HW4 REPORT

Kahraman Arda KILIÇ – 1901042648

Q1.

If we cut 2 sticks together, we may get 3 or 4 sticks. This means that we doubled the number of sticks (4). n is the length of stick. n > 2^k where k is the minimum number of cuts. In this algorithm, we cut the stick in the middle and k increases by 1, thus minimum number of cuts increases. After we hit the 1 meter for every stick, then we may trace back and find the count.

```python
import math

def cutter(n):
    count = 0

    if n == 1:
        return 1
    else:
        count += cutter(math.floor(n/2))
        count += 1

    return count

n = 10
print("Minimum number of cuts needed:",cutter(n))
```

There is a recursive function. There should be a recurrence relation. It is T(n) = T(n/2) + O(1)

- ⇨ T(n) = T(n/2) + O(1)
- ⇨ By using Master Theorem $n^{(\log)_b^a}$ = $n^{\log_2 1}$ = O(1) , It is second case
- ⇨ θ ($n^{\log_2 1}$ logn) = θ (logn)

Q2.

In this question, we are supposed to calculate the worst and best experiment. We have 2 arrays, one is experiments and the other one is success rates. In the function, it goes recursively until it reaches 2 elements in the success_rates array. When it finds it will compare 2 numbers and return max, min and their indexes. Otherwise, it will divide the array into 2 pieces and call the function for both pieces recursively. At the end, it will return the maximum of the two numbers from left and right side.

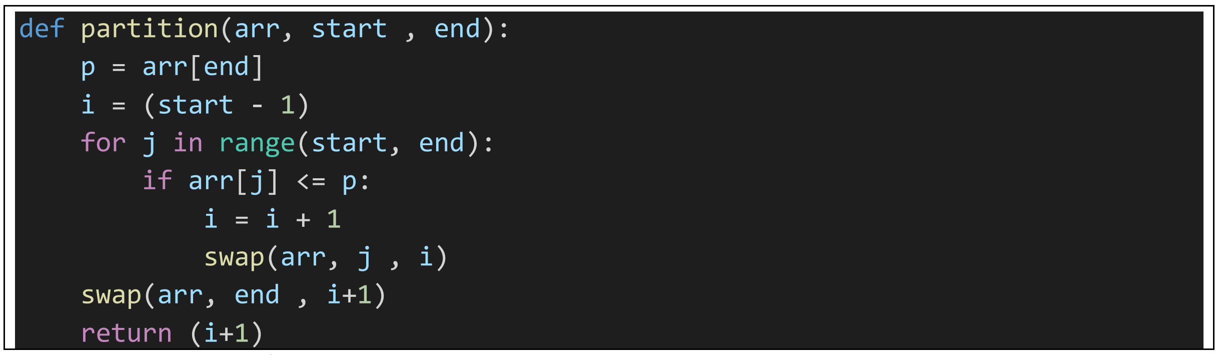
There is a recursive function. The recurrence relation is T(n) = 2T(n/2) + O(1)

- ⇨ T(n) = 2T(n/2) + O(1)
- ⇨ By using Master Theorem, $n^{(\log)_b^a}$ = $n^{\log_2 2}$ = n^1 = n > O(1), It is first case
- ⇨ θ ($n^{\log_2 2}$) = θ (n)

Q3.

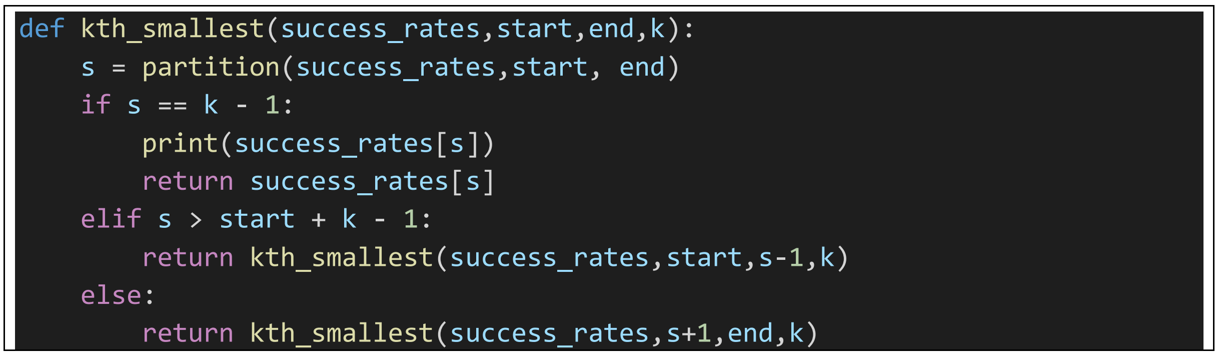
In this question, it is expected to find kth minimum number in the given array. As it stated by our teacher in the lesson, we can use partition to solve this problem because we are not allowed to use "sort". Here is the explanation how this method works: In partition function, it will collect the numbers to the left of the selected pivot. It will go till the end of the array. At the end, it will swap the end index of that subarray and the value at the next location of the latest smallest value of the selected pivot. If the next location is 1 higher than k then the number in that location is our kth smallest element. This method do not sort the whole array, it just finds the highest and puts it into the latest place of that subarray.

There is 2 function. One is recursive, the other one is not recursive but there is a for loop in it.



```python
def partition(arr, start , end):
    p = arr[end]
    i = (start - 1)
    for j in range(start, end):
        if arr[j] <= p:
            i = i + 1
            swap(arr, j , i)
    swap(arr, end , i+1)
    return (i+1)
```

For this function: $\sum_{k=0}^{n-1} 1 = 1 + 1 + .1 + \cdots 1 = n \cdot 1 = \theta(n)$



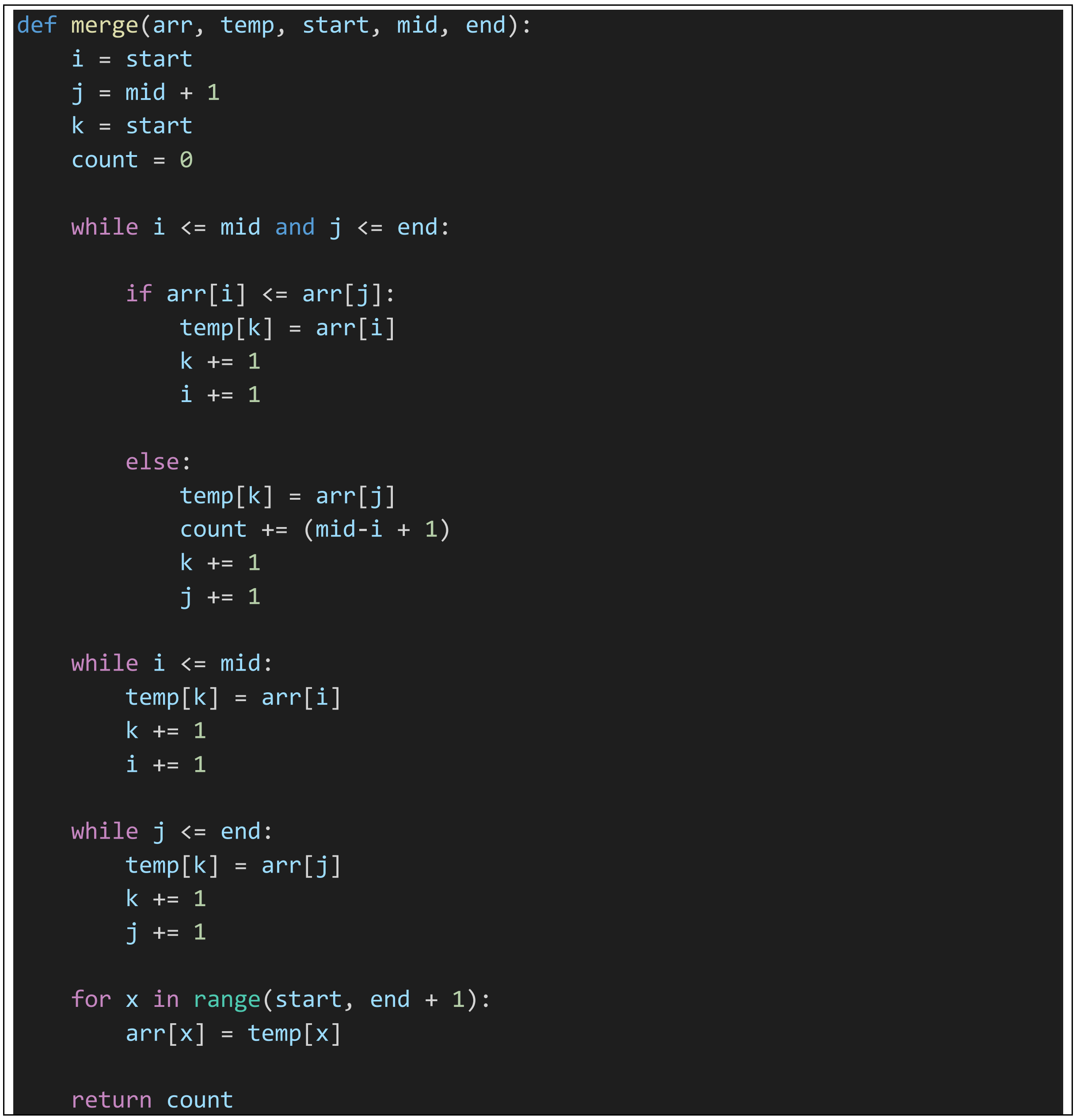
```python
def kth_smallest(success_rates,start,end,k):
    s = partition(success_rates,start, end)
    if s == k - 1:
        print(success_rates[s])
        return success_rates[s]
    elif s > start + k - 1:
        return kth_smallest(success_rates,start,s-1,k)
    else:
        return kth_smallest(success_rates,s+1,end,k)
```

For this function: $T(n) = T(n-1) + O(n)$

- ⇨ $T(1) = 1 \Rightarrow T(2) = T(1) + 0(2) \Rightarrow T(2) = 3$
- ⇨ $T(3) = T(2) + 0(3) \Rightarrow 3 + 3 = 6$
- ⇨ $T(1) = 1 \Rightarrow T(2) = 3 \Rightarrow T(3) = 6 \Rightarrow T(n) = n * \frac{(n+1)}{2}$
- ⇨ $T(n+1) = \frac{(n+1)\cdot(n+2)}{2}$ *is true then*
- ⇨ $T(n+1) = T(n) + O(n+1) \Rightarrow T(n) = \frac{n\cdot(n+1)}{2} =>$
- ⇨ $T(n+1) = \frac{n\cdot(n+1)}{2} + n + 1 = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$ *then the solution of this recurrence is* $O(n^2)$

Q4.

In this question, it is asked to find reverse-ordered pairs in an array by using Divide and conquer. So that is why I need to divide the array and find the reverse ordered pairs and count them. The easiest solution for this problem is Merge Sort. Because merge sort divides an array into a smaller pieces (a single element) and after that it compares the number in the leaf nodes and merges them into a single subarray. While doing a comparison, I can count the number of the reverse ordered pairs. For example: There are 2 subarray: [5,9] and [3,10]. Firstly, 3 and 5 will be compared. [3,5, … , …] is our new temp array. 3 is in higher index and less than 5, then they are reverse-ordered pairs. So we will increase count by (mid – (index of 5) + 1). After that 5 and 10 will be compared, count will be same. 9 and 10 will be compared, count will be same. At the end of the operation, count will be 2.



```python
def merge(arr, temp, start, mid, end):
    i = start
    j = mid + 1
    k = start
    count = 0

    while i <= mid and j <= end:

        if arr[i] <= arr[j]:
            temp[k] = arr[i]
            k += 1
            i += 1

        else:
            temp[k] = arr[j]
            count += (mid-i + 1)
            k += 1
            j += 1

    while i <= mid:
        temp[k] = arr[i]
        k += 1
        i += 1

    while j <= end:
        temp[k] = arr[j]
        k += 1
        j += 1

    for x in range(start, end + 1):
        arr[x] = temp[x]

    return count
```

In this function, there are 4 loops.

- The worst case for the first while loop is O(n). For example: [2,4,6] and [1,3,5] have worst case for this loop.
- Second while loop: $\sum_{i=0}^{mid} 1 = 1 + 1 + 1 \ldots + 1 = \frac{n}{2} \cdot 1 = \frac{n}{2}$
- Third while loop $\sum_{j=0}^{end} 1 = 1 + 1 + 1 \ldots + 1 = \frac{n}{2} \cdot 1 = \frac{n}{2}$
- For loop: $\sum_{i=start}^{end} 1 = 1 + 1 + 1 \ldots + 1 = n \cdot 1 = n$

The complexity of this algorithm is O(n)

```
def _mergeSort(arr, temp, start, end):
    count = 0

    if start < end:

        mid = math.floor((start+end) / 2)

        count += _mergeSort(arr, temp,start, mid)

        count += _mergeSort(arr, temp,mid + 1, end)

        count += merge(arr, temp, start, mid, end)

    return count
```
This function is recursive. The recurrence relation is T(n) = 2T(n/2) + O(n).

- ⇨ We can solve this by using Master's Theorem.
- ⇨ $n^{\log_b a} = n^{\log_2 2} = n^1 = O(n), We\ will\ use\ Second\ Case$
- ⇨ $\theta\left(n^{\log_b a}\, log\, n\right) = \theta(n\, log\, n)$

Q5.

In this question, there are 2 functions. One is coded by using divide and conquer, the second one is brute force. It is asked to code a program to calculate the given exponent of a number. For brute force, I calculated all the possible multiplications. For example: a = 5 and n = 2;

- Mult = 1 (initial)
- Mult = 6 (Loop 1) mutl = a
- Mult = 36 (Loop 2) mult = a * a

For the divide and conquer method, I divided the multiplication to 2 and calculated both sides and find the solution to the problem. a * a * a * a * a * a = a^6 , First three a will be calculated and after that other a's will be calculated. For example: a = 5 and n = 3;

- Mult = 1 (initial)
- Mult = 6 (first recursive call: divide_and_conquer(a,1)) ~ Calculates mult *= a
- Mult = 6 (second recursive call: divide_and_conquer(a,2)) ~ Calculates mult *= a * a
- Mult = 6 (third recursive call: divide_and_conquer(a,1)) ~ Calculates mult *= a
- Mult = 36 (fourth recursive call: divide_and_conquer(a,1)) ~ Calculates mult *= a
- Mult = 216 (After all calls and returns) ~ Calculates (a) * (a * a)

```
def brute_force(a , n):
    mult = 1

    for i in range(0,n):
        mult *= a

    return mult
```

For Brute Force Algorithm:

- ⇨ $\sum_{i=0}^{n-1} 1 = 1 + 1 + 1 \ldots + 1 = n \cdot 1 = n$ = O(n)

```
def divide_and_conquer(a , n):
    mult = 1
    if 1 == n:
        return a
    else:
        mult *= divide_and_conquer(a, math.floor(n/2))
        mult *= divide_and_conquer(a , math.ceil(n/2))

    return mult
```

For Divide and Conquer Algorithm:

- ⇨ Recurrence Relation = T(n) = T(n/2) + O(1)
- ⇨ We can solve this by using Master's Theorem
- ⇨ $n^{\log_a b} = n^0 = 1$, Second case
- ⇨ $O(n^{\log_2 1} \, log\, n) = O(logn)$