# Lecture Notes
CS260 - Algorithms

## Stable Matching

Each one of $n$ hospitals wants to hire ones of $n$ doctors. Both have preferences, need a self-enforcing assignment process.

1. Applicant $d$ and hospital $h$ are **unstable** if given already assigned $d', h'$, it holds that $d(h) > d(h')$ and $h(d) > h(d')$ where $\text{var}_1(\text{var}_2)$ is $\text{var}_1$'s preference of $\text{var}_2$, ($d, h$ prefer each other to their assignments, but not assigned), else **stable**.

2. **Self-enforcing**: selfish action doesn't destabilise system.

3. **Perfect matching**: everyone matched bijectively (1 doc to 1 hosp). **Stable matching** if also has no unstable pairs.

**Propose-and-reject [Gale-Shapley]** $O(n^2)$

```
d,h = free, free # all doc d, hosp h start free
# exists free doc who hasn't applied to every hospital
while (∃ d, h : d free, ∄ d(h)) {
 d, h = doctor, first prefernce unapplied h on d's list
 if h == free: match(d,h)
 elif h(d) > h(d'): match(d,h); d' = free # swap better
 else: h rejects d}
```

*Obs 1*: Docs apply to hosp in decreasing order of preference; *Obs 2*: Once matched, $h$ never becomes unmatched.

4. **Termination**: algorithm terminates in T(G-S) = $O(n^2)$
*Proof:* Each while loop iteration $d$ applies to a new $h$, hence only $n^2$ possible pairings, $n(n-1)+1$ proposals. □

5. **Perfection**: All doctors and hospitals get matched.
*Proof*: FTSOC suppose ¬matched($d$) upon termination, so $\exists h$ : ¬matched($h$), hence was $h$ never applied to (by *Obs 2*), but $d$ applied everywhere. ⊥

6. **Stability**: G-S does not produce any unstable pairs.
*Proof*: suppose $d$, $h$ is an unstable pair.
Case: ¬applied$(d,h) \rightarrow d(h') > d(h)$, hence d,h stable
Case: applied$(d,h) \rightarrow h(d') > h(d)$, hence d,h stable.⊥

7. **Doctor Optimality**: GS matching $S^*$ is doctor-optimal.
*Proof*: FTSOC suppose $\exists h' | d(h) < d(h')$: $\forall i \mid d_i(h_i) > d_{i+1}(h_{i+1})$, hence $\exists$rej$(d_1)$ by some $h_1$. Let matching S $(d_1, h_1)$ be stable. Rej $d_1 \rightarrow$ pair$(d_2, h_1)$ s.t. $h_1(d_2) > h_1(d_1)$. Now S=$(d_2, h_2) \rightarrow$ ¬rejected$(d_2)$ when rej$(d_1)$ by $h_1 \rightarrow d_2(h_1) > d_2(h_2)$, but $h_1(d_2) > h_1(d_1)$, hence matching $S = (d_2, h_1)$ is not stable. □

8. **Hospital Pessimality**: doctor-optimality makes hospitals receive their worst preference. *Proof*: FTSOC suppose $S^*(d_1, h_1) | \neg \min(h_1(d_1)) \rightarrow \exists S(d_2, h_1) | h_1(d_2) < h_1(d_1)$. Let $S(d_1, h_2)$, but $S^* \rightarrow d_1(h_1) > d_1(h_2)$, so $S(d_1, h_1)$ unstable.⊥

9. **Generalised GS**: number of doctors and hospitals don't need not match $|d| \neq |h|$, $\exists$ unacceptables, |matches| $> 1$.

## Greedy Algorithms

Pick the next thing to do that looks like the best option. *Earliest start/finish time, shortest duration, fewest conflicts.* Completes jobs in order of $s_j$, $f_j$, $f_j - s_j$ or $c_j$ $j$'s conflicts.

### Interval Scheduling

Job $j$ starts at $s_j$, finishes at $f_j$. Two jobs compatible if they don't overlap $j_1 \cap j_2 = \varnothing$. **Goal**: Find maximum subset of mutually compatible jobs.

**Earliest finish time first** $O(n \log n)$

```
Sort jobs by finish time s.t. f_1 ≤ f_2 ≤ ... ≤ f_n #nlogn
A ← φ # set of jobs selected
for j=1 to n {
    if (job j compatible with A): A ← A ∪ {j}
} return A
```

**Theorem**: it's optimal. FTSOC, assume greedy isn't optimal. Let $i_1, ...i_k$ be selected jobs, $j_1, ..j_m$ be jobs in optimal solution with $i_1 = j_1, ...i_r = j_r$ for largest possible $r$.

### Interval Partitioning

Depth of a set of open intervals is the maximum number that contain any given moment in time

**Fewest conflicts first** $O(n \log n)$

```
Sort intervals by start time s_1 ≤ s_2 ≤ ... ≤ s_n
d ← 0 # num allocated classrooms (depth)
for j=1 to n {
    if (lecture j compatible with room k):
        schedule j in first free classroom k
    else
        allocate new classroom d+1
        schedule j in d+1
        d ← d + 1
} # keep k in priority queue
```

**Theorem**: it's optimal. Notice $i \cap j \neq \varnothing \rightarrow d_i \neq d_j$. Let $d$ = num classrooms greedy allocates, so $k_d$ $j \cap j_i$ incompatible with all $d-1$ others ("witness"). These $d$ jobs end after $s_j$, otherwise could've used one, all incompatibilities caused by lectures starting $\leq s_j$. So have $d$ overlapping lectures at time $s_j + \epsilon \rightarrow$ all correct schedules must use $\geq d$ rooms. □

## Minimising lateness

Single resource processes 1 job $j$ at once requiring $t_j$ time and is due at time $d_j$. It starts at $s_j$ and finishes at $f_j = s_j + t_j$. It has lateness $L_j = \max\{0, f_j - d_j\}$, and the goal is to minimise max. lateness.

**Observ.1:** There $\exists$ an optimal schedule with no idle time.

**Proof:** Inversion is pair of jobs $i, j | i < j$, but $j$ scheduled before $i$, swap to fix (no inversions $\rightarrow$ sorted $\rightarrow$ earliest deadline first order). □

**Earliest-deadline first** $O(n \log n)$

```
Sort n jobs by deadline s.t. d_1 ≤ d_2 ≤ ... ≤ d_n
t ← 0
for j=1 to n {
    Assign job j to interval [t,t+t_j]
    s_j ← t, f_j ← t + t_j
    t ← t + t_j
} return intervals [s_j, f_j]
```

**Observ:** If idle-free schedule has inversion, then it has an adjacent inversion.

**Proof**: Let $(i, j)$ be inversion $(i < j)$ that is closest but not adjacent, $k$ immediately follow $j$, then either $j > k \rightarrow (j, k)$ adjacent or $j < k \rightarrow (i, k)$ is closer as $i < j < k$, so either find adjacent inversion or $(i, j)$ not closest. □

**Exchange argument:** Swapping two adjacent, inverted jobs reduces the number of inversion by 1 and doesn't increase the max lateness. **UNDERSTAND WHY!**

**Theorem:** Greedy schedule $S$ is optimal.

**Proof:** $S^* \stackrel{\text{def}}{=}$ optimal schedule with fewest num of inversions. Assume $S^*$ has no idle time, if $S^*$ has no inversions, then $S = S^*$, else let $(i, j)$ be an adjacent one. Then swapping $i, j$ doesn't increase max lateness, strictly decreases num of inversions, but contradicts $S^*$ having fewest inversions. □

## Coin-Changing

Given currency denominations, find a way to pay full amount using least number of coins.

**Cashier's algorithm**: At each iteration, add coin of the largest value that doesn't go past the total amount. Only works for certain denominations.

**Cashier's algorithm** $O(n \log n)$

```
Sort coins denominations by value c_1 < ... < c_n
S ← φ # coins selected
while (x≠ 0 {
    let k be max(int(k)) s.t. c_k ≤ x
    if (k==0): return "no solution"
    x ← x - c_k
    S ← S ∪ {k}
} return S
```

## Offline Caching

Capacity to store $k$ items, sequence of $m$ item requests $d_1, ..., d_m$. Cache hit: item in cache upon request, cache miss: item isn't there, must fetch item into cache, evict some other item if it's full. Goal: find eviction schedule to minimise number of cache misses. Offline caching: sequence of requests is known in advance.

**Theorem [Belady, 1960s]**: Furthest-in-future (FF) eviction strategy: Evict item that isn't requested until furthest in future. **Reduced** schedule is one that only inserts an item into cache in the same step as it is requested. **Unreduced** otherwise.

**Claim** Given any unreduced schedule $S$, can transform it into a reduced schedule $S'$ with no more cache misses

***Proof:*** By induction on number of unreduced items. Suppose $S$ brings $d$ into cache at time $t$, without a request. Let $c$ be the item $S$ evicts when it brings $d$ into cache. Case 1: $d$ evicted at time $t'$, before next request for $d$: skip it. Case 2: $d$ requested at time $t'$ before $d$ evicted: delay it. □

**Invariant:** $\exists S$ (optimal reduced schedule) that makes same eviction schedule as $S_{FF}$ through first $j$ requests.

**PROVE THAT FF IS OPTIMAL OFFLINE**

LRU is **k-competitive**, or only $k$ (cache size) times worse than offline. LIFO can be arbitrarily bad.

## Selecting breakpoints

Make as few refuelling stops as possible on a journey. Greedy: go as far as you can before stopping.

**Truck driver's algorithm** $O(n \log n)$

```
Sort breakpoints s.t. 0 = b_0 < ... < b_n = L
S, x = {0}, 0 # breakpoints, curr location
while (x ≠) {
    let p = max(int) s.t. b_p ≤ x + C
    if (b_p = x): return "no solution"
    x, S = b_p, S ∪ {p}
} return S
```

***Proof:*** FTSOC assume greedy isn't optimal. Let $0 = g_0 < ... < g_p = L$ be breakpoints chosen by greedy, and $0 = f_0 < ... < f_q = L$ to be optimal. Need $\forall f, g | f_i = g_i$ but $g_{r+1} > f_{r+1}$. **DON'T UNDERSTAND** ⊥

# Graph Algorithms

1. Undirected graph, Adj matrix/list, (simple)path, connected graph, cycle, tree, rooted tree

### BFS

1. **BFS**: explore from node $s$ in all directions, one layer at a time.
2. **Theorem**: $\forall i : L_i$ consists of all nodes at distance $i$ from $s$. There is a path from $s$ to $t$ iff $t \in L_i$.
3. **Property**: Let $T$ be a BFS tree of $G = (V, E)$ and $(x, y) \in G$ be an edge. Then layers of $x$ and $y$ differ by at most 1.
4. **Theorem**: Adj. list BFS runs in $O(m + n)$. **Proof**: Consider node $u$, there are $\deg(u)$ incident edges $(u, v)$, so $T = \sum_{u \in V} \deg(u) = 2m$ $\square$.
5. **Connected component**: all nodes reachable from $s$; can be found using any exhaustive search like BFS.

### Bipartite Graphs

1. **Bipartite** graphs: no incident edges of the same colour. Used for stable matching, scheduling and recommender systems. Use BFS, alternating neighbour colours between red, and blue; takes $O(n + m)$ for adj. list.
2. **Lemma**: if graph $G$ is bipartite, it can't contain an odd length cycle. **Proof**: Impossible to 2-colour odd cycle.
3. **Lemma**: Let graph $G$ be connected, $L_0, .., L_k$ be layers produced by BFS starting at node $s$. Then either:
   **1.** No edge within same layer, and $G$ is bipartite. **Proof**: Suppose $\nexists (x, y) \in G$ s.t. $x, y \in L_i$, by BFS property, all edges join nodes on adj. layers. bipartite with red when even $i$ else blue. $\square$
   **2.** Some edge of $G$ joins two nodes of the same layer, so odd-len cycle - not bipartite. **Proof**: suppose $(x, y) \in G$ s.t. $x, y \in L_j$. Let $z \in L_i$ =lca$(x, y)$ (**lowest common ancestor**). Cycle $x \to y \to z \to x$ has length $1 + (j - i) + (j - 1)$, which is odd. $\square$
4. **Corollary**: Graph $G$ is bipartite iff it contains no odd length cycle. **Proof**: follows from above lemmas.

### Graph Connectivity

1. **Directed graph** $G = (V, E)$ - edges are ordered pairs. Digraph problems include: directed reachability, digraph exploratoin, shortest path and strong connectivity.
2. Nodes $u, v$ are **mutually reachable** if there are paths from $u$ to $v$ and $v$ to $u$. **G** is **strongly connected** if all node pairs are mutually reachable.
3. **Lemma**: Given any node $s$, $G$ is strongly connected iff every node is reachable from $s$ and $s$ is reachable from every node. **Proof**: first follows from definition, reverse: $u \rightsquigarrow v = u \rightsquigarrow s + s \rightsquigarrow v$; $v \rightsquigarrow u = v \rightsquigarrow s + s \rightsquigarrow u$. $\square$
4. **Theorem**: Can test if $G$ is strongly connected in $O(m + n)$. **Proof**: Pick any node $s$, run BFS from $s$ in $G$, then BFS from $s$ in $G^{\text{rev}}$ (reverse orientation of every edge in $G$), return true if all nodes are reached in both. Correctness follows from previous lemma. $\square$

### Minimum Spanning Tree

1. **Minimum spanning tree**. Given an undirected connected graph $G = (V, E)$ with edge weights $c_e \in \mathbb{R}$, MST is a subset of edges $T \subseteq E$ s.t. $T$ is a spanning tree (covers every node) whose sum of edge weights is minimised.
2. **Cutset** $D$ is subset of edges with exactly one endpoint in $S$. A cut is a subset of nodes $S$.
3. **Cycle-cut intersection property** (C-CI): a cycle and a cutset intersect in an even number of edges.
4. **Cut property**: Let $S$ be any subset of nodes, and $e$ - min cost edge with exactly one endpoint in $S$, then MST $e \in T^*$. **Cycle property**: Let $C$ be any cycle in $G$, $f \in C$ - max cost edge, then MST $f \notin T^*$. Assume edge costs $c_e$ distinct. **Proof on week 3 page 41**.
5. **Cayley's formula**: There are $n^{n-2}$ spanning trees of $K_n$, so brute force is infeasible. Use greedy instead:
6. **Kruskal's algorithm**: start with $T = \varnothing$, consider edges in ascending order of cost, insert edge $e$ in $T$ (cut prop.) unless doing so would create a cycle (cycle prop.). Use **union-find** on sets $O(\log m)$ and sorting takes $O(m \log m)$.
7. **Reverse-Delete algorithm**: start with $T = E$, consider edges in descending order of cost, delete edge $e$ from $T$ unless it disconnects $T$.
8. **Prim's algorithm**: start at some root node $s$, greedily grow a tree $T$ from $s$ by iteratively adding cheapest edge $e$ that has exactly one endpoint in $T$ (cut property). Use PQ for nodes to consider, set of explored nodes $S$. $\forall v \in$ PQ, store cost of cheapest edge $(v, u \in S)$ - $a[v]$. Runs in $O(n^2)$ with array, $O(m \log n)$ with binary heap.
9. **Boruvka's algorithm**: start with each node in its own cluster, add cheapest edge outgoing from each cluster and merge them. Iterate until no more merges.
10. **Lexicographic tiebreaking**: perturb cost of $e_i$ by $\frac{i}{n^2}$.

### Dijkstra's algorithm

1. **Shortest path problem**: Given directed graph $G$ with edge weights $w(e) \geq 0$, src $s$ and dest $t$, write path $P$ is a sequence of edges $e_{|P|} = (v_{|P|-1}, v_{|P|})$, so $w(P) = \sum_{i=1}^{|P|} w(e_i)$, find shortest directed path $s$ to $t$.
2. **Dijkstra's algorithm**: maintain a set of explored nodes $S$ with shortest path $d(u)$ from $s$ to $u$; initialise $S = \{s\}, d(s) = 0$, then repeatedly choose unexplored node $v$ s.t. $c(v) = min_{e=(u \in S,v)} d(u) + w(e)$ (shortest path to $u$ + edge $(u, v)$) and add $v$ to $S$, set $d(v) = c(v)$.
3. **Invariant**: $\forall u \in S : d(u)$ is length of shortest $s - u$ path. **Proof**: Base case: $|S| = 1$: $d(s) = 0$, I.H.: Assume true for $|S| = k \geq 1$. Let $v$ be next node added to $S$, choose edge $u - v$. Shortest $s - u$ path + $(u, v)$ is $s - v$ path of len $c(v)$. Any $s - v$ path $w(P) >= c(v)$: Let $x - y$ be first edge in $P$ to leave $S$, $P'$ - subpath to $x$, then $w(P) > c(v)$ when reaches $y$.

   $x(P) \geq w(P') + w(x, y) \geq d(x) + w(x, y) \geq c(y) \geq c(v) = d(v)$
4. Efficient implementation uses PQ of unexplored nodes, prioritised by $c(v)$ - $O(m + n \log n)$ with Fibonnaci heap.

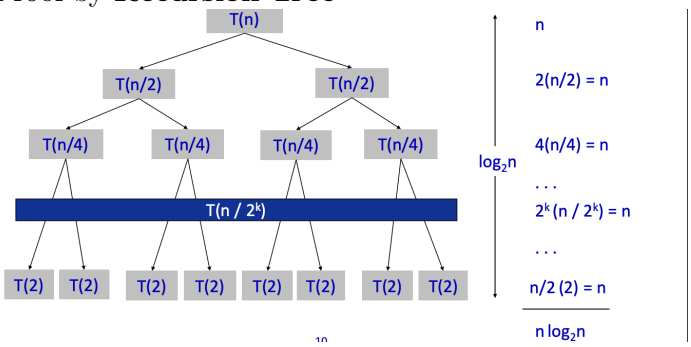
## Divide-and-Conquer Algorithms

Break problem into several parts (divide), solving each one recursively (conquer), and merge the solutions.

### Sorting

1. Mergesort: divide array into two halves $O(1)$, sort each $2T(n/2)$ and merge $O(n)$.
2. Set $T(n)$ = number of **comparisons** to mergesort an input of size $n$. Base case: $T(n) = 0$ if $n = 1$ (single element is sorted), else $T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$. Solution: $T(n) = O(n \log_2 n)$. Assume $n$ is a power of 2.

   *I*: Proof by **Recursion Tree**




   *II*: Proof by **Telescoping**. **Claim**: if $T(n)$ satisfies the recurrence, then $T(n) = n \log_2 n$. **Proof**: For $n > 1$:

$$\frac{T(n)}{n} = \frac{2T(\frac{n}{2})}{n+1} = \frac{T(\frac{n}{2})}{(\frac{n}{2})+1} = ... = \frac{T(\frac{n}{n})}{(\frac{n}{n})+1+..+1} = \log_2 n$$

   *III*: Proof by **Induction**: Base case $n = 1$ follows immediately, I.H.: assume $T(n) = n \log_2 n$, goal: show that $T(2n) = 2n \log_2(2n)$. So, $T(2n) = 2T(n) + 2n = 2n \log_2 n + 2n = 2n(\log_2(2n) - 1) + 2n = 2n \log_2(2n)$. □

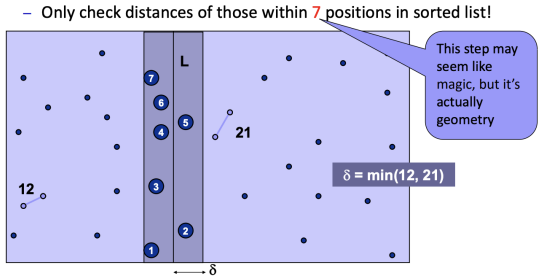
   *IV*: optional page 13-14

### Counting Inversions

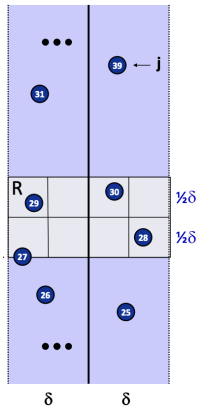
1. **Inversion** between $i, j$ is if $i < j$ but $a_i > a_j$ for rank $a$.
2. To count inversions: separate list into two pieces, recursively count inversions in each half, count inversions where $a_i, a_j$ are in different halves (assume sorted, so if $a_j < a_i$, it needs $n - i$ inversions, so $O(n)$), return sum of three quantities.

### Closes Pair of Points

1. Given $n$ points in the plane, find a pair with smallest Euclidean distance between them. Assume unique $x$ coordinate




   **Proof**: Let $s_i$ be the point in the $2\delta$-strip with $i^{th}$ smallest $y$ coord. **Claim**: if $|i - j| \geq 7$ then distance between $s_i, s_j \geq \delta$. In $2\delta \times \delta$ rectangle $R$ with min $y$ coord of $s_i$, points outside $R$ are further than $\delta$. Divide $R$ into 8 equal squares, no two points in same $\frac{\delta}{2} \times \frac{\delta}{2}$ box as otherwise contradicts conquer step. So, at most 7 other points in $R$.




2.

**Closest Pair (cp) Algorithm** — $O(n \log^2 n)$

```
Closest-Pair(p_1, ..., p_n) {
 Compute sep. line L s.t. |left|≈|right|
 δ = min(cp(left), cp(right))
 Del. points further > δ from L, sort others by y
     coord. Update δ if 7-neighbour distance is
     smaller.
 return δ
}
```

   Don't sort points in strip from scratch each time $O(n \log n)$ - maintain $x, y$ coord-sorted lists, merge.

### Master Method

1. Used to solve recurrences of form $T(n) = aT(\frac{n}{b}) + f(n)$ with branching factor $a$, $a^i$ sub problems of size $\frac{n}{b^i}$ at $i^{th}$ level (total $1 + \log_b n$ levels), and $f(n)$ work for merging.
2. Take $f(n) = n^c$, then work changes by $r = \frac{a}{b^c}$ each level, resulting in total work $T(n) = n^c \sum_{i=0}^{\log_b n} r^i$. **Proof**: if $c > \log_b a$ then $r < 1$, so $T(n)$ is geometric series with common ratio $r$.
   **Case 1**: $r < 1 \to T(n) = \Theta(n^c)$ (1 is highest factor);
   **Case 2**: $r = 1 \to T(n) = \Theta(n^c \log n)$ (1 for each level);
   **Case 3**: $r > 1 \to T(n) = \Theta(n^{\log_b a})$ ($r^k$ is highest fac).
3. Now $n$ doesn't have to be a power of $b$. Let $a \geq 1, b \geq 2, c \geq 0$. Suppose $T(n) = aT(\frac{n}{b}) + n^c$ with $T(0) = 0, T(1) = \Theta(1)$; $\frac{n}{b} = \lceil \frac{n}{b} \rceil = \lfloor \frac{n}{b} \rfloor$. Then:

$$\begin{aligned} &\textbf{Case 1.} \quad c > \log_b a \quad \Rightarrow \quad T(n) = \Theta(n^c) \\ &\textbf{Case 2.} \quad c = \log_b a \quad \Rightarrow \quad T(n) = \Theta(n^c \log n) \\ &\textbf{Case 3.} \quad c < \log_b a \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a}) \end{aligned}$$

4. Doesn't work for: num of subproblems isn't a constant, is less than 1 or the work to dovide and combine them isn't $\Theta(n^c)$, use induction or telescoping.

### Integer Multiplication

1. **Long multiplication** takes $n$ binary shifts and $n$ $O(n)$-bit additions, so $\Theta(n^2)$, use **Karatsuba Multiplication** instead: Add two $\frac{1}{2}n$-bit integers, multiply **three** such integers recursively $a = a_1 x + a_0$; $b = b_1 x + b_0$ for $x = 2^{\frac{n}{2}}$

$$\begin{aligned} ab &= a_1 b_1 x^2 + (a_1 b_0 + b_1 a_0)x + a_0 b_0 \\ &= a_1 b_1 x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0)x + a_0 b_0 \end{aligned}$$

   Hence, $n$-bit integer multiplication takes $\mathrm{O}(n^{\log_2 3})$

**Karatsuba Algorithm** $\mathbf{O}(n^{\log_2 3})$

```
def FM(x,y,n): # Fast Multiply
if (n==1): return x*y; else:
m,a,b,c,d = ⌈n/2⌉, ⌊x/2^m⌋, x mod 2^m, ⌊y/2^m⌋, y mod 2^m
e,f,g = FM(a,c,m), FM(b,d,m), FM(a+b,c+d,m)
return 2^2m e+2^m (g+h)+f
```

### Matrix Multiplication

1. Given $n \times n$ matrices $A, B, C = AB$: $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$ taking $\Theta(n^3)$, so use **Block Matrix Multiplication**:

$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

2. Partition $A, B$ into $\frac{1}{2}n \times \frac{1}{2}n$ blocks, multiply 8 such pairs recursively, add products using 4 matrix additions.
3. Still $O(n^3)$, so partition $A, B$ into $\frac{1}{2}n \times \frac{1}{2}n$ blocks, compute 14 such matrices via 10 additions, recursively multiply 7 pairs, combine into 4 terms using 8 additions - this is **Strassen** algorithm, taking $O(n^{\log_2 7})$.
4. **Sparsity**: many matrices are mostly 0, **caching effects**: fast cache memory makes a difference, **numerical stability**: handle small and large values, **odd matrix dimensions**, **parallelism**, **base case** at $n = 128$.
5. **Conjecture**: $\forall \epsilon > 0 : O(n^{2+\epsilon})$ is optimal, not practical.

## Dynamic Programming

Break up a problem into a series of overlapping subproblems, and build up solutions to the whole.

### Weighted Interval Scheduling (WIS)

1. Job $j$ starts/finishes at $s_j, f_j$ with weight or value $v_j$. Two jobs are **compatible** if they don't overlap. **Goal**: find maximum weight subset of mutually compatible jobs. Label jobs by finishing time $f_1 \leq ... \leq f_n$, $p(j) \stackrel{\text{def}}{=}$ largest index $i < j$ s.t. $i, j$ compatible.
2. Write OPT($j$)= optimal solution for $1, 2, ..., j$. **Case 1**: OPT selects $j$: collect profit $v_j$, must include OPT from remaining compatible jobs $1, 2, .., p(j)$ **Case 2**: OPT doesn't select $j$, include OPT from $1, 2, ..., j-1$.
3. **Bellman equation**:

$$\mathrm{OPT}(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{else} \end{cases}$$

4. **Memoization** means to store results of each subproblem in a table to avoid re-computation. **Claim**: memoized WIS takes $O(n \log n)$. **Proof**: Sort by finish time: $O(n \log n)$, each lookup table invocation is $O(1)$ and either return an existing value $M[j]$, or fills in a new entry and makes 2 recursive calls. Write progress measure $\Phi =$ # nonempty entries of M[], since $\Phi \leq n \to$ at most $2n$ recursive calls, so $O(n)$ after pre-sorted by $s, f$.
5. **Bottom-up dynamic programming**: unwind recursion, fill in the table from smallest to biggest.

**Bottom-up WIS** $\mathbf{O}(n \log n)$

```
Inp: n, s1,...,sn, f1,...,fn, v1,...,vn
Sort jobs by finish times s.t. f1 ≤ f2 ≤ fn
Compute p(1),...,p(n)
Iterative-Compute-Opt { M[0] = 0
  for j=1 to n : M[j] = max(vj + M[p(j)], M[j-1])}
```

## Segmented Least Squares

1. Given $n$ plane points $(x_1, y_1), .., (x_n, y_n)$ with $x_1 < .. < x_n$ find a sequence of lines minimising some $f(x)$. Sum of squared error is $SSE_{1,n} = \sum_{i=1}^{n}(y_i - ax_i - b)^2$, choose $a, b$ to minimise SSE. Use partial derivatives: $SX_{1,n} = \sum_{i=1}^{n} x_i$, $SXY_{1,n} = \sum_{i=1}^{n} x_i y_i$, etc takes $O(n)$ per sum.

2. Combine SSE for each segment as $E$ with $L$ lines, $f(x) = E + cL$(**regularisation** term) for some $c > 0$. Write OPT$(j)$ be min. cost for points $p_1, .., p_j$ and $e(i, j)$ = their min. sum of squares or OPT$(i, j)$ use **Bellman**:

**Segmented-Least-Squares** — $O(n^3) \to O(n^2)$

```
Inp: n, p_1,..,p_N, c
def SLS() { M[0] = 0
  for j=1 to n: for i=1 to j:
    compute e_ij for segment p_i,..,p_j
  for j=1 to n:
    M[j] = min_{1≤i≤j}(e_ij+c+M[i-1])
  return M[n]}
```

3. To speed up to $O(n^2)$, use cumulative prefix sums:

$$a_{ij} = \frac{nSXY_{i,j} - SX_{i,j}SY_{i,j}}{nSXX_{i,j} - (SX_{i,j})^2}; \quad b_{ij} = \frac{SY_{i,j} - aSX_{i,j}}{n}$$

computing $a_{ij}, b_{ij}, e(i, j)$ takes $O(n)$ each. Prefix sums: $SX_{i,j} = SX_{1,j} - SX_{1,i-1}$ - precompute $SX_{1,j}$ for $j = 1..n$ in $O(n)$ to find any $SX_{i,j}$ in $O(1)$, same for $SY, SXX, SXY$, so now $e(i, j)$ takes $O(1)$ per lookup, hence $\mathbf{O(n^2)}$ overall.

## Knapsack Problem (2-D)

1. Fill knapsack to maximise total value given $n$ objects $i^{th}$ of which has weight and value $w_i, v_i > 0$.

2. Create 2-D $v \times w$ lookup table, OPT$(i, w) \stackrel{\text{def}}{=}$ max. profit subset of items $1, .., i$ with weight limit $w$. **Case 1**: OPT$(i, w)$ doesn't select item $i$: best of $\{1, .., i-1\}$ using $w$. **Case 2**: OPT$(i, w)$ selects $i$ - update weight limit $w - w_i$, and use to select best of $\{1, .., i-1\}$.

**Knapsack Bottom-Up** — $O(nW)$

```
Inp: n, W, w_1,..,w_N, v_1,..,v_N
for w=0 to W: M[0, w] = 0
for i=1 to n: for w=1 to W:
  if (w_i >w): M[i,w] = M[i-1,w]
  else: M[i,w] = max{M[i-1,w], v_i+M[i-1,w-w_i]}
return M[n, W]
```

**Not polynomial** input size $W$! It's specified in $\log W$ bits, so time is exponential, hence called **pseudo-polynomial**, however NP-complete.

## RNA Secondary Structure

1. Express RNA as string $B = b_1 b_2..b_n$ over alphabet $\{A, C, G, U\}$, possible pairs: $A, U$ and $G, C$.

2. **Secondary structure** is a set of pairs $S = \{(b_i, b_j)\}$ s.t.: **Watson-Crick** - $S$ is a matching and each pair in $S$ is a base pair complement: $AU, UA, CG, GC$. **No sharp turns**: all pairs' edges separated by $\geq 4$ bases: $(b_i, b_j) \in S \to i < j - 4$, and **Non-crossing**: if $(b_i, b_j), (b_k, b_l) \in S \to \neg(i < k < j < l)$. Secondary structure forms with optimum total **free energy**. **Goal**: find secondary struct $S$ maximising num base pairs.

3. OPT$(i, j) \stackrel{\text{def}}{=}$ max. num of base pairs in $S$ of $b_i b_{i+1}..b_j$:
**Case 1**: $i \geq j - 4 \to$ OPT$(i, j) = 0$ by no-sharp turns.
**Case 2**: base $b_j$ not in a pair: OPT$(i, j) =$ OPT$(i, j-1)$, (nothing changed);
**Case 3**: base $b_j$ pairs with $i \leq t \leq j - 4 : b_t$, then non-crossing decouples sub-problems: OPT$(i, j) = 1 + \max_t\{\text{OPT}(i, t-1) + \text{OPT}(t+1, j-1)\}$

**Bottom-Up DP Over Intervals** — $O(n^3)$

```
RNA(b_1,..,b_n) {
  for k=5 to n-1: for i=1 to n-k:
    Compute M[i,i+k] using above formula
  return M[1,n]}
```

## Sequence Alignment

1. Given gap penalty $\delta$ and mismatch penalty $\alpha_{pq}$, and their sum being cost, find alignment of minimum cost. **Alignment** $M$ is set of ordered pairs $x_i, y_j$ one-to-one. Shouldn't **cross**: $x_i, y_j$; $x_{i'}, y_{j'}$ cross if $i < i'$ but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

2. OPT$(i, j) \stackrel{\text{def}}{=}$ min cost of aligning strings $x_1..x_i$, $y_1..y_j$.
**Case 1**: OPT matches $x_i, y_j$, so pay $\alpha_{ij}$+ min cost of aligning preceding $x_1..x_{i-1}$, $y_1..y_{j-1}$.
**Case 2a**: OPT leaves $x_i$ unmatched: pay $\delta_{x_i}$ + min cost of $x_1..x_{i-1}$, $y_1..y_j$.
**Case 2b**: OPT leaves $y_j$ unmatched: pay $\delta_{y_j}$ + min cost of $x_1..x_i$, $y_1..y_{j-1}$.

**Sequence Alignment** — $O(m \cdot n)$

```
Seq-Align(m,n, x_1,..,x_m, y_1,..,y_n,δ,α) {
for i=0 to m: M[i,0]=iδ; for j=0 to n: M[0,j]=jδ
for i=1 to m: for j=1 to n:
  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]} # O(m·n) space is too bad
```

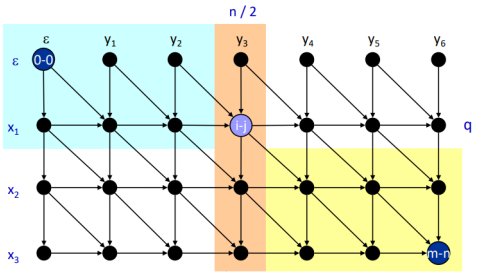
### Sequence Alignment in Linear Space

1. Want to avoid quadratic space: build up table one row at a time, forget old rows, need to compute $\text{OPT}(i, \bullet)$ from $\text{OPT}(i-1, \bullet)$.

2. **Edit distance graph**: let $f(i,j) = \text{OPT}(i,j)$ be shortest path from $(0,0)$ to $(i,j)$, so can compute $f(\bullet, j)$ for any $j$ in $\text{O}(m \cdot n)$ time and $\text{O}(m+n)$ space.

3. For shortest $(i,j) - (m,n)$ path $g(i,j)$, reverse edges to swap roles of $(0,0)$ and $(m,n)$, and compute as before.

4. Notice cost of shortest path via $(i,j)$ is $f(i,j)+g(i,j)$, so given index $q$ that minimises $f(q, \frac{n}{2}) + g(q, \frac{n}{2})$, the shortest $(0,0) - (m,n)$ path uses $(q, \frac{n}{2})$. **Divide**: Align $x_q, y_{\frac{n}{2}}$ in DP solution, **Conquer**: recursively compute optimal alignment in each piece.




5. **Theorem**: Let $T(m,n) = \max$ runtime on strings of $m, n$ at longest, then $T(m,n) = O(m \cdot n \log n)$.
**Proof** (by induction on $n$): $\text{O}(m \cdot n)$ to compute $f(\bullet, \frac{n}{2}), g(\bullet, \frac{n}{2})$ and find $q$, then $T(q, \frac{n}{2}) + T(m-q, \frac{n}{2})$ time for two recursive calls. Choose constant $c$ s.t. $T(m,2) \leq cm$ and $T(2,n) \leq cn$ and $T(m,n) \leq cmn + T(q, \frac{n}{2}) + T(m-q, \frac{n}{2})$. **Base case**: $m = n = 2$, **I.H.**: $T(m,n) \leq 2cmn$:

$$\begin{aligned} T(m,2) &\leq T(q, \frac{n}{2}) + T(m-q, \frac{n}{2}) \\ &\leq \frac{2cqn}{2} + \frac{2c(m-q)n}{2} + cmn = 2cmn \end{aligned}$$

Induction holds, so $T(m,n) \leq 2cmn = \text{O}(m \cdot n)$.

## Week 6

### P, Complexity Classes, Tractability

1. An algorithm is a well-specified set of instructions that, when followed, provide the answer to a given question.

2. A proof is a convincing argument that a claim is true. Provide proof of **Correctness** and **Efficiency**.

3. Complexity of a problem is the time/space efficiency of the best algorithm that solves it.

4. **Complexity class** is a set of problems all fulfilling some property. A problem $X \in$ NP iff there is a piece of evidence (**witness**) that can be quickly confirm that answer is $T$ for particular instance.

5. A problem is **tractable** if it can be solved in practice, by computer in reasonable time. Set of problems in $P$ is exactly the set of problems which are tractable. Many problems aren't known to be neither tractable nor intractable, so find **relative difficulty**.

6. Algorithmic problem $X$ is **polynomial** ($\in$ **P**) PTIME if $\exists$ algorithm solving $X$ in $O(n^k)$ for some constant $k$, or $\bullet$ $X$ is tractable, or $\bullet$ $X$ admits an efficient algorithm.

$$\text{PTime} \stackrel{\text{def}}{=} \cup_{k \in \mathbb{N}} O(n^k)$$

7. Problems can be **search**, where output must fulfill some predicate, **optimisation** - such that output is max/min somehow and **decision** $X : inp \to$ Yes/No (use for NP).

### NP and Independent-Set

1. **Independent-Set** $S$ in graph $G = (V, E)$ is a subset of $V$ s.t. no two vertices in $S$ are adjacent in $G$. **IS problem**: given an undirected graph $G$ and integer $k$, is there an $IS$ of size $k$ in $G$?

2. **NP** is a complexity class containing **decision problems** that are "easy to verify". Decision problem $X$ is $\in$ NP iff $\exists$ $\text{O}(n^k)$ **verifier** $C$ taking problem instance $i$ and **candidate witness** $w$ as inputs. If answer to $X(i) = F$ then $C(i,w)$ is always $F$, else $\exists w$ s.t. $C(i,w) = T$.

3. **NP** (nondeterministic polynomial-time), if have **verifier** for problem $X$, we have a nondeterministic algorithm for $X$: Given an instance $i$ of $X$: Guess a possible polynomially-sized witness $w$ for $i$, run verifier $C$ on input $i$ and $w$. If any $w$ is a witness for $i$, return $T$.

4. **Theorem**: P $\subseteq$ NP. **Proof**: $\forall X \in \text{P} : \exists C \in \text{P}$, hence $C(i,w) = A(i)$ is a verifier for $X$ ($w$ is ignored). $\square$

5. **Theorem**: $IS \in NP$. **Proof**: construct a polynomial-time verifier $C$ for IS with input comprising IS problem instance $G = (V,E)$ and $k$, **candidate** IS $S \subseteq V$ to act as a witness. Then, if $|S| \neq k$ return $F$, if for any $v_1, v_2 \in S : \{v_1, v_2\} \in E$, also return $F$, else $T$. $\text{O}(n^2)$
This checks that $S$ is an IS of size $k$ in $G$, and if so, $S$ is a witness that the answer to IS is $T$, so $C$ is a verifier.

6. Problem $X$ is **NP-complete** if $X \in$ NP and for every problem $Y \in \text{NP} : Y \leq_P X$. ($X$ is at least as hard as every other problem in NP). Definition on the next page.

## Reduction

1. **Reduction** is a description of how to solve a problem using a solution for another problem. e.g. reducing from Find-Min (min int in a sequence) to Find-Max by creating sequence $B = b_1..b_n$ s.t. $b_i = -a_i$, let $m =$ Find-Max$(B)$, return $-m$.
2. In reducing from $X$ to $Y$, we're not allowed to know anything about specific solution to $Y$: it is a black box, and is called an **oracle** for $Y$.
3. Problem $X$ polynomial-time reduces to problem $Y$, or $X \leq_P Y$ iff any instance of $X$ can be solved using: Pol. number of calls to oracle for $Y$ + Pol. number of standard compute steps. All instances of $Y$ are Pol. size. Can say that $X$ is at most polynomially harder than $Y$. $(X \leq_P Y) \wedge (Y \leq_P X) \rightarrow X \equiv_P Y$ Pol. time equivalent.
4. **Theorem**: $(X \leq_P Y) \wedge (Y \in P) \rightarrow X \in P$
   **Corollary**: $(X \leq_P Y) \wedge (X \notin P) \rightarrow Y \notin P$

## Vertex-Cover

1. **Vertex cover** in $G$ is set of vertices s.t. every edge in $G$ has at least one endpoint in the set. Given an undirected graph $G = (V, E)$ and int $k$, does there exist set $S \subseteq V$ s.t. $|S| = k$ and every edge in $G$ has $\geq 1$ endpoint in $S$?
2. **Theorem**: Complement of vertex cover ($G \setminus V$-$C$) is IS.
   **Corollary**: Graph with $n$ vertices has IS of size $k$ iff it has vertex cover of size $n - k$.
   **Corollary**: complement of vertex cover of min. size is IS of maximum size.

## CNF, SAT

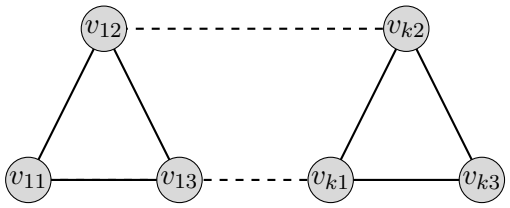
1. **Literal** is either a variable or its negation: $x$ or $\neg x$.
   **Clause** is a disjunction (OR) of literals: $x \vee y$.
2. A formula is in conjunctive normal form (**CNF**) if it's a outermost conjunction of clauses (AND/OR): $(x \vee y) \wedge (a \vee \neg b)$
3. **Theorem**: Every boolean formula $\phi$ has equivalent formula $\phi'$ in CNF.
4. **Boolean Satisfiability problem** (SAT): given boolean formula $\phi$ in CNF, is there an assignment of values to variables that makes $\phi =$YES? If so, $\phi$ is satisfiable.
5. **Theorem**: SAT is NP-complete.
6. **k-SAT** problem is SAT where each clause in $\phi$ has at most $k$ literals.
7. **Theorem**: 1-SAT is O$(n)$, 2-SAT $\in P$, 3-SAT is NP-complete.
8. **Theorem**: IS is NP-complete. **Proof**: IS $\in$ NP, prove that any one NP-complete problem can be reduced from 3-SAT to it.

## NP-completeness of IS

1. **Theorem**: Independent Set (IS) is NP-complete.
   Show IS $\in$ NP. Witness for IS is an IS $X$ of size $k$ in $G$.

   Can efficiently check it's a witness by: Confirming it's of size $k$ O(1), and Checking $\forall v, v' \in X : \{v, v'\} \notin E$ O$(n^2)$

   Show NP $\leq_p$ IS by showing NP $\leq_p$ 3-SAT $\leq_p$ IS. Suppose have 3-SAT instance $\phi$. Then construct graph $G$ as follows: For each literal create vertex, connect conflicting $(x, \neg x)$ literals or those in some clause by an edge.




   Return IS(G,k). **Proof of correctness**: Assume $\phi$ is satisfiable, then $\exists$ some assignment $A$ to $\phi$ s.t. $\phi(A) =$ True. Then at least one literal $\in A$ clause is True. Select one true literal from each clause, then corresponding vertices form an IS.

   Suppose $\exists$ IS of size $k$, then each vertex is in a different clause and no two are connected by a conflict edge. Assign True to literals that have vertices in IS. Then $\phi$ is True under that assignment. $\square$

## Set-Cover, NP problems

1. **Set-Cover** problem: given set $X$, powerset $S \subseteq \mathcal{P}(X)$ and int $k \leq |S|$, is there a set $S' \subseteq S$ of size $k$ s.t. $\cup S' = X$? Can cover the whole universe with subsets?
2. **Theorem**: Set-Cover is NP-complete. **Proof**: by reduction from vertex-cover to set-cover. For graph $G = (V, E)$, set $X = V; S = outEdge_i$ for $i^{th}$ vertex.
3. **Boolean Programming**: subset of integer programming where a solution for each variable $\in \{0, 1\}$ for some set of constraints.
4. **Feedback arc/node set** $f$ if $(V, E \setminus f)$ or for node set: $(V \setminus f, E)$ is acyclic.
5. **Graph colouring**: colour edges of a graph s.t. no two of the same colour are incident.

6. **Hamiltonian cycle**: visit every node exactly once and return to the same point.
7. **Hitting set**: choose least amount of subsets s.t. every value appears in all sets.
8. **Steiner-tree**: find an MST for a set $S \subseteq V$.
9. **Hypergraph Matching**: matching problem with edges each containing 3 vertices.
10. **Exact-cover**: non-overlapping set cover.
11. **Job-Sequencing**: matching numerous schedules.
12. **Clique-cover** - clique is opposite of independent set in that every vertex is adjacent to every other vertex. Clique cover - finding clique of size $k$.
13. **Max-cut** - finding a cut of maximum size in a graph.
14. SAT/3-SAT
15. **Set-packing**: select $k$ pairwise exclusive subsets.
16. **Partition**: split a set into two s.t. their sums are equal.
17. **Subset-sum** given a set $S$ and a number $n$, check whether $\exists$ subset $S'$ s.t. $\text{sum}(S') = n$.
18. **READ THE WIKI PAGE ABOUT EACH ONE**

# EXPTIME

1. **Corollary** There are no known problems that are in NP but definitely harder than P.
2. Problem is in **EXPTIME** (EXP) iff it admits an algorithm that can solve it in time $O(2^{f(n)})$ where $f(n)$ is some polynomial of $n$.
3. **Theorem**: 3-SAT is in EXPTIME. **Proof**: num of variables is $O(n)$ in size $n$ of the instance, and each can be assigned T,F. So, there are at most $2^n$ possible assignments, each can be checked in polynomial time. So the complete algorithm for 3-SAT is: generate each possibility in turn and check them until the term evaluates to T. It runs in time $O(2^{f(n)} \times n^k) \leq O(2^{g(n)})$. $\square$
4. **Theorem**: NP $\subseteq$ EXPTIME. **Proof**: Let problem $X \in$ NP. Since 3-SAT is NP-complete, there exists polynomial-time reduction from $X$ to 3-SAT, which is in EXPTIME, so $X$ can be solved in at most $O(2^{g(n)} \times n^l) = O(2^{h(n)})$ time, so $X \in$ EXPTIME. $\square$
5. **Corollary**: above algorithm uses polynomial space, so NP $\subseteq$ **PSPACE** - total needed amount of memory is bounded by a polynomial.
6. Restricting possible inputs to a problem makes it easier to solve. Hence, Tree-IS is in P.
7. Given a tree $T$ and number $k$, how to determine whether an IS exists of size $k$? Start at leaves and work up to the root. **Algorithm**: if $T$ is empty, return $k = 0$, else set $S = 0$, while $T$ has $\geq 1$ edge and $|S| < k$: select any leaf $v \in T$ and the edge $(u, v)$ incident to $v$, set $S = S \cup \{v\}$. Delete $u, v$ and all edges incident to $u, v$ from $T$. Let $S'$ be the union of $S$ and all remaining vertices in $T$. Return $S' \geq k$.
8. **Fixed Parameter Tractability** (FPT): fixing some parameter of the problem makes it tractable.

**Flow Networks**

1. **Flow Network** is a 5-tuple $(V, E, s, t, c)$ of vertices, edges (ordered pairs), $s \in V$ source vertex, $t \in V$ sink vertex and $c : E \to \mathbb{R}^+$ capacity function. Enforce $\deg_{in}(s) = 0$, $\deg_{out}(t) = 0$.
2. **Flow** is an assignment of number $f(e) \in \mathbb{R}^+$ to each edge $e \in E$ s.t. $\forall e \in E : f(e) \leq c(e)$ and $\forall v \in V \setminus \{s, t\}$: the **flow conservation** aka **Kirchoff constraint** holds:
$$\sum_{(u,v)\in E} f((u, v)) = \sum_{(v,u\in E)} f((v, u))$$
or total in = total out.
3. An $s$-$t$ **cut** is a partition $(A, B)$ of $V$ s.t. $s \in A, t \in B$. Capacity of an $s$-$t$ cut $(A, B)$ is the sum of capacities of edges from $A$ to $B$: $c((A, B)) = \sum_{(u,v)\in E} c((u, v))$ given $u \in A, v \in B$, back edges from $B$ to $A$ are ignored.
4. **Min-Cut** (search) problem: given flow network $N$, find an $s$-$t$ cut $C$ s.t. capacity of $C$ is minimal for $N$.
5. **Max-Flow** problem: given flow network $N$, find a flow $f$ s.t. value of $f$ is maximal for $N$. Value of $f$ is the sum of the net flow out of $s$ (second sum is usually 0), or the maximum total "volume" going through the network:
$$\text{val}(f) = \sum_{(s,v,)\in E} f((s, v)) - \sum_{(v,s,)\in E} f((v, s))$$
6. Given any cut the net flow doesn't change (flow value lemma)
7. **Max-flow greedy augmenting path algorithm**: start with flow value 0 everywhere, consider $s \rightsquigarrow t$ paths, "spend" the minimum remaining capacity of edges along that path. But may produce wrong paths: if optimal and sub-optimal paths share some part, and you encounter the sub-optimal one first - you "spend" your capacity on it, hence preventing from choosing the optimal path, so **incorrect**.

8. **Residual path**: a $p = s \rightsquigarrow t$ path with positive **residual capacity**, or the min free capacity along that path.

9. **Residual network**: $\forall e \in E$ and capacity $c_i$, it contains a **forward edge** with remaining capacity $r$, and **reverse edge** with used capacity $c_i - r$. When consider new paths, can now "take back" some of the "spent" flow, hence allowing for optimal solution.

10. **Ford-Fulkerson Algorithm** finds max-flow in network $N$. Start with flow of 0 everywhere, while $\exists$ an augmenting path $p = s \rightsquigarrow t$ in the residual network of $N$:
Let $c$ be min edge weight along $p$, for each forward edge in $p$, increase its flow by $c$, and decrease by $c$ for each reverse edge. Update the residual network accordingly.

11. Ford-Fulkerson: if augmenting path doesn't have reverse edges - it augments the network, if has at least one reverse edge, it reduces flow along some edges and reallocates it to others.
Since the path chosen was augmenting, it must still increase the flow overall. Doesn't get stuck in bad choices. Can use DFS $O(|E|)$ at most $O(\sum_c)$ times if augmenting by 1, so overall runtime: $O(|E| \times \sum_c)$, but exponential in input size (magnitude of the numbers). If weights are real-valued - might never terminate.

12. **Net-flow** across an $s$-$t$ cut $(A, B)$ is the sum of flow values of all edges from $A$ to $B$, minus the sum of those from $B$ to $A$.

13. **Flow Value Lemma**: let $f$ be any flow in flow network $N$, and let $(A, B)$ be any $s$-$t$ cut in $N$. Then, the value of $f$ equals to net flow across the cut $(A, B)$. Any "slice" between the source and the sink has "all the water" going through the system.

$$f = \sum_{s \rightsquigarrow e} f(e) - \sum_{e \rightsquigarrow s} f(e) = \sum_{v \in A} \left( \sum_{v \rightsquigarrow e} f(e) - \sum_{e \rightsquigarrow v} f(e) \right)$$

$$= \left( \sum_{A \rightsquigarrow e} f(e) - \sum_{e \rightsquigarrow A} f(e) \right) \quad \square$$

14. **Max-Flow-Min-Cut Theorem**: the maximum flow in $N$ is equal to the minimum capacity $s$-$t$ cut in $N$.

## Christmas Special

1. A matching $M \subseteq E$ is a set of edges s.t. $\forall v \in V : |\{\{$

# CheatSheet

## Runtime

1. Runtime comparison

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} : \begin{cases} 0 \to f(n) = O(g(n)) \\ \neq 0 \to f(n) = \Theta(g(n)) \\ \infty \to f(n) = \Omega(g(n)) \end{cases}$$

2. Log rules:

$$log_a x = \frac{log_b x}{log_b a}$$

$$a^{\log_b c} = c^{\log_b a}$$

3. Maximum number of nodes on the last level of a $k$-ary tree of height $h$ (assume root is level 0), is

$$k^h$$

4. Total number of nodes for a $k$-ary tree of height $h$ is:

$$\frac{k^{h+1} - a}{k - 1}$$

## Greedy

1. **Greedy rule**: simple local decision made at each step (e.g. pick shortest job, earliest deadline, etc.).
2. **Invariant**: property maintained throughout the algorithm or proof (e.g. greedy's prefix $\leq$ optimal's prefix)
3. **Stays-ahead**: compare first $r$ steps of greedy vs optimal ad show greedy's partial objective is at least as good.
4. **Inversion**: a local "bad" pair in a candidate solution (e.g. two adjacent jobs in wrong order by deadline)
5. **Exchange Argument**: Find an adjacent inverted pair, swap them, use an invariant (cost never goes up) to show each swap keeps us optimal, repeat until you match the greedy order.
6. Urgency-weighted ratio: given two jobs in a schedule, sort $R_j = \frac{u_i}{t_i}$ in non-increasing order for urgency $u$ and completion time $t$ to create an optimal schedule.
7. **Exchange argument proof template**
    1. Define **greedy strategy**: e.g. sort by $\frac{u_i}{t_i}$ in non-increasing order
    2. Define an **inversion**: e.g. $\frac{u_i}{t_i} > \frac{u_j}{t_j}$ but $i$ scheduled after $j$.
    3. Define **solutions**: e.g. Let $X$ be the greedy solution of our algorithm, and $X^*$ be the optimal solution (schedule).
    4. Compare solutions, show that if $X \neq X^*$, then

# Exchange Argument Proof Template

Let $\mathcal{S}$ denote the set of feasible solutions, and let $\Phi : \mathcal{S} \to \mathbb{R}$ be the objective function we wish to minimise (or maximise).

**1: Define the Greedy Strategy.** Clearly state the criterion your greedy algorithm optimises at each step.

*Example:* Schedule jobs in decreasing order of a key $K_i$.

**2: Define Inversions.** Clearly define what it means for a solution to have an inversion with respect to the greedy criterion.

Formally, given two adjacent elements $i, j$ in a solution, we say that the pair $(i, j)$ is an *inversion* if and only if:

$$K_i < K_j, \quad \text{yet element } i \text{ appears before element } j.$$

**3: Set up Solutions to Compare.** Define explicitly:

- $X$: The schedule produced by the greedy algorithm.
- $X^*$: An optimal schedule minimising (or maximising) the objective $\Phi$.

**4: Existence of an Adjacent Inversion.** Prove rigorously that if $X^* \neq X$, then $X^*$ must contain at least one inversion. Furthermore, show that there always exists at least one *adjacent inversion* by considering any existing inversion and moving step-by-step between jobs until an adjacent inversion is found.

**5: Swap Lemma.** Consider an adjacent inversion $(i, j)$ in the schedule $X^*$. Evaluate the objective function $\Phi$ before and after swapping these two adjacent elements:

$$\Phi(\dots, i, j, \dots) \quad \text{versus} \quad \Phi(\dots, j, i, \dots).$$

Demonstrate rigorously that swapping these elements *does not increase the cost* (for minimisation problems) or *does not decrease the cost* (for maximisation problems):

$$\Phi(\dots, i, j, \dots) \geq \Phi(\dots, j, i, \dots).$$

Conclude that the swap either strictly improves or does not worsen the solution.

**6: Iteration and Optimality.** Each swap reduces the number of inversions by exactly one and does not worsen the solution quality. Therefore, after finitely many swaps, the optimal solution $X^*$ transforms into the greedy solution $X$. It follows rigorously that the greedy solution $X$ is optimal.

## Graphs

1. **Cycle property**: Let $G$ and $T$ be a graph and its MST. Let $C$ be any cycle in $G$. Let $f \in C$ be the maximum cost edge in the cycle. Then, $f \notin T$, as it can be excluded (red edge).

2. **Cut property**: Let $G$ and $T$ be a graph and its MST. Let $S$ be any subset of $G$. Let $e \in S$ be the minimum cost edge with exactly 1 endpoint in $S$. Then, $e \in T$, as it can be included (blue edge).

3. MST will remain an MST if you apply the same monotonically increasing function to each of the edge weights: e.g. $w^2, w+5, \sqrt{w}$

4. **Dijkstra's**: finds shortest distance from start node $s$ to every other node in non-negatively-weighted ($\forall e \in E : w[e] \geq 0$) graph $G = (V, E)$.

   Maintain known shortest distances $d[s, v_i]$ from $s$ to each node $v_i \in V$. At each iteration, choose the shortest distanced vertex, and use it to find distances from it to other nodes: $d[s, u] = \min d[s, v_i] + w[v_i, u]$.

   Mark visited, continue until all nodes expanded.

## Divide and Conquer

1. **Recursion tree**: $T(n) = aT(n/b) + \Theta(f(n))$ has $\log_b n$ levels $L_i$, each containing $a^i$ subproblems of size $n/b^i$. Each subproblem contributes $\Theta(f(n/b^i))$ work. Expand the tree, find the pattern, sum the work per level: $a^i \cdot \Theta(f(n/b^i))$. Sum over all levels to find the total work:

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot \Theta\left(f\left(\frac{n}{b^i}\right)\right).$$

2. **Telescoping**: substitute values of $n$ derived from each recursive term, rewrite the formula and continue until a pattern emerges. $T(n) = T(n-1) + T(\frac{n}{2}) + \Theta(1)$ becomes

## P, NP, Reductions

1. **P**: PTIME problem, **NP**: easy-to-check problems, **NP-complete**: NP, reducible from any other NP-complete problem, **NP-hard**: at least as hard as any NP-complete, but doesn't have to be NP.

2. Decision problem $X \in$ NP iff $\exists$ **verifier** $C$ s.t.:
   • $C(i, w)$ takes 2 inputs: problem instance $i$ and a candidate witness $w$.
   • $C$ runs in polynomial time.
   • If answer to $X(i)$ is NO, then $\forall i, w : C(i, w) =$ NO.
   • If answer to $X(i)$ is YES, then $\exists$ some witness $w$ for $i$ s.t. $C(i, w) =$ YES.

3. **Oracle** of a problem $Y$ is a black box, we can only put inputs and get outputs, not a specific algorithm.

4. **Reduction** is description of solving a problem via solution (oracle) to another problem.
   $X \leq_P Y$ ($X$ polynomial-time reduces to $Y$) iff any instance $i$ of $X$ can be solved using **polynomial** number of calls to oracle for $Y$ + polynomial number of standard computational steps.
   Instances $i'$ of $Y$ must be polynomial-sized. Simply, solve $X$ by solving $Y$ and doing polynomial time of extra work.

5. $X \leq_P Y$ **Reduction proof template:**
   1. Convert problem instance $i_1$ of $X$ into instnce $i_2$ of $Y$.
   2. **Prove Correctness**: $i_1$ is YES iff $i_2$ is YES:
      *a)* Suppose $i_1$ is YES, then ... $i_2$ is YES.
      *b)* Suppose $i_2$ is YES - ... then $i_1$ is YES.
   3. Show that reduction takes PTIME.

6. Problem $X$ is **NP-complete** if $X \in$ NP and every other NP $Y$ problem is reducible to it $\forall Y \in$ NP : $Y \leq_P X$.
   Or, if any **one** NP-complete problem reducible $Z \leq_P X$.

7. **Theorem**: $(X \leq_P Y) \wedge (Y \in P) \rightarrow X \in P$
   **Corollary**: $(X \leq_P Y) \wedge (X \notin P) \rightarrow Y \notin P$

   Prove if $A$ **can** be solved in PTIME/**has** PTIME algorithm, then $B$ does: **POSITIVE: reduce** $B \leq_P A$

   Prove if $A$ **can't** be solved in PTIME/**doesn't** have PTIME algorithm: **NEGATIVE: reduce** $A \leq_P B$

8. *Packing problems:*
   • Independent-Set $S$: $\forall (u, v) \in E : \neg(u \in S \wedge v \in S)$.
   Set of vertices s.t. no two vertices are adjacent.

   *Covering problems:*
   • Vertex-Cover $S$: $\forall (u, v) \in E : u \in S \vee v \in S$.
   Set of vertices touching every edge. Complement of IS.

   • Set-Cover: given a set $U$ of $n$ elements, its subsets $S_1, ..S_m$ and a number $k$, does there exist a collection of $\leq k$ of these sets whose union is equal to all of $U$?

   *Constraint Satisfaction problems:*
   • k-SAT: is a CNF boolean formula $\phi$ with clauses of at most $k$ literals satisfiable?