Yusuf Can Kan
161044007

CSE 221 - HW4

1) The algorithm looks first match first. After that it tries to find a 0010 pattern in rest of the text.

Text: 0000 ... 0 (length n)

Pattern: 0010

→

0 0 0 0
0 0 1 0

when it tries to find pattern it will do 3 comparisons each time.

↓

Algorithm moves this process $n-3$ times.

$3 * (n-3) = 3n-9$ (3: comparisons, $(n-3)$: times.)

$\Theta(n)$

Worst case 3 bit input patter is 001 because, every time when it find a match it will compare 3 bit. since first two bit will match it will also compare third bit too.

2) In order to apply brute-force we need to find all hamiltonian circles and find the one that has minimum cost.

ABDCEA → 21
ABDECA → 27
ABCDEA → 22
ABCEDA → 25
ABEDCA → 19
ABECDA → 16
ACDBEA → 18
ACDEBA → 19
ACEDBA → 27
ACEBDA → 21
ACBEDA → 22
ACBDEA → 27
ADBCEA → 24
ADBECA → 21
ADCBEA → 16
ADCEBA → 16
ADECBA → 25
ADEBCA → 22

AEDBCA → 27
AEDCBA → 22
AECBDA → 24
AECDBA → 21
AEBDCA → 18
AEBCDA → 16

Min cost path => 16

paths => ABECDA
ADCBEA
ADCEBA
AEBCDA

3)
```
def log(n):
    if n==1:
        return 0;
    else
        return log(n/2)+1;
```

base case is $n == 1$.

last line of the code divides input into 2. So;

$A(n) = A(n/2) + 1$;

From master theorem;

$a = 1$, $b = 2$, $d = 0$

$a = b^d$

$1 = 2^0$

$1 = 1$

$\Theta((\log n))$

4) In this problem I chose to follow a decrease by constant approach. My approach is, taking 1 bottle every time and checking its weight one-by-one. I can express my algorithm as;

```
procedure find(bottles):
    while(take next bottle until no bottle left):
        if(bottle setted incorrect):
            stop procedure
```

Best Case: In best case we can find the bottle in first iteration. So;

$B(n) = O(1)$

Worst case: In worst case we can find the bottle in last iteration.

$W(n) = \Theta(n)$

Average case:

Prob. that i iteration will be done; $P(i)$

so

$$A(n) = \sum_{i=1}^{W(n)} i \cdot P(i)$$

Probability that $x = i$'th bottle is $\frac{p}{n}$ so;

$$A(n) = \sum_{i=1}^{n} i \cdot \frac{p}{n} = \frac{p}{n} \cdot \frac{n(n-1)}{2} = \frac{pn-p}{2} \Rightarrow n \in \Theta(n)$$

Average case.

5) procedure findkth(arr1[0:n], arr2[0:m], k):

```
if n==0:
    return arr2[k]
if m==0:
    retun arr1[k]

middle1 = n//2
middle2 = m//2

if (middle1+middle2) < k:
    if arr1[middle1] > arr2[middle2]:
        return findk(arr1, arr2[middle2+1:], k-middle2-1)
        // Recursive call without taking unnecessary half of the
        // array.
    else
        return findk(arr1[middle1+1:], arr2, k-middle1-1)
else:
    if arr1[middle1] > arr2[middle2]:
        return findkth(arr1[:middle1], arr2, k)
    else
        return findkth(arr1, arr2[:middle2], k)
```

The main goal of this algorithm, in each iteration always try to eleminate half of the one array. In order to accomplish this it looks to the length of half of the arrays. If sum of this lengths are smaller then the k, it means that one of the upper half of the arrays can be ignored. It will compare the middle elements and removes the upper half of small array. At the same time we decrease the kth index with respect to removed element count. With this we obtain a subproblem for our problem. We do the same process for other cases such as, If half of the arrays length are smaller and middle element of arr1 is bigger than other, we ignore upper half of arr 1.

We apply same ignoring process for lower half of the arrays if sum of lengths (half of the arrays length) are bigger than the k.

∝ If lower half of the array lengths sum < k
  ∝ Compare middle elements and ignore the lower half of the smaller one.

∝ Else
  ∝ compare middle elements and ignore the upper half of the bigger one.

---

lets say m and n are the lengths of these arrays. Each time we ignore lower half of the array 1 it becomes m/2 and, each time we ignore lower or upper half of the array 2 it becomes n/2. In worst case both of the arrays divides until there is 1 element left. So all the elements divides into two until the end.

$$\theta(\log n + \log m)$$

↙ for dividing array 1

↘ for dividing array 2.