Assignment 1 - COMP3121

Bailey Ivancic — z5162498

March 2019

1 Question 1

> "You're given an array A of n integers, and must answer a series of n queries, each of the form: "How many elements a of the array A satisfy $L_k \leq a \leq R_k$?", where L_k and R_k $(1 \leq k \leq n)$ are some integers such that $L_k \leq R_k$. Design an $O(n \cdot \log n)$ algorithm that answers all of these queries."

Since we have an array of n integers, and we are doing n queries on the array, simply looking through each array element for each query would result in an $O(n^2)$ time complexity. Thus, to get an algorithm with time complexity of $O(n \cdot \log n)$, sorting the array A is necessary.

The first step is to run a standard Mergesort on the array A. This gives the algorithm an initial time complexity of $O(n \cdot \log n)$ in the worst case.

Now that the array A is sorted, we are able to perform a Binary search on it to find the largest element lower than L_k and the smallest element larger or equal to R_m.

Once we have found these two elements, we record the indexes of both. Knowing the indexes, we are able to then see all the elements within the array that are contained within these two bounds. As a result, finding the difference m - k will give us the number of elements a contained within the bounds.

1. Mergesort($A[\,]$, 1, n)
2. For each query of n queries:
   - 2.1 Binary search for index k of the largest element not exceeding L_k
   - 2.2 Binary search for index m of the smallest element larger or equal to R_m
   - 3.3 Elements a of array A = m - k

Looking at time complexity, the initial Mergesort costs O(n log n) time. Once the array is sorted, the binary search will complete in worst case O(log n). As we complete two of these for each query, the total complexity for the searching becomes O($2 \cdot \log n$). This search gets completed each time we want to query the array, and since we are performing n queries, the searching complexity becomes O($2 \cdot n \cdot \log n$).

Combining the two sections, we have a total complexity of O(n log n) + O(2 $\cdot n \cdot \log n$) = $\mathbf{O(3 \cdot n \cdot \log n)}$. Since the linear overheads are ignored when looking at Big O notation, we find the final complexity is $\mathbf{O(n \cdot \log n)}$.

2 Question 2

"You are given an array S of n integers and another integer x.
(a) Describe an O($n$log$n$) algorithm (in the sense of the worst case performance) that determines whether or not there exist two elements in S whose sum is exactly x.
(b) Describe an algorithm that accomplishes the same task, but runs in O(n)expected(i.e.,average) time."

a. Before searching the array, we are required to sort it. This again will be done using Mergesort, which has a time complexity of O(n log n). Once the array is sorted, the small and large integers will be on opposite ends of the array, meaning that the solution to $a + b = x$ will likely have a and b on opposite sides. As such, starting two variables at either end of the array and performing a search converging to the middle will determine if there are any such combination of integers.

   Initialising variables a at the start of array A and b at the end, the variables are incremented or decremented depending on the sum of the two variables. If $a + b = x$, then we have found the set of numbers, and can return from the algorithm. If $a + b < x$, then the sum is too small, so a is incremented to the next element in the array. Similarly, if the sum is $> x$, the sum is reduced by decrementing the variable b and moving it to the next lowest integer in the array. This runs within a loop with the condition that $b > a$. Since this is a linear search, it runs in O(n) time.

   searchPairs(Array[ ], size, x)

   1 Sort Array into ascending order.
   2 Initialise a to the first index of Array (0)
   3 Initialise b to the last index of Array (size - 1)
   4 While $a < b$
      4.1 If Array[a] + Array[b] = x then Return True
      4.2 If Array[a] + Array[b] $< x$ then a++

4.3 Else b–

5 Return False

When looking at the time complexity, the initial Mergesort will have a growth rate of O($n \cdot \log n$). Following this, the linear search will have a worst case of O(n - 1), since the worst case will be moving one of the endpoints all the way to the other end to the point just before the last spot. Therefore, the total time complexity will be the following:

O($n \cdot \log n$) + O($n + 1$) = $n \cdot (\log n + 1)$ - 1
Since constants become irrelevant as n gets sufficiently large,
$n \cdot (\log n) \rightarrow$ Therefore $\mathbf{O(n \cdot \log n)}$.

b. To solve this algorithm in O(n) expected time, we implement a hash data structure which has the a **O(1)** expected time complexity in the average case. This allows for instantaneous searching while also performing n queries. Going through the array A, calculate the sum - $A[i]$, and check to see if the hash already contains this number. If it does, then a pair has been found and the algorithm can exit, else the number that is being compared in the array is added to the hash, and the i value increments. As a result, the only time complexity that is of importance in the algorithm is the Array querying, which will be $\mathbf{O(n)}$ for the case of a full search of the array.

The algorithm is as follows:

searchPairsN(Array A, size, x)

1. Initialise hash H data structure to 0.
2. For each index i up to *size*:
   2.1 *complement* = x - *A[i]*
   2.2 IF *complement* >0 AND *H[complement]* is initialised
      2.2.1 Return True
   2.3 *Initialise H[A[i]]*
3. Return False

3 Question 3

You are at a party attended by n people (not including yourself), and you suspect that there might be a celebrity present. A celebrity is someone known by everyone, but does not know anyone except themselves. You may assume everyone knows themselves. Your task is to work out if there is a celebrity present, and if so, which of the n people present is a celebrity. To do so, you can ask a person X if they know another person Y (where

you choose X and Y when asking the question).
(a) Show that your task can always be accomplished by asking no more than 3n3 such questions, even in the worst case.
(b) Show that your task can always be accomplished by asking no more than $3n$-$\lfloor log 2n \rfloor$-2 such questions, even in the worst case.

a. The algorithm starts off by asking person X if they know person Y. This will be denoted as $X \rightarrow Y$. From this, there are two possible outcomes that can occur. If Person X knows Person Y, we are able to eliminate person X as celebrities know no one. Otherwise, we cross of person Y, since everyone should know a celebrity. This is then repeated down the n list of people. Since we do not ask a person if they know themselves, there will be one person remaining on the list if there is indeed a celebrity present, resulting in an initial time complexity of n - 1.

The next step to the algorithm is to check that all n-1 people know the potential celebrity. (This is n - 2 since we do not need to check if the potential celebrity knows themselves, but we also have the initial person who was asked, hence two less comparisons than n).

The final step of the algorithm is to check that the potential celebrity knows no one at the party. Again, it is unnecessary to ask the celebrity if they know themselves, so we simply ask n - 1 times.

Thus, we have a total time complexity of $(n - 1) + (n - 2) + (n - 1)$
$= \mathbf{3 \cdot n - 4}$.

Thus, we have the following algorithm: findCelebrity(n)

1 Pick X from the list of people who have not been chosen
2 Pick Y from the list of people who have not been chosen
3 WHILE remaining people >1
  3.1 If person X knows person Y:
    3.1.1 Eliminate person X
    3.1.2 Choose next person on list as X
  3.2 Else:
    3.2.1 Eliminate person Y
    3.2.2 Choose new person on list as Y
4 *potential* = X
5 For each remaining person on list excluding *potential*:
  5.1 If person does not know *potential*:
    5.1.1 Return False
6 For each remaining person on list excluding *potential*:

6.1 If *potential* knows person:
6.1.1 Return False
7 Return *potential*

b. Was not sure how to complete this one.

4 Question 4

Pair One By simplification of $g(n)$, we get:
$g(n) = log_2(n^{log_2 n}) + 2log_2 n = log_2 n \cdot log_2 n + 2log_2 n = (log_2 n)^2 + 2log_2 n$
Since $g(n) = f(n) + 2 \cdot log_2 n$, we can clearly see that g(n) is larger.
This results in f(n) = O(g(n)).

Pair Two We start by looking at $\frac{n^{100}}{2^{\frac{n}{100}}}$
By taking $\sqrt{100}$ of both sides, we end up with the result $\frac{n}{2^{\frac{n}{1000}}}$
This can be rewritten as $\frac{n}{(2^{\frac{1}{1000}})^n}$
Continuing by letting $a = 2^{\frac{1}{1000}}$, we can simplify to $\frac{n}{a^n}$
Consequently, performing L' Hopital's Rule on this: $\lim_{n\to\infty} \frac{n}{a^n} = 0$
We see that $f(n) < g(n)$ and therefore $f(n) = O(g(n))$

Pair Three Using L' Hopital's Rule again, we start with $\frac{f(n)}{g(n)} = \frac{n^{\frac{1}{2}}}{2^{\sqrt{log_2 n}}}$
Simplifying the fraction, we go through the following steps:

Taking $\log_2$ of the fraction: $\frac{log_2 n^{\frac{1}{2}}}{log_2 2^{\sqrt{log_2 n}}} = \frac{\frac{1}{2} log 2n}{\sqrt{log_2 n} \cdot log_2 2} = \frac{\frac{1}{2} log 2n}{\sqrt{log_2 n}}$
Rationalising the denominator: $\frac{\frac{1}{2} \cdot log_2 n \cdot \sqrt{log_2 n}}{log_2 n}$
And finally cancelling gives us $\frac{1}{2}\sqrt{log_2 n}$
Since $\lim_{n\to\infty} \frac{1}{2}\sqrt{log_2 n} = \infty$, we see that $f(n) > g(n)$
Therefore, $f(n) = \Omega(g(n))$

Pair Four Initially using L' Hopital's Rule, we get
$\frac{f(n)}{g(n)} = \frac{n^{1.001}}{n \cdot log_2 n} = \frac{n^{0.001}}{log_2 n}$
$\lim_{n\to\infty} \frac{n^{0.001}}{log_2 n} = \infty$
Therefore we look at $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$
$\frac{1.001 \cdot n^{0.001}}{\frac{\ln n + 1}{\ln 2}} = \frac{1.001 \cdot n^{0.001} \cdot \ln(2)}{\ln(n) + 1}$
Ignoring the coefficients, we are left with $\frac{n^{0.001}}{\ln(n)}$
From this, it can be seen that the graph of $n^{0.001}$ eventually overtakes $\ln n$, so therefore we have $f(n) = \Omega(g(n))$

Pair Five Using the L' Hopital Rule with *f(n)* and *g(n)*: $\frac{n^{\frac{1+sin(\frac{\pi \cdot n}{2})}{2}}}{n^{1/2}}$
Simplifying the fraction using index notation, this becomes: $n^{\frac{1+sin(\frac{\pi \cdot n}{2})}{2} - \frac{1}{2}}$

$= n^{sin(\frac{\pi \cdot n}{2})}$
As a result of the *sin* term in the power, the curve will oscillate and as such the limit cannot be defined.
Therefore, we get $f(n) = \theta(g(n))$

5 Question 5

a. $T(n) = 2T(n/2) + n(2 + sin(n))$
$n^{log_2 2} = n^1 = n$
Since the sin term oscillates between 1 and -1:
$1 \leq 2 + sin(n) \leq 3$
$n \leq n(2 + sin(n)) \leq 3n$
Since the c term is in between, case 2 is satisfied, thus $f(n) = \theta(n)$
Thus $T(n) = \theta(nlog_2 n)$

b. $T(n) = 2T(n/2) + \sqrt{(}n) + logn$
$n^{log_2 2} = n^1 = n$
$f(n) = O(\sqrt{n})$
Case 1 is satisfied, therefore $T(n) = \theta(n)$

c. $T(n) = 8T(n/2) + n^{logn}$
$n^{log_2 8} = n^3$
Case 3 applies, and so $T(n) = \theta(n^{logn})$

d. The master theorem does not apply here, so the recurrence must be unwound manually.
$T(n) = T(n-1) + n$
Following the recurrence down:
$T(n-1) = T(n-2) + (n-1), T(n-2) = T(n-3) + (n-2), T(n-2) = T(n-3) + (n-3)...T(1) = T(0) + 1$
Using the sum of series formula, we obtain $\frac{n(n+1)}{2} = \frac{n^2+n}{2}$
Taking n^2 as the fastest growing element, we thus get:
T(n) = $\theta(n^2)$