Kamal Giri(W10026697) and Ben Wilkin(W10015903)
Assignment 1
COSC 2030

1. Asymptotic Complexity

In the lectures, we said that logarithms with different bases don't affect the asymptotic complexity of an algorithm. Prove that O(log3 n) is the same as O(log13 n). Use the mathematical definition of O – do a formal proof, not just the intuition. Start by giving the formal definition of O. Total 5 points.

We have,

The definition of O:

$T(n) \in O(f(n)$ if there are positive constants c and n_0 such that $T(n) <= c\, f(n)$ for all $n >= n_0$

We have to prove that, different bases doesn't affect the asymptotic complexity

i.e. $O(\log_3 n)$ is same as $O(\log_{13} n)$.

Now,

According to the definition of O, we can write that,

Let, $T(n) = \log_{13} n$ and $f(n) = \log_3 n$

So, according to our definition, we can say that,

$\text{Log}_{13}\, n <= c \log_3 n$

For positive constant c and n_0 , $n>=n_0$.

Now, let say $c = \log_{13} 3$ as a constant

Now, substituting it back to the above equation,

We get,

$\text{Log}_{13} n <= \log_{13} 3 . \log_3 n$

$= \log_{13} n <= (\log_3 n) * \log_{13} 3$

$= \log_{13} n <= \log_{13} 3^{(\log_3 n)}$ //using logarithmic property

$= \log_{13} n <= \log_{13} n$ // since, $3^{(\log_3 n)} = n$

Kamal Giri(W10026697) and Ben Wilkin(W10015903)

Now, above equation is satisfied for any value of n >1.

We, can also use same method to prove the other way around as well talking about the general case

We can take the bases of log to be some variable i and j and we can prove that different bases don't matter the asymptotic relationship. We can choose the value of c which can make the any general inequality true for any n>1.

2 Runtime Analysis

Analyze the running time of the following recursive procedure as a function of n and find a tight big-O bound on the runtime for the function. You may assume that each operation takes unit time. You do not need to provide a formal proof, but you should show your work: at a minimum, show the recurrence relation you derive for the runtime of the code, and then how you solved the recurrence relation.

1. If n <= 1, return. T(1) = 1
2. The nested loops count up to n or n^2, doing a total of n^5 work
3. Call mystery(n/3) three times 3T(n/3)

Recurrence Relation

T(n) = 1 when n <=1

T(n) = 3T(n/3) + n^5 when n > 1

Solving

When i=1:

$T(n) = 3(3T(n/9 + (n/3)^5) + n$^5

$= 9T(n/9) + n^5/3^5 + n^5$

When i =2:

$=27T(n/27) + n^5/3^8 + n^5/3^4 + n^5$

Now, we can deduce that

$= 3^iT(n/3^i) + n^5(1/3^{4i} + 1/3^{4(i-1)} + \ldots\ldots + 1/3^8 + 1/3^4 + 1/3^0)$

We will to solve the series part first:

Kamal Giri(W10026697) and Ben Wilkin(W10015903)

$1/3^{4i} + 1/3^{4(i-1)} + \ldots\ldots + 1/3^8 + 1/3^4 + 1/3^0$

This can be expressed as :

$$\sum_{j=0}^{k} \frac{1}{3^{4i}}$$

Now, We will use the formula:

$$\sum_{k=0}^{n} a^i = \frac{1-a^i}{1-a}$$

So, we will have,

$\sum_{j=0}^{k} \frac{1}{3^{4i}} = \frac{1-\frac{1}{3^{4i}}}{1-1/3^4}$

Solving our original equation we have:

$3^iT(n/3^i) + n^5(1/3^{4i} + 1/3^{4(i-1)} + \ldots\ldots + 1/3^8 + 1/3^4 + 1/3^0)$

$=3^iT(n/3^i) + n^5 (\frac{1-\frac{1}{3^{4i}}}{1-1/3^4}$

Put, $i=\log_3 n$

We get,

$nT(1) + 81(n^5(1-\frac{1}{n^4}))/80$

$= n+ \frac{81(n^5-n)}{80}$

Hence, from the above equation, we can say that tight big -O bound is $O(n^5)$.

3 Sorting – Insertion Sort.

Kamal Giri(W10026697) and Ben Wilkin(W10015903)

Sort the list 4,3,1,0,-1,8 using insertion sort, ascending. Show the list after each outer loop. Do this manually, i.e. step through the algorithm yourself without a computer. Total 5 points.

Answer:

We have the code provided:

```
function insertionSort(arr) {
for(var i = 1; i < arr.length; i++) {
        var val = arr[i];
        var j;
        for(j = i; j > 0 && arr[j-1] > val; j--) {
        arr[j] = arr[j-1];
}
arr[j] = val;
}
}
```

Given input array

arr = [4,3,1,0,-1,8]

i. First loop when i=1, 1< arr.length(6); i++
   Val = arr[i] = 3;
   Var j = 1
   For(j= 1; 1>0 && 4 > 3; j = 1-1)
       Arr(1) = 4
   Arr(0) = 3

Hence, the array after first loop is [3,4,1, 0, -1, 8]

ii. Second loop when i=2; 2< 6; i=2+1
   Val= arr[2] = 1
   For(j = 2; j> 0 && 4>1 ; j=2-1)
       Arr(2) = arr(1) = 4
   And
   Arr(1) = 1

Now, our new array is
Arr = [3,1,4,0, -1, 8]
Now, inner loop will execute again
For(j = 1; j > 0 && arr(0)(3) > 1; j= 0)
    Arr(1) = arr(0) = 3
And
Arr(0) = val = 1;
Now, our inner loop will not execute because j will become 0 after above inner loop and j>0 or 0>0 condition is not satisfied for the inner loop to execute.
Hence, our outer loop after second loop is [1,3,4,0,-1,8]

iii. Third loop when i=3;
The code will run in similar fashion as above where the element in arr(4) is compared to all the elements left to it giving us the array
Arr = [0,1,3,4,-1,8]
After third loop execution

iv. Fourth loop when i=4;
In the fourth loop element of arr[5] is compared to all the element right of it giving us the Array
Arr = [-1,0,1,3,4,8]

v. Fifth loop when i=5 and i<6(T)
In the fidth loop, element of arr, arr[5] = 8 is compared to all the elements left of it and since 8 is the biggest element here, it will not change it place and we will get the new array
Arr = [-1, 0,1,3,4,8]

vi. Sixth loop when i=6; i<6
Now, the condition 6<6 is not true so our outer loop terminates and the program is complete.

So, our final array is [-1, 0,1,3,4,8]

4 Sorting – Merge Sort

Analyse the time complexity of your implementation and give a Θbound
for its worst-case runtime.
Total 13 points.

Answer: I have provided the time complexity of the code in the comment within my code. In case, if I need to have here, I will copy the text here:

Kamal Giri(W10026697) and Ben Wilkin(W10015903

Time Complexity of the Code:

We are using the iterative version of the merge sort in the above code.

We are solving the above problem through two way merge method where we merge the two sorted list.

This also make the merging in place.

So, essentially, what we are doing is taking each element of the array as the list and merging them

after sorting. When we visualize this:

```
    9  4  2  1  7  5  0  3
    \ / \ / \ / \ /
    4 9  1 2   5 7  0 3      1st pass
     \     /     \    /
      \    /      \   /
    1  2  4  9   0  3  5 7     2nd pass
        \            /
         \           /
    0  1  2  3  4  5  7  9     3rd pass
```

Now, In each pass, we are dealing with n number of elements and merging them as appropriate.

This is exactly done in our merge function where are working with n elements each pass.

Also, we can infer that this is inversed binary tree with height logn so we exactly doing the logn

number of passes for n number of input so,combining n merges with logn total passes, we can sat that

The complexity of the problem is O(nlogn)

Also, for worst case complexity is also Θ(nlogn) as we have to go through all the elements(compare) to merge

No matter, how the elements are sorted and we still have to go through logn passes to complete the merge so, the worst time complexity = Θ(nlogn).

Kamal Giri(W10026697) and Ben Wilkin(W10015903

5 Sorting – Quicksort

In the lectures I only briefly mentioned strategies for determining a good pivot for quicksort. The implementation on the slides simply picks the leftmost element in the part of the array that we consider as a pivot. I also mentioned a

few other ways of picking a good pivot, e.g. randomly.
Median-of-three is also a good way of picking a pivot – inspect the first, middle, and last elements of the part of the array under consideration and choose the median value. Using the probabilities for picking a pivot in a particular part of the array (in the same way as we did on slide 34), argue whether this method is more or less (or equally) likely to pick a good pivot compared to simply choosing the first element. Assume that all permutations are equally likely, i.e. the input array is ordered randomly.
Your answer must derive probabilities for choosing a good pivot and quantitatively reason with them.
Total 5 points.

With a randomly selected pivot, you have a 50% chance to get a 'good' pivot. With a median of three approach, that chance goes up to 68.75%. Let 0's represent bad choices for pivots and 1's represent good choices for pivots. If each randomly selected entry has a 50% chance to be selected, there are 8 equally likely scenarios

000-bad

001-50/50 bad or good

010-50/50 bad or good

011-good

100-50/50 bad or good

101-good

110-good

Kamal Giri(W10026697) and Ben Wilkin(W10015903

111-good

In the case where there are at least two good pivots out of three, the median pivot will be a good pivot. There are four of these, so they account for a 50% chance to select a good pivot. In the case where no good pivots are selected, the median will not be a good pivot (000). In the three other cases, the two bad pivots have a 50% chance to both be either low or high. This means that for the 001, 010, and 100 scenarios there is a 50% chance for a good pivot to be selected, which accounts for a +18.75% chance for a good pivot to be selected (12.5*3/2). This puts the overall chance for a good pivot at 68.75%.