CS 31: Homework 1

Thomas Monfre

January 14, 2019

Problem 1-1

Prove the following. Let A be an event and $X_A = I\{A\}$ be the indicator random variable for A. Then for all real numbers $c > 0$, we have $E[X_A^c]$ =Pr$\{A\}$.

Proof. By the definition of an indicator random variable from equation (5.1) in CLRS, let us define:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs },\\ 0 & \text{otherwise} \end{cases}$$

Since we are told $X_A = I\{A\}$, let us therefore simplify X_A:

$$X_A = \begin{cases} 1 & \text{if } A \text{ occurs },\\ 0 & \text{otherwise} \end{cases}$$

We seek to prove $E[X_A^c]$ =Pr$\{A\}$. Let us first observe that, for all real numbers $c > 0$, $X_A^c = X_A$, since $1^c = 1$ for all $c > 0$ and $0^c = 0$ for all $c > 0$. Therefore, $E[X_A^c] = E[X_A]$.

Then let us prove $E[X_A]$ =Pr$\{A\}$. Using the definition of expected value from CLRS 5.2, we know $E[X_A]$ is simply the sum, for all possible outcomes (i.e. either A occurs or does not occur), of the probability of the outcome multipled with its corresponding indicator random variable value.

Therefore, we have:

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1{*}\Pr\{A\} + 0{*}\Pr\{\bar{A}\} \\ &= \Pr\{A\} \end{aligned} \tag{1}$$

where, for the sample space S that A occupies, $\bar{A}$ denotes $S - A$, the compliment of A.

We therefore have $E[X_A]$ =Pr$\{A\}$.

Since $E[X_A] = E[X_A^c]$, we then have $E[X_A^c]$ =Pr$\{A\}$. Hence the claim.

■

Problem 1-2

Exercise 2.3-7: Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exists two elements in S whose sum is exactly x.

```
ContainsSumElements(S, x)
 1  // first create a copy of S with an implicit order in Θ(n) time
 2  A = [ ]    // empty array
 3  for i = 1 to S.length
 4      A.append(S[i])
 5
 6  // sort A in Θ(n lg n) time
 7  mergesort(A, 0, A.length)
 8
 9  // for each element, check to see if the corresponding value required to get x has been found yet
10  B = [ ]    // empty array
11  for i = 1 to A.length
12      check = x − A[i]    // element we need to find in B in order to sum to x
13      if binarysearch(B, 0, B.length, check)    // perform binarysearch in Θ(lg n) time
14          return True
15      B.append(A[i])
16
17  return False
```

This function first creates an empty array A at line 2, which is then used at lines 3-4 to create a copy of the input set S in $\Theta(n)$ time. We create a copy of the set in the form of an array so as to impose an implicit order on the collection. A set has no order, but so as to effectively search the set using binary search, we need to impose an order on it.

Once we've constructed A, we then ensure it is in sorted order by running merge sort at line 7 in $\Theta(n \lg n)$ time. We sort A to allow us to effectively search it using binary search.

Next, we create an empty array B at line 10 which will then be used to help determine if two elements exist in S that sum to x by holding each element of A after we examine it.

Then, we iterate over A at line 11. For each element, we know the value of the element we are currently iterating on. Let us define that element as $A[i]$. Since we know we are looking for two elements that sum to x, all we have to do is determine if the element $x - A[i]$ exists in S. We then define that element we are looking for as *check* at line 12.

Then, since B holds all values from A that we have previously iterated on, and A is in sorted order, B must be in sorted order as well. This allows us to perform binary search on the array B to determine if B contains *check*. We perform this search at line 13 in $\Theta(\lg n)$ time. If B contains *check*, then we must have previously iterated on *check*. This tells us that A contains *check*, which means that S contains *check*. Since $check + A[i] = x$, we then know two elements exist in S that sum to x.

In this case, we return True at line 14. After performing the search for *check*, we then add $A[i]$ to B at line 15. This will then be used in the subsequent iteration of the for-loop to determine if two elements that sum to x have been found.

If we make it out of the for-loop without returning, then it never was the case that we found *check* in B. This indicates no two elements exist in A that sum to x which means no two elements exist in S that sum to x. In this case, we then return false at line 17.

Let us now analyze the run-time of this algorithm. The initial creation of A runs in $\Theta(n)$ time since we iterate over each value in S which is of size n. The call to merge sort at line 7 runs in $\Theta(n \lg n)$ time since A is of size n. Lastly, for each element in A, we perform binary search on B (which increases in size but ultimately is of size n). Since binary search runs in $\Theta(\lg n)$ time, we have n runs of $\Theta(\lg n)$ which is $\Theta(n \lg n)$.

Therefore, the run-time of this algorithm is $\Theta(n) + \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n)$.

Problem 1-3

In class, we saw a lower-bound argument showing that the worst-case running time of insertion sort is $\Omega(n^2)$. The argument was based on dividing the array of n elements into three sections, each of size $\frac{n}{3}$.

Suppose that α is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through the middle $1 - 2\alpha$ array positions?

In class, we showed that by diving the array of n elements into three sections, each of size $\frac{n}{3}$, we could find a lower bound of $\Omega(n^2)$ so long as the $\frac{n}{3}$ largest elements were located in the first $\frac{n}{3}$ positions of the array. Since these $\frac{n}{3}$ elements were positioned in the first third of the array and had to end up in the last third, we knew by insertion sort that each element in the first third must traverse through the center third of the array one at a time in order to end up in the last third. This meant each of $\frac{n}{3}$ elements must traverse through each of $\frac{n}{3}$ positions, one at a time.

Therefore, we have $\frac{n}{3} * \frac{n}{3} = \frac{n^2}{9}$ moves. This yields $\Omega(n^2)$.

Now, we are given a fraction α in the range $0 < \alpha < 1$. We can generalize the above argument beyond separating the array into even thirds in order to show that the worst-case running time of insertion sort is $\Omega(n^2)$ by imposing a further restriction on α: Let us require that $0 < \alpha < \frac{1}{2}$.

Then let us split the given array into three sections, where the first and last sections are both of length αn and the middle section is of length $(1 - 2\alpha)n$ length. Then let us place the αn largest elements of the array into the first αn positions, thereby completely filling the first section.

Then, since the first and last sections are both of equal size and the αn largest elements are in the entire first section, we know every value in the first section must end up in the last section. Therefore, by insertion sort we must move each of αn elements through the middle $(1 - 2\alpha)n$ positions, one at a time.

Therefore, we have $\alpha n * (1 - 2\alpha)n = \alpha n^2(1 - 2\alpha) = \alpha n^2 - 2\alpha^2 n^2$ moves. This yields $\Omega(n^2)$.

What value of α maximizes the number of times that the αn largest values must pass through the middle $1 - 2\alpha$ array positions?

$\alpha = \frac{1}{4}$ will require $n^2/8$ traversals through the middle $1-2\alpha$ array positions, which is the maximum number of movements possible. This is because we know we must split the first and third sections into equal sizes. In order to get a large number of traversals, we need both the size of the first section and the size of the middle section to be relatively balanced. Having a small first section and a large middle section will not create as many traversals, because the denominator will begin expanding faster than the numerator. $\alpha = \frac{1}{4}$ creates a middle section of size $\frac{n}{2}$ and a first section of size $\frac{n}{4}$. This yields a maximum $n^2/8$ traversals.

Problem 1-4

Exercise 4.2-4: What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

This problem is a variant of Strassen's Algorithm, described in full in section 4.2 of CLRS. Specifically, it asks the largest number of recursive matrix multiplications required if we split the input matrices A and B into size $\frac{n}{3} \times \frac{n}{3}$ pieces rather than $\frac{n}{2} \times \frac{n}{2}$. Since we are told this variant runs in time $o(n^{\lg 7})$, let us use the master method (from section 4.5 of CLRS) to compute the expected run-time of this algorithm in terms of k, then compare with $o(n^{\lg 7})$ to solve for k. Using our result for k, we can then determine the final run-time.

Using the master method, we have $T(n) = kT(\frac{n}{3}) + \Theta(n^2)$. We then get $a = k$, $b = 3$ and $f(n) = n^2$. Therefore, we must compare $n^{\log_3 k}$ with n^2.

Intuitively, we can assume $k > 9$ since splitting the matrices into $\frac{n}{3}$ x $\frac{n}{3}$ pieces will create 9 submatrices per matrix. Therefore, $f(n) = n^2$ is polynomially smaller than $n^{\log_3 k}$. This indicates case 1 of the master method applies.

Therefore, we have $\Theta(n^{\log_3 k})$.

Then, we are told we can generalize this variant of Strassen's Algorithm to $n \times n$ matrices in time $o(n^{\lg 7})$. Since we have computed the run-time of this algorithm to be $\Theta(n^{\log_3 k})$, we then know:

$n^{\log_3 k} < n^{\lg 7}$
$\log_3 k < \lg 7$
$\log_3 k < 2.8073549221$

Since we are seeking the largest k, we then can then set an equality to solve for k and round-down to the nearest whole number.

$\log_3 k = 2.8073549221$
$k = 3^{2.8073549221}$
$k = 21.8498622259 \approx 21$ (rounding-down to observe the given bounds)

Therefore, the largest k is $k = 21$. We can then solve for the run-time by plugging 21 into our equation above. This yields $\Theta(n^{\log_3 21})$.

Let us observe that this solution makes sense since $\log_3 21 = 2.7712437492 < \lg 7 = 2.8073549221 < \log_3 22 = 2.8135880922$.

Problem 1-5

Problem 4-1. Use the master method when applicable. Don't worry about base cases.

(a) $T(n) = 2T(n/2) + n^4$

Using the master method we get $a = 2$, $b = 2$, and $f(n) = n^4$. Therefore, we must compare $n^{\log_2 2}$ with n^4.

Since $\log_2 2 = 1$, and $1 < 4$, $f(n) = n^4$ is polynomially larger than $n^{\log_2 2}$. This indicates case 3 of the master method applies. Checking the regularity condition we see $\frac{1}{8}(n^4) \leq cn^4$ for some constant $c < 1$ and all sufficiently large n.

Therefore, we have $\Theta(n^4)$.

(b) $T(n) = T(7n/10) + n$

Using the master method we get $a = 1$, $b = \frac{10}{7}$, and $f(n) = n$. Therefore, we must compare $n^{\log_{\frac{10}{7}} 1}$ with n.

Since $\log_{\frac{10}{7}} 1 = 0$, and $0 < 1$, $f(n) = n$ is polynomially larger than $n^{\log_{\frac{10}{7}} 1}$. This indicates case 3 of the master method applies. Checking the regularity condition we see $\frac{7}{10}(n) \leq cn$ for some constant $c < 1$ and all sufficiently large n.

Therefore, we have $\Theta(n)$.

(c) $T(n) = 16T(n/4) + n^2$

Using the master method we get $a = 16$, $b = 4$, and $f(n) = n^2$. Therefore, we must compare $n^{\log_4 16}$ with n^2.

Since $\log_4 16 = 2$, and $2 = 2$, $f(n) = n^2$ is within a polylog factor of $n^{\log_4 16}$, but is not smaller. This indicates case 2 of the master method applies.

Therefore, we have $\Theta(n^2 \lg n)$.

(d) $T(n) = 7T(n/3) + n^2$

Using the master method we get $a = 7$, $b = 3$, and $f(n) = n^2$. Therefore, we must compare $n^{\log_3 7}$ with n^2.

Since $\log_3 7 = 1.7712437492$, and $1.7712437492 < 2$, $f(n) = n^2$ is polynomially larger than $n^{\log_3 7}$. This indicates case 3 of the master method applies. Checking the regularity condition we see $\frac{7}{9}(n^2) \leq cn^2$ for some constant $c < 1$ and all sufficiently large n.

Therefore, we have $\Theta(n^2)$.

(e) $T(n) = 7T(n/2) + n^2$

Using the master method we get $a = 7$, $b = 2$, and $f(n) = n^2$. Therefore, we must compare $n^{\log_2 7}$ with n^2.

Since $\log_2 7 = 2.8073549221$, and $2.8073549221 > 2$, $f(n) = n^2$ is polynomially smaller than $n^{\log_2 7}$. This indicates case 1 of the master method applies.

Therefore, we have $\Theta(n^{\log_2 7)} = \Theta(n^{\lg 7})$.

(f) $T(n) = 2T(n/4) + \sqrt{n}$

Using the master method we get $a = 2$, $b = 4$, and $f(n) = \sqrt{n} = n^{\frac{1}{2}}$. Therefore, we must compare $n^{\log_4 2}$ with $n^{\frac{1}{2}}$.

Since $\log_4 2 = \frac{1}{2}$, and $\frac{1}{2} = \frac{1}{2}$, $f(n) = n^{\frac{1}{2}}$ is within a polylog factor of $n^{\log_4 2}$, but is not smaller. This indicates case 2 of the master method applies.

Therefore, we have $\Theta(\sqrt{n}\lg n)$.

(g) $T(n) = T(n-2) + n^2$

We cannot use the master method here. Therefore, let us guess that $T(n) = \Theta(n^3)$ and use the substitution method to prove. We begin by proving $T(n) = O(n^3)$.

Guess: $T(n) \leq cn^3$ for some constant c that we define.

$$\begin{aligned} T(n) &\leq T(n-2) + n^2 \\ &= c(n-2)^3 + n^2 \\ &= c(n^3 - 4n^2 + 8n - 8) + n^2 \\ &= cn^3 - 4cn^2 + 8cn - 8c + n^2 \\ &\leq cn^3 \quad \text{if } (-4cn^2 + 8cn - 8c + n^2) \leq 0 \end{aligned} \tag{2}$$

Therefore, $T(n) = O(n^3)$. Let us now prove $T(n) = \Omega(n^3)$.

Guess: $T(n) \geq cn^3$ for some constant c that we define.

$$\begin{aligned} T(n) &\geq T(n-2) + n^2 \\ &= c(n-2)^3 + n^2 \\ &= c(n^3 - 4n^2 + 8n - 8) + n^2 \\ &= cn^3 - 4cn^2 + 8cn - 8c + n^2 \\ &\geq cn^3 \quad \text{if } (-4cn^2 + 8cn - 8c + n^2) \geq 0 \end{aligned} \tag{3}$$

Therefore, $T(n) = \Omega(n^3)$.

Since we have shown $T(n) = O(n^3) = \Omega(n^3)$, $T(n) = \Theta(n^3)$.

Problem 1-6

Problem 4-3, parts g and j. Don't worry about base cases.

(g) $T(n) = T(n-1) + \frac{1}{n}$

We cannot use the master method here. Therefore, let us guess that $T(n) = \Theta(\lg n)$ and use the substitution method to prove. We begin by proving $T(n) = O(\lg n)$.

Guess: $T(n) \leq c \lg n$ for some constant c that we define.

$$\begin{aligned}
T(n) &\leq T(n-1) + \frac{1}{n} \\
&= c\lg(n-1) + \frac{1}{n} \\
&= c\lg n - c\lg(1 - \frac{1}{n}) + \frac{1}{n} \\
&\leq c\lg n \quad \text{ if } \ (c\lg(1-\tfrac{1}{n}) + \tfrac{1}{n}) \leq 0
\end{aligned} \tag{4}$$

Therefore, $T(n) = O(\lg n)$. Let us now prove $T(n) = \Omega(\lg n)$.

Guess: $T(n) \geq c \lg n$ for some constant c that we define.

$$\begin{aligned}
T(n) &\geq T(n-1) + \frac{1}{n} \\
&= c\lg(n-1) + \frac{1}{n} \\
&= c\lg n - c\lg(1 - \frac{1}{n}) + \frac{1}{n} \\
&\geq c\lg n \quad \text{ if } \ (c\lg(1-\tfrac{1}{n}) + \tfrac{1}{n}) \geq 0
\end{aligned} \tag{5}$$

Therefore, $T(n) = \Omega(\lg n)$.

Since we have shown $T(n) = O(\lg n) = \Omega(\lg n)$, $T(n) = \Theta(\lg n)$.

(j) $T(n) = \sqrt{n}\ T(\sqrt{n}) + n$

We cannot use the master method here. Therefore, let us guess that $T(n) = \Theta(n \lg \lg n)$ and use the substitution method to prove.

Guess: $T(n) = n \lg \lg n$

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &= \sqrt{n}(\sqrt{n} \lg \lg \sqrt{n}) + n \\ &= n \lg(\lg \sqrt{n}) + n \\ &= n \lg(\lg n^{\frac{1}{2}}) + n \\ &= n \lg(\frac{1}{2} \lg n) + n \\ &= n \lg(\frac{\lg n}{2}) + n \\ &= n \lg \lg n - n \lg 2 + n \\ &= n \lg \lg n - n + n \\ &= n \lg \lg n \end{aligned} \tag{6}$$

Therefore, $T(n) = \Theta(n \lg \lg n)$.

Problem 1-7

Exercise 5.2-5: Let $A[1...n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A. Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, ..., n\rangle$. Use indicator random variables to compute the expected number of inversions.

To solve this problem, we know from Lemma 5.1 of CLRS that we first need to identify the probability that a given pair of indices (i, j) is an inversion of A. Knowing this probability will then allow us to compute the total expected number of inversions.

The condition of having an inversion requires that $i < j$ **and** $A[i] > A[j]$. Since we need two conditions to hold in order to achieve an inversion, let us compute the expected number of inversions by only observing cases in which $i < j$. This allows us to simplify the probability that a given pair of indices (i, j) is an inversion of A to the probability that $A[i] > A[j]$.

Let us then compute the probability that $A[i] > A[j]$ as follows: for some index $1 \leq j \leq n$, $Pr\{A[i] > A[j]\} = \frac{n-A[j]}{n-1}$. Essentially, there are $n - A[j]$ values greater than $A[j]$ and $n - 1$ total values left for $A[i]$ to take on. This value is independent of our assumption that $i < j$ since the likelihood that $i < j$ is independent from the likelihood that $A[i] > A[j]$.

Since we have a uniform random permutation, we then know each element in A is equally likely to hold each possible value of $1...n$. In other words, the probability that $A[j]$ holds some value in $A[1...n]$ is $\frac{1}{n}$.

Therefore, $Pr\{A[i] > A[j]\} = \frac{1}{n} * \left(\frac{(n-1)+(n-2)+...+(n-n)}{n-1}\right) = \frac{1}{2}$.

Using this value, for each pair of indices (i, j), let us define the indicator random variable X_{ij}, for $1 \leq i < j \leq n$, by the following:

$$\begin{aligned} X_{ij} &= I\{(i,j) \text{ is an inversion of A}\} \\ &= \begin{cases} 1 & \text{if } (i,j) \text{ is an inversion of } A \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{7}$$

Notice that we have constrained X_{ij} such that $i < j$. Using this, we can compute the expected value of X_{ij} as:

$$\begin{aligned} E[X_{ij}] &= \Pr\{(i,j) \text{ is an inversion of } A\} \\ &= \frac{1}{2} \end{aligned} \tag{8}$$

We define $Pr\{(i, j)$ is an inversion of $A\}$ as $\frac{1}{2}$ because we constrainted ourselves to cases in which $i < j$ holds. Therefore, the probability of having an inversion is simply $Pr\{A[i] > A[j]\} = \frac{1}{2}$.

Then, letting X be the random variable that counts the total number inversions of A, we have:

$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Applying linearity of expectation, we get:

$$\begin{aligned} E[X] &= E\Big[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\Big] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \binom{n}{2} * \frac{1}{2} \\ &= \frac{n(n-1)}{2} * \frac{1}{2} \\ &= \frac{n(n-1)}{4} \end{aligned} \tag{9}$$

Therefore, the expected number of inversions is $\frac{n(n-1)}{4}$.