Contents

1 Basic Probability Questions . . . 3
- 1.1 Basic Probability . . . 3
- 1.2 Target Practice . . . 3
- 1.3 Basic Expectation . . . 3
- 1.4 More Expectation . . . 4
- 1.5 Balls and Urns . . . 4
- 1.6 Coin Tosses . . . 4
- 1.7 Dice Rolling . . . 4
- 1.8 The Monte Hall Dilemna . . . 4
- 1.9 Passing Students . . . 5
- 1.10 Maximal Elements . . . 5
- 1.11 The Miner's Problem . . . 7
- 1.12 The Birthday Paradox Using Variance . . . 7
- 1.13 Generating a Random Permutation . . . 8
- 1.14 Matchings . . . 8

2 Randomized Algorithms - Basic Analysis . . . 9
- 2.1 Finding a Large Independent Set in a 3-Regular Graph . . . 9
- 2.2 Finding a Small Dominating Set in a 3-Regular Graph . . . 9
- 2.3 Frequency Assignment in Wireless Networks . . . 10
- 2.4 Multiplicative Hashing — The Wrong Way . . . 10
- 2.5 Universal Hashing . . . 10
- 2.6 Success Amplification . . . 11
- 2.7 A Recursive Algorithm on Lists . . . 11
- 2.8 An RFID Wakeup Protocol . . . 11
- 2.9 Computing the OR of a Bit String . . . 12
- 2.10 3-Way Partitioning . . . 12
- 2.11 Searching an Unsorted Array . . . 12
- 2.12 List Ranking by Independent Set Removal . . . 13
- 2.13 Tightening Up Quicksort . . . 13
- 2.14 Approximating Max-2-Sat . . . 13
- 2.15 Ternary 0-1 Game Trees . . . 14
- 2.16 A Randomized Recurrence . . . 14
- 2.17 Analysis of Random Binary Search . . . 14
- 2.18 Analysis of Hoare's Find Algorithm . . . 15
- 2.19 Finding the One . . . 16
- 2.20 Matrix Equality Testing . . . 16
- 2.21 A Monte-Carlo Min-Tricut Algorithm . . . 17
- 2.22 The height of a skiplist . . . 17

3 Applications of Chernoff's Bounds **19**
3.1 Monte-Carlo Landslide Finding . . . 19
3.2 Estimating π . . . 19
3.3 An Independent Set of Friends . . . 20
3.4 Coarse-grained parallel sorting . . . 20
3.5 A Random Walk on a Grid . . . 21
3.6 Kick the Stones . . . 21
3.7 Finding large independent sets in parallel . . . 21
3.8 McDiarmid's Inequality . . . 22

4 Amortized Analysis **23**
4.1 Stacks with Multipops . . . 23
4.2 A Min Queue . . . 23
4.3 Dynamically Growing and Shrinking Arrays . . . 24
4.4 Redundant Binary Counters . . . 24
4.5 Data Structures for Disjoint Sets . . . 25
4.6 Tree Data Structures for Disjoint Sets . . . 25
4.7 Lazy Deletion . . . 26
4.8 Lazy Insertion Data Structures . . . 26
4.9 List-Based Priority Queues . . . 26
4.10 Jeff Erickson's Countdown Trees . . . 27

5 NP-Hard Problems and Algorithms **28**
5.1 NP-Completeness of PARTITION . . . 28
5.2 NP-Completeness of $(\frac{1}{3}, \frac{2}{3})$-PARTITION . . . 28
5.3 NP-Completeness of CROSS-COVER . . . 29
5.4 NP-Completeness of LIGHT-SWITCHING . . . 30
5.5 Converting Decision Algorithms to Optimization Algorithms . . . 30
5.6 NP-hardness of PLANAR-CIRCUIT-SAT . . . 30
5.7 NAND-SAT is NP-hard . . . 31
5.8 NP-Hardness of INDEPENDENT-SET . . . 31
5.9 NP-Hardness of SUBGRAPH-ISOMORPHISM . . . 31
5.10 NP-Hardness of SET-COVER . . . 31
5.11 NP-Hardness of HAMILTONIAN-PATH . . . 31
5.12 NP-Hardness of and Algorithms for 3-HITTING-SET . . . 32
5.13 Self-Reducibility of VERTEX-COVER . . . 32
5.14 A Fast Algorithm for MAX-CLIQUE in Graphs of Bounded Degree . . . 32
5.15 Independent Set of Squares . . . 33

Chapter 1

Basic Probability Questions

1.1 Basic Probability

Pick a random integer x uniformly in $\{1, \ldots, 1000\}$.

1. What is $\Pr\{x \text{ is divisible by } 5\}$?
2. What is $\Pr\{x \text{ is divisible by } 10\}$?
3. What is $\Pr\{x \text{ is divisible by } 11\}$?
4. What is $\Pr\{x \text{ is divisible by } 5 \text{ or } x \text{ is divisible by } 11\}$?
5. What is $\Pr\{x \text{ is divisible by } 5 \text{ or } x \text{ is divisible by } 10\}$?
6. What is $\Pr\{x \text{ is divisible by } 2 \text{ or } x \text{ is divisible by } 5 \text{ or } x \text{ is divisible by } 11\}$? (Hint: Read about inclusion-exclusion)

1.2 Target Practice

Al and George go target shooting. Al hits the target with probability p_1 and George hits the target with probability p_2. Assuming they both shoot simultaneously and *someone hits the target*,

1. what is the probability that both Al and George hit the target?
2. what is the probabilily that Al hit the target?

1.3 Basic Expectation

It costs \$4 to play the following game: Roll a 6-sided die and get paid the amount shown on the die.

1. What is the expected amount will get paid if you play this game?
2. What is the expected amount you will gain if you play this game?

1.4 More Expectation

The chances of winning a particular lottery are one in ten-million, and it costs \$1 to buy a lottery ticket.

1. How large must the jackpot be so that the expected amount you gain by buying a lottery ticket is at least 0?
2. How large must the jackpot be so that the expected amount you gain by buying 5 lottery tickets is at least 0?

1.5 Balls and Urns

We toss n balls into m urns, so that each ball is equally likely to land in each urn and all tosses are independent.

1. What is the expected number of balls in the first urn?
2. What is the probability that the first ball does not land in the first urn?
3. What is the probability that urn 1 is empty?
4. What is the expected number of empty urns?
5. What is the expected number of urns with at least one ball?

1.6 Coin Tosses

We repeatedly toss a fair coin until it comes up tails, then we stop. What is the expected number of coin tosses we make?

1.7 Dice Rolling

We repeatedly roll a pair of dice until their sum is 7.

1. What is the probability that the first roll is a 7?
2. What is the expected number of times we have to roll the dice?

1.8 The Monte Hall Dilemna

You are on a game show with three doors, D_1, D_2 and D_3. Behind a randomly selected door is a new sports car. Behind the other two doors are donkeys. The game proceeds as follows:

1. You select some door (say D_1)
2. The game show host opens one of the other doors that contains a donkey behind it (say D_3)
3. You are asked to guess whether you want to stick with your first guess (D_1) or change your guess and select the other unopened door (D_2).
4. The host opens the door you selected in Step 3 and you win the prize behind the door (the donkey or the car).
5. Before the game begins, what is the probability that the sports car is behind door 1?
6. If you stick with your original guess in Step 3, what is the probability that you win the car in Step 4?
7. If you change your guess in Step 3, what is the probability that you win the car in Step 4?

1.9 Passing Students

In a certain fourth-year course at a certain university, the expected number of students that successfully complete the course is $n/2$, where n is the number of students enrolled in the course. Some years the professor is harder than other and students sometimes collaborate, so the event "Student A successfully completes the course" is not necessarily independent of the event "Student B successfully completes the course."

1. Given an upper bound on the probability that more than $3n/4$ students successfully complete the course.

2. Given an upper bound on the probability that less than $n/4$ students successfully complete the course.

1.10 Maximal Elements

Let S be a set of n points uniformly distributed in the unit square, so that each point $p_i = (x_i, y_i)$ where x_i and y_i are uniformly distributed on $[0, 1]$. We say that a point $p \in S$ is *maximal* if there is no other point of S in the upper right quadrant of p (see below):

What is the expected number of maximal points in S?

1.11 The Miner's Problem

For this question you may make use of *Wald's Equation*: If $X_1, X_2, \ldots$ are independent and identically distributed random variables having expected value $\mathbf{E}[X]$ and N is a non-negative integer chosen independently of the X_i then

$$\mathbf{E}\left[\sum_{i=1}^{N} X_i\right] = \mathbf{E}[N]\mathbf{E}[X]$$

Now for the question: A miner is trapped in a room containing 3 doors. Door one leads to a tunnel that returns to the room after 4 days. Door two leads to a tunnel that returns to the room after 7 days. Door three leads to freedom after a 3 day journey. Suppose the miner is equally likely to choose each of the three doors at any given time.

1. What is the expected number of attempts (trying doors) the miner will make before finding the door that leads to safety?
2. Assuming the miner chooses one of the first two doors, what is the expected time (in days) the miner will spend before returning to the room?
3. Using Wald's Equation, compute the expected time it takes for the miner to reach safety.

1.12 The Birthday Paradox Using Variance

Here's an interesting approach to the birthday paradox using variances.

Suppose we have m people who have birthdays spread uniformly over n days. We want to bound m such that the probability that there are are at least two people with the same birthay is about one-half.

For 1 i ¡ j m, let Ai,j be a random variable taking value 1 if the ith and jth person share the same birthday and zero otherwise. Let A be the sum of the Ai,j. At least two people have the same birthday if A 1.

E(Ai,j) = 1/n so by linearity of expectations, E(A) = m(m-1)/2n. By Markov's inequality, Prob(A 1) E(A) so if m(m-1)/2n 1/2 (approximately m n1/2), the probability that two people have the same birthday is less than 1/2.

How about the other direction? For that we need to compute the variance. Var(Ai,j) = E(Ai,j2)-E2(Ai,j) = 1/n-1/n2 = (n-1)/n2.

Ai,j and Au,v are independent random variables, obvious if i,ju,v = but still true even if they share an index: Prob(Ai,jAi,v = 1) = Prob(The ith, jith and vth person all share the same birthday) = 1/n2 = Prob(Ai,j=1)Prob(Ai,v=1).

The variance of a sum of pairwise independent random variables is the sum of the variances so we have Var(A) = m(m-1)(n-1)/2n2.

Since A is integral we have Prob(A 1) = Prob(A ¿ 0) Prob(—A-E(A)— ¿ E(A)) Var(A)/E2(A) by Chebyshev's inequality. After simplifying we get Prob(A 1) 2(n-1)/(m(m-1)) or approximately 2n/m2. Setting that equal to 1/2 says that if m 2n1/2 the probability that everyone has different birthdays is at most 1/2.

If m is the value that gives probability one-half that we have at least two people with the same birthday, we get n1/2 m 2n1/2, a factor of 2 difference. Not bad for a simple variance calculation.

Plugging in n = 365 into the exact formulas we get 19.612 m 38.661 where the real answer is about m = 23.

1.13 Generating a Random Permutation

Suppose you have access to a function $\textsc{Rand}(i)$ that returns an integer uniformly distributed in the set $\{0, \ldots, i\}$. Subsequent calls to the function produce independent results.

1. Starting with an array A containing the elements $1, \ldots, n$ in sorted order show how to compute a random permutation of A in $O(n)$ time. Note, you must prove that the permutation generated by your algorithm random. I.e., you must prove that each of the $n!$ possible permutations is equally likely.
2. What does your algorithm produce if the array A is not initially in sorted order, but contains the elements $1, \ldots, n$ is some arbitrary order?

1.14 Matchings

We have a bag of n candies, $n/2$ of which are lemon and $n/2$ of which are lime. Consider the following experiment: We reach into the bag and pull out two candies. If they of the same type, we put them both back in the bag. If they are of different types (one lemon and one lime) we put them both in our mouths.

1. What is the probability that we put the candies in our mouth? (Warning: the correct answer is not $1/2$, but it is close when n is big.)
2. What is the expected number of times we have to repeat this experiment until we get to put some candy in our mouth?
3. Suppose we consider the same experiment except that now the bag contains n_1 lemon candies and n_2 lime candies. What is the probability that we put the candies in our mouth?
4. If we start with a bag having $n/2$ lemon candies and $n/2$ lime candies then what is the expected number of times we have to repeat this experiment before the bag is empty?
5. Show that, if we start with $n/3$ lime candies and $2n/3$ lemon candies in the bag, then the expected number of times we have to repeat this experiment before we run out of lime candies is $\Omega(n \log n)$. (Hint: you will need to use the fact that $H_n = \Theta(\log n)$).

Chapter 2

Randomized Algorithms - Basic Analysis

2.1 Finding a Large Independent Set in a 3-Regular Graph

In this question we give an algorithm to find a large independent set in a 3-regular graph. (An independent set is a set of vertices, no two of which are adjacent.)

We have a graph $G = (V, E)$ in which every vertex has degree 3 and $|V| = n$. For each vertex $v \in V$, we color v *black* with probability p and color it *white* with probability $1 - p$ and this is done indpendently for each vertex. We say that a vertex v is *good* if v is colored black and v's three neighbours are all colored white.

1. What is the probability that a particular vertex v is good?
2. What is the expected number of good vertices?
3. Find the value p that maximizes the expected number of good vertices. (Hint: this is a calculus question that you may use a computer math system like Maple to help you solve.)
4. The above algorithm does not generate a very large independent set. It's main advantage is that it's easy to implement in parallel. Describe and analyze a simple algorithm that guarantees an independent set of size at least $n/4$. (Hint: The solution relies only on the fact that G is 3-regular.)

2.2 Finding a Small Dominating Set in a 3-Regular Graph

In this question we give an algorithm to find a small dominating set in a 3-regular graph. A *dominating set* is a set of vertices such that every other vertex is either in the set or adjacent to a vertex in the set.

We have a graph $G = (V, E)$ in which every vertex has degree 3 and $|V| = n$. For each vertex $v \in V$, we color v *black* with probability p and color it *white* with probability $1 - p$ and this is done independently for each vertex. We say that a vertex v is *covered* if v or one of its neighbours is black, otherwise it is *uncovered*.

1. What is the expected number of uncovered vertices?
2. If we take the set of black vertices plus the set of uncovered vertices, we obtain a dominating set of G. What is the expected size of this dominating set?
3. What is the value of p that minimizes the expected size of this dominating set? (Hint: you probably want to use a computer math system for this.)

4. What is the smallest dominating set G could possibly have? (Hint: The solution relies only on the fact that G is 3-regular.)

2.3 Frequency Assignment in Wireless Networks

We have a graph $G = (V, E)$ in which every vertex has degree 6, $|V| = n$ and $|E| = m$. For each vertex $v \in V$, we color v uniformly (and independently from all other vertices) at random with a color selected from the set $\{1, \dots, k\}$.

1. We say that an edge $e = (u, v)$ is *good* if u and v are assigned different colors in the above experiment and *bad* otherwise. What is the probability that an edge e is bad?
2. What are the expected numbers of bad edges and good edges?
3. We say that a vertex v is *dead* if all 6 of v's incident edges are bad. What is the probability that a particular vertex v is dead? What is the expected number of dead vertices?
4. How many colors k do we need if we want the expected number of dead vertices to be at most: (a) $n/10$, (b) $n/100$, and (c) $n/1000$

2.4 Multiplicative Hashing — The Wrong Way

This exercise studies why the choice of a in the multiplicative universal hashing algorithms is important. Recall that the multiplicative hashing scheme that takes elements from $\{0, \dots, 2^k - 1\}$ onto $\{0, \dots, 2^\ell - 1\}$ using the hash function $h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell}$.

1. Suppose m/ℓ is an integer and consider the set of keys
$$S = \{j(2^m/2^\ell) : j \in \{0, \dots, 2^\ell - 1\}\} \ .$$
Suppose a is of the form $2^i t$ where t is an odd integer. How many distinct values are in the set
$$h_a(S) = \{h_a(x) : x \in S\} \ ?$$
2. Suppose we choose a uniformly at random from $\{0, \dots, 2^k - 1\}$. What is the probability that a is of the form $2^i t$ where t is an odd integer?
3. Suppose we choose a uniformly at random from $\{0, \dots, 2^k - 1\}$ and use this to store the set S described in Part 1. What is the expected time to search for a value $x \in S$ in the resulting hash table?

2.5 Universal Hashing

Suppose we store n distinct elements $x_1, \dots, x_n$ using an array of m lists $L_1, \dots, L_m$ in the following way. For each x_i we have a *hash value* $h(x_i) \in \{1, \dots, m\}$ and we store x_i in the list $L_{h(x_i)}$. The hash funcion h is chosen in such a way that, for $x \neq y$,
$$\Pr\{h(x) = h(y)\} \leq 1/m \ .$$

1. Consider a value $x \notin \{x_1, \dots, x_n\}$. Use indicator variables to find the best upper bound possible on the expected size of the list $L_{h(x)}$. That is, prove that $\mathbf{E}[|L_{h(x)}|] \leq$ *blah* for the appropriate value *blah*
2. Consider a value $x_i \in \{x_1, \dots, x_n\}$. Give the best upper bound you can on $\mathbf{E}[|L_{h(x_i)}|]$.
3. For a particular list L_j with $j = h(x)$ and a particular number $d > 0$, give the best upper bound you can on $\Pr\{|L_j| \geq d\}$.

2.6 Success Amplification

Suppose you have a Monte-Carlo algorithm $\mathcal{A}$ for testing whether a graph has some property $\mathcal{P}$. When we run $\mathcal{A}$, it runs in $O(n+m)$ time and produces the correct answer (yes or no) with probability $5/9$.

1. If we run $\mathcal{A}$ k times, what is the expected number of times $\mathcal{A}$ produces the correct answer?
2. Give a tight upper bound on the probability that $\mathcal{A}$ produces the correct answer fewer than $k/2$ times.
3. Describe how, using $\mathcal{A}$ as a subroutine, we can get a Monte-Carlo algorithm that runs in $O(k(n+m))$ time and produces the correct answer with probability at least $1-e^{-\Omega(k)}$.

2.7 A Recursive Algorithm on Lists

Consider a subroutine that operates on a list. If we give the subroutine a list of length n then it runs in $O(n)$ time and, once it is complete, the input list has expected size $n/2$. We want to repeatedly call the subroutine until the list has size $O(1)$.

1. We say that a call to the subroutine is a *success* if it reduces the size of the list by a factor of at least $3/4$ ($|L'| \le 3|L|/4$). Give a lower bound on the probability that a particular call to the subroutine is a success.
2. Let X_k denote the number of successes that occur during a sequence of k consecutive calls. Give a lower bound on $\mathbf{E}[X_k]$.
3. Give an upper bound on the probability that the algorithm requires more than $c\log_{4/3} n$ calls to the subroutine.

2.8 An RFID Wakeup Protocol

The Radio Frequency Identification (RFID) wakeup protocol works something like this. An individual with a scanner walks into a room containing n RFID tags and presses the scan button. The scanner sends a signal to the RFID tags notifying them that they must identify themselves within t time units. Each tag u picks a random integer in $t_u \in \{0,\ldots,t-1\}$ and responds to the scanner at time t_u. If two or more tags respond at the same time then the scanner is unable to identify either of them, but can count the number of tags that responded.[1]

1. Consider a particular tag u. What is the probability that u's response is successful? I.e., what is the probability that u is the only tag that decided to respond at time t_u?
2. Suppose that the scanner (by initially scanning with the value $t=1$) knows the number of tags that have not responded yet and scans with the value $t=n$. Show that, the probability that a tag u is successful is at least $1/4$ (Hint: For $x \ge 2$, $(1-1/x)^x \ge 1/4$.)
3. Suppose that the scanner repeats the above experiment k times. Each time it resets the current value of t to be equal to the number of nodes that have not yet responded. Give an upper bound on the probability that a particular node u has not responded after all k iterations? What the expected number of nodes that have not responded after k iterations?
4. Set $k = c + \log_{4/3} n$ in the previous question and compute the expected number of nodes that have not responded after k iterations.

[1] This is a big assumption that we use to simplify the problem a bit. RFID scanners can't really do this.

2.9 Computing the OR of a Bit String

We have are given a bit-string $B_1, \ldots, B_n$ and we want to compute the OR of its bits, i.e., we want to compute $B_1 \vee B_2 \vee \cdots \vee B_n$. Suppose we use the following algorithm to do this:

```
1: for i ← 1, ..., n do
2:    if B_i = 1 then
3:       return 1
4: return 0
```

1. In the worst case, what is the number of times line 2 executes, i.e., how many bits must be inspected by the algorithm? Describe an inputs that achieve the worst case when the output is 0 and when the output is 1.

2. Consider the following modified algorithm:

```
Toss a coin c
if c comes up heads then
   for i ← 1, ..., n do
      if B_i = 1 then
         return 1
else
   for i ← n, ..., 1 do
      if B_i = 1 then
         return 1
return 0
```

   Assume that exactly one input bit $B_k = 1$. Then what is the expected number of input bits that the algorithm examines.

2.10 3-Way Partitioning

Suppose you are working on a system where two values can only be compared using the $<$ operator. (Sorting in Python is an example.) Here is an algorithm that, given an array $A[1], \ldots, A[n]$ and a value x classifies the elements of A as either less than, greater than or equal to x.

3-Way-Partition(A, x)

```
1: for i = 1 to n do
2:    if A[i] < x then
3:       add A[i] to S_<
4:    else if A[i] > x then
5:       add A[i] to S_>
6:    else
7:       add A[i] to S_=
```

1. Let $n_<$, $n_>$ and $n_=$ denote the number of elements of A that less than, greater than or equal to x, respectively. State the exact number of comparisons performed by 3-Way-Partition.

2. Show that there exists a randomized algorithm that uses only 1 random bit (coin toss) and performs and expected number of comparisons that is $2n_= + \frac{3}{2}(n_< + n_>)$.

2.11 Searching an Unsorted Array

Consider the problem of searching an unordered array $A[1], \ldots, A[n]$ for an element x. The only comparison operation you can do is to test if $x = A[i]$ for any $1 \leq i \leq n$. (Since the array is unordered this is the only useful kind of comparison anyway.)

1. Show that any deterministic algorithm must perform at least n comparisons in the worst case. I.e., show that, for any deterministic algorithm there exists an input $A[1], \ldots, A[n]$ for which the algorithm performs n comparisons.

2. Show that there exists a randomized algorithm that uses only 1 random bit and performs an expected number of comparisons that is at most $(n+1)/2$.

2.12 List Ranking by Independent Set Removal

We have n distinct points arranged on a circle and we perform the following experiment: We color each point black or white, independently, each with probability 1/2. We say that a point v is good if v is colored black and v's counterclockwise neighbour is colored white.

1. Let p be the probability that a point v is good. What is the value of p? What is the expected number of good points?

2. Suppose that we repeat this experiment k times and after each iteration we tag all the good points. What is the probability that a point v remains untagged after k iterations? What is the expected number of untagged points after k iterations?

3. What is the expected number of untagged points after $\log_{1/(1-p)} n + c$ iterations? Give an upper bound on the probability that, after $\log_{1/(1-p)} n + c$ iterations, there are any untagged points. (Hint: Don't use Chernoff's bounds)

2.13 Tightening Up Quicksort

Recall that the quicksort algorithm sorts an array of n elements by choosing a pivot p, partitioning the array so that the elements that are less than p appear before the elements that are greater than p and then recursing on both parts.

The running time of quicksort depends on the choice of p. In particular, it requires that p be not too close to the minimum or maximum value of the array. Suppose we choose p using the following *rejection algorithm*: We pick a random element p from the array and then, using linear time, we count the number of elements greater than p and the number of elements less than p. If either of these quantities is greater than $3n/4$ then we discard p and try again. Otherwise, we keep p and use it as our quicksort pivot.

1. Give a lower bound on the probability that the first p we pick is a good pivot (so that we keep p).

2. Give an upper bound on the expected number of choices of p we make before accepting one of our choices. Using this, give an upper bound on the expected running time of the algorithm for selecting the pivot p.

3. Give a recurrence that describes the expected running time of the quicksort algorithm with the above method of choosing the pivot. Show that this recurrence solves to $O(n \log n)$.

2.14 Approximating Max-2-Sat

A 2-CNF formula is the conjunction of a set clauses, where each clause is the disjunction of two (possibly negated, but distinct) variables. For example, the boolean formula

$$(a \vee b) \wedge (b \vee \neg d) \wedge (\neg a \vee c)$$

is a 2-CNF formula with 3 clauses. When we assign truth values to the variables (a, b, c and d above) we say that the assignment *satisfies* the formula if the formula evaluates to true. In general, it is not always possible to satisfy a 2-CNF-Formula, so we may try to satisfy most of the clauses.

1. Describe an analyze a very simple randomized algorithm that takes as input a 2-CNF formula with n clauses and ouputs a truth-assignment such that the expected number of clauses satisfied by the assignment is at $3n/4$. (Prove that the running time of your algorithm is small and that the expected number of clauses it satisfies is at least $3n/4$. You may assume that the variables are named $a_1, \ldots, a_m$, $m \le n$, so that you can associate truth values with variables by using an array of length m.)
2. Your algorithm implies something about all 2-CNF formulas having at most 3 clauses. What does it imply?
3. What does your algorithm guarantee for d-CNF formulas? (Where each clause contains d distinct variables.)

2.15 Ternary 0-1 Game Trees

Consider a 0-1 Game Tree where each node has degree 3. That is, the tree has $n = 3^{2k}$ leaves. Each leaf is labelled with 0 or 1 and the internal nodes alternate between AND and OR nodes at alternating levels. The goal is to evaluate the output of the root (AND) node.

1. Give the most efficient randomized algorithm for this problem you can come up with. State the algorithm clearly and analyze its expected running time.
2. Generalize your algorithm and its analysis to the case where each node has k children. (Hint: The solution to Question 2.9.3 might be helpful here.)

2.16 A Randomized Recurrence

Suppose we have a routine that takes as input a list of length n and outputs another list whose expected length is $n/2$ but is never more than $n-1$. The routine runs in $O(n)$ time.

1. Give an upper bound on the probability that the routine, given a list of length n output a list of length greather than $3n/4$.
2. Prove that if we recursively call the routine on the resulting list until we get a list of size 1 then the total cost of all recursive calls is $O(n)$. (Hint: You can use anything from the notes.)
3. Give an upper bound on the probability that the algorithm requires more than $c \ln n$ recursive calls.

2.17 Analysis of Random Binary Search

Consider the *random binary search* algorithm for finding an element x in a sorted array $A[1], \ldots, A[n]$:
RANDOM-BINARY-SEARCH(A, x)

```
1: ℓ ← 1
2: h ← n
3: while l < h do
4:    i ← RANDOM(ℓ, h)
5:    r ← compare(x, A[i])
6:    if r < 0 then
7:       h ← i − 1
8:    else if r > 0 then
9:       ℓ ← i + 1
10:   else
11:      ℓ ← h ← i
```

12: return l

RANDOM(l, h) is a procedure that returns a random number uniformly distributed in $\{l, l+1, l+2, \ldots, h\}$. The function **compare**(a, b) is a function that returns -1 if $a < b$, $+1$ if $a > b$ and 0 if $a = b$.

We will analyze this algorithm using the indicator variable

$$I_i = \begin{cases} 1 & \text{if } x \text{ is compared to } A[i] \\ 0 & \text{otherwise.} \end{cases}$$

1. Suppose $A = 1, \ldots, n$ and x is some integer $1 \leq x \leq n$. Given an exact formula in terms of x and i for $\Pr\{I_i = 1\}$ (which also happens to be $\mathbf{E}[I_i]$).
2. What is the expected number of comparisons performed while searching for x?
3. What is the value of x that maximizes the expected number of comparisons?

2.18 Analysis of Hoare's Find Algorithm

Hoare's FIND algorithm attempts to find the element of rank r in an array $A_1, \ldots, A_n$ of n elements. It does this by picking a random element p, partitioning A into the elements less than p, equal to p, and more than p and then recursively searching in one of the first or third subarray. Here's the pseudocode:

FIND$(A_1, \ldots, A_n, r)$

1: $p \leftarrow$ random element of $A_1, \ldots, A_n$
2: $j \leftarrow$ PARTITION$(A_1, \ldots, A_n, p)$ $\{p$ is now at position $j\}$
3: **if** $j > r$ **then**
4:     **return** FIND$(A_1, \ldots, A_{j-1}, r)$
5: **else if** $j < r$ **then**
6:     **return** FIND$(A_{j+1}, \ldots, A_n, r - j)$
7: **else**
8:     **return** j {found it!}

Suppose we run FIND on an array containing a permutation of $\{1, \ldots, n\}$ to find the element of rank r. Let $I_{i,j}$ $(i < j)$ be the indicator variable that is equal to 1 if and only if FIND compares i and j.

1. If $r \leq i < j$ then what is $\Pr\{I_{i,j} = 1\}$?
2. If $i < r < j$ then what is $\Pr\{I_{i,j} = 1\}$?
3. If $i < j \leq r$ then what is $\Pr\{I_{i,j} = 1\}$?

4. What is $\mathbf{E}[\sum_{i=r}^{n-1} \sum_{j=i+1}^{n} I_{i,j}]$? (Hint: On a piece of scrap paper, write the sum out longhand, using a new row for each new value of i.)

5. What is $\mathbf{E}[\sum_{i=1}^{r-1} \sum_{j=r+1}^{n} I_{i,j}]$?

6. What is $\mathbf{E}[\sum_{i=r-1}^{n-1} \sum_{j=i+1}^{r} I_{i,j}]$?

7. Give a tight upper bound on the expected number of comparisons done by FIND, as a function of r and n.

2.19 Finding the One

We are given a boolean array $B_1, \ldots, B_n$ and we know that there is exactly 1 location i such that $B_i = \text{true}$ and all the other locations are false. Our goal is to find location i.

In words, the following recursive algorithm tosses a coin and if it comes up heads the algorithm searches for i in the first half of the array and if it was unsuccessful then it looks for i in the second half of of the array. If the coin comes up tails it first searches in the second half of the array and then (if necessary) in the first half.

FIND-TRUE(ℓ, h)

1: **if** $\ell \geq h$ **then**
2: {* solve the base case *}
3: **if** $B_\ell = \text{true}$ **then**
4: return ℓ {* found it :) *}
5: **else**
6: return -1 {* didn't find it :(*}
7: $m \leftarrow i + \lfloor (h - \ell - 1)/2 \rfloor$
8: toss a coin c
9: **if** c is heads **then**
10: $i \leftarrow$ FIND-TRUE(ℓ, m) {* recurse on first half *}
11: **if** $i = -1$ **then**
12: $i \leftarrow$ FIND-TRUE($m + 1, h$){* recurse on second half *}
13: **else**
14: $i \leftarrow$ FIND-TRUE($m + 1, h$){* recurse on second half *}
15: **if** $i = -1$ **then**
16: $i \leftarrow$ FIND-TRUE(ℓ, m){* recurse on first half *}
17: return i

1. If $B_\ell, \ldots, B_h$ does not contain any true value then how many comparisons (line 3) will get performed by a call to FIND-TRUE(ℓ, h).

2. Suppose n is a power of 2 and we call FIND-TRUE($1, n$). Give the recurrence that describes the expected number of comparisons performed. Give the exact solution of the recurrence by expanding it into a familiar-looking sum. (Hint: If it gets very messy then you've gotten the recurrence wrong.)

3. Suppose n is a power of 2 and we call FIND-TRUE($1, n$). Derive the probability that the algorithm finds i using exactly 1 comparison in line 3.

2.20 Matrix Equality Testing

Let $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ be two real-valued vectors of length n and let $r = (r_1, \ldots, r_n)$ be a random binary vector of length n. That is, the r_i's are chosen independently and uniformly at random to be either 0 or 1. For a vector x, let $r \cdot x$ denote the sum $r_1 x_1 + r_2 x_2 + \cdots + r_n x_n$.

1. It is clear that, if $a = b$ then $r \cdot a = r \cdot b$. Show that if $a \neq b$ then $\Pr\{r \cdot a = r \cdot b\} \leq 1/2$. (Hint: Consider what has to happen at a particular index i such that $a_i \neq b_i$. What is the probability that this happens?)

2. Using the above fact (whether you could provide a proof or not), show that for two $n \times n$ matrices A and B that are not equal, $\Pr\{r \times A = r \times B\} \leq 1/2$. (Note that computing $r \times A$ is an n vector that takes $O(n^2)$ time to compute.)

3. Let A, B and C be three $n \times n$ matrices. Describe an algorithm that runs in $O(n^2)$ time and

   (a) If $A = B \times C$ then the algorithm always outputs yes.

   (b) If $A \neq B \times C$ then the algorithm will output no with probability at least $1/2$.

   (Hint: Matrix multiplication, i.e., computing $B \times C$ explicitly takes too long. You'll have to find some other way.)

2.21 A Monte-Carlo Min-Tricut Algorithm

A *tricut* of an undirected graph $G = (V, E)$ is a subset of E whose removal separates G into at least 3 connected components. A min-tricut of G is a tricut of minimum size (over all possible tricuts of G). This question studies a problem of computing the min-tricut.

1. Let C be a min-tricut of G. Prove the best upper bound you can on the size of C in terms of $|V|$ and $|E|$. (Hint one possible tricut can be obtained by separating two vertices from the rest of G by deleting all their incident edges.)

2. If we pick a random edge $e \in E$ give an upper bound on the probability that $e \in C$.

3. Suppose we repeat the following $|V| - 4$ times: Select a random edge e of G, contract e (identify the two endpoints of e) and eliminate any loops (edges with both endpoints at the same vertex). Give a lower bound on the probability that all $n - 4$ edge contractions avoid the edges of C.

4. In a graph with 4 vertices, give an upper bound on the probability that a randomly selected edge is part of the Min-Tricut. (Hint: Your bound in part 1 may not be strong enough to give a non-trivial upper bound. You will really have to see what a tricut in a graph with 4 vertices looks like.)

5. The previous question gives a lower bound on the probability that a monte-carlo algorithm finds the min-tricut C. Unfortunately, the probability is very small. How many times would we have to run the algorithm so that the probability of finding C is at least

   (a) $1 - 1/e$

   (b) $1 - 1/1\,000$

   (c) $1 - 1/1\,000\,000\,000$

2.22 The height of a skiplist

Suppose we start with a list $L_0 = l_1, \ldots, l_n$. We obtain a new list L_1 by tossing a fair coin for each element l_i and adding l_i to L_1 iff the coin toss comes up heads.

1. What is the probability that l_i is in L_1? From this, compute the expected size of L_1.

2. Suppose we continue in this manner to obtain a list L_2 by tossing coins for each element of L_1. In general, to obtain L_i ($i > 0$), we toss a coin for each element in L_{i-1} and add that element to L_i iff the coin toss comes up heads.

   What is the probability that any particular element l_j is in L_i? From this, compute the expected size of L_i.

3. Show that the expected time required to build all the lists $L_1, L_2, L_3, \ldots$ is $O(n)$.

4. Define the indicator variable $I_i = 1$ if L_i is non-empty and $L_i = 0$ if L_i is empty. Observe that I_i never exceeds the size of L_i. The random variable $X = \sum_{i=0}^{\infty} I_i$ is the *height* of the skip list. Show that $\mathbf{E}[X] = O(\log n)$.

Chapter 3

Applications of Chernoff's Bounds

3.1 Monte-Carlo Landslide Finding

We are given an array $A_1, \ldots, A_n$ and we are told that some element x occurs $2n/3$ times in the array, but we are not told the value of x. Our goal is to use a fast Monte-Carlo algorithm (that may report the incorrect value) to find x.

1. Describe a constant-time Monte-Carlo algorithm to find x that is correct with probability $2/3$.
2. Suppose we sample k elements at random (with replacement) from the array to obtain k sample values $S_1, \ldots, S_k$. Give a good upper-bound on the probability that x occurs less than $(1-\epsilon)2k/3$ times in this sample.
3. Give a good upper bound on the probability that x occurs less than $k/2$ time in the sample. (Hint: This is the same as the previous question except we are using a specific value of ϵ.)
4. Describe a Monte-Carlo algorithm that runs in $O(k)$ time and reports x with probability at least $1 - 1/e^{\Omega(k)}$. (Just describe the algorithm. The error probability follows from the previous questions.)

3.2 Estimating π

Let π be the area of a circle whose radius is 1. For $i = 1, \ldots, n$, let (X_i, Y_i) be a random point in the unit square, that is X_i and Y_i are uniformly and independently drawn from the real interval $[0, 1]$.

1. Define the random variable
$$I_i = \begin{cases} 1 & \text{if } X_i^2 + Y_i^2 \leq 1 \\ 0 & \text{otherwise.} \end{cases}$$
Argue that $E[I_i] = \pi/4$.
2. Recall that an *IEEE 754 double precision* floating point number has only 53 bits of precision so, even though, $\pi/4$ is an irrational number, there a rational number $\mathrm{rep}(\pi/4)$ that corresponds to the IEEE 754 representation of $\pi/4$. In fact, there is a whole interval $[(1-\epsilon)\mathrm{rep}(\pi/4), (1+\epsilon)\mathrm{rep}(\pi/4)]$, where $\epsilon = 2^{-53}$ that all have the same IEEE 754 representation as $\pi/4$.

   Let $X = \sum_{i=1}^{n} I_i$. Give a lower bound on the probability that $\mathrm{rep}(X) = \mathrm{rep}(\pi)$, i.e., that $X \in [(1-\epsilon)\mathrm{rep}(\pi/4), (1+\epsilon)\mathrm{rep}(\pi/4)]$.
3. For what values of n is the probability from the previous question at least
   (a) 1/2?

(b) 999/1000?

(c) 999999/1000000?

3.3 An Independent Set of Friends

Suppose a group of n (n is an even integer) friends $f_0, \ldots, f_{n-1}$ stand around in a circle. Each friend tosses a coin. If f_i's coin comes up heads then they keep it. If it comes up tails then they give it to $f_{(i+1) \bmod n}$.

1. What is the expected number of friends who now have 2 coins?
2. Let B_i be the event "f_i has 2 coins after playing the game." What $\Pr\{B_i | B_{i+1}\}$?
3. What is $\Pr\{B_i | B_{i+2}\}$?
4. Prove that the probability that there are fewer than $(1-\epsilon)n/8$ friends with 2 coins is very small.
5. Prove that the probability that there are fewer than $(1-\epsilon)n/4$ friends with 2 coins is also very small. (Hint: Use Boole's inequality.)

3.4 Coarse-grained parallel sorting

Let $a_1, \ldots, a_n$ be a set of real numbers such that $a_1 < a_2 < \cdots < a_n$. We create a set of *samples* $s_1, \ldots, s_k$ ($s_1 < s_2 < \cdots < s_k$) by adding each a_i to the sample set with probability $1/p$, and all these choices are independent. Here, p is much smaller than n, so that $p \le \sqrt{n}$.

1. Let k be the number of samples obtained. What is the expected value of k?
2. Use Chernoff's bounds to prove that, for any $0 < \epsilon < 1$,

$$\Pr\{k > (1+\epsilon)n/p\} \le e^{-\Omega(n/p)}$$

and

$$\Pr\{k < (1-\epsilon)n/p\} \le e^{-\Omega(n/p)}$$

3. Use Chernoff's bounds to give an upper bound on

$$\Pr\{a_1, \ldots, a_{(1+\epsilon)n/p} \text{ contains less than } n/p^2 \text{ samples}\}$$

4. Suppose we take the sample elements $s_{n/p^2}, s_{2n/p^2}, s_{3n/p^2}, \ldots, s_{(p-1)n/p^2}$ and use these to partition A in to p groups, where the ith group ($1 < i < p$) consists of all elements a_i such that $s_{in/p^2} \le a_i < s_{(i+1)n/2}$. The first and last groups contain all elements less than s_{n/p^2} and all elements greater than or equal to $s_{(p-1)n/p^2}$, respectively. The same analysis used in the previous question shows that, for any $1 \le i \le p$, and any $0 < \epsilon < 1$,

$$Pr\{\text{the } i\text{th group contains more than } (1+\epsilon)n/p \text{ elements}\} \le e^{-\Omega(n/p)} \quad .$$

Use Boole's inequality (from the first lecture) to prove that

$$Pr\{\textit{any} \text{ group contains more than } (1+\epsilon)n/p \text{ elements}\} \le pe^{-\Omega(n/p)} \quad .$$

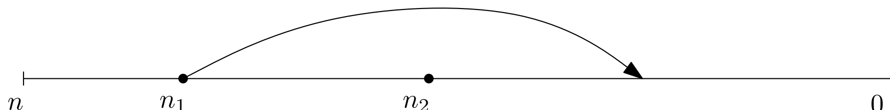
3.5 A Random Walk on a Grid

Imagine the following random random walk on the integer grid. You begin by standing on the Euclidean plane at the origin $(0, 0)$ and then repeat the following n times: Toss a 4 sided die and depending on the result you either take a step north ($y \leftarrow y + 1$), east ($x \leftarrow x + 1$), south ($y \leftarrow y - 1$), or west ($x \leftarrow x - 1$).

1. At the end of this process, what is your expected x coordinate and your expected y coordinate?
2. Argue that, with high probability (for reasonable values of ϵ), the number of steps you take in any particular direction (to the west, say) is in the interval $[(1 - \epsilon)n/4, (1 + \epsilon)n/4]$.
3. Argue that, with high probability, your random walk finishes inside a square of side length $2\epsilon n/4$ centered at the origin.
4. What is the smallest value of ϵ for which the above argument gives a meaningful result. What is the result?

3.6 Kick the Stones

Consider the following game, whose analysis comes up in the analysis of a particular graph construction algorithm. We are standing on the line at location n and walking towards location 0. Also on the line are two stones that are somewhere between us and our destination, both on integer values. Whenever we reach a stone at location x we kick the stone and the stone lands at a random integer location in $\{0, \dots, x - 1\}$. For this question we are interested in how many times we have to kick the stones before they are both at location 0.



1. Consider two consecutive kicks. Before these two kicks the stones are at locations n_1 and n_2 and after these two kicks they are at locations n'_1 and n'_2 (possibly $n'_1 = n_1$ or $n'_2 = n_2$). We say that these two kicks were *successful* if $\max\{n'_1, n'_2\} \leq \max\{n_1, n_2\}/2$. Prove that, no matter what the values of n_1 and n_2 the probability that a pair of consecutive kicks are sucessful is at least $1/4$.
2. Consider the entire sequence of kicks that occurs during our walk and pair off consecutive kicks by making pairs of the first and second, third and fourth, and so on. Some of these pairs are successful and some are not. Prove that the number of successful pairs is at most $c \log_2 n$ for some constant c. Be sure to specify the value of the constant c.
3. Give a tight upper bound on the probability that more than $(1 + \epsilon)8c \log_2 n$ kicks are required to move both stones to location 0.

3.7 Finding large independent sets in parallel

Let $L = l_0, \dots, l_{n+1}$ be a list of items. We find an independent set in L using the following algorithm: For each element l_i ($1 \leq i \leq n$), we toss a fair coin. We say that l_i is included in the independent set if l_i's coin toss came up heads and both its neighbours' (l_{i-1} and l_{i+1}) coin tosses came up tails.

1. What is the probability that l_i is included in the independent set. Using this, compute the expected size of the independent set.
2. Let E_i be the event "l_i is included in the independent set." Are E_i and E_{i+1} independent?

3. Are E_i and E_{i+2} independent?
4. Are E_i and E_{i+3} independent?
5. Use Chernoff's bounds to show that, with very high probability, this algorithm produces an independent set of size at least cn for some constant $c > 0$.

3.8 McDiarmid's Inequality

Chernoff's bounds is only one of many *concentration inequalities* that probability theory offers to us. In this question we explore an extremely powerful and general inequality due to McDiarmid.

Theorem 1 (McDiarmid's Inequality)**.** *Let A be some set of values and let $f : A^n \to \mathbb{R}$ be a function that satisfies*

$$|f(x_1, \ldots, x_n) - f(x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, \ldots, x_n)| \le c_i$$

for all $x_1, \ldots, x_n, x'_i \in A^{n+1}$ and all $1 \le i \le n$. Then, if $X_1, \ldots, X_n$ are independent random variables that only take on values in A then

$$\Pr\{|f(X_1, \ldots, X_n) - \mathbf{E}[f(X_1, \ldots, X_n)]| \ge t\} \le \frac{2}{e^{2t^2/\sum_{i=1}^n c_i^2}}$$

In words, McDiarmid's Inquality says that if we have a function f that doesn't change too much if we only change one of f's inputs then $f(X_1, \ldots, X_n)$ is stronly concentrated around its expected value.

1. A Bernoulli(p) random variable takes on values in the set $A = \{0, 1\}$. If $X_1, \ldots, X_n$ are independent Bernoulli(p) random variables and $f(x_1, \ldots, x_n) = \sum_{i=1}^n x_i$ then what does McDiarmid's Inequality tell us about $f(X_1, \ldots, X_n)$? Does this remind you of anything?
2. Let $X_1, \ldots, X_n$ be independent random variables that are uniformly distributed in the unit interval $A = [0, 1]$. Let $f(x_1, \ldots, x_n)$ be the function that counts the number of inversions in $x_1, \ldots, x_n$ (an inversion is a pair (x_i, x_j) with $i < j$ and $x_i > x_j$. What is $\mathbf{E}[f(X_1, \ldots, X_n)]$?
3. Using the same setup as the previous question. If we change one value x_i to x'_i, what is the maximum value of $|f(x_1, \ldots, x_n) - f(x_1, \ldots, x'_i, \ldots, x_n)|$.
4. What does McDiarmid's inequality tell us about $\Pr\{|f(X_1, \ldots, X_n) - \binom{n}{2}/2| \ge \epsilon n^2\}$
5. Suppose we run the insertion sort algorithm on $X_1, \ldots, X_n$ independently and uniformly distributed in $[0, 1]$. Then what does the above imply about (a) the number of swaps performed and (b) the number of comparisons performed.

Chapter 4

Amortized Analysis

4.1 Stacks with Multipops

Consider an algorithm that works with a list L and runs in m rounds. During round i, the algorithm either appends one element to L (at a cost of $C_i = O(1)$) or deletes some number, $0 \leq k_i \leq |L|$, of elements from L (at a cost of $C_i = O(1 + k_i)$).

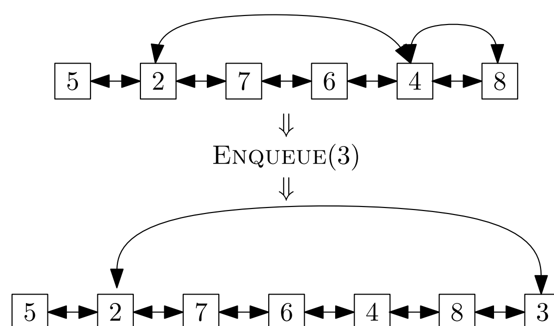
1. Define a non-negative potential function $\Phi(L)$ and use it show that the amortized cost of the ith round is $O(1)$.

4.2 A Min Queue

1. Enqueue(x): Add x to the end of the queue
2. Dequeue(): Remove an element from the front of the queue.
3. FindMin(): Return a pointer to the smallest element currently in the queue.

One way to implement a MinQueue is as 2 lists. The first list, L_1, contains the elements in the queue, in the order in which they have been enqueued. That is, we enqueue to the tail of L_1 and we dequeue from the front of L_1. The second list is defined recursively: The first element of L_2 is the smallest value in L_1 and every x element in L_2 has as it's successor the smallest element in L_1 that was enqueued after x.

To Dequeue an element we simply delete it from L_1 and, if necessary, delete it from L_2. To execute a FindMin operation we simply report the first element of L_2. To Enqueue an item x we append x to L_1 and delete items from the tail of L_2 until we reach an item whose value is smaller than x, at which point we append x to L_2. See the example below:



1. Define a function $\Phi(L_1, L_2)$ that will allow you to prove that the amortized cost of any operation is $O(1)$. Your potential function should be always non-negative and should be equal to 0 if the queue is empty.

2. Using your potential function, show that the amortized cost of FINDMIN is $O(1)$

3. Using your potential function, show that the amortized cost of DEQUEUE is $O(1)$

4. Using your potential function, show that the amortized cost of ENQUEUE is $O(1)$.

4.3 Dynamically Growing and Shrinking Arrays

Consider the following memory-efficient implementation of a stack as an array A. Initially, we allocate an array A of size 1. Let n denote the size (number of elements on the stack) at some point in time.

If the user pushes an element on to the stack but the array is already full (because it has size n, i.e, $|A| = n$) then we allocate a new array A' of size $\lceil 3n/2 \rceil$, copy A into A', and set $A \leftarrow A'$.

Conversely, if the user pops an element from the stack, but the stack has size $2n$ ($|A| = 2n$) then we allocate a new array A' of size $\lceil 3n/2 \rceil$, copy the first n elements of A into A' and then set $A \leftarrow A'$.

1. Define a non-negative potential function, $\Phi(n, |A|)$, that will allow you to prove that both push and pop take $O(1)$ amortized time. (Hint: Having $|A| = 3n/2$ is good, but having $|A| = n$ or $|A| = 2n$ is bad.)

2. Using your potential function, show that push takes $O(1)$ amortized time.

3. Using your potential function, show that pop takes $O(1)$ amortized time.

4.4 Redundant Binary Counters

A *redundant binary counter* consists of k values $b_{k-1}, \ldots, b_0$. Each b_i can be equal to either 0, 1, or 2. The *value* of a redundant binary counter is $\sum_{i=0}^{k-1} 2^i b_i$. Notice that a particular value does not have a unique representation since, for example, 2 could be represented as 10 or 02. The operations we perform on redundant binary counters are are INCREMENT and DECREMENT where we increase, respectively, decrease the value of the counter by 1.

To INCREMENT a redundant binary counter we find the smallest index i such that $b_i \neq 2$. We then set each of $b_0, \ldots, b_{i-1}$ to 1 and increse the value of b_i by 1. For example

$$100122222 + 1 = 100211111$$

To DECREMENT a redundant binary counter whose value is greater than 0 we find the smallest index i such that $b_i \neq 0$, we set $b_0, \ldots, b_{i-1}$ to 1 and we decrease b_i by 1. For example

$$12100000 - 1 = 12011111$$

In this exercise we want to prove that the amortized cost of INCREMENT and DECREMENT operations is $O(1)$.

1. Define a potential function $\Phi(b_{k-1}, \ldots, b_0)$ that will allow us to prove that increment and decrement each have an amortized cost of $O(1)$. Hint: your function can be obtained by summing the potential of the individual bits.

2. Using your potential function, prove that the amortized cost of IN CREMENT is $O(1)$

3. Using your potential function, prove that the amortized cost of DECREMENT is $O(1)$

4. Show that normal binary counters (where the bits are equal to 0 or 1) do not have this property. That is, describe a sequence of m increment and decrement operations that take $\Omega(m \log m)$ time to execute using a normal binary counter.

4.5 Data Structures for Disjoint Sets

It may be helpful to look at Section 21.2 of CLRS (Section 22.2 of CLR) before you do this exercise. A *disjoint set* data structure maintains a collection of sets of elements so that, for any pair of elements we can test if they belong to the same set (FIND) and, at any time, we can merge two sets (UNION).

The simplest such data structure is to maintain each set as a list and have each element x maintain a pointer, parent(x), to the first element of its list. Then determining if two elements x and y belong to the same list can be done in $O(1)$ time by checking if $\text{parent}(x) = \text{parent}(y)$. To merge two sets we take the smaller of the two sets and add each of its elements to the larger of the two sets by appending it to the list and resetting it's parent value. Note that this takes time linear in the size of the smaller of the two sets.

We would like to prove that, if we start with a collection of n singleton (size 1) sets and perform a sequence of n UNION (merge) operations the total cost of all these operations is $O(n \log n)$. That is, we want to show that the amortized cost of each UNION operation is $O(\log n)$.

1. Define a potential function $\Phi(x)$ on individual elements that is $c \log_2 n$ if x belongs to a set of size 1 and is O if x belongs to a set of size n.

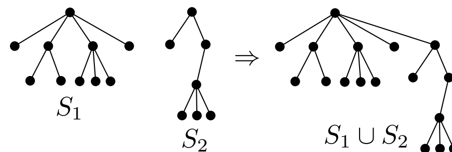
2. Using this potential function, show that the amortized cost of a merge operation is at most 0.

3. Explain why this means that a sequence of n merges costs $O(n \log n)$ and not 0.

4.6 Tree Data Structures for Disjoint Sets

A *disjoint set* data structure maintains a collection of sets of elements so that, for any pair of elements we can test if they belong to the same set (FIND) and, at any time, we can merge two sets (UNION).

One such data structure is to maintain each set as a tree. Each element of the set maintains a pointer parent(x) to its parent in the tree. The root, r, of the tree has $\text{parent}(r) = r$. To merge two sets S_1 and S_2, we take the smaller of the two trees and have the parent of its root become the root of the other tree. See the figure below:



To perform a FIND operation, for two nodes x and y we follow the parent pointers for x and y until we reach the root(s) of the their respective trees. If we reach the same root we conclude that they belong to the same set, otherwise we conclude that they belong to different sets. To make searching for x and y faster the next time we reset the parent pointers of x and y so that they become children of the root(s) of their respective subtrees.

1. For a node x, define the potential of x as $\Phi(x) = c_1 d(x) + c_2(\log n - \log_2 s(x))$, where $s(x)$ is the size of the set to which x belongs and $d(x)$ is the distance from x to the root of its tree. Define the potential of the data structure as the sum of the potentials over all nodes. Using this potential function, show that the amortized cost of a MERGE operation is zero (0) and the amortized cost of a FIND operation is $O(1)$.

2. Explain why this means that a sequence of m FIND and m MERGE operations takes $O(m + n \log n)$ time.

3. Explain why this data structure might be more efficient than the data structure based on lists that was in the problem set.

4.7 Lazy Deletion

Suppose a student has implemented a balanced binary search tree (e.g., AVL-tree, red-black tree, etc.) that performs insertion and search operations in $O(\log n)$ time, but was too lazy to implement deletion. Instead, they have implemented a *lazy* deletion mechanism: To delete an item, we search for the node that contains it (in $O(\log n)$ time) and then *mark* that node as deleted. When the number of marked nodes exceeds the number of unmarked nodes (during a deletion) the entire tree is rebuilt (in $O(n)$ time) so that it contains only unmarked (i.e., undeleted) nodes.

1. Define a non-negative potential function and use it to show that the amortized cost of deletion is $O(\log n)$.
2. How does your potential function affect the amortized cost of insertion?
3. Suppose that, during a rebuilding step, we only remove half the marked nodes (still at a cost of $O(n)$. Show that the amortized cost of deletion is still $O(\log n)$.

4.8 Lazy Insertion Data Structures

Suppose we have a static data structure for some search problem. Given n elements, we can build a data structure in $O(n \log n)$ time that answers queries in $O(\log n)$ time. We convert this into an insertion-only data structure as follows: We maintain two static data structures, D_1 has size at most $\sqrt{n}$ and D_2 has size at most n. To insert a new element we first check if the number of elements in D_1 is less than $\sqrt{n}$. If so, we add the newly inserted element to D_1 and rebuild D_1 at a cost of $O(\sqrt{n} \log n)$. Otherwise (there are too many elements in D_1) we take all the elements of D_1, move them to D_2 and rebuild D_2 at a cost of $O(n \log n)$. To search for an element, we search for it in both D_1 and D_2 at a cost of $O(\log n + \log \sqrt{n}) = O(\log n)$.

1. Define a potential function on D_1 and D_2 to show that the amortized cost of insertion is $O(\sqrt{n} \log n)$.
2. Show that during the second case of the insertion procedure, even if we only insert half the elements of D_1 into D_2, the amortized cost of insertion is still only $O(\sqrt{n} \log n)$.
3. Suppose we generalize this data structure so that we maintain d static data structures $D_1, \ldots, D_d$ where D_i has maximum size $n^{i/d}$. Whenever D_i becomes full we empty it and put all its elements in D_{i+1}.

   Define a potential function on $D_1, \ldots, D_d$ to show that the amortized cost of insertion is $O(n^{1/d} \log n)$.

4.9 List-Based Priority Queues

In this question, we investigate an implementation of priority queues based on a collection of sorted lists. In this implementation we store $O(\log n)$ sorted lists $L_0, \ldots, L_k$, where the list L_i has size at most 2^i. To find the minimum element, we simply look at the first element of each list (remember, they are sorted) and report the minimum, so the operation FindMin takes $O(\log n)$ time.

1. To do an insertion, we find the smallest value of i such that L_i is empty, merge the new element as well as $L_0, \ldots, L_{i-1}$ into a single list and make that list be L_i. At the same time, we make $L_0, \ldots, L_{i-1}$ be empty.

   Prove, by induction on i, that the list L_i has size at most 2^i.
2. Show how to merge $L_0, \ldots, L_{i-1}$ so that the cost of this merging (and hence the insertion) is $O(2^i)$.
3. Starting with an empty priority queue and then performing a sequence of n insertions, how many times does list L_i go from being empty to being non-empty.

4. Using your answer from the previous question, what is the total running time of a sequence of n insertions beginning with an empty priority queue?
5. As this data structure evolves, the elements move to lists with larger and larger indices. Define a non-negative potential on the element x so that when x is in L_0 its potential is $\log n$ and when x is in $L_{\log n}$, its potential is 0.
6. Define a non-negative potential function on this data structure so that, when we build the list L_i, the potential decreases by at least $c2^i$, for some constant c.
7. Show that the amortized cost of insertion (using your potential function from the previous question) is $O(\log n)$.

4.10 Jeff Erickson's Countdown Trees

In this question, we investigate an implementation of balanced binary search trees based on timers. Like the search trees talked about in class, we periodically rebuild entire subtrees into perfectly balanced binary search trees. However, we don't explicitly maintain the sizes of subtrees. Instead, each node in our tree has a *timer* and whenever the node u is part of a subtree that is rebuilt, we set $\text{timer}(u) = \frac{1}{3}\text{size}(u)$. Whenever the size of u's subtree changes (because of an insertion or deletion), we decrement $\text{timer}(u)$. When $\text{timer}(u)$ is less than or equal to 0, we rebuild the subtree rooted at u into a perfectly balanced binary search tree.

1. We claim that the search trees built this way are weight-balanced. Argue that there exists some constant $\alpha < 1$, such that for any node v except the root, $\text{size}(v) \leq \alpha \times \text{size}(\text{parent}(v))$. (Hint: When $\text{parent}(v)$ was last rebuilt, $\text{size}(v) \leq \frac{1}{2} \times \text{size}(\text{parent}(v))$. How much damage could have happened since then?)
2. Use your answer from the previous question to show that countdown trees have height $O(\log n)$.
3. The cost of rebuilding the subtree rooted at $\text{parent}(v)$ is $O(\text{size}(\text{parent}(v)))$. Define a non-negative potential function on the tree so that rebuilding $\text{parent}(v)$ results in the potential decreasing by $c \times \text{size}(\text{parent}(v))$, for some constant c.
4. Show that the amortized cost of searching in a countdown tree is $O(\log n)$
5. Show that the amortized cost of insertion in a countdown tree is $O(\log n)$.
6. Show that the amortized cost of deleting a leaf in a countdown tree is $O(\log n)$.

Chapter 5

NP-Hard Problems and Algorithms

5.1 NP-Completeness of Partition

The goal of this question is to walk you through an NP-completeness proof for the PARTITION problem. An instance of PARTITION is a multiset X of non-negative integers. The goal is to decide if you can split the set into two parts whose sum is equal, i.e., is there a subset $X' \subseteq X$ such that

$$\sum_{x \in X'} x = \frac{1}{2} \sum_{x \in X} x \ ?$$

1. Recall that to prove a problem is NP-complete, we have to prove that (1) it is in NP and (2) that it is NP-hard. Prove that PARTITION is in NP.
2. PARTITION looks a lot like SUBSET-SUM, which we already know is NP-hard. In fact, it is a special case of SUBSET-SUM where the target value $t = \frac{1}{2}\sum_{x \in S} x$. This doesn't immediately mean that PARTITION is NP-hard since it might a special case of SUBSET-SUM that is easy to solve.

   Show that SUBSET-SUM $\leq_P$ PARTITION, thereby proving that PARTITION is NP-hard. Remember that there are three things to you have to do:

   Give a polynomial time algorithm that takes an instance (S, t) of SUBSET-SUM and produces an instance (X) of PARTITION whose size is polynomial in the size of the SUBSET-SUM instance.
3. Now you need to show that the answer to the SUBSET-SUM instance (S, t) is true if and only if the answer to the PARTITION instance (X) is true. We'll do this in two parts:

   Show that if S is contains a subset whose sum is t then X contains a subset whose sum is half the sum of the elements in X.
4. Show that if X contains a subset whose sum is half the sum of the elements in X then S constains a subset whose sum is t.

5.2 NP-Completeness of $(\frac{1}{3}, \frac{2}{3})$-Partition

The goal of this question is to walk you through an NP-completeness proof for the $(\frac{1}{3}, \frac{2}{3})$-PARTITION problem. An instance of $(\frac{1}{3}, \frac{2}{3})$-PARTITION is a set X of non-negative integers. The goal is to decide if you can split the set into two parts, one of whose sum is three times the sum of the other. In other words, we ask for a subset $X' \subseteq X$ such that

$$\sum_{x \in X'} x = \frac{1}{4} \sum_{x \in X} x \ .$$

1. Recall that to prove a problem is NP-complete, we have to prove that (1) it is in NP and (2) that it is NP-hard. Prove that $(\frac{1}{3}, \frac{2}{3})$-Partition is in NP.

2. $(\frac{1}{3}, \frac{2}{3})$-Partition looks a lot like Subset-Sum, which we already know is NP-hard. In fact, it is a special case of Subset-Sum where the target value $t = \frac{1}{3}\sum_{x \in S} x$. This doesn't immediately mean that $(\frac{1}{3}, \frac{2}{3})$-Partition is NP-hard since it might a special case of Subset-Sum that is easy to solve.

   Show that Subset-Sum $\leq_P (\frac{1}{3}, \frac{2}{3})$-Partition, thereby proving that $(\frac{1}{3}, \frac{2}{3})$-Partition is NP-hard. Remember that there are three things to you have to do:

   Give a polynomial time algorithm that takes an instance (S, t) of Subset-Sum and produces an instance (X) of $(\frac{1}{3}, \frac{2}{3})$-Partition whose size is polynomial in the size of the Subset-Sum instance.

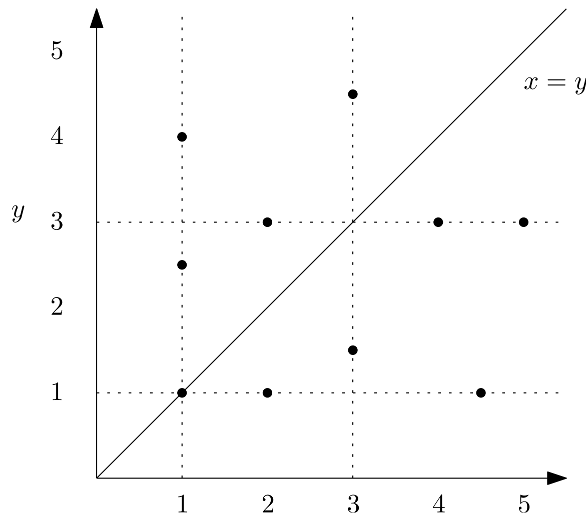
3. Now you need to show that the answer to the Subset-Sum instance (S, t) is true if and only if the answer to the $(\frac{1}{3}, \frac{2}{3})$-Partitioninstance (X) is true. We'll do this in two parts:

   Show that if S contains a subset whose sum is t then X contains a subset whose sum is one quarter the sum of the elements in X.

4. Show that if X contains a subset whose sum is one quarter the sum of the elements in X then S constains a subset whose sum is t.

5.3 NP-Completeness of Cross-Cover

The Cross-Cover problem takes as input a set of points in the plane and an integer k and asks: Does there exist a set of k crosses such that every point is contained in at least 1 cross. A *cross* at location x is the union of a horizontal and vertical line that pass through the point (x, x). For example, the set of points shown below is covered by $k = 2$ crosses at locations 1 and 3.



1. Prove that Cross-Cover is NP-complete. Don't forget to include all the steps from the previous question. Use the next page if you have to. (Hint: Reduce from Vertex-Cover; the edges in your Vertex-Cover instance will become points in your HV-Line-Cover instance.)

5.4 NP-Completeness of Light-Switching

The Light-Switching problem is the following: We are given an $m \times n$ matrix M where each entry $M_{i,j}$ is either 0, +1, or −1. We are asked whether it is possible to negate a subset of the rows of M so that every column of M contains at least one +1 entry. For example in the figure below if we negate the first and third rows of the matrix on the left hand side we get the matrix on the right hand side.

$$\begin{array}{c} - \\ \\ - \\ \\ \end{array} \begin{bmatrix} +1 & -1 & +1 & +1 \\ +1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & +1 \end{bmatrix} \Rightarrow \begin{bmatrix} -1 & +1 & -1 & -1 \\ +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & +1 \end{bmatrix}$$

1. Prove that Light-Switching is NP-complete. Don't forget to include all the steps from the previous question. Use the next page if you have to. (Hint: The easiest thing to do is show that 3-Sat $\leq_P$ Light-Switching.)

5.5 Converting Decision Algorithms to Optimization Algorithms

1. Suppose we have an efficient algorithm $\mathcal{A}$ for solving the Vertex-Cover problem. In other words, given a graph G, $\mathcal{A}$ will tell us: Does G have a vertex cover of size k? Show how to use $\mathcal{A}$ to give an efficient algorithm for the Min-Vertex-Cover problem: Given a graph G what is the size of the smallest vertex cover of G. For full marks, your algorithm should be as efficient as possible. Express the running time as a function of the running time $T(n)$ of the Vertex-Cover algorithm.

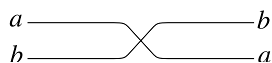
5.6 NP-hardness of Planar-Circuit-Sat

The problem Circuit-Sat is the first problem Cooke showed to be NP-hard. In this exercise we will show that Circuit-Sat remains NP-hard even if the circuit does not have any wires that cross.

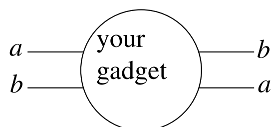
1. Suppose that we have circuits that can use the XOR operator, defined as

$$x \oplus y = \begin{cases} \text{true} & \text{if exactly 1 of } x \text{ and } y \text{ is true} \\ \text{false} & \text{otherwise.} \end{cases}$$

   Show how we can use XOR gates to replace pairs of crossing wires with gadgets that don't have crossing wires. I.e., design a gadget with no crossing wires to replace:



   with



2. Now design a gadget to replace an XOR gate with a circuit made up of AND, OR and NOT gates having no crossing wires.

   Note: In doing this, you have proven that any circuit using AND, OR and NOT gates can be replaced by an equivalent circuit using AND, OR and NOT gates that doesn't have any crossing wires (this is called a *planar* circuit). With only a bit more work, you could show that this can be done in polynomial time and this would prove that PLANAR-CIRCUIT-SAT is NP-complete. (This is for your information only, it's not required for the assignment.)

5.7 NAND-Sat is NP-hard

1. In class we showed that SAT (formula satisfiability) was NP-hard. Recall that SAT formulas are made up of NOTs, ANDs, and ORs.

   Consider formulas that are made up only of NAND operators, where the NAND operator is defined by

   $$x \,\mathrm{NAND}\, y = \neg(x \wedge y) \ .$$

   The NAND-SAT problem is the problem of determining if a formula made up of variables combined using the NAND operator is satisfiable. Prove that NAND-SAT is NP-hard by showing that SAT $\leq_P$ NAND-SAT (Hint: you only need to show how each of the operators in a SAT formula (AND, OR, NOT) can be implemented using only the NAND operator).

5.8 NP-Hardness of Independent-Set

An *independent set* in a graph G is a set of vertices of G, no two of which are adjacent. The INDEPENDENT-SET problem takes as input a graph G and an integer k and asks whether G has an independent set of size k. Prove that INDEPENDENT-SET is NP-Complete.

5.9 NP-Hardness of Subgraph-Isomorphism

The SUBGRAPH-ISOMORPHISM problem takes as input two graphs G_1 and G_2 and asks whether G_1 contains a subgraph that is isomorphic to G_2. (Can we map the vertices of G_2 onto a subset of the vertices of G_1 so that two vertices of G_2 are adjacent if and only if the two corresponding vertices in G_1 are adjacent?) Prove that SUBGRAPH-ISOMORPHISM is NP-hard.

5.10 NP-Hardness of Set-Cover

An instance of SET-COVER is a collection $S_1, \ldots, S_r$ of integers in the range $1, \ldots n$ and an integer k. The problem is to decide if there exist k subsets $S_{i_1}, \ldots, S_{i_k}$ such that

$$\bigcup_{j=1}^{k} S_{i_k} = \{1, \ldots, n\} \ .$$

In other words, is it possible to cover the n integers with k of the sets.

1. Prove that SET-COVER is NP-hard.

5.11 NP-Hardness of Hamiltonian-Path

A *Hamiltonian path* in a graph G is a path in G that visits each vertex exactly once.

1. Show that HAM-PATH, the problem of determining whether a graph G has a Hamiltonian path is NP-hard.

5.12 NP-Hardness of and Algorithms for 3-Hitting-Set

The 3-HITTING-SET problem is the following: You are given a set $T = \{T_1, \ldots, T_n\}$ of n triples, where each triple consists of 3 integers. A *hitting set* for T is a set of integers such that every triple in T contains at least one of the integers. For example the set $\{1, 2, 3\}$ is a hitting set for

$$\{(1, 2, 3), (3, 4, 7), (1, 5, 6), (2, 7, 8)\} \ .$$

1. Show that the decision problem: Does T have a hitting set of size k is NP-complete. (Hint: it is a generalization of one of the NP-complete problems described in class.)
2. Give an algorithm for the 3-HITTING-SET decision problem that runs in $O(n3^k)$ time.
3. Give a fast 3-approximation algorithm for finding the smallest hitting set.
4. Using linear programming we can find $x_1, \ldots, x_m$, with $0 \le x_i \le 1$, such that

   $$x_{a_i} + x_{b_i} + x_{c_i} \ge 1 \tag{5.1}$$

   for all $1 \le i \le n$ and

   $$\sum_{i=1}^{m} x_i$$

   is minimized.

   Describe how to find $x'_1, \ldots, x'_m$ that satisfy (5.1), such that each x'_i is either 0 or 1, and

   $$\sum_{i=1}^{m} x'_i \le 3 \sum_{i=1}^{m} x_i \ .$$

5.13 Self-Reducibility of Vertex-Cover

Suppose we have an algorithm $\mathcal{A}$ for VERTEX-COVER that runs in $O(T(n, m))$ time on graphs with n vertices and m edges.[1] More precisely, the algorithm takes a graph G and an integer k and outputs *true* if G has a vertex cover of size k and *false* otherwise. *The algorithm doesn't output anything else.*

1. Using the above algorithm, describe an $O(T(n, m) \log n)$ time algorithm to determine the smallest value k' such that G has a vertex cover of size k'.
2. Describe how to improve the running time of the above algorithm to $O(T(n, m) \log k')$, where k' is the size of the smallest vertex cover of G.
3. Notice that the above algorithms don't tell us the actual vertices in the vertex cover, just whether or not one exists. Show how, if we already know some value k for which G has a vertex cover of size k we can find the vertices of a vertex cover of size k in $O(nT(n, m))$ time.

5.14 A Fast Algorithm for Max-Clique in Graphs of Bounded Degree

In class we saw that the CLIQUE decision problem is NP-complete. In this question you are asked to show that finding the largest clique in a graph G with n vertices can be done in $O(n)$ time if all vertices of G have constant degree.

[1] Assume $T(n, m)$ is a non-decreasing function of both n and m.

1. Let G be a graph with maximum degree 3. Give the best upper bound you can come up with on the size of the largest clique in G. Show that your bound is optimal, i.e., if your bound is k then give an example of a graph with max-degree 3 that contains a clique of size k

2. Let G be a graph with maximum degree 3. Give an $O(n)$ time algorithm to find the largest clique in G.

3. Let G be a graph with maximum degree d. Give an $O(nd^2 2^d)$ time algorithm to find the largest clique in G.

5.15 Independent Set of Squares

Let $S = \{s_1, \ldots, s_n\}$ be a set of axis-aligned squares, ordered by increasing size. A subset $S' \subset S$ is *independent* if no two squares in S' intersect each other. The goal of this question is to find an independent subset of maximum area.

1. Consider the square s_n (the largest square). Let S_n denote the set of squares in S that intersect s_n. Give an upper bound on the maximum total area of an inependent subset of S_n.

2. Give a greedy constant factor approximation algorithm that finds the an independent subset of S that approximates the area of the maximum area independent subset of S.