CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 3
Submit via Gradescope

Name: Xuran Feng
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(10 points)*** *Exercise 8.1-3.*

Solution:
From a decision tree model, we know that leaf node represents a possible output for input with size n, and there could be $n!$ outputs and one of them is a valid sorted output. In other words, we need to provide $n!$ candidate inputs for a size n input to ensure we can get a valid sorted sequence. The best case for running a comparison sort algorithm is linear time and we need to compute the largest number of leaf nodes that can have heights of $\theta(n)$. Because a decision tree is a binary tree, the maximum leaf nodes with $\theta(n)$ height is $2^{\theta(n)} = 2^{cn}$, if we can prove the maximum number of leaf nodes with $\theta(n)$ height is $\leq \frac{n!}{2}$, then we finish our proof. To prove $2^{cn} \leq \frac{n!}{2}$, we know $\frac{(\frac{n}{2})^{\frac{n}{2}}}{2} \leq \frac{n!}{2}$, so we need to prove $2^{cn} \leq \frac{(\frac{n}{2})^{\frac{n}{2}}}{2}$, the equation can be rearranged as $(4^c)^{\frac{n}{2}} \leq \frac{1}{2} \cdot (\frac{n}{2})^{\frac{n}{2}}$, since c is a constant base that grows slower than a n base when n is large enough, then the equation must be valid and we finish the proof.
For a fraction of $\frac{1}{n}$ of $n!$ inputs, we need to prove $2^{cn} \leq \frac{1}{n} \cdot n! = (n-1)!$, so we need to prove $2^{cn} \leq \frac{(\frac{n-1}{2})^{\frac{n-1}{2}}}{2}$, same as $(4^c)^{\frac{n}{2}} \leq \frac{1}{2} \cdot (\frac{n-1}{2})^{\frac{n-1}{2}}$, since $n-1$ and n has same growth rate when n is large enough, we also conclude that this equation is valid.
For a fraction of $\frac{1}{2^n}$ of $n!$ inputs, we need to prove $2^{cn} \leq \frac{1}{2^n} \cdot n!$, the equation can be rearranged as $(4^{c+1})^{\frac{n}{2}} \leq (\frac{n}{2})^{\frac{n}{2}}$, since $c+1$ is also a constant and same conclusion can be applied.
So we can conclude that at least half or $\frac{1}{n}$ or $\frac{1}{2^n}$ of $n!$ inputs cannot have linear running time for comparison sort.

2. ***(15 points)*** *Exercise 8.1-4.*

Solution:
Since we are given a sequence of n elements instead of a random n size inputs, we know the possible sorted outputs are $(k!)^{\frac{n}{k}}$, by using a decision tree model, we know the tree height equals the maximum comparison number, so $2^h \geq (k!)^{\frac{n}{k}}$ should be valid for a binary tree, thus $h \geq \frac{n}{k} \cdot log(k!) \geq \frac{n}{k} \cdot log(\frac{k}{2})^{\frac{k}{2}} = \frac{n}{2} \cdot log(\frac{k}{2}) \simeq nlog(k)$ since constants can be ignored in this case. Thus $h = \Omega(nlgk)$ can be proved.

3. ***(5 points)*** *Exercise 8.2-1.*

Solution:
We can get array C as [2,2,2,2,1,0,2] and after computing prefix sum, the C is updated as [2,4,6,8,9,9,11]; repeating the procedure of 8.2, array $B = [, , , , , 2, , , , ,]$ and then $B = [, , , , , 2, , 3, , ,]$ and finally we have $B = [0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6]$.

4. ***(5 points)*** *Exercise 8.2-4.*

Solution:
We can follow the counting sort algorithm to get array C and then update each cell of C by calculating prefix sum, now C is an array with each cell value denoting how many numbers are less or equal than the index value. For example, the prefix sum C array in Exercise 8.2-1 is [2,4,6,8,9,9,11], 4 means there are 4 numbers that are less or equal than 1. So to answer how many numbers fall into range [a,b], we simply return C[b] - C[a-1]. This takes $O(1)$ time. And the first step to fill in the

original C array takes $\theta(n)$ time, the second step to update C array takes $\theta(k)$ time (k is length of C), total is $\theta(n+k)$ time.

```
let C[0..k] be a new array, A is the given array with n integers
for $i$ = 0 to $k$
    C[i]=0
for $j$ = 1 to A.length
    C[A[j]] = C[A[j]]+1
for $i$ = 1 to k
    C[i]=C[i]+C[i-1]
return C[b]-C[a-1]
```

5. ***(5 points)*** *Exercise 8.3-1.*

Solution:

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

6. ***(10 points)*** *Exercise 8.3-3.*

Solution:
The induction hypothesis is the last $i-1$ digits are already sorted. We now radix sort on ith digit, if two numbers have same ith digit, their relative order will not change because we assume the $i-1$ digits are already sorted in last radix sort and it is a stable sort algorithm; else if two numbers doesn't have same ith digit, they will be correctly sorted according to ith number. Finally, after doing radix sort on ith digit, all numbers are sorted accordingly on last i digits.
The loop will stop when we finish doing radix sort on all digits and the induction is trivially valid for last 0 digit. Now all numbers are sorted and we can prove radix sort works. We need the assumption that intermediate sort is stable when the ith digit of two numbers are same.

7. ***(5 points)*** *Exercise 8.3-4.*

Solution:
We first convert n integers into base n value and then we can do normal radix sort. The standard running time of radix sort is $\theta(d(n+k))$ with d being number of digits and k being possible values. In this case, $d = log_n(n^3) = 3$ and $k = n$ with values $(0,...,n-1)$, so the running time now is $\theta(6n) = O(n)$ which can satisfy requirement.

```
let A be the array with n integers
for $j$ = 1 to A.length
    A[j]=CONVERT-BASE(A[j])
for $i$ = 1 to 3
    COUNTING-SORT array A on digit i

CONVERT-BASE(i)
let s be 0, a=1
while(i!=0)
    s=s+a*(i mod 2)
    i=floor(i/2)
    a=a*10
return s
```

8. ***(20 points)*** *Exercise 9.1-1.*

Solution:
We divide the original array into a bunch of subarrays with size 2, and for each subarray we make a comparison and pick the smaller number, at the same time we also use a map C to track whom the smaller number gets compared with and this is exactly where lgn comes from. We continue recursing the whole process till there is only one element left and this element must be the smallest number. The array C exactly has ceil(lgn) length and we simply do $lgn-1$ comparisons to find the smallest number from C, then it must be the second smallest element of the original array. For example, if we have an original array A[5,6,1,2,7,3,8,10]$-$ >A[5,6 | 1,2 | 7,3 | 8,10]$-$ >A[5,1,3,8], because now we don't know 1 is the smallest number, we need to keep track of smaller number of each pair at the same time: C{[5: 6] [1: 2] [3: 7] [8: 10]}; repeating the process, A[5,1,3,8]$-$ >A[5,1 | 3,8]$-$ >A[1,3], now C is updated to C{[5: 6] [1: 2, 5] [3: 7, 8] [8: 10] }; repeating the process, A[1,3]$-$ >A[1], C will be updated to C{[5: 6] [1: 2, 5, 3] [3: 7, 8] [8: 10] }. Since A only has one element right now, the loop stops and we now find the smallest number of the original array after $n-1$ comparisons. Map C stores the numbers 1 has compared to and it must have a length with ceil(lgn) because everytime half of new array will never be compared to 1. We can retrieve the values from map C of 1 since they must contain the second smallest number of the original array, because the smallest element 1 must win over the second smallest element to become the last element so the second smallest element must ever be compared with element 1. Now we simply use textbook codes to find the smallest element of [2,5,3] and the comparison number is ceil(lgn)-1. The smallest number of [2,5,3] is 2 and it is exactly the second smallest number of the original array. Total comparison number will be $n-1$ + ceil(lgn)-1=n+ceil(lgn)-2 and we finish the proof.

```
FIND-SECOND-SMALLEST(A)
let C be a map
while(A.length>1)
    for $i$ = 1 to A.length-1
        if(A[i]<A[i+1])
            smaller =A[i]
            larger =A[i+1]
        else
            smaller =A[i+1]
            larger =A[i]
        A.add(smaller)
        C.update(smaller, larger)
        A.remove(A[i])
        A.remove(A[i+1])
        i=i+2
smallest = A[1]
return MINIMUM(C.get(smallest))
```

```
MINIMUM(A)
min=A[1]
for $i$ = 2 to A.length
    if(min>A[i])
        min=A[i]
return min
```

9. ***(10 points)*** *Exercise 9.3-7.*

Solution:
We first find median value, then we subtract every array cell value from median value to get absolute difference. Of all the absolute differences, we find the kth smallest value, last we iterate the array to find absolute differences that are less than the kth smallest value. The three steps cost $3 * O(n) = O(n)$ time.

```
K-CLOSEST-TO-MEDIAN(A,k)
let C be a new array with A.length
median = QUICK-SELECT(A[begin:end], $ceil(\frac{A.length}{2})$)
if(A.length is odd)
    medianValue=A[median]
else
    medianValue=(A[median]+A[median+1])/2
for $j$ = 1 to C.length
    C[j]=Math.abs(A[j]-medianValue)
kthSmall = QUICK-SELECT(C[begin:end], k)
for $i$ = 1 to C.length
    if(C[i] $\leq$ C[kthSmall])
```

```
        print A[i]

QUICK-SELECT(A[begin:end], rank r)
pivot=PARTITION(A)
if (pivot=r)
    return pivot
else if (pivot < r)
    QUICK-SELECT(A[begin:pivot-1], r)
else
    QUICK-SELECT(A[pivot+1:end], r-pivot)
```

10. ***(20 points)*** *Exercise 9.3-8.*

Solution:
We can show that the process will be same regardless of whether n is odd or even.
When n is even, medianX=(X[n/2]+X[n/2+1])/2, medianY=(Y[n/2]+Y[n/2+1])/2, if medianX=medianY, then X[n/2], X[n/2+1], Y[n/2] and Y[n/2+1] are the last four numbers we need; else if medianX<medianY, meaning X[1..n/2-1] should not be considered because medianX needs to be moved right and the left half of original medianX must not be candidate median statistics source, accordingly, Y[n/2+2,...n] should not be considered because medianY needs to be moved left and the right half of original medianY must not be candidate median statistics source of all 2n elements; else medianX>medianY, we simply do opposite operations of 'medianX<medianY'. After one process, we can get rid of half of the X array as well as Y array to achieve a binary partition. Then, we repeat the whole process until there are only four elements left.
When n id odd, medianX=X[(n+1)/2], medianY=Y[(n+1)/2], if medianX=medianY, then (medianX+medianY)/2 is exactly what we want; else if medianX<medianY, meaning X[1..(n-1)/2] and Y[(n+1)/2+1...n] should not be considered any more; else medianX>medianY, meaning Y[1...(n-1)/2] and X[(n+1)/2+1...n] should not be considered any more. After first process, we can find the subarray size of X and subarray of Y now become even number again, so we can apply the n=even scenario again to continue recursing.

```
FIND-MEDIAN-OF-TWO-EQUAL-SIZE-SORTED-ARRAYS(X,Y)
let B store the helper function returned result
B=FIND-MEDIAN(X[1:n],Y[1:n])
if(B.length=1)
    return first element of B
else if(B.length=4)
    return median number of the last four elements

FIND-MEDIAN(X[1:n],Y[1:n])
if(X.length is odd)
    medianX=X[(n+1/2)]
    medianY=Y[(n+1)/2]
    if(medianX==medianY)
        return subarray[(medianX+medianY)/2]
    else if(medianX<medianY)
        return FIND-MEDIAN(X[(n+1)/2:n],Y[1:(n+1)/2])
    else
        return FIND-MEDIAN(X[[1:(n+1)/2],Y[(n+1)/2:n])
else if(X.length is even)
    medianX=(X[n/2]+X[n/2+1])/2
    medianY=(Y[n/2]+Y[n/2+1])/2
    if(medianX==medianY)
        return subarray[X[n/2],X[n/2+1],Y[n/2],Y[n/2+1]]
    else if(medianX<medianY)
        return FIND-MEDIAN(X[n/2:n],Y[1:n/2+1])
    else
        return FIND-MEDIAN(X[1:n/2+1],Y[n/2:n])
```

11. ***(15 points)*** *Exercise 9.3-9.*

Solution:
The exercise requires to minimize $d = \sum_{i=1}^{n} \sqrt{(y_i - y)^2} = \sum_{i=1}^{n} |y_i - y|$, and this is exactly the property of median statistics. So the problem converts into how to calculate median statitics given y coordinates of all wells.

```
MINIMIZE-DISTANCE(A)
let A be the array with $y$ coordinates of all wells
median = QUICK-SELECT(A[begin:end], $ceil(\frac{A.length}{2})$)
if(A.length is odd)
    return A[median]
else
    return (A[median]+A[median+1])/2

QUICK-SELECT(A[begin:end], rank r)
pivot=PARTITION(A)
if (pivot==r)
    return pivot
else if (pivot < r)
    QUICK-SELECT(A[begin:pivot-1], r)
else
    QUICK-SELECT(A[pivot+1:end], r-pivot)
```

12. ***(10 points)*** *Exercise 8.4-3.*

Solution:
In two flips of a fair coin, the number of heads could be 0, 1, 2, and the possibility of each case is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, $2 \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$ and $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

$E[x^2] = 0^2 * \frac{1}{4} + 1^2 * \frac{1}{2} + 2^2 * \frac{1}{4} = 1.5$
$E^2[X] = (0 * \frac{1}{4} + 1 * \frac{1}{2} + 2 * \frac{1}{4})^2 = 1$

13. ***(Extra credit 10 points)*** *Problem 9-1.*

Solution:

14. ***(Extra credit 20 points)*** *Problem 8-1.*

Solution:

15. ***(Extra credit 20 points)*** *Problem 8.4.*

Solution: