Name : Fadi Alahmad Alomar

ID : 1200180049

Assignment 3: Divide and Conquer

Question 1

(a) $T(n) = 3T(\frac{n}{2}) + \mathcal{O}(n)$

Using the master theorem we get:

$b = 2,\ a = 3,\ f(n) = n$
$c_{crit} = \log_b a = \log_2 3 = 1.585$
$f(n) = n = \Omega(n^c)$ where $c = 1$
$\because c_{crit} > c$
$\therefore T(n) = \Theta(n^{c_{crit}}) = \Theta(n^{1.585})$

(b)
$11112222 * 33334444$

n	8
a	1111
b	2222
c	3333
d	4444

(c)

$a * c$	$1111 * 3333$
$b * d$	$2222 * 4444$
$(a + b) * (c + d)$	$3333 * 7777$

(d) $T(n) = 5T(\frac{n}{3}) + O(n)$

Using the master theorem we get:

$b = 3,\ a = 5,\ f(n) = n$
$c_{crit} = \log_b a = \log_3 5 = 1.465$
$f(n) = n = \mathcal{O}(n^c)$ where $c = 1$
$\because c_{crit} > c$
$\therefore T(n) = \mathcal{O}(n^{c_{crit}}) = \mathcal{O}(n^{1.465})$

Question 2

(a) using the first fact we get that $A[1]...A[mid]$ has at most $\frac{kn}{2}$ digits and that $A[mid+1]...A[k]$ has at most $\frac{kn}{2}$ digits
then using the first fact we can multiply these two numbers in $\mathcal{O}(size^{1.6}) = \mathcal{O}((\frac{kn}{2})^{1.6}) = \mathcal{O}((kn)^{1.6}) = \mathcal{O}(k^{1.6}n^{1.6})$

(b)

```
FUNCTION NumberProduct(number1,number2):
    n -> number1.len() # how many digits the number is made of
    if n == 1:
        return number1 * number2
    a -> number1//10^(n//2) # upper half of the number1
    b -> number1%10^(n//2)  # lower half of the number1
    c -> number2//10^(n//2) # upper half of the number2
    d -> number2%10^(n//2)  # lower half of the number2
    x -> NumberProduct(a+b,c+d)
    y -> NumberProduct(a,c)
    z -> NumberProduct(b,d)
    return 10^(n//2)*x + 10^(n)*y+z-10^(n//2)*(y+z)
end

FUNCTION ArrayProduct(arr,i,j):
    if i==j:
        return arr[i]
    mid = (i+j)//2
    return NumberProduct(ArrayProduct(arr,i,mid),ArrayProduct(arr,mid+1,j))
end
```

The algorithms recurrence equation is:
$T(k) = 2T(\frac{k}{2}) + \mathcal{O}(k^{1.6}n^{1.6})$
we can consider $n^{1.6}$ as an overhead at this point as it is not part of the reccurence and can be

ignored while using the master theorem.
Using the master theorem we get:

$b = 2,\ a = 2,\ f(k) = n^{1.6} * k^{1.6}$
$c_{crit} = \log_b a = \log_2 2 = 1$
$f(k) = n^{1.6} * k^{1.6} = \Omega(k^c)$ where $c = 1.6$
$\because c > c_{crit}$
$\therefore T(k) = \mathcal{O}(f(k)) = \mathcal{O}(n^{1.6} * k^{1.6})$

Question 3

```
FUNCTION search(arr,k):
    low -> 0
    high -> arr.size - 1
    mid -> (low+high)//2
    while low <= high, do:
        if k > arr[mid]:
            low -> mid + 1
        else if k <arr[mid]:
            high -> mid - 1
        else:
            break
        end
        mid -> (low+high)//2
    end
    if k != arr[mid]:
        mid +=1
    end
    return mid
end

FUNCTION search_arr(arr1,arr2,location):
    low -> 0
    high -> arr1.size - 1
    mid -> (low+high)//2
    while low <= high, do:
        place = search(arr2,arr1[mid])+mid
        if place == l:
            return arr1[mid]
        if place > l:
            high -> mid - 1
        else:
            low -> mid + 1
        end
        mid -> (low + high) // 2
    end
    return None
end

FUNCTION find_median(a,b):
    na -> a.size
    nb -> b.size
    needed -> [(na+nb)//2]
    found -> [NULL]
    d -> 1
    if (na+nb)%2 == 0:
        needed -> [needed, needed+1]
        found -> [NULL,NULL]
        d -> 2
    for int i=0, i<needed.size, i->i+1, do:
        f -> search_arr(a,b,needed[i])
        if f is not NULL:
```

```
            found[i] = f
    for int i=0,i<needed.size, i->i+1, do:
        if found[i] is not NULL:
            continue
        f -> search_arr(b,a,needed[i])
        if f is not NULL:
            found[i] = f
    s -> 0
    for int i=0, i<found.size, i->i+1, do:
        s->s+found[i]
    return s/d
end
```

the above algorithm chooses the middle value of one of the arrays then searches the other one to find the place of that value if it is place in the original array plus the found place is higher than the median place then we search the second array with the element that is in the lower middle half of the first array, if it is lower we search with the the element that is in the higher middle half of the first array.
if we have not found the median element(s) we do the same but with switching the arrays
$\therefore$ the algorithm has a run time of $\mathcal{O}(\log n * \log m)$ where n, m are the sizes of the arrays A, B

Question 4

(a)

```
FUNCTION check(arr):
    narr -> sorted(arr) # sorting the array
    s->0
    for int i = 0, i< narr.size,i->i+1, do:
        s->s+narr[i]
        if (i+1)^3>s:
            return False
        end
    end
    return True
end
```

the only way that the sum will be smaler is if it had the smallest elements thus if we sort the array and loop over it summing the elements as we pass them then check if that sum is smaller than the size which is the index of the element + 1 to the third power
$f(n) = n \log n + n$, $n \log n$ for the sorting, n for looping over the elements.
$\therefore f(n) = \mathcal{O}(n \log n)$

(b)

```
FUNCTION check(arr,k)
    heap->max_heap(k)  # max heap implementation using arrays
    for int i = 0, i< k+1,i->i+1, do:
        heap.push(arr[i])
    end
    for int i = k+1, i< n+1,i->i+1, do:
        if arr[i]<heap.root:
            heap.pop()
            heap.push(arr[i])
        end
    end
    s -> 0
    for int i = 0, i< k+1,i->i+1, do:
        s->s+heap[i]
    end
    if (k^3)>s:
        return False
    end
    return True
end
```

the only way that the sum will be smaler is if it had the smallest elements.
we start by a max heap that is implemented using and array of size k, then we push the first k elements to it then we iterate over the main array and check if the element is smaller than the maximum element in the heap we pop it and push the smaller element. this way we are sure that in the heap array the subarray that has a size k and has the smallest elements.
then we sum all the elements in it and check if they are smaller than their cardinality^3
the heap push and pop functions take $\mathcal{O}(\log x)$ where x is the size
$\therefore f(n,k) = k\log k + 2(n-k)\log k + k$
where:
$k\log k$, k is for the first loop, $\log k$ is for the push.
$2(n-k)\log k$, $(n-k)$ is for the secound loop, $2\log k$ is for the pop and push.
k is for the last loop.
$\therefore f(n,k) = \mathcal{O}(k\log k + 2(n-k)\log k + k) = \mathcal{O}(k\log k + (n-k)\log k) = \mathcal{O}(\log k(n-k+k)) = \mathcal{O}(n\log k)$