Logarithms(Useful rules)

1. $a = b^{\log_b a}$
2. $\log ab = \log a + \log b$ In particular, $\log n! = \sum_{i=1}^{n} \log i$
3. $\log_c(ab) = \log_c a + \log_c b$
4. $\log_b a^n = n \log_b a$
5. $\log_b a = \frac{\log_c a}{\log_c b}$
6. $\log_b\left(\frac{1}{a}\right) = -\log_b a$
7. $\log_b a = \frac{\log_a a}{\log_a b} = \frac{1}{\log_a b}$
8. $a^{\log_b c} = c^{\log_b a}$
9. $\log_a f(x) = \frac{1}{\ln(a) * f(x)} f'(x)$ examples: $\frac{d}{dx}\ln x = \frac{1}{x}$
10. $\frac{d}{dx} e^{f(x)} = f'(x) e^{f(x)}$

Stirling's approximation:

- $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right) \approx O(n^n)$
- $\log(n!) = \theta(n \lg n)$

Arithmetic series:

- $\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \theta(n^2)$
- $\sum_{k=1}^{n} k^2 = 1 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} = \theta(n^3)$

Harmonic series

- (Note: we can write it backwards, it's the same thing: $\frac{1}{n} + \frac{1}{n-1} + \cdots + 1$)
- $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$

Geometric Series

- $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1}-1}{x-1}$
- $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ *when* $|x| < 1$

Limits: Assume f(n), g(n) > 0

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \to f(n) = o(g(n))$	$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) \neq \infty \to f(n) = O(g(n))$
$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) \neq 0 \to f(n) = \Omega(g(n))$	$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \to f(n) = \omega(g(n))$
$0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \to f(n) = \theta(g(n))$	

Definition and Properties of big-O

- $O(g(n)) = \{f(n): \exists c, n_0 > 0 \; st \; 0 \le f(n) \le cg(n) \; for \; all \; n \ge n_0$
- $\Omega(g(n)) = \{f(n): \exists c, n_0 > 0 \; st \; 0 \le c.g(n) \le f(n) \; for \; all \; n \ge n_0\}$
- $\theta(g(n)) = \{f(n): \exists c_1, c_2, n_0 > 0 \; st \; \forall n \ge n_0 \; (0 \le c_1 g(n) \le f(n) \le c_2 g(n))\} = O(g(n)) \cap \Omega(g(n))$
- $o(g(n)) = \{f(n): \forall c > 0, \exists n_0 > 0 \; st \; 0 \le f(n) < cg(n) \; for \; all \; n \ge n_0\}$
- $\omega(g(n)) = \{f(n): \forall c > 0, \exists n_0 > 0 \; st \; 0 \le cg(n) < f(n) \; for \; all \; n \ge n_0\}$

Transitivity:

- $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \to f(n) = \Theta(h(n))$
- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \to f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \to f(n) = \Omega(h(n))$
- $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \to f(n) = o(h(n))$
- $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \to f(n) = \omega(h(n))$

Reflexivity:

- $f(n) = \Theta(f(n)) = O(f(n)) = \Omega(f(n))$

Symmetry:

- $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Complementarity:

- $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
- $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

Common useful facts

- Degree-k polynomials are $O(n^k)$,$o(n^{k+1})$ and $\omega(n^{k-1})$
- Polynomials dominate logarithms: $(\log n)^{100} = o(n^{0.001})$
- Exponentials dominate polynomials: $n^{1000} = o(2^{0.001n})$

Master Theorem

- Given $T(n) = aT\left(\frac{n}{b}\right) + \Theta(f(n))$, the branching factor is $\log_b a$

Case 1 Branching factor dominates work done per recursion level by some $\epsilon > 0$.	$f(n) = O(n^{\log_b a - \epsilon})$ $T(n) = \Theta(n^{\log_b a})$ If $\epsilon = 0$, then **case 2**
Case 2 Branching factor is some log-factor of work done per branch.	$f(n) = \Theta(n^{\log_b a} \log^k n)$ $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
Case 3 Work done per branch dominates the branching factor at each level of recursion. Regularity condition must hold, where the work done in the next recursion must be bounded by the work done in current recursion.	$f(n) = \Omega(n^{\log_b a + \epsilon})$ $af\left(\frac{n}{b}\right) \le cf$, where $c < 1$ $T(n) = \Theta(f(n))$

Hashing

A family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ to $[M]$ is said to be universal if for any two distinct elements x, y $\in \mathcal{U}$, it is the case that:

$$\Pr_{h\sim\mathcal{H}}[h(x) = h(y)] \le \frac{1}{M}$$

A family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ to $[M]$ is said to be pairwise independent if for any two distinct elements x, y $\in \mathcal{U}$, and for any two hash values i_1, i_2:

$$\Pr_{h\sim\mathcal{H}}[h(x) = i_1, h(y) = i_2] = \frac{1}{M^2}$$

Loop Invariants

A loop invariant is:

- True at the beginning of an iteration
- Remains true at the beginning of the next iteration.

To prove the correctness of an iterative algorithm, we need to show 3 things:

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

Amortized Analysis

Accounting Method

- Assign each operation a cost
- Invariant to maintain: Bank balance never drops below 0 → actual costs is upper bounded by amortized cost → We have provided an upper bound for actual runtime
- Suppose actual cost of the i-th is denoted by c_i, amortized cost of i-th operation denoted by $\hat{c}_i$, then the following has to be maintained: $\sum_{i=1}^{n} \hat{c}_i \ge \sum_{i=1}^{n} c_i$

Potential Method

- Instead of credit in a bank, we view it as "potential" of the data-structure.
- A potential function ϕ maps each data structure D_i to a real number, $\phi(D_i)$, a measure of its potential.
- Amortized cost of the i-th operation $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$; Note that $\Delta\phi$ can be negative or positive.
- Total amortized cost is hence $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$
- If we can **show that $\phi(D_n) - \phi(D_0) \ge 0$ for any n**, then we have **successfully upper-bounded total actual costs**, $\sum_{i=1}^{n} c_i$, with total amortized cost $\sum_{i=1}^{n} \hat{c}_i$. If we can define $\phi(D_0) = 0$, then all we need to prove is that $\phi(D_n) \ge 0$ for all n
- To get amortized cost for each type of operation, we can use:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Dynamic Programming

Optimal substructure

A problem has optimal substructure when an optimal solution to the problem can be expressed in terms of optimal solutions of smaller subproblems.

Overlapping Subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times

Longest Common Subsequence

- LCS(n,m) = $\begin{cases} empty\ string & if\ n = 0\ or\ m = 0 \\ LCS(n-1, m-1) :: a_n & if\ a_n = b_m \\ bigger\ of\ \{LCS(n, m-1), LCS(n-1, m) & if\ a_n \neq b_m \end{cases}$

0-1 Knapsack O(nW)

- Let $m[i, j]$ be the maximum value that can be obtained using a subset of items in $\{1, 2, \ldots, i\}$ with a total weight capacity of j
- $m[i,j] = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ \max\{m[i-1, j-w_i] + v_i, \quad m[i-1, j]\} & if\ w_i \le j \\ m[i-1, j] & otherwise \end{cases}$

Cut and Paste Argument

Suppose you came up with an optimal solution to a problem by using suboptimal solutions to subproblems. Then, if you were to replace ("cut") those suboptimal subproblem solutions with optimal subproblem solutions (by "pasting" them in), you would improve your optimal solution. But, since your solution was optimal by assumption, you have a contradiction. Thus, we must have used optimal solutions to subproblems.

Greedy Algorithms

1. Cast the problem such that we have to make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe
3. 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.

Incremental Algorithms & Max-flow, Min Cut

Any cut is a bottleneck for any flow f. Thus max flow, $|f| \le$ capacity of min-cut.

Capacity of a cut

$C(S, T)$ = sum of all capacities from the set S to T

Ford-Fulkerson Algorithm

If a path exists from source **s** to sink **t**, push a flow through that path, update capacities. Repeat until there are no more augmenting paths. The total sum of flows pushed is equal to max-flow. The cut formed by the vertices reachable from **s** and those not reachable from **s** is also a min-cut.

Linear Programming

Standard form:

Maximize objective function (as opposed to minimize)	$Maximize\ f(x_1, \ldots, x_n)\ such\ that,$ $x_1, \ldots, x_n > 0$ $a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le b_1$ $a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \le b_2$ $\ldots$ $a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \le b_m$
All decision variables are required to be non-negative	
All constraints are less-than-or-equal constraints.	

Example

Minimize $x_1 + 4x_2$	Maximize $-(x_1 + 4x_2)$
$x_1 \geq 0$ (x_2 not required to be non-negative)	$x_1, x_2', x_2'' \geq 0$ (Replace all instances of x_2 with $(x_2' - x_2'')$, This is valid since $(x_2' - x_2'')$ can take the same range of values as x_2)
$x_1 - 2x_2 \geq 10$ $3x_1 + 5x_2 = 20$	$-(x_1 - 2x_2) \leq -10$ $3x_1 + 5x_2 \leq 20$ $-(3x_1 + 5x_2) \leq -20$

Reductions

Definition of reducibility

- $A \leq_P B$ if there is a p(n)-time reduction from A to B for some polynomial function p(n)
- $A \leq_p B \rightarrow$ If B has a polynomial time algorithm, so does A → If B is easy, so is A → If A is hard, then B is hard (intuitively, reducing A into B means that A is a special case of B. So if a special case is hard, then the generalized version is just as hard)
- A decision problem is a special case of the optimization problem (decision reduces to optimization), so it suffices to study decision problems for hardness

Reductions between Decision Problems

Given two decision problems A and B, a poly-time reduction from A to B denoted $A \leq_p B$, is a transformation from instances α of A to instances β of B such that:

1. α is a YES-instance for A **iff** β is a YES-instance for B
2. The transformation takes polynomial time in the size of α

Pseudo-polynomial

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

Intractability and NP-hardness

Circuit Satisfiability $\leq_p$ CNF-SAT $\leq_p$ 3-SAT

P – Class of problems solvable in deterministic polynomial time

NP (Non-deterministic Polynomial) – Class of problems for which polynomial time verifiable certificates of YES-instances exist

Co-NP – Class of problems for which polynomial time verifiable certificates of NO-instances exist

NP-Hard – A problem X is said to be in NP-hard if for any problem B in NP: $B \leq_p A$

NP-Complete – A problem is said to be in NP complete if it is in NP and NP-hard (i.e. poly-time verifiable certificate exists, and proven to be as hard as any problem in NP)

Cook-Levin Theorem

Any problem in NP poly-time reduces to 3-SAT. Hence, 3-SAT is NP-hard and NP-complete

Proving NP-completeness

To show that a problem A is NP-complete, we need to show it is in NP and NP-hard. To prove the latter, it suffices to reduce a known NP-complete problem B into A, e.g. $3SAT \leq_p A$

- Hence, we need to provide the poly-time reduction, and that a yes-instance for A is a yes-instance for B and vice versa.
- We also need to provide the poly-time certificate to prove NP.

Independent Set

Given a graph G=(V,E) and integer k, is there a subset of $\geq k$ vertices such that no two are adjacent.

Vertex Cover

Given a graph G=(V,E) and integer k, is there a subset of $\leq k$ vertices such that each edge is incident at least one vertex in the subset?

Set Cover

Given integers k and n, and a collection $\mathcal{S}$ of subsets of $\{1, \dots, n\}$, are there $\leq k$ of these subsets whose union equals $\{1, \dots, n\}$

Int Program

Given a set of m linear constraints in n variables, is there an assignment of values of {0,1} to the $x_i's$ such that all the constraints are satisfied.

Satisfiability (SAT)

- **Literal**: A Boolean variable or its negation – $x_i, \overline{x_i}$
- **Clause**: A disjunction of literals - $C_j = (x_1 \vee x_2 \vee x_3)$
- **Conjunctive Normal Form (CNF)**: A formula ϕ that is a conjunction of clauses
- **Satisfying Assignment**: An assignment of T/F values to each variable that makes ϕ evaluates to True
- **SAT:** Given a CNF formula ϕ over n variables, does it have a satisfying assignment?
- **3SAT:** SAT where each clause in the given formula contains exactly 3 literals corresponding to different variables.

Approximation Algorithms

Approximation Ratio – always larger than 1

- $Minimisation: \frac{C}{C^*} \qquad Maximization: \frac{C^*}{C}$

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) and $f(\epsilon)$ 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) and $poly\left(\frac{1}{\epsilon}\right)$ and has approximation ratio $(1 + \epsilon)$