Assignment 6

CS 577 *QuickSort & QuickSelect – Randomized Algorithms* Fall 2024

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Related Readings: `http://pages.cs.wisc.edu/~hasti/cs240/readings/`

Name: ______________________ Wisc id: ______________________

- **Purpose of Homework:**
  - Algorithm design and analysis, like any skill, can only be developed through consistent practice and feedback. Whether it's cooking, playing basketball, integration, gardening, interviewing, or teaching, theoretical knowledge alone is not sufficient. The comfortable feeling of "Oh, sure, I get it" after following a well-presented lecture or hearing a TA explain a homework solution is a seductive, yet dangerous trap. True understanding comes from doing the thing—by actually solving the problems yourself.
  - The homework assignments are your opportunity to practice. Lectures, textbooks, office hours, labs, and guided problem sets are designed to build intuition and provide justification for the skills we want you to develop. However, the most effective way to develop those skills is by attempting to solve the problems on your own. The process is far more important than the final solution.
  - Expect to get stuck. It's normal to have no idea where to start on some problems. That's why you have access to a textbook, lecture slides, and discussions. The journey of wrestling with the problem is an essential part of the learning process.

Scoring Guidelines

You do not need to solve all the exercises.
Choose based on your familiarity with the probability concepts.

The exercises are divided into three categories:

1. Category A: No Probability Knowledge Required
2. Category B: Basic Probability Knowledge Required
3. Category C: Advanced Probability Knowledge Required

⋆ Category A: No Probability Knowledge Required

1a) 10 points	1b) 10 points	1c) 10 points	1d) 10 points	1e) 10 points
2) 10 points	3) 10 points	3b) 10 points		
4a) 2.5 points	4b) 2.5 points	4c) 5 points		
8a) 5 points	8b) 5 points			
14a) 2.5 points	14b) 2.5 points	14c) 2.5 points	14d) 2.5 points	

⋆ Category B: Basic Probability Knowledge Required

5) 10 points	6a) 5 points	6b) 5 points	6c) 5 points	6d) 5 points
6e) 5 points	6f) 15 points	7a) 2.5 points	7b) 2.5 points	7c) 5 points
9a) 5 points	9b) 5 points	10a) 2.5 points	10b) 2.5 points	
11a) 12.5 points	11b) 5 points	11c) 12.5 points		

⋆ Category C: Advanced Probability Knowledge Required

12a) 2.5 points	12b) 2.5 points	12c) 2.5 points	12d) 2.5 points	12e) 2.5 points
12f) 2.5 points	13) 10 points	15a) 5 points	15b) 10 points	
16a) 12.5 points	16b) 5 points	16c) 12.5 points		

Scoring Summary: Your total score is the sum of the points you collect from solving the exercises: As you can see the total scores per categories are $(A, B, C) = (110, 105, 70)$

Homework Guidelines

- **Collaboration and Academic Integrity:**
  - You are encouraged to work together on homework problems, but you must list everyone you worked with for each problem.
  - You must write everything in your own words and properly cite every external source you use, including ideas from other students. The only sources that you are not required to cite are the official course materials (lectures, notes, homework solutions).
  - Plagiarism is strictly prohibited. Using ideas from other sources or people without citation is considered plagiarism. Copying verbatim from any source, even with citation or permission, is also considered plagiarism. Don't cheat.
- **Submission Instructions:**
  - Submit your homework solutions as PDF files on Gradescope. Submit one PDF file per numbered homework problem.
  - Gradescope will not accept other text file formats such as plain text, HTML, LaTeX source, or Microsoft Word (.doc or .docx).
  - Homework submitted as images (.png or .jpg) will not be graded.
  - Each submitted PDF file should include the following information prominently at the top of the first page: [your full name]_[course title]_[homework assignment number].pdf
- **Solution Writing:**
  - When writing an algorithm, a clear description in English is sufficient. Pseudo-code is not required.
  - Ensure that your algorithm is correct by providing a justification, and analyze the asymptotic running time of your solution. Even if your algorithm does not meet the requested time bounds, you may receive partial credit for a correct, albeit inefficient, solution.
  - Pay close attention to the instructions for each problem. Partial credit may be awarded for incomplete or partially correct answers.

Sorting by k Means

1. *Consider an array $A[1 \ldots n]$ with n elements, and suppose we divide it into k contiguous subarrays:*

$$A_1[1 \ldots \frac{n}{k}], \quad A_2[\frac{n}{k}+1 \ldots 2\frac{n}{k}], \quad \ldots, \quad A_k[(k-1)\frac{n}{k}+1 \ldots n]$$

where each subarray contains $\frac{n}{k}$ elements. We will say that the array A is sorted by k means if, for each i and j where $1 \leq i < j \leq k$, every element in subarray A_i is less than or equal to every element in subarray A_j. In other words, the elements are sorted "externally" between the subarrays but not necessarily sorted "internally" within them.

For example, consider the array:

$$A = [\ 12 \quad 14 \quad 20 \quad 18 \quad 25 \quad 22 \quad 29 \quad 32 \quad 37 \quad 42 \quad 34 \quad 50 \quad 67 \quad 59 \quad 52 \quad 76\ ]$$

This array is sorted by 4 means. If it is necessary in your analysis, recall that using Stirling's approximation $\log(m!) = m \log m - m + O(\log m)$.

(a) **Algorithm for Sorting by k Means:** Design a comparison-based algorithm with a time complexity of $O(n \log(k))$ that sorts an array A of n elements by k means.

(b) **Sorting the Subarrays:** Design a comparison-based algorithm with a time complexity of $O(n \log(n/k))$ that fully sorts an array A that is already sorted by k means.

(c) **Optimality of the Algorithm (Lower Bound):** Prove that the algorithm you designed in part (b) is optimal, i.e., every comparison-based sorting by k means algorithm for an array of n elements has a worst-case time complexity of $\Omega(n \log(n/k))$ using the comparison-tree method.

(d) **Comparison-Based Sorting by k Means (Lower Bound):** Prove that every comparison-based sorting by k means algorithm has a worst-case time complexity of $\Omega(n \log(k))$.

(e) **Interconnection of the Lower Bounds:** Assume you already know one of the two lower bounds, i.e., either $\Omega(n \log(k))$ or $\Omega(n \log(n/k))$. Can you deduce the other lower bound more easily from the known one?

(f) Recall the exercise from Assignment 3, Exercise 5, where we showed that we can sort a k-confused array using a heap in time $O(n \log k)$. Recall that a k-confused array is an array in which each element is at most k positions away from its final sorted position. Now, prove that the lower bound is also $n \log k$. Is this lower bound tight?
Hint: Explain why a sorted-by-k-means array can be considered k-confused. Then, explain why the lower bound for sorting by k-means also applies to sorting k-confused arrays.

Nuts and Bolts

The following interesting problem was first proposed by object-oriented programming professor Gregory Rawlins around 1992.

2. Suppose we are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, and we're playing with them in the dark, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut with a bolt, however, the nut will be either too big, too small, or just right for the bolt. How quickly can we match each but to the corresponding bolt? Describe your algorithm, its expected and worst-case performance
Hint: Gregory loved a comparator.

Multi-set Selection

3. In the following exercise, we will investigate how we can use statistics of the input, such as the `cdf` (cumulative distribution function), to solve a selection problem more efficiently.
   (a) Let S be a multiset of positive integers, all smaller or equal to a given integer M. We have access (only) to the distribution F_S of the elements of the collection. Specifically, we are provided with a function $F_S(\ell)$, which for every natural number ℓ, returns the number of elements in S that do not exceed ℓ, i.e.:
   $$F_S(\ell) = |\{x \in S : x \leq \ell\}|$$
   You are tasked with designing a provably correct algorithm that, when given an input integer k ($1 \leq k \leq n$), computes the k-th smallest element of S by using F_S. Prove the correctness of your algorithm and calculate (by invoking F_S) the number of required calls to F_S (in the worst-case scenario). Estimate the number of calls to F_S without examining n (though it may depend on M).
   Hint: Please take care of corner case like {2nd element in $S = [1, 2, 2, 2, 4, 10]$*}*
   (b) Let $A[1 \ldots n]$ be an array of n distinct positive integers, and let M be the maximum element of A. Consider the multiset S composed of all the non-negative differences between pairs of elements of A. Specifically, we define:
   $$S = \{A[i] - A[j] : i \neq j \text{ and } A[i] > A[j]\}$$
   Design a provably correct algorithm to compute the k-th smallest element of S. Determine the computational complexity of your algorithm (as a function of n and M), and justify its correctness.
   Hint: Attempt to implement F_S and use the algorithm from part (a).

Searching into Medical Records

Imagine a hospital's database that contains two sorted lists of patient records, A and B, each of size n. One list contains patients with chronic illnesses, while the other contains patients admitted for surgeries. Both lists are sorted by patient ID, and all patient IDs are distinct across the two lists. Now, the hospital's data team needs to find the *median* patient ID across both lists. However, manually merging the lists to find the median can be too time-consuming with thousands of records. Instead, how can they efficiently find the median without combining the two lists?
Remark: Due to the size of A and B, they can not be at the same data center.

4. By leveraging an efficient algorithm that takes advantage of the sorted nature of both lists, we can quickly find the median without having to merge them. Let's explore how this can be done.
   (a) Before solving the selection problem, let's first address the rank problem: For any given patient ID x, devise an algorithm to find the number of patient IDs in $A \cup B$ that are less than x.
   (b) Give an algorithm running in $O(\log^2(n))$ time that finds the patient ID appearing in the median place when we sort $A\&B$ using the previous part, *(obviously without merging the two lists).*
   (c) Give an algorithm running in $O(\log(2n))$ time that finds the patient ID appearing in the kth place when we sort $A\&B$ *(obviously without merging the two lists).*

How Many Times Can a Minimum Change?

5. Let $a_1, a_2, \ldots, a_n$ be a set of n numbers, and let us randomly permute them into the sequence $b_1, b_2, \ldots, b_n$. Define $c_i = \min_{k=1}^{i} b_k$, which is the minimum of the first i elements of the sequence. Let X be the random variable representing the number of distinct values that appear in the sequence $c_1, c_2, \ldots, c_n$, that is, the number of times the minimum value changes as we progress through the sequence.
   (a) What is the expected value of X, the number of times the minimum changes in the linear search?

Closest Pair in Expected Linear Time

6. Throughout this section, we are going to assume that every hashing operation takes (in the worst case) constant time. This is quite a reasonable assumption when true randomness is available (e.g., using perfect hashing as explained in [CLRS]). In this exercise, we will design a randomized linear-time algorithm for the closest pair problem:

   Closest Pair Problem: Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(P) = \min_{p,q \in P} \|p - q\|$.

   (a) *Mapping Points to a Grid Sieve*: Let G_r be a grid with cell width r, and assume you have a point set $\mathbf{P}$ in the plane. For each point $\mathbf{p} = (x, y)$, the grid cell it belongs to is determined by $\text{id}(\mathbf{p}) = (\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.

   - For the points $\mathbf{p}_1 = (2.3, 4.7)$, $\mathbf{p}_2 = (6.5, 8.3)$, and $\mathbf{p}_3 = (10.1, 14.9)$, and grid width $r = 3$, compute the grid cell IDs $\text{id}(\mathbf{p}_1)$, $\text{id}(\mathbf{p}_2)$, and $\text{id}(\mathbf{p}_3)$.
   - If $|x_{\max}(P) - x_{\min}(P)| \le \mathcal{X}$ and $|y_{\max}(P) - y_{\min}(P)| \le \mathcal{Y}$, what is the maximum number of grid cells that exist with different IDs?

   (b) Explain how someone can efficiently store and look up points that belong to the same or neighboring grid cells. What is the space complexity of your solution? Additionally, handle the following scenario: You have the points $\mathbf{p}_1 = (5.1, 6.2)$, $\mathbf{p}_2 = (2.7, 9.3)$, and $\mathbf{p}_3 = (4.8, 7.1)$, with grid width $r = 3$. Describe how the points would be stored in the grid-based data structure, and how you would efficiently retrieve all points that fall into the same grid cell as $\mathbf{p}_1$.

   (c) Prove that if our grid sieve has squares of side length $\alpha = \mathcal{CP}(P)$, then $|P| \le 4$ in any grid cell.

   (d) Suppose you have a set P of n points in the plane such that $r = \mathcal{CP}(P)$, along with the grid structure $G_r(P)$ and the closest pair p_i^*, p_j^*. A new point p is inserted. Show how in constant time we can compute the new closest pair.

   (e) Given a set P of n points in the plane, and a distance r, explain how one can verify in linear time whether $\mathcal{CP}(P) < r$ or $\mathcal{CP}(P) \ge r$.

   (f) *Closest Pair in Expected Linear Time:* Using the above method, show how one can compute the closest pair of points of P in expected linear time.
   Hint: How many points q in P exist such that $\mathcal{CP}(P \setminus \{q\}) > \mathcal{CP}(P)$?

Maximum Load in Random Binning

When using random hash functions in chained hash tables, the expected search time is constant. However, we are concerned about the worst-case search time, which can be analyzed using the following binning problem. Suppose we toss n balls independently and uniformly at random into n bins. Can we determine the maximum number of balls in the fullest bin?

7. In this exercise, we will show that if n balls are thrown independently and uniformly into n bins, then with high probability $\left(1 - \frac{1}{\text{poly}(n)}\right)$, the fullest bin contains $O\left(\frac{\log n}{\log \log n}\right)$ balls. To do so, let X_j denote the number of balls in bin j, and let $\hat{X} = \max_j X_j$ be the maximum number of balls in any bin. Clearly, the expected value of X_j is 1 for all j.

   (a) Show that the probability $\Pr[X_j \ge k]$ is bounded as $\Pr[X_j \ge k] \le \frac{1}{k!}$.
   Hint: Consider the probability that bin j contains at least k balls. This occurs if a specific subset of k balls lands in bin j, which happens with probability $\binom{n}{k}\left(\frac{1}{n}\right)^k$.

   (b) Let $k = 2c\frac{\log n}{\log \log n}$, where c is a constant. Show that:

$$\Pr\left[X_j \ge 2c\frac{\log n}{\log \log n}\right] \le \frac{1}{n^c}.$$

Hint: Use $k! \geq \left(\frac{k}{2}\right)^{k/2}$ *&* $n = \omega(\sqrt{n}\log n)$.

(c) Conclude that:

$$\Pr\left[\max_j X_j \geq k\right] \leq \frac{1}{n^{c-1}}.$$

Hint: $\Pr[\max_j X_j \geq k] = \Pr[\exists j \text{ s.t. } X_j \geq k]$. *(Why?)*

(d) We will conclude with the inverse question. Prove that for any $\varepsilon > 0$, if n balls are thrown independently and uniformly into $n^{2+\varepsilon}$ bins, then with high probability $(1 - \frac{1}{n^{\epsilon}})$, no bin contains more than one ball.

Solving Faster for Sum Equations

In the previous assignment, we worked on the following problem:

> *We are given an array A of n integers. The task is to give a naive algorithm that runs in $O(n^3)$ time to determine if there exist $x, y, z \in A$ (not necessarily distinct) such that $x + y + z = 0$. We then improved the solution to $O(n^2 \log n)$ time by considering all sums of two elements and using binary search.*

8. *Now, we will further optimize the solution using hash tables and generalize the approach to handle k-tuples of integers.*
   (a) Solve the problem in expected $O(n^2)$ time using hash tables.
   (b) Generalize the above algorithm for a k-tuple. Suppose k is even. Give an algorithm to determine if there is a k-tuple of entries in A, not necessarily distinct, such that they sum to 0. The algorithm should run in expected $O(n^{k/2})$ time.

One hash, Two Hashes, Three Hashes: How Many HashMaps?

Based on hashing, we want to develop a data structure that maintains a summary of a set S, consisting of n positive integers, and efficiently implements (practically, in constant time) the following two operations: *a)* **Add x to S.** *b)* **Check if $x \in S$.**

9. For this purpose, we maintain a summary of S in an array A with m positions, each of size one binary digit, and we use hash functions $h : \mathbb{N}_+ \to [m]$ such that $\text{Prob}[h(x) = j] = \dfrac{1}{m}$ for every $x \in \mathbb{N}_+$ and every $j \in [m]$. Finally, we require that the use of randomness does not lead to false negatives (i.e., every negative answer to the question "is $x \in S$?" must be correct), although it may lead to false positives (i.e., a positive answer to the question "is $x \in S$?" may be incorrect) but with small probability.
   (a) Design such a data structure using only one hash function and compute the probability of a false positive answer. What is the probability of a false positive answer if $m = 8n$?
   (b) Design such a data structure using $k \geq 1$ independent hash functions and compute the probability of a false positive answer. What is the optimal value of k and what is the corresponding probability of a false positive answer if $m = 8n$?

Verifying Polynomial Identity via Random Sampling

10. You are given two polynomials $F(x)$ and $G(x)$, each of degree at most d. You are tasked with verifying whether the identity $F(x) \equiv G(x)$ holds, but instead of directly computing their canonical forms, you will use a randomized algorithm. The algorithm proceeds as follows:

- Choose a random integer r uniformly from the range $\{1, \ldots, 100d\}$.
- Evaluate $F(r)$ and $G(r)$.
- If $F(r) \neq G(r)$, conclude that $F(x) \neq G(x)$.
- If $F(r) = G(r)$, conclude that $F(x) = G(x)$.

(a) Prove that the algorithm has one-sided error and give a condition that the algorithm gives a wrong answer.

(b) Prove that the probability of the algorithm giving the wrong answer (i.e., $F(x) \neq G(x)$ but the algorithm concludes that $F(x) = G(x)$) is at most $\frac{1}{100}$.

(c) Suppose we repeat the algorithm k times, choosing a new random r each time. What is the probability that the algorithm gives the wrong answer after k iterations?

(d) Prove that by repeating the algorithm enough times, you can reduce the probability of error to less than ϵ, for any given small $\epsilon > 0$. How many iterations k are needed to ensure that the probability of error is less than ϵ?

Sorting Check in Middle-earth

In the ancient land of Middle-earth, the Elves and the Wise speak of a great challenge—one that tests the knowledge of order and chaos among the scrolls of numbers. A list of integers $A[1 \ldots n]$ is said to be *almost sorted* if there are at most $n/4$ elements that, if banished from the scroll, leave behind a sequence that follows the ancient law of order. For example, the scroll $[1, 2, 3, 5, 4, 7, 9, 6, 10, 11, 12, 8]$ is almost in order, for if we strike from it positions 5 and 6, the sequence $[1, 2, 3, 4, 7, 9, 10, 11, 12]$ shall remain, which obeys the Law of Order. For simplicity, it is said that all elements in the scroll are distinct, like the stars in the sky.

11. The task of the quest is to design a probabilistic algorithm with the speed of the swiftest Elven arrows, one that can distinguish between lists that are almost ordered and those that are not[1]. Specifically, if the scroll is fully ordered, the algorithm shall always declare it so. Furthermore, with great certainty (with a probability of at least 9 out of 10), if the scroll is almost ordered, the algorithm must accept it. However, if chaos reigns in the scroll, the algorithm must, with a similar probability of 9 out of 10, declare that the scroll is far from ordered.

    (a) Consider an ancient algorithm, one that selects k positions in the scroll A at random, much like the wanderings of hobbits through Middle-earth. It declares that the scroll is almost ordered if, for every chosen position a_i, it holds true that $A[a_i] \leq A[a_i + 1]$. Should any a_i defy this law, the algorithm declares the scroll to be unfit and chaotic. Provide an example of a scroll where the algorithm must inspect $k = \Omega(n)$ positions to ensure that the likelihood of error is less than 0.1, lest the scrolls of knowledge be corrupted.

    (b) Suppose we call upon the wisdom of the Elves to use a variant of the legendary Binary Search to search within a list A that may not follow the ancient Law of Order (with the risk, of course, that the search may fail, as even the wisest cannot always find what is not there). The search proceeds as follows:

```
BINARY-SEARCH(A, x, low, up)
    if low == up then return low;
    else mid = ceil [ (low + up)/2 ];
    if x < A[mid] then return BINARY-SEARCH(A, x, low, mid - 1);
    else return BINARY-SEARCH(A, x, mid, up);
```

[1]At first glance, this may seem a lofty goal indeed, for no deterministic algorithm could solve this without much toil—at least as much toil as examining $3n/4$ of the elements in the scroll A.

Imagine now, for two values x and y, that BINARY-SEARCH$(A, x, 1, n)$ returns the position k, and BINARY-SEARCH$(A, y, 1, n)$ returns position ℓ. Show that if $k < \ell$, then $x < y$, as one path must always be less than another in the great search for order.

(c) Now, suppose we have an algorithm that selects k positions in the scroll A, much like the gathering of an Elven council, and declares the scroll almost ordered if, for each chosen position a_i, it holds true that $a_i =$ BINARY-SEARCH$(A, A[a_i], 1, n)$. Should any a_i not satisfy this, the algorithm will reject the scroll as unworthy of the Law of Order. Using part (b), show that for any scroll A, with $k = \Theta(1)$ chosen positions, the probability of failure becomes less than 0.1, a triumph worthy of the most learned Elves and Wise of Middle-earth.

Las Vegas Selection: Average Case Analysis of QuickSelect

QuickSelect receives an array $t[1 \dots n]$ of n real numbers, and a number k, and returns the element of rank k in the sorted order of the elements of t. *Formally, QuickSelect chooses a random pivot, splits the array according to the pivot, and recurses only on the subproblem containing the required element. This implies that we now know the rank of the pivot, and if it's equal to k, we return it. Otherwise, we recurse on the subproblem containing the required element (adjusting k as we proceed). QuickSelect is a modification of QuickSort that performs only a single recursive call instead of two.*

12. We will now perform an average-case analysis of the number of comparisons in QuickSelect, similarly to what we did in class for QuickSort. To bound the expected running time, we will analyze the expected number of comparisons. Let $S_1, \dots, S_n$ be the elements of t in sorted order. For $i < j$, let X_{ij} be the indicator variable that is one if S_i is compared to S_j during the execution of QuickSelect. We will now calculate the expected value of the number of comparisons in various cases.

    (a) Compute $\alpha_1 = \mathbb{E}\left[\sum_{i<j<k} X_{ij}\right]$, the expected number of comparisons for the case $i < j < k$. Prove that $\alpha_1 \le 2(k-2)$.

    (b) Compute $\alpha_2 = \mathbb{E}\left[\sum_{j=k+1}^{n} \sum_{i=k+1}^{j-1} X_{ij}\right]$, the expected number of comparisons for the case $k < i < j$. Prove that $\alpha_2 \le 2(n-k)$.

    (c) Compute $\alpha_3 = \mathbb{E}\left[\sum_{i=1}^{k-1} \sum_{j=k+1}^{n} X_{ij}\right]$, the expected number of comparisons for the case $i < k < j$, and over all possibilities for i and j. Prove that $\alpha_3 \le \sum_{\Delta=3}^{n} 2(\Delta-2)/\Delta \le 2n$.

    (d) Compute $\alpha_4 = \sum_{j=k+1}^{n} \mathbb{E}[X_{ij}]$, the expected number of comparisons for the case $i = k$. Prove that $\alpha_4 \le \ln n + 1$.

    (e) Compute $\alpha_5 = \sum_{i=1}^{k-1} \mathbb{E}[X_{ij}]$, the expected number of comparisons for the case $j = k$. Prove that $\alpha_5 \le \ln n + 1$.

    (f) Finally, conclude by summing $\sum_i \alpha_i$ to show that the expected number of comparisons is $O(n)$.

Verifying properties : Monte Carlo Sampling

One of the big advantages of randomized algorithms is that they sample the world; that is, learn how the input looks like without reading all the input. For example, consider the following problem: We are given a set U of n objects $u_1, \dots, u_n$, and we want to compute the number of elements of U that have some property. Assume that one can check if this property holds, in constant time, for a single object, and let $\psi(u)$ be the function that returns 1 if the property holds for the element u and 0 otherwise. Now, let Γ be the number of objects in U that have this property. We want to reliably estimate Γ without computing the property for all the elements of U. A natural approach would be to pick a random sample R of m objects, $r_1, \dots, r_m$ from U (with replacement), and compute $Y = \sum_{i=1}^{m} \psi(r_i)$. The estimate for Γ is $\beta = (n/m)Y$. It is natural to ask how far β is from the true value Γ.

13. Let U be a set of n elements, with Γ of them having a certain property ψ. Let R be a uniform random sample from U (with repetition), and let Y be the number of elements in R that have the property ψ, and let $Z = (n/m)Y$ be the estimate for Γ. Then, for any $t \geq 1$, we have that

$$\mathbb{P}\left[\Gamma - \frac{tn}{2\sqrt{m}} \leq Z \leq \Gamma + \frac{tn}{2\sqrt{m}}\right] \geq 1 - \frac{1}{t^2}.$$

Similarly, we have that

$$\mathbb{P}[\mathbb{E}[Y] - t\sqrt{m}/2 \leq Y \leq \mathbb{E}[Y] + t\sqrt{m}/2] \geq 1 - \frac{1}{t^2}.$$

Endless Versions of Average Case Analysis of QuickSort

14. In class, we discussed the derivation of the recurrence relation for the average-case complexity of QuickSort, $\overline{T}(n)$:

$$\overline{T}(n) = \frac{1}{n}\sum_{k=1}^{n}(T(k-1) + T(n-k)) + O(n)$$

In this exercise, we will solve the bound for this recurrence step-by-step.

(a) Firstly, show that:

$$n\overline{T}(n) = 2\sum_{k=0}^{n-1} T(k) + 2n^2 - n$$

(b) Prove the following relation:

$$n\overline{T}(n) - (n-1)\overline{T}(n-1) = 2T(n-1) + 4n - 3$$

Hint: Shift and subtract terms (Simplicity is Magic).

(c) Define a new function $t(n) = \frac{\overline{T}(n)}{n+1}$ and prove that:

$$t(n) = t(n-1) + \frac{7}{n+1} - \frac{3}{n} \implies t(n) = 4H_n - 7 + \frac{7}{n+1}$$

where $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \log n + \gamma$, the n-th order harmonic number

(d) Finally, substitute back $\overline{T}(n) = (n+1)t(n)$, we can derive a tight $O(n \log n)$ form of the form:

$$\overline{T}(n) = (n+1)(4H_n - 7 + \frac{7}{n+1}) = 4nH_n - 7n + 4H_n$$

15. In this exercise, we aim to prove that `QuickSort` is efficient using conditional expectation. Consider a specific element x in an input array of n elements that is being sorted by `QuickSort`. Let X_i denote the size of the recursive subproblem at the i-th level of the recursion that contains x. If x does not participate in a subproblem at this level, then $X_i = 0$. It is clear that $X_0 = n$.

(a) As a first step, compute the probability that the i-th pivot is an element between the $X_{i-1}/4$-th and $(3/4)X_{i-1}$-th positions. Prove that $\mathbb{E}[X_i \mid X_{i-1}] \leq (7/8)X_{i-1}$.

(b) Using the result from part (a), show that $\mathbb{E}[X_i] = \mathbb{E}[\mathbb{E}[X_i \mid X_{i-1}]] \leq (\frac{7}{8})^i n$. Using Markov's inequality, prove that:

$$\mathbb{P}\left(x \text{ participates in more than } c\ln n \text{ levels of recursion}\right) \leq \frac{1}{n^{\beta}},$$

where $\beta > 1$ is a constant.

16. In class, we discussed Chernoff bounds :

    Let $X_1, \ldots, X_n$ be n independent coin flips, such that $\mathbb{P}[X_i = 0] = \mathbb{P}[X_i = 1] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have $\mathbb{P}[|Y - n/2| \geq \Delta] \leq 2\exp\left(-\frac{2\Delta^2}{n}\right)$..

    Let's see how we could use this powerful statistical tool for the `QuickSort` problem.

    (a) Prove that Chernoff bounds can be generalized to infinite series: *Let $X_1, X_2, \ldots$ be an infinite sequence of independent random 0/1 variables. Let M be an arbitrary parameter. Then the probability that we need to read more than $2M + 4t\sqrt{M}$ variables of this sequence till we collect M ones is at most $2\exp(-t^2)$, for $t \leq \sqrt{M}$. If $t \geq \sqrt{M}$, then this probability is at most $2\exp(-t\sqrt{M})$.*

    (b) Let's revisit `QuickSort` one last time. Consider an arbitrary element u being sorted, and the i-th level recursive subproblem that contains u. Let S_i be the set of elements in this subproblem. We consider u to be *successful* in the i-th level, if $|S_{i+1}| \leq |S_i|/2$. Namely, if u is successful, then the next level in the recursion involving u would include a considerably smaller subproblem. Let X_i be the indicator variable which is 1 if u is successful. We first observe that if `QuickSort` is applied to an array with n elements, then u can be successful at most $T = \lceil \lg n \rceil$ times, before the subproblem it participates in is of size one, and the recursion stops. Thus, consider the indicator variable X_i, which is 1 if u is successful in the i-th level, and zero otherwise.
    Prove that the X_i's are independent, and $\mathbb{P}[X_i = 1] = 1/2$.

    (c) *Typically, if u participates in v levels, then we have the random variables $X_1, X_2, \ldots, X_v$. To make things simpler, we will extend this series by adding independent random variables, such that $\mathbb{P}[X_i = 1] = 1/2$ for $i \geq v$. Thus, we have an infinite sequence of independent random variables, that are 0/1 and get 1 with probability 1/2.* Prove that for any $c > 0$, the probability that `QuickSort` performs more than $(6 + c)n \lg n$ comparisons is smaller than $1/n^c$.