

CS 170 Midterm 2 Cheat Sheet

Asymptotic Notation

$f(n) = O(g(n))$ There exists $c > 0$ such that for large enough n , $f(n) \leq c \cdot g(n)$.

$f(n) = \Omega(g(n))$ There exists $c > 0$ such that for large enough n , $f(n) \geq c \cdot g(n)$.

$f(n) = \Theta(g(n))$ $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Limit Rule

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) = O(g(n)) \\ c > 0 & \text{then } f(n) = \Theta(g(n)) \\ \infty & \text{then } f(n) = \Omega(g(n)). \end{cases}$$

Divide and Conquer

High Level Approach:

1. Break a problem into subproblems
2. Recursively solve these subproblems
3. Combine the results

Master Theorem: Useful if solving a size- n problem requires recursively solving a size- $\frac{n}{b}$ subproblems and $O(n^d)$ work to prepare/combine the results from the subproblems. In recurrence form:

If

$$T(n) = aT(n/b) + O(n^d) \text{ for } a > 0, b > 1, \text{ and } d \geq 0,$$

where $T(n)$ is the work to solve a problem of size n , then,

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

Mergesort

Main Idea: Break down the list into several sublists until each sublist consists of a single element. Merge the sublists to return the sorted list.

```
def mergesort(a[1...n]):
    if (n > 1):
        return merge(mergesort(a[1...n/2]),
                     mergesort(a[n/2+1...n]))
    else: return a

def merge(x[1...k], y[1...l]):
    if k=0: return y[1...l]; if l=0: return x[1...k]
    if x[1] <= y[1]:
        return x[1] + merge(x[2...k], y[1...l])
    else:
        return y[1] + merge(x[1...k], y[2...l])
```

Recurrence: $T(n) = 2T(n/2) + O(n)$. **Runtime:** $T(n) = O(n \log n)$.

Fast Fourier Transform

Goal: Given a degree d polynomial $P(x) = p_0 + p_1x + \dots + p_dx^d$ evaluate P on the n th roots of unity $1, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$, where $\omega_n = e^{\frac{2\pi i}{n}}$ (n is the smallest power of 2 greater than or equal to $d + 1$).

Main Idea: Let $E(x) = p_0 + p_2x + p_4x^2 + \dots$ and $O(x) = p_1 + p_3x + p_5x^2 + \dots$. We can represent $P(x) = E(x^2) + xO(x^2)$.

The problem reduces to evaluating two degree $n/2 - 1$ polynomials, E and O , on the $n/2$ roots of unity and combining the results.

Runtime: $O(n \log n)$.

Matrix View: We speed up this computation (where n is a power of 2):

$$\begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

$$\text{Inversion Formula: } M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

Strongly Connected Components (SCC)

To find an SCC, run DFS on $G^{Reverse}$ to get the post-order of the graph. Run DFS on G starting at the max post-order of $G^{Reverse}$ (sink of G). When there are no more vertices to explore = one SCC.

Runtime: $O(|V| + |E|)$.

Graph Traversals (DFS & BFS)

	Pseudocode	Explanation
DFS	<pre>def explore(G,v): visited(v) = true previsit(v) for each edge(v,u) in E: if not visited(u): explore(u) postvisit(v) def dfs(G): for all v in V: if not visited(v): explore(v)</pre>	<ul style="list-style-type: none"> - Runtime: $O(V + E)$ - Uses a stack - $pre(v)$ = the "time" when v was about to be explored. - $post(v)$ = the "time" when v has already been explored. - Topological sort: If G is a DAG and $(u, v) \in E$, then $post(u) > post(v)$.
BFS	<pre>def bfs(G, s): for all u in V : dist(u) = infinity dist(s) = 0 Q = [s] while Q is not empty: v = Q.eject() for each edge(v,u) in E: if dist(u) == infinity: Q.inject(u) dist(u) = dist(v) + 1</pre>	<ul style="list-style-type: none"> - Runtime: $O(V + E)$ - Uses a queue - Treats all edges as having the same length - All vertices not connected to start vertex are ignored

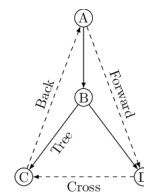
Types of Edges

Tree Edge: part of the DFS forest

Forward Edge: leads to a non-child descendant in the tree

Back Edge: leads to an ancestor in the tree

Cross Edge: leads to a node that's neither descendant nor ancestor in the tree



An edge $(u, v) \in E$ is a:

Tree/Forward Edge if

$$pre(u) < pre(v) < post(v) < post(u) \quad \begin{bmatrix} [] \\ uvvu \end{bmatrix}$$

Back Edge if

$$pre(v) < pre(u) < post(u) < post(v) \quad \begin{bmatrix} [] \\ vuuu \end{bmatrix}$$

Cross Edge if

$$pre(v) < post(v) < pre(u) < post(u) \quad \begin{bmatrix} [] \\ vuuu \end{bmatrix}$$

Shortest Paths

Dijkstra's Pseudocode	Explanation
<pre>def dijkstra(G,l,s): for all u in V: dist(u) = infinity prev(u) = nil dist(s) = 0 H = makequeue(V) while H is not empty: u = deletemin(H) for all edges (u,v) in E: if dist(v) > dist(u) + l(u,v): dist(v) = dist(u) + l(u,v) prev(v) = u decreasekey(H,v)</pre>	<ul style="list-style-type: none"> - Runtime: $O((V + E) \log(V))$ - Calculates shortest path from s to every other vertex - Assumes no negative edges
Bellman-Ford Pseudocode	Explanation
<pre>def bellman-ford(G,l,s): for all u in V : dist(u) = infinity prev(u) = nil dist(s) = 0 repeat V - 1 times: for all e in E: update(e) def update(e = (u, v)): d(v) = min(d(v), d(u) + l(u, v))</pre>	<ul style="list-style-type: none"> - Runtime: $O(V \cdot E)$ - Procedure updates all edges $V - 1$ times - Detect negative cycles with one extra round of BF, if dist value changes
DAG Shortest Path	Explanation
<pre>def dag-shortest path(G,l,s): Linearize G for each u in V in linear order: for all edges (u, v) in E: update(u, v)</pre>	<ul style="list-style-type: none"> - Runtime: $O(V + E)$ - Works for negative edges - Visits vertices in topological order

CS 170 Midterm 2 Cheat Sheet

Greedy Algorithms

High Level Approach: Greedy algorithms make the locally optimal choice at each step. Hence, greedy algorithms work for problems where making locally optimal choices yields a global optimum.

Minimum Spanning Trees (MST)

Goal: Given a weighted undirected graph $G = (V, E)$, find the lightest weight tree that connects all vertices V .

Cut Property: The lightest edge across a cut is in some MST.

Cut Property 2: Suppose edges X are part of some MST. Let $(S, V \setminus S)$ be any cut for which edges in X do not cross the cut, and let e be the lightest edge across the cut. Then edges $X \cup \{e\}$ are part of some MST.

Kruskal's Algorithm

Main Idea: Repeatedly add the next lightest edge that doesn't produce a cycle.

Runtime: $O(|E| \log(|V|))$.

Prim's Algorithm

Main Idea: On each iteration, pick the lightest edge between a vertex in the current subtree S and a vertex outside S .

Runtime: $O(|E| \log(|V|))$.

Huffman Coding

Goal: Given characters/frequencies $\{(i, f_i)\}$, encode each character in binary so the final encoding is maximally efficient.

Main Idea: Find the two symbols with the smallest frequencies, say i and j , and make them children of a new node (meta-node), which then has frequency $f_i + f_j$, and repeat until one node remaining.

Cost (average bits/character): $\sum_{i=1}^n f_i l_i$, where l_i is the number of bits to encode the i th character.

Set Cover (Greedy Approximation)

Goal: Find the minimum number of subsets that cover a set $U = \{1, 2, \dots, n\}$.

Main Idea: Pick the set S_i with the largest number of uncovered elements. Repeat until all vertices are covered.

Performance: Not optimal, but pretty good: $k_g \leq k_o \cdot \ln(n) + 1$, where k_g is number of subsets output by greedy and k_o is optimal.

Dynamic Programming

High Level Approach: Define subproblems s.t. the solution to a big problem can be easily derived from the solutions to its subproblems. Solve all subproblems from small to large, using results from previous subproblems to solve the current subproblem. Both recursion with memoization (top-down) and iteration (bottom-up) approaches exist.

Shortest Path in a DAG

Goal: Given a DAG, find the length of the shortest path from s to t .

Main Idea: For every $v \in V$, we define $dist(v)$ as the length of the shortest path from s to v . Order of subproblems: topological order. Base cases: $dist(s) = 0$, and $dist(v) = \infty$ if $v \neq s$ is a source. To calculate the shortest path to v , we look at edges (u, v) going into v , we use all the pre-computed $dist(u)$ + the length of the edge (u, v) and find the minimum:

$$dist(v) = \min_{(u,v) \in E} \{dist(u) + l(u, v)\}.$$

Knapsack

Goal: Given knapsack with max capacity W and n items of weight w_1, \dots, w_n and dollar value v_1, \dots, v_n , find the most valuable combination of unique items to fit in this knapsack

Main Idea: Define $f(i, u)$ as the max value achievable with a knapsack of a capacity u and items $1, \dots, i$. The optimal solution to $f(i, u)$ has two cases: to include item i , or to not include it in the knapsack. If we include it, we need to subtract its capacity. Thus, the recurrence is:

$$f(i, u) = \max(f(i-1, u), f(i-1, u-w_i) + v_i)$$

Runtime: Storing our $f(i, u)$ values in a 2-D array of $n+1$ rows and $W+1$ columns, the algorithm fills this array from left to right, with each entry taking $O(1)$ steps, giving total runtime of $O(nW)$.

Tips

1. Define an appropriate subproblem with relevant parameters, ensuring that the parameters fully determine the subproblem.
2. Given access to all previously solved subproblems, develop an appropriate relation to solve the current subproblem.
3. Test on smaller cases

Max Flow

Network Flow:

A flow f from s to t in a directed graph G whose edges have capacities $c_e \geq 0$ has the following properties:

- Flow on each edge f_e must satisfy: $0 \leq f_e \leq c_e$.

- Conservation of flow: flow entering vertex = flow leaving vertex

Goal: Find maximum flow $\text{val}(f^*)$ where $\text{val}(f) = \sum_{(s,i) \in E} f_{(s,i)}$.

Ford-Fulkerson Algorithm

Residual graph: A directed graph R whose edge weight is denoted as residual capacity $r(u, v)$:

- $r(u, v) = c(u, v) - f(u, v)$ if $(u, v) \in E$ and $f(u, v) < c(u, v)$
- $r(u, v) = c(u, v) + f(v, u)$ if $(v, u) \in E$ and $f(v, u) > 0$

where $c(u, v)$ is capacity of (u, v) on G ($c(u, v) = 0$ if edge doesn't exist), $f(u, v)$ is flow on (u, v) . e.g. if $c(u, v) = 3$, $c(v, u) = 4$ and $f(u, v) = 2$ then $r(u, v) = 1$ and $r(v, u) = 6$.

Algorithm:

1. Find a path from s to t and route max flow through this path
2. Update capacities by updating the residual graph
3. Repeat 1 and 2 until algorithm halts (no path from s to t is found)

Explanation:

- Intuition: The back edges in residual graph can undo suboptimal flows in some edges for future iterations.

- Runtime: $O(|V| \cdot |E|^2)$ if using BFS to find paths.

Max-Flow Min-Cut Theorem

An (s, t) -cut partitions V into two subsets S, T , where $s \in S, t \in T$.

Capacity of (s, t) -cut C : $\text{val}(C) = \sum_{e \in \{(u,v) | u \in S, v \in T\}} c_e$.

Theorem: $\text{val}(f^*) = \text{val}(C^*)$. Maximum flow is equal to minimum cut. In the residual graph, the set of vertices, X , reachable from s , yields the cut $(X, V \setminus X)$ whose capacity equals the max flow.

Linear Programming

High Level Approach: Define your problem as an objective function with a set of linear constraints while following the given convention, and the Simplex algorithm does the rest. If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide.

Generic Linear Program:

Primal:

$$\max \mathbf{c}^T \mathbf{x}$$

$$\mathbf{Ax} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0$$

Dual:

$$\min \mathbf{y}^T \mathbf{b}$$

$$\mathbf{y}^T \mathbf{A} \geq \mathbf{c}^T$$

$$\mathbf{y} \geq 0$$

Constraint Transformations:

Changing Objective	Inequality to Equality
$\max \mathbf{c}^T \mathbf{x} = \min -\mathbf{c}^T \mathbf{x}$	$ax \leq b \rightarrow ax + s = b,$
$\min \mathbf{c}^T \mathbf{x} = \max -\mathbf{c}^T \mathbf{x}$	$s \geq 0$
Equality to Inequality	Unrestricted Variable
$ax = b \rightarrow ax \leq b,$	$x \in \mathbb{R} \rightarrow x = x_+ - x_-$
$ax \geq b$	$x_+, x_- \geq 0$

Zero-Sum Games

Setup: Given a 2D matrix where the rows are player A's moves, the columns are player B's moves, and the values are A's reward for each move. Whichever player goes first has to announce their strategy first.

Strategy is the vector probabilities of playing specific moves

Main Idea: If player A goes first, player B will pick the move that minimizes A's reward. So A's best strategy will be the maximum of the minimum of B's moves. If player B goes first, player A will pick the move that maximizes A's reward. So B's best move will be the minimum of A's moves. These two equations come out to be the dual of each other.

Example:

	B1	B2
A1	a	c
A2	b	d

A's strategy is $[x_1, x_2]$

B's strategy is $[y_1, y_2]$

A picks $[x_1, x_2]$ to maximize the value $\min\{ax_1 + bx_2, cx_1 + dx_2\}$

B picks $[y_1, y_2]$ to minimize the value $\max\{ay_1 + cy_2, by_1 + dy_2\}$

Min-Max Theorem: $\max_x \min_y \sum_{i,j} G_{ij} x_i y_j = \min_y \max_x \sum_{i,j} G_{ij} x_i y_j$

Simplex

Main Idea: The optimal point is at an intersection of the geometrical representation of the constraints. If the chosen vertex is not optimal, then the objective function can be improved by following an outgoing edge from this vertex.