CSE 321 HW 4 1901042640 - Emre Sezer

1.
```
def cutting(wire):
    if (wire <= 1):     } Θ(1)
        return 0
    else:                          integer division
        return cutting((wire+1) // 2) + 1   } T(n/2)
```

$T(n) = T\left(\frac{n}{2}\right) + c$ I will apply Master's Theorem.

$a=1,\ b=2,\ f(n) \in \Theta(1)$

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \in \Theta(1)$, Also, $f(n) \in \Theta(1)$.

so, $f(n) \in \Theta(n^{\log_b a}) = 1$

<u>Case 2:</u>

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

At each step, it cuts each piece into 2 pieces. Since, it is possible to cut multiple pieces at the same time. It will continue until each piece's length is 1 meter. It is possible that one of the piece's length is odd and other one's is even. This situation is not a problem. Because, they can be cut in parallel without a problem. For this situation: while calling recursively it does (wire +1 //2). It adds 1 to the wire. If n is an odd number. It won't miss a required cut. If i had done (wire //2), result would be false.

2)

```
def worst_best_rec (arr, low, high, best, worst):
    mid = low + (high - low) // 2
    if(arr[mid] > best):
        best = arr[mid]
    if(arr[mid] < worst):
        worst = arr[mid]
    if(low >= high):
        return (best, worst)
    x = worst_best_rec(arr, low, mid -1, best, worst)
    y = worst_best_rec(arr, mid + 1, high, best, worst)
    a = 0    # best value of the pair
    b = 0    # worst value of the pair
    if(x[0] > y[0]):
        a = x[0]
    else:
        a = y[0]
    if(x[1] < y[1]):
        b = x[1]
     else:
        b = y[1]
    pair = (a,b)
    return pair
def worst_best(arr):
    best = arr[len(arr) // 2]
    worst = arr[len(arr) // 2]
    worst_best_rec(arr, 0, len(arr) - 1, best, worst)
    pair worst_best_rec (arr, 0, len(arr) - 1, best, worst)
    return pair
```

2.

This algorithm is very similar to the binary seach algorithm. But, in binary search, you pick one of the 2 recursive calls and continue with it. At this algorithm, both of the recursive calls will be executed. Because of that: $T(n) = 2T(\frac{n}{2}) + c$

$T(n) = 2T(\frac{n}{2}) + c$ I will apply Master's Theorem.

$a=2$, $b=2$, $f(n) \in \Theta(1)$ (1)

$n^{\log_b a} = n^{\log_2 2} = n^1 = n \in \Theta(n)$ (2)

From (1) and (2)

$f(n) \in O(n)$

Case 1: $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$

It divides problem into 2 parts and executes both of them. At each recursive call it looks on the array's mid element. If it is less than worst or greater than best, updates worst and best. If low is greater or equal to high, returns best and worst as a pair data structure. After executing both of the recursive calls, it checks both pairs returned from recursive calls. It finds the the smallest and greatest values among 2 pairs returned. Then, returns these values as a pair.

3)

```
def partition(arr, l, r):
    p = arr[l]
    s = l
    for i in range(l + 1, r + 1):
        if(arr[i] < p):
            s += 1
            temp = arr[i]
            arr[i] = arr[s]
            arr[s] = temp

    temp = arr[s]
    arr[s] = arr[l]
    arr[l] = temp
    return s

def meaningful(arr, l, r, k):
    pivot = partition(arr, l, r)

    if (pivot - l == k - 1):
        return arr[pivot]

    if (pivot - l > k - 1):
        return meaningful(arr, l, pivot - 1, k)
    else:
        return meaningful(arr, pivot + 1, r, k - pivot + l - 1)
```

3) <u>partition function</u> is the support function, It works like the Lomuto Partitioning algorithm we saw on the course. It partitions array into segments like below and returns s and swaps A[l] and A[s]

l		s s+1	i	r
p	< p	≥ p	?	

p is the pivot value.
s is the last index of the "<p" segment.

<u>meaningful function</u> is the recursive decrease and conquer algorithm. It divides problem into 2 pieces and continues with one of them. It stops when the base condition is fulfilled which is k^{th} smallest element is found in the array. There are k+1 meaningless test values and first meaningful test value is the k^{th} one, (if array is sorted in increasing order), So, my algorithm returns the first meaningful test value which is the k^{th} smallest one. This algorithm is very similar to quickselect algorithm, we saw on the course.

$T(n) = T(\frac{n}{2}) + cn$ I will apply Master's Theorem

$a = 1,\ b = 2,\ f(n) \in \Theta(n)$ (1)

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \in \Theta(1)$ (2)

From (1) and (2):

$f(n)\ \Omega\ \Theta(1)$ and

$a f(\frac{n}{b}) \leq c f(n)$

$f(\frac{n}{2}) \leq c f(n)$

$\frac{cn}{2} \leq c_1 c n$

$c_1 \leq \frac{1}{2},\ c_1 < 1, n > n_0$

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

4)

```
def find_rop(arr):
    temp_arr = []
    temp_arr.extend(arr)
    counter = find_rop_rec(temp_arr, 0, len(arr) - 1)
    return counter


def find_rop_rec(arr, low, high):
    mid = low + (high - low) // 2
    if(low > high):
        return 0
    temp = reverse_ordered_pairs(arr, mid)
    x = find_rop_rec(arr, low, mid - 1)
    y = find_rop_rec(arr, mid + 1, high)
    return x + y + temp


def reverse_ordered_pairs(arr, i):
    counter = 0
    for j in range(i, len(arr)):
        if(arr[i] > arr[j]):
            counter += 1
    return counter
```

4. This algorithm is similar to the binary search algorithm. In binary search you pick one of the 2 recursive calls and continue with it. At this algorithm, both of the recursive calls will be executed. Because of that $T(n) = 2T(\frac{n}{2}) + cn$.

$T(n) = 2T(\frac{n}{2}) + cn$ I will apply Master's Theorem.

$a=2, b=2, f(n) \in \Theta(n)$ (1) From (1) and (2)

$n^{\log_b a} = n^{\log_2 2} = n^1 = n \in \Theta(n)$ (2) $f(n) \in \Theta(n)$

Case 2: $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$

It divides problem into 2 parts and executes both of them. At each recursive call it will call reverse-ordered-pairs for mid and arr. If low is greater than high returns 0. Function returns summation of returns of both recursive calls and return of reverse-ordered-pairs for the current mid. reverse-ordered-pairs works with $\Theta(n)$ time complexity. It returns the number of reverse ordered pairs for the input index. Recursive function simply, divides array into 2 parts and calculates mid. Then, recursively calculates other 2 parts. This is a divide and conquer algorithm.

find-rop function is the main function that backs up array in $\Theta(n)$ time and calls recursive function.

$T(n) = \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

5.

a) Brute Force:

```
def exponent_brute_force(a, n):
    result = 1                    } Θ(1)
    for i in range(0, n):         }
        result *= a               } Θ(n)
    return result                 }
```

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

It simply calculates a^n by calculating $a * a$ n-times.

b)

```
def exponent_divide_and_conquer(a, n):
    if (n==0):         } Θ(1)
        return 1       }
    if (n % 2 == 0):
        return exponent_divide_and_conquer(a*a, n//2)
    else:
        return a * exponent_divide_and_conquer(a*a, n//2)
```

If n is even it calls itself with $(a^2, \frac{n}{2})$. At some point n will be equal to 0 and return 1. If n is odd, it will return a * exponent_divide_and_conquer(a*a, n//2). With that approach we don't calculate a^n with linear time. We calculate it in logarithmic time. At each step it divides problem into 2 pieces. It continues with one of them.

For example: it can calculate 2^4 in 2 steps.

$(2^2)^2$. First calculates 2^2. Then $(2^2)^2$.

$T(n) = T(\frac{n}{2}) + c$ I will apply Master's Theorem.

$a=1$, $b=2$, $f(n) \in \Theta(1)$ (1)

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \in \Theta(1)$ (2)

Case 2:

From (1) and (2)

$f(n) \in \Theta(n^{\log_b a}) = \Theta(1)$

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

$= \Theta(\log n)$