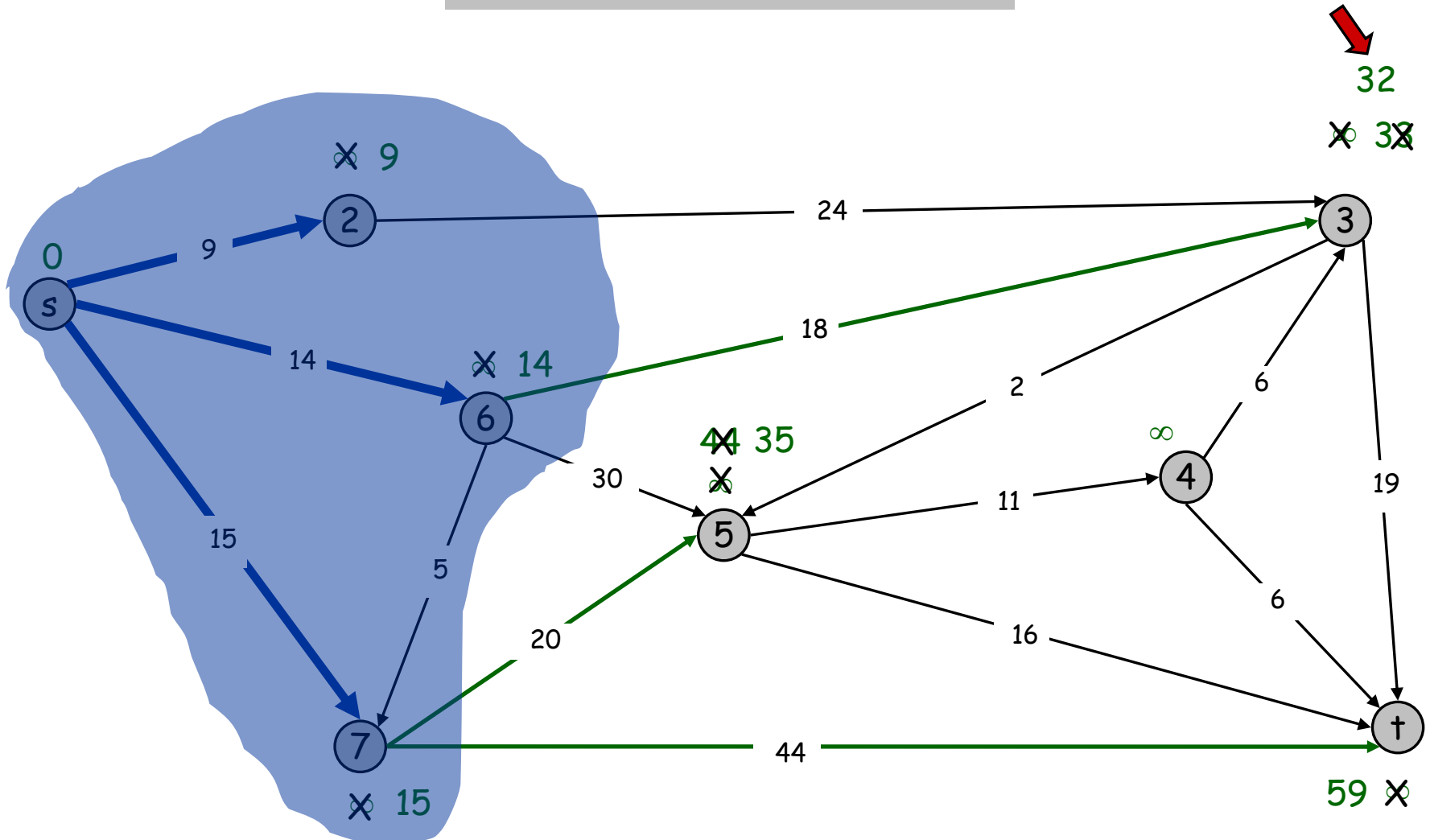
$PQ = \{3, 4, 5, \dagger\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6, 7\}$

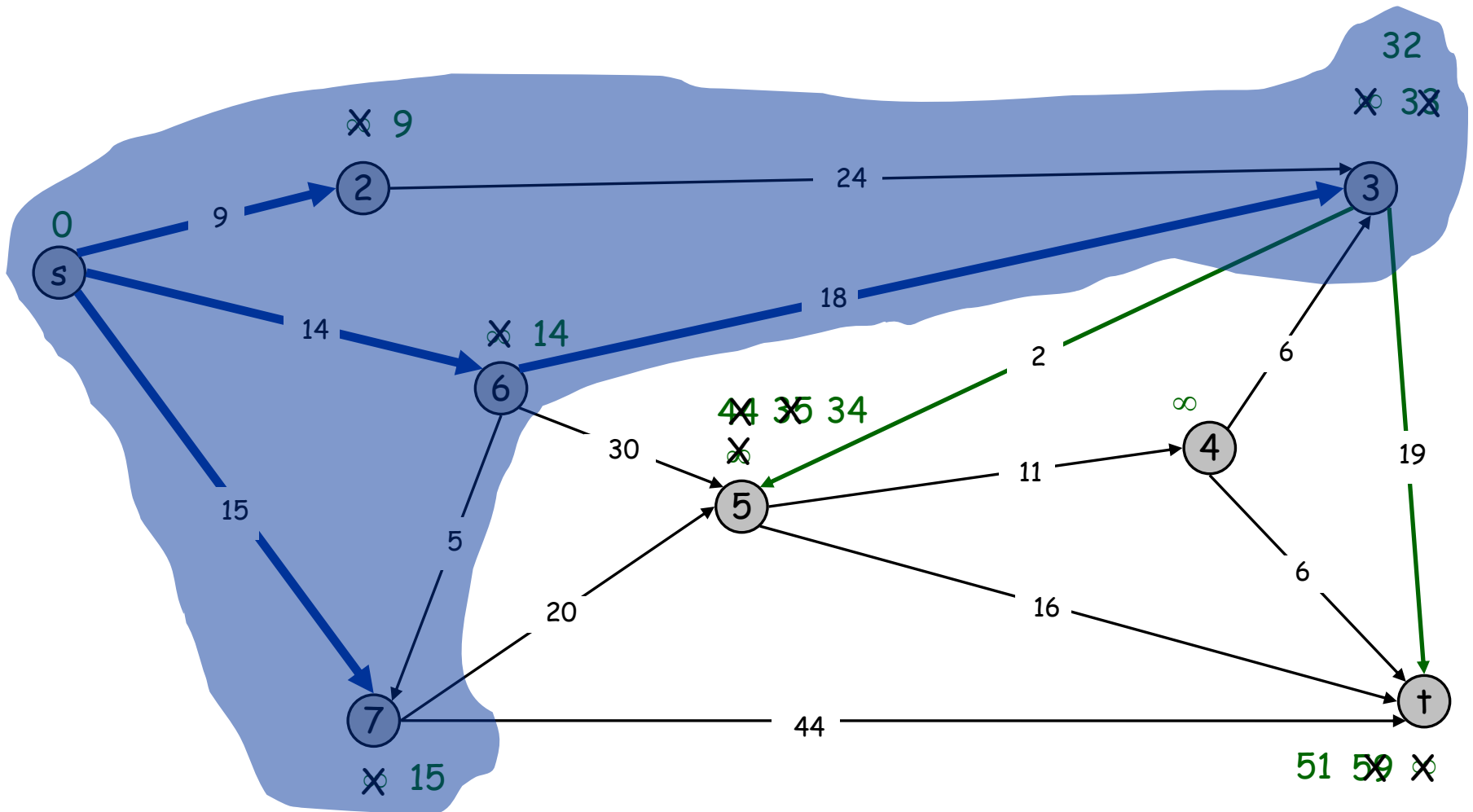
$PQ = \{3, 4, 5, \dagger\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 6, 7\}$

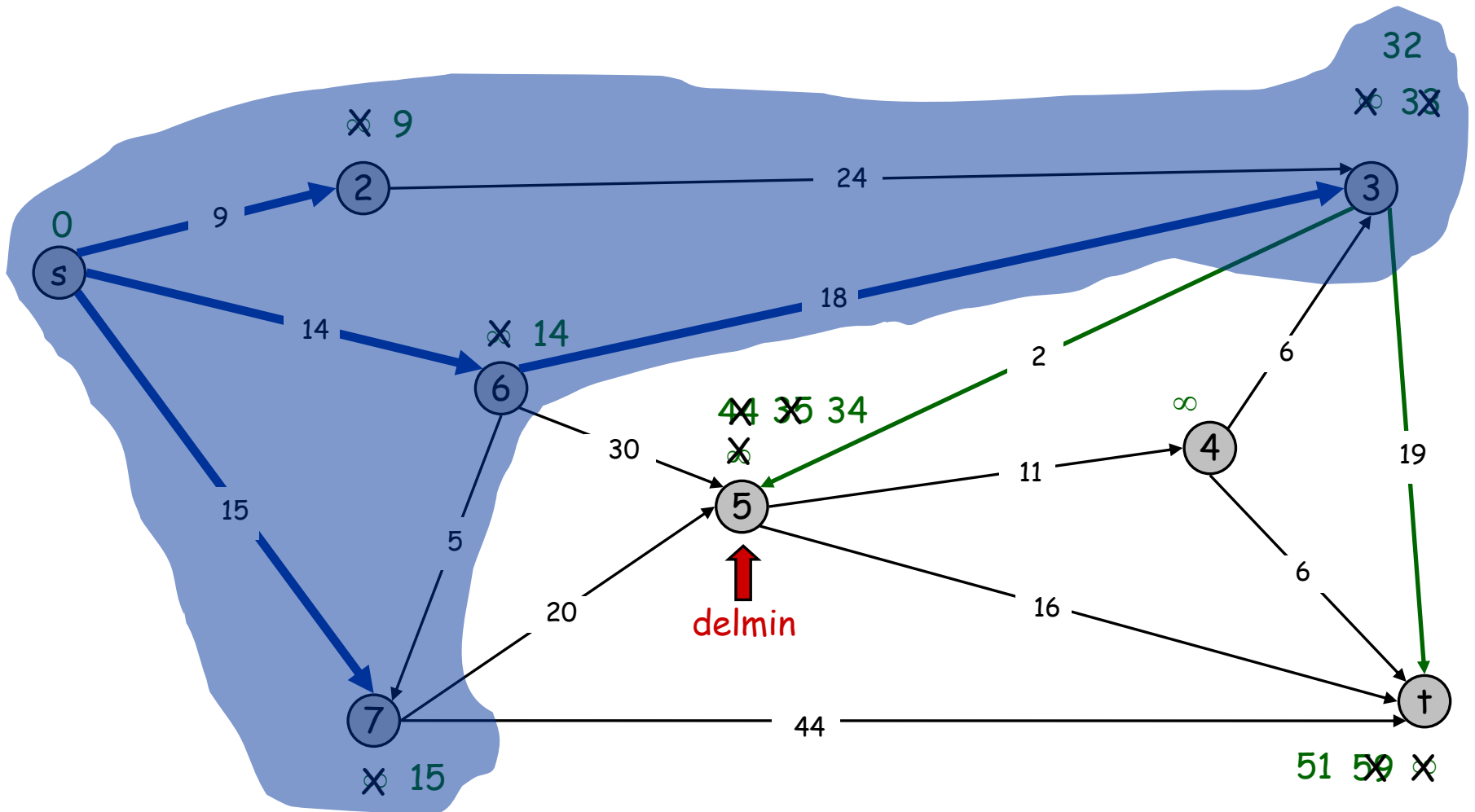
$PQ = \{4, 5, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 6, 7\}$

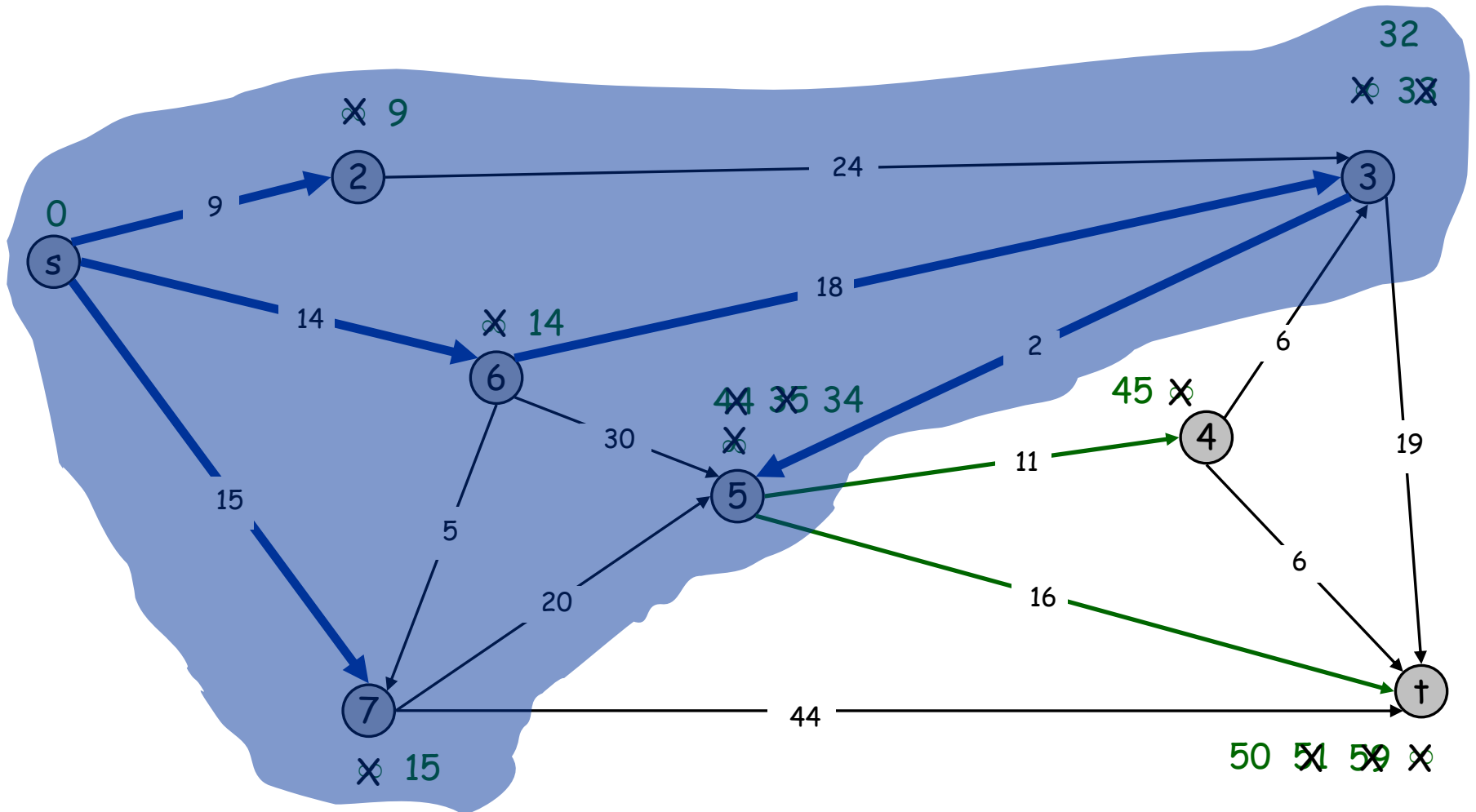
$PQ = \{4, 5, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 5, 6, 7\}$

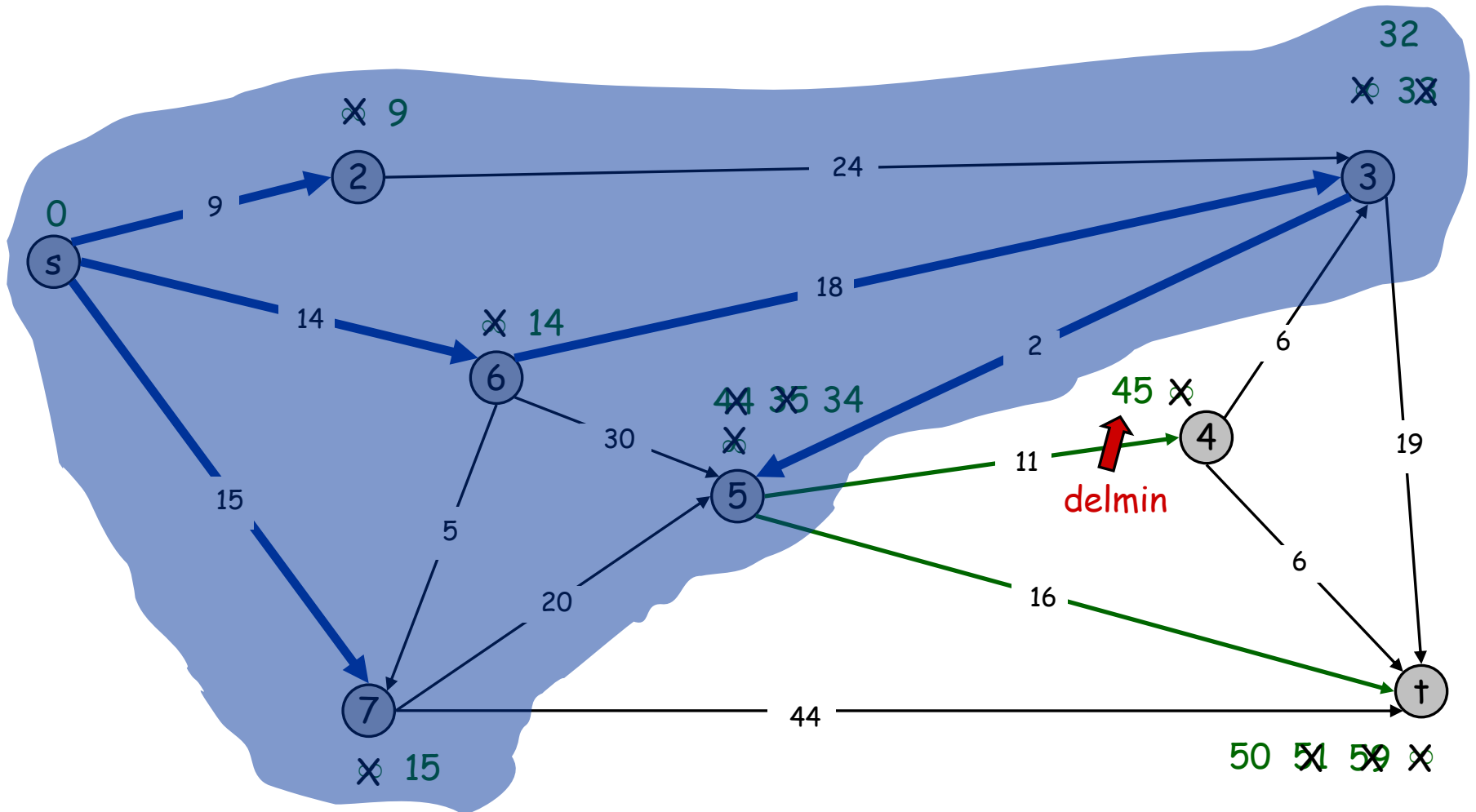
$PQ = \{4, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 5, 6, 7\}$

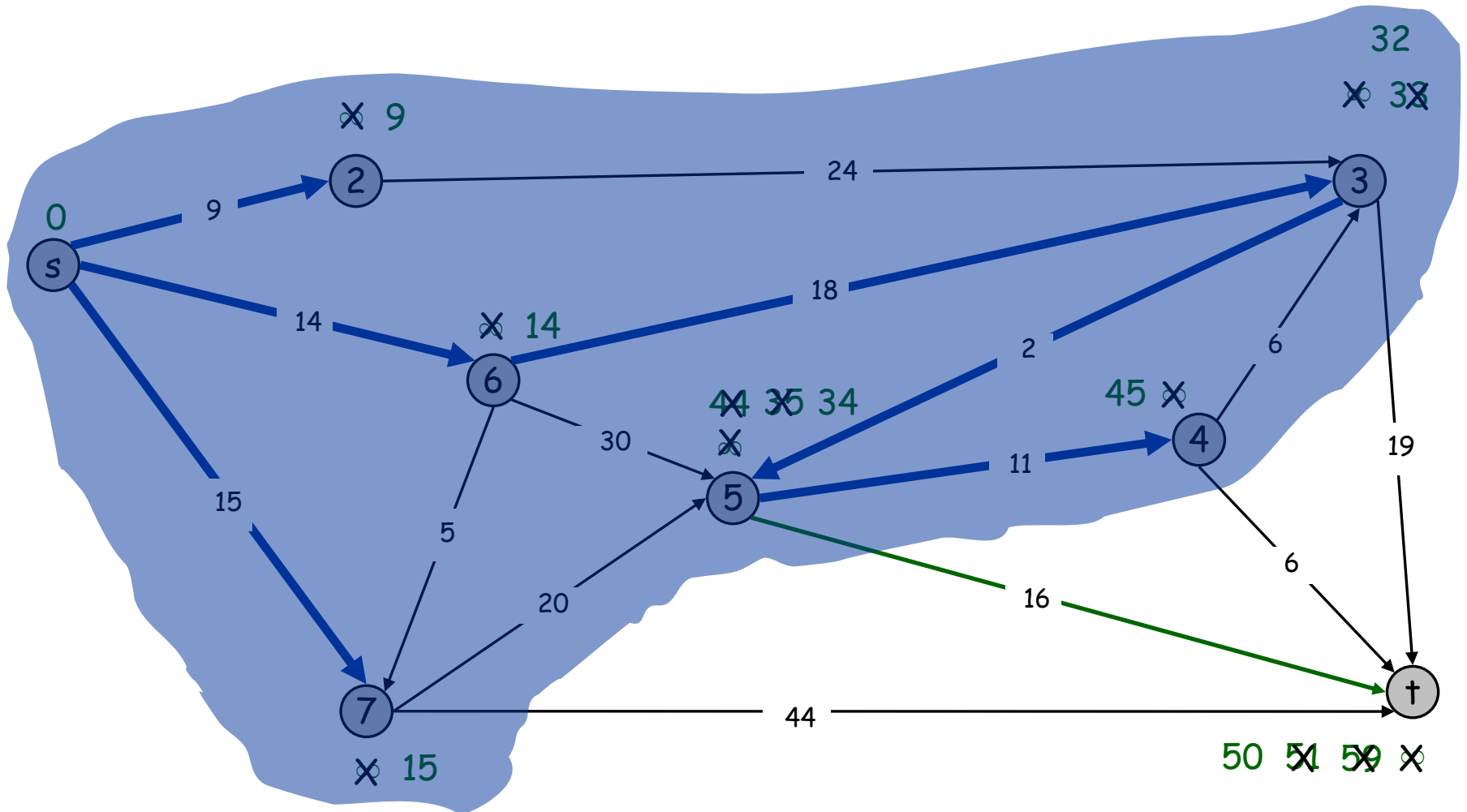
$PQ = \{4, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7\}$

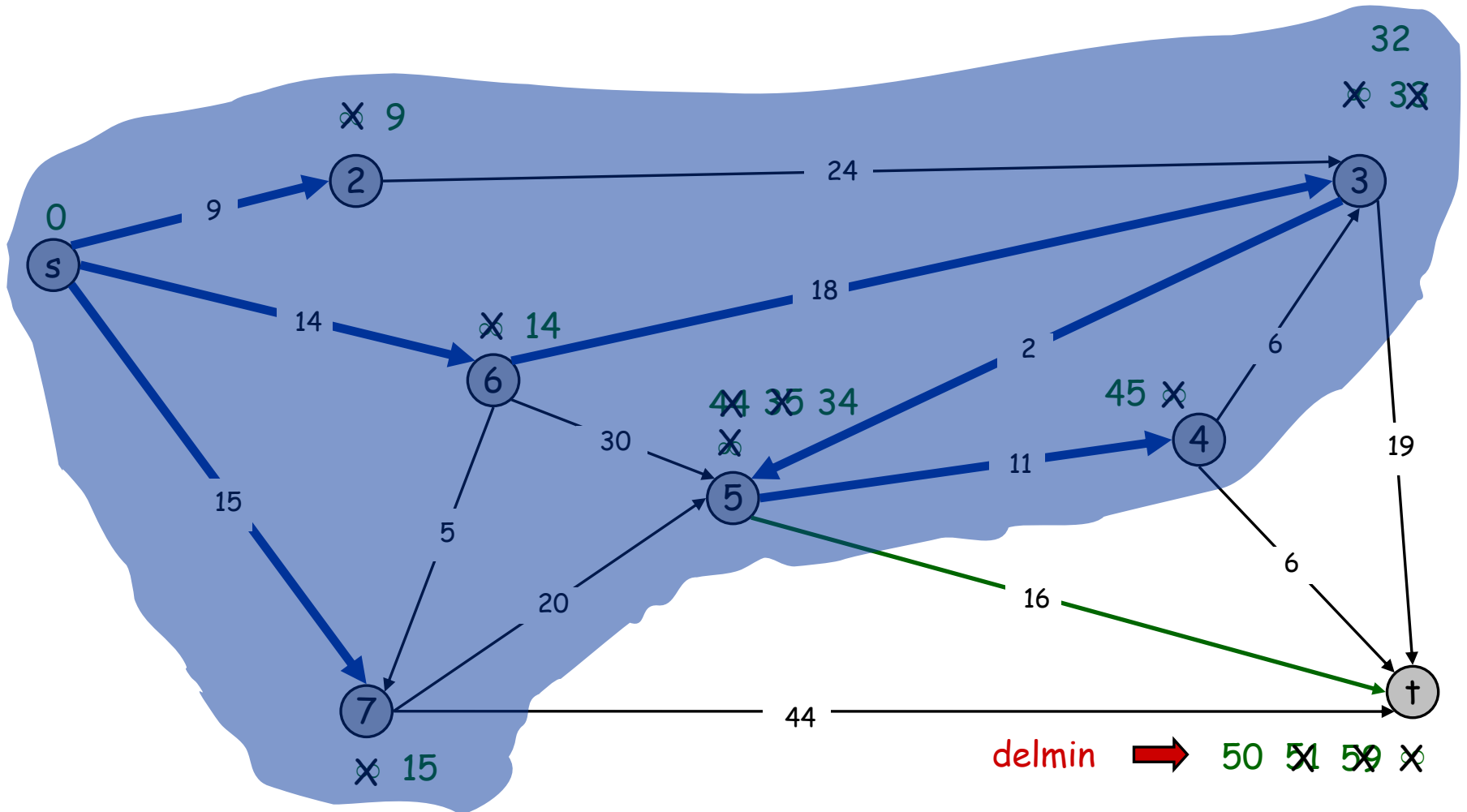
$PQ = \{t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7\}$

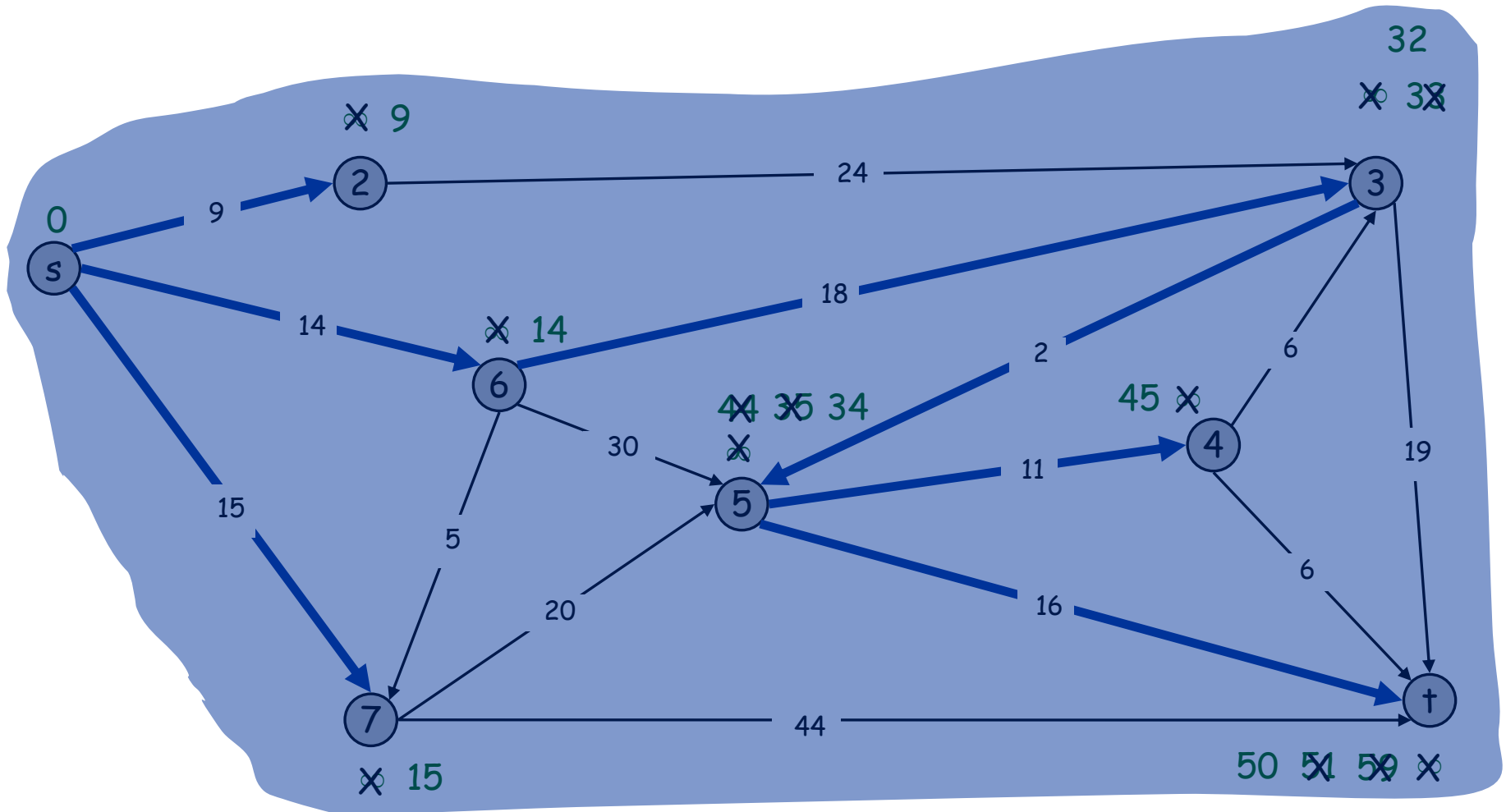
$PQ = \{t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7, t\}$

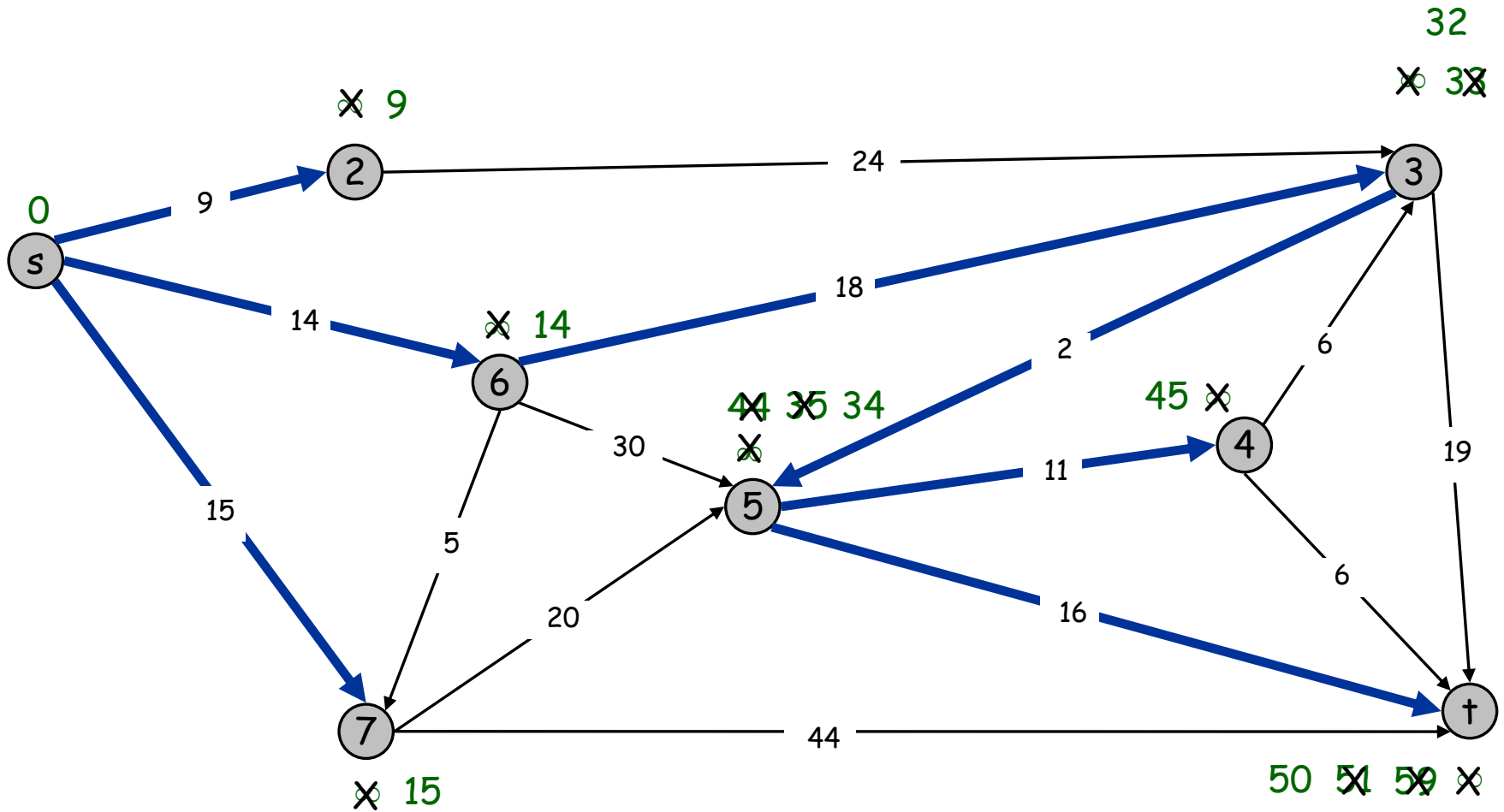
$PQ = \{\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7, t\}$

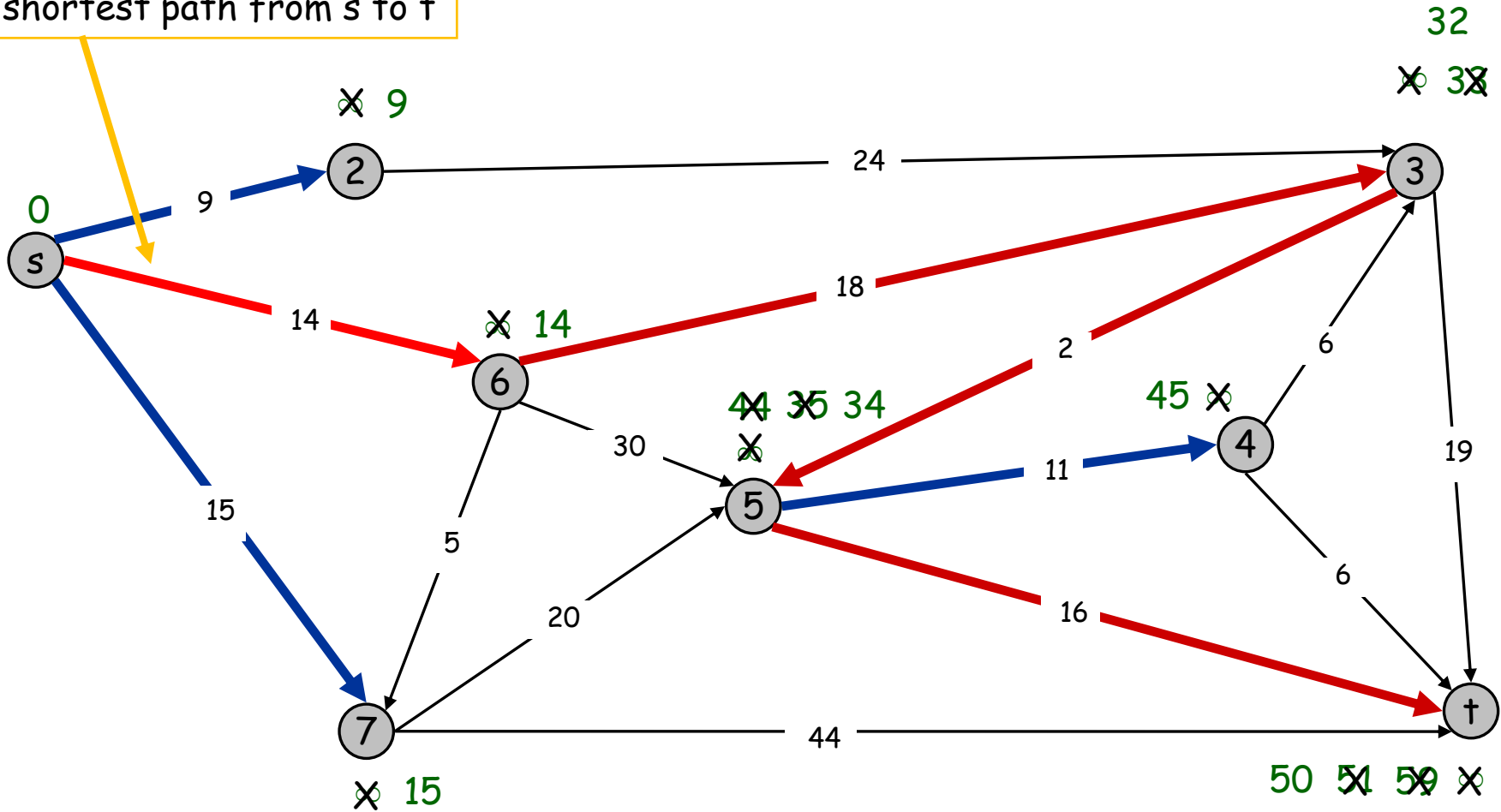
$PQ = \{\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7, t\}$
 $PQ = \{\}$

shortest path from s to t



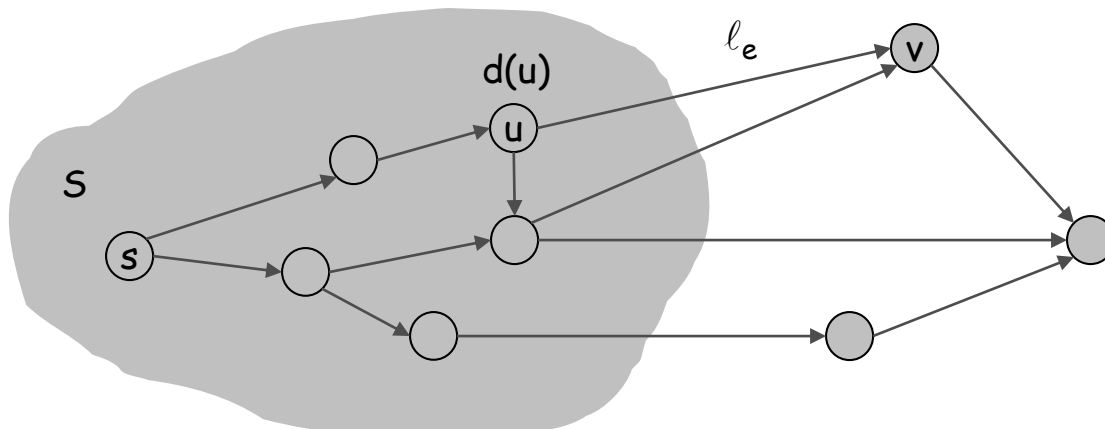
Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of **explored nodes** S for which we have determined the **shortest path distance** $d(u)$ from s to u .
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose **unexplored node** v which **minimizes**:

$$\rho(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm

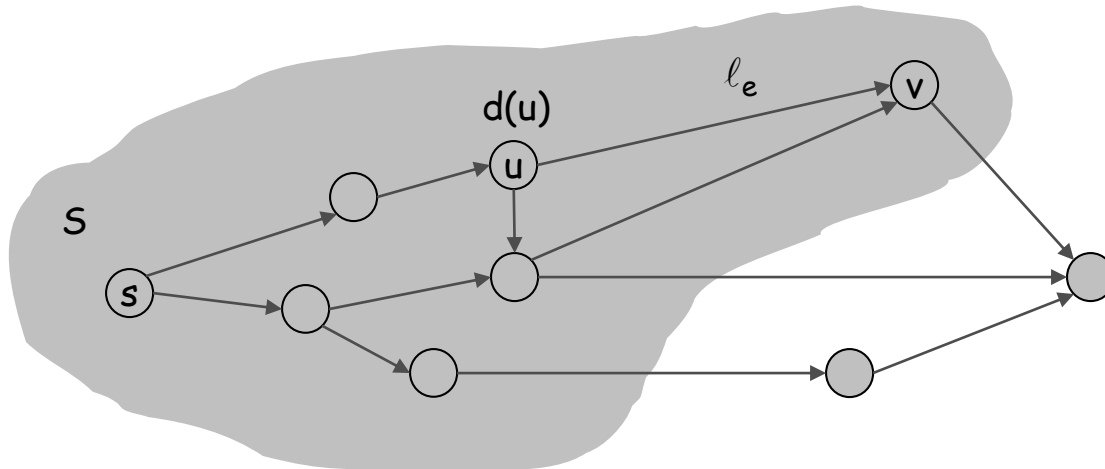
Dijkstra's algorithm.

- Maintain a set of **explored nodes** S for which we have determined the shortest path distance $d(u)$ from s to u .
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\rho(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$

add v to S , and set $d(v) = \rho(v)$.

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the **shortest s-u path**.

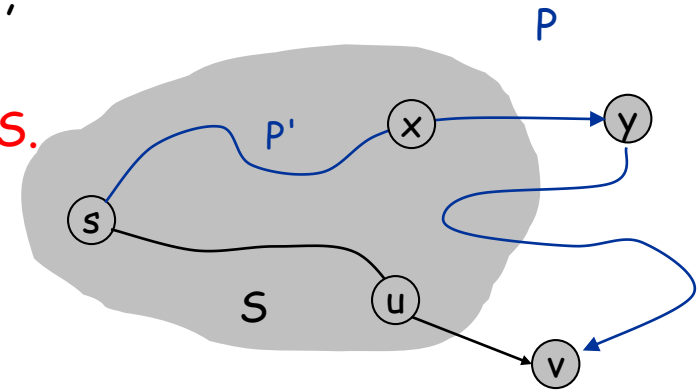
Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S , and let $u-v$ be the chosen edge.
- The shortest $s-u$ path plus (u, v) is an $s-v$ path of length $\pi(v)$.
- Consider any $s-v$ path P . We'll see that it's no shorter than $\pi(v)$.
- Let $x-y$ be the first edge in P that leaves S , and let P' be the $s-x$ path.

➤ P is already too long as soon as it leaves S .



$$\begin{array}{ccccccc}
 l(P) & \geq & l(P') + l(x, y) & \geq & d(x) + l(x, y) & \geq & \pi(y) \geq \pi(v) \\
 \uparrow & & \uparrow & & \uparrow & & \uparrow \\
 \text{nonnegative} & & \text{inductive} & & \text{defn of } \pi(y) & & \text{Dijkstra chose } v \\
 \text{weights} & & \text{hypothesis} & & & & \text{instead of } y
 \end{array}$$

Dijkstra's Algorithm: Implementation

To Do:

For each unexplored node, explicitly maintain: $\pi(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v , for each incident edge $e = (v, w)$

update: $\rho(w) = \min \{ \rho(w), \rho(v) + \ell_e \}$.

Implementation & Running Time

Dijkstra's Algorithm:

$S = \{\}$; $d[s] = 0$; $d[v] = \text{infinity}$ for $v \neq s$

while $S \neq V$

Choose v in $V \setminus S$ with minimum $d[v]$

Add v to S

foreach w in the neighborhood of v

$d[w] = \min(d[w], d[v] + \ell(v, w))$

endif

endwhile

The running time depends on the implementation of these steps.

Implementation & Running Time

Dijkstra's Algorithm:

$S = \{\}$; $d[s] = 0$; $d[v] = \text{infinity}$ for $v \neq s$

while $S \neq V$

Choose v in $V \setminus S$ with minimum $d[v]$

$O(n)$

Add v to S

foreach w in the neighborhood of v

$d[w] = \min(d[w], d[v] + \ell(v, w))$

endif

try all candidate pairs:
 \Rightarrow (worst case) every
time we are processing
edges to all $(n-1)$
neighbours
 $\Rightarrow O(n)$

endwhile

Array Based Implementation

$\triangleright (n-1)$ iterations with $O(n + n)$ every time $\Rightarrow O(n^2)$

Implementation & Running Time

Dijkstra's Algorithm:

$S = \{\}$; PQ with $d[s] = 0$ and $d[v] = \text{infinity}$ for $v \neq s$ ← Maintain in a PQ

while PQ is not empty

$v \leftarrow \text{ExtractMin from PQ}$ ⇒ $O(\log n)$

Add v to S

foreach w in the neighborhood of v

Update value for w in PQ ← Single update requires $O(\log n)$

endif

endwhile

Priority Queue Based Implementation

- It is easy to think how many times do we have to run, altogether.

Assuming Adjacency lists are used, "for" loop runs $\text{in-deg}(v)$ for every $v \neq s$.

- m edges, once for every edge ⇒ $O(m \log n)$

Dijkstra's Algorithm

Tree of Shortest paths from s to any u :

Take an array $parent$ P of size n :

- Set $P[u] = v$, when $d[u]$ gets updated due to v .
- While setting u ($u \neq s$) as Explored (ExtractMin from PQ), add the edge $(u, parent[u])$ to the tree.

Space Requirements:

- A system of shortest paths from s is saved in $O(n)$ space.
- Applying Dijkstra's algorithm to every source node, we get a system of shortest paths from every node to every other node, stored in $O(n^2)$ space, which is optimal.

All edge lengths are EQUAL:

- Breadth First Search.

Dijkstra's Algorithm

Dijkstra's Algorithm design technique?

➤ Greedy?

- Next node is chosen based on a greedy rule.

➤ Dynamic Programming?

- We consider optimal shortest path for **subpaths** from s-t.

- Design techniques are not **RIGID** classification of efficient algorithms.
- Take them as **general design principles**.
 - Single algorithm can combine several.

Network Flow Algorithms



Some slides' contents adapted form:

http://homes.cs.washington.edu/~anderson/iucee/Slides_421_06/Lecture22_23.ppt

<http://elderlab.yorku.ca/~elder/teaching/cse3101/lectures/07%20Network%20Flow%20Algorithms.ppt>

Flow Networks

Internet

Telephone

Highways

Rail

Electrical Power

Gas

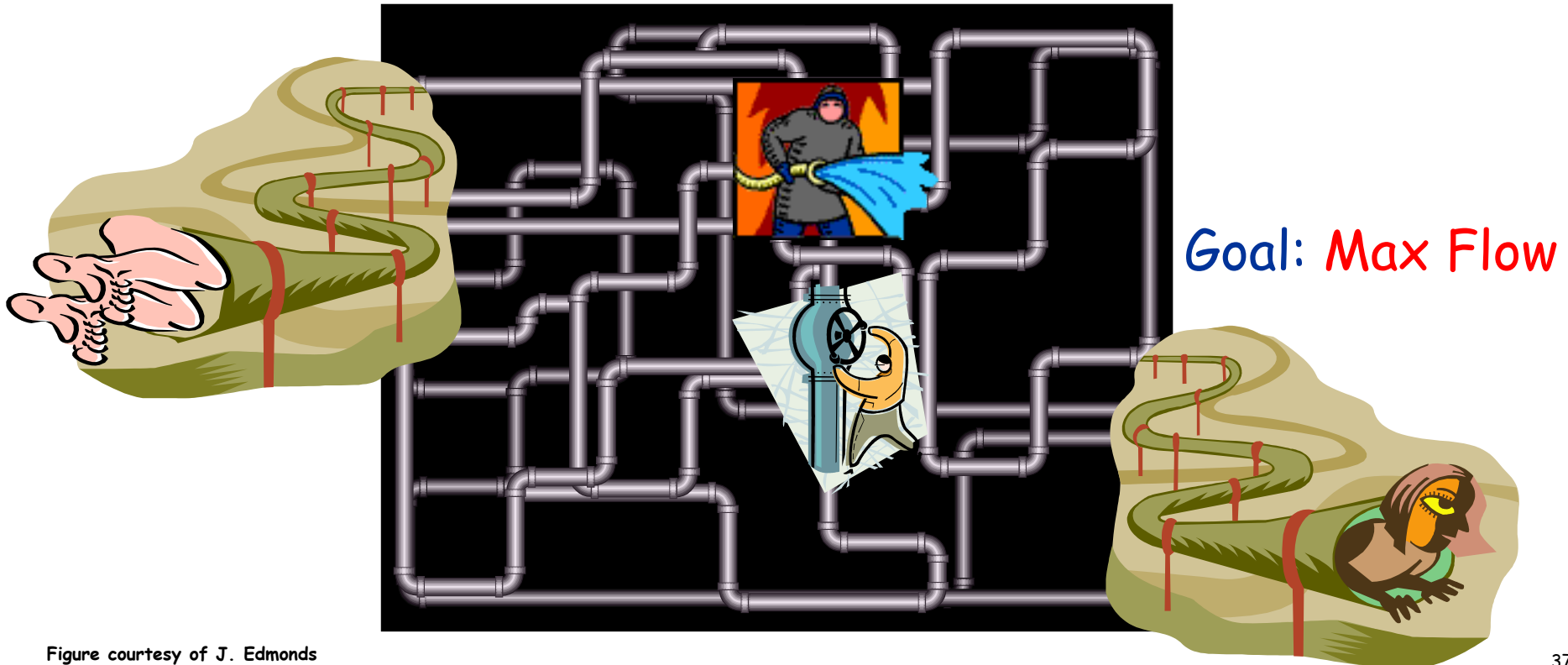
Water

...

Flows

• Warmup:

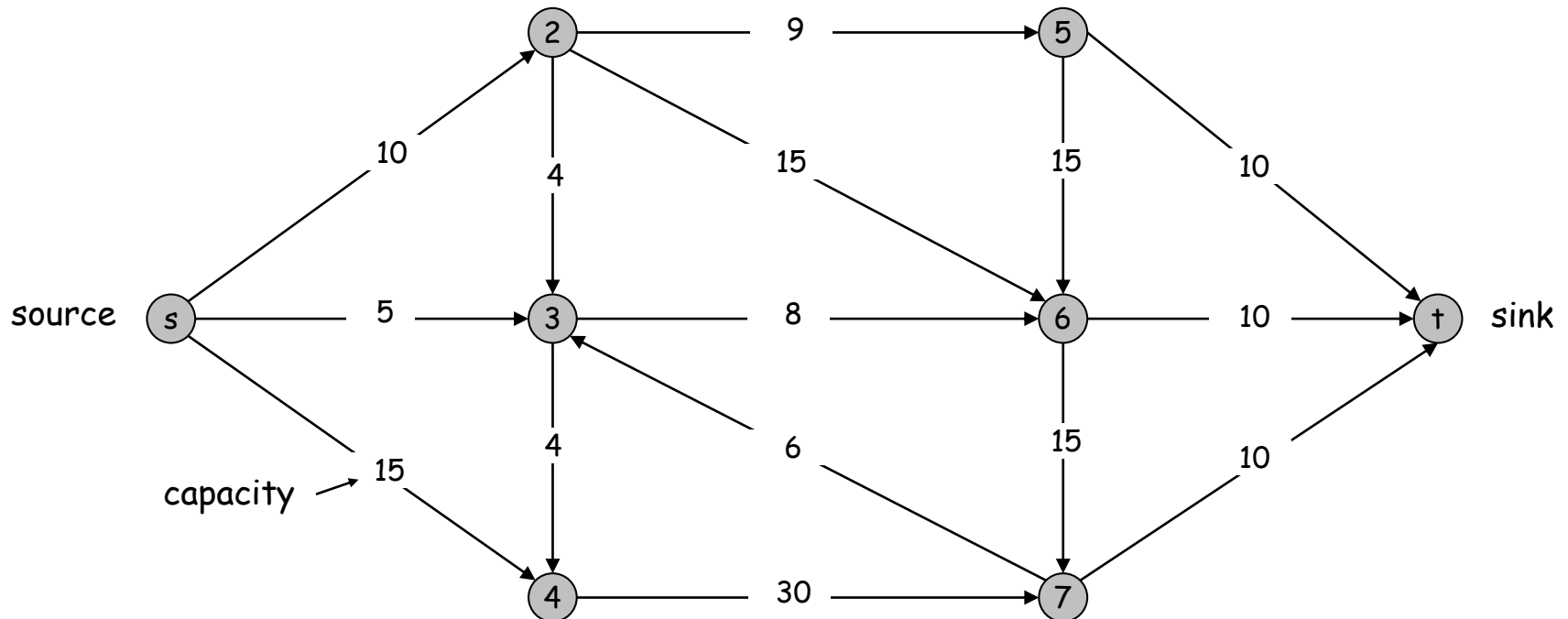
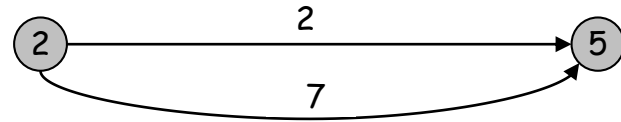
- A **network** of pipes
- pipes carry **flow**
- Each pipe has a maximum **capacity**
- A **source** in which flow **arrives**
- A sink at which flow **leaves**
- Only the **positive** flows in the flow network.



Flows

Flow network.

- Abstraction for material **flowing** through the edges.
- $G = (V, E) =$ **directed** graph, **no parallel edges**.
- Two distinguished nodes:
 - **s** = source, **in-degree(s) = 0**.
 - **t** = sink, **out-degree(t) = 0**.
- **c(e)** or **c(u,v)** : capacity of edge e (u,v).
- **f(e)** or **f(u,v)** : flow through edge e (u,v).

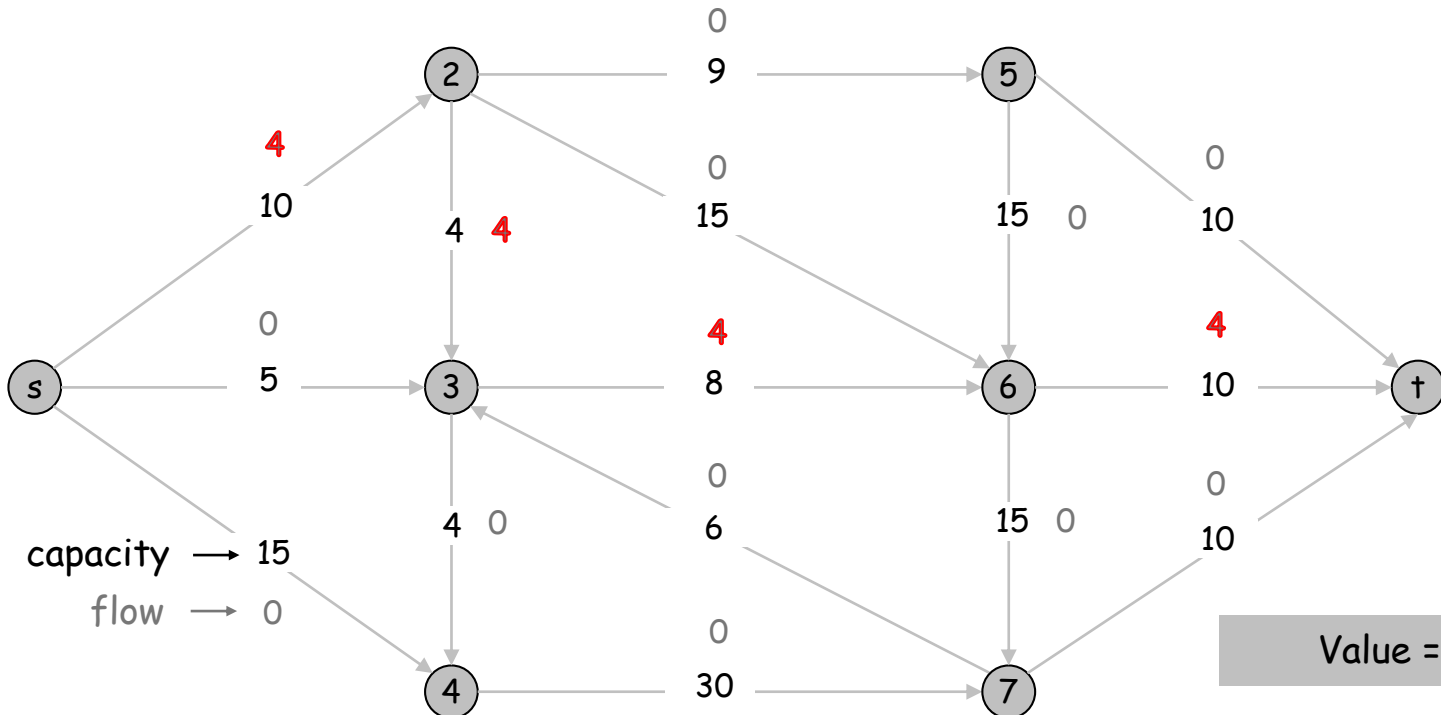


Flows

Def. An **s-t flow** is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [conservation]

Def. The **value** of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e) = \sum_{e \text{ in to } t} f(e)$.



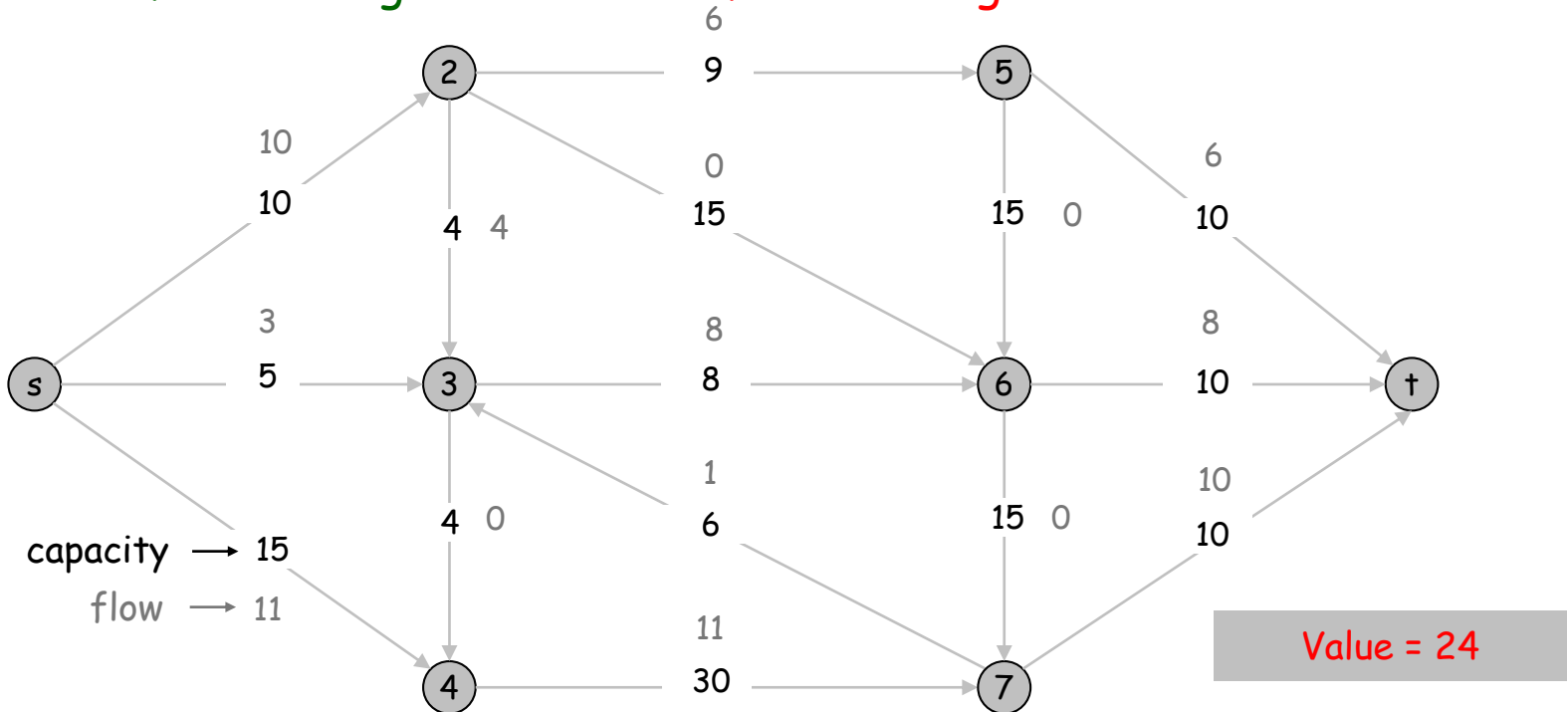
Flows

Def. An **s-t flow** is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [conservation]

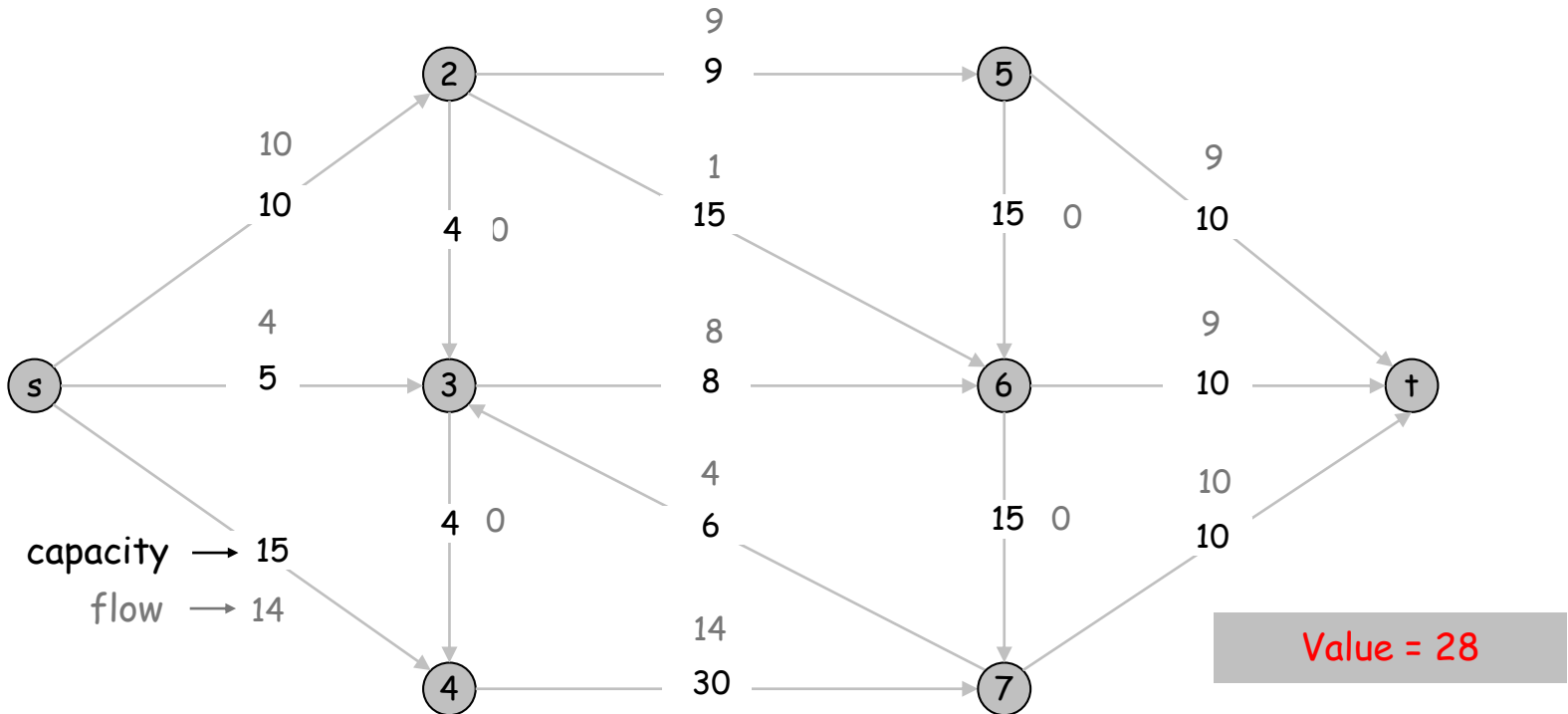
Def. The **value** of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e) = \sum_{e \text{ in to } t} f(e)$.

the total flow leaving s = the total flow arriving in t .



Maximum Flow Problem

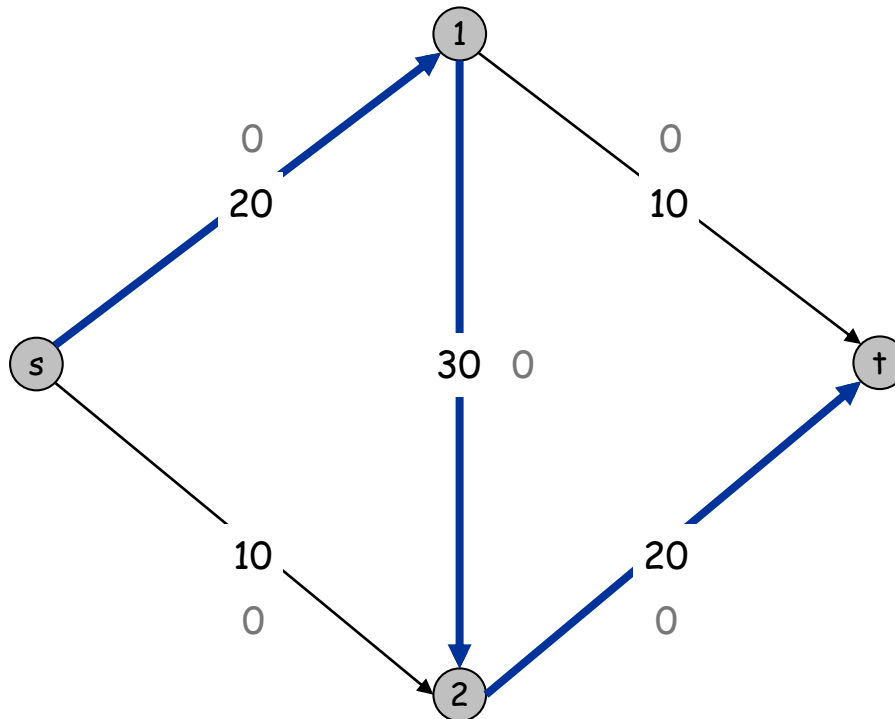
Max flow problem. Find s-t flow of **maximum** value.



Towards a Max Flow Algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an s - t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get stuck.

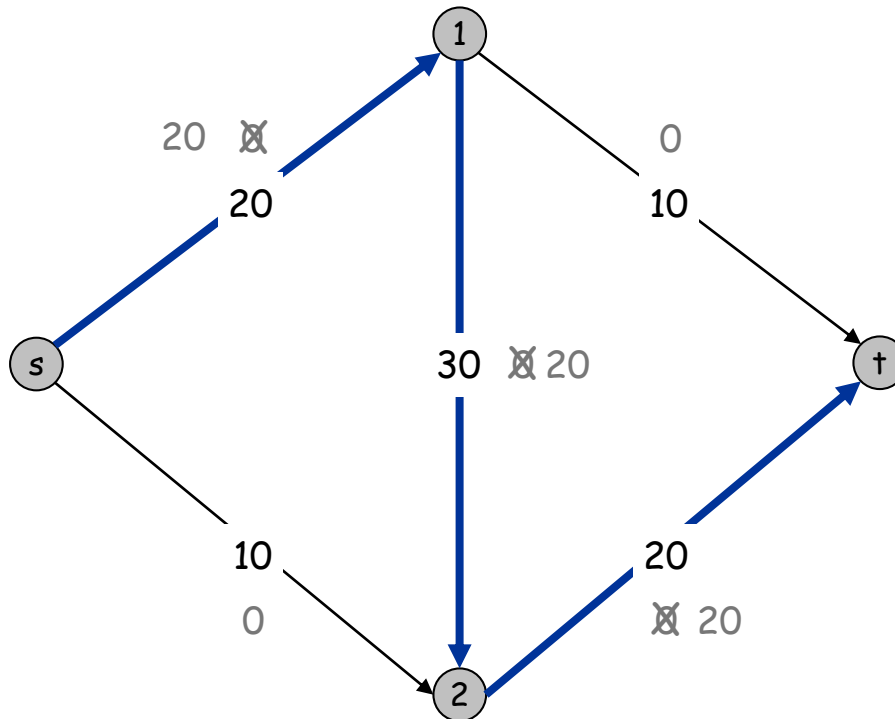


Flow value = 0

Towards a Max Flow Algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an s - t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get stuck.



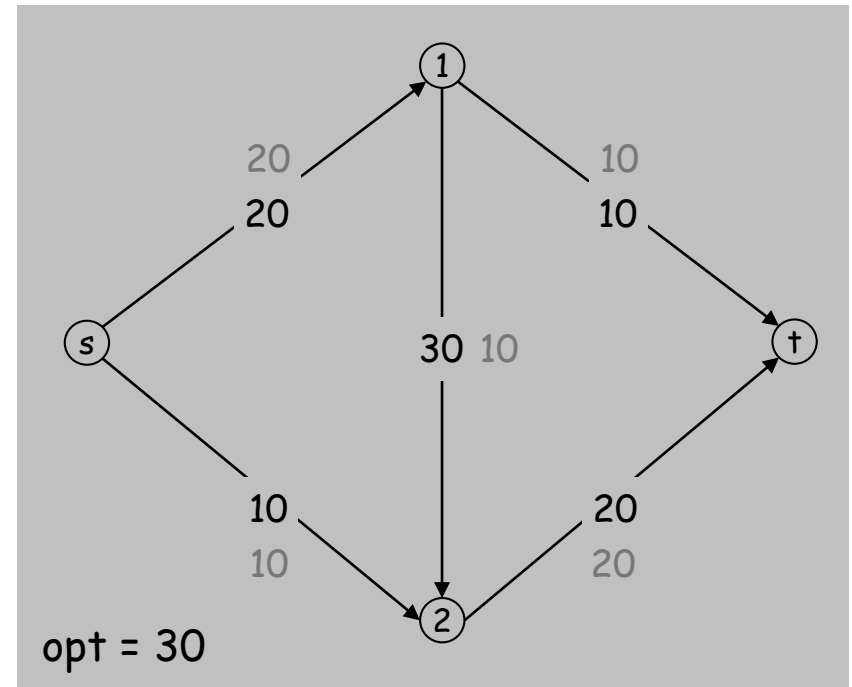
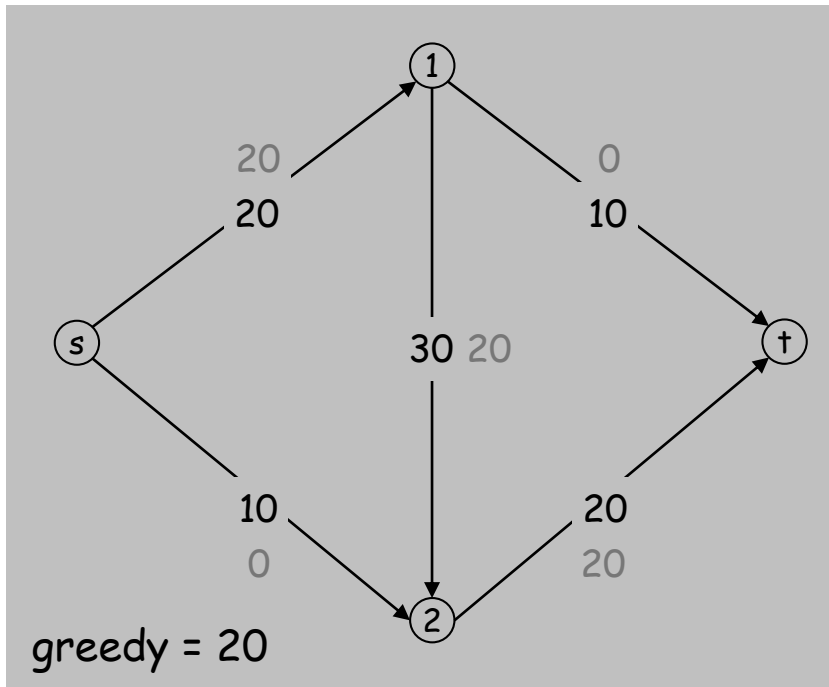
Flow value = 20

Towards a Max Flow Algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an s - t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get **stuck**.

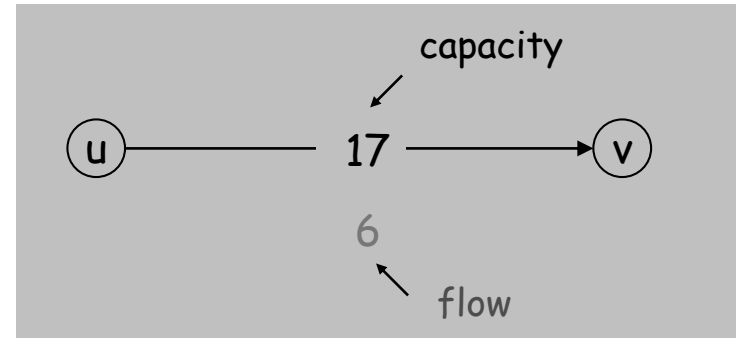
local optimality \nRightarrow global optimality



Residual Graph

Original edge: $e = (u, v) \in E$.

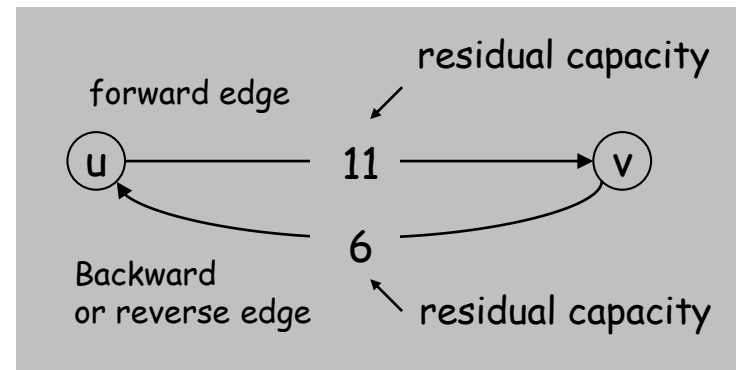
- Flow $f(e)$, capacity $c(e)$.



Residual edge.

- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

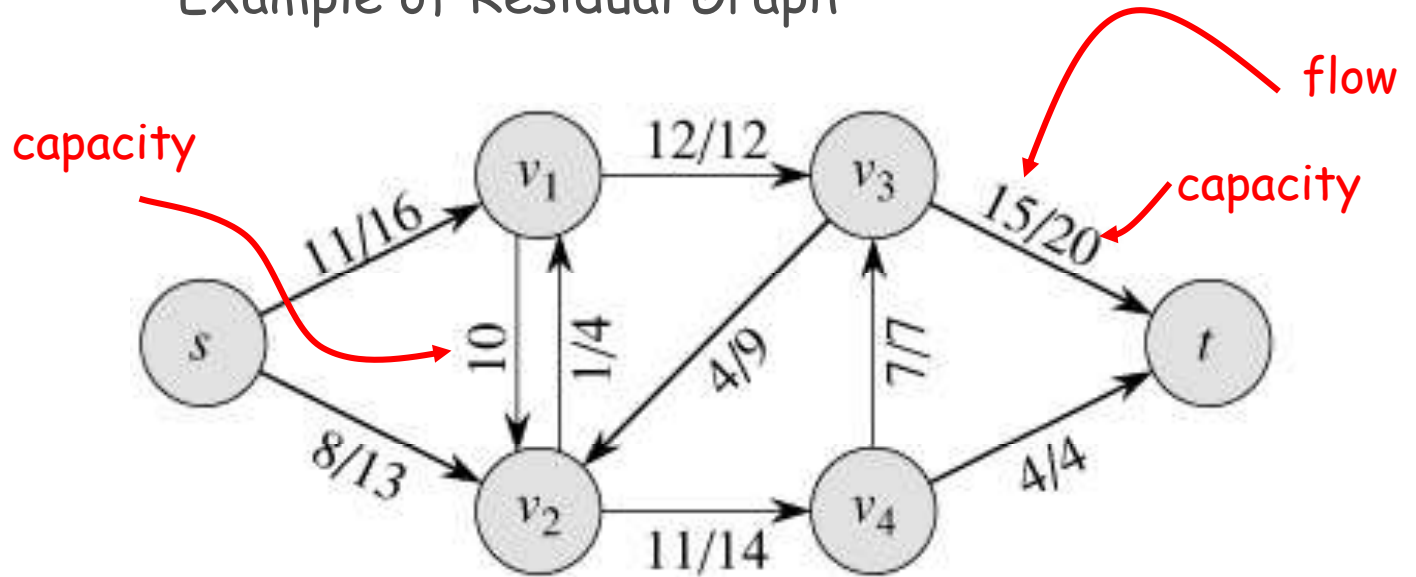


Residual graph: $G_f = (V, E_f)$.

- Residual edges with positive residual capacity.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$.

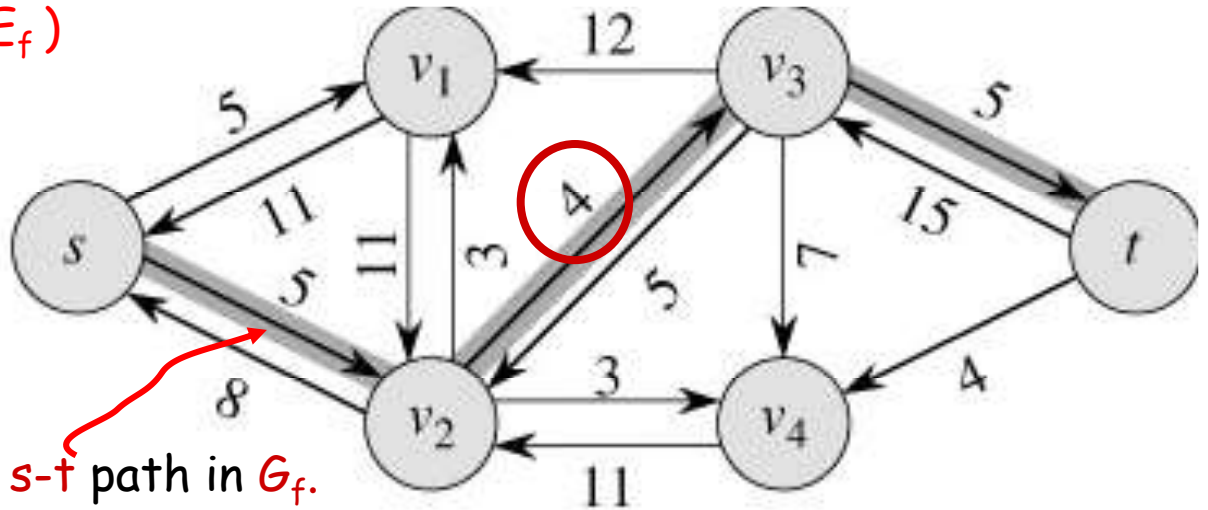
Example of Residual Graph

Graph $G = (V, E)$



Residual Graph $G_f = (V, E_f)$

wrt the flow f ,
whose value is 5 here





$P =$ any simple directed s - t path in G_f .

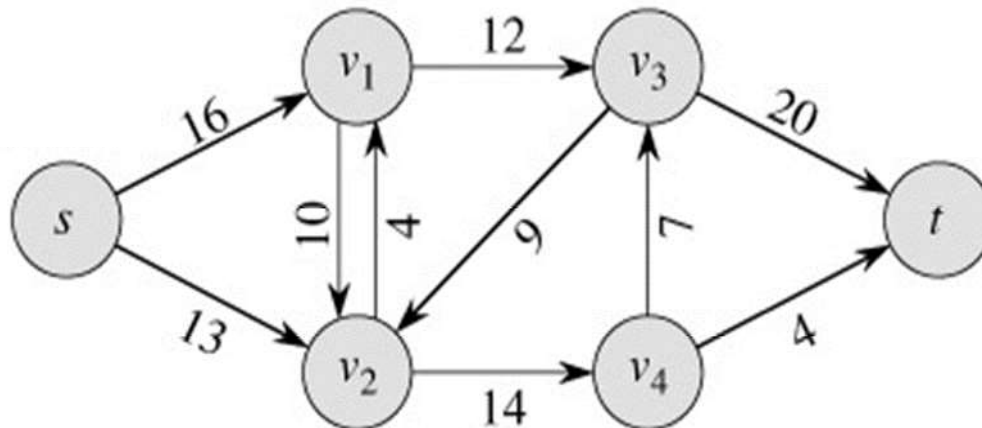
Bottleneck $b =$ the smallest residual capacity along P

P is called **Augmenting path** as we can push a flow s to t equal to b on P

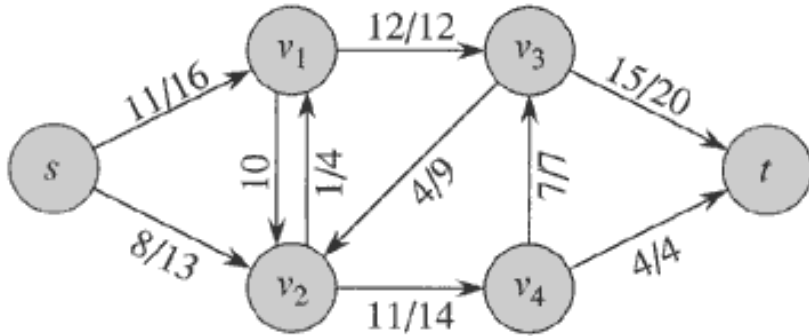
Computing Max Flow

Correcting the Greed: Ford-Fulkerson algorithm

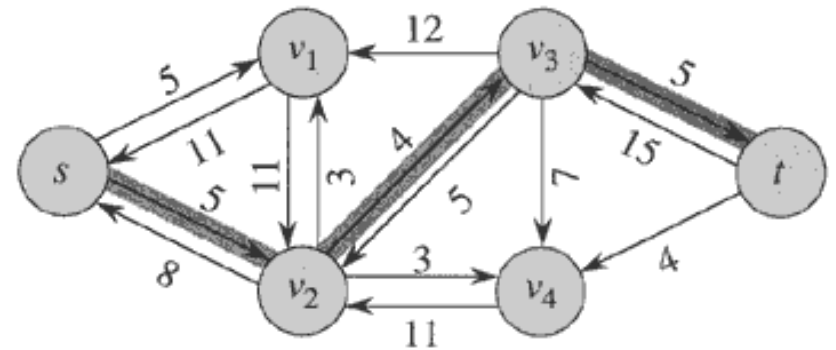
- Start with $f(e) = 0$ for all edge $e \in E$.
 - Construct residual graph G_f (in first iteration it is same as G)
 - Find an s-t **augmenting path** P in G_f with capacity $b > 0$
 - **Augment flow in G by b**
 -  If $e = (u, v)$ is a forward edge then increase $f(e)$ in G by b
 -  Else (u, v) is a backward edge, and let $e = (v, u)$ be in G decrease $f(e)$ in G by b
 - Repeat until you get stuck.



Example

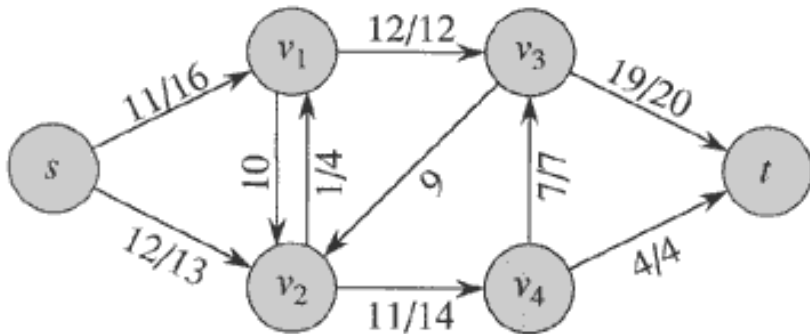


G at some intermediate step.
Flow achieved is 19

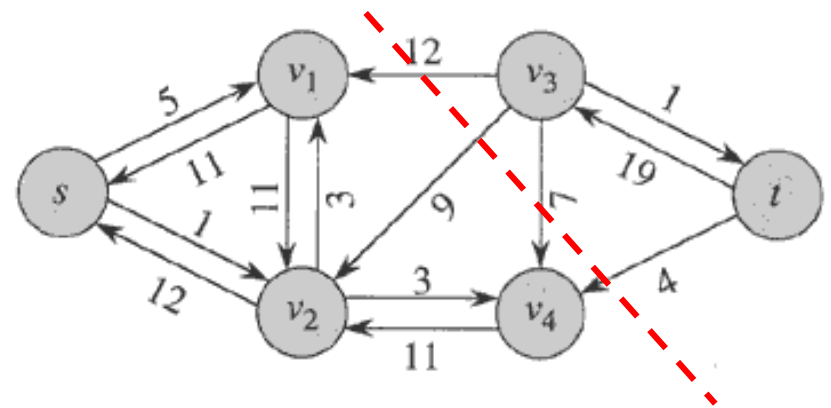


Residual Graph has P with $b = 4$

No more augmenting s - t paths \rightarrow max flow attained.

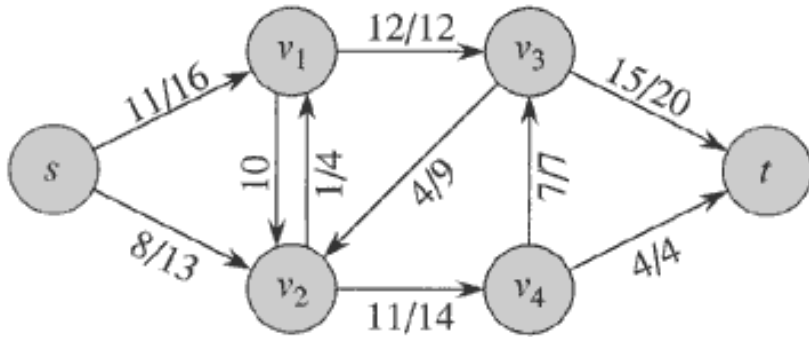


$\text{Val}(f) = 23$

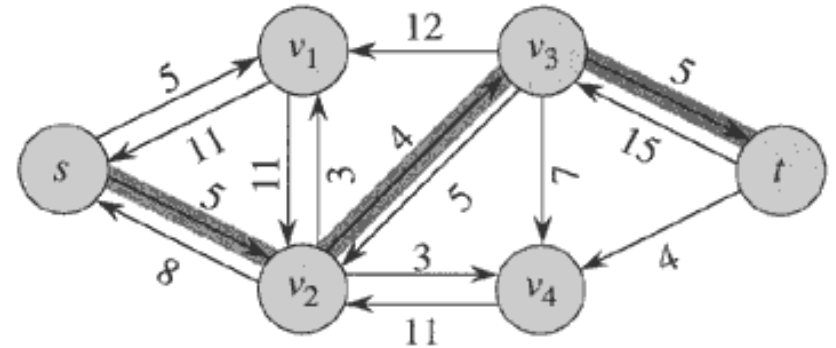


Stuck, no P possible

Example

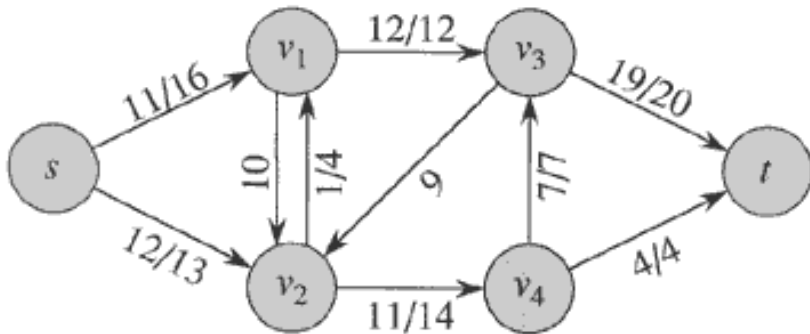


G at some intermediate step.
Flow achieved is 19

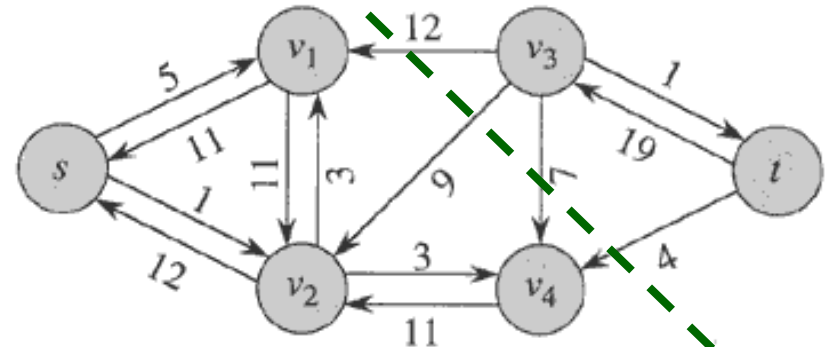


Residual Graph has P with $b = 4$

How to verify the claim: No more augmenting s - t paths \rightarrow max flow attained.



$Val(f) = 23$



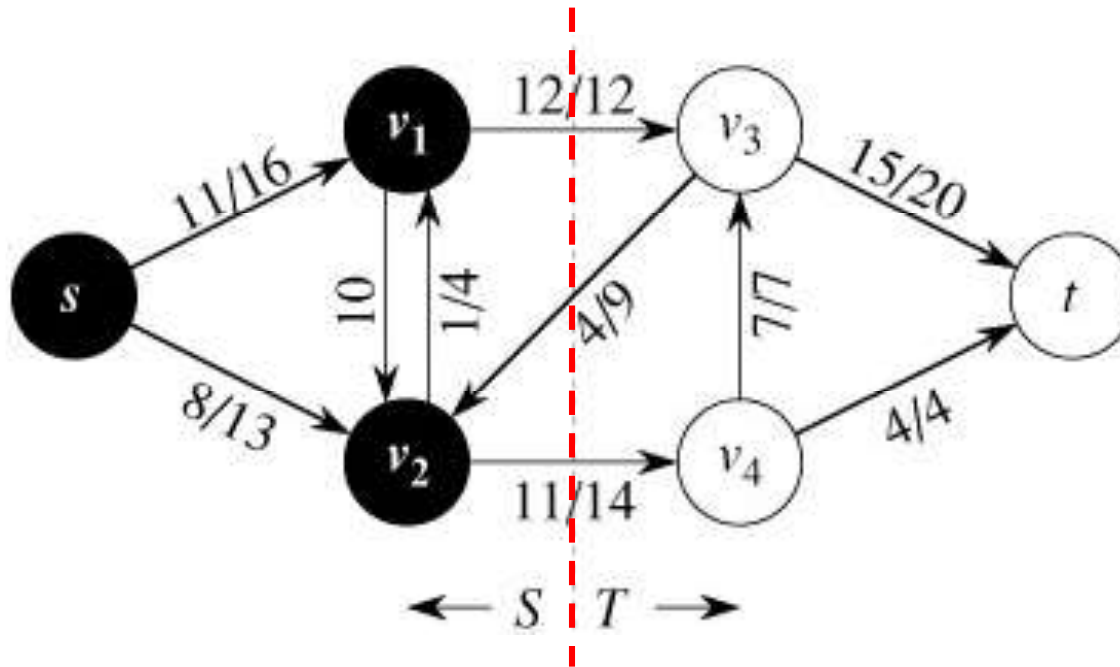
Stuck, no P possible

Answer lies in: Cut

Is Ford-Fulkerson algorithm correct?

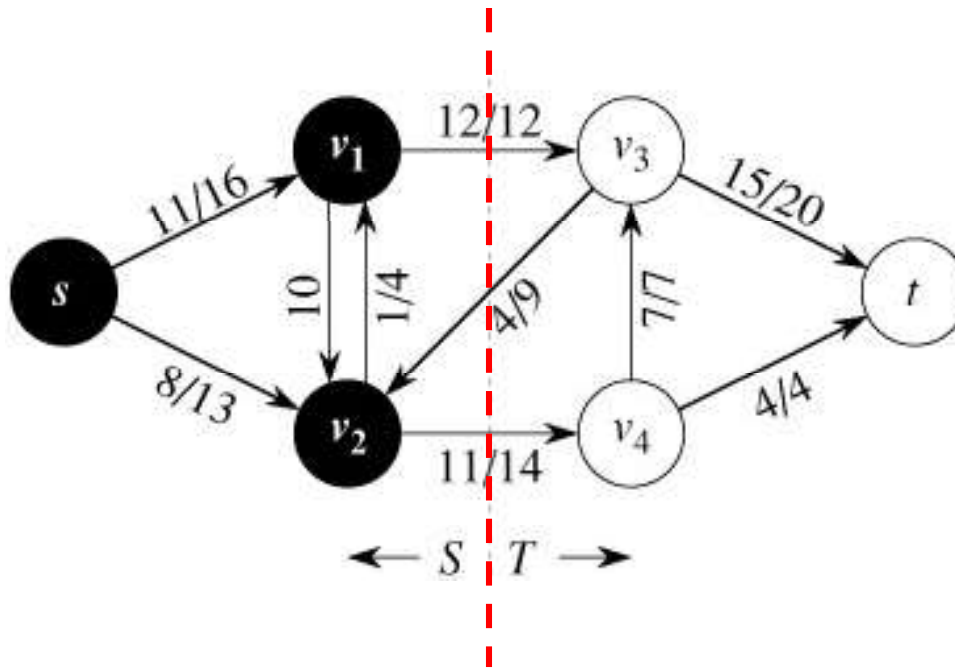
Cuts of Flow Networks

A **cut** (S, T) of a flow network is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.



The Capacity of a Cut (S,T)

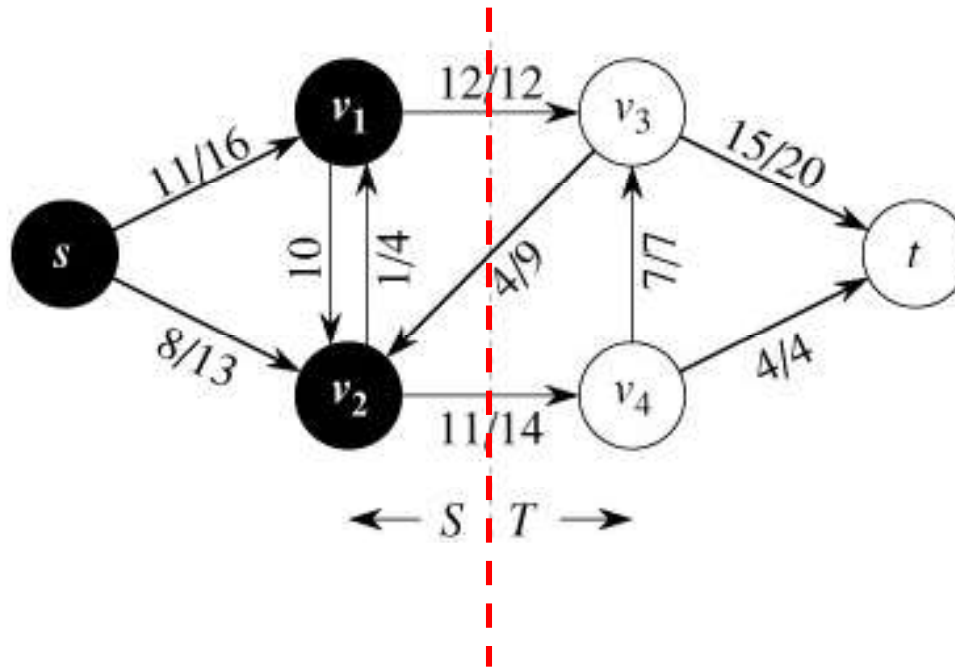
$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$



Maximum possible flow through the cut(S, T):
Sum of capacities $c(u,v)$, where $u \in S$ and $v \in T$
 $c(S,T) = 12 + 14 = 26$

The Net Flow through a Cut (S,T)

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v) - \sum_{v \in T, u \in S} f(v,u)$$

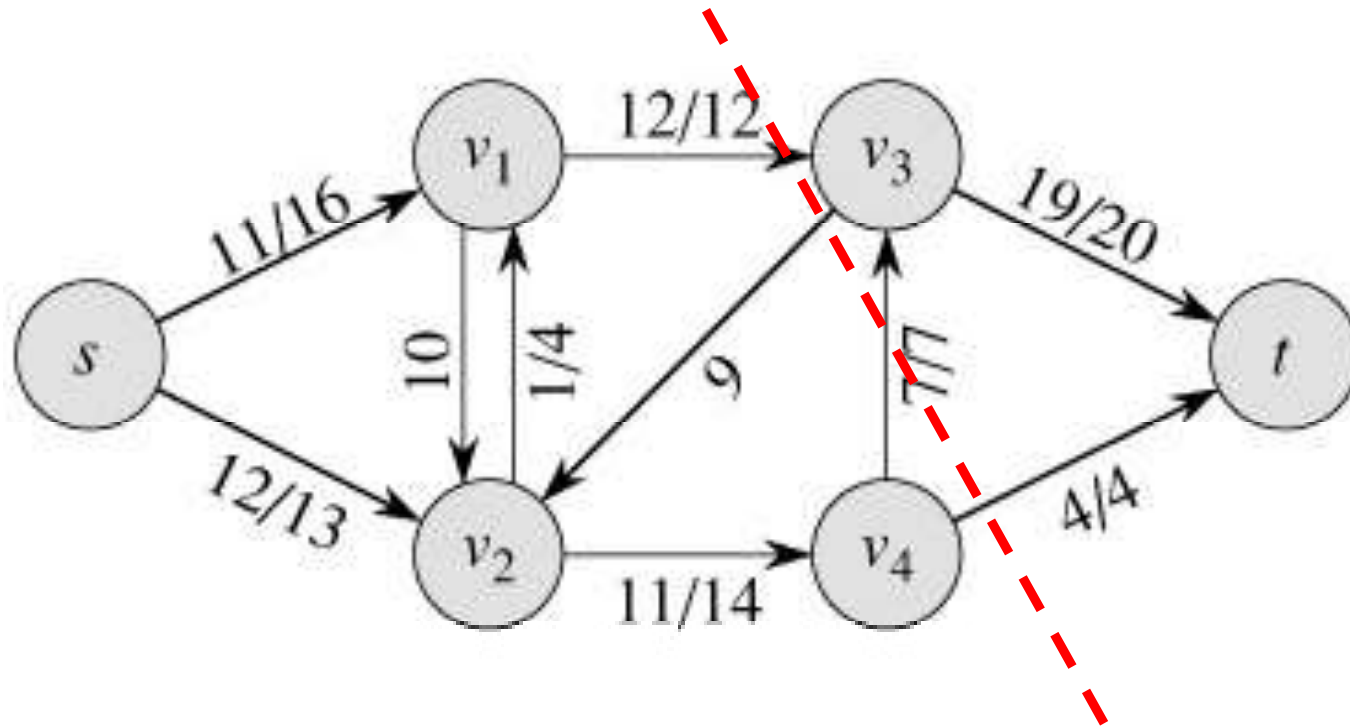


Net flow through a cut = Flow out - Flow in
 $f(S,T) = 12 + 11 - 4 = 19$

Bounding the Network Flow

The **value** of any **flow** f in a flow network G is **bounded from above** by the **capacity** of any **cut** (S, T) of G .

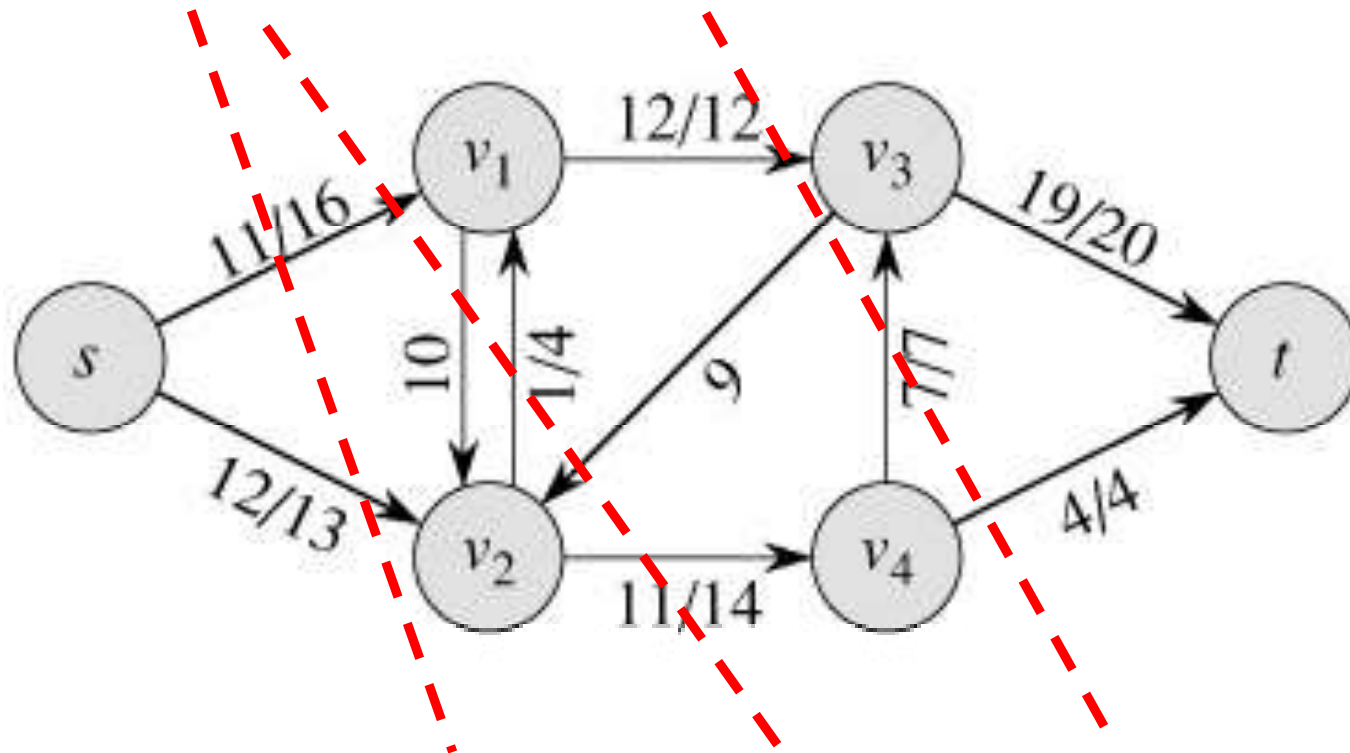
- Follows from Capacity constraint



Net Flow of a Network

The net flow value across any cut(S , T) is the same and equal to the flow value of the network.

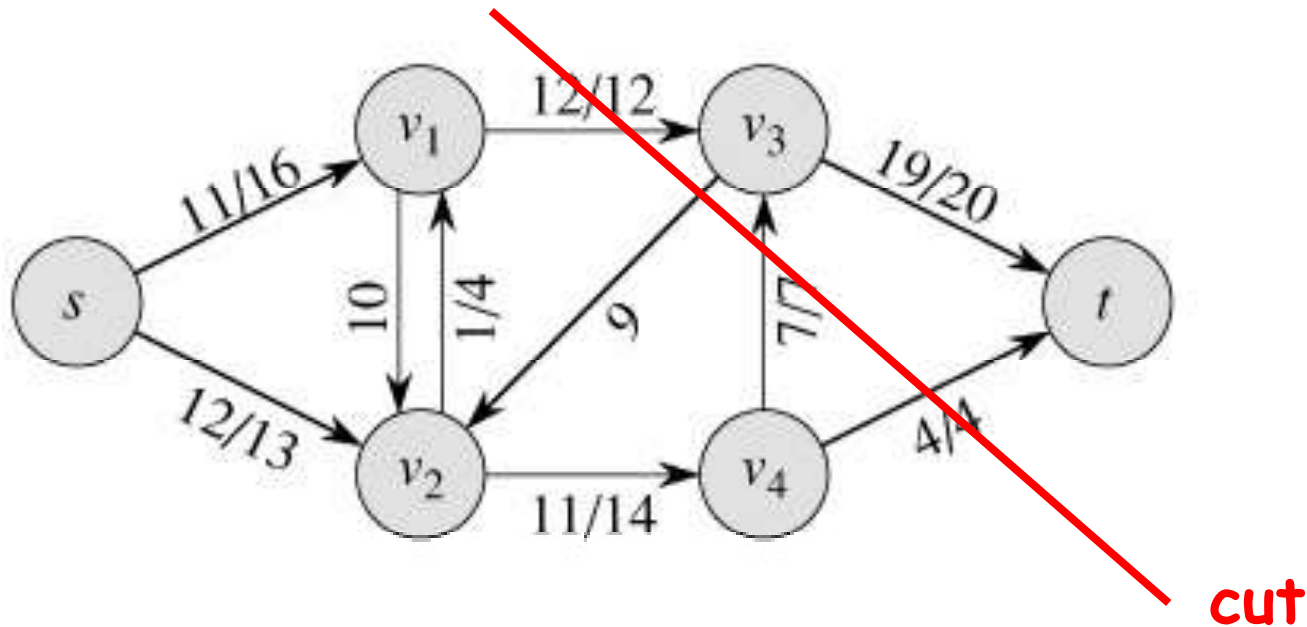
- Follows from Conservation constraint



Min-Cut Problem

Capacity of the cut

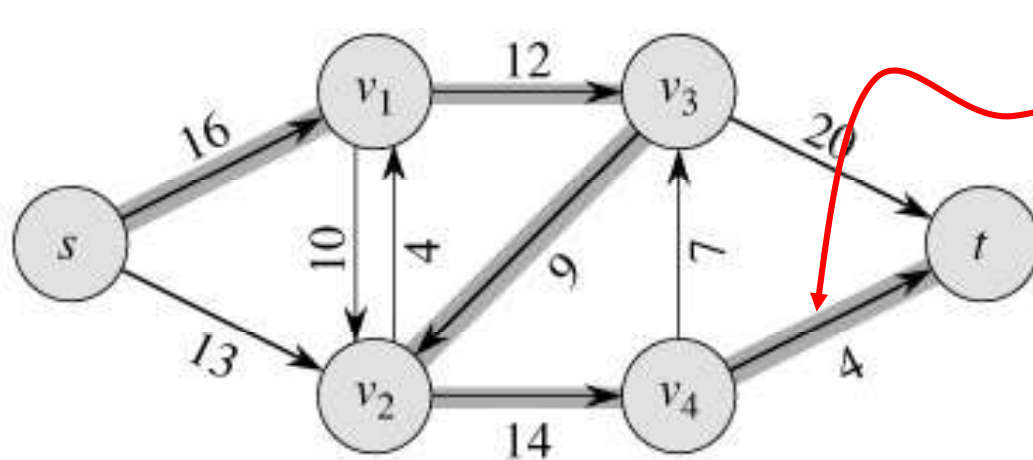
= maximum possible flow through the cut
= $12 + 7 + 4 = 23$



- The network has a capacity of **at most** 23.
- Here, the network **does** have a **capacity of 23**, because this is a **minimum cut**.
- How to find a cut(S , T) of minimum capacity?

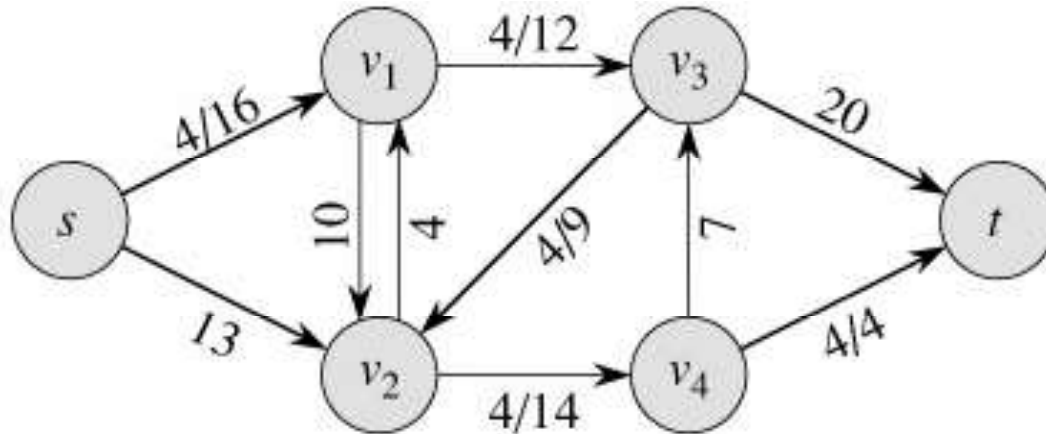
➤ Ford-Fulkerson

Example: Max-Flow Min-Cut



augmenting path

Original Network

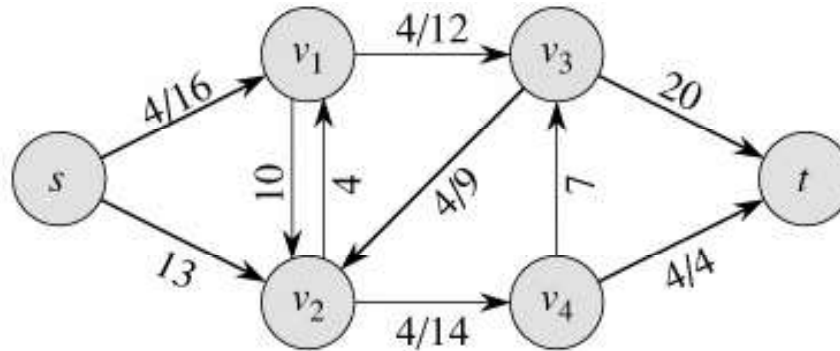


Flow Network

Resulting Flow = 4

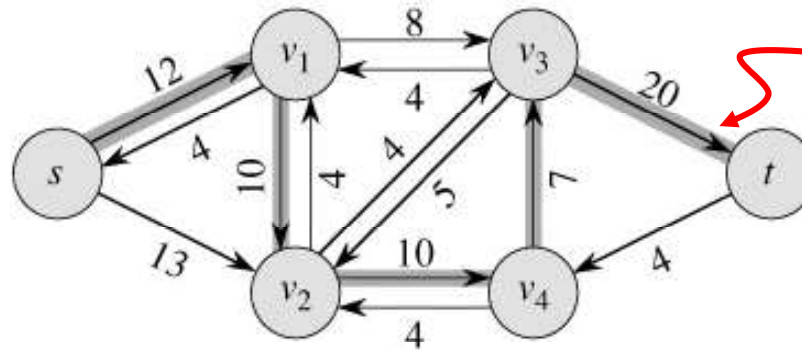
Example: Max-Flow Min-Cut

Flow Network



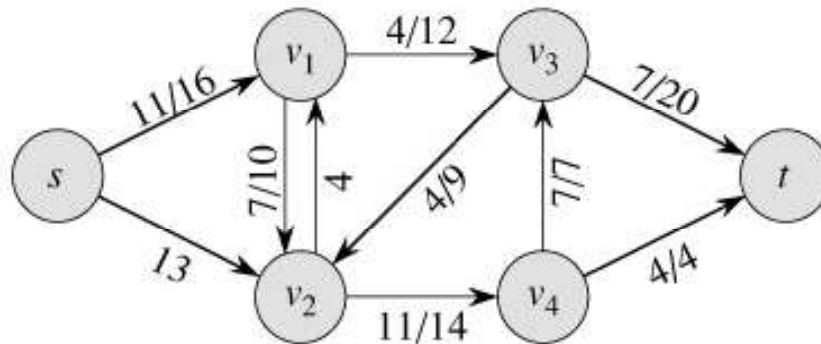
Resulting Flow = 4

Residual Network



augmenting path

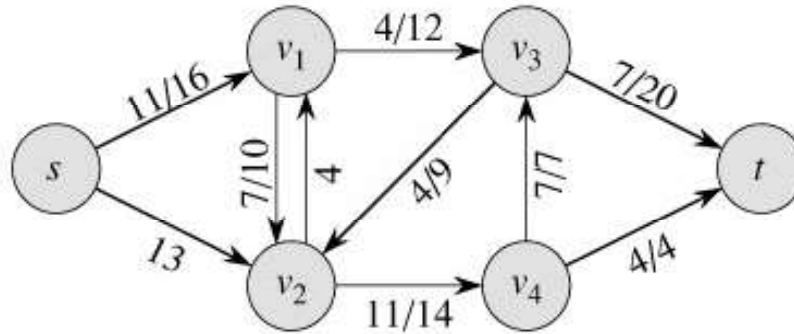
Flow Network



Resulting Flow = 11

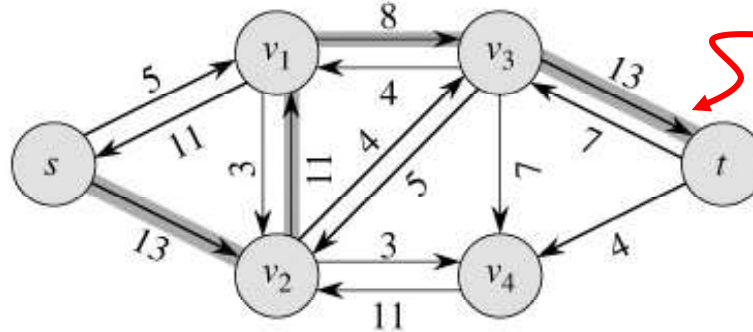
Example: Max-Flow Min-Cut

Flow Network



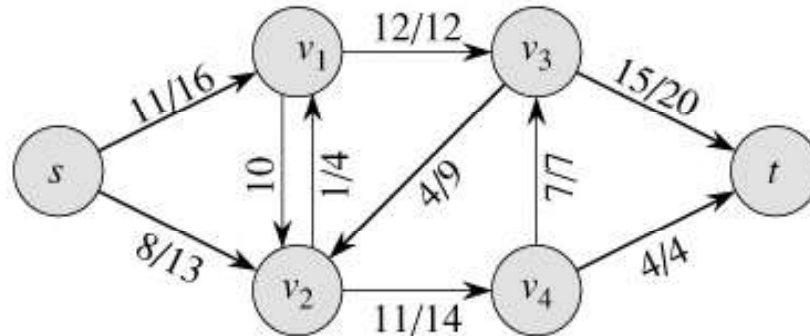
Resulting Flow = 11

Residual Network



augmenting path

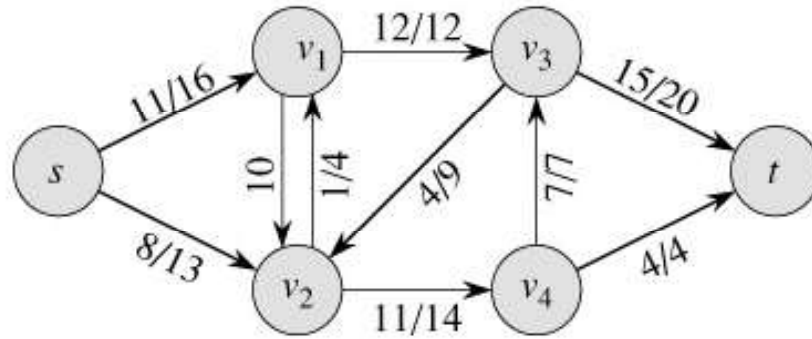
Flow Network



Resulting Flow = 19

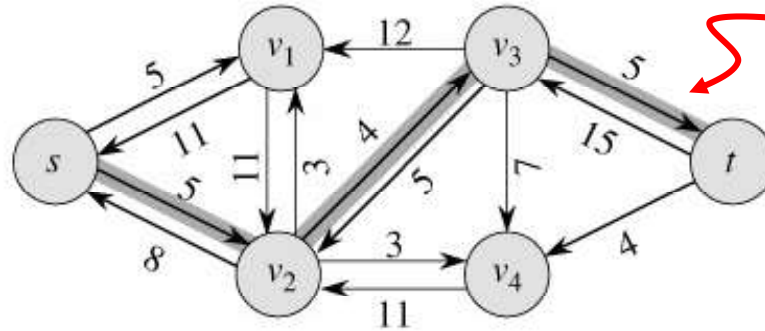
Example: Max-Flow Min-Cut

Flow Network

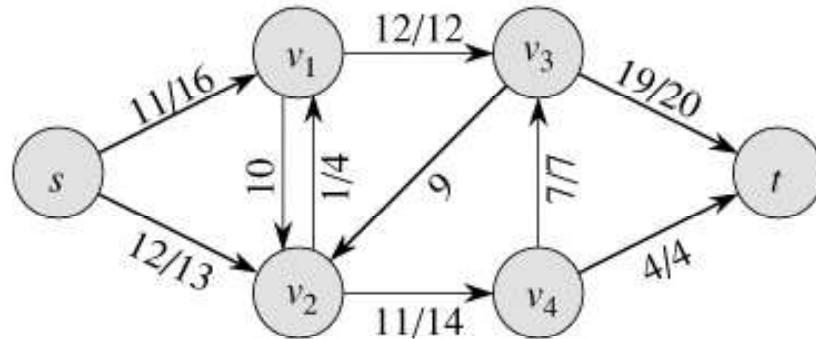


Resulting Flow = 19

Residual Network

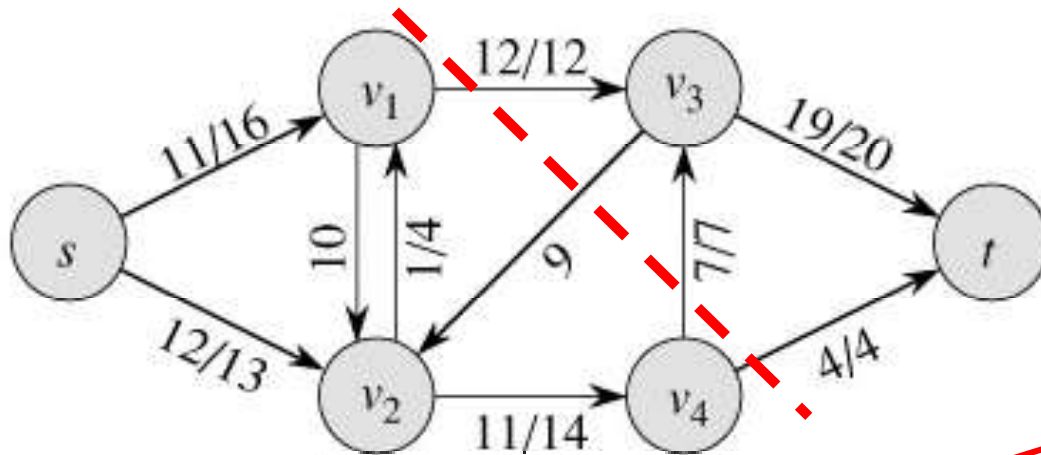


Flow Network



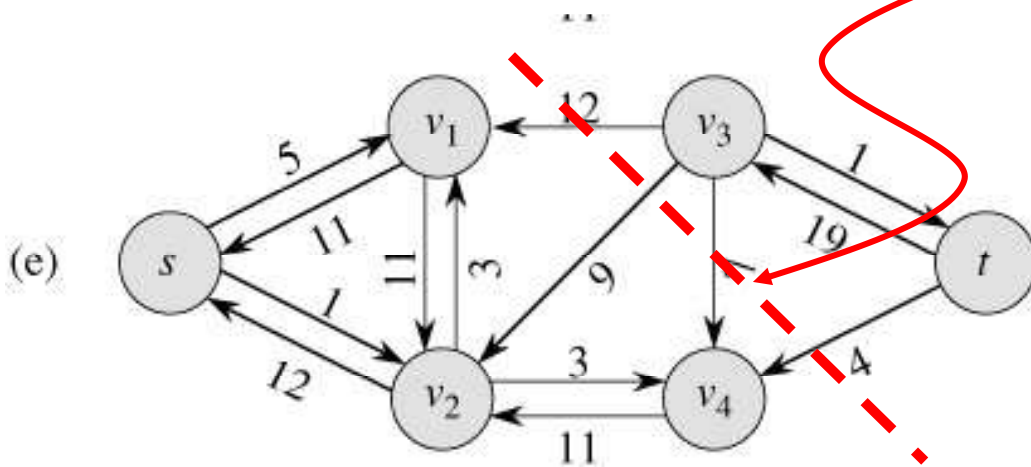
Resulting Flow = 23

Example: Max-Flow Min-Cut



Resulting
Flow = 23

No augmenting path:
Maxflow=23



Residual Network

Max-Flow Min-Cut Theorem

Let f is a flow in a flow network $G=(V,E)$, with source s and sink t :

1. Since $\text{val}(f) \leq c(S,T)$ for all cuts of (S,T) where $s \in S$ and $t \in T$.
If $\text{val}(f) = c(S,T)$ then $c(S,T)$ must be the **min-cut** of G .
2. This implies that f is a maximum flow of G .
3. This implies that G_f contains no augmenting paths.
 - If there were augmenting paths this would contradict that we found the maximum flow of G

$1 \rightarrow 2 \rightarrow 3 \rightarrow 1$... and from $2 \rightarrow 3$ we have that the **Ford Fulkerson** method finds the maximum flow if the residual graph has no augmenting paths.

Hence, **Max-flow min-cut theorem**:

max-flow = min-cut.

Max-Flow = Min-Cut

How to find out the min-cut:

When Ford-Fulkerson stops, **no directed s - t paths exists in G_f**

- $S =$ all v such that s - v path exists in G_f
- $T = V \setminus S$

Clearly (S, T) is a cut.

It is also minimum because:

For all $(u, v) \in G$, $u \in S$ and $v \in T$.

- $f(e) = c_e$
- All available capacity is used.

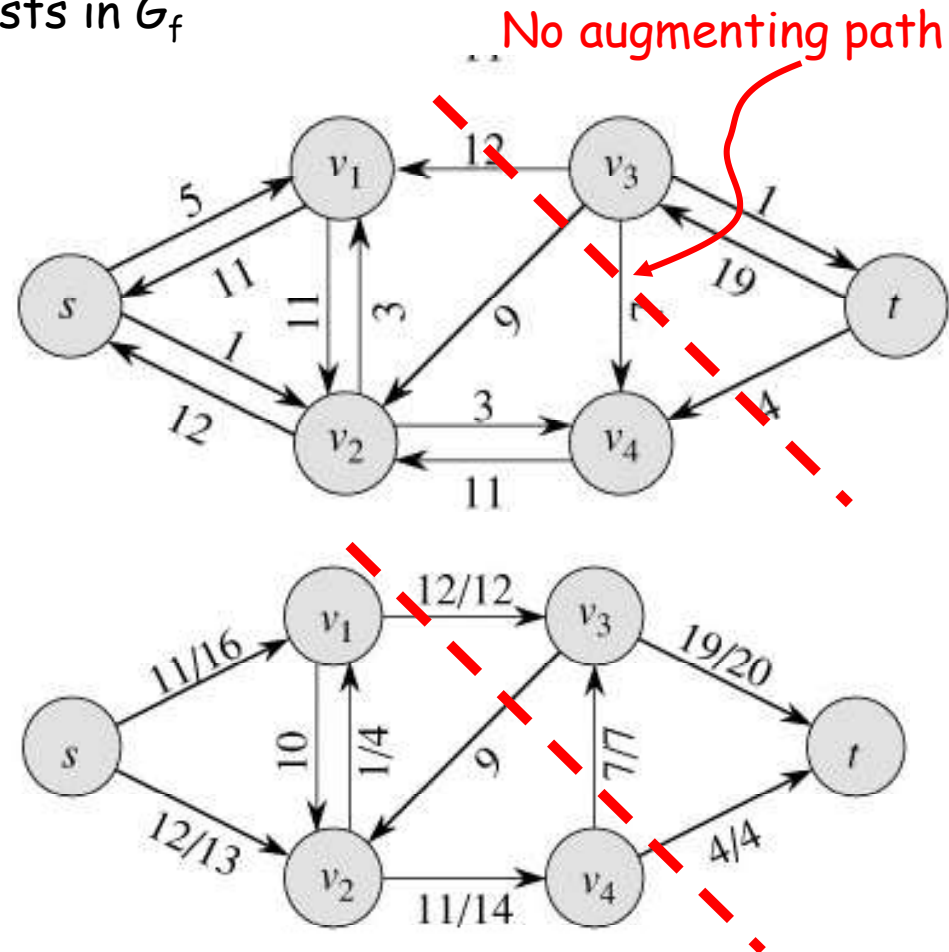
For all $(v, u) \in G$, $v \in T$ and $u \in S$.

- $f(e) = 0$

By the conservation constraint:

The **net flow** across any **cut** (S, T) is the **same** and **equal** to the flow of the network.

$$\Rightarrow \text{val}(f) = C(S, T).$$



Analysis

Ford-Fulkerson's Max-Flow

Initially $f(e) = 0$ for all e in G } $O(m)$

While there is an s - t path in the residual graph G_f ?

Let P be a simple s - t path in G_f

$f' = \text{augment}(f, P)$

Update f to be f'

Update the residual graph G_f to be $G_{f'}$

$O(m)$

Endwhile

Return f

Analysis

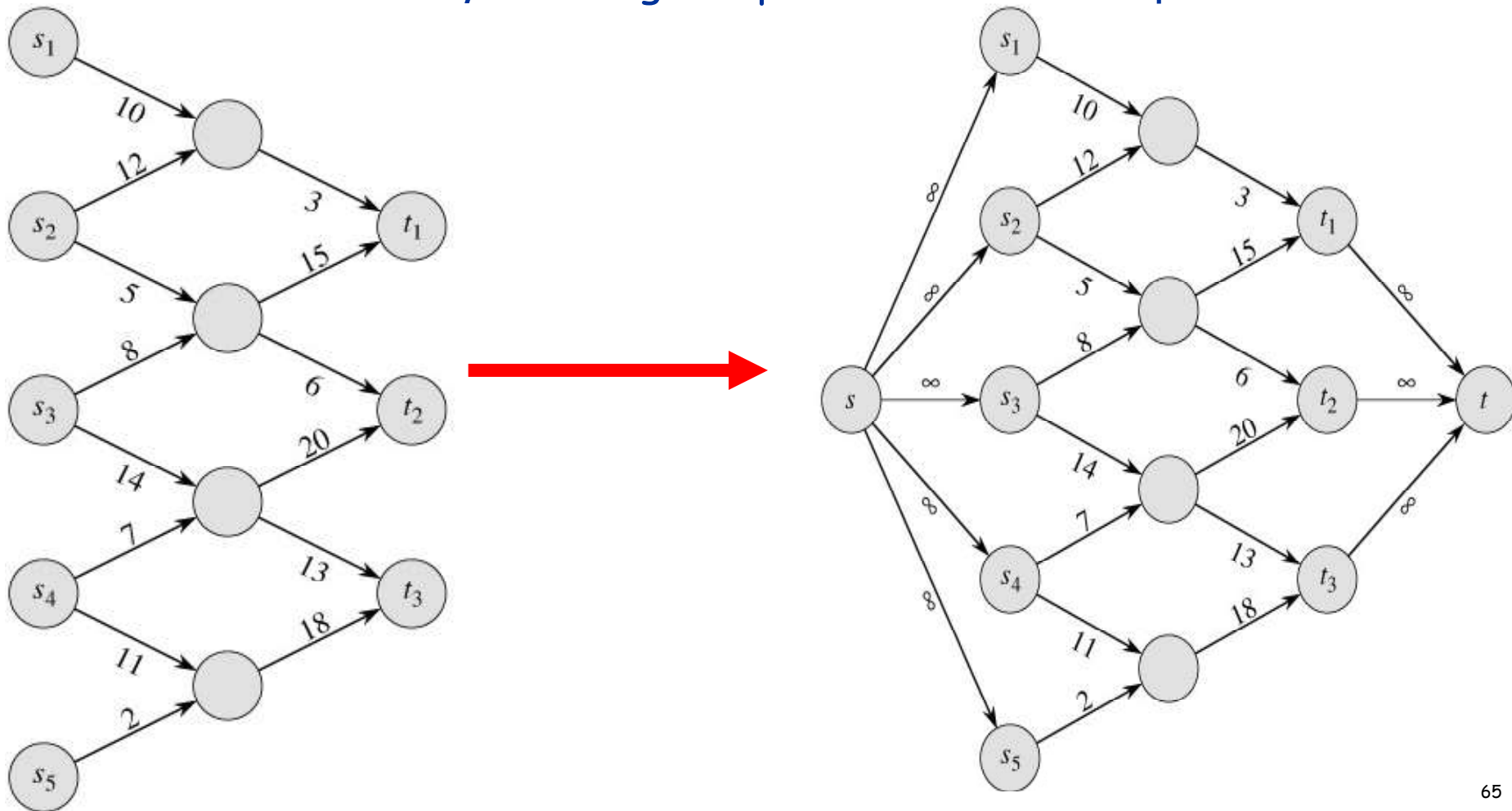
- Let $C = \sum_{e \text{ out of } s} f(e)$.
- Then the time can be written as: $O(m C)$
 - In worst case, each iteration of the while loop increments flow by 1
- Note that this running time is **not polynomial** in input size due to similar reasons as Knapsack
- Curious to know more?
 - More in Algorithms Adv. Course.

Multiple Sources Network

We have several sources and several targets.

Want to maximize the total flow from all sources to all targets.

Reduce to max-flow by creating a super-source and a super-sink:

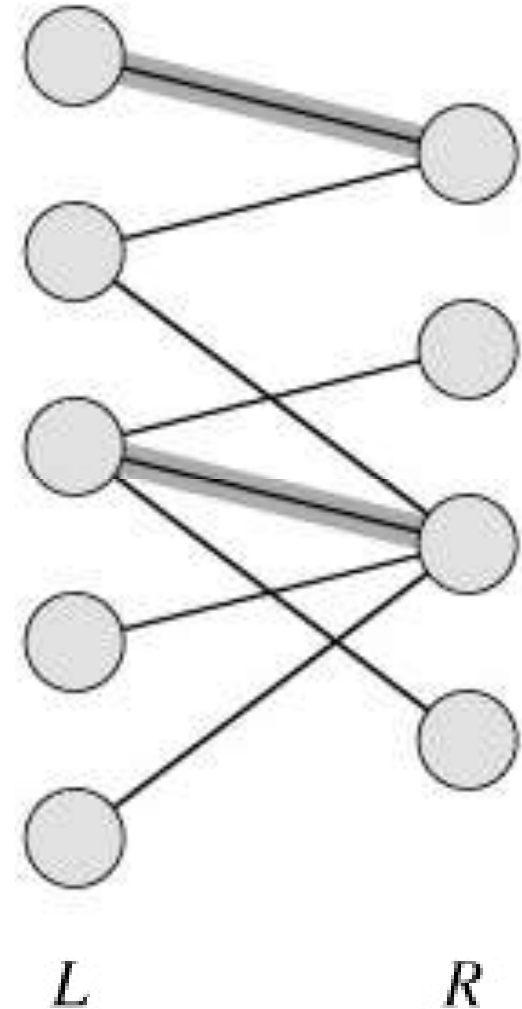


An Application of Max Flow:

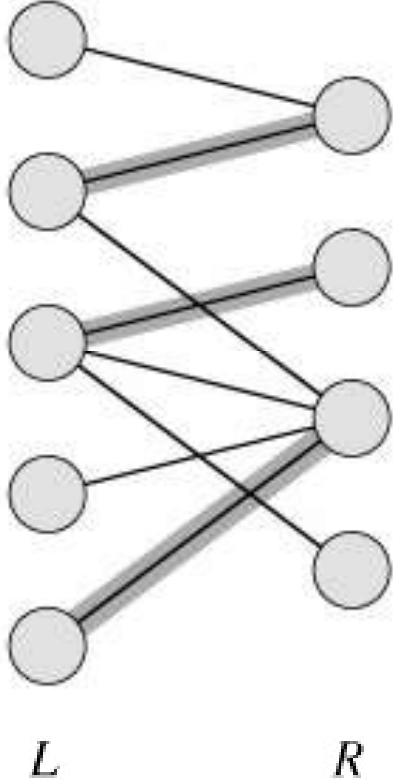
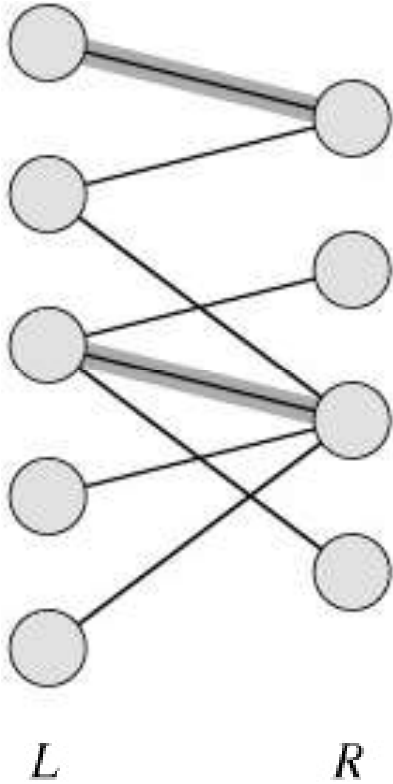
Maximum Bipartite Matching

Maximum Bipartite Matching

- A **bipartite graph** is a graph $G=(V,E)$ in which V can be divided into two parts L and R such that every edge in E is between a vertex in L and a vertex in R .
- e.g. vertices in L represent skilled **workers** and vertices in R represent **jobs**. An edge connects workers to jobs they can perform.

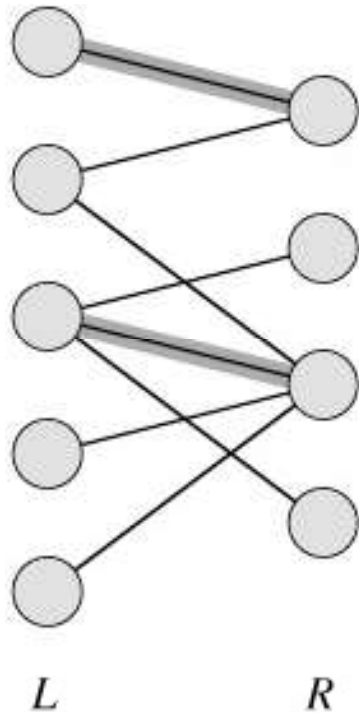


A **matching** in a graph is a subset M of E , such that for all vertices v in V , at most one edge of M is incident on v .

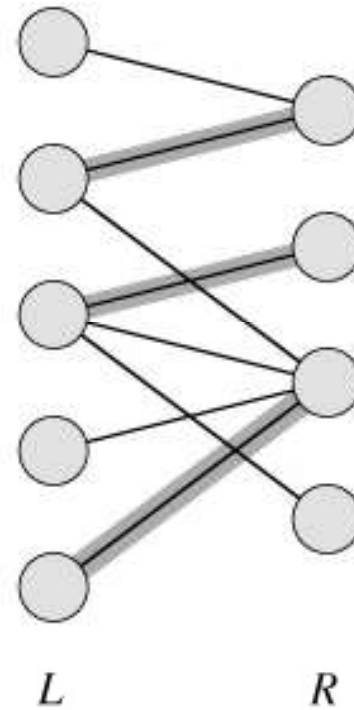


A **maximum matching** is a matching of maximum cardinality (maximum number of edges).

not maximum



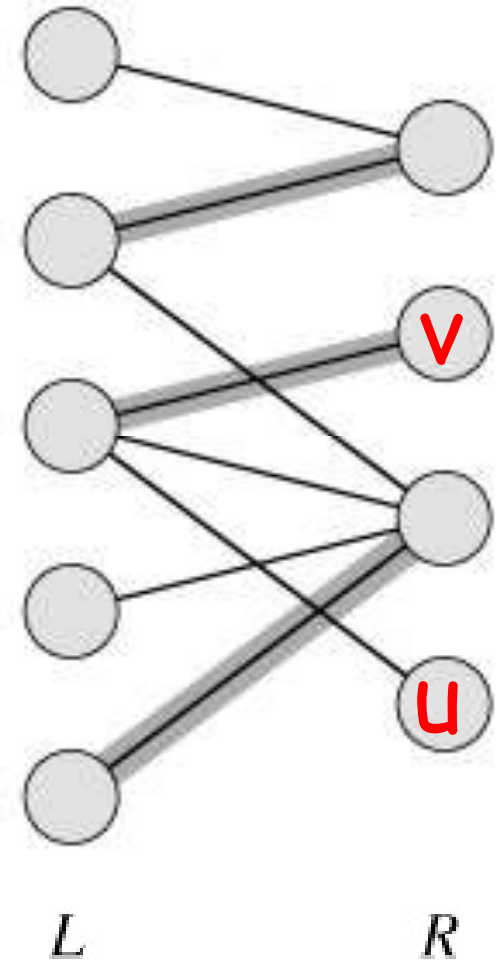
maximum



A Maximum Matching

No matching of cardinality 4, because only one of v and u can be matched.

In the **workers-jobs** example a max-matching provides work for as many people as possible.



Solving the Maximum Bipartite Matching Problem

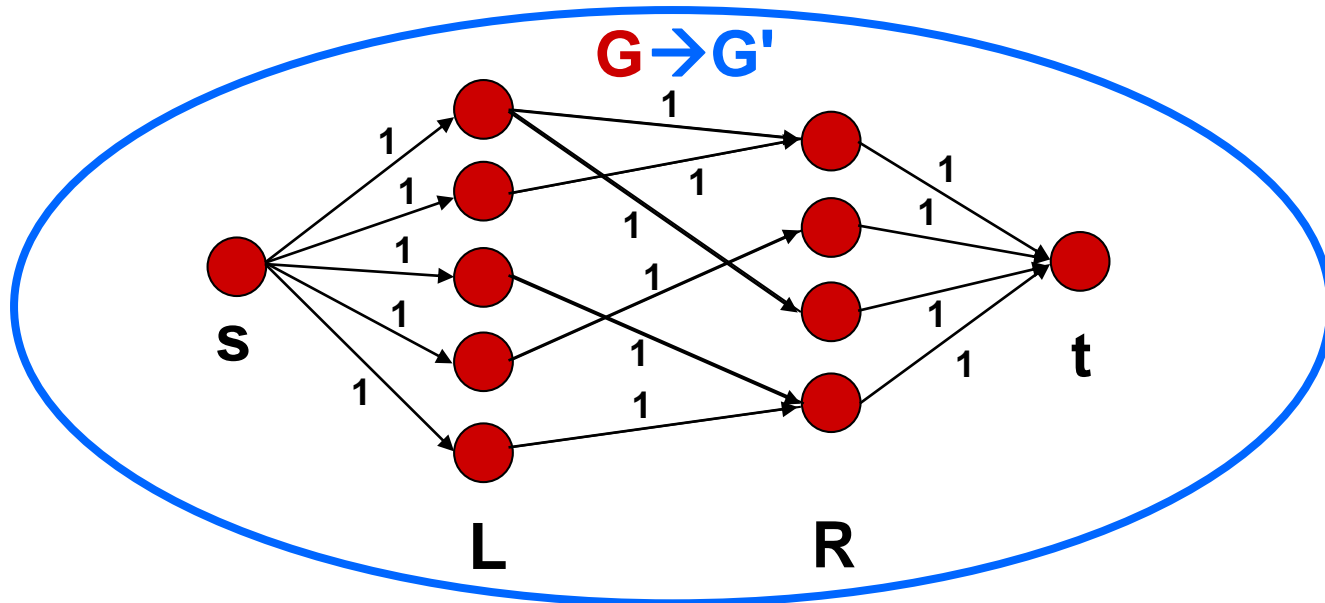
- Reduce the maximum bipartite matching problem on graph G to the max-flow problem on a corresponding flow network G' .
 - Solve using Ford-Fulkerson method.

Corresponding Flow Network

To form the corresponding flow network G' of the bipartite graph G :

- Add a source vertex s and edges from s to L .
- Direct the edges in E from L to R .
- Add a sink vertex t and edges from R to t .
- Assign a capacity of 1 to all edges.

Claim: max-flow in G' corresponds to a max-bipartite-matching on G .



Solving Bipartite Matching as Max Flow

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$.

Let $G' = (V', E')$ be its corresponding flow network.

If M is a matching in G ,

then there is an integer-valued flow f in G' with value $|f| = |M|$.

Conversely if f is an integer-valued flow in G' ,

then there is a matching M in G with cardinality $|M| = |f|$.

Thus $\max |M| = \max(\text{integer } |f|)$

Does this mean that $\max |f| = \max |M|$?

Problem: we haven't shown that the max flow $f(u,v)$ is necessarily integer-valued.

Integrality Theorem

If the capacity function c takes on only integral values, then:

1. The maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is integer-valued.
2. For all vertices u and v the value $f(u,v)$ of the flow is an integer.

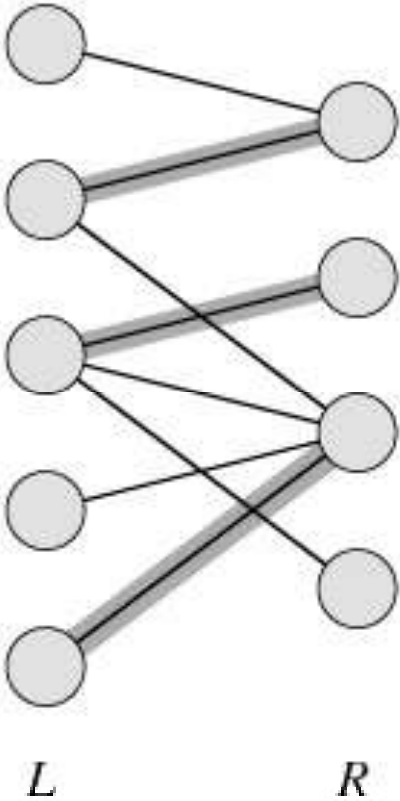
So, $\max |M| = \max |f|$

Running Time:

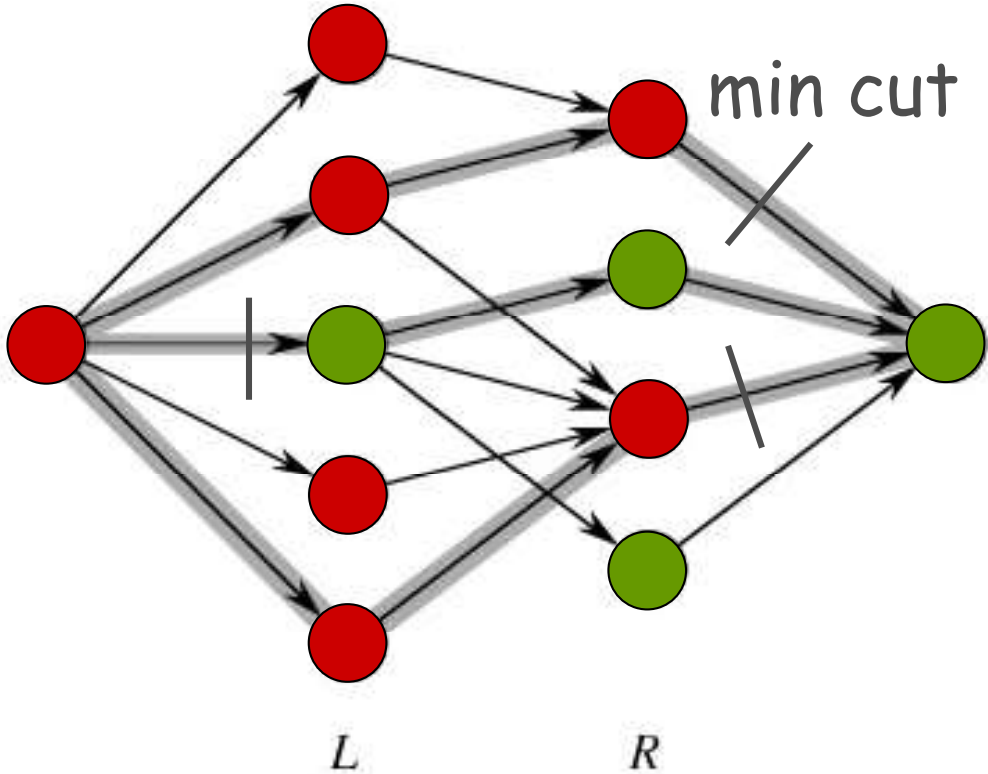
Remember: $C = \sum_{e \text{ out of } s} f(e)$.

Here C is at most n ,
Hence $O(m C)$ becomes $O(mn)$

Example



$|M| = 3$



$\max \text{ flow} = |f| = 3$



Conclusion

- Network flow algorithms allow us to find the maximum bipartite matching fairly easily.
- Similar techniques are applicable in other combinatorial design problems.

- Remark about Exercise 3 from week 2:

- "witches watching watches"
- It was asked to provide a counterexample for the given greedy algorithm
- Bipartite Matching exactly solves the same
 - ✓ Now you have an algorithm for "witches watching watches"

Wednesday, LV 7 (2016-10-12)

DP Shortest Path

Given a grid with obstacles, we wish to find the shortest path from the upper left corner $(0,0)$ to (n,n) .

Shortest path problem in a graph

One way to solve the problem is by constructing an undirected graph $G=(V,E)$.

Let $V = \{(i,j) \mid i,j \text{ integers, } 0 \leq i \leq n, 0 \leq j \leq n\}$ and $E = A \cup D$ where

$A = \{((i,j)(i',j')) \mid |i-i'|+|j-j'|=1\}$ is the set of vertical/horizontal edges and

$U = \{((i,j)(i',j')) \mid |i-i'|=1 \wedge |j-j'|=1\}$ is the set of diagonal edges

Let l_e be the length of an edge:

$$l_e := 1 \text{ if } e \in A$$

$$l_e := \sqrt{2} \text{ if } e \in D \text{ and square } (i,j) \text{ is not blocked}$$

$$l_e := \infty \text{ if } e \in D \text{ and square } (i,j) \text{ is blocked}$$

The paths in G are exactly the path in the grid.

The graph construction needs $\mathcal{O}(n^2)$ time. G has $\mathcal{O}(n^2)$ nodes and $\mathcal{O}(n^2)$ edges.

If we apply Dijkstra's algorithm we can solve this shortest path problem in $\mathcal{O}(n^2 * \log(n))$. However, if we design the graph a bit more cleverly, we can do better. If we create a directed graph and do not add the edges that goes backwards (it will never be beneficial to go backwards in a shortest path). We have then created a DAG and shortest paths on DAGs are quicker ($\mathcal{O}(n)$).

Using DP

In the exam problem a function was already given: $OPT(i,j) := \text{length of a shortest path from } (0,0) \text{ to } (i,j)$. (If such a function is *not* given, it is important that you write this yourself!)

Let $b(i,j)$ denote a blocked square, and $b(i,j) := \begin{cases} \infty & \text{square } (i,j) \text{ is blocked} \\ \sqrt{2} & \text{else} \end{cases}$

$$\text{OPT}(i,j) = \min \begin{cases} \text{OPT}(i-1,j)+1 & \text{horizontal} \\ \text{OPT}(i,j-1)+1 & \text{vertical} \\ \text{OPT}(i-1,j-1) + b(i,j) & \text{diagonal} \end{cases}$$

Implementation, not required for the exam but we need to argue the time complexity so it's convenient! Also, you should always use for loops and memoization instead of recursive calls.

```

for i = 0 to n:
    OPT(i,0) = i
for j = 0 to n:
    OPT(0,j) = j

for i = 1 to n:
    for j = 1 to n:
        OPT(i,j) = ...

```

The solution (the path and it's length) is given by backtracing!

Graphic TSP

Given as input an undirected graph $G = (V,E)$ with equal-length edges (distance between two nodes, u and v , $\equiv d(u,v)$) we want to find a shortest tour that visits every node at least once and starts and ends in the same node.

We write the problem as a decision problem: Given a graph G and a number t we ask: "Is there a tour of length at most t ?".

Hamiltonian Cycle

$G = (V,E)$ is an undirected graph and the HCP asks if there is a tour that visits every node exactly once, and starts and ends in the same node.

HCP is a known \mathcal{NP} .

Graphic TSP $\in \mathcal{NP}$?

Given a tour T , can we verify that:

- T contains all nodes
- Consecutive nodes on T are joined by edges
- The total length of T is at most t

in polynomial time?

Yes we can!

Hamiltonian Cycle \leq_p Graphic TSP

A Hamiltonian Cycle is a tour of size n (the number of nodes). We want a function f s.t. $f(G) = (H,t)$ s.t. G has a Hamiltonian Cycle $\Leftrightarrow H$ has a tour of length at most t . In our instance $H := G$ and $t := n$.

Graphic TSP $\in \mathcal{NP}$, HCP is \mathcal{NPC} and $\text{HCP} \leq_p \text{Graphic TSP}$, hence Graphic TSP is \mathcal{NPC} !

Outlook