[CS202]-Algorithms I **Fall 2020-21**

Homework: 02

Name: Gopal Ramesh Dahale RollNo: 11840520 email:gopald@iitbhilai.ac.in

Collaborators Names:

Solution of problem 1.
(a) Time complexity: Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
Example: Consider the problem of finding whether the given element x exist in the array A or not.

SEARCH(A, x)
1: **for** $i = 1$ **to** $A.length$ **do**
2: **if** $A[i]$ is equal to x **then**
3: **return true**
4: **return false**

If the i^{th} line takes c_i cost for execution then there can be two scenarios for analysing the time complexity of the above algorithm. Worst case running time of the algorithm deals with the maximum time taken by the algorithm. In this case it will happen if the element x is not present in the array A and the for loop will run for the entire length of the array (n). We can express the total execution time in the following form.

$$\begin{aligned} T(n) &= c_1(n+1) + c_2(n) * c_3(0) + c_4(1) \\ &= n(c_1 + c_2) + c_1 + c_4 \end{aligned}$$

Note that line 3 will not execute if the element x is not present in the array.
Another scenario is of the best case running time of an algorithm which deals with the minimum time taken by the algorithm to execute. In this case it will happen if the element x is present first in the array. Total time of execution will be:

$$\begin{aligned} T(n) &= c_1(1) + c_2(1) + c_3(1) \\ &= c_1 + c_2 + c_3 \end{aligned}$$

Both the cases (Best and Worst) depends on the type of input.

Solution of problem 2.
(a) $T(n) = 16T(n/2) + n^3$

Proof.

$$n^{\log_b a} = n^{\log_2 16} = n^4$$

$$f(n) = n^3$$

$$f(n) = O(n^{4-\epsilon})$$

for $\epsilon = 1$

$$\therefore \; T(n) = \Theta(n^4)$$

□

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

Proof.

$$n^{\log_b a} = n^{\log_2 4} = n^2$$
$$f(n) = n^2$$

1

$$f(n) = \Theta(n^2)$$
$$\therefore \;\; T(n) = \Theta(n^2 \log n)$$

□

(c) $T(n) = 3T(n/2) + n^2$

Proof.

$$n^{\log_b a} = n^{\log_2 3} = n^{1.58}$$
$$f(n) = n^2$$
$$f(n) = \Omega(n^{1.58+\epsilon})$$

for $\epsilon = 0.02$. Also

$$af(n/b) = 3\left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2 \leq cn^2 = cf(n)$$

for $c = 1$.

$$\therefore \;\; T(n) = \Theta(n^2)$$

□

(d) $T(n) = 2T(n/4) + n^{0.58}$

Proof.

$$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$$
$$f(n) = n^{0.58}$$
$$f(n) = \Omega(n^{0.5+\epsilon})$$

for $\epsilon = 0.02$. Also

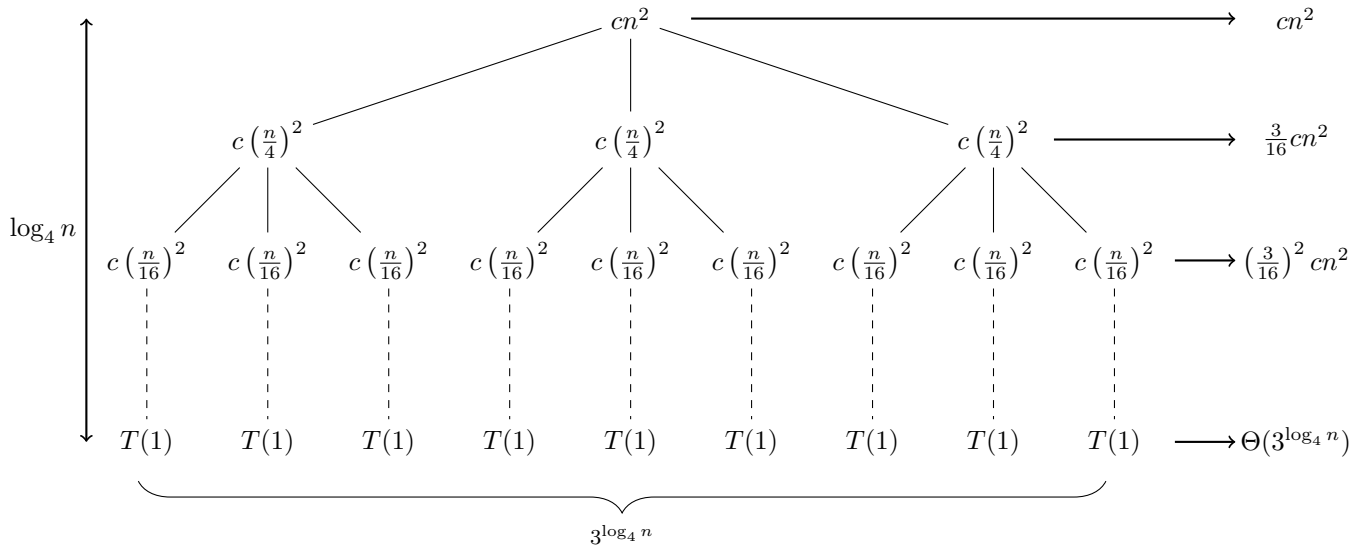
$$af(n/b) = 2\left(\frac{n}{4}\right)^{0.58} = \frac{2}{4^{0.58}}n^{0.58} \leq \frac{2}{4^{0.5}}n^{0.58} = cf(n)$$

for $c = \dfrac{2}{4^{0.5}}$.

$$\therefore \;\; T(n) = \Theta(n^{0.58})$$

□

Solution of problem 3.
(a) Let $\Theta(n^2) = cn^2$ for some $c > 0$. The solution assumes that n is an exact power of 4.



Proof. Total cost

$$\begin{aligned}
&cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(3^{\log_4 n}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}$$

□

(b) $T(n) = 8T(n/2) + n^3 \log n$

Proof.

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

$$f(n) = n^3 \log n$$

$$f(n) = \Omega(n^3)$$

but

$$f(n) \neq \Omega(n^{3+\epsilon})$$

for any $\epsilon > 0$. That is $f(n)$ is not polynomially larger. Consider the ratio

$$\frac{f(n)}{n^{log_b a}} = \frac{n^3 \log n}{n^3} = \log n$$

The ratio is asymptotically less than n^ϵ for any $\epsilon > 0$. Therefore the master method cannot be applied here.

$$T(n) = 8T(n/2) + n^3 \log n$$

$$T(\frac{n}{2}) = 8T(\frac{n}{2^2}) + \left(\frac{n}{2}\right)^3 \log \frac{n}{2}$$

$$\implies T(n) = 8^2 T(\frac{n}{2^2}) + n^3 \left(\log \frac{n}{2} + \log n\right)$$

Consider $T(\frac{n}{2^2})$

$$T(\frac{n}{2^2}) = 8T(\frac{n}{2^3}) + \left(\frac{n}{2^2}\right)^3 \log \frac{n}{2^2}$$

$$\implies T(n) = 8^3 T(\frac{n}{2^3}) + n^3 \left(\log \frac{n}{2^2} + \log \frac{n}{2} + \log n\right)$$

By observation we can write $T(n)$ after k iterations.

$$T(n) = 8^{K+1} T(\frac{n}{2^{k+1}}) + n^3 \left(\log \frac{n}{2^k} + \log \frac{n}{2^{k-1}} + \cdots + \log \frac{n}{2} + \log n\right)$$

For base case i.e. $T(1) = 1$, we put $\frac{n}{2^{k+1}} = 1$

$$n = 2^{k+1}$$

$$k = \log n - 1$$

$$\implies T(n) = 8^{k+1}(1) + n^3 \left((k+1)\log n + k + k - 1 + \cdots + 1 + 0\right)$$

$$T(n) = 8^{k+1}(1) + n^3 \left((k+1)\log n + \frac{k(k+1)}{2}\right)$$

$$T(n) = 8^{k+1}(1) + n^3 \left((\log n)\log n + \frac{(\log n - 1)(\log n)}{2}\right)$$

$$\implies T(n) = n^3 \log^2 n$$

□

Solution of problem 4.
Since the given input is sorted in non-increasing order. For any i^{th} paper if the number of citations is greater than the value of i. It means at least i papers have at least i citations each because all the values at indices which are less than i are greater than the value at index i because the array is sorted. We further have to look for elements after i^{th} index.

If the number of citations at index i is less than the value of i it means that after i^{th} index the value of index will always be greater than number of citations. It means we have to stop and the answer lies in the left half of i i.e 0 *to* i. The idea is to use binary search here. We look at the middle of the array and compare the index with its value. If the

value of the element is greater than (or equal to) the index then we know that the answer may lie in the right half. We store the current index in h as the current $h\ index$.

If the value of the element is less than the index value then we know that the answer lies in the left half for sure.

HINDEX(A, x)
1: $low = 1$
2: $high = A.length$
3: $h = 0$
4: **while** $low <= high$ **do**
5: $mid = (low + high)/2$
6: **if** $A[mid] >= mid$ **then**
7: $low = mid + 1$
8: $h = mid + 1$
9: **else**
10: $high = mid + 1$
11: **return** h

Asymptotic analysis (rough idea):

At every iteration the length of the array(n) gets halved and the loop will terminate after k iterations when $\frac{n}{2^k} = 1$.

$$\begin{aligned} n &= 2^k \\ k &= \log n \\ \implies T(n) &= \Theta(\log n) \end{aligned}$$

Solution of problem 5.
(a) In brute force algorithm, for each element of A we will find its absolute difference with each element of B and update the minimum absolute difference if is it greater than the currently calculated value. In this way we can find the minimum absolute difference.
We again do the same but this time we check whether absolute difference of the elements is equal the absolute minimum difference. If it is, then we print them.
(b)

MINIMUM_ABSOLUTE_DIFFERENCE(A, B, n)
1: // sort A and B in ascending order
2: $sort(A)$
3: $sort(B)$
4: $i = 1$
5: $j = 1$
6: $min = \infty$
7: $diff = 0$
8: **while** $i < n$ **and** $j < n$ **do**
9: $diff = \boldsymbol{abs}(A[i] - B[j])$
10: **if** $min > diff$ **then**
11: $min = diff$
12: **if** $A[i] < B[j]$ **then**
13: i++
14: **else**
15: j++
16: // Once the minimum absolute difference is found
17: // We again use the same way to find the pairs

18: // We check for the elements whose $diff$ is equal to min
19: $i = 1$
20: $j = 1$
21: **while** $i < n$ **and** $j < n$ **do**
22: $diff = \boldsymbol{abs}(A[i] - B[j])$
23: **if** $min == diff$ **then**
24: **print** $A[i], B[j]$
25: **if** $A[i] < B[j]$ **then**
26: i++
27: **else**
28: j++

The strategy is to check whether the difference of the current two elements $diff$ is less than min. If it is, then we update the min. After that we increment either i if $A[i] < B[j]$ else we increment j. This is correct.
Consider the case when $A[i] < B[j]$ and we increment j. Initially the absolute difference is $|A[i] - B[j]|$. Since the arrays are sorted $B[j+1] > B[j]$ then the difference $|A[i] - B[j+1]|$ will definitely increase which we don't want. Increment i in this case tends to decrease the difference.
Proof of correctness:
We show proof of correctness from line 4 to 15.
Loop Invariant : At the start of each iteration of the while loop min contains the minimum absolute difference of $A[1 \dots i-1]$ and $B[1 \dots j-1]$.
Initialisation: Prior to first iteration of the loop, we have i = 1 and j = 1. $A[1 \dots i-1]$ and $B[1 \dots j-1]$ contains zero elements. Therefore min is set to infinity to say that the minimum absolute difference of zero elements is infinity.
Maintenance: Assume Loop Invariant to be true before an iteration such that min contains the minimum absolute difference of $A[1 \dots i-1]$ and $B[1 \dots j-1]$. There are two cases.
Case 1: $diff = abs(A[i] - B[j])$ is the minimum absolute difference of $A[1 \dots i]$ and $B[1 \dots j]$. Then in lines 10 and 11 min is set to diff.
Case 2: $diff = abs(A[i] - B[j])$ is not the minimum absolute difference then the minimum absolute difference of $A[1 \dots i]$ and $B[1 \dots j]$ lies in $A[1 \dots i-1]$ and $B[1 \dots j-1]$. By our assumption min already has this value and is unchanged in this iteration.
Incrementing i (if $A[i] < B[j]$) or j (if $A[i] >= B[j]$) re establishes the loop invariant.
Termination: At $i = n+1$ or $j = n+1$ the loop terminates. By loop invariant min has the minimum absolute difference of $A[1 \dots n]$ and $B[1 \dots n]$. If the loop terminates when $i < j \implies A[i] > B[j]$. Even if we consider the difference of $A[i+1]$ and $B[j]$ it will certainly be greater the previous one because $A[i] < A[i+1]$. The same holds when $i > j$.
For lines 19 to 28, it only prints the elements which have their absolute difference equal to min in the same fashion.
Running time of the algorithm:
We can sort the arrays in at least $n \log n$ time. In the worst case the while loop (line 8) runs for n+1 times and same goes for the while loop (line 21). Since all other operations inside the while loop are of constant time, both loops will take approximately $2n$ time.
Therefore the running time will be of the order of $O(n \log n + n \log n + 2n)$ = $O(n \log n)$