Proof of Correctness

Total Correctness: Termination and Partial Correctness Partial Correctness: Loop invariants and induction Loop Invariant: A property that holds before and after

each iteration of a loop

Initialization: The loop invariant holds before the first iteration

Maintenance: If the loop invariant holds before an iteration, it holds after the iteration

Termination: When loop terminates, invariant gives useful property to show the algorithm is correct

Iterative: Usually loop invariants Recursive: Usually induction

Stable Marriage

Perfect Matching with Stability

Self-enforcing: Self-interest itself prevents offers from being retracted and redirected

Gale-Shapley Algorithm

Men get the best women while women get the worst men

Greedy algorithm that picks the best available women for each man

Time: $O(n^2)$

Complexity

```
a^{\log_b x} = x^{\log_b a}
\log_b x = \frac{\log_c x}{\log_c b}\log_b M \cdot N = \log_b M + \log_b N
\log_b \frac{M}{N} = \log_b M - \log_b N\log_b M^k = k \log_b M
Big Oh: f(n) is O(g(n)) if f(n) \leq cg(n) for n \geq n_0:
c, n_0 > 0
Big Omega: f(n) is \Omega(g(n)) if f(n) \geq cg(n) for n \geq n_0:
c, n_0 > 0
Big Theta: f(n) is \Theta(g(n)) if f(n) is O(g(n)) and f(n)
is \Omega(q(n))
Little Oh: Strict Big Oh
Little Omega: Strict Big Omega
\lim_{n\to\infty}\frac{f(n)}{g(n)}:
0 if f(n) is o(g(n)), \infty if f(n) is \omega(g(n))
<\infty if f(n) is O(g(n)), > 0 if f(n) is \Omega(g(n))
0 < \infty if f(n) is \Theta(g(n))
Growth Rates: 1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 <
n^c < 2^n < c^n < n! < n^n
Harmonic: \sum_{k=1}^{n} \frac{1}{k} \sim \ln n
```

Triangular: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \sim \frac{n^2}{2}$ Squares: $\sum_{k=1}^{n} k^2 \sim \frac{n^3}{3}$

Geometric: $\sum_{k=0}^{n} ar^k = \frac{a(r^{n+1}-1)}{r-1}$

Stirling's Approximation: $\log_2(n!) \sim n \log_2 n$ Master's Theorem: $T(n) = aT(\frac{n}{h}) + f(n), \epsilon > 0$

```
f(n) = O(n^{\log_b(a) - \epsilon}) \to T(n) = \Theta(n^{\log_b a})
f(n) = \Theta(n^{\log_b a}) \to T(n) = \Theta(n^{\log_b a} \log n)
f(n) = \Omega(n^{\log_b(a) + \epsilon}) \land af(\frac{n}{b}) \le cf(n) \text{ for some } c < 1 \to T(n) = \Theta(f(n))
```

Graphs

n =
$$|V|$$
, m = $|E|$, e = (u, v)
Complete: $|E| = \frac{n(n-1)}{2}$

Adjacency Matrix

Space: n^2 , Check if (u, v) is an edge: $\Theta(1)$, Check all edges: $\Theta(n^2)$

Adjacency List

```
Space: n + m, Check if (u, v) is an edge: O(\deg(u)) (actually O(1 + \deg(u))), Check all edges: \Theta(n + m) (actually \Theta(n + 2m))
```

Paths

Sequence of vertices v_1, v_2, \ldots, v_k such that (v_i, v_{i+1}) is an edge

Simple Path: No repeated vertices

Connected: There is a path between every pair of vertices

Cycle: Simple path with $v_1 = v_k$ and k > 2Tree: Connected graph with no cycles, |E| = n - 1

Breadth First Search

O(n+m) if adjacency list, $O(n^2)$ if adjacency matrix

Depth First Search

O(n+m) if adjacency list, $O(n^2)$ if adjacency matrix

Directed Graphs

Mutually Reachable: $u \to v$ and $v \to u$ Strongly Connected: Every node mutually reachable Directed Acyclic Graph: No directed cycles Topological Ordering: Ordering of vertices v_1, v_2, \dots, v_n such that if (v_i, v_j) is an edge, i < jDAG \rightleftarrows Topological Ordering

Bipartite Graphs

No odd cycles, can be colored with 2 colors

Connected Components

Strongly Connected Components (Di-graph only)

Greedy Algorithms

Use local optimization to find a global solution.

Interval Scheduling

Maximum number of non-overlapping intervals Sort by finish time, take the first interval that doesn't overlap with the previous interval

Time: $O(n \log n)$

Interval Partitioning/Scheduling all Intervals

Minimum number of rooms to schedule all intervals Sort by start time, if interval overlaps with all preceding intervals, allocate new room

Time: $O(n \log n)$

Scheduling to Minimize Lateness

Minimizes the maximum lateness Sort by deadline, schedule in order of increasing deadline Time: $O(n \log n)$

Dijkstra

Single-source shortest path; weighted BFS

Assumes non-negative edge weights, non-deterministic (consistent) otherwise

Use Bellman-Ford if negative edge weights, which uses dynamic programming

Time: O(nm) if naive, $O(m \log n)$ if priority heap

Minimum Spanning Trees

Assumes connected, undirected, distinct and non-negative edges

Kruskal

Sort edges by weight, add edge if it doesn't form a cycle Time: $O(m^2)$ if naive, $O(m \log n)$ if union-find

Prim

Start with any vertex, add the minimum weight edge that connects to the tree(one inside tree and one outside)

Time: $O(n^2)$ if naive, $O(m \log n)$ if priority heap

Reverse-Delete

Sort edges in decreasing order, remove edge if it doesn't disconnect the graph

Time: $O(e \log e)$ in PPT, $O(E \log(V)(\log \log V)^3)$ online

Greedy Stays Ahead

Define: Optimal- O and Greedy- G

Measurement: Define criteria to compare each element

of $\mathbb O$ and $\mathbb G$

Stays Ahead: Show \mathbb{G}_i is better than \mathbb{O}_i ($\forall i$), usually

through induction

Optimality: Show that \mathbb{G} is optimal since it is always better than \mathbb{O} , usually through contradiction by assuming greedy isn't optimal but it stays ahead of optimal

Exchange Argument

Define: Optimal- $\mathbb O$ and Greedy- $\mathbb G$

Compare Solutions: Show differences in order or exis-

tence

Exchange Pieces: Show that transforming $\mathbb O$ to $\mathbb G$

doesn't reduce optimality Iterate: Repeat until \mathbb{O} is \mathbb{G}

Divide and Conquer/Combine

Divide: Break problem into smaller subproblems Conquer/Combine: Solve subproblems recursively and combine

Recurrence Relation

A function defined in terms of itself

Mirrors the recursive algorithm it represents

Analysis of the running time of a divide and conquer algorithm generally involves solving a recurrence relation

1,2,3 Method Example: (Merge Sort)

$$\begin{array}{c|c|c} T(n) = 2T(\frac{n}{2}) + n - 1, \ T(1) = 0 \\ \hline \text{Level} & \text{Problem Size} & \text{Total Time} \\ \hline 0 & n & n \\ \hline 1 & \frac{n}{2} & 2\frac{n}{2} = n \\ 2 & \frac{n}{4} & 4\frac{n}{4} = n \\ \vdots & \vdots & \vdots \\ k & \frac{n}{2^k} = 1 & 2^k \frac{n}{2^k} = n \\ \hline \rightarrow (\sum_{i=0}^{k-1} n) + 0 \cdot 2^{\log_2 n} \rightarrow \sum_{i=0}^{\log_2 n-1} n \rightarrow n \log_2 n \\ \text{or } \rightarrow \sum_{i=0}^k n \rightarrow \sum_{i=0}^{\log_2 n} n \rightarrow n \log_2 n \end{array}$$

Dynamic Programming