Logarithm Basics; Asymptotic Notation; Master Theorem

- **Big-O** is the upper bound, **Big-Θ** is the tightest bound, **Big-Ω** is the lower bound
- $\mathcal{O}(1) < \mathcal{O}(\log(\log(n))) < \mathcal{O}((\log(n))^{c_1}) < \mathcal{O}(n^{c_2}) < \mathcal{O}(n) < \mathcal{O}(n\log n) < \mathcal{O}(\log n!) < \mathcal{O}(n^2) < \mathcal{O}(n^{c_3}) < \mathcal{O}({c_1}^n) < \mathcal{O}(n!)$; where $c_1 \geq 1$ and $0 < c_1 < 1$ and $c_3 > 2$
- $b^y = a, \log_b(a) = y$; $\log(x*y) = \log(x) + \log(y)$; $\log(x/y) = \log(x) - \log(y)$; $\log(x^a) = a\log(x)$
- **Master Theorem**: $T(n) = aT(\lceil(n/b)\rceil) + \mathcal{O}(n^d)$ where $a > 0; b > 1; d > 0$
  a is the number of subproblems ; n/b is the size of the subproblem ; n^d time for combining the subproblems

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{when } d > \log_b(a) \\ \mathcal{O}(n^d\log(n)) & \text{when } d = \log_b(a) \\ \mathcal{O}(n^{\log_b(a)}) & \text{when } d < \log_b(a) \end{cases}$$

Divide and Conquer

- The general design recipe:
  1. Divide : break the problem into subproblems that are an instance of the same problem
  2. Conquer: Solve the subproblems recursively, using the base cases in which the problem can easily be solved
  3. Combine : Combine the problems cleverly
- Proof of correctness: by induction
- Merge Sort takes $\mathcal{O}(nlogn)$; $T(n) = 2T(n/2) + c*n$

```
function Merge(X[1, .., k], Y[1, ... l])
    if k = 0 then return Y[1, ..., l]            ▷ X is empty; base case 1
    if l = 0 then return X[1, ..., l]            ▷ Y is empty; base case 2
    if X[1] ≤ Y[1] then return MergeSort(X[1] + Merge(X[2,.., k], Y[1,.., l]))
    else return MergeSort(Y[1] + Merge(X[1,.., k], Y[2,.., l]))
function MergeSort(A[1, ..., n])
    if n > 1 then return Merge(MergeSort(A[1,..,n/2], MergeSort(A[(n/2)+1,.., n]))
    else return A
```

- Proof of correctness: by induction
  - base case : n = 1 ; algorithm does nothing correctly sorts
  - induction : assume mergesort correctly sorts arrays of size n-1, then since $m < n-1$ A[1,..,m] and A[m+1,..,n] are sorted correctly
  - lemma: merge correctly merges the two sorted arrays
- In every divide and conquer assume recursion gives the correct solution, and focus on how to combine solution

Greedy Algorithms

- The general design recipe:
  - make whatever seems optimal at the moment and not worry too much about future consequences
  - build piece by piece, choose the next piece which offers most obvious and immediate effect
  - try with a simple example first
- Proof of correctness (1) : use either the exchange argument (left), or the greedy stays ahead argument (right).
  - assume optimal solution O is 'out of order'
  - exchange two of the out of elements to get O'
  - argue that cost of O' ¡ cost of O to get a contradiction
  - A : greedy solution with size k (i is index of element)
  - O : optimal solution with size m (j is index of element)
  - show F[i_r] $\leq$ F[j_r] $\forall r$ such that $1 \leq r \leq k$ if m > k adding j_{k+1} to m is a contradiction
- Minimum Spanning Trees (MST) is a connected graph with no cycles and has minimal cost ; Kruskal to get MST ; Union Find data structure; Prim to get MST uses priority queues
  - Greedy Algo to get MST of a graph (Kruskals) : repeatedly choose the edge lightest that does not produce a cycle.
  - Cut property: cut is any partition of vertices to two groups ; in MST the cut property says that it is safe to add lightest edge in cut
  - Union find:
    1. makeset(x) : makes a set x $\mathcal{O}(1)$
    2. Union(A, B) : $\mathcal{O}(1)$
    3. Find(x) : returns the set that contains x $\mathcal{O}(\log(n))$

```
function Prim(G, w)                              ▷ Prim to get MST ; uses queues
    for u ∈ V do
        cost(u) = ∞
        prev(u) = nill
    any node u_0 cost(u_0) = 0
    H = makequeue(V)
    while H is not empty do
        v = delmin(H)                            ▷ get the vertex with min cost
        for (v, z) ∈ E do
            if cost(z) > w(v,z) then
                cost(z) = w(v,z)
                prev(z) = v
                decreasekey(H, z)
    return X
```

- Matriod : M = ($\mathcal{S}$, $\mathcal{I}$) such that :
  1. $\mathcal{S}$ is finite (it is a set)
  2. $\mathcal{I}$ is non-empty , hereditary , $\phi \in \mathcal{I}$
  3. Exchange Property for $A \in \mathcal{I}$ and $B \in \mathcal{I}$ and $|A| < |B|$ then $\exists x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$
  4. $A \in \mathcal{I}$ is a basis if A is not a proper subset of a set in $\mathcal{I}$

  In a graphic matriod:
  1. non emptiness : no cycles, $\phi \in \mathcal{I}$
  2. (V, I) acyclic so is (V, I') $\forall$ $\mathcal{I}' \subseteq \mathcal{I}$
  3. Exchange don't create cycles

Dynamic Programming "smart recursion"

- The general design recipe:
  - identify collection of subproblems and tackle
  - common subproblems and their runtimes :
    + input : $x_1, ..., x_n$, subproblem $x_1, ..., x_j$ takes $\mathcal{O}(n)$
    + inputs : $x_1, ..., x_n$ and $y_1, ..., y_m$, subproblem $x_1, ..., x_j$ and $y_1, ..., y_j$ takes $\mathcal{O}(mn)$
    + input : $x_1, ..., x_n$, subproblem $x_i, ..., x_j$ takes $\mathcal{O}(n^2)$
    + input : tree , subproblem is any subtree
  - determine subproblems. write recurrences, update DP table (be careful with update order)
  - choice of whether the item belongs to the solution at each step
- Proof of correctness is by induction

```
Alternating Sign Subsequence sign(a) = 1 if a > 0 otherwise sign(a) = 0
    base cases
    F[0,j,k] = 0 ∀1 ≤ j ≤ n, k ∈ {0, 1}
    F[i,0,k] = 0 ∀1 ≤ i ≤ n, k ∈ {0, 1}
    function AltSignSubSeq(A[1, ..., n], B[1,...n])
        F[i, j, k] = 0 ∀1 ≤ i ≤ n, 1 ≤ j ≤ n, k ∈ {0, 1}
        for i from 1 to n do
            for j from 1 to n+1 do
                if A[i] = B[i] and sign(A[i]) = 1 then
                    F[i, j, 1] = max(1+F[i-1, j-1, 0], F[i-1, j, 1], F[i, j-1, 1])
                    F[i, j, 0] = max(F[i-1, j, 0], F[i, j-1, 0])
                else if i thenf A[i] = B[i] and sign(A[i]) = 0
                    F[i, j, 0] = max(1+F[i-1, j-1, 1], F[i-1, j, 0], F[i, j-1, 0])
                    F[i, j, 1] = max(F[i-1, j, 1], F[i, j-1, 1])
                else
                    F[i, j, 0] = max(F[i-1, j, 0], F[i, j-1, 0])
                    F[i, j, 1] = max(F[i-1, j, 1], F[i, j-1, 1])
        return max(F[n, n, 0], F[n, n, 1])
```

Randomized Algorithms

- Some Facts of Probability and Expectations:

- $\binom{n}{k}$: choose k from n
- $(n/k)^k \leq \binom{n}{k} \leq (en/k)^k$
- $P[A \cup B] = P[A] + P[B] - P[A \cap B]$
- if A and B are mutually exclusive $P[A \cap B] = 0$
- $P[A \cap B] = P[A] * P[B|A] = P[B] * P[A|B]$
- $E[X] = \sum_x xP[X=x] = \sum_{i=0}^{\inf} iP[X=i]$ if integer values
- $E[X+Y] = E[X] + E[Y]$
- $E[XY] = E[X]E[Y]$ if X and Y are independent
- $Var[X] \geq 0$; $Var[X] = E[X^2] - E[X]^2$
- $Var[aX] = a^2 Var[X]$
- $Var[\sum_{i=1}^n X_i] = \sum_{i=1}^n Var[X_i]$
- $P[X > k] = 1 - P[X \leq k]$
- $T(n)$ is the runtime random variable, to get the runtime, we want to get $E[T(n)]$
- binary (0, 1) random variable which indicate an event occurring or not

- Union Bound : used to get the upper bound of the probability of a union of events ; $P[\bigcup_{i=1}^n X_i] \leq \sum_{i=1}^n P[X_i] \leq nP[X_i]$
- Balls (m) and Bins (n) example: X_i^j which is 1 if ball i lands in bin j ; when m = n, $E[X^j] = m/n$; using union bound to get the $P[X_1 = k] = \binom{n}{k} * (1/n)^k * (1 - 1/n)^{n-k}$; $P[X_1 \geq k] = 1 - P[X_1 \leq k] = \binom{n}{k} * (1/n)^k \leq 1/k!$; in union bound the probability that there exists a bin i with at least k balls is : $P[X_i \geq k] \leq n * [X_1 \geq k] \leq n/k!$
- In Quick sort we have runtime of $\Theta(n^2)$ in the worst case ; (1) choose a random pivot (p), (2) partition into two subrrays : elements $\leq$ p and elements > p, (3) sort the two subarrays recursively ; $X_{ij} = 1$ if elem i is compared to elem j, 0 otherwise ;X_i represents the number of comparisons to i; so the expected runtime is $\sum_{i=1}^n \sum_{i=1}^n X_{ij} = E[X] = n \log n$
- In randomized algorithms think about what we are making a random choice about what is the key question ; in analysis focus on upper bound ; certain decisions are made based on coin flips outcomes in algorithm ; analysis is done without assuming anything about input distribution ; done in space of all possible outcomes for coin flips made in Algorithm
- Two main types of Randomised algorithms : Monte Carlo (decision problem correct with high probability) and Las Vegas (always correct, runtime is the random variable like Quick Sort)

Network Flow and its applications

- f(u, v) is the flow from u to v , c is the capacity, s is the source, t is the sink.
- flow conservation constraint: the property that no vertex, except the source and sink, of a flow network creates or stores flow. More formally, the incoming flow is the same as the outgoing flow, or, the net flow is 0 . For v $\neq$ s , t ; $\sum_u f(u,v) = \sum_w f(v,w)$
- $|f| = \sum_u f(u,t) - \sum_w f(t,w)$ the network flow to t ; maxflow wants to maximize this flow
- $0 \leq f(u,v) \leq c(u,v)$ for all edges (u, v)
- **augmenting path** is a simple path from s to t in the residual graph G
- Runtime of Ford-Fulkerson : $\mathcal{O}(EVC)$ which is psuedo polynomial ; E is the number of edges, V is the number of node, C is the maximum capacity ; If we choose the augmenting path cleverly we can reduce the runtime to polynomial time. If we use BFS, maxflow runtime = $\mathcal{O}(E^2V)$
- in general C(S, T) $\leq |f|$ where C is the capacity of the cut
- Theorem : There is **no** path from s to t in the residual graph means: (Max-flow, Min-cut)
  1. the flow is maximal
  2. let S be the set of all nodes which are reachable **from** s and includes s, let T be all the other nodes. Then C(S,T) in a min-cut which is equal to the flow (the bottle neck)
- Maxflow applications : reduce the problem to maxflow; normally it is a problem that wants to assign values given some constraints.
  - what does the network look like?
  - what are the capacities of the edges?
  - what should the value of the flow be to satisfy the constraints?
  - prove that the solution to maxflow is the solution of the problem

Linear Programming (LP)

- want to minimize cost or maximize objective given some constraints that are linear in the optimization variables
- Simplex Algorithm to solve the LP: choose origin as vertex, if all the constraints are ≤ 0 then the origin is optimal. otherwise choose a new neighbor vertex by incrementing a variable x_i by some delta and when the constraint becomes tight, change the coordinate system so that the new vertex is the origin.

Primal LP

maximize $4x_1 + x_2 + 3x_3$
s.t. $x_1 + 2x_2 \leq 1$
$3x_1 - x_2 + x_3 \leq 4$
$x_1, x_2, x_3 \geq 0$

$c = \begin{pmatrix} 4 \\ 1 \\ 3 \end{pmatrix}, A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & -1 & 1 \end{pmatrix}$

$b = \begin{pmatrix} 1 \\ 4 \end{pmatrix}$

Dual LP

minimize $y_1 + 4y_2$
s.t. $y_1 + 3y_2 \leq 3$
$\begin{pmatrix} 1 & 3 \\ 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \geq \begin{pmatrix} 4 \\ 1 \\ 3 \end{pmatrix}$
$y_1, y_2 \geq 0$

minimize $y_1 + 4y_2$
s.t. $y_1 + 3y_2 \geq 4$
$2y_1 - y_2 \geq 1$
$y_2 \geq 3$
$y_1, y_2 \geq 0$

Primal	Dual
max $c^\intercal x$	min $b^\intercal y$
$\sum_j a_{ij}x_j \leq b_i$	$y_i \geq 0$
$\sum_j a_{ij}x_j \geq b_i$	$y_i \leq 0$
$\sum_j a_{ij}x_j = b_j$	N/A
$x_j \geq 0$	$\sum_i a_{ij}y_i \geq c_j$
$x_j \leq 0$	$\sum_i a_{ij}y_i \leq c_j$
N/A	$\sum_i a_{ij}y_i = c_j$
$x_j = 0$	N/A

- Primal and Dual

(a) (20 points) Write a linear program (LP) to find a dominating set of minimum size of G. Hint: your LP should have one variable for each vertex of G; do not worry about whether the solution of the LP is integral or fractional.
Let x_v be the variable indicating whether the vertex v is in the dominating set or not.

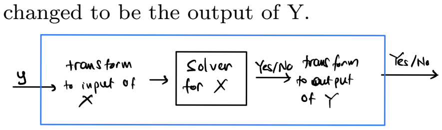
$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} x_v \\ \text{subject to:} \quad & \sum_{u \in N_v} x_u \geq 1 \quad \forall\, v \in V \\ & x_v \geq 0 \quad \forall\, v \in V \end{aligned} \tag{3}$$

(b) (20 points) Find the dual of your LP.
We have a dual variable y_{N_v} corresponding to the set N_v of each vertex $v \in V$. For each vertex $u \in V$, let $\mathcal{F}_u = \{N_v : u \in N_v\}$ be the collection of all sets that contain u.

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} y_{N_v} \\ \text{subject to:} \quad & \sum_{N_v \in \mathcal{F}_u} y_{N_v} \leq 1 \quad \forall\, u \in V \\ & y_{N_v} \geq 0 \quad \forall\, v \in V \end{aligned} \tag{4}$$

P vs NP ; NP-Complete, NP-Hard

- a decision problem is a problem whose answer is YES/NO ; for example : does graph G, have an independent set of size k, an independent set means that we have a set of vertices such that no two vertices share an edge.
- P : class of decision problems that can be solved in polynomial time
- NP : class of decision problems whose YES instance can be verified in polynomial time
- co-NP : class of decision problems whose NO instance can be verified in polynomial time
- $Y \leq_p X$ means Y is polynomial time reducible to X ; for any two problem $Y \in P$ and $X \in P$ then $Y \leq_p X$
- $Y \leq_p X$ if given a black box for solving X, we can solve Y by changing the input of Y to be the input of X and the output from the black box of X is changed to be the output of Y.



an algorithm for solving Y

- $X \in$ NP-Complete if :
  1. $X \in NP$
  2. $\forall\, Y \in NP$ $Y \leq_p X$ / for any $Y \in$ NP-Complete $Y \leq_p X$
- $X \in$ NP-Hard if :
  1. unknown if $X \in NP$
  2. for any $Y \in$ NP-Hard $Y \leq_p X$
- for any reduction $Y \leq_p X$ we need to prove the correctness of reduction by showing $y \in$ YES/correct $\iff$ $x \in$ YES/correct where y and x are the solutions to the problems Y and X respectively

Approximation Algorithms

- Approximation Algorithms are polynomial time algorithms which approximate the solution to an optimization problem which is NP-Hard.
- Approximation ratio $\alpha(n) \geq \frac{A(x)}{OPT(x)}$ for some input x when we want to minimize the cost
- Approximation ratio $\alpha(n) \geq \frac{OPT(x)}{A(x)}$ for some input x when we want to maximize the objective
- want to minimize the approximation ratio
- to get the approximation ratio, look at the upper and lower bounds of $OPT(x)$; and bounds of $A(x)$. We want to define some facts then get the bounds of $A(x)$ based on facts
- Traveling Sales Problem (TSP) does not have a polynomial time approximation ratio but metric TSP has a 2 and 1.5 approximation algorithm using Eulers tour (needs even degree vertices) and MST doubling