*Homework 2 - Yeşim Yalçın 200102004094

(1)

$\rightarrow$ Form: $x(n) = a\,x(\frac{n}{b}) + f(n)$

$\rightarrow a \geq 1$ $\rightarrow x(n)$ = non-decreasing

$\rightarrow b \geq 2$ $\rightarrow f(n)$ = polynomial +

$\rightarrow x(1) > 0$

$$* \quad x(n) \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) = O(n^{\log_b a}) \\ \Theta(n^{\log_b a}\log n) & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^{\log_b a}) \ \&\ af(\frac{n}{b}) < cf(n) \end{cases}$$

$\boxed{1 < c}$

a-) $T(n) = 16T(\frac{n}{4}) + n!$

$a = 16$ ✓

$b = 4$ ✓

$f(n) = n!$ ✓

$\rightarrow n^{\log_b a} = n^{\log_4 16} = n^2$

$\rightarrow f(n) \in \Omega(n^{\log_b a})$

$n!$ has a lot more growth rate.

$* \ 16 f(\frac{n}{4}) < c f(n)$

$16 \frac{\cancel{n!}}{4!} < c \cancel{n!}$

$\frac{16^{\cancel{4}2}}{\cancel{4}\cdot 3 \cdot 2 \cdot 1} < c$

$\frac{2}{3} < c$

$c < 1$ ✓

*It is the 3rd case.

$\boxed{\begin{array}{l} \rightarrow x(n) \in \Theta f(n) \\ \rightarrow x(n) \in \Theta(n!) \end{array}}$

b-) $T(n) = \sqrt{2}\,T(\frac{n}{4}) + \log n$

$a = \sqrt{2}$ ✓

$b = 4$ ✓

$f(n) = \log n$ ✓

$\rightarrow n^{\log_b a} = n^{\log_4 \sqrt{2}} = n^{\log_4 2^{\frac{1}{2}}} = \boxed{\frac{1}{2} n^{\log_4 2}}$

$\rightarrow \frac{1}{2} n^{\log_4 2} \cong n^{\log_4 2} = n^{0.5}$

*Comparing $n^{0.5}$ and $\log n$

$\rightarrow \lim_{n \to \infty} \frac{n^{0.5}}{\log n} = \frac{\infty}{\infty} = \frac{0.5\,n^{-0.5}}{\frac{1}{n} \cdot \frac{1}{\ln 10}} = \frac{n \cdot 0.5 \cdot \ln 10}{n^{0.5}}$

$= n^{0.5} \cdot 0.5 \cdot \ln 10 = \infty$ $\quad \log n \in O(n^{0.5})$

* it is the first case

$\boxed{\begin{array}{l} \rightarrow x(n) \in \Theta(n^{\log_b a}) \\ \rightarrow x(n) \in \Theta(n^{0.5}) \end{array}}$

c-) $T(n)=8T(\frac{n}{2})+4n^3$

$a=8$ ✓ $\quad \log_b^a=\log_2^8=3$

$b=2$ ✓

$f(n)=n^3$ ✓ $\quad f(n)\in\Theta(n^3)$

* It is the second case.

$\boxed{T(n)\in\Theta(n^3\log n)}$

d-) $T(n)=64T(\frac{n}{8})-n^2\log n$

$a=64$ ✓

$b=8$ ✓

$f(n)=-n^2\log n$ ✗

$\boxed{\text{* Theorem cannot be applied because } f(n) \text{ is negative.}}$

e-) $T(n)=3T(\frac{n}{3})+\sqrt{n}$

$a=3$ ✓ $\quad \log_b^a=\log_3^3=1$

$b=3$ ✓

$f(n)=\sqrt{n}$ ✓

* Comparing $\sqrt{n}$ and n

$\rightarrow \lim_{n\to\infty}\frac{n^{0.5}}{n}=\frac{1}{n^{0.5}}=\frac{1}{\infty}$

$\sqrt{n}\in O(n)$

* it is the first case

$\rightarrow T(n)\in\Theta(n^{\log_b a})$

$\rightarrow T(n)\in\Theta(n)$

f-) $T(n)=2^nT(\frac{n}{2})-n^n$

$a=2^n$ ✗ $\Rightarrow a$ is not a constant

$f(n)=-n^n \Rightarrow f(n)$ is negative

$\boxed{\text{* Theorem cannot be solved}}$

g-) $T(n)=3T(\frac{n}{3})+\frac{n}{\log n}$

$a=3$ ✓ $\quad \log_b^a=\log_3^3=1$

$b=3$ ✓

$f(n)=\frac{n}{\log n}$

* The ratio: $\frac{f(n)}{n}=\frac{1}{\log n}$. The ratio is not polynomial. Theorem cannot be applied.

(2)

a-)

$\rightarrow$ Dividing into nine

$\rightarrow$ Size $\frac{1}{3}$

$\rightarrow$ Quadratic time $\Theta(n^2)$

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

$a=9 \quad \log_b^a=\log_3^9=2 \quad =n^2$

$b=3$

$f(n)=n^2 \quad f(n)\in\Theta(n^2)$

* 2nd condition:

$\boxed{\rightarrow T(n)\in O(n^2\log n)}$

b-)

$\rightarrow$ Dividing into 8

$\rightarrow$ Size $\frac{1}{2}$

$\rightarrow O(n^3)$

$T(n)=8T(\frac{n}{2})+O(n^3)$

$a=8 \quad \log_b^a=\log_2^8=3 \quad f(n)\in O(n^3)$

$b=2$

$n^3\in\Theta(n^3)$

* 2nd condition

$\rightarrow T(n)\in O(n^3\log n)$

c-)

→ Dividing into 2
→ $\frac{1}{4}$ size
→ $O(\sqrt{n})$

$$\left\{ \begin{array}{l} T(n) = 2T(\frac{n}{4}) + O(\sqrt{n}) \\ a = 2 \quad \log_b^a = \log_4^2 = \frac{1}{2} \\ b = 4 \end{array} \right.$$

$\sqrt{n} \in \Theta(n^{\frac{1}{2}})$

* 2nd condition:
→ $T(n) \in \Theta(\sqrt{n} \log n)$

* Comparing a and b

$$\lim_{n \to \infty} \frac{n^2 \log n}{n^3 \log n} = \frac{1}{n} = \frac{1}{\infty} \quad n^3 \log n \in \omega(n^2 \log n)$$

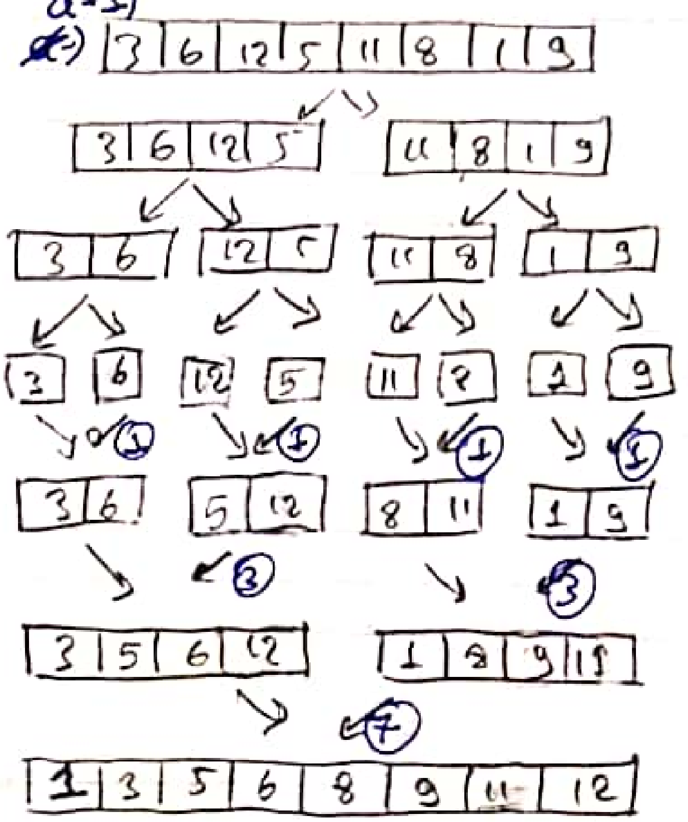
* Comparing a and c

$$\lim_{n \to \infty} \frac{n^2 \log n}{\sqrt{n} \log n} = n^{1.5} = \infty \quad \sqrt{n} \log n = o(n^2 \log n)$$

* $n^3 \log n > n^2 \log n > \sqrt{n} \log n$ (growth rates)
$O(\sqrt{n} \log n)$ is the best option to choose ⇒ c

3

a-1)



→ 3 and 6
→ 12 and 5
→ 8 and 11
→ 1 and 9

4 — 1 for each pair $(1+1-1) = 1$

→ 3 and 5
→ 6 and 5
→ 6 and 12
→ 8 and 1
→ 8 and 9
→ 11 and 9

6 — 3 for each pair $(2+2-1) = 3$

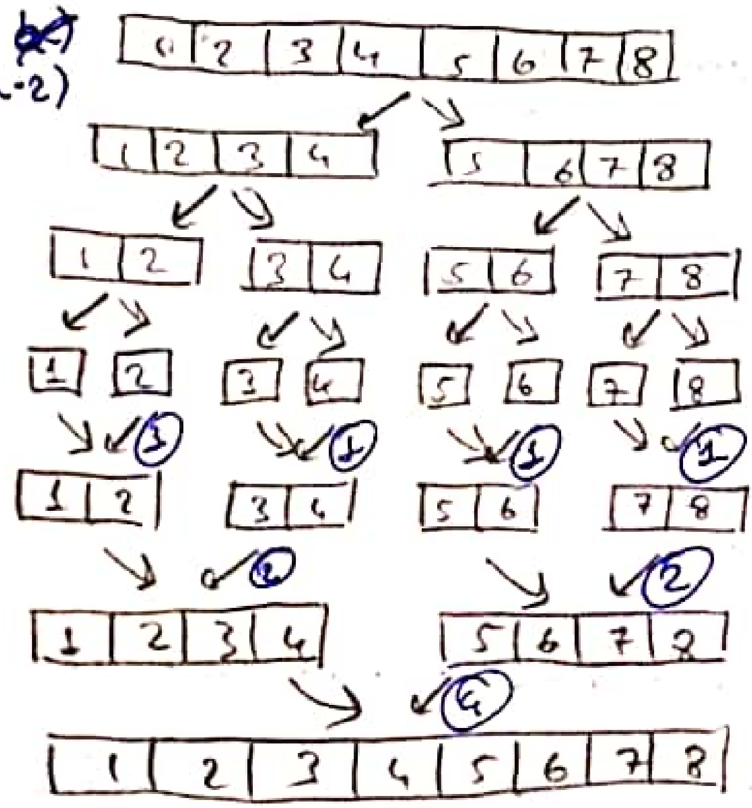
→ 3 and 1
→ 3 and 8
→ 5 and 8
→ 6 and 8
→ 12 and 8
→ 12 and 9
→ 12 and 11

7 — 7 for the pair $(4+4-1) = 7$

+ / 17 comparison

* Merging 2 sorted arrays take k+m-1 comparisons at most.
k and m are sizes of the arrays.

a-2)



1 and 2
3 and 4
5 and 6
7 and 8 } 4

1 and 3
2 and 3
5 and 7
6 and 7 } 4

1 and 5
2 and 5
3 and 5
4 and 5 } 4

+ 12 Comparisons

* If the elements in one array are smaller than the merging array, the comparisons can be minimized. Because we only need m amount of comparisons where m is the array size.

b-1) According to the algorithm we have discussed on class, the swap operations happen when:
→ Placing the pivot to it's proper position. (regardless of if the position does not change)
→ Swapping a value which is smaller than the pivot and bigger than the pivot.

* To have the maximum amount of swaps, we need to generate maximum amount of pivots also maximum amount of sub swaps for each pivot.
* When the array is ordered we get max amount of partitions however, the swap will happen for only placing the pivots which will be equal to n-1 swaps.

Because of that using an already sorted array would not be a good example. Instead I have chosen an array that guarantees maximum of swaps in each step.

Array: 10 11 13 12 4 3 5 2

→ 2 and 11
→ 5 and 13
→ 3 and 12
→ 10 and 4

swapped ⇒ 4 2 5 3 10 12 13 11

4 2 5 3 ↓ 12 13 11

3 2 4 5 ↓ 11 12 13

3 2 → 23

→ 3 and 5
→ 4 and 3
→ 3 and 2
→ 11 and 13
→ 12 and 11

9 swaps in total.

* An even number array can have max $\frac{n}{2}$ swaps. An odd number array can have max $\frac{n+1}{2}$ swaps. These are satisfied.

b-2) If we have the min amount pivots and min amount sub swaps in each step, we can have the min amount of swaps. For this, I have chosen an array that needs to change the middle element with the pivot and rest don't need to be swapped.

Array: 20 13 15 14 23 21 22 24

→ 20 and 14 ⇒ 14 13 15 20 23 21 22 24
→ 14 and 13 — 14 13 15 — 23 21 22 24
→ 23 and 22 — 13 14 15 — 22 21 23 24
→ 22 and 21 — 13 15 — 22 21 24

21 22

4 swaps in total

* There cannot be less than 4 swaps because in each sub step there needs to be min 1 swaps. A 8 element array can be divided into min 3 sub steps. When we include the start array it is 4 steps in total.

(4) Best Case: $\Theta(1)$ if the mid element is 0. It directly returns after a few constant time operations.

* Generating recurrence relation for rest of the cases:
This algorithm will keep dividing an array by 2 till it reaches 0 element. We will assume there is always a zero in the array so that the algorithm never fails.
→ The recurrence repeats itself with $\frac{n}{2}$ input each time it is called.
→ Each time the recurrence relation is called, algorithm function will be called once inside. Therefore the problem is not divided into subproblems.
→ In each call, $\Theta(1)$ time operations will be applied.

↳ $T(n) = T(\frac{n}{2}) + 1$ is the recurrence relation.

* $a = 1$, $b = 2$, $\log_b^a = \log_2^1 = 0$

$f(n) = 1 \in \Theta(n^0) \in \Theta(1)$

<u>* case two:</u>

$T(n) \in \Theta(n^{\log_b^a} \log n)$

$\boxed{T(n) \in \Theta(\log n)}$

5-) n different sized gifts ⟷ n different sized matching boxes

* My aproach to the problem:
→ Save gifts with their sizes in a 2D array A
ex: A[i][0] = gift no , A[i][1] = gift size.
→ Save boxes with their sizes in a 2D array B
ex: B[i][0] = box no, B[i][1] = box size.
* Array A and array B lengths must be the same also the elements inside A[0-n][1] and B[0-n][1] must be the same as well (in different orders) because for each distinct sized gift there is only 1 same sized box. All will have their unique boxes with the same size.
→ Use help of Quicksort. Take the element A[0][1] as pivot and do a rearrange operation on array B. However do not add the pivot into array B. Repeat for all elements of A. Then do the reverse to sort array B. By doing this we compare only a gift with a box and a box with a gift utilizing Quicksort's rearrange function.
→ In the end we will have array A and B sorted according to their sizes and we can match the ones with the same index.

a-)

```
sort (array, pivot)
    partition(array, 0, array length-1, pivot)
    return array
end sort

swap (array, pIndex, cIndex)
    tempF = array[pIndex][0]
    tempS = array[pIndex][1]
    array[pIndex][0] = array[cIndex][0]
    array[pIndex][1] = array[cIndex][1]
    array[cIndex][0] = tempF
    array[cIndex][1] = tempS
end swap
```

```
partition(array, start, end, pivot)
    up=start
    down=end
    loop while up<down and array[up][1]<=pivot
        up=up+1
    end loop
    loop while table[down][1]>pivot
        down=down-1
    end loop

    if up<down
        swap(array, up, down)
    endif
end partition
giftBoxSort(array, pivotArray, count)
    if(count==pivotArray.length)
        return array  endif
    array=sort(array, pivotArray[count][1])
    count=count+1
    return giftBoxSort(array, pivotArray, count)
end giftBoxSort
main:
    arrayB=giftBoxSort(arrayB, arrayA, 0)
    arrayA=giftBoxSort(arrayA, arrayB, 0)
    for(i=0 to arrayA.length )
        arrayC[i][0]=arrayA[i][0]
        arrayC[i][1]=arrayB[i][0]
        arrayC[i][2]=arrayA[i][1]
    end for
end main
```

☆ **Note:** Array C is a 2D array which saves the matched box and gift numbers also their sizes.

b-)

→ Swap function is $O(1)$ because all the operations inside takes constant time.

→ partition function is the same partition function of Quicksort algorithm. The only difference, it does 1 less swap which does not have an effect on the complexity. Quicksort algorithm's partition function has n+1 operations. meaning $O(n)$.

→ Sort function does only 1 partition function meaning $O(n)$.

→ giftBoxSort is the recursive function.

↳ The recurrence increases the count by 1 each time to end at a specific point.

↳ Each time the recurrence relation is called, algorithm function will be called once inside. Therefore, the problem is not divided into subproblems.

→ In each call $O(n)$ time operations will be done

↳ $T(n) = T(n-1) + n$

$$T(n) = T(n-1) + n$$
$$T(n-1) = T(n-2) + n-1$$
$$T(n-2) = T(n-3) + n-2$$

$$T(n) = T(n-2) + n-1 + n$$
$$T(n) = T(n-3) + \underbrace{n-2 + n-1 + n}$$
$$\vdots$$

$$\frac{n(n-1)}{2} = n^2 \Rightarrow \boxed{O(n^2)}$$

→ the main function includes giftBoxSort method which is $O(n^2)$