UCSD CSE 21 Review

Taanish Reja and Kevin Jacob

Spring 2024

Table of Contents

2. Distributions and Probability
    - Probability Basics
    - Uniform/Non-Uniform distributions
    - Binomial Distribution
    - Conditional Probability
    - Bayes Theorem
    - Independence
    - Random Sampling
    - Random variables
    - Expectation
        - Linearity of Expectation
    - Variance
3. Runtime Analysis
    - Min Sort
    - Bubble Sort
    - Insertion Sort
    - Linear Search
    - Binary Search
    - Asymptotic Classes
        - Big Theta
        - Big Omega
        - Little O
        - Big O
        - Big O Class Properties
        - Growth Rate of Common Functions
        - Disjoint Lists Function
    - Product Rule
    - Loop Invariant Induction
        - Selection Sort Loop Invariant Induction
        - Find Max Proof

4. Recursion

- Recursive Find Max
- Merge Sort (Divide and Conquer)
- Master theorem
- Homogenous recurrence relations
  - Domino Tilings
  - Characteristic polynomial
  - Fibonacci
- Recursive Counting
  - Permutations/Stirling's Approximation
  - n-bit strings
  - Solving Recurrences
- Encoding
  - Lossy and loseless encoding
  - Fixed Width Encoding
  - Huffman Encoding
  - Fibonacci Encoding
  - Ranking/Unranking
  - Theoretical Best Encoding
- Graph Theory
  - Graphs
    - Directed graphs
    - Directed Acyclic Graphs
    - Undirected graphs
    - Connectedness
  - Hamilitonian paths
  - Eulerian paths
  - Adjacency Matrix
  - Trees
- Randomized Algorithms
  - Las Vegas Algorithms
  - Monte Carlo Algorithms

0.1 Sets

0.1.1 Multi-set

A set that allows for repeats (1,1,2,3)

Anagram: A string that is a rearrangement of a multi-set of characters

Example: How many anagrams of AEESSSSR

We need to consider over counting since swapping repeated letters will be the same, so the number we over count by is $2! \times 4!$. Therefore the total number of anagrams is $\frac{8!}{2! \times 4!}$.

0.1.2 Subset

A set A is a subset of another set B if B contains all the elements in A.

Example: finding the number of subsets of k elements from set of n elements

To find the number of subsets of k elements from set of n elements, we can do $\binom{n}{k}$

1 Counting

1.1 Product Rule

For any set, $|A \times B| = |A||B|$

$A \times B = \{(a, b) : a \in A, b \in B\}$

1.2 Sum Rule

For disjoint sets, $A \cap B = \emptyset$, $A \cup B = A + B$

1.3 Power Rule

For any set, $|A \times A \times A \times \cdots \times A| = |A|^n$

Examples:

- # of binary strings of length n $= 2^n$
- # of strings of length n over alphabet $A = |A|^n$
- # of n-length sequences consisting of elements from sets of the same cardinality $= |A|^n$
- # of ways to distribute $|A|$ distinct objects among n people $= n^{|A|}$

1.4 Quotient Rule

If set A can be partitioned into disjoint sets

$X_1, X_2, \ldots, X_k$, where $|X_1| = |X_2| = \cdots = |X_k|$

$k = \frac{|A|}{|X_1|}$

Example with object symmetries:

- Color a square with 4 colors
- 4! different ways to order/construct quare
- 8 object symmetries (4 rotations and 2 reflections)

# of theoretical objects = 4! = 24

# of physical objects = $\frac{24}{8} = 3$

1.5 Inclusion Exclusion Principle

For two sets: $|A \cup B| = |A| + |B| - |A \cap B|$

If $A_1, A_2, \ldots, A_n$ are finite sets then

Generalized:

$|A_1 \cup A_2 \cup \cdots \cup A_n| =$

$\sum_{1 \le i \le n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n+1} |A_1 \cap A_2 \cap \cdots \cap A_n|$

Example with fixed points:

Take the string 1234. We can set a fixed point, say, at position 1. This means the 1 will be fixed, and that we can form anagrams by shifting the other positions.

We want to figure out how many permutations of length 4 have a fixed point at position 1 or 2 or 3 or 4. To do this, we can use inclusion-exclusion. Let P_i represent the number of permutations at each fixed point i.

$|P_1 \cup P_2 \cup P_3 \cup P_4| = \sum(P_i) - \sum(P_i \cap P_j) + \sum(P_i \cap P_j \cap P_k) - (P_1 \cap P_2 \cap P_3 \cap P_4)$

Here, we use permutations because if one spot is fixed, there is 3 spots we can permutate, etc... $|P_1 \cup P_2 \cup P_3 \cup P_4| = \sum(3!) - \sum(2!) + \sum(1!) - (0!)$

Here, we multiply by $\binom{4}{x}$ because if x positions are fixed, $\binom{4}{x}$ gives us the # of ways to arrange those fixed positions.

$|P_1 \cup P_2 \cup P_3 \cup P_4| = \binom{4}{1}(3!) - \binom{4}{2}(2!) + \binom{4}{3}(1!) - \binom{4}{4}(0!)$

Fixed points can be generalized with the formula:

$n! \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k!}$

Derangements

Using our newly defined fixed points formula, as well as counting with complements, we can count derangements, or the number of permutations of the string 1,2,. . . ,n without a fixed point.

$n! - n! \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k!} = \sum_{k=1}^{n} \frac{(-1)^k}{k!}$

1.6 Counting With Complement

Universal Set (U): Set that contains all elements

Set (A): Subset of the universal set

Set Complement (A^c): Set that contains all elements in universal set that aren't in A

$U = A + A^c$ or $A = U - A^c$

Example: How many 4 digit strings of digits 0-9 have at least one 0?

U = set of all 4 digits strings where $|U| = 10^4$

A^c = set of all strings that don't have 0 where $|A^c| = 9^4$

$10^4 - 9^4$ is the number of 4 digit strings that have at least one 0

1.7 Permutations

r-permutations: # of ways to arrange r objects out of n objects. (Order matters here, whereas in combinations, order doesn't matter)

$P(n, r) = {}_nP_r = n(n-1)(n-2)\dots(n-r+1) = \frac{n!}{(n-r)!}$

n-permutations: rearrangement of n distinct objects so that each object appears exactly once

$n!$

Example with r words over an alphabet of length n

How many 3 letter words can you form from a 10 letter alphabet?

$P(10, 3) = \frac{10!}{(10-7)!}$

Example with different ways athletes can finish in a race

How many ways can 10 athletes finish in a race?

$n! = 10!$

1.8 Combinations

of ways to chose r different elements from a set of n distinct elements $C(n, r) = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!}$

Example with $\binom{n}{k}$:

- How many k-element subsets of a set of cardinality n
- How many length n binary strings with exactly k ones (density = # of 1 bits = k)

1.9 Binomial Coefficients

Binomial coefficient: $\binom{n}{k}$

The number of ways to choose k objects out of a set of k. Order doesn't matter, but since it counts sets,

two sets cannot have the same elements.

1.9.1 Symmetry Identity

Symmetry Identity: $\binom{n}{k} = \binom{n}{n-k}$

1.9.2 Binomial Theorem

$(x+y)^n = (x+y)(x+y)\dots(x+y) = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n-1}xy^{n-1} + \binom{n}{n}y^n$
Number of ways we can choose k of n factors to contribute to y and $n-k$ factors to contribute to x: $\binom{n}{kz^{n-k}y^k}$

1.9.3 Pascal's Identity

Pascal's Identity $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$
Example: Combinatorial Proof
Consider Pascal's Identity: $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$. The left hand side counts the number of strings of length $n+1$ with k 1's. Meanwhile, the right hand side counts the number of length n strings with k 1's plus the number of length n strings with $k-1$ 1's. The number of length n strings with k 1's is the same as the number of length $n+1$ strings with a 0 in the first term and k 1's in the last n terms, and the number of length n strings with $k-1$ 1's counts the number of length $n+1$ strings with a 1 in the first term and $k-1$ 1's in the last n terms. Thus, the sum of these terms counts the number of length $n+1$ strings with k 1's. Therefore, the LHS and RHS are the same.

1.9.4 Sum Identity

$\sum_{k=0}^{n}\binom{n}{k} = 2^n$

1.10 Stars and Bars

of ways to put n indistinguishable objects in k groups in $\binom{n+k-1}{k-1}$
Example: Partitioning n indistinguishable knights into k castles
Here, we can think of the castles as barriers. Thus, the problem becomes the standard stars and bars problem, and can be solved with $\binom{n+k-1}{k-1}$
Example: Integer Equations

$$a_1 + a_2 + a_3 + a_4 + a_5 = 33$$

How many solutions of positive integers: $\binom{37}{4}$ since we want to split 33 among 5 variables
Find the number of solutions where at least one variable is less than or equal to 3: $\binom{37}{4} - \binom{12}{4}$ since we can use counting with complement to subtract the solutions where every number is at least 5 from the total number of solutions.

How many solutions of integers where $0 \leq a_1 \leq 10$, $0 \leq a_2 \leq 10$, $0 \leq a_3 \leq 10$, $0 \leq a_4 \leq 10$, and $0 \leq a_5 \leq 10$: $\binom{37}{4} - (\binom{5}{1}\binom{26}{4} - \binom{5}{2}\binom{15}{4} + \binom{5}{3}\binom{4}{4})$ since we consider the cases where we give one, two, or three variable at least 11 in order to break the rule. For the term $\binom{5}{1}\binom{26}{24}$ we account for how many different sets of 1 variable we can select to add11 to and how many ways to distribute the remaining 22 among the variables. We have to consider the inclusion exclusion principle since we have to consider all the cases when one, two, or three variables have a minimum of 11. We subtract the total number of ways that don't satisfy the inequalities from the total number of solutions in order to get the answer.

1.11 The 12-fold way

Elements of N	Elements of X	Any f	f injextive	f surjective
Distinguishable	Distinguishable	x^n	$P(X, N)$	inclusion/exclusion, counting with complement
Indistinguishable	Distinguishable	$\binom{n+x-1}{x-1}$	$\binom{X}{N}$	$\binom{n-1}{x-1}$/at least one item in each group

2 Distributions and Probability

2.1 Basics

Sample Space: A set where outcomes in probability are elements of the sample space

Events: subsets of the sample space

sample space of rolling a six-sided die is $\{1, 2, 3, 4, 5, 6\}$

Distribution: A function from the sample space to [0,1]

$f : \Omega \to [0, 1]$

2.2 Uniform/Non-uniform Distribution

Given sample space S, in a uniform distribution, for any event E, $p(E) = |E|/|S|$

Non-uniform distributions: Any distribution that is not uniform, i.e. all events do NOT have the same probability.

2.3 Binomial Distribution

Bernoulli Trial: a performance of an experiment with two possible outcomes

Binomial distribution: probability of exactly k successes in n independent Bernoulli trials, when the probability of success is p.

$$\text{Prob}(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

2.4 Conditional Probability

$P(A|B) = \frac{P(A \cap B)}{P(B)}$

2.5 Bayes Theorem

Bayes Theorem: $P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{\sum_j P(B|A_i)P(A_I)}$
Law of Total Probability: $P(E) = P(E|F)P(F) + P(E|\overline{F})P(\overline{F})$
Generalized Law of Total Probability: $P(A) = \sum_j P(A_j|B)P(B) = \sum_j P(A_j \cap B)$

2.6 Independence

Two events E and F are independent if the occurence of one event does not affect the likelihood of the other event
$P(E|F) = P(E)$
$P(E \cap F) = P(E)P(F)$

2.7 Random Sampling

Rejection Sampling: Given a uniform distribution, rejection sampling only takes samples if they're within some region of the distribution

Example: Simulating a die with a coin
Here, we'll flip a coin 3 times, giving us 8 potential outcomes. We'll assign 6 outcomes to a side on the die, then reject two outcomes. If we get our rejected outcomes, we will reflip the coin 3 times.

2.8 Random Variables

A random variable X assigns a real number to each possible outcome of an experiment ($X : S \to \mathbb{R}$)
The distribution of a random variable X is the function $r \to P(X = r)$

2.9 Expectation

The expectation (weighted average/ average expected value) of a random variable X on a sample space S is $E(X) = \sum_{s \in S} P(s)X(s)$ **Example:** What is the expectation of rolling two dice where X is the sum of the value on each die?
$E(x) = 2(\frac{1}{36}) + 3(\frac{1}{18}) + 4(\frac{1}{12}) + 5(\frac{1}{9}) + 6(\frac{5}{36}) + 7(\frac{1}{6}) + 8(\frac{5}{36}) + 9(\frac{1}{9}) + 10(\frac{1}{12}) + 11(\frac{1}{18}) + 12(\frac{1}{36}) = 7$

2.9.1 Linearity of Expectation

Law of Total Expectation: We can split up the expectation of a random variable into the expectation over a partition of the subspace.

Indicator Functions: Useful tool for calculating expectation with law of total expectation. It is defined in the following way:

$$X_i = \begin{cases} 1 & \textbf{event occurs} \\ 0 & \textbf{otherwise} \end{cases}$$

Example: Suppose 10 dancers select their dresses in a non-uniform way. First, five dancers are randomly picked, and the order in which they're picked determines their color (red is always first, polka-dot second, etc...). The last 5 dancers have their dress color uniformly sampled. What is the expected number of patterns that are only worn by one dancer?

Let X represent the number of patterns worn only once, E(X) the expected number of patterns worn only once, and X_i be an indicator variable representing if a dress is worn once.

$$X_i = \begin{cases} 1 & \textbf{If pattern is worn only once} \\ 0 & \textbf{otherwise} \end{cases}$$

Then,

$E(x) = \sum_{i=1}^{5}(E(X_i)) =$

$\sum_{i=1}^{5}(P(X_i = 1)1) =$

$\sum_{i=1}^{5}((\frac{4}{5}^5)(1))$

$\approx 1.64]$

2.10 Variance

Unexpectedness: $U = |X - E|$, where X is random variable and E is expected value

Average unexpectedness: $AU(X) = E(|X - E|) = E(U)$

Variance: $V(X) = E(|X - E|^2) = E(U^2)$

Standard deviation: $\sigma(X) = (E(|X - E|^2))^{\frac{1}{2}} = V(X)^{\frac{1}{2}}$

If X and Y are independent, then $V(X + Y) = V(X) + V(Y)$

Example: Let you and your friend be equally good at tennis, so that you both have a $\frac{1}{2}$ chance of winning. Let X_n be the win differential. Calculate $E(X_n)$ and the variance of X_n:

- The expected value is 0. Take for example the win differential after one game. It can either be -1 or 1, so the differential is 0. After 2 games, it can either be 2, 0, or -2. This can be shown more rigorously with the equation $X_n = \sum_{i=0}^{n}(\binom{n}{i}p^i(1-p)^{n-i}(i-(n-i)) = 0$

- The variance is n. As the number of matches goes up, the random variable can vary from the mean by n because either player can have a win differential of n or $-n$. We can show this rigorously: $V(X) = E(X^2) - (E(X))^2 = \sum_{i=0}^{n}(\binom{n}{i}p^i(1-p)^{n-i}(i-(n-i))^2 - (\sum_{i=0}^{n}(\binom{n}{i}p^i(1-p)^{n-i}(i-(n-i)))^2 = n$

3 Runtime Analysis

3.1 Min Sort

Given a list, we start by search iterate over the list starting from the first element to find the lowest value which is then swapped with the first element. We then find the lowest value starting from the second index and swap the second lowest value with second element. We continue this patter of finding the lowest value up to the $n - 1$ element where n is the number of elements.

	Index 1	Index 2	Index 3	Index 4	
Orginal List →	5	3	2	4	
	2	3	5	4	Best/worst case swaps/comparisons
	2	3	5	4	
	2	3	4	5	
	2	3	4	5	

	min	max
swaps	0 (ordered elements)	$n - 1$ (cyclically shift ordered elements)
comparisons	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$

3.2 Bubble Sort

Start at the first two elements and swap them if they are out of order. Continue with the second and third element, etc... until we are at the end of the list. Repeat this process from the beginning, reducing the index where we stop each time (every run through the list sorts the ending elements).

Run 1:

	Index 1	Index 2	Index 3	Index 4
Orginal List →	5	3	2	4
	3	5	2	4
	3	2	5	4
	3	2	4	5

Run 2:

	Index 1	Index 2	Index 3	Index 4
	3	2	4	5
	2	3	4	5
	2	3	4	5

Best/worst case swaps/comparisons

	min	max
swaps	0	$\frac{n(n-1)}{2}$
comparisons	$n-1$	$\frac{n(n-1)}{2}$

3.3 Insertion Sort

Given a list, we start with the second element and move it left until it is in order. We do this for each element up to the last, and the indices before the element that is being moved are considered to be in sorted order.

	Index 1	Index 2	Index 3	Index 4
Orginal List →	5	3	2	4
	3	5	2	4
	2	3	5	4
	2	3	4	5

Best/worst case swaps/comparisons

	min	max
swaps	0 (ordered elements)	$\frac{n(n-1)}{2}$ (reverse order elements)
comparisons	$n-1$	$\frac{n(n-1)}{2}$

3.4 Linear Search

Iterate over a list from the first element until item is found. If at the end of the list and item wasn't found, then the item isn't contained in the list.

	min	max
comparisons	1 (item at front)	n (item at end or not present)

3.5 Binary Search

Binary search uses three positional arguments, hi, lo, and mid, and narrows down the search region every iteration to find an element in a sorted list. For example, given a list of size n, lo and hi will be 0 and 5 respectively on first iteration, and thus mid will be 2. If the item at mid is greater than the element to search for, mid becomes hi, otherwise mid becomes lo, and the process continues.

Example: Finding 3 in $[1, 3, 4, 5, 6]$:

1	3	4	5	6
1	3	4	5	6

After the second iteration, the element is found

	min	max
comparisons	1 (item at middle)	$\lceil \log_2(n+1) \rceil$ (item at ends or not present)

3.6 Asymptotic Classes

3.6.1 Big Theta

Definition: $f(n) \in \Theta(g(n))$ if there are constants C, C', and k such that $f(n) \leq Cg(n)$ and $g(n) \leq C'f(n)$ for all $n \geq k$. In other words, Big Θ defines a tightly bound relationship (grows just as fast).

Limit definition: $f(n) \in \theta(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where c is finite and $c \neq 0$

3.6.2 Big Omega

Definition: $f(n) \in \Omega(g(n))$ if there are constants C and k such that $f(n) \geq Cg(n)$ for all $n \geq k$. In other words, Big Ω defines a lower bound relationship.

Limit definition: $f(n) \in \Omega(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where $c > 0$ or $c = \infty$

3.6.3 Little O

Definition: if $g(n)$ grows strictly faster than $f(n)$, then $f(n) \in o(g(n))$. Put another way, $f(n) \in o(g(n))$ and $f(n) \notin o(g(n))$

Limit definition: $f(n) \in o(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0$

3.6.4 Big O

Definition: $f(n) \in O(g(n))$ if there are constants C and k such that $f(n) \leq Cg(n)$ for all $n \geq k$. In other words, $g(n)$ grows just as fast or faster than $f(n)$.
Limit definition: $f(n) \in O(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where c is finite
Examples:

- $2^n \in O(n^2)$
  $\lim_(n \to \infty) \frac{2^n}{n^2} =$ (Apply L'Hopital's rule)
  $\lim_(n \to \infty) \frac{2^n \ln 2}{2n}$ (Apply L'Hopitals rule
  $\lim_(n \to \infty) \frac{2^n \ln 2^2}{2} = \infty$
  Thus, the statement is false.

- $F_n \in O(2^n)$ where F_n is a function representing the fibonacci sequence.
  $F_n = F_{n-1} + F_{n-2}, F_0 = 1, F_1 = 1$
  Claim: $F_n \leq 2^n$ for all $n \geq 0$
  Base cases:
  $F_0 = 1, 2^0 = 1$
  $F_1 = 1, 2^1 = 2$
  Induction step:
  Let k be an arbitrary integer such that $k > 1$.
  Assume that $F_m \leq 2^m$ for all m in the range $0 \leq m \leq k$.
  WTS $F_k \leq 2^k$:
  $F_k = F_{k-1} + F_{k-2} \leq 2^{k-1} + 2^{k-2} = 2^{k-2}(2+1) \leq 2^{k-2}(4) = 2^k$
  Thus, the statement is true.

3.6.5 Big O Class Properties

Domination: If $f(n) \leq g(n)$ for all n then $f(n) \in O(g(n))$
Transitivity: If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$
Additivity/ Multiplicativity: If $f(n) \in O(g(n))$ and $h(n)$ is non-negative then
$f(n) + h(n) \in O(g(n) + h(n))$ and $f(n) \cdot h(n) \in O(g(n) \cdot h(n))$
Sum is maximum: $f(n) + g(n) \in O(\max\{f(n), g(n)\})$
Ignoring constants: For any constant c, $c \cdot f(n) \in O(f(n))$

3.6.6 Growth Rate of Common Functions

1 (fastest)	$\log(\log n)$	$\log n$	$(\log n)^k$	n	$n \log n$	n^k	a^n	$n!$	n^n (slowest)
constant	double-logarithm	logarithm	poly-logarithm	linear	log-linear	polynomial	exponential	factorial	

3.6.7 Disjoint Lists Function

Given two sorted lists, $a[1], \ldots, a[n]$ and $b[1], \ldots, b[n]$, determine if they are disjoint or not.

Method 1:

for i in $1 \ldots n$

    if **BinarySearch**$((b[1], \ldots, b[n]), a[i]) \neq 0$

        return false;

    return true;

Takes $O(nlogn)$ time.

Faster ways: Hashmap, double while loop that we implement

Method 2, double while loop:

i=1

j=1

while $i < n$

    while $j < n$

        if $a[i] = b[j]$ **then**

            return FALSE

        else

            if $a[i] < b[j]$ **then**

                $i = i + 1$

            if $a[i] > b[j]$ **then**

                $i = j + 1$

return TRUE

3.7 Product Rule

If the inner loop runs $O(T_1(n))$ and the outer loop runs $O(T_2(n))$, worst case runtime is $O(T_1(n)T_2(n))$. Note, this is not always a tight bound (total time could be much faster).

3.8 Loop Invariant Induction

Loop Invariant: A property that remains true after each time the body of a loop is executed.

3 Step Plan for iterative algorithm

1. Look for a loop invariant
2. Prove that it is an invariant
3. Use invariant to prove correctness

3.8.1 Selection Sort Loop Invariant Induction

Loop Invariant: After t iterations, the first t elements are in sorted order and the first t elements are the smallest.

Base Case: $(t = 0)$ After 0 iterations,

- The first 0 elements are in sorted ordered (vacuously true)
- The first 0 elemets are the smalles (vacuously true)

Inductive Hypothesis: Suppose that for some $t \geq 0$, the loop invariant is true after t iterations.

WTS: The inductive hypothesis is true after $t + 1$ iterations

During the $(t + 1)$th iteration, the algorithm sets a_m to be the minimum value of $a_{t+1}, \ldots, a_n$, and a_m is then swapped with a_{t+1}. So after the $(t + 1)$th iteration, a_{t+1} is the minimum value of $a_{t+1}, \ldots, a_n$.

By the inductive hypothesis, $a_1, \ldots, a_t$ are in sorted order and are the smallest t elements. Therefore we have that $a_1, \ldots, a_t$ are all less than a_{t+1} and a_{t+1} is less than all of $a_{t+1}, \ldots, a_n$, so $a_1, \ldots, a_{t+1}$ are the smallest $(t+1)$ elements of the original list. Additionally, we have that $a_1, \ldots, a_t$ are all in sorted order and since $a_1, \ldots, a_t$ are all less than a_{t+1}, $a_1, \ldots, a_{t+1}$ are all in sorted order.

3.8.2 Find Max Loop Invariant Induction

Invariant: After each iteration, max is the maximum value of the list $(a[1], a[2])(i = 2)$.

Proof:

Base case: After 0 iterations, max is maximum $(a[1])$. After one iteration, max is the maximum of $(a[1], a[2])$ $(i = 2)$

Let $t \geq 1$ be an arbitrary integer.

Assume that after t iterations, max is a maximum of $(a_1, \ldots, a_t)$.

Let $i' = i + 1$. **WTS** after t+1 iterations, max is a maximum of $(a_1, \ldots, a_{i'})$.

During the t+1 iteration, $i' = i + 1$. There are two cases to consider:

Case 1: $a'_i >$ max. max is maximum of $a_1, \ldots, a_i$. Thus, the maximum is a'_i, and max updates to be a'_i. Therefore, max is a maximum of $(a_1, \ldots, a_{i'}$.

Case 2: $a'_i <$ max. max is maximum of $a_1, \ldots, a_i$ and max never updates. Therefore, max is a maximum of $(a_1, \ldots, a_{i'}$.

4 Recursion

4.1 Recursive find max proof

Findmaxrec(a[1...n] array of distinct integers)

if n == 1 then return a[1]

M1 = Findmaxrec(a[1,...n-1])

return MAXIMUM(M1,a[n])

Proof

Base Case: $n = 1$, the list has one element $a[1]$ and the one element is the maximum of the list.

Inductive Hypothesis: Let k be an arbitrary integer such that $k > 1$. Assume that Findmaxrec is correct on all inputs of length $k - 1$.

Inductive Step: $M1$ is set to Findmaxrec(a[1], ... ,a[k-1]), and by the inductive hypothesis, $M1$ is the max of a[1], ... a[k-1]. The maximum of a[1],...,a[k] is the maximum of $M1$ and a[k] which is what the algorithm returns.

4.2 Merge sort (divide and conquer)

Algorithm

Rmerge($[a_1, \ldots, a_k], [b_1, \ldots, b_l]$)

if $k == 0$ **then return** $[b_1, \ldots, b_l]$

if $k == 0$ **then return** $[a_1, \ldots, a_k]$

if $a_1 < b_1$**then:**

return $a_1 \cdot$ Rmerge($[a_2, \ldots, a_k], [b_1, \ldots, b_l]$)

else

return $a_1 \cdot$ Rmerge($[a_1, \ldots, a_k], [b_2, \ldots, b_l]$) Recursive Proof:

Proof by strong induction:

Base case: $n = 0$. Then, return the empty list (trivially sorted)

Suppose $n = 1$. Then return a_1, a trivially sorted list containing all elements

Induction step: Suppose $n > 1$. Assume, as the strong induction hypothesis, that

MergeSort correctly sorts all lists with k elements for any $0 \leq k < n$ WTS: prove that MergeSort($a_1, \ldots, a_n$) returns a sorted list containing all n elements for any arbitrary list $a_1, \ldots, a_n$

$m = \lfloor \frac{n}{2} \rfloor$ (Note that $0 \leq m < n$)

$L_1 =$ MergeSort($a_1, \ldots, a_m$) ($m < n$) so L_1 is a sorted list of these elements (By IH)

$L_2 =$ MergeSort($a_{m+1}, \ldots, a_n$) so L_2 is a sorted list of these elements (By IH)

By the correctness of RMerge, RMerge(L_1, L_2) is a sorted list of all elements.

Recurrece relation:

Rmerge($[a_1, \ldots, a_k], [b_1, \ldots, b_l]$) T(n)

if $k == 0$ **then return** $[b_1, \ldots, b_l]$ O(1)

if $k == 0$ **then return** $[a_1, \ldots, a_k]$ O(1)

if $a_1 < b_1$**then:** O(1)

return $a_1 \cdot$ Rmerge($[a_2, \ldots, a_k], [b_1, \ldots, b_l]$) T($\frac{n}{2}$)

else

return $a_1 \cdot$ Rmerge($[a_1, \ldots, a_k], [b_2, \ldots, b_l]$) T($\frac{n}{2}$)

$T(n) = 2T(\frac{n}{2}) + O(n)$

Solving recurrence relation

Unravelling:

$T(n) = 2T(\frac{n}{2}) + cn$

$T(n) = 2(2T(\frac{n}{2^2}) + c(\frac{n}{2})) + cn$

$T(n) = 2^2T(\frac{n}{2^2}) + cn + cn$

$T(n) = 2^2(2T(\frac{n}{2^3}) + c(\frac{n}{2^3})) + cn + cn$

$T(n) = 2^3T(\frac{n}{2^3}) + cn + cn + cn$

$T(n) = 2^kT(\frac{n}{2^k}) + kcn$

$\vdots$

$\log_2 nT(n) = 2^{\log_2 n}T(\frac{n}{2^{\log_2 n}} + (\log_2 n) \cdot c \cdot n = nT(1) + cn \log_2 n = c_1 n + cn \log_2 n$

4.3 Master theorem

$T(n) = aT(\frac{n}{b}) + O(n^d)$

- **a**: The number of recursive calls
- **b**: Fraction of the original input size of recursive calls
- **d**: Degree of polynomial of the number of non recursive part

$O(n^d)$	if $a < b^d$
$O(n^d logn$	if $a = b^d$
$O(n^{log_b a}$	if $a > b^d$

4.3.1 Mergesort

Alternative way of solving recurrence relation (original way is unravelling)

Recurrence relation: $T(n) = 2T(\frac{n}{2}) + O(n)$

$a = 2, b = 2, d = 1$. Therefore, $a = b^d$, and the recurrence relation is $O(n^1 \log n)$

4.4 Homogeneous Recurrence Relations

4.4.1 Domino Tilings

How many ways can we fill a 2 by n grid with dominos? (Domino can either be vertical or horizontal)
$DT(n)$ is the number of different domino tilinigs of a 2 by n grid
$DT(n) = DT(n-1) + DT(n-2)$ for $n > 2$ $DT(n-1)$ represents placing a domino vertical in first column, and $DT(n-2)$ represents placing placing two horizontally stacked dominos in the first two columns. The base cases are $DT(1) = 1$ and $DT(2) = 2$.

4.4.2 Characteristic Polynomial

Guess the polynomial: $DT(n) = Ar^n$
Eliminate:
If recurrence relation is $DT(n) = DT(n-1) + DT(n-2)$, we replace the recurring terms with a polynomial such that $Ar^n = Ar^{n-1} + Ar^{n-2}$, then simplify so that it become $r^2 = r + 1$
Solve the roots:
In this case, the roots are $\frac{1 \pm \sqrt{5}}{2}$.
Write a characteristic polynomial:
We can rewrite the recurrence relation as $DT(n) = A_1 x_1^n + \cdots + A_k x_k^n$ when the polynomial has k roots.

4.4.3 Fibonacci

Fibonacci sequence: 1,1,2,3,5,8,13,21,...
$Fib(n) = Fib(n-1) + Fib(n-2)$ for all $n > 2$ where $Fib(1) = 1$ and $Fib(2) = 2$

4.5 Recursive Counting

4.5.1 Permutations/Stirling's Approximation

Stirling's Approximation $n! \approx (\frac{n}{e})^n \sqrt{2\pi n}$ (nearest integer)
Permutations $S(n) = n \cdot S(n-1) = n!$
Approximate with stirling's approximation

4.5.2 n-bit strings

Avoid the substring 11
List out substrings:

1. A(0) = 1
2. A(1) = 2

3. A(2) = 3

4. A(3) = 5

$A(n)$ = number of n-bit 11-avoiders starting with 0 + number of n-bit 11-avoiders starting with 1
$A(n) = A(n-1) + A(n-2)$
$A(0) = 1, A(1) = 2$ Devolves into fibonacci.
Avoid the substring 111
$B(n) = B(n-1) + B(n-2) + B(n-3)$ where $B(0) = 0$, $B(1) = 2$, and $B(2) = 4$

4.5.3 Solving Recurrences

1. **Guess and Check**: Start with small values of n and look for a pattern. Confirm guess and check with proof by induction.

2. **Unravel**: Start with the general recurrence and keep replacing n with smaller input values. Keep unraveling until you reach the base case.

3. **Characteristic Polynomial**: If the recursion is of a certain form, you can guess that the closed form is Cw^n and solve for w by finding the root of the polynomial. (Note: better for finding a bound rather than closed form)

5 Encoding

5.1 Lossy vs. Lossless encoding

Lossy encoding: Some of the data is lost in tge encoding process.
Lossless econding: The data can be restored to its original state without any losses.

5.2 Fixed-width encoding

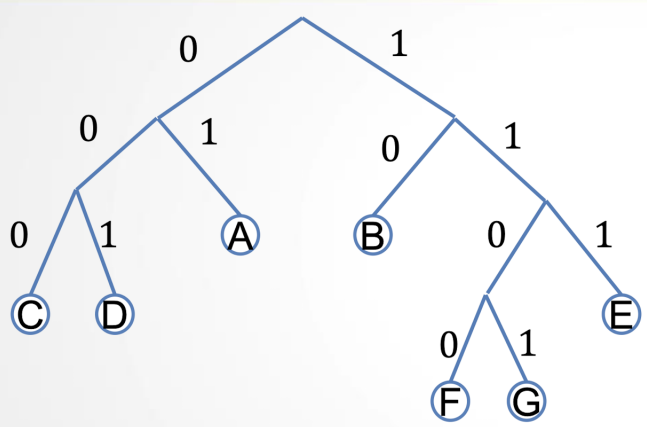
First determine the number of bits we need to encode all symbols in the alphabet by calculating $\lceil \log_2 n \rceil$, where n is the number of symbols in the alphabet. Then, assign binary numbers to each character with the number of bits we have. We can then construct new strings with these binary numbers.

5.3 Huffman encoding

type of variable length encoding where symbols that appear more frequent are represented by shorter length encodings.

Building Huffman Tree

Group together the two lowest frequencies and break ties by using alphabetical order left to right.



Letter	Encoding	Code length
A	01	2
B	10	2
C	000	3
D	001	3
E	111	3
F	1100	4
G	1101	4

Use the Huffman tree above to decrypt:

1 0 0 1 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1

5.4 Fibonacci Encoding

Zeckendorf's Theorem: Each positive integer can uniquely be represented as the sum of non-consecutive Fibonacci numbers.

- We can represent a sequence of positive integers with binary strings that use two ones in a row to signal the start of a new integer.

Example: We want to encode (2,5,24,15,1) which is 01100011001000110100011111

n	1	2	3	5	8	13	21
2	0	1					
5	0	0	0	1			
24	0	0	1	0	0	0	1
15	0	1	0	0	0	1	
1	1						

5.5 Ranking and Unranking

Ranking:

For each 1 in the binary string, calculate the sum of binomials $\binom{n}{k}$, where n is each 1's position and k is the order of the 1 (going left to right).

Unranking:

Convert the binary string into a number, r. Then, start from the highest position, k. Find the greatest n such that $\binom{n}{k} \leq r$. Then subtract this value from r to net a new r. Repeat until r is 0.

5.6 Optimal length encoding

The optimal length encoding is $\lceil \log_2 n \rceil$, where n is the total number of unique strings. This encoding can be achieved with a dictionary, or with special algorithms.

6 Graph Theory

6.1 Graphs

6.1.1 Directed Graphs

- Edges have a direction
- $n(n-1)$ potential edges on n vertices
- $2^{n(n-1)}$ different directed graphs on n vertices
- InDegree is the number of edges going to vertex
- OutDegree is the number of edges going from vertex
- A directed graph is strongly connected if for any ordered pair of vertices (v, w), there is a directed path from v to w
- A directed graph is weakly connected if the arrows are removed and the resulting undirected graph is connected

6.1.2 Directed Acyclic Graph (DAG)

Directed acyclic graph: directed graph with no cycles.
Sources: Vertices with no incoming edges **Sinks:** Vertices with no outgoing edges **Topological ordering/linearization:** An ordered list of all of a graphs' vertices such that for each directed edge (v,w) in the list, v comes before w.

6.1.3 Undirected graphs

- Edges don't have direction
- $\binom{n}{2}$ potential edges on n vertices
- $2^{\binom{n}{2}}$ different simple undirected graphs on n vertices
- Degree of vertex is the total number of edges incident with it (self-loop contributes twice)
- $2 \cdot E = s$ where E is the number of edges and s is the sum of degrees
- A undirected graph is connected if for any pair of vertices (v, w) there is a path from v to w

6.1.4 Connectedness

- Connected graph: A graph is connected if for any pair of vertices, v and w, there is a path between them.
- Weakly connected graph: A graph is weakly connected when if you remove all the arrows, the resulting graph is connected.
- Strongly connected graph: A graph is strongly connected if there is a directed path between any pair of vertices, v and w.
- Connected components: A connected component is a subsection of a graph that is connected.
- Bipartite graphs: A bipartite graph is when all the vertices can be arranged into two groups such that two vertices in the same group are not connected by an edge.
- Complete graphs: A complete graph is an undirected graph such that there is an edge between any two vertices.

6.2 Hamiltonian paths

Hamiltonian paths visit every vertex exactly once.

Example

DNA reconstruction where we have N 3-character DNA strings and we want to find the shortest possible length DNA string that includes them all $(N + 2)$. We can solve this by considering the 2 characters that the strings can share. We can create a hamiltonian path by defining the vertex set as each DNA string, and the edge set as the edge from v to w where the first 2 characters of w match the last 2 characters of v.

6.3 Eulerian paths

Eulerian paths visit every edge exactly once.

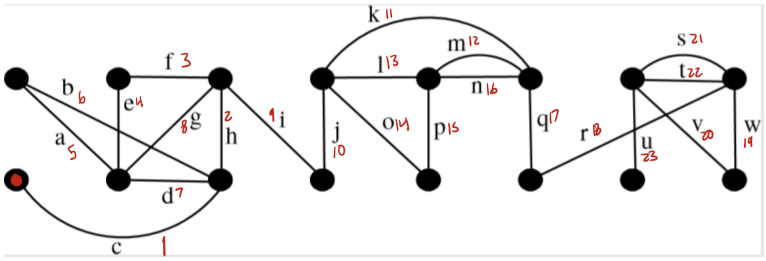
If G is an undirected path and has an Eulerian trail, G has at most two odd-degree vertices

Example

Based on the example with DNA strings in the previous section, we can use a eulerian path by defining the vertex set as all length-two strings that appear and the edge set as an edge from ab to bc such that abc is in the set of strings.

Fleury's algorithm:

- Start at vertex v with odd-degree if possible
- Continue to cross any edge that is not a bridge



6.4 Adjacency matrix

Matrix which encodes the connections between two vertices. In directed graphs, the connections between two vertices are not recorded twice. Self loops are recorded along the main diagonal. For directed graphs, the number of parallel edges are specified, not just a 1 or 0 denoting the connection

6.5 Trees

Types

- Binary tree: each node has at most 2 children
- Complete binary tree: a binary tree in which every level, except possibly the last, is completely filled and all nodes on the last level are as far left as possible
- Full binary tree: a binary tree where every node has 0 or 2 children
- Unrooted tree: An undirected graph if it is connected and has no cycles
  - Leaves in tree have vertices of degree 1
  - All trees with n vertices have $n-1$ edges
  - A set of trees is called a forest
  - 1 simple path between each pair of vertices
- Binary Search Tree: For each node, every node in the left sub-bst has a smaller value than the node and every node in the right sub-bst has a greater value that the node. $O(logn)$ for searching, insertings, and deleting.
- Decision Tree: binary tree that uniquely characterizes elements of a set based on the answers to yes or no questions.

Runtime

operation	best case	average case	worst case
search	$O(1)$	$O(\log n)$	$O(n)$
insert	$O(1)$	$O(\log n)$	$O(n)$
delete	$O(1)$	$O(\log n)$	$O(n)$

Worst case is only possible with unbalanced tree

7 Randomized Algorithms

Deterministic Algorithms: Generate an output based on a determined set of rules and always work exactly the same on the same input.

Random Algorithms: Generate an output based on a source of random numbers and determined set of rules. May work differently on the same input.

7.1 Las Vegas Algorithms

- Always returns correct answer
- Runtime varies

Example: QuickSort

Quicksort

Like merge sort, but chooses pivot instead of dividing into 2. As opposed to mergesort, quicksort is an in-place sorting algorithm. Below is the general algorithm.

- Start with middle index as pivot and swap data so values greater than pivot are on the right and values left of the pivot are on the left
- Partition function to move values around pivot
- Start with lowindex and increment until number at lowindex is greater than pivot
- Start with highindex and decrement until number at highindex is less than pivot
- If lowindex is greater than highindex, then partition is done
- Otherwise swap lowindex and highindex and shift indexes by one and return high index
- Quicksort the left portion and right portion again recursively (low index to high index, high index + 1to n)

Runtime

Best case: $O(n \log n)$

Worst case: $O(n^2)$

While quicksort always returns the list in the correct, sorted order, because its runtime varies, it is a Las Vegas Algorithm.

7.2 Monte Carlo Algorithms

- Not always correct
- Runtime is consistent