# Part 1:

**GTU Department of Computer Engineering CSE 222/505 - Spring 2021**

**Homework 2**

I colorized methods in the other method to make it easier to follow as you can see in the screen shots.

1. Searching a product.



In searchProduct method, I used findProduct method of the branch. It basically takes all the branches and search for product in them. Let m be the number of the branches and n be the number of products in the branches.

In best case, product can be found in the first branch’s first product and method can return the branch num of the branch and it takes constant time **θ(1).**

We must analyze the worst case now.

In findProduct method, we can’t find the product in the ith branch and loop n times. Time complexity is **θ(n) in this case.**

In searchProduct method, we can’t find the product in any branches and loop m times. Time complexity is **θ(m) in this case.**

Since they are nested loops, **Tworst(m,n) = θ(m\*n) and Tbest(m,n) = θ(1).**

**We can also say T(m,n) = O(mn).**

1. Add/remove product.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

In my addProduct method “getBranch().getAllProducts().add(add);” line does the adding operation. Other lines are for keeping data in a file and their time complexity is **θ(1), constant time.**

getBranch() and getAllProducts() also takes constant time, **θ(1).** So, I have to analyze add() method.

(n is the size of our array.)

In add(), E is Product in this case. It can’t return in first line because it is instance of Product. But it will call contains method.

contains’s complexity is constant time **θ(1) if Product is found in index 0 and return true in the best case. In worst case it does not contain, or Product is in the nth index and loop whole array n times, so worst time complexity is θ(n). So we can say Tcontains(n) = O(n).**

Then next 2 lines take constant time. After that we have if condition. If we enter the condition, changeCapacity’s time complexity is **θ(n)** because of the loop. Then we have another loop in the if condition and its time complexity is also **θ(n)** because of the loop. We add running times in consequtive statements so, general **T1(n) = θ(n+n) = = θ(n). (if condition)**

So, for the add method, in worst case array is out of capacity and it must change its capacity with its double. In this case, it enters the if condition. **Tw(n) = θ(n).**

In best case it just adds the product in the end and capacity is not full. It won’t enter the if condition. And adding in the best case **Tb(n) = θ(1). In general, we can say T(n) = O(n).**

**Thus, addProduct method’s T(n) = O(n). Again Tb(n) = θ(1) and Tw(n) = θ(n). Also we can say adding is amortized constant time.**



In my removeProduct method, I again call a method of my own container.

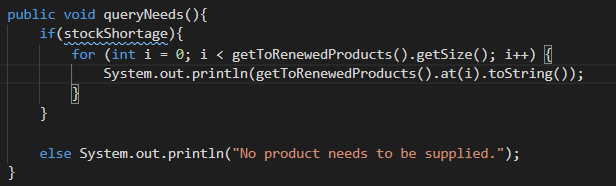
In remove method, getting the index of the to-be-removed Product takes **θ(n) time.** Shifting the array is also takes **θ(n) time. So, remove method’s time complexity is θ(n).**

**Thus, removeProduct method’s T(n) = θ(n) also.**

1. Querying the products that need to be supplied.



Employees make Boolean stockShortage true when any product needs to be supplied and add the product into toRenewedProducts. With this information,



queryNeeds function just prints if stockShortage is false, which is our best case in terms of time complexity. **Tb(n) = θ(1). (n is the size of the toRenewedProducts array)**

In our worst case we enter the loop and run n times. **Tw(n) = θ(n).**

**In general case, T(n) = O(n).**

# Part 2:

1. Explain why it is meaningless to say: “The running time of algorithm A is at least O(n2)”.

We use big-O notation for asymptotic upper bounds, O(n2) is upper bound of the function. So, it is meaningless to say **at least.** Running time of the algorithm A can at most O(n2). Because f(n) should be equal or smaller than n2**.**

1. Let f(n) and g(n) be non-decreasing and non-negative functions. Prove or disprove

that: max(f (n), g(n)) = Θ(f(n) + g(n)).

To prove max(f (n), g(n)) = Θ(f(n) + g(n)) we must also prove max(f (n), g(n)) = O(f(n) + g(n)) and max(f (n), g(n)) = Ω (f(n) + g(n)) by the definition of the theta that I state in part c.

Let’s start with max(f (n), g(n)) = O(f(n) + g(n)). Let max(f (n), g(n)) be t(n) and (f(n) + g(n)) be h(n).

**So, we must prove t(n) = O(h(n)).**

We can write inequality like, t(n) ≤ f(n) or t(n) ≤ g(n), because maximum of them can be smaller or equal to f(n) and g(n).

If we sum these we get, 2\*t(n) ≤ g(n) + f(n) = h(n).

We can also write it as, If we compare what we got and the definition of big-oh, we can see it is basically same ½ is some constant.

**Now, we must prove t(n) = Ω(h(n)).**

We can write t(n) like t(n) ≥ f(n) and t(n) ≥ g(n) too because maximum of them can be greater or equal to f(n) and g(n). If we apply same steps we get, And it is like the omega definition.

**Since we proved max(f (n), g(n)) = O(f(n) + g(n)) and max(f (n), g(n)) = Ω (f(n) + g(n)) then max(f (n), g(n)) = Θ(f(n) + g(n)) is also proved.**

1. Are the following true? Prove your answer.

* **T(N) = θ(h(N)) if and only if T(N) = O(h(N)) and T(N) = Ω(h(N)).**
* **T(N) = O(f(N)) if there are positive constants c and n0 such that T(N) ≤ c f(N) when N ≥n0.**
* **T(N) = Ω(f(N)) if there are positive constants c and n0 such that T(N) ≥ c f(N) when N≥ n0.**

I will use these definitions to prove.

* 1. 2n+1 = Θ(2n)

T(n) = Θ(2n) iff T(n) = O(2n) and T(n) = Ω(2n).

Firstly, we can check big-oh notation. We should find a positive constant c and n0 that meet the requirements. We can choose c = 2 and n0 = 0 and we can see, 2n+1 ≤ 2.2n = 2n+1 for all n ≥ 0. Thus, we can say, 2n+1 = O(2n).

Secondly, we should check omega notation. We can choose c = 1 and n0 = 0 and we can see, 2n+1 ≥ 1.2n  for all n ≥ 0. Thus, we can say 2n+1 = Ω (2n).

**Since, 2n+1 = O(2n) and 2n+1 = Ω(2n), then 2n+1 = Θ(2n).**

We can also prove with the limit rules. If the solution of the limit is c ∈ R ≠ 0 then f(n)=Θ(g(n)).

= 🡪 L’Hospital Rule 🡪 = 2 🡪 ≠ 0.

2n+1 = Θ(2n) proved again. **This statement is true.**

* 1. 22n = Θ(2n)

We should find c1 and c2 such that c1.2n≤22n≤c2.2n just like the Ist problem. If we look at the part that we try to prove it is O(2n). 22n≤c2.2n = 4n ≤c2.2n we can see that there is no such c2 and n0 that meet the requirements for all n≥ n0. **So, 22n  ≠** **O(2n).**

**Since, the definition of the theta notation stated if 22n  ≠** **O(2n) then 22n ≠** **Θ(2n). Thus, this statement is false.**

* 1. Let f(n)=O(n2) and g(n)= Θ(n2). Prove or disprove that: f(n) \* g(n) = Θ(n4).

By the big-oh definition we can say, f(n) ≤ c1.n2 for n ≥ n0.

Also, by the theta definition we can say, c1. n2≤ g(n) ≤c3.n2 for n ≥ n0.

If we multiply these inequalities, c1. n2 ≤ f(n) \* g(n) ≤ c1.c3.n4

As we can see, right-side of the inequality proved that f(n) \* g(n) ⊂ O(n4), but we can’t say same thing about the left-side.

**Thus, this statement is false. We can’t directly say f(n) \* g(n) = Θ(n4). We can only say f(n) \* g(n)** ⊂ **O(n4).**

# Part 3:

List the following functions according to their order of growth by explaining your assertions. n1.01, nlog2n, 2n, √n, (log n)3, n2n, 3n, 2n+1, 5 log n, log(n)

2



I used graph plotting application to see general behavior of the functions. As we can see, exponential functions grow a lot faster than others. Polynomials and then logarithmic functions next in the order.

Now I will use limit to compare their grow rate between each type of function.

f(x) grows faster than g(x) if = ∞.

If solution is constant other than 0, f(x) and g(x)’s grow rate is same. If it is 0, then g(x) grow faster than f(x).

|  |  |
| --- | --- |
| = ∞ 🡪 | 🡪 |
| = 0 🡪 log(n) < nlog2n | = ∞  🡪 (log(n))3 < nlog2n |
| = ∞ 🡪 | = ∞ 🡪 |
| 🡪 | 🡪 |

is an exponential but its power is logarithmic. So, among all the exponentials, it’s grow rate is smallest.

So, with the help of the graph and my limit comparison results, I can list them as follow:

# Part 4:

Give the pseudo-code for each of the following operations for an array list that has n elements and analyze the time complexity:

* Find the minimum-valued item.

|  |  |
| --- | --- |
| **INPUT:** array list with n element  **OUTPUT:** minimum-valued item  **IF** n is 0  **THROW** exception  **ENDIF**  **SET** min index 0 of array list  **FOR n times**  **IF** element is smaller than min  min is the element  **ENDIF**  **ENDFOR**  **RETURN MIN** | **ANALYZE:**  For the best case, array have 1 element and it is the minimum valued item and this operation takes constant time. (It can also throw exception and terminate)  **Tb(n) = Θ(1).**  For the worst case we must check each element and it takes linear time, because it loops n times. **Tw(n) = Θ(n).**  We can generally say **T(n) = O(n).** |

* Find the median item. Consider each element one by one and check whether it is the median.

|  |  |
| --- | --- |
| **INPUT:** array list with n element  **OUTPUT:** median item  **IF** n is 0  **THROW** exception  **ENDIF**  **IF** n is even  **SET** median index n/2  **ELSE**  **SET** median index (n+1)/2  **ENDIF**  **FOR** n times  **SET** count 0  **For** n times  **IF** element indexes are not same and inner loop element is smaller than outer loop element  **INCREMENT** count  **ENDIF**  **ENDFOR**  **IF** median index is equal to count  **RETURN** outer loop element  **ENDIF**  **ENDFOR** | **ANALYZE:**  For the best case:  median item is the first element of the array list. So, outer loop will run 1 time, but still inner loop will run n times.  **Tb(n) = T1(n) \* T2(n) = Θ (n\*1) = Θ(n).**  For the worst case:  Median item is the last element of the array list, so it will run n time outer loop and n time inner loop.  **Tw(n) = T1(n) \* T2(n) = Θ (n\*n) = Θ(n2).**  We can generally say, **T(n) = O(n2) and**  **T(n) = Ω(n).** |

* Find two elements whose sum is equal to a given value

|  |  |
| --- | --- |
| **INPUT:** array list with n element, given value  **OUTPUT:** two elements  **IF** n is smaller than 2  **THROW** exception  **ENDIF**  **FOR n times**  **For n times**  **IF** element indexes are not same and sum of the 2 elements are equal the given value  **PRINT** x and y  **RETURN** true  **ENDIF**  **ENDFOR**  **ENDFOR**  **RETURN** false | **ANALYZE:**  It can throw an exception and don’t enter the loops and it takes constant time. **Θ(1).**  Otherwise we have two nested loops that runs n times both. But we can find our output in first index in outer loop and second index in inner loop (can’t be first because elements shouldn’t be same) and it takes constant time either because it returns afterwards.  **Tb(n) = Θ(1).**  For the worst case, there is no such elements and we must loop n times for both loops.  Since they are nested loops,  **Tw(n) = Θ(n2).**  **We can generally say, T(n) = O(n2).** |

* Assume there are two ordered array list of n elements. Merge these two lists to get a single list in increasing order.

|  |  |
| --- | --- |
| **INPUT:** two ordered array list with n element  **OUTPUT:** merged single list  **CREATE** a new array list with 2n size  **SET** i 0  **SET** j 0  **WHILE** i and j is smaller than n  **IF** ith element of array list1 is smaller than jth  element of array list2  **ADD** ith element of array list 1 to new  array list  **INCREMENT** i  **ELSE**  **ADD** jth element of array list 2 to new  array list  **INCREMENT** j  **ENDIF**  **ENDWHILE**  **WHILE** i is smaller than n  **ADD** ith element of array list 1 to new array list  **INCREMENT** i  **ENDWHILE**  **WHILE** j is smaller than n  **ADD** jth element of array list 2 to new array list  **INCREMENT** j  **ENDWHILE**  **RETURN** new array list | **ANALYZE:**  For the best case:  if n is 1 first while loop only run 1 time. Adding to a list is amortized constant time. So, time complexity of first while loop is constant time, **T1(n) = Θ(1).**  One of the while loops will run 1 time to add the last element of our merged single list and it takes constant time too.  **T2(n) = Θ(1).**  **Tb(n) = T1(n) + T2(n) = max (Θ(1), Θ(1)) = Θ(1).**  For the worst case:  First while loop will run n-1 times, functions n-1 and n grow at the same rate, so we can just say **T1(n) = Θ(n).**  One of the second loops will run only 1 time to add the last element, so it is constant time. We initialize new array list with the total capacity of other 2, so it won’t be out of the capacity.  **T2(n) = Θ(1).**  **Tw(n) = T1(n) + T2(n) = max (Θ(n), Θ(1)) = Θ(n).**  **We can generally say T(n) = O(n).** |
|  |  |
| **Part 5:**  Analyze the time complexity and space complexity of the following code segments: |  |
| int p\_1 (int array[]):  {  return array[0] \* array[2])  } | **Time complexity:** Θ(1), it takes constant time array size doesn’t matter.  **Space complexity:** Θ(1), because it’s not making any memory allocation depends on n, other than input array’s size. |
| b)  int p\_2 (int array[], int n):  {  Int sum = 0  for (int i = 0; i < n; i=i+5)  sum += array[i] \* array[i])  return sum  } | **Time complexity:** Θ(n), the loop runs n/5 time but constants doesn’t effect the grow rate and other statements are simple.  **Space complexity:** Θ(1), because it’s not making any memory allocation depends on n, other than input array’s size. |
| c)  void p\_3 (int array[], int n):  {  for (int i = 0; i < n; i++)  for (int j = 0; j < i; j=j\*2)  printf(“%d”, array[i] \* array[j])  } | **Time complexity:** Program doesn’t terminate, infinite loop. Therefore, we can’t analyze its time complexity. But if we change like int j = 1; then, inner loop is Θ(logn) because it’s growing exponentially. Outer loop is Θ(n), it runs n times. Since they are nested **T(n) = Θ(n.logn).**  **Space complexity:** Θ(1), because it’s not making any memory allocation depends on n, other than input array’s size. |
| d)  void p\_4 (int array[], int n):  {  If (p\_2(array, n)) > 1000)  p\_3(array, n)  else  printf(“%d”, p\_1(array) \* p\_2(array, n))  } | **Time complexity:**  If statement 🡪 p­­\_2’s time complexity is Θ(n).  **T0(n) = Θ(n).**  If the statement is true than p\_3 will take Θ(nlogn).  **T1(n) = Θ(nlogn).**  If the statement is not true p\_1 and p\_2 called. p\_1’s time complexity is Θ(1) and p\_2’s Θ(n). **T2(n) = Θ(1\*n) = Θ(n).**  Both in best case and worst case inside of if statement will run.  But in best case it will enter else statement.  So, **Tb(n) = max(T0(n),T2(n)) = Θ(n).**  **Tw(n) = max(T0(n),T1(n)) = Θ(nlogn).**  **In general, we can say O(nlogn) and Ω(n).**  **Space complexity: :** Θ(1), because neither of p\_1, p\_2 and p\_3 and p\_4 methods are not making any memory allocation depends on n, other than input array’s size. |