Running Time and Growth

The *time complexity* $T(n)$ of an algorithm is the maximum amount of time taken on any input of size n.
$T(n) \in \mathcal{O}(g(n))$ if $\exists c > 0$ and $n_0 \ge 0$ s.t. $\forall n \ge n \Rightarrow T(n) \le c * g(n)$.
$T(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \ge 0$ s.t. $\forall n \ge n \Rightarrow T(n) \ge c * g(n)$.
$T(n) \in \Theta(g(n))$ if $T(n) \in \mathcal{O}(g(n))$ and $T(n) \in \Omega(g(n))$.
Order: $\mathcal{O}(\log_a n) < \mathcal{O}(n^d) < \mathcal{O}(r^n)$

$\mathcal{O}(\log n)$	binary search s. list
$\mathcal{O}(n)$	max element in list
$\mathcal{O}(n \log n)$	sort elements
$\mathcal{O}(n^2)$	find pair of closest
$\mathcal{O}(n^3)$	find disjoint sets
$\mathcal{O}(n^k)$	check $\exists e = (u,v)$
$\mathcal{O}(2^n)$	enum all subsets
$\mathcal{O}(n!)$	naive TSP

Graph Representation

Adjacency matrix: $n \times n$ matrix with $A_{uv} = 1$ if $(u,v) \in E$. Takes space n^2. If $(u,v) \in E$ takes $\Theta(1)$ time. Identifying all $e \in E$ takes $\Theta(n^2)$ time. *Adjacency list*: Node indexed array of lists. Takes space $m+n$. If $(u,v) \in E$ takes $\mathcal{O}(deg(u))$ time. Identifying all $e \in E$ takes $\Theta(m+n)$ time.

Greedy Algorithms

Greedy-choice property: globally optimal solution with locally optimal choices.

Interval Scheduling

Goal: Find maximum subset of mutually compatible jobs.

procedure IntervalScheduling
  Sort by finish time
  $A \leftarrow \emptyset$
  $f_{j*} = 0$
  for $i = 1$ to n **do**
    if $s_j \le f_{j*}$ **then**
      $A \leftarrow A \bigcup \{j\}$
      $f_{j*} = f_j$
  return A ▷ $\mathcal{O}(n \log n)$

Proof. Let $A : a_1, \ldots, a_k$ be the greedy solution and let $O : o_1, \ldots, m$ with $m \ge k$ be the optimal solution. We claim that $\forall r \le k \Rightarrow f(a_r) \le f(o_r)$. For $r = 1$, $f(a_1) \le f(o_1)$ by greedy choice. Suppose this holds for $r-1$: $f(a_{r-1}) \le f(o_{r-1})$. Then $f(o_{r-1}) \le s(o_r) \le f(o_r)$ by solution feasibility. Thus $f(a_{r-1}) \le s(o_r)$, so o_r was an option for greedy not picked. Since a_r was chosen instead, $f(a_r) \le f(o_r)$. Now assume $k < m$ for contradiction. Then $f(a_k) \le f(o_k) \le s(o_{k+1})$ becayse O is feasible, so o_{k+1} is an option for A, which contradicts $|A| = k$. ∎

Interval Partiton

Goal: Find minimum number of rooms to schedule all jobs without overlaps within a room.

procedure IntervalPartition
  Sort by start time
  $d \leftarrow 0$
  for $j = 1$ to n **do**
    if j compatible with class k **then**
      Add j to k
    else
      Add new class room $d+1$
      Add j to $d+1$
      $d \leftarrow d+1$ ▷ $\mathcal{O}(n \log n)$

Proof. Let d be the number of classrooms from greedy. The last room d is opened because job j was incompatible with $d-1$ rooms. These rooms have jobs ending after s_j. Thus we have d lectures overlapping at time $s_j + \epsilon$ and the depth is $\ge d$. But the number of rooms $\ge$ depth, so greedy must have $d \ge$ depth, so solution with $d =$ depth is optimal. ∎

Minimize Lateness

Goal: Schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

procedure MinLate
  Sort by increasing deadline
  $t \leftarrow 0$
  for $i = j$ to n **do**
    Assign j to $[t, t+t_j]$
    $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
  return Intervals $[s_j, f_j]$ ▷ $\mathcal{O}(n \log n)$

Def. An *inversion* is a pair of jobs i and j with $d_i < d_j$ but j scheduled before i.

Proof. Note that greedy has no idle time, $\exists O$ with no idle time, and greedy has no inversions. All schedules with no inversions and no idle time have the same lateness, because all jobs with same d come in a block of consecutive jobs if no inversions. Any reordering of these jobs with retain the last job having the lateness $f - d$. Let ℓ be the lateness before the swap and ℓ' be it after. $\ell'_k = \ell_k \ \forall k \ne i,j$. $\ell'_i \le \ell_i$ for i moved earlier. $\ell'_j = f'_j - d_j = f_i - d_j \le f_i - d_i \le \ell_i$, so swapping inverted jobs does not increase max lateness. Thus there is some O with no inversions and no idle time by this. Since greedy has no idle time or inversions, it has the same lateness as O and is optimal. ∎

Shortest Path

Goal: Find shortest directed path from s to other nodes.

procedure Dijkstra
  $S \leftarrow \{s\}$
  $\pi(s) = 0,\ \pi(i) = \infty, i \ne s$
  Add nodes Q keys $\pi(v)$
  while $Q \ne \emptyset$ **do**
    $u = Q.ExtractMin()$
    $S \leftarrow S \bigcup \{u\}$
    for (u,v) with $v \notin S$ and $\pi(u) + \ell_{(u,v)} < \pi(v)$ **do**
      $Q.ChangeKey(v, \pi(v)) = \pi(u) + \ell_{(u,v)}$

Proof. For each node $u \in S$, $\pi(u)$ is the shortest length of $s-u$. This is trivially true for $|S| = 1$. Suppose it is true for $|S| = k \ge 1$. Let v be the next node added to S and $u - v$ be the chosen edge. The shortest $s-u$ path plus (u,v) is an $s-v$ path of $\pi(v)$. Consider any other $s-v$ path P; we will show it is not shorter than $\pi(v)$. Let $x-y$ be the first edge in P that leaves S, and let P' be the subpath to x. P is already too longer, for $\ell(P) \ge \ell(P') + \ell(x,y) \ge \pi(x) + \ell(x,y) \ge \pi(y) \ge \pi(x)$. ∎

Minimum Spanning Tree

Def. A *path* is a sequences of edges which connects a sequence of nodes.
Def. A *cycle* is a path with no repeated nodes or edges besides the start and end nodes.
Def. An undirected graph is a *tree* if it is connected and does not contain a cycle.
Def. An undirected graph is a *spanning tree* if it is a tree and touches every vertex in G.
Goal: Given a connected graph G, find a tree T that is spanning with minimized sum of edge weights.

Prim

Greedily grow a tree from a root node.
Def. A *cut* is a partition of nodes into two nonempty sets S and $V - S$.
Def. The *cutset* of a cut S is the set of edges with exactly one endpoint in S.
Cut Property: Let S be any subset of nodes and let e be the min cost edge with exactly one endpoint in S. Then every MST contains e.

Proof. Given e from above and an MST T^*. Suppose $e \notin T^*$. Then $(u,v) = e$ must be connected by a path in T^* for it to be spanning. If we add e to T^* we create a cycle C. Thus $\exists f \in C, CutSet(S)$. $T' = T^* \bigcup \{e\} - \{f\}$ is also a spanning tree and $c_e < c_f \Rightarrow cost(T') < cost(T^*)$, which is a contradiction. ∎

procedure Prim
  for $v \in V$ **do** $a[v] \leftarrow \infty$
  Q
  for $v \in V$ **do** $Q.add(v)$
  $S \leftarrow \emptyset$
  while $Q \ne \emptyset$ **do**
    $u = Q.PullMin()$
    $S \leftarrow S \bigcup \{u\}$
    for $e = (u,v)$ **do**
      if $v \notin S$ & $c_e < a[v]$ **then**
        $a[v] = c_e$

Reverse-Delete

Cycle property: Let C be any cycle, and let f be the max cost edge in C. Then $f \notin MST$.

Proof. Given $f \in C$ from above and an MST T^*. Suppose $f \in MST$. Deleting f disconnects T^* and creates a cut S. f is both in C and in $CutSet(S)$, so $\exists e \in C, CutSet(S)$. $T' = T^* \bigcup \{e\} - \{f\}$ is also a spanning tree and $c_e < c_f \Rightarrow cost(T') < cost(T^*)$, which is a contradiction. ∎

procedure Reverse-Delete
  $T \leftarrow E$
  Order $e \in T$ in descending order of cost
  for $e \in T$ **do**
    if $T - \{e\}$ does not disconnect T **then** $T - \{e\}$

Kruskal

procedure Kruskal
  Sort edge weights in descending order
  $T \leftarrow \emptyset$
  for $u \in V$ **do** Create $\{u\}$
  for $i = 1$ to m **do**
    $(u,v) = e_i$
    if u,v in different sets **then**
      $T \leftarrow T \bigcup \{e_i\}$
      Merge sets containing u,v
  return T

Clustering

Goal: Given set U of n objects classify into coherent groups.

procedure Clustering
  Form a graph on U with n clusters
  Find closest pair of objects in different clusters and add edge between them
  Repeat $n-k$ times until there are k clusters

Divide-and-Conquer

Break up problems into several smaller parts and solve recursively.

procedure MS(L)
  Divide into halves L, R
  $MS(L), MS(R)$
  Combine results

Recurrence: $T(n) = \begin{cases} 0 & n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \end{cases}$
If satisfied, $T(n) \le n \lceil \lg n \rceil$.

Proof. Base case $n = 1$ is true. Let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$. Then $T(n) \le T(n_1) + T(n_2) + n \le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n = n \lceil \lg n_2 \rceil + n \le n(\lceil \lg n \rceil - 1) + n = n \lceil \lg n \rceil$. ∎

Master Theorem

Let $T(n)$ be a monotonically increasing function with $T(n) = aT(n/b) + f(n)$, $T(1) = c$, where $a \ge 1$, $b \ge 2$, $c > 0$. Then if $f(n) \in \Theta(n^d)$ for $d \ge 0$:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

This cannot be used if $T(n)$ is not monotone, if $f(n)$ is not polynomial, or if b cannot be expressed as a constant.

Dynamic Programming

Breaks problem into series of overlapping sub-problems and builds up a solution.

Coin-Changing

Goal: Make minimum change for S cents with a finite supply of coins from $C = \{v_1, \ldots, v_n\}$.
Solve $z(T,i)$ for minimum coins to reach $T \le S$ with first i coins.
Uses *memoization* to use values already computed.

procedure CoinChange
  for $T = 1$ to S **do** $z(T,0) \leftarrow \infty$
  for $i = 0$ to n **do** $z(0,i) \leftarrow 0$
  for $i = 1$ to n **do**
    for $T = 1$ to S **do**
      $z(T,i) \leftarrow z(T,i-1)$
      if $T - v_i \ge 0$ **then**
        $z(T,i) \leftarrow \min(z(T,i), z(T - v_i, i))$

Weighted Interval Scheduling

Goal: find maximum weight subset of mutually compatible jobs.
Let $p(j)$ be the largest index $i<j$ such that i is compatible with j. Then this has recurrence relation
$$OPT = \begin{cases} 0 & j=0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} \end{cases}$$
Bottom up algorithm:

```
procedure WIntSch(L)
    Sort by increasing f_i
    Compute p(i) ∀i
    M[0] = 0
    for j = 1 to n do
        M[j] = max(v_j + M[p(j)], M[j−1])
```

Longest Common Subsequence

Goal: Given two strings, find the longest sequence of letters appearing in both, not necessarily contiguously.
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & x_i = y_i \\ max\{c[i-1,j], c[i,j-1]\} \end{cases}$$

```
procedure LSC(X, Y)
    m = len(X), n = len(Y)
    for i = 1 to m do c[i,0] = 0
    for j = 1 to n do c[0,j] = 0
    for i = 1 to m do
        for j = 1 to n do
            if x_i = y_j then
                c[i,j] = c[i−1,j−1] + 1
            else c[i,j] = max(c[i−1,j], c[i,j−1])
    return c
```

Sequence Alignment

Goal: Given two strings find alignment of minimum cost.
Def. An *alignment* M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.
Def. The pair $x_i - y_j$ and $x_{i'} - y_{j'}$ *cross* if $i < i'$ but $j > j'$.

Let α be a mismatch cost and δ be an unmatch cost. Then:

```
procedure SeqAlign(X, Y)
    for i = 0 to m do M[i,0] = iδ
    for j = 0 to n do M[0,j] = jδ
    for i = 1 to m do
        for j = 1 to n do
            M[i,j] = min(α[x_i, y_j] + M[i−1, j−1], δ + M[i−1, j], δ + M[i, j−1])
    return M[m,n]
```

Knapsack

Goal: Fill knapsack to maximize total value.

```
procedure Knapsack
    for w = 0 to W do M[0,w] = 0
    for i = 1 to n do
        for w = 1 to W do
            if w_i > w then
                M[i,w] = M[i−1,w]
            else
                M[i,j] = min(M[i−1,w], v_i + M[i−1, w−w_i])
    return M[n,W]    ▷ Θ(nW)
```

NPC

Class P consists of decision problems that are solvable in poly time. These problems are *tractable*. Otherwise problems are *intractable*.
Class NP consists of problems where a candidate solution can be verified in poly time.
Y is **NP-Hard** if $X \le_p Y \ \forall X \in NP$.
Y is **NPC** if $Y \in NP$ and Y is NP-hard.
To establish Y is NPC:

1. Show $Y \in NP$
2. Choose X that is NPC and
3. Prove $X \le_p Y$

Reduction from A to B is showing that we can solve A using the algorithm that solves B. Thus A is easier than B. The idea is to transform an instance of A into inputs for B and show that they give identical answers.

NPC Problems

Minimum Vertex Cover: Find the smallest subset $S \subseteq V$ such that each edge has at least one endpoint in S.

Set Cover: Given a universe U and subsets S_i and integer k, does there exist a collection of S_i less than k such that $\bigcup S_i = U$.

3SAT: Given a conjunctive normal form Φ with three literals per clause, find an assignment of values to x_i so that Φ is True.

Independent Set: Given a graph, find the largest $S \subseteq V$ with no edges between them.

CLIQUE: Given a graph, is there a completely connected subgraph of size at least k.

HAM-CYCLE: Given an undirected graph, does there exist a simple cycle that contains every node in V.

TSP: Given a set of n cities and a pairwise distance function, is there a tour of length less than D.

Graph Coloring: Given a graph, can it be minimally colored so that every edge has difference node colors.

Subset-Sum: Given natural numbers w_i and integer W, is there a subset of w_i that adds up to W

Solving NPC

Branch-and-Bound

Keeps track of best solution found so far (upper bound) and for each partial solution computes a lower bound on the objective function. If any bounds are not better than the best solution so far, they are pruned.

```
procedure BnB(P)
    F ← {(∅, P)}, B ← (∞, (∅, P))
    while F ≠ ∅ do
        Choose (X, Y) in F and expand it to (X_i, Y_i) configurations
        for (X_i, Y_i) do
            if Solution found then
                if cost(X_i) < B then B ← (cost(X_i), (X_i, Y_i))
            if Not dead end then
                if LB(X_i) < B then F ← F ⋃ {(X_i, Y_i)}
    return B
```

Local Search

In this, the algorithm starts from an initial position then iteratively moves to a neighboring position using an evaluation function.
Hill-climbing: Choose neighbor with largest improvement as next state.
Stochastic: Randomize initialization step and search steps to allow for suboptimal choices.
Simulated Annealing: Select a neighbor at random. If it is better than current state, go there. Otherwise go there with some probability that decreases over time.
Tabu Search: In each step move to best neighbor solution even if worse than current. Avoids revisiting previously seen solutions with a tabu list of forbidden attributes.
Iterated LS: Generate an initial candidate then perform a LS on it. While termination conditions are not met, perturb current solution and do LS or return to original.

Approximation

In this, performance bounds are guaranteed with quick run-time.
Def. The *Ratio bound* is $max\left(\frac{A(x)}{OPT(X)}, \frac{OPT(X)}{A(X)}\right) \le \rho(n)$ for an input X of size n. Generally $\rho(n) \ge 1$, though unless $P = NP$ this equality will not hold.
For this, you can prove that an algorithm will give a solution no worse that $x * OPT$.

Integer Linear Programming

Given a_{ij}, b_i, c_i find x_j that satisfy $\min c'x$ subject to $Ax \ge b$ with x integral.

Network Flows

An abstraction for material flowing through edges. A directed graph without parallel edges with $c(e)$ equaling the capacity of edge e.
Def. An *s-t flow* is a function f from $E \to \mathbb{R}$ that satisfies

- For $e \in E$: $0 \le f(e) \le c(e)$
- For $v \in V - \{s,t\}$: $\sum_{\text{e into v}} f(e) = \sum_{\text{e out of v}} f(e)$

Def. The *value* of the flow is $v(f) = \sum_{\text{e out of s}} f(e)$.
Def. An *s-t cut* is a partition (A,B) of V with $s \in A$ and $t \in B$.
Def. The *capacity* of cut (A,B) is $cap(A,B) = \sum_{\text{e out of A}} c(e)$.
Let f be any flow and let (A,B) be any s-t cut. Then $\sum_{\text{e out of A}} f(e) - \sum_{\text{e in to A}} f(e) = v(f)$.
Weak duality: Let f be any flow and let (A,B) be any s-t cut. Then the value of the flow is at most the capacity of the cut. Corollary: If $v(f) = cap(A,B)$, then f is a max flow and (A,B) is a min s-t cut.
Def. A *residual edge* of $e = (u,v)$ is $e^R = (v,u)$ with $c_f(e) = \begin{cases} c(e) - f(e) & e \in E \\ f(e) & e^R \in E \end{cases}$

```
procedure Ford-Fulkerson
    for e ∈ E do f(e) ← 0
    G_f ← residual graph
    while ∃s − t path P ∈ G_f do
        f ← Aug(f, c, P)
        update G_f
    return f
procedure Aug(f, c, P)
    b ← bottleneck(P)
    for e ∈ P do
        if e ∈ E then f(e) ← f(e) + b
        else[e^R ∈ E] f(e^R) ← f(e^R) − b
    return f
```

Augmenting path theorem: Flow f is a max flow $\iff$ there are no augmenting paths.
Max-flow min-cut theorem: The value of the max flow is equal to the value of the min s-t cut. We have equivalence between

1. $\exists (A,B)$ such that $v(f) = cap(A,B)$
2. Flow f is a max flow
3. There is no augmenting path relative to f

Bipartite Matching

Given an undirected, bipartite graph, find the max cardinality matching.
Can be reduced to a max flow problem by adding nodes s,t with edge capacities 1.
Theorem: Max cardinality matching equals value of max flow.

Disjoint Paths

Given a directed graph and nodes s,t, find the max number of edge-disjoint (no edge in common) s-t paths.
Theorem: Max number edge-disjoint s-t paths equals max flow by assigning all $c(e) = 1$.

Network Connectivity

Given a digraph and nodes s,t, find min number of edges whose removal disconnects s-t.
Theorem: The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects s-t.
Choosing path with highest bottleneck capacity increases flow by max possible amount.

```
procedure Max-Flow
    for e ∈ E do f(e) ← 0
    Δ ← min 2^n ≥ c
    G_f ← residual graph
    while Δ ≥ 1 do
        G_f(Δ) ← Δ − G_res
        while ∃P ∈ G_f(Δ) do
            f ← aug(f, c, P)
            update G_f(Δ)
        Δ ← Δ/2
    return f
```