I. Geometric Series

$$\sum_{k=0} ar^k = a(\frac{1-r^{n+1}}{1-r})$$
$$= \frac{a}{1-r} \forall |r| < 1$$

II. SMP

It is always true that employers with the same preferences will have the same stable matching.

Proof. Prove inductively, remove the most stable pair, then the remaining employers and employees have the same preferences. Continue until all pairs are removed. □

III. Asymptotic

log < polynomial < exponential < factorial

IV. Greedy Algorithms

Chase only the local optimum, not the global optimum.

A. Greedy stays ahead

If a greedy algorithm always makes a choice that stays ahead (at least as good) of the optimal solution, then the greedy algorithm is optimal.

Inductively show that at each stage, the greedy solution is at least as good as the optimal solution.

V. Divide and Conquer

Divide the input into smaller instances (subproblems), solve recursively, then combine to get the solution.

A. Recurrence Relations

1) Recurrence Tree: Each level of the tree represents the cost of the work done at that level. The total cost is the sum of the costs at each level. Usually $\log n$ levels with a $f(n)$ cost at each level.

2) Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2) If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. Note the **+1** in the exponent.
3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

B. Prune and Search

Definition. ***Prune and Search*** *is a problem-solving strategy that divides the search space into two parts, one of which is pruned (discarded) and the other is searched.*

To find the kth smallest element in an array, we can use the QuickSelect algorithm. The algorithm is as follows:

1) FunctionQuickSelect($A[1:n], k$)
2) If $n = 1$ return $A[1]$
3) Choose a random pivot element p from A
4) Partition A into $L = \{x \in A : x < p\}$, $E = \{x \in A : x = p\}$, $G = \{x \in A : x > p\}$
5) If $k \leq |L|$, return QuickSelect(L, k)
6) Else if $k \leq |L| + |E|$, return p
7) Else return QuickSelect($G, k - |L| - |E|$)
8) End Function

VI. Dynamic Programming

Typical DP examples:

$$\text{DP}[i] = \begin{cases} \text{base case} & \text{start} \\ \text{combine}(\text{DP}[i-1], \text{DP}[i-2], \ldots, \text{DP}[i-k]) & \text{otherwise} \end{cases}$$

Or

$$\text{OPT}(j) = \max\{\text{OPT}(j-1), \text{OPT}(j-2) + v_j\}$$

Iterative starts with the base case and builds up to the solution. Recursive starts with the solution and breaks it down to the base case.

VII. NP-Completeness

As long as the algorithm runs in $O(n^k)$ time, it is considered polynomial time. We think it is efficient.

A. Decision V.S. Optimization

Definition. *Optimization problem: we want to find the solution s that maximizes or minimizes some objective function $f(s)$.*

Definition. *Decision problem: given a parameter k, we want to know if there is a solution s such that $f(s) \geq k$ (maximization) or $f(s) \leq k$ (minimization).*

B. P and NP

Definition. ***P*** *is the class of decision problems that can be solved in polynomial time. (We have the solver)*

Definition. ***NP*** *is the class of decision problems for which a solution can be verified in polynomial time. (We have the verifier)*

The NP-complete problems are the NP problems with the property that, if we can solve this problem in polynomial time, we can solve any problem in NP in polynomial time.

C. Proof of NP-completeness

1) Show that the problem is in NP.
2) Show that the problem is in NP-hard.

1) Proof of NP: To prove that P is in NP, we need to **efficiently verify** a **certificate**.

- A certificate is a proof that the solution is correct.
- A verifier is an algorithm that checks the certificate. Needs to run in polynomial time.
- For example, an optimization problem can be: Does there exists X such that Y is true?
  - X is the certificate.
  - Y is the verifier.

2) Proof of NP-hard:

Definition. *A problem A is **NP-hard** if P is at least as hard as any other problem in NP.*

We prove "at least as hard" via polynomial-time reduction.

- Pick a known NP-complete problem B. Let A be the problem we want to prove is NP-hard.
- Show that B can be reduced to A in polynomial time with the same YES/NO answer.
- Since B is NP-complete, A is NP-hard.
- Intuition: Since we can use a solver for A to solve B, this tells us that A is at least as hard as B (It is powerful enough to handle B)

VIII. Good to know

- $f(n) \in \theta(g(n)) \implies 2^{f(n)} \in \theta(2^{g(n)})$
- Spanning tree with two leaves is Hamiltonian cycle.
- Watch out for **All matchings/ set of possible answers**
- polynomial time $\times$ polynomial time = polynomial time. If $P \neq NP$, then not solvable in polynomial time.
- Recursion will have a **base case** and a **recursive case** where the problem is **broken down**.
- Reduction from A to B just needs all A instances to be transformed to some specific B instances.
- QuickSort: $O(n^2)$ worst case, $O(n \log n)$ average case.
  1) Find a pivot element.
  2) Partition the array into two parts: elements less than the pivot and elements greater than the pivot.
  3) Recursively sort the two parts. Then combine.
- 4 SAT to 3 SAT: $(a \vee b \vee c \vee d) = (a \vee b \vee x) \wedge (\neg x \vee c \vee d)$
- To memorize a DP solution, use a 2D array.
- Recursive to iterative: watch out for $1 \to n$ or $n \to 1$. Solve the base case first.
- Choose function: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \in O(n^i)$. Some polynomials.
- A memorized recursive solution can use a helper method which takes in the array and the index. If still the initialized value, then calculate it.
- If you can reduce a problem to another NP-complete problem, then it is in NP.
- Try not to set a reward/weight to 0. It might cause problems.

A. Some NP-complete problems

1) Sequencing Problems:

1) The traveling salesman problem (TSP): Given a list of cities and the distances between them, what is the shortest possible route that visits each city exactly once and returns to the origin city?
2) The Hamiltonian cycle problem: Given a graph, does there exist a cycle that visits each vertex exactly once?
3) The Hamiltonian path problem: Given a graph, does there exist a path that visits each vertex exactly once?

2) Partition Problems:

1) The 3 Dimensional Matching problem: Given three sets X, Y, Z each with n elements, and a set $T \subseteq X \times Y \times Z$, is there a subset $T' \subseteq T$ such that $|T'| = n$ and no two elements in T' share a common index?
2) MMCP: Given a set of n items, each with a weight and a value, and a knapsack of capacity W, what is the most valuable subset of the items that can fit into the knapsack?
3) Graph colouring (Not NP-complete, but 3-colouring is NP-complete): Given an undirected graph, is it possible to colour the vertices using at most k colours such that no two adjacent vertices share the same colour?

3) Numerical Problems:

1) The subset sum problem: Given a set of integers, is there a non-empty subset whose sum is zero?
2) Schedule problems: Given a set of tasks, each with a start time, end time, and a weight, what is the maximum weight subset of tasks that can be scheduled without overlapping?

4) Packing Problems:

1) Independent set: Given a graph, is there a set of vertices such that no two vertices are adjacent?
2) Set Packing: Given a set of sets, is there a subset of the sets such that no two sets share a common element?

5) Covering Problems:

1) Hiiting Set: Given a set of sets, is there a subset of the sets such that every element is covered at least once?
2) Dominating Set: Given a graph, is there a set of vertices such that every vertex is either in the set or adjacent to a vertex in the set?
3) Vertex Cover: Given a graph, is there a set of vertices such that every edge is incident to at least one vertex in the set?
4) Set Cover: Given a set of sets, is there a subset of the sets such that every element is covered at least once?