Graduate Algorithms HW 1

Piyush Hinduja

September 2023

1 Answer 1

Q-1) a) Yes.
We know that $T(n) = O(f(n))$ iff for some constants c and n0, $T(n) \leq c * f(n)$ for all $n \geq n_0$.
Informally, it means Growth rate of T(n) $\leq$ Growth rate of f(n).
Hence, $f(n) \in O(n^2)$ then f(n) can be anything greater than or equal to n^2 i.e. n^2, $n^2 * \log n$, n^3, 2^n, and so on.

Q-1) b) Cannot say.
We know that $T(n) = o(f(n))$ iff for some constants c and n0, $T(n) < c * f(n)$ for all $n \geq n_0$.
Informally, it means Growth rate of T(n) <(Strictly less than) Growth rate of f(n).
Because if $f(n) = n^2 * \log n$ then the statement stands true i.e. it has $\Omega(n^2)$ and $o(n^3)$.
But if $f(n) = n^3$ or anything greater then $o(n^3)$ doesn't hold true.

Q-1) c) Yes.
For smaller values of 'n' (till 15), this statement is false but after that it holds true and we know that exponential time complexity is the highest one and $f(n) = n^{\log_2 n}$ is a polynomial form of time complexity and $o(2^n)$ is an exponential form.
Also while considering time complexities we majorly focus on large inputs (or worst cases).

2 Answer 2

Q-2) a) Lets have a look on the bubble sort algorithms once,

```
function bubble(int[] arr):
    for(i in arr):
        for(j in arr):
            if(arr[j] > arr[j+1]):
                swap
```

Here we can observe that there are two nested for loops on same array of size 'n' so the algorithm's time complexity becomes $n * n = n^2$.
Now we take an integer 'k' such that 1¡k¡n and aim to reduce the complexity from n^2 to $n * k$.

Solution:
If we give the algorithm an array such that it's $n - k$ elements are sorted, and we notice that in a pass, we can stop it there itself.
Example: Consider the array A=[6, -5, 0, 7, 10, 15, 20, 25].
Here we can observe that later 4 (n-k) elements of the array are sorted.
So if the algorithm runs n^2 times here too, it will be waste of efficiency.
Hence, we can introduce a boolean checker which checks for swaps at each pass.
If no swap is done in a particular pass, we can stop the loops and return the array.
Hence, the modified running time will become

$$(n - (n - k)) * n$$

i.e. we will reduce the time of sorted part from 'n' which is $n - k$.
Therefore, final complexity will become:

$$O(n * k)$$

Modified algorithm will be:

```
function modified_bubble(int[] arr):
    for(i in arr):
    swapped = false
        for(j in arr):
            if(arr[j] > arr[j+1]):
                swap
                swapped = true
    if(swapped is false): break
```

Q-2) b) In Binary search, our search space gets split in half during each iteration so we can express it's time complexity as,

$$T(n) = T(\frac{n}{2})$$

$$= T(\frac{n}{4})$$
$$= T(\frac{n}{8})$$
$$\cdots\cdots\cdots$$
$$= T(\frac{n}{2^r})$$

Since 'r' is a constant we can ignore it.
Also we know that base case for binary search is T(1) = 1.
Hence we get,

$$\frac{n}{2^r} = 1$$
$$n = 2^r$$
$$i.e. r = \log_2(n)$$

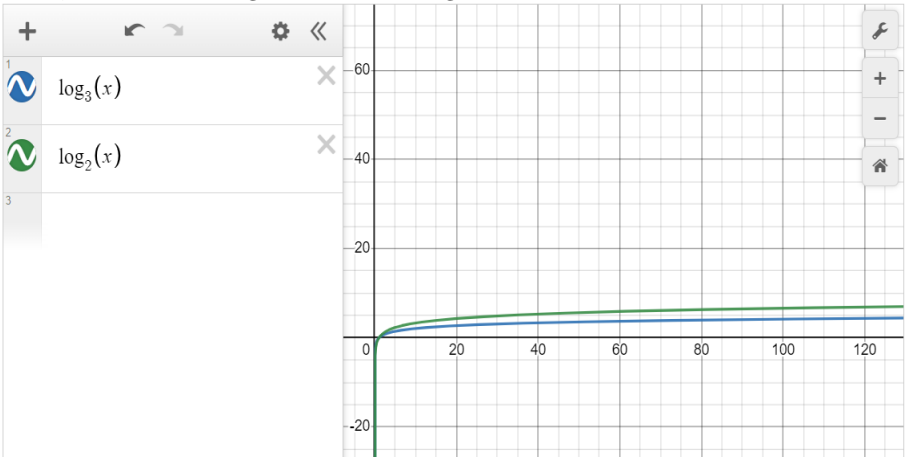
Now, if we divide the search space in 3 parts rather than 2, we get,

$$T(n) = T(\frac{n}{3})$$
$$= T(\frac{n}{9})$$
$$= T(\frac{n}{27})$$
$$\cdots\cdots\cdots$$
$$= T\left(\frac{n}{3^r}\right)$$

Solving similarly by taking base case as T(1) = 1 as above,

$$n = 3^r$$
$$i.e. r = log_3(n)$$

Both $log_2(n)$ and $log_3(n)$ belong to log class and we can note from the below image that $log_3(n)$ works a little bit better than $log_2(n)$. But as the input size increases, this difference goes on increasing.



Hence, in case of large input size, $log_3(n)$ works better than $log_2(n)$.

3 Answer 3

While storing a set of numbers in their binary forms, each node of our prefix tree will have 2 children (0 and 1). We can keep the root node as 1 since we need to store numbers in the range $[1, N^3]$, we will not have 0 as an input.

To store values, we will need to start from the root node and keep adding the bits as nodes (if the tree does not contain that bits) and finally on completion we can put a check mark on the final node or bit of the number.

Hence to store a n-bit number we will require n-nodes and number of bits in binary representation vary from number to number.

We can calculate n using formula,

$$n = log_2(x)$$

Where 'x' represents the decimal form of the number and 'n' represents the number of bits required to represent it in binary form.

Sub-questions:

a) To add a new integer, in worst case, we will have to add 'n' new bits to the tree and check mark it on completion.
These new bits will be equal to the length of binary representation of these integers i.e., $\log_2(x)$.
Hence, complexity will be $O(\log_2(n))$.

b) Similarly, to query an integer 'x', we will require to travel that number of nodes and then check for the check mark. So, the complexity here will also be $O(\log_2(n))$.

Bonus: As far as I understand trees and tries, tries are better to store a set of strings or integers when we need to frequently query stuff. But if we want to store a set of integers in sorted manner, the complexity of Prefix trees can go really high. Hence, it is beneficial for us to use BST.

4 Answer 4

Q-4) a) Akra-Bazzi theorem:

$$T(n) = 2 * T(\frac{n}{2}) + T(\frac{n}{3}) + n$$

Comparing the equation with,

$$T(n) = a1 * T(b1 * n) + a2 * T(b2 * n) + g(n)$$

We get,$a1 = 2, a2 = 1, b1 = 1/2, b2 = 1/3$ and $g(n) = n$
Substituting the above values in,

$$a1 * (b1)^p + a2 * (b2)^p = 1$$

We get,

$$p = 1.365$$

Now, we know $\theta(n)$ formula by Akra-Bazzi, i.e.,

$$T(n) = \theta(n^p + n^p \int_1^n \frac{g(u))}{u^{p+1}} du)$$

Substituting values of p and g(n) in above equation,

$$T(n) = \theta(n^{1.36} + n^{1.36} \int_1^n \frac{u}{u^{2.36}} du)$$

$$= \theta(n^{1.36} + n^{1.36} \int_1^n \frac{1}{u^{1}.36} du)$$

$$= \theta(n^{1.36} + n^{1.36} * [\frac{2.7778}{u^{0.36}}]_1^n)$$

$$= \theta(n^{1.36} + n^{1.36} * [\frac{2.7778}{n^{0.36}} - 2.7778])$$

$$= \theta(\frac{n^{1.36}}{n^{0.36}}(n^{0.36} + 2.7778 * (1 - n^{0.36})))$$

$$= \theta(n * (n^{0.36} + 2.7778 - 2.7778 * n^{0.36}))$$

$$= \theta(n * (2.7778 - 1.7778 * n^{0.36}))$$

$$\approx \theta(n)$$

We can prove the correctness by putting the given base conditions I above raw form,

$$T(n) = \theta(n * (2.7778 - 1.7778 * n^{0.36}))$$

$$T(0) = \theta(0 * (2.7778 - 1.7778 * 0^{0.36})) = \theta(0) = 0$$

$$T(1) = \theta(1 * (2.7778 - 1.7778 * 1^{0.36})) = \theta(1) = 1$$

Q-4) b)

$$T(n) = n * T(\sqrt{n})^2$$

We will start solving this problem using Plug and chug and derive the recurrence relation:

$$T(n) = n * T(\sqrt{n}) * T(\sqrt{n})$$
$$T(n) = n * [\sqrt{n} * T(\sqrt[4]{n}) * T(\sqrt[4]{n})]^2$$
$$T(n) = n^2 * T(\sqrt[4]{n})^4$$
$$T(n) = n^2 * [\sqrt[4]{n} * T(\sqrt[8]{n}) * T(\sqrt[8]{n})]^4$$
$$T(n) = n^3 * T(\sqrt[8]{n})^8$$
$$.....$$
$$T(n) = n^r * T(n^{\frac{1}{2^r}})^{2^r}$$

Now in general problems we used to use the base cases to get the value of 'r' and then proceed further, but this is a special case.
Here we will use T(2) to get 'r' and then we can find time complexity.
We can also use any other number ¿ 1 but we will go with the closest of 1 since T(1) is known.
Substituting $n^{\frac{1}{2^r}} = 2$,

$$Takinglogonboththesides,$$
$$\frac{1}{2^r} * \log_2 n = \log_2 2 = 1$$
$$\log_2 n = 2^r$$
$$r = \log\log n$$

Substituting this value of r in recurrence relation,

$$T(n) = n^{\log\log n} * T(2)^{2^{\log\log n}}$$

We can find the value of T(2) by using the T(1) value as follows,
We know,

$$T(n) = n * T(\sqrt{n})^2$$
$$T(2) = 2 * T(\sqrt{2})^2$$
$$T(2) = 2 * T(1.414)^2$$

Since, $1 \approx 1.414$ & $T(1) = 4$

$$T(2) = 2 * 4^2 = 32$$

We get,

$$T(n) = n^{\log\log n} * T(2)^{2^{\log\log n}}$$

$$T(n) = n^{\log\log n} * 32^{2^{\log\log n}}$$

$$Using formula, 2^{log_2 n} = n$$

$$T(n) = n^{\log\log n} * 32^{\log n}$$

$$T(n) = n^{\log\log n} * 2^5 * \log n$$

$$T(n) = n^{\log\log n} * 2 * \log n^5$$

Using formula,

$$2^{\log_2 n} = n$$

$$T(n) = n^{\log\log n} * n^5$$

We are also given, $T(1) = 16$

$$T(2) = 2 * 16^2 = 512$$

We get,

$$T(n) = n^{\log\log n} * T(2)^{2^{\log\log n}}$$

$$T(n) = n^{\log\log n} * 512^{2^{\log\log n}}$$

$$Using formula, 2^{log_2 n} = n$$

$$T(n) = n^{\log\log n} * 512^{\log n}$$

$$T(n) = n^{\log\log n} * 2^9 * \log n$$

$$T(n) = n^{\log\log n} * 2 * \log n^9$$

Using formula,

$$2^{\log_2 n} = n$$

$$T(n) = n^{\log\log n} * n^9$$

Q-4) c) We are given that in a divide and conquer procedure, divide step is O(n) for 'n' elements and the conquer step is proportional to the product of the sizes of sub-problems being combined.

Hence, if we divide a problem into $\frac{n}{2}$ and $\frac{n}{2}$, we can express its time complexity as:

$$T(n) = [2T(\frac{n}{2})] + [(\frac{n}{2}) * (\frac{n}{2})]$$

$$T(n) = [2T(\frac{n}{2})] + [(\frac{n^2}{4})]$$

Solving by Akra bazzi, Comparing the equation with,

$$T(n) = a1 * T(b1 * n) + g(n)$$

We get,$a1 = 2, b1 = 1/2$ and $g(n) = \frac{n^2}{4}n^2$ Substituting the above values in,

$$a1 * (b1)^p + a2 * (b2)^p = 1$$

We get,

$$p = 1$$

Now, we know $\theta(n)$ formula by Akra-Bazzi, i.e.,

$$T(n) = \theta(n^p + n^p \int_1^n \frac{g(u))}{u^{p+1}} du)$$

Substituting values of p and g(n) in above equation,

$$T(n) = \theta(n + n \int_1^n 4 * \frac{u^2)}{u^2} du)$$

$$T(n) = \theta(n + 4 * n * (n-1))$$

$$T(n) = \theta(4 * n^2 + n - 4)$$

$$T(n) = O(n^2)$$

i) If we divide the sub-problem into two parts of $\frac{n}{4}$ and $\frac{3n}{4}$, the,

$$T(n) = [T(\frac{n}{4}) + T(\frac{3n}{4})] + [(\frac{n}{2}) * (\frac{n}{2})]$$

$$T(n) = [2T(n/2)] + [(\frac{n^2}{4})]$$

$$T(n) = O(n^2)$$

We can observe that this kind of split also has n^2 complexity, so it wouldn't matter until its 2 sub-problems.
ii) If we divide the sub-problem into 3 parts of $\frac{n}{3}$:

$$T(n) = [3T(\frac{n}{3})] + [(\frac{n}{3}) * (\frac{n}{3}) * (\frac{n}{3})]$$

$$T(n) = [3T(\frac{n}{3})] + [(\frac{n^3}{9})]$$

Solving this too by Akra Bazzi ($p = 1$),

$$T(n) = \theta(n + n \int_1^n 9 * \frac{u^3)}{u^2} du)$$

$$T(n) = \theta(n + n \int_1^n 9 * u * du)$$

$$T(n) = \theta(n + \frac{9}{2} * n * n^2)$$

$$T(n) = \theta(\frac{9}{2} n^3 + n)$$

$$T(n) = O(n^3)$$

Since, the time complexity increased from n^2 to n^3, it is better to have 2 equal sized splits rather than 3.

5 Answer 5

In this approach, we store a set of numbers along with each word which indicates the index number of documents in which they are present.
Then we find the intersection of all the query words and search or return that documents.

In this (Inverted index) approach, taking the intersection by naive approach takes $O(s1 + s2 + ... + st)$ time for 't' words.
But we can improve it's efficiency by starting with the query word having the smallest set and then continue by (binary) searching its elements in other sets rather than some big set and traversing it's elements throughout other sets.

Pseudo code:
function(modified_invertedindex):

1) s = min(s1, s2, s3, ..., sn)
2) Sort all the other sets in increasing order of their lengths
3) Perform Binary Search one by one and keep discarding elements if they are not present in even one of the sets.
4) Return the final remaining list of documents

We know that Binary search takes log(n) time and hence we can say that total time to find intersection of all the sets using Binary Search given that we start from the set having smallest length i.e. 's' is:

$$O(s(log(s1) + log(s2) + ... + log(st)))$$

6 Answer 6

Let's assume that we have an array of initial size 100 and we start adding elements to it. What do we do once it gets filled up?

One method is, whenever our array is filled and we need to insert a new element then we make a new array of double size and repeat this process till the end.
But in this method, after a few double operations, the size of array becomes very large and if we are left with only a few or even 1 element and our array is full, we will have to make an array of double size for that 1 or few elements.

So to avoid this waste of memory, what we can do is instead of doubling the array, we can increase it's size by some constant spaces (of same size each time) every time it is filled.

For example, every time the array is full we can increase its space by 100 i.e. we can make a new array of n+100 size and copy all the elements from previous array and start inserting in a new one.

Time complexity for this later approach can be written as follows:

$$T(n) = [(0+100)+(100+100)+(200+100)+(300+100)+.....+(n+100)]+[n*1]$$

In the above formula, first square bracket is the time taken to make new arrays after each one is filled and each new array has 100 more spaces than its previous one.
Second square bracket indicates time taken to add 'n' elements.

Further simplifying the above formula,

$$T(n) = [100 + 200 + 300 + 400 + ... + (n + 100)] + n$$

$$T(n) = 100(1 + 2 + 3 + ... + n) + n$$

$$T(n) = \frac{n * (n + 1)}{2} + n$$

Since we need to find amortized (average) time for 'n' additions, we can divide the above equation by n,

$$T(n) = \frac{n + 1}{2} + 1$$

$$\approx O(n)$$

7 References

1) Khan Academy

2) Stack Exchange

3) Towards Data Science

4) Kunal Kushwaha - YouTube

5) Lars Quentin - YouTube

6) Gaurav Sen - YouTube