GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
HOMEWORK 2

Yusuf Arslan
200104004112

1-) Specify following statements true or false (proved)

Assume c, c_1, c_2, n_0 are constants

a. $\log_2 n^2 + 1 \stackrel{?}{=} O(n)$

$0 \leq \log_2 n^2 + 1 \leq c.n$ for all $n \geq n_0$ (definition of O)

Let's look at if there is an asymptotic upper bound.

Simplify → $0 \leq \log n \leq c.n$ is true, because $y_1 = \log n$ stay bellow. $y_2 = n$ (TRUE)

b. $\sqrt{n(n+1)} \stackrel{?}{=} \Omega(n)$

$0 \leq c.n \leq \sqrt{n^2+n}$

Let's look at if there is an asymptotic lower bound.

simplify → $0 \leq c.n \leq \sim n$ for all $n \geq n_0$ (Definition of Ω)

this statement is true, there are constants satisfy the expression (TRUE)

c. $n^{n-1} \stackrel{?}{=} \Theta(n^n)$

The definition of Θ say: $0 \leq c_1 . n^n \leq n^{n-1} \leq c_2 n^n$, $n \geq n_0$

Simplify → $0 \leq c_1 . n^n \leq \frac{n^n}{n} \leq c_2 . n^n$

↳ this inequality does not correct. (FALSE)

2-) Order the functions by using limit.

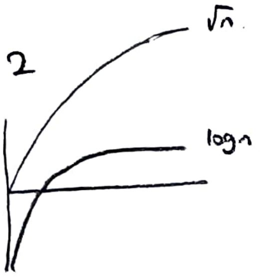
Grows

1. $\log n$
2. $\sqrt{n}$
3. n^2
4. $n^2 \log n$
5. n^3, $8^{\log_2 n}$
6. 2^n
7. 10^n

Check the limits

a. Compare 1 and 2

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$$



$T(\sqrt{n}) > T(\log n)$.

b. Compare 2 and 3

$$\lim_{n\to\infty} \frac{\sqrt{n}}{n^2} = \lim_{n\to\infty} \frac{n^{1/2}}{n^2} = 0 \text{ (by def)}$$

$T(n^2) > T(\sqrt{n})$

c. Compare 3 and 4

$$\lim_{n\to\infty} \frac{n^2}{n^2 \log n} = \lim_{n\to\infty} \frac{1}{\log n} = 0$$

$T(n^2 \log n) > T(n^2)$

d. Compare 4 and 5.

firstly let's look at the n^3 and $8^{\log_2 n}$.

$8^{\log_2 n} = n^{\log_2 8}$ (logarithm def)

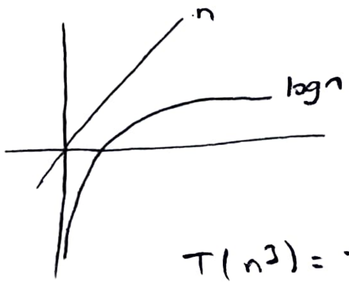
$= n^3$.

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

Then,

$$\lim_{n\to\infty} \frac{n^2 \log n}{n^3} = \lim_{n\to\infty} \frac{\log n}{n} = 0.$$

Let's look at their graphs



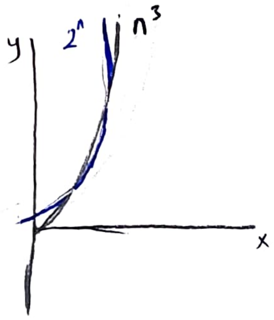
n grows faster then log n therefor limit goes to 0.

$T(n^3) = T(8^{\log_2 n}) > T(n^2 \log n)$

e. Compare 5 and 6

$$\lim_{n\to\infty} \frac{n^3}{2^n} = 0.$$

Let's look at their graph



until $n \sim 10$, n^3 grows faster. BUT then 2^n grows faster then n^3. therefor limits goes to 0.

$T(2^n) > T(n^3)$

f. Compare 6 and 7.

$$\lim_{n\to\infty} \frac{2^n}{10^n} = \lim_{n\to\infty} \left(\frac{1}{5}\right)^n = 0. \text{ (def.)}$$

$T(10^n) > T(2^n)$

3-) Calculate time complexity of the following programs

a.
```
int p-1(int my_array[]){
    for(int i=2; i<=n; i++){
        if(i%2 == 0)
            count++;
        else
            i = (i-1)i;
    }
}
```

Let's start the iteration.

step 1: $i = 2$ and $i \leq n$ $\Theta(1)$

first condition will be satisfy.

$\rightarrow$ count++. $T(n) = \Theta(1)$

then, increase i++.

step 2: $i = 3$ and $i \leq n$. $(\Theta(1))$

second condition will be satisfy.

$\rightarrow$ $i = (3-1).3$ $\rightarrow$

1. subtraction
2. multiplication
3. assignment.

$T(n) = \Theta(3)$.

then, increase i++

step3 $i = 7$ $i \leq n$. $(\Theta(1))$

second condition will be satisfy.

$i = (7-1).7$ $T(n) = \Theta(3)$

then increase i++.

And from now on, the 2nd condition will always be satisfy. Which is:

$i = (i-1).i$

Let's look $\underline{\underline{i}}$ valus during the iteration:

2, 3, 7, 43, 1807, 3263443....

loop will be continue until:

$2, 3, (3^2-3+1), (3^2-2)^2-(3^2-2)+1$

...

simplify ~ $3^k, (3^k)^k, ((3^k)^k)^k$ ...

$= 3^k, 3^{k^2}, 3^{k^3}, \ldots, 3^{k^{\log_k(\log n)}}$

If $3^{k^{\log_k(\log n)}} >= n$ loop will be break.

number of steps will be $\log(\log(m))$.

We determine $\Theta(1)$ $\xrightarrow{}$ constant for the operations inside the loop.

number of steps in the loop will give us the time complexity which is:

$T(N) = \log(\log(n)).+$(constant times)

$O(\log(\log(n)))$

b.

```
  int p-2(int my-array[]){
1     first-element = my-array[0];
2     second-element= my-array[0];
3     for(int i=0; i < size ; i++){
4         if(my-array[i] < first-element){
5             second-element = first-element;
6             first-element = my-array[i];
7         } else if(my-array[i]<second){
8             if(my-array[i] != first){
9                 second-element = my-array[i];
              }
          }
      }
  }
```

Step 1

line 1 and 2 makes a simple assignment : $\Theta(1) + \Theta(1)$

step 2

iteration will be done as much as size of array. (linear): $O(n)$

step 3

condition 1 ($\Theta(1)$) check if element of array smaller then first element.

assignment ($\Theta(1)$)

assignment ($\Theta(1)$).

condition 2 ($\Theta(1)$) check if element of array smaller then second-element.

condition 2.1 ($\Theta(1)$) if the element of equal the first.

assignment ($\Theta(1)$)

if the conditions not satisfy, then iterate loop.

Finally inside the loop all operations are constant-time.

$T(N) = c.n +$ (constan times).

$O(n)$

c.

```
int p-3(int array[]){
    return array[0] * array[2];
}
```

Index operators of array will be constant ($\Theta(1)$)

multiplication will be constant ($\Theta(1)$

operation will be constant

$\Theta(1)$.

d.

```
  int p-4(int array[], int n){
1     int sum = 0
2     for(i=0; i<n ; i+=5)
3         sum += array[i] * array[i].
4     return sum
  }
```

line 1 assignment ($\Theta(1)$).

line 2 i. increas by 5, therefor loop will be iterate $n/5$ times.

line 3 assignment, addition and multiplication are constant.

Therefor time complexity will be $O(n/5)$ + constans $\rightarrow$ $O(n)$.

$\Theta(n)$ is also correct.

e.

```
void p-5 (int array[], int n) {
    for (i=0; i<n ; i++)
        for (j=1; j<i ; j*=2)
            print (array[i]array[j])
}
```

step 1 outer loop is depend on the n. so it is linear and time complexity is $O(n)$.

step 2 inner loop is depend on i, but, j is increasing by x2. Therefor its time complexity is logn.

$T(N) = O(n \log n)$.
$\Theta(n \log n)$ is also correct.

f.

```
int p-6 (int array[], int n) {
    if (p-4(array, n)) > 0
        p-5 (array, n).
    else
        print ( p-3 (array) * p_4(array, n))
}
```

step 1 if first condition is satisfy.

(if statetment) -> $T(p\text{-}4()) = O(n)$

$T(p\text{-}5()) = O(n \log n)$.

$T(N) = O(n \log n)$
$\Omega(n)$.

step 2 If first does not satisfy.

To check condition -> $T(p\text{-}4()) = O(n)$.

To print -> $T(p\text{-}3()) = \Theta(1)$.
$T(p\text{-}4()) = O(n)$.
multiplication $\Theta(1)$.

$T(N) = O(n) + \Theta(1)$.

Total time complexity will be $O(n \log n)$.

first condition always be executed therefor $\Omega(n)$ is correct also for the statement.

g.

```
int p-7 (int n) {
    int i=n;
    while (i>0) {
        for(int j=0; j<n; j++).
            system.out.println("*");
        i = i/2 ;
    }
```

step 1
while loop logarithmically decrease depend on n.. $\Theta(\log n)$

step 2
for loop is depend on n. and linearly increase. $\Theta(n)$.

step 0
assign n to the i is constant time $\Theta(1)$.

Total complexity is $O(n \log n)$

It is also correct $\Theta(n \log n)$

h.

```
int p_8 (int n) {
    while (n>0) {
        for (int j = 0; j<n; j++)
            system.out.println ("*");
        n = n/2;
    }
}
```

step 1 outer loop (while) starts with n and decrease n/2 in every iteration.
Therefor its time complexity is $\Theta(\log n)$

step 2 inner loop (for) is depends on n and counter increase linearly. $\Theta(n)$.

Total time complexity is
$O(n \log n)$.
$\Theta(n \log n)$ also correct.

i.

```
int p-9 (n) {
    if (n = 0)
        return 1
    else
        return n * p-9(n-1).
}
```

Base Case: if n equals 0 it will return. time complexity will be constant $\Theta(1)$.

recurence part.

$T(n) = \Theta(1) + T(n-1)$.
↳ multiplication.

until n equals 0, [(n-1) time] there will be multiplication.
so there will be n-1 multiplication operation.
there for its complexity is linear.
$O(n)$ and $\Theta(n)$ is correct for the statement.

J.

```
int p-10( int A[], int n) {
    if (n = = 1) return;
    p-10(A, n-1);
    J = n-1;
    while (J>0 and A[J] < A[J-1]) {
        SWAP (A[J], A[J-1]);
        J = J-1;
} }
```

step1 base case: $\Theta(1)$.
step2 recursive: $\Theta(n)$.
step3 assign n-1 to J. $\Theta(1)$.
step4 while loop. $(\Theta(n))$

- J decrease linearly, but, if A[J] smaller than A[J-1] condition will be $\Theta(1)$. will be $\Theta(1)$.
- SWAP will be $\Theta(1)$.
- J = J-1 will be $\Theta(1)$.
- But, while loop will be $O(n)$ [it depends on n].

If we combine recursive call with while loop
$T(n) = O(n^2)$.

4-) True - False (prove)

a. "The running time of algorithm A is at least $O(n^2)$."
What is wrong with this statement?

Big O notation is used for define asymptotic upper bounds.
So, 'at least' expression is not correct.
If, big O notation is to be used, we can say
" The running time of algorithm A is $O(n^2)$ at most "
Otherwise, if "at least" is to be used, Ω notation should be use.
" The running time of algorithm A is at least $\Omega(n^2)$."
Ω notation is used for express asymptotic lower bounds.

b.

1. $2^{n+1} \stackrel{?}{=} \Theta(2^n)$

$c_1 . 2^n \leq 2^{n+1} \leq c_2 2^n$

If $c_1 = 2$ the expression will be TRUE.

II. $2^{2n} = \Theta(2^n)$

$c_1 . 2^n \leq 2^{2n} \leq c_2 . 2^n$

To the make this statement true, c_1 should be 2^n which is not constant. Therefor, FALSE

III. $F(n) = O(n^2)$ and $g(n) = \Theta(n^2)$ prove / disprove $F(n) + g(n) = \Theta(n^4)$.

step1 $0 \leq F(n) \leq c_1 . n^2$ → multiply these expressions.

step2 $c_2 . n^2 \leq g(n) \leq c_3 . n^2$

x

$0 \leq F(n) \leq c_1 . c_3 . n^4$ → The lower limit of the two expression is O(zero)

This is outside the asymptotically tight bound condition in the definition of the Θ function.

$\Theta(n^4)$ suggest that
$k . n^4 \leq F(n) + g(n) \leq l . n^4$

FALSE

5-) Solve the reccurence relations and express most apropriade asymptotic notation.

a. $T(n) = 2T(n/2) + n$, $T(1) = 1$

$T(n) = 2T(n/2) + n$

$\hookrightarrow$ $T(n/2) = 2.T(n/4) + n/2$ (apply it to the formula)

$= 2T(n/2) + n$

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

$= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$

$= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n$

$\vdots$

$= n * T(1) + \log_2(n) * n$

at each call, n dividing by 2. Therefor, height(number of iteration) will be $\log_2 n$.

So, $T(n) = n.T(1) + \log_2(n) * n$

$= \underline{O(n \log n)}$.

b. $T(n) = 2T(n-1) + 1, \quad T(0) = 0.$

$T(n) = 2T(n-1) + 1$

$\quad \hookrightarrow T(n-1) = 2T(n-2) + 1$

$\quad = 2T(n-1) + 1$ apply it.

$= 2[2T(n-2) + 1] + 1 = 4T(n-2) + 3$

$\quad \hookrightarrow T(n-2) = 2T(n-3) + 1$ apply it.

$= 4[2T(n-3) + 1] + 3 = 8T(n-3) + 7$

$\quad \hookrightarrow T(n-3) = 2T(n-4) + 1$ apply

$= 8[2T(n-4) + 1] + 7 = 16T(n-4) + 15.$

$\vdots$

$2^n T(0) + 2^n - 1$

$T(n) = 2^n \underbrace{T(0)}_{0} + 2^n - 1$

$= 2^n . 0 + 2^n - 1$

$= 2^n - 1$

So, asymptotic notation of this relation will be $\Theta(2^n)$.

6-) In an array of numbers (positive or negative), find pairs of numbers with the given sum. Design an iterative algorithm for the problem…

```python
1 def sum_detector_iterator(Array, Sum, counter):
2     for i in range(0, size):
3         for j in range(i, size):
4             if (Array[i] + Array[j] == Sum):
5                 counter += 1
```

This is a Python code that adds every two element of `Array` to each other then compares the result with the `Sum`. The function takes a randomly filled array (`Array`), a random value (`Sum`) and `counter`. If the `Sum` equals that result then `counter` increases by one.

<u>Time Complexity:</u>

- In the 2nd row, first loop starts with `i=0` and continue until `i=size-1`. So, time complexity will be: *T(n) = n*

- In the 3rd row, second loop is depend on value of `i`. Loop starts with `j=0` and continue until `j=i-1`.

  Take a look at iteration behavior and iteration numbers step by step:
  [1]- `j=0`, range is `size`. Iteration number: `size-1`
  [2]- `j=1`, range is `size`. Iteration number: `size-2`
  [3]- `j=2`, range is `size`. Iteration number: `size-3`
  .
  .
  [size]- `j=size-1`, range is `size`. Iteration number: `1`

Then, we can take average of iterations.
(1+2+3+…+(size-1))/size = size/2
We can assume that the loop iterates size/2 times.

Therefore, time complexity will be: *T(n) = n/2*

- So for every iteration, code in 4th line will execute.
  The Array index operations, addition and comparision will be take constant time.
  Lets say time complexity of line 4 is *T(n) = 1*

  Finally, total time complexity T(N) will be,
  T(N) = n*(n/2 * 1) = 1/2 * n^2 and
  = θ(n^2)

Let's run the program to test time complexity results. with different values.
When `size` equal `1000`.

```
$ python test.py

Total elapsed time:
0.04807162284851074

$ python test.py

Total elapsed time:
0.047557830810546875

$ python test.py

Total elapsed time:
0.04894852638244629
```

When `size` equal `10000`.

```
$ python test.py

Total elapsed time:
5.23456335067749

$ python test.py

Total elapsed time:
5.1165759563446045

$ python test.py

Total elapsed time:
5.086673021316528
```

As can be seen, when the `size` increases by 10 times, the elapsed time increases approximately 100 times. [θ(n^2)]

7-) Write a recursive algorithm for the problem in 6 and calculate its time complexity. Write a recurrence relation and solve it.

```python
1  def sum_detector_recursive(Array1, Array2, Sum, counter):
2      if (len(Array1) == 0):
3          return
4      elif (len(Array2) == 0):
5          Array2 = Array1
6          sum_detector_recursive(Array1[1:], Array2, Sum, counter)
7      else:
8          if (Array1[0] + Array2[0] == Sum):
9              counter += 1
10         sum_detector_recursive(Array1, Array2[1:], Sum, counter)
```

This is a Python code that adds every two element of Array to each other recursively then compares the result with the Sum. The function takes a randomly filled array (Array), same Array as second paramter to make comparision, a random value (Sum) and counter.
If the Sum equals that result, then counter increases by one.

<u>Time Complexity:</u>

- In the 2nd row condition check if length of Array1 which we use for iterate traverse our array. If it is empty hereafter, terminates the reccurence.
  Its time complexity is *T(n) = 1*

- In the 4th row, if the Array2, which we are using to compare Array1 with, size became 0, Array1 being shifted recursively.
  But before shift Array1, to reinitialize the second array, we assign it to the Array1.
  It will take constant time *T(n) = 1*

This shifting process will be continue as much as size of our array. Therefore, its time complexitiy will be T(n) = n

- In the 7^{th} row, if condition, which compares pairs with `Sum`, and increasing `counter`, will take constant time. T(n) = 1

  But then, another recursive call will be proceed, `Array2` will be shifted. But, as we remember, in the line 5, we assign `Array1` -which is shrinks by 1 in every iteration- to the `Array2`. Therefore, this recursive call will be cost n/2 average. T(n) = n/2

  So, total time complexity will be,
  T(N) = n*(n/2) = 1/2 * n^2 and
  = θ(n^2)