...rove a greedy algorithm by taking an opt solution, transform it to get another opt solution that is "one step closer" to the solution our algorithm would produce. By induction we reach the algorithm's solution.

A numeric algorithm runs in pseudopolynomial time if its running time is polynomial in the numeric value but is exponential in the length of the input. Ex: Checking if n is prime by dividing n with $\{1,2,\dots\sqrt{n}\}$ yields an integer result.

Time complexity for Div & Conq

n: instance size

a: # sub instances

b: divisor

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

$a > b^k: O(n^{\log_b a})$

$a = b^k: O(n^k \log n)$

$a < b^k: O(n^k)$

if $cn^k \notin P \Rightarrow alg \notin P$

$X \leq_p Y$

X is reducable to Y if an algorithm for Y can be used to solve X, after a polynomial time manipulation of X's instance. If this works, solving X cannot be more difficult than solving Y.

To prove $Y \in NPC$ we must prove that $Y \in NP$ and that $Y \in NP\text{-Hard}$

$Y \in NP \Rightarrow$ We can verify a solution of Y in polynomial time.

$Y \in NP\text{-Hard}$: Take a known NPC problem X. Show $X \leq_p Y$ and that the reduction is polynomial and correct.

Every problem in NP is reducible to *Y
polynomial time

DAG

1. Count #incoming edges for each node, $O(n)$
2. Put all nodes w indegree 0 in a queue.
3. While queue is nonempty: take the next node u and subract 1 from all indegrees w v s.t. $(u,v) \in E$. $O(m)$

BFS

Adj. matrix: $O(n^2)$, Adj. list: $O(m)$ $m \geq n-1$

Implement w queue! All edges in BFS goes from one layer to next layer.

Shortest Paths in Dags

- Let d be an array w the same length as V, with $d[s]=0$ and $d[u]=\infty$ for all $u \neq s$.
- Let p be an array of length V, initialized to null.
- Starting from s, loop over all vertices u in V.
  - For each vertex v following u:
    - Let w = the weight of edge (u,w)
    - Relax the edge: if: $d[v] > d[u]+w$:
      $d[v] = d[u]+w$
      $P[v] = u$

DFS

Adj. list: $O(m)$

Undirected graph $\Rightarrow$ No cross-/forward edges.

Applications

- Both reaches only the nodes connected to S. $\Rightarrow$ $O(m)$ to test if graph is connected.
- By restarting our search we can find all connected: $O(m+n)$
- $O(m)$ to see if G is strongly connected. Run once on G w edges (u,v) and once on G w (v,u). If the same nodes are reached, G is strongly connected.
- Run DFS from every node in G. Iff the root has more than one child it is an articulation point: $O(mn)$.
- Determine if G is bipartite. BFS $\Rightarrow$ L_i gets C_1 L_{i+1} C_2.

Reductions

Let $G=(V,E)$ be an undirected graph.

Clique $\leq_p$ Ind. Set : $f(G,k) = \bar{G}, k$ where $\bar{G}$ is
Indp.set $\leq_p$ Clique : the complement of G.

VertexCover $\leq_p$ Ind Set: $C \subseteq V$ is a vertex cover iff
Ind.set $\leq_p$ Vertex cover $V \setminus C$ is an independent set.
$f(G,k) = G, n-k$.

QuickSort: Choose p as a pivot, put all elems $< p$ in one subset and vice versa. Recursiveley sort the subsets and concatenate putting p between.

Median finding: Choose random splitter s and compare all elements to s to get rank r of s: $O(n)$.
If $r > k$ repeat on list$[0, s-1]$
$r < k$ repeat on list$[s+1, n]$ w/ $k = k-r$
(if $k = \frac{n}{2}$ this gives the median, else elem of rank k).
$O(n^2)$ but $O(n)$ if good pivot.

Relaxed inversion

Split the array into two sub-arrays. The recursion should report the number of inversion in each sub-array and sort it. Inv is #inversions in a subarray. If $B[i] > 4C[i]$ increase inv by # remaining elem in B $(n/2-(i-1))$. If $B[i] \leq C[i]$ append $B[i]$ to $\bar{A}$ and increase i++. If $B[i] > C[i]$ —||— $C[i]$ ——— ||—— j++. If i or $j > n/2$ append the rest of the other array.

Interval scheduling $\leq_p$ Inep Set: Draw a graph where each interval is a vertex and an edge is drawn between two vertices iff their intervals intersects. (The opposite cannot be shown, circle graph.)

Interval Partitioning

Sort the intervals by start time.

for all i : $X_i = \emptyset$

for all $j = 1$ to n:

put $[s_j, f_j]$ in X_i with smallest i s.t it does not intersect any other elem i X_i.

Clique $\leq_p$ Subgraph Isomorphism (is H subgraph of G?)

Given a graph and number k we ask if G contains a clique of size k. Hence ~~EX~~ $f(G,k) = (G,H)$ (H is clique of size k). H is exp larger than k, this is polynomial time reduction because we consider the instance as a whole.

Median finding 2 sorted arrays

if $n==1$ return median of 3 elems

else: $n1 = \lceil \frac{n}{2} \rceil$, $n2 = \lfloor \frac{n}{2} \rfloor$

if $A[a+n1] > B[b+n1]$: return med$(a+n1, b, n2)$

else $A[a+n1] < B[b+n1]$ —||— $(a, b+n1, n2)$

call med$(1,1,n)$

Segment prob/Sequence partitioning (lect 5) $OPT(j) = \min_i(OPT(i-1)) + e_{ij}$

Alg speed vs. computer speed

If the speed of computer doubles...
If it takes t time to run an $O(n!)$ algorithm; $t = \frac{n!}{s}$. If t is fixed we can see how the speed allows for n to change as: $t = \frac{n!}{s} = \frac{m!}{as}$ where a is an inc/dec in speed. Solve for m to see possible inc.

Greedy Exchange

1. Label your algorithm's solution as optimal, A and another solution O as optimal too.
2. Compare greedy with optimal, assume they differ. Then:
   a) There is an element of O not in A and an element of A not in O, or
   b) There are two consequtive elements in O in a different order than they are in A.
3. Swap or exchange the elements in question in O to make it more similar to A (swap one element out and another in for the first case or swap the order in the second case) and argue that you have a solution no worse than before.
   If we continue swapping we can create $O=A$ w/o worsening quality → greedy algorithm optimal.

Note

- Don't assume $A \neq O$ ~~if you're not sure you~~
- You must argue why the 2 elems even exist out of order, or exist in O but not in A.
- Remember to argue multiple swaps!

Bottleneck in Graph from x to y

```
Path(G, x, y):
    S := {u}
    for w in V:
        d(w) := 0
    while S != V:
        d' = 0, w_next = null, w_curr = null
        for e = (w, w') in E ∩ S × (V \ S):
            t = min(d(w), c_e)
            if t > d':
                d' = t, w_next = w', w_curr = w
        S := S ∪ {w_next}, d(w_next) := d', p(w_next) := w_curr
    if d(u) = 0:
        return NOPATH
    else:
        return p


P(p, u, w):
    P_w = [w]
    while u != w:
        w := p(w), P_w := w :: P_w
    return P_w
```

Let $P_w = P(p,u,w)$ where $p = path(G,u,v)$

Shortening ...

Check w_{i-1}, w_i, w_{i+1} and as long as either i and $i-1$ or i and $i+1$ are equal we incr the required abbr-length.

Hamiltonian Path ≤p Min Bound-deg Spanning Tree

$G=(V,E)$ instance of HPP. In the reduction, use the same graph. Let $c(e)=1$, $b(v)=2$, total cost $= |V|-1$.

Cycles in graphs

If a connected graph has nodes with only even in degree it has a cycle. Start from arbitrary vertex and follow edges that have not been used. Eventually we will visit some node again since the graph is connected and has indeg ≥ 2.

Eulerian Tour

If G has a cycle C of length at least 2 contained in itself we can use it to prove that G has an euler tour.
Remove C from G. Let the remaining connected subgraphs be $G_1, \dots, G_k$. Each G_i still have even degree and has less edges than G, so the induction hypothesis states that it has an euler tour P_i. (Induction hyp: When there are less than k edges in the graph there is such an euler tour.)
C must have some vertex in G_i, choose a unique one (v_i), we can stitch the euler tour P_i and C together to form an euler path P for the entire graph G.
Tranverse the edges in C and add them into P one by one. If we visit some vertex v_i in G_i, then add the path p_i (starting and ending in v_i) into P. ⇒ Euler Path.
Algorithm: Recursion. Find cycle C in G, let $G_1, \dots, G_k$ be the remaining connected subgraphs. Find eulerian path P_i in G_i using recursion. Construct the eulerian path P using C and P_i.