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 $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
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)*

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)