CSOR W4231 Analysis of algorithms I
Assignment 2

Xijiao Li (xl2950)

September 29, 2020

Problem 1

1.

During the i-th step, we have already seen $i-1$ numbers appear in the former $i-1$ cards, so we have $n-i+1$ possible numbers to choose and
$P(i\text{th guess is correct}) = \frac{1}{n-i+1}$

2.

Calculate the expectation:

$$E[X] = \sum_{i=1}^{n} \frac{1}{n-i+1} = \frac{1}{n} + \frac{1}{n-1} + ... + 1 = H_n$$

This is a term of harmonic sequence, and we can use it property

$$E[X] = H_n < \int_1^n \frac{1}{x} dx + 1 = \ln n + 1 = O(\log n)$$

Problem 2

1.

Proof by contradiction.
Assume that m is the majority element of A and it is a majority element of neither the first half of the array nor the second half of the array.
Let t_1, t_2 denote the times of appearances of m in the first and second halves of the array, respectively. So by definition, we have $t_1 \leq \frac{n}{4}$ and $t_2 \leq \frac{n}{4}$. Therefore, $t_1 + t_2 \leq \frac{n}{4} + \frac{n}{4} = \frac{n}{2}$, which is a contradiction, since m is the majority element of A and $t_1 + t_2 > \frac{n}{2}$. Thus we have proved the original statement.???

Divide and conquer algorithm:
I will use recursion: repeatedly divide the array in half, and call MajorityElement() on each half. When there is only one single element in the array, that element is returned as the majority element. At any other level, there will be two return values from its two recursive calls.

```
MajorityElement(A)
  if length(A) == 0:
    return null
  if length(A) == 1:
    return A[1]
  left <- MajorityElement(A[1...n//2])
  right <- MajorityElement(A[n//2+1...n])
  if left == null and right == null:
    return null
  else:
    left = CheckMajority(left, A)
    right = CheckMajority(right, A)
    return left or right // there can be at most 1 majority

CheckMajority(x, A)
  if x == null:
    return null
  set count <- 0
  for a in A:
    if x == a:
      count <- count + 1
  if count > length(A)//2:
    return x
  else:
    return null
```

The recurrence can be written as

$$T(n) = 2T(\frac{n}{2}) + 2n$$

Apply Master Theorem, we have $a = 2, b = 2, f(n) = 2n$,
because $n^{\log_b a} = n^{\log_2 2} = n$, $f(n) = \Theta(n)$, case 2 of the Master Theorem applies and we have $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$

2.

a.
Assume that array A has $2n$ elements, so there will be n pairs. Also assume that x is the majority element of A and it appears t times among the $2n$ elements, where $t > n$ by definition. Let k denotes the number of equal pairs that are kept (i.e. the number of remaining elements), and d denotes the number of unequal pairs that are discarded, so $2k + 2d = 2n$.
In the worst case scenario, x appears in every single discarded pair, so there are $\frac{t-d}{2}$ remaining x.

$$\frac{t-d}{2} > \frac{n-d}{2} = \frac{k}{2}$$

therefore x is still the majority element among the elements that are kept.

b.
Assume that array A has $2n+1$ elements, so there will be n pairs and one leftover element. Also assume that x is the majority element of A and it appears t times among the $2n + 1$ elements, where $t > n$ by definition. The rules are:

- if the last element is the majority element, keep it;
- otherwise, discard it.

Next, we prove that the property of part (a) holds in this case too.
Let k denotes the number of equal pairs that are kept, and d denotes the number of unequal pairs that are discarded, so $2k + 2d + 1 = 2n + 1$. In the worst case scenario, x appears in every single discarded pair.
There are two situations:

1. The last element is the majority element.
   Following the rule, we keep it. Therefore, after the operation there will be $k + 1$ elements left, among which $\frac{t-d-1}{2} + 1 = \frac{t-d+1}{2}$ are x.

   $$\frac{t-d+1}{2} > \frac{n-d+1}{2} = \frac{k+1}{2}$$

   The property of part (a) holds.

2. The last element is the majority element.
   Following the rule, we discard it. Therefore, after the operation there will be k elements left, among which $\frac{t-d}{2} = \frac{t-d}{2}$ are x.

$$\frac{t-d}{2} > \frac{n-d}{2} = \frac{k}{2}$$

   The property of part (a) holds.

Thus, we have proved that the property of part (a) always holds under this rule.

c.
Code:

```
MajorityElement(A, partial)
  set n <- length(A)
  if n == 0:
    return null
  if n == 1:
    return A[1]
  initialize array B
  set i <- 1
  set j <- 1
  while i < n-1:
    if A[i] == A[i+1]
      set B[j] <- A[i]
      set j <- j+1
    set i <- i+2
  if i == n-1:
    if partial == true or A[i] == A[i+1]
      set B[j] <- A[i]
    else:
      set j <- j-1
  else:
    set B[j] <- A[i]
    set partial <- true
  return MajorityElement(B[1...j], partial)
```

The recurrence can be written as

$$T(n) = T(\frac{n}{2}) + n/2$$

Apply Master Theorem, we have $a = 1, b = 2, f(n) = n/2$
because $n^{\log_b a+\epsilon} = n^{\log_2 1+1} = n, f(n) = \Omega(n)$, case 3 of the Master Theorem applies and we have $T(n) = \Theta(f(n)) = \Theta(n)$.

Problem 3

We will use the deterministic selection told in lecture.
First, we preprocess the array so that each element now becomes a tuple, with the first value being the distance to the first element of A and the second value being the original element value. When we do selection and median-finding later, we do comparison based only on the first value of the tuple.

```
K-Closet(A, k)
  Preprocess(A)
  set res <- Select(A, k)
  return res[2] // return the original value of elements

Preprocess(A)
  for a in A:
    set x <- absolute(A[1]-a)
    set a <- {x, a}

Select(A, k)
  set n <- length(A)
  set q <-Median(A, n)
  if k == n/2:
    return q
  Partition all elements in A based on q into A1, A2
  if k < n/2:
    set res <- Select(A1, k)
  else:
    set res <- Select(A2, i-k)
  return res[2] // return the original value of elements

Median(A, n)
  Divide n elements into groups of 5
  Select median of each group // return n/5 selected elements
  Use Select recursively to find median q of selected elements
```

The partition goes through all the elements once and thus runs in $O(n)$ time. From the lecture, we know the Median-Selection function runs in $O(n)$ time. So

in the worst case, the recurrence of `Select(A, k)` is

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + cn \\ &= c(n + n/2 + n/4 + n/8 + ... + T(1)) \\ &\leq 2cn \\ &= O(n) \end{aligned}$$

The preprocess goes through all the elements once and therefore overall the algorithm runs in $O(n)$ time.

Problem 4

We will again use the deterministic selection told in lecture and the `Select(A, k)` function. Its pseudocode is the same as the one I wrote in Problem 3. Without loss of generality assume that n and k are powers of 2. Basically, we want to first find the $k-1$ elements in a set that partition the set into k equal-sized sets A_1, A_2, ...A_k such that all elements in A_i are smaller than all elements in A_{i+1} (the last set may have less than k elements).

```
K-Rank(A, k):
  set n <- length(A)
  Partition A using K-Partition(A, 1, n, k)
  Select the max value of A1, A2, ..., A(k-1)

K-Partition(A, p, r, k):
  if k > 1 then
    q = Select(A, (p+r)/2)
  K-Partition(A, p, (p+r)/2, k/2)
  K-Partition(A, (p+r)/2+1, r, k/2)
```

The Maximum-selection goes through all the elements once on a subset `Ai` one and thus runs in $O(n)$ time. From the lecture, we know the Selection function runs in $O(n)$ time. So in the worst case, the recurrence is

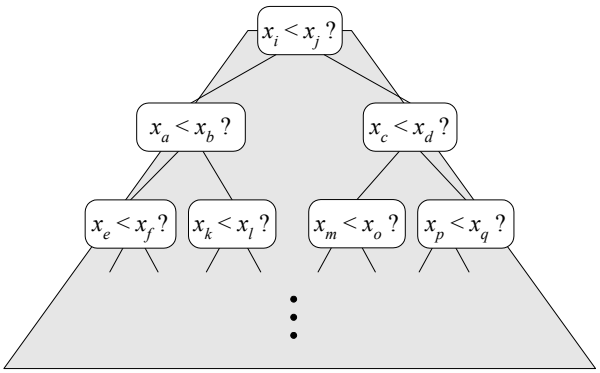
$$T(n) = 2T(\frac{n}{2}) + cn$$

Apply Master Theorem, we have $a = 2, b = 2, f(n) = cn$, case 2 of the Master Theorem applies and we have $T(n) = \Theta(n \log n)$.

Problem 5

a.
There are $\frac{(2n)!}{(2n-n)!}$ ways to select n elements and form an list with a random order, and we need to divide this number with the number of all possible orders (the rest elements will automatically be grouped as the other ordered list). So there are $\frac{(2n)!}{n!n!} = \binom{2n}{n}$

b.



Any decision tree to merge two sorted lists must have at least $\binom{2n}{n}$ leaf nodes, so it has height at least $\log\left(\frac{(2n)!}{n!n!}\right)$

$$\begin{aligned}\log\left(\frac{(2n)!}{n!n!}\right) &= \log\left((2n)!\right) - 2\log(n!) \\ &= \sum_{k=1}^{2n}\log(k) - 2\sum_{k=1}^{n}\log(k)\end{aligned}$$

Since $\log(k)$ is monotonically increasing, we use Approximation by integrals formula to approximate the sum:

$$\begin{aligned}\int_0^n \log(x)dx \leq \sum_{k=1}^{n}\log(k) &\leq \int_1^{n+1} log(x)dx \\ \frac{n\ln(n) - n}{\ln 2} \leq \sum_{k=1}^{n}\log(k) &\leq \frac{(n+1)\ln(n+1) - n}{\ln 2}\end{aligned}$$

Therefore:

$$\begin{aligned}
\log\left(\frac{(2n)!}{n!n!}\right) &= \sum_{k=1}^{2n}\log(k) - 2\sum_{k=1}^{n}\log(k) \\
\log\left(\frac{(2n)!}{n!n!}\right) &= \log\left((2n)!\right) - 2\log(n!) \\
&\geq \frac{2n\ln(2n) - 2n}{\ln 2} - 2\frac{(n+1)\ln(n+1) - n}{\ln 2} \\
&= 2n + 2n\log(n) - 2n\log(n+1) - 2\log(n+1) \\
&= 2n + 2n\log(n/(n+1)) - 2\log(n+1) \\
&= 2n - o(n)
\end{aligned}$$

c.
Suppose the consecutive elements are x_1, x_2 and $x_1 < x_2$, and the two lists are A_1: $[\ldots a_{1,i}, x_1, a_{1,i+1}\ldots]$, and A_2 : $[\ldots a_{2,i}, x_2, a_{2,i+1}\ldots]$. Suppose that there is another pair of two sorted arrays A_1': $[\ldots a_{1,i}, x_2, a_{1,i+1}\ldots]$, and A_2' : $[\ldots a_{2,i}, x_1, a_{2,i+1}\ldots]$, which is obtained by just swapping x_1, x_2 between lists. If x_1, x_2 are not being compared during merging, then there is no way to distinguish between A_1, A_2 and A_1', A_2', so we then require to add a different action to be taken in order to correctly merge the lists, and thus x_1, x_2 must be compared.

d.
Suppose the finally sorted list is A_f: $[a_1, a_2, \ldots, a_{2n}]$. From part c, we know that we must compare a_1 with a_2, a_2 with a_3, $\ldots$, a_{2n-1} with a_{2n}. So the lower bond is $2n - 1$.

Problem 6

We will use the Bucket Sort told in lecture.
Code:

```
SortLists(L1...Lr):
  initialize a list B with size n
  initialize each element bi in B with an empty linked list
  for j in 1, 2, ..., r:
    for element i in Lj:
      bi.add j
  empty L1, L2, ..., Lr
  for bi in B:
    for j in bi:
      Lj.add i
  return L1 ... Lr
```

The first nested for loop iterates over all elements in these r lists and perform a O(1) operation on each of them, so it runs in O(n) time since the sum of the sizes of the r lists is n. The second nested for loop iterate B as well as the linked lists in `B`, so it runs in O($n + n$) times since the sum of the sizes of the linked lists is equal to the total number of elements, which is n.
Since all other operations run in O(n) time, so overall this algorithm runs in O(n) time.

Reference

CLRS, chapter 8&9