CSORE 4231 ANALYSIS OF ALGORITHMS I Homework 2

Francis Zhang xz3279

February 17, 2024

Problem 1

Mo' Money Mo' Problems.
(a) Consider the following game. There are n cards with the numbers $1, \ldots, n$ written on them, where each of the numbers $1, \ldots, n$ is written on exactly one of the cards. The cards are face down on the table, so we do not see their numbers. We turn over the cards one by one. Before turning each card, you guess the number written on the card; then you see the card, and you gain 1 dollar if you guess correctly and 0 if incorrectly. A random strategy is to guess in each step uniformly at random a number that has not appeared so far in the previous cards. What is the probability that the guess in the i^{th} step is correct?
(b) Using indicator random variables, show that the expected total number of dollars that you gain using the random strategy from part (a) is $\Theta(\log n)$.
(c) CLRS 5.3-2
Professor Kelp decides to write a procedure that produces at random any permutation besides the identity permutation. He proposes the following procedure:

```
PERMUTE-WITHOUT-IDENTITY(A)
1   n = A.length
2   for i = 1 to n - 1
3       swap A[i] with A[RANDOM(i + 1, n)]
```

Does this code do what Professor Kelp intends?
(d) CLRS 5.3-4
Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```
PERMUTE-BY-CYCLIC(A)
1   n = A.length
2   let B[1..n] be a new array
3   offset = RANDOM(1, n)
4   for i = 1 to n
5       dest = i + offset
6       if dest > n
7           dest = dest - n
8       B[dest] = A[i]
9   return B
```

Show that each element $A[i]$ has $\frac{1}{n}$ probability of winding up in any particular position in B. Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.
Solution:
(a) As the random strategy used here is to guess in each step at random a number that has not appeared so far in the previous cards. When i = 1, the probability is $\frac{1}{n}$. When i = 2, as we know the first card, the probability is $\frac{1}{n-1}$... When i = i, the probability is $\frac{1}{n-i+1}$ as we know the previous $i-1$ cards, we are picking a number from the rest: $n-(i-1) = n-i+1$ cards.
(b) Recall from lecture, Let A be an event, the indicator variable $I\{A\}$ is defined by:

$$I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

Let Y be a random variable that denotes the guess of the card. Let X be the i.r.v. that counts earned dollars. I represents the earned dollars. Then:

$$I\{Y \text{ is i}\} = \begin{cases} 1 & \text{if the card is actually i} \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} E[X] &= \sum_{i=1}^{n} Pr(Y \text{ is the card after turning}) * \frac{1}{n-i+1} * 1 + \sum_{i=0}^{n} Pr(Y \text{ is the card after turning}) * \frac{n-i}{n-i+1} * 0 \\ &= \sum_{i=0}^{n} Pr(Y \text{ is the card after turning}) * \frac{1}{n-i+1} \\ &= \frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{1} \\ &\approx \ln n = \Theta(\log n) \end{aligned}$$

(c) NO. Consider the case where $A = \{1, 2, 3\}$. The permutation $\{3, 2, 1\}$ should be a possible output with some possibility, but cannot be generated by this code. Swap $A[1]$ with $A[2]$ we cannot make it, as $A[1]$ will never be altered after the first iteration. However, if we swap $A[1]$ and $A[3]$, we have $\{3, 2, 1\}$. During the second iteration, we have no choice but swap $A[2]$ and $A[3]$.
In general for $A.length = n$, where $A = \{1, 2, 3, ..., n\}$, we can never have $\{n, ..., 1\}$ by this procedure. As to keep $A[1] = n$, we need to swap $A[1]$ and $A[n]$. Now $A[n] = 1$. After those iterations, the last iteration must be $A[n-1]$ and $A[n]$, 1 just goes away.
(d) Analyzing each element $A[i]$ has $\frac{1}{n}$ probability of winding up in any particular position in B from the for loop. Starting from $i = 1$, as dest = i + offset. Where offset = `RANDOM(1,n)`, as for each possibilities offset can be, those entries $1, ..., n$ all have possibility $\frac{1}{n}$. So, for entries $2, ..., n+1$ of dest, they all have possibility $\frac{1}{n}$. Also recall line 6 and 7 of the code, if dest > n, then dest = dest - n. That makes the entry $n+1$ with possibility $\frac{1}{n}$ becomes the entry 1 with possibility $\frac{1}{n}$. So, $A[1]$ has $\frac{1}{n}$ probability of winding up in any particular position in B. Similarly, in general all i's entry if happens dest > n, those bigger than n entries would be minus n and also have the same possibility $\frac{1}{n}$. That is ach element $A[i]$ has $\frac{1}{n}$ probability of winding up in any particular position in B.
However, it can still not showing the resulting permutation is uniformly random. We randomed offset, but in the loop, offset can not be changed. That is for $A = \{1, 2, 3\}$, B can only be consecutive 3 numbers from the array 123123123...123, which is when offset = 1, $B = \{3, 1, 2\}$, offset = 2, $B = \{2, 3, 1\}$, and offset = 3, $B = \{1, 2, 3\}$. But no possibility of B to become $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{3, 2, 1\}$.

Problem 2

Heapsort Correctness. CLRS 6.4-2
Argue the correctness of HEAPSORT using the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the n - i largest elements of $A[1..n]$, sorted.

Solution:
Initialization: $i = A.length$, $A[1, ..., n]$ do contain n smallest elements of A, $n - i = n - n = 0$ so we don't need to worry about subarray at the moment.
Maintenance: After applying BUILD-MAX-HEAP(A), $A[1]$ is the largest among $A[1, ..., n]$, exchange with A[i] and put it out of the heap. After MAX-HEAPIFY(A,1), the new $A[1]$ is smaller than the old $A[1]$ and is greater then rest elements in the heap and the index is just 1 before the old $A[1]$. So it is $A[1, ...i]$ represents unsorted i smallest elements of A, and $A[i+1, ..., n]$ contains the $n - i$ largest elements of A, sorted.
Termination: When $i = 2$, $A[1]$ exchange with $A[2]$ and put out of the heap. The loop terminates. The only $A[1]$ left in the heap and the sorted $A[2, ..., n]$. Where $A[1]$ is smaller than $A[2, ..., n]$. HEAPSORT works and sorted the entire A.

Problem 3

Merging sorted lists. CLRS 6.5-9
Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)
Solution:
That is in total $n * O(\lg k) + O(k) + O(1) = O(n \lg k)$ based on Algorithm 5 below.

Algorithm 1 Min-Heapify

```
1: function MIN-HEAPIFY(A, i)
2:     l ← LEFT(i)
3:     r ← RIGHT(i)
4:     if l ≤ A.heap-size & A[l] < A[i] then
5:         smallest ← l
6:     else
7:         smallest ← i
8:     end if
9:     if r ≤ A.heap-size & A[r] < A[smallest] then
10:        smallest ← r
11:    end if
12:    if smallest ≠ i then
13:        exchange A[i] with A[smallest]
14:        MIN-HEAPIFY(A, smallest)
15:    end if
16: end function                          ▷ O(lg n) Recall from CLRS.
```

Algorithm 2 Build-Min-Heap

```
1: function BUILD-MIN-HEAP(A)
2:     A.heap-size = A.length
3:     for i = ⌊A.length/2⌋ downto 1 do
4:         MIN-HEAPIFY(A, i)
5:     end for
6: end function                           ▷ O(n) Recall from CLRS.
```

Algorithm 3 Heap Decrease Key and Min-Heap Insert

```
1: function HEAP-DECREASE-KEY(A, i, key)
2:     if key > A[i] then
3:         error "new key is larger than current key"
4:     end if
5:     A[i] ← key
6:     while i > 1 and A[PARENT(i)] > A[i] do
7:         exchange A[i] with A[PARENT(i)]
8:         i ← PARENT(i)
9:     end while
10: end function                                   ▷ O(lg n) Recall from CLRS.

11: function MIN-HEAP-INSERT(A, key)
12:     A.heap-size ← A.heap-size + 1
13:     A[A.heap-size] ← ∞
14:     HEAP-DECREASE-KEY(A, A.heap-size, key)
15: end function                                   ▷ O(lg n) Recall from CLRS.
```

Algorithm 4 pop

```
1: function POP(A)
2:     if A.length < 1 then
3:         error there is nothing to pop in A
4:     end if
5:     X = A[1]
6:     A = A[2, ...]
7:     return X
8: end function                                    ▷ O(1)
```

Algorithm 5 Merge-K-Lists

```
1: function MERGE-K-LISTS(A_1, A_2, ..., A_k)
2:     C = []                                      ▷ O(1)
3:     B = [POP(A_1), POP(A_2), ..., POP(A_k)]     ▷ k * O(1)
4:     BUILD-MIN-HEAP(B)                           ▷ O(k)
5:     while B.length ≠ 0 do                       ▷ pass all elements we are discussing in this question:n
6:         smallest = POP(B)                       ▷ O(1)
7:         C = C ∪ {smallest}                      ▷ O(1)
8:         if A_smallest's origin is not empty then
9:             nextElement = POP(A_smallest's origin)   ▷ O(1)
10:            MIN-HEAP-INSERT(B, nextElement)          ▷ O(lg k)
11:        end if
12:    end while
13:    return C                                    ▷ O(1)
14: end function
```

Problem 4

Other versions of Selection.
In the deterministic selection algorithm, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into:

a. groups of 7?

b. groups of 3?

For each case, provide the worst-case running time.
Solution: Recall from CLRS Page220-222, for groups of 5 case:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140 \\ T(\left\lceil \frac{n}{5} \right\rceil) + T\left(\frac{7n}{10} + 6\right) + O(n) & \text{if } n \geq 140 \end{cases}$$

Where $T\left(\frac{7n}{10} + 6\right)$ comes from:

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 \tag{1}$$

At least half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least $\frac{3n}{10} - 6$. So for worst case, we assume all the rest is less than x and doing `SELECT`. And 140 is just a guess, so we can skip the guess temporally.

Think about groups of k. (1) becomes (2) in general:

$$\left\lceil \frac{k}{2} \right\rceil \left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{k} \right\rceil \right\rceil - 2\right) \geq \frac{n}{4} - k \tag{2}$$

At least $\frac{n}{4} - k$ are less than x, so at most $\frac{3n}{4} + k$ are greater than x.
So when n is greater than some constant, we have $T(n) \leq T(\lceil \frac{n}{k} \rceil) + T(\frac{3n}{4} + k) + O(n)$. Doing some substitution:

$$\begin{aligned} T(n) &\leq T(\left\lceil \frac{n}{k} \right\rceil) + T(\frac{3n}{4} + k) + O(n) \\ &\leq c\left\lceil \frac{n}{k} \right\rceil + c(\frac{3n}{4} + k) + an \\ &\leq \frac{cn}{k} + c(\frac{3n}{4} + k) + an \\ &= cn + (\frac{-cn}{4} + \frac{cn}{k} + ck + an) \\ &\leq cn \end{aligned}$$

The last step holds if $\frac{-cn}{4} + \frac{cn}{k} + ck + an \leq 0$, which is $c(-\frac{n}{4} + \frac{n}{k} + k) + an \leq 0$. As c and a are positive integers, it could be $-\frac{n}{4} + \frac{n}{k} + k \leq 0$. Let $-\frac{n}{4} + \frac{n}{k} + k = f(k)$. We can see that $f(4) = 4 > 0$, $f(5) = -\frac{n}{20} + 5 \leq 0$ when $n \geq 100$. So we can always find a n_0 such that $f(k) \leq 0$ when $k > 5$. When $T(n) \leq cn$, the running time is linear. So, groups of 7 runs linear time but groups of 3 does not.($T(n) = T(n/3) + T(\frac{2n}{3}) + O(n)$ is $\Theta(n \lg n)$).

Problem 5

Weighted medians CLRS 9-2
For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the **weighted (lower) median** is the element x_k satisfying

$$\sum_{x_i<x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i>x_k} w_i \leq \frac{1}{2}.$$

For example, if the elements are 0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2 and each element equals its weight (that is, $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = \frac{1}{n}$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The **post-office location problem** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the **Manhattan distance** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Solution:

a. When $x_1, ..., x_n$ are weighted equally where $w_i = \frac{1}{n}$ for $i = 1, ...n$, assume n is odd first. The median $x_i = x_{\frac{n+1}{2}}$, $x_{i-1} = x_{\frac{n-1}{2}}$, where $\sum_{i=1}^{\frac{n-1}{2}} \frac{1}{n} = \frac{n-1}{2n} < 1/2$ and $\sum_{i=\frac{n+3}{2}}^{n} \frac{1}{n} = \frac{n-1}{2n} \leq 1/2$ satisfies x_i is also the weighted median. Assume n is even, The median $x_i = x_{\frac{n}{2}}$. It is obvious that $\sum_{i=1}^{\frac{n}{2}} \frac{1}{n} = \sum_{i=\frac{n+1}{2}}^{n} \frac{1}{n} = \frac{1}{2}$. So the sum of the weights before x_i is $\frac{1}{2} - \frac{1}{n} < \frac{1}{2}$.

Algorithm 6 Weighted-median

b.
```
1:  function WEIGHTED-MEDIAN(x_1, x_2, ..., x_n)
2:      w = 0
3:      HEAP-SORT(x_1, x_2, ..., x_n)                    ▷ O(n lg n)
4:      for i = 1 upto n − 1 do                          ▷ n * O(1)
5:          w = w + x_i
6:          if w > 1/2 then
7:              return x_i
8:          end if
9:      end for
10: end function                                         ▷ O(n lg n) + O(n) = O(n lg n)
```

c. We can use the `SELECT` function below, where `MEDIAN` is doing similarly as CLRS 9.3 and Problem 4, based on the size of n, we can then choose how many groups to divide.

Algorithm 7 Sum

```
1: function SUM(A)
2:     total = 0
3:     for i = 1 upto n do
4:         total = total + A[i]
5:     end for
6: end function                                   ▷ O(n)
```

Algorithm 8 Weighted-median

```
1: function WEIGHTED-MEDIAN(A, i, n)
2:     if n = 1 then
3:         return A[1]
4:     end if
5:     p ← MEDIAN(A)
6:     L ← {x ∈ A : x ≤ p}
7:     H ← {x ∈ A : x > p}
8:     temp ← SUM(L)
9:     if i ≤ temp then
10:        return SELECT(L, i, |L|)
11:    else
12:        return SELECT(H, i − temp, |H|)
13:    end if
14: end function          ▷ Similar as SELECT function we seen before, Θ(n)
```

d. Let p be the minimizer, and suppose that p is not the weighted mean. And let $\epsilon > 0$ where $\epsilon = \min_i(|p - p_i|)$ and $p_m > p$. Then we have:

$$\sum_{i=1}^{n} w_i d(p+\epsilon, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + \epsilon \left(\sum_{p_i<p} w_i - \sum_{p_i>p} w_i \right) < \sum_{i=1}^{n} w_i d(p, p_i)$$

Since for p_m, $\sum_{p_i<p_m} w_i \leq \sum_{p_i>p_m} w_i$ and $p_m > p$, so $\sum_{p_i<p} w_i - \sum_{p_i>p} w_i < 0$. Then we find out another minimizer, which contradicts the assumption.

e. Manhattan distance is just 2-d variation of part(d.):

$$\sum_{i=1}^{n} w_i d(p, p_i) = \sum_{i=1}^{n} w_i |p_x - (p_i)_x| + \sum_{i=1}^{n} w_i |p_y - (p_i)_y|.$$

It will suffice to minimize each sum separately, which we can do since we choose p_x and p_y individually. The optimal answer here is $p = (p_x, p_y)$ where p_x is the weighted mean of the x-coordinates of the p_i's and p_y is the weighted mean of the y-coordinates of the p_j's.