Stable Marriage

- Worst case scenario we check every man and every woman $\Rightarrow O(|M||N|)$
- Brute force complexity: $O(n!)$
- Observation 1: Men propose to women in decreasing order of preference.
- Observation 2: Once a woman is matched, she never becomes unmatched, she only "trades up"
- Thm (Man-Optimality): The GS algorithm always pairs a man with his best valid partner, say m prefers w_0 to w_1 since GS matched m with w_0
- Thm (Woman-Pessimality): The GS algorithm always pairs a woman with her worst valid partner. This means that out of all the men that w could have been matched with that produces a stable matching, GS will match w with the least preferred out of these.

Complexity

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. $\Omega(g(n))$ if $f(n) \geq cg(n)$. $\Theta(g(n))$ if both $O(g(n))$ and $\Omega(g(n))$
- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$
- $f(n)$ is $o(g(n))$ if $f(n)$ is asymptotically **less than** $g(n)$
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$
- $f(n)$ is $\omega(g(n))$ if $f(n)$ is asymptotically **greater than** $g(n)$
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal** to $g(n)$

Formulas to calculate bounds

- $\lim_{n\to\infty}[f(n)/g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty}[f(n)/g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty}[f(n)/g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty}[f(n)/g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty}[f(n)/g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$

Properties

- $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$, etc
- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$, $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Growth Rate

$1 \quad \log n \quad \sqrt{n} \quad n \quad n\log n \quad n^2 \quad n^3 \quad \dots \quad n^c \quad 2^n \quad 3^n \quad \dots \quad c^n \quad n! \quad n^n$

Logarithm and Exponent Rules

- Natural log $\ln a = \log_e a$
- Binary log $lga = \log_2 a$
- $lg^2 a = (lg\ a)^2$
- $lg\ lg\ a = lg(lg\ a)$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c + b$
- $\log_b a^n = n\log_b a$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_b(1/a) = -\log_b a$
- $\log_b a = \frac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$
- $a^{-1} = \frac{1}{a}$
- $(a^m)^n = a^{mn}$
- $a^m a^n = a^{m+n}$
- $\lim_{n\to\infty}\frac{n^b}{a^n} = 0 \Rightarrow n^b \Rightarrow o(a^n)$

- Taking the log to compare the complexity of exponential functions is not always a good idea - Consider 3^n and 2^n
  - if you take log of both sides then you get $n\log 3$ and $n\log 2$ but this can be simplified and means that they both have proportional growth rate which is wrong
  - but if you use limit
  - $\lim_{n\to\infty}\frac{2^{n\log 3}}{2^{n\log 2}} = 2^{n(log3-log2)} = \infty$

Graphs

- $n = |V|$, $m = |E|$
- Complete Graph: every pair of distinct vertices is connected by a unique edge
  - Max n Undirected: $\frac{n(n-1)}{2}$
  - Max n Directed: $n(n-1)$
- Adjacency Matrix
  - $n \times n$ matrix with $A_{uv} = 1$ if (u, v) is an edge
  - two representations of each edge
  - Space proportional to n^2
  - Checking if (u, v) is an edge takes $O(1)$
  - Identifying all edges takes $O(n^2)$
- Adjacency List
  - Node indexed array of lists
  - two representations of each edge
  - Space proportional to $m + n$
  - Checking if (u, v) is an edge takes $O(1 + deg(u))$ where $deg(u)$ is the degree = number of neighbors of u
  - Identifying all edges takes $O(2m + n)$
    - n: total number of nodes (number of pointers in list)
    - $2m$: since we are counting each edge twice
- Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected
  - G does not contain a cycle
  - G has $n - 1$ edges
- The height of a tree is the number of edges in the longest level. Height of empty tree is -1
- level of root is 0
- every n node tree has $n - 1$ edges
- Can determine if G is strongly connected (every pair nodes mutually reachable) in $O(m + n)$ time
  - Pick an node s
  - Run BFS from s in G
  - Run BFS from s in G^{rev} (rev = reverse orientation of every edge in G)
  - Return true iff all nodes reached in both BFS executions
- An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end
- A graph G is bipartite iff it contain no odd length cycle
- A **DAG** (directed acyclic graph) is a directed graph that contains no directed cycles
- A **topological order** of a directed graph G is an ordering of its nodes so that for every edge (v_i, v_j) we have $i < j$, i.e. they have some natural ordering
- if G has a topological order, then G is a DAG
- if G is a DAG then G has a topological ordering
- If G is a DAG then G has a node with no incoming edges

Paths

- BFS
  - $O(m + n)$ if graph given by it's adjacency representation
  - total time processing edges is $\sum_{u\in V} deg(u) = 2m$
  - Use a PQ, dequeue, loop through neighbors, if not visited then mark as visited, set distance to parent.currdistance + 1 and enqueue
- DFS
  - Use a stack, pop, if popped not visited, mark as visited, loop through neighbors and push neighbors to stack
- Let G be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds:
  - No edge of G joins two nodes of the same layer, and G is bipartite
  - An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite)

Greedy Algorithms

- Exchange Argument: Every algorithm can be gradually transformed to the greedy one without hurting it's quality
- Stays Ahead: In every step the greedy is no worse than the optimal
- Interval Scheduling Problem:
  - We have some set of jobs each with a start, s_i, and a finish, f_i time. Two jobs are compatible if they don't overlap
  - Obtain maximum number of mutually compatible jobs
  - Optimal Solution: Earliest Finish Time
  - $O(n \log n)$
- Interval Partitioning Problem:
  - Given a set of lectures where each lecture j has start time s_j and finish time f_j
  - each lecture needs a classroom
  - each classroom can have one lecture at same time
  - Obtain the minimum number of classrooms to schedule all lectures such that no two lectures occur at the same time
  - Depth of a set of open intervals: maximum number that passes over any single point in the time-line
  - key observation: number of classrooms needed $\geq$ depth
  - If the interval under consideration overlaps with all preceding intervals we need to allocate a new classroom
  - $O(n \log n)$
  - greedy never scheduled two incompatible lectures in same room
- Scheduling to Minimize Lateness:
  - We have a single resource which processes one job at a time, job j requires t_j units of processing time and is due at time d_j
  - if j starts at s_j and finishes at $f_j = s_j + t_j$ time. **Lateness** is measured using the formula $\max(0, f_j - d_j)$
  - Schedule all jobs in such a way as to minimize the lateness $L_j = \max(0, f_j - d_j)$
  - Earliest Deadline First
  - Lateness does not change if you swap jobs with the same deadline
  - deadlines are arbitrary and might be less then the processing times
  - there exists an optimal schedule with no idle time
  - The schedule produced by the greedy has no inversions
  - **Inversions**: The next deadline $d + 1$ should never be less than d. A schedule A' has an inversion if a job i with a deadline d_i is scheduled before another job j with an earlier deadline such that $d_j < d_i$
  - All scheduled with no inversions and no idle time have the same lateness
- Dijkstra's Algorithm
  - Single-source shortest path algorithm
  - Assumes:
    - The start node s has a path to every other node
    - All edge weights are ≥ 0
  - $d(u)$: the distance of the shortest path from s to u
  - $\ell(u, v)$: the length of the edge $\ell \in E$
  - $O(mn)$, using priority heap can get to $O(m \log n)$
  - Bellman-Ford algorithm can be used if graph has negative edges (involved DP)

MST

- Given a graph $G = (V, E)$ find a subset of the edges T such that $T \subseteq E$ so that G is connected and the total cost of all edges $\sum c_e \in T$ is as small as possible
- Assumptions:
  - G is connected
  - All edges have non-negative weights
  - G is undirected
  - All edges have distinct weights
- Cut Lemma: Assume that the cost of all edges is distinct. Let S be a subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be a minimum cost edge with one node in S and the other in $V - S$. Then every minimum spanning tree will contain (v, w)