CS3230 Cheatsheet by MA

Asymptotic Analysis

1. O**-notation** (upper bound)

$O(g(n)) = \{f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0\}$

2. Θ**-notation** (tight bound)

$\Theta(g(n)) = \{f(n)$: there exist constants $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$

3. Ω**-notation** (lower bound)

$\Omega(g(n)) = \{f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0\}$

4. o**-notation** (tight upper bound)

$o(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0\}$

5. ω**-notation** (tight lower bound)

$\omega(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0\}$

Properties of Math

Stirling's approximation

$$\log(n!) = \theta(n \lg n)$$

Harmonic Series

$$\begin{aligned} H_n =& 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \\ =& \ln n + O(1) \end{aligned}$$

Limit

Assume $f(n), g(n) > 0$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \to f(n) = o(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \to f(n) = O(g(n))$$

$$0 < \lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \to f(n) = \Theta(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \to f(n) = \Omega(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \to f(n) = \omega(g(n))$$

L'Hopital's Rule

If we have an indeterminate form $\frac{0}{0}$ or $\frac{\infty}{\infty}$, then

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{x\to\infty}\left(\frac{f'(n)}{g'(n)}\right)$$

Correctness of Algorithms

Iterative Algorithms

A **loop invariant** is:

- true at the beginning of an iteration, and
- remains true at the beginning of the next iteration
- if true at the end, then it implies algorithm's correctness

To use invariant to show the correctness of an iterative algorithm, we need to show three things:

- **Initialization:** The invariant is true before the first iteration of the loop.
- **Maintenance:** If the invariant is true before an iteration, it remains true before the next iteration.
- **Termination:** When the algorithm terminates, the invariant provides a useful property for showing correctness.

Recursive Algorithms

To show the correctness of a recursive algorithm:

- Use strong induction
- Prove base cases
- Show algorithm works correctly assuming algorithm works correctly for smaller cases.

Solve Recurrences

Recursion tree

Draw out the recurrence in the form of a tree

Master method

Master theorem applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive

When comparing $f(n)$ and $n^{\log_b a}$ There are three cases to master theorem.

Define a, b, $f(n)$ and $n^{log_b a}$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   $T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$.
   $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ **and** $f(n)$ satisfies the **regularity condition** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.
   $T(n) = \Theta(f(n))$

Substitution method

Guess the time complexity and verify that it is correct by induction

Randomized Algorithms

An algorithm is called **randomized** if its output and running time are determined by its input and also by values produced by a random-number generator.

Random Variables and Expectation

A **discrete random variable** is a function from a sample space to the integers.
Expectation: $E[X] = \sum_i i \cdot Pr[X = i]$
Linearity of expectations: $E[X + Y] = E[X] + E[Y]$ for any two random variables X and Y.

Hashing

Universal Hashing

Suppose H is a set of hash functions mapping U to $[M]$. We say H is **universal** if for all $x \neq y$:

$$\frac{|h \in H : h(x) = h(y)|}{|H|} \leq \frac{1}{M}$$

For any $x \neq y$, if h is chosen uniformly at random from a universal H, there's at most $\frac{1}{M}$ probability that $h(x) = h(y)$.

Collision Analysis

Suppose H is a *universal* family of hash functions mapping U to $[M]$. For any N elements, $x_1, ..., x_N$, the expected number of collisions between x_N and the other elements is $< \frac{N}{M}$.

Expected Cost

Suppose H is a *universal* family of hash functions mapping U to $[M]$. For any sequence of N insertions, deletions and queries, if $M \geq N$, then the expected total cost for a random $h \in H$ is $O(n)$.

Construction of universal family

Suppose U is indexed by u-bit string, and $M = 2^m$. For any binary matrix A with m rows and u columns:

$$h_A(x) = Ax \pmod 2$$

Claim: $\{h_A : A \in \{0,1\}^{m \times u}\}$ is universal.

H can be used for dictionaries. In addition to storing the hash table, matrix A also needs to be stored. Additional storage overhead $\theta(\log N \cdot \log U)$ bits, if $M = \theta(N)$.

Perfect Hashing

Quadratic Space

If H is *universal* and $M = N^2$, then if h is sampled uniformly from H, the expected number of collisions is < 1. Therefore, there is a hash function $h : U \to [N^2]$ from H for which there are no collisions.

2-Level Scheme

If H is *universal*, then if h is sampled uniformly from H:

$$\mathbb{E}\left[\sum_k L_k^2\right] < 2N$$

Amortized Analysis

An amortized analysis *guarantees* the average performance of each operation in the worst case.

Types of Amortized Analysis

Aggregate method

We count the complexity of each operation and determine a pattern and come up with an overall bound for the time complexity.

Accounting method

Charge i-th operation a fictitious **amortized cost** $c(i)$. This amortized cost $c(i)$ is a fixed cost for each operation, while the true cost $t(i)$ varies depending on what operation is called. Amortized cost $c(i)$ must satisfy, for all n:

$$\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$$

The total amortized costs provide an upper bound on the total true costs. Different operations can have different amortized costs.

Potential method

ϕ: Potential function associated with the algorithm/DS
$\phi(i)$: Potential at the end of the ith operation
Important conditions to be fulfilled by ϕ

- $\phi(i) \geq 0$ for all i

Amortized cost of i-th operation
= Actual cost of ith operation + ($\Delta\phi(i)$)
= Cost of ith operation + ($\phi(i) - \phi(i-1)$)

Amortized cost of n operations
=Σ_iAmortized cost of ith operation
=Actual cost of n operations + $(\phi(n) - \phi(0))$
$\geq$ Actual cost of n operations $-\phi(0)$

Select a suitable potential function ϕ, so that for the costly operation, $\Delta\phi_i$ is negative such that is nullifies or reduces the effect of the actual cost.
Try to find a quantity that is decreasing in the expensive operation

Dynamic Programming

Optimal Substructure: An optimal solution to a problem (instance) contains optimal solutions to subproblems.
Overlapping Subproblems: A recursive solution contains a "small" number of distinct subproblems repeated many times

Greedy Algorithms

At each step, cast the problem such that we have to make a choice and are left with one subproblem to solve. Prove that there is always an optimal solution to the original problem that *makes the greedy choice*. Use optimal substructure to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem.

Linear Programming

Standard form of an LP:
Objective is to maximise:

$$\sum_{j \in [n]} c_j \cdot x_j$$

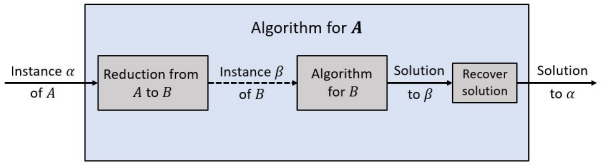
Constraints:

$$x_1, ..., x_n \geq 0$$
$$a_{11}x_1 + ... + a_{1n}x_n \leq b_1$$
$$a_{21}x_1 + ... + a_{2n}x_n \leq b_2$$
$$...$$
$$a_{m1}x_1 + ... + a_{mn}x_n \leq b_m$$

Optimal feasible solution (if it exists) is always at a vertex
Note: an integer program is an LP with the additional constraint that each $x_i \in \{0, 1\}$

Reductions

We say that a problem A can be reduced to an instance of problem B if:



Additionally, A has a $p(n)$-time reduction to B, $A \leq_P B$ when:
For an instance α of A *of size* n,
1. α can be converted to an instance β of B **in** $p(n)$ **time**
2. the solution of α can be recovered **in** $p(n)$ **time**

Encoding

n is the *length of the encoding of the problem instance*, such as the number of bits required to write down the instance

Pseudo-polynomial algorithms

An algorithm that runs in time polynomial in the *numeric value* of the input but is exponential in the *length of the representation* of the input is called a **pseudo-polynomial** time algorithm.

Reductions between Decision Problems

Two decision problem A and B have **a polynomial time reduction** from A to B, denoted $A \leq_p B$, which is a transformation from instances α of A to instances β of B if:

- Reduction runs in polynomial time
- If α is a YES-instance of A, β is a YES-instance of B
- If β is a YES-instance of B, α is a YES-instance of A

Example: 3-SAT $\leq_P$ INDEPENDENT-SET
Given an instance ϕ of 3-SAT, goal is to construct an instance (G, k) of INDEPENDENT-SET so that G has an independent set of size k iff ϕ is satisfiable.

Reduction
G contains 3 vertices for each clause - one for each literal in the clause. Connect the 3 literals in each clause in a triangle. Connect each literal to each of its negations. Set k = number of clauses.
YES-instance to YES-instance
If ϕ is satisfiable, take any satisfying assignment and, for each clause, some literal that is set to be true by this assignment. The vertices corresponding to these literals form an independent set of size k.
NO-instance to NO-instance
Suppose (G, k) is a YES-instance, and has independent set S of size k. Each of the k triangles must contain exactly one vertex from S. Set these literals to true, and all clauses will be satisfied, so ϕ is satisfiable.

Complexity Classes

NP-complete	Problems computationally equivalent to 3-SAT and are **NP** and **NP**-Hard
NP	Problems that poly-time reduce to 3-SAT. Polynomial time verifiable **certificates** of YES-instances exist. Any problem in P is in NP.
P	Problems solvable in (deterministic) poly time
co-NP	A problem is in **co-NP** if polynomial time verifiable certificates of NO-instances exist. The complement of any **NP** problem is in **co-NP**.
NP-Hard	Problem A is **NP-Hard** if for any problem B in **NP**, $B \leq_P A$.

Example: SUBSET-SUM
Certificate is the subset S' itself. Verifier checks whether the sum of elements of S' is t in polynomial time. Hence SUBSET-SUM is in NP.

Approximation

Let C^* be the cost of the optimal solution.
Let C be the cost of the solution found by your algorithm.

$$\text{Approximation ratio } = \begin{cases} \frac{C}{C^*} > 1 & \text{for min problems} \\ \frac{C^*}{C} \leq 1 & \text{for max problems} \end{cases}$$

A good algorithm will have an approximation ratio as close to 1 as possible, ideally a constant

Approximation Schemes

A *Polynomial-Time Approximation Scheme* (PTAS) for a problem is an algorithm that given an instance and an $\epsilon > 0$ runs in time $poly(n)f(\epsilon)$ for some function f and has approximation ratio $(1 + \epsilon)$
A *Fully Polynomial-Time Approximation Scheme* (FPTAS) for a problem is an algorithm that given an instance and an $\epsilon > 0$ runs in time $poly(n, 1/\epsilon)$ and has approximation ratio $(1 + \epsilon)$