cs 5800 - hw3

Yijing Xiao

October 2021

1 Problem 1

From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements, Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have $n! \leq l \leq 2^h$, which, by taking logarithms, implies

$$h \geq \lg(n!) \quad (since\ the\ lg\ function\ is\ monotonically\ increasing)$$

If half of the $n!$, then $h \geq \lg(\frac{n!}{2}) = \lg(n!) - \lg 2 = \lg(n!) - 1$

According to the Stirling's approximation, $\lg(n!) = \Theta(n \lg n)$ **(Book Equation 3.19)**

$$h \geq \lg(n!) - 1 \geq \Theta(n \lg n) = \Theta(n \lg n)$$

If fraction of $\frac{1}{n}$, then $h \geq \lg(\frac{n!}{n}) = \lg n! - \lg n = \Theta(n \lg n) - \lg n = \Theta(n \lg n)$
If fraction of $\frac{1}{2^n}$, then $h \geq \lg(\frac{n!}{2^n}) = \lg n! - \lg(2^n) = \lg n! - n = \Theta(n \lg n) - n = \Theta(n \lg n)$

Since the running time is $\Theta(n \lg n)$, there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n, for a fraction of $\frac{1}{n}$ of the inputs of length n, and for a fraction of $\frac{1}{2^n}$ inputs of length n.

2 Problem 2

Since the input sequence consists of $\frac{n}{k}$ sub-sequence, each containing k elements, there are $k!^{\frac{n}{k}}$ possible output permutations. Since a binary tree of height h has no more than2^h leaves, we have $k!^{\frac{n}{k}} \leq 2^h$, which, by taking logarithms, implies
$h \geq (\frac{n}{k}) \lg(k!)$
$\geq (\frac{n}{k}) \lg((\frac{k}{2})^{\frac{k}{2}}) \quad Properties\ of\ logarithms:\ for\ all\ k,\ k! \geq (\frac{k}{2})^{\frac{k}{2}}$
$\geq (\frac{n}{k})(\frac{k}{2}) \lg(\frac{k}{2})$
$= (\frac{n}{2})(\lg(k) - \lg 2)$
$= \frac{1}{2} n \lg k - \frac{1}{2} n$
$= \Omega(n \lg k)$

3 Problem 3

In the code for counting sort, we assume that the input is an array A[1..n], and thus $A.length = n$. We require two other arrays: the array B[1..n] holds the sorted output, and the array C;0..k] provides temporary working storage.

Initial:

A:	1	2	3	4	5	6	7	8	9	10	11
	6	0	2	0	1	3	4	6	1	3	2

B:	1	2	3	4	5	6	7	8	9	10	11

C:

0	1	2	3	4	5	6
2	2	2	2	1	0	2

$\Longrightarrow$

0	1	2	3	4	5	6
2	4	6	8	9	9	11

Start counting sort:

(a)

B:

1	2	3	4	5	6	7	8	9	10	11
					2					

C:

0	1	2	3	4	5	6
2	4	5	8	9	9	11

(b)

B:

1	2	3	4	5	6	7	8	9	10	11
					2		3			

C:

0	1	2	3	4	5	6
2	4	5	7	9	9	11

(c)

B:

1	2	3	4	5	6	7	8	9	10	11
			1		2		3			

C:

0	1	2	3	4	5	6
2	3	5	7	9	9	11

(d)

B:

1	2	3	4	5	6	7	8	9	10	11
			1		2		3			6

C:

0	1	2	3	4	5	6
2	3	5	7	9	9	10

(e)

B:

1	2	3	4	5	6	7	8	9	10	11
			1		2		3	4		6

C:

0	1	2	3	4	5	6
2	3	5	7	8	9	10

(f)

B:

1	2	3	4	5	6	7	8	9	10	11
			1		2	3	3	4		6

C:

0	1	2	3	4	5	6
2	3	5	6	8	9	10

(g)

B:

1	2	3	4	5	6	7	8	9	10	11
		1	1		2	3	3	4		6

C:

0	1	2	3	4	5	6
2	2	5	6	8	9	10

(h)

B:

1	2	3	4	5	6	7	8	9	10	11
	0	1	1		2	3	3	4		6

C:

0	1	2	3	4	5	6
1	2	5	6	8	9	10

(i)

B:

1	2	3	4	5	6	7	8	9	10	11
	0	1	1	2	2	3	3	4		6

C:

0	1	2	3	4	5	6
1	2	4	6	8	9	10

(j)

B:

1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	2	2	3	3	4		6

C:

0	1	2	3	4	5	6
0	2	4	6	8	9	10

(k)

B:

1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	2	2	3	3	4	6	6

C:

0	1	2	3	4	5	6
0	2	4	6	8	9	9

4 Problem 4

Pseudocode 1 Algorithm

```
1:  function ALGORITHM(A, B, k)
2:      let C[0..k] be a new array
3:      for i=0 to k do
4:          C[i] ← 0
5:      end for
6:      for j=1 to A.length do
7:          C[A[j]] ← C[A[j] + 1]          ▷ C[i] now contains the number of elements equal to i
8:      end for
9:      for i=1 to k do
10:         C[i] = C[i] + C[i − 1]         ▷ C[i] now contains the number of elements less than or equal to i
11:     end for
12:     n = C[b] − C[a − 1]
13:     return n
14: end function
```

This algorithm will begin by preprocessing exactly as Counting Sort does before line 10, so that $C[i]$ contains the number of elements less than or equal to i in the array. Thus, we compute $C[b] - C[a-1]$ to know how many of the n integers fall into a range [a..b] in O(1) time.

5 Problem 5

input	sort for last word	sort for middle word	sort for first word
COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	BIG	SEA	DIG
MOB	RUG	TEA	DOG
BOX	DOG	DIG	EAR
TAB	DIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RIG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

6 Problem 6

After sorting on digit i, we need to show that if we restrict to just the last i digits, the list is in sorted order. Since we just claim that digits we just sorted were in ordered order, now we suppose it's true for $i-1$, we need to show it for i. Suppose there are two elements, when restricted to the i last digits, are not in sorted order after the ith step. Then, we must make sure they have the same ith digit. Since they have the same first digit, their relative order is determined by their restrictions to their last $i-1$ digits. However, these were placed in the correct order by the $i-1th$ step. Since the sort on the ith digit was stable, their relative order is unchanged from the previous step. This means that they are in correct order still.
We use stability to show that being in the correct order prior to doing the sort is preserved.

7 Problem 7

According to the *lemma* 8.3, Given n-digit numbers in which each digit can take on up to k possible values, Radix-Sort correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time. First run through the list of integers and convert each one to base n, then Radix-Sort them. Since it takes on up to $n^3, k = n^3$ then each number will have at most $\log_n(n^3) = 3$ digits so there will be only 3 passes. For each pass, there are n possible values which can be take on, so we can use counting sort to sort each digit in $O(n)$ time.

8 Problem 8

If we build a Binary Search Tree(BST) from bottom to top and each node contains the smaller one of a pair, then we need $n-1$ comparisons to build BST. The top of BST is minimum, then the second smallest must be in the path of generating the top. Since the second smallest is only smaller than the smallest

9 Problem 9

Pseudocode 2 Randomized-Select

```
1:  function RANDOMIZED-SELECT(S, b, e, k)
2:      if b == e then
3:          return S[b]
4:      end if
5:      q = RANDOMIZED-PARTITION(S, b, e)
6:      i = q-b+1
7:      if k == i then                          ▷ the pivot value is the answer
8:          S[q]
9:      else
10:         if k < i then
11:             return RANDOMIZED-SELECT(A, b, q − 1, k)
12:         else
13:             return RANDOMIZED-SELECT(A, q + 1, e, k − i)
14:         end if
15:     end if
16: end function
```

First step: Find the median in a set S
Second step: For all integers in a set S, each integer minus the median and get the absolute value saving for a new set
Third step: Use Randomized-Sort(S, b, e, k) to find the kth smallest number x from a set S[b..e]

Final step: Iterate new set to choose all the absolute value that are less than y

10 Problem 10

Pseudocode 3 Median

```
1: function MEDIAN(X, Y, n)
2:     if n == 1 then
3:         return MIN(X[1], Y[1])
4:     end if
5:     if X[n/2] < Y[n/2] then
6:         return MEDIAN(X[n/2..n], Y[1..n/2], n/2)
7:     else
8:         if X[n/2] ≥ Y[n/2] then
9:             return MEDIAN(X[1..n/2], Y[n/2..n], n/2), n/2
10:        end if
11:    end if
12: end function
```

11 Problem 11

If n = 2: when only have y_1, y_2 two points, y_0 could be anywhere between y_1 and y_2. In this case, the distance of sub-pipes is $|y_1 y_2|$
If n = 3: In this case, y_0 should put on the position of y_2, the distance of sub-pipes is $|y_1 y_2| + |y_2 y_3|$

Pseudocode 4 Median

```
1: function MEDIAN(A, size)
2:     median = size  mod 2
3:     if size != 0 then                                   ▷ size is odd
4:         pick the y coordinate of the main pipeline to be equal to the median of all the y coordinates of the wells.
5:     else                                                ▷ size is even
6:         pick the y coordinate of the pipeline to be anything between the y coordinates of the wells with y-coordinates which have order statistics (size+1)/2 and the (size+1)/2
7:     end if
8: end function
```

Therefore, in general, when n is even the y_0 need to be the place between the two pipes which are median of all pipes. When n is odd, the y_0 need to be the median of all pipes. These can all be found in linear time using the Median algorithm from Book Chapter 9.

12 Problem 12

X is 0 or 2 which probability $\frac{1}{2} * \frac{1}{2} = \frac{1}{4}$ each, and 1 with probability $\frac{1}{2}$

$$E[X] = 2 * \frac{1}{4} + 1 * \frac{1}{2} + 0 * \frac{1}{4} = 1$$

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

$$E^2[X] = E[X] * E[X] = 1$$

13 Problem 13

13.1 a. Sort the numbers, and list the i largest

Sorting by using MergeSort takes time $\Theta(n \lg n)$, and listing them out takes time $\Theta(i)$, so the total runtime is $O(n \lg n + i)$

13.2 b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times

Build the heap which takes $\Theta(n)$ time, then call EXTRACT-MAX i times which takes $\Theta(i \lg n)$. So the total runtime is $\Theta(n + i \lg n)$.

13.3 c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers

Use the SELECT algorithm in Book Chapter 9.3 to find the largest number in $\Theta(n)$ time. Partition around the ith largest takes $\Theta(n)$ time. Sorting the i largest numbers take $\Theta(i \lg i)$time. So the total runtime is $O(n + i \lg i)$

14 Problem 14

14.1 a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $\frac{1}{n!}$ and that the rest are labeled 0.

There are $n!$ possible permutations of the input array because the input elements are all distinct, so each occurs with probability $\frac{1}{n!}$.