Aniruddha Tekade (B00618834) (Section-02) February 26, 2017

CS 575
Homework 2

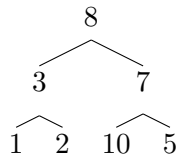
I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I under- stand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of 0 for the involved assignment for my first offense and that I will receive a grade of F for the course for any additional offense.

Sign : _____

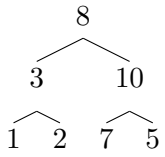
1: You are given an input array of integers: [8, 3, 7, 1, 2, 10, 5]

(a) The `max-heap` build process stepwise :

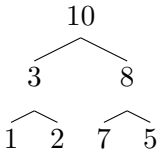
- Step I ⇒



- Step II ⇒



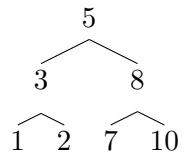
- Step III ⇒



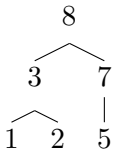
(b) The non-descending heapsort algorithm steps :

- Step I ⇒
  `Map-Heap` array represention : | 5 | 3 | 8 | 1 | 2 | 7 | 10 |

Heap after swapping 5 & 10 :

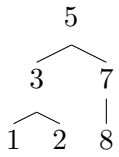


Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [5] [3] [8] [1] [2] [7]
Destination Array : [10]
Max-Heap after heapify:

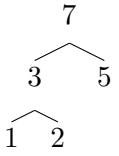


- Step II ⇒
  `Map-Heap` array represention : [8] [3] [1] [2] [7] [5]

  Heap after swapping 5 & 8 :

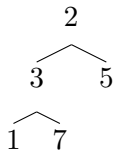


  Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [5] [3] [1] [2] [7]
  Destination Array : [10] [8]
  Max-Heap after heapify:

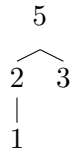


- Step III ⇒
  `Map-Heap` array represention : [7] [3] [5] [1] [2]

  Heap after swapping 2 & 7 :

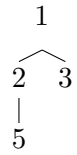


  Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [2] [3] [5] [1]
  Destination Array : [10] [8] [7]
  Max-Heap after heapify:



- Step IV ⇒
  `Map-Heap` array represention : [5] [2] [3] [1]

Heap after swapping 1 & 5 :



Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [1] [2] [3]
Destination Array : [10] [8] [7] [5]
Max-Heap after heapify:

- Step V ⇒
`Map-Heap` array represention : [3] [1] [2]

Heap after swapping 2 & 3 :

Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [2] [1]
Destination Array : [10] [8] [7] [5] [3]
Max-Heap after heapify:

- Step VI ⇒
`Map-Heap` array represention : [2] [1]

Heap after swapping 1 & 2 :

Remove last element from heap, insert it into the destination array & reduce size of heap by 1. `Map-Heap` array represention : [1]
Destination Array : [10] [8] [7] [5] [3] [2]
Max-Heap after heapify: 1

- Insert the element into the destination array. The final non-descending array will look like this ⇒ [10] [8] [7] [5] [3] [2] [1]

2: Assume that you are given an array of n elements sorted in non-descending order where n ≥ 1. Given the assumption, solve the following problem via divide and conquer.

(a) Ternary search algorithms psuedocode ⇒

```
1.  TernarySearch(low, high, S, x)
2.    if low > high then
```

```
3.        return NoSuchKey
4.    else
5.        mid1 <-- low + (floor(high - low) / 3)
6.        if (x == S[mid1])
7.            return mid1
8.        mid2 <-- high - (floor(high - low) / 3)
9.        if (x == S[mid2])
10.           return mid2
11.       if (x < S[mid1])
12.           return TernarySearch(low, mid1 - 1, S, x)
13.       else if (x > S[mid2])
14.           return TernarySearch(mid2 + 1, high, S, x)
15.       else
16.           return TernarySearch(mid1 + 1, mid2 - 1, S, x)
```

(b) Time complexity of Ternary Search algoriths is define by following recurrence definition ⇒

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 1 + T(n/3) & \text{if } n > 1 \end{cases}$$

Assume, input data size is n = 3^k
$\therefore$ k = $\log_3$ n
Solving the above equation

$\Rightarrow$ T(n) = 1 + T(n/3)
$\Rightarrow$ T(n/3) = 1 + 1 + T(n/9) = 2 + T(n/9)
$\Rightarrow$ T(n/9) = 1 + 1 + 1 + T(n/27) = 3 + T(n/27)
$\Rightarrow$ . . .
$\Rightarrow$ . . .
$\Rightarrow$ T(n/3) = k + T(n/3^k) = $\log_3 n$ + 1

Therefore, worst case time complexity of `TernarySearch` algorithms = $\Theta(\log_3 n)$

3: Use the radix sort algorithm to sort the following list of numbers in non-descending order: 321, 38, 15, 3, 9, 82, 10, 11.

(a) Explanation ⇒ The radix sort algorithsm working steps on the given input set are as follows:

- Perform radix sort at unit's place of each element of input.

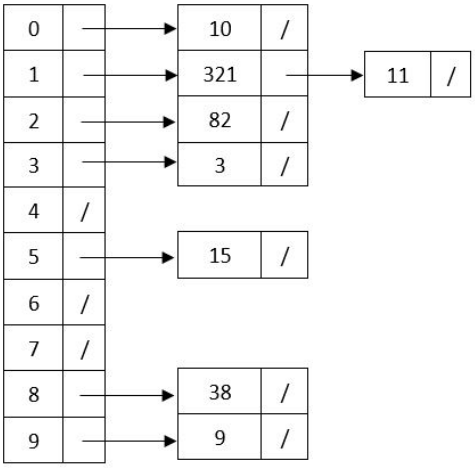


Figure 1: RadixSort Step(1)

Pass 1 ⇒ 10, 321, 11, 82, 3, 15, 38, 9

- Perform radix sort at ten's place of each element of pass 1

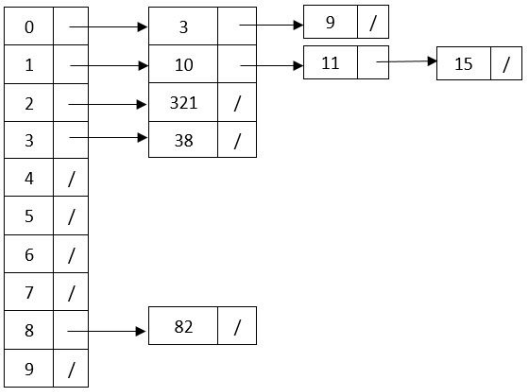


Figure 2: RadixSort Step(2)

Pass 2 ⇒ 3, 9, 10, 11, 15, 321, 38, 82

- Perform radix sort at hundred's place of each element of pass 1
  Pass 3 ⇒ 3, 9, 10, 11, 15, 38, 82, 321

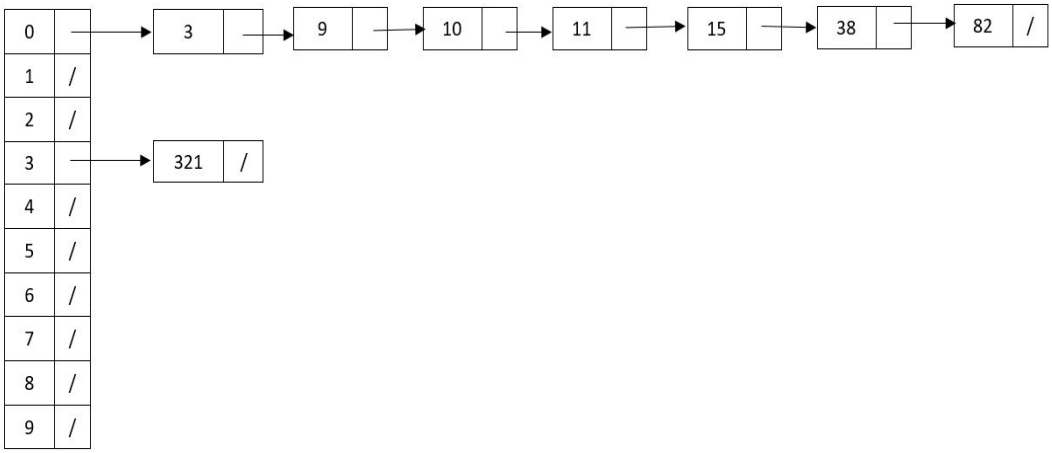


Figure 3: RadixSort Step(3)

(b) Radix Sort using most signifincant digit first ⇒

Unlike Radix sort with `least significant digit first`, Radix sort with `most significant digit first`, peforms the sorting in lexicographical order which is best suitable for set of characters, strings, words, very long integers & alpabetical input data set.

However, applying this technique on small integers may not work efficiently as after a centrain number of processing, it repeats the sorting output. This phenomenon can be realized using following diagrams ⇒

- Perform RadixSort MSD first : Step(I)

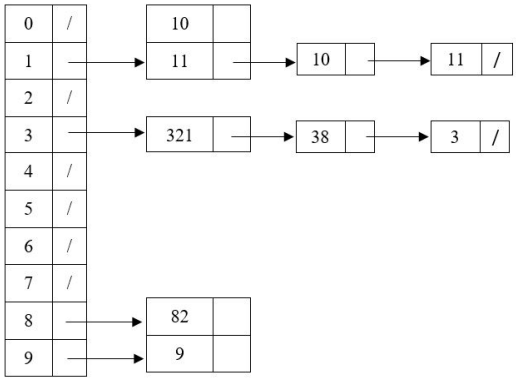


Figure 4: RadixSort MSD first : Step(1)

Output after pass 1 ⇒ 10, 15, 11, 321, 38, 3, 82, 9

- Perform RadixSort MSD at each element of pass 1 : Step(II)

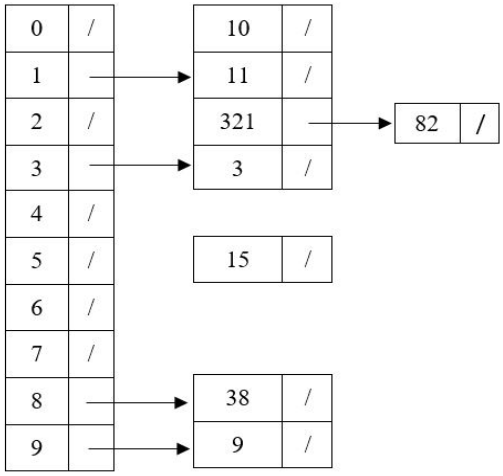


Figure 5: RadixSort MSD first : Step(2)

Pass 2 ⇒ 10, 11, 321, 82, 3, 15, 38, 9

- Perform radix sort at nest MSD of each element of pass 2

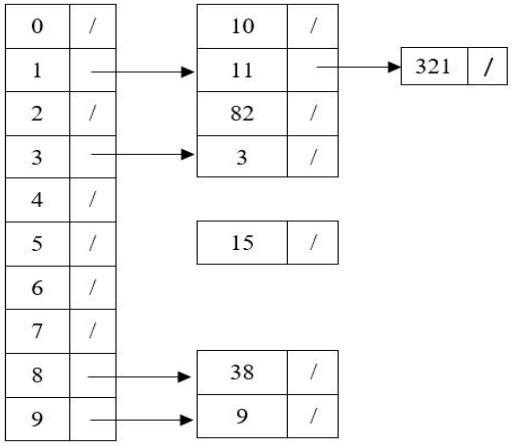


Figure 6: RadixSort with MSD first : Step(3)

Pass 3 ⇒ 10, 11, 321, 82, 3, 15, 38, 9

And now, we can see that after pass 3 for every new interation, the same output will be returned since, for every new pass, same digit will be considered for sorting.

Therefore, radix sort using most significant digit may not maintain the sorted order of input and may be inefficient for small size input. In short, it does not sort the small integers like given data set properly.

Radix sort with `most significant digit first` should only be applied to integers where every element of the integer dataset have equal length.

4: Prove that the lower bound of sorting based on comparions is $\Omega(nlgn)$

Explanation ⇒

- When a list of n integers is given as input to sorting algorithm like heapsort (which is a comparison based sorting), the input data can be arranged in n! ways.
- We build a decision tree that has n! leaf nodes where each leaf could be a sorted permutation & in each node of a decision tree, we compare two specific numbers at every interation or recursive call.
- Based on the result of the comparisons, we traverse to the left or right branch of the current node.
- Depth of the decision tree indicates total number of comparisons to reach a sorted permutation. So mathematically ⇒
  - Depth = lg n! ; where n! is approximately $(n/e)^n$
  - $\therefore$ lg n! = n lg(n/e) which is Ω(n lg n)
- Therefore, the time complexity of sorting algorithms based on comparison is Ω(n lg n).

In each node of a decision tree, compare two specific numbers Based on the result of the comparisons, take left or right branch Depth of the decision tree indicates # total comparisons to reach a sorted permutation

Depth: lg n! n! is approximately (n/e) n lg n! = nlg(n/e) Hence, sorting by comparisons is (nlgn)

5: The worst case time complexity of quick sort is $O(n^2)$. Regarding this, answer the following questions.

(a) In `quick-sort`, the time complexity is worst if the input is a `sorted array`. Therefore, whenever we provide a `sorted array` as input to `quick-sort` algorithms, it runs with its worst case time complexity of $O(n^2)$.

(b) The recurrence equation for quick sort is as follows -

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \text{ or } 1 \\ n + T(n-1) & \textbf{if } n > 1 \end{cases}$$

Solving the above equation

⇒ T(n) = n + T(n-1)
⇒ T(n-1) = n + (n-1) + T(n-2)
⇒ T(n-2) = n + (n-1) + (n-2) + T(n-3)
⇒ T(n-2) = n + (n-1) + (n-2) + (n-3) + T(n-4)

$\Rightarrow$ . . .
$\Rightarrow$ . . .
$\Rightarrow$ T(3) = 3 + T(2)
$\Rightarrow$ T(2) = 2 + T(1)
$\Rightarrow$ T(1) = 0

Therefore, T(n) = n + (n-1) + (n-2) + (n-3) + . . . + 3 + 2 = $\frac{n^2}{2}$ which is equal to n^2. Hence worst case time complexity of `quick sort` = $O(n^2)$

(c) We randomize the quick sort input. The process of randomization makes sure that the all the input data get shuffled well and then feed to the quicksort algorithm.

6: Is it a good idea to apply the divide and conquer method to compute a Fobonacci number?

(a) No

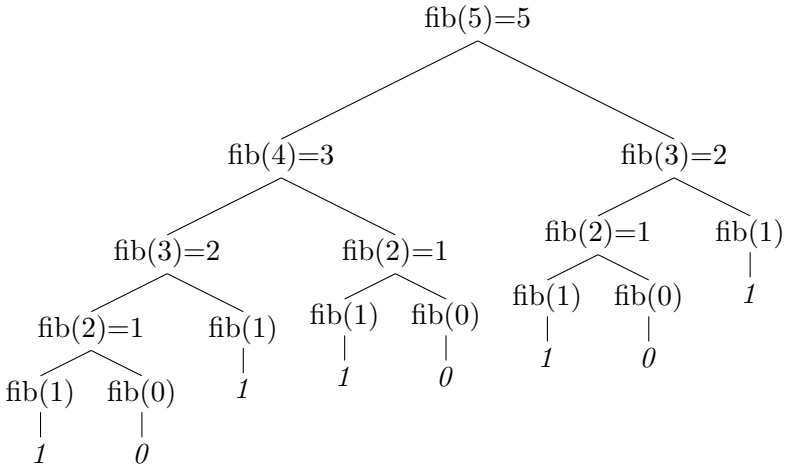
(b) Explanation $\Rightarrow$ solving the fibonacci number series can not be efficient using `divind and conquer` because of following reasons :

- The function definition of fibonacci number is -

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{if } n > 1 \end{cases}$$

  This clearly explains thats we need to required output from last terms to calculate each next term in the series.

- This can be realized using following tree. Lets consider n = 5.



- However, in divide and conquer technique, the input size always gets divided in $\frac{n}{2}$ size and then recursion is performed and output is calculated in a bottom up approach.

- Although, fibonacci has recursion in it & the input can be divided as per the requirement of Divide-and-Conquer technique, same operations needs to be repeated multiple times which is an inefficient way of applying Divide-and-Conquer.

- Therefore, we should not use Divide-and-Conquer to fibonacci number series. Instead, it is a dynamic programming problem where previous decisions need to be remembered and applied.

7: Prove the following

(a) $n^k = o(2^n)$; where k is a positive real constant.
Explanation ⇒ Let's apply limits to prove this complexity →

$$\lim_{n\to\infty} \frac{n^k}{2^n}$$

Now, we know that $2^n = e^{nln2}$
Taking derivating of above expression ⇒ $(2^n)' = (e^{nln2})' \Rightarrow \ln 2e^{nln2} = \ln 2(2^n)$
Continuing with the derivation & using L'Hospital's rule ⇒

$$\lim_{n\to\infty} \frac{kn^{k-1}}{2^n ln2}$$

⇒

$$\lim_{n\to\infty} \frac{k(k-1)n^{k-2}}{2^n ln^2 2}$$

⇒

$$\lim_{n\to\infty} \frac{\cdots}{\cdots}$$

⇒

$$\lim_{n\to\infty} \frac{k!}{2^n ln^k 2} = 0$$

Therefore, When the exponent k is very small, we need to look at very large values of n to see that $2^n > n^k$ and we can say that $n^k = o(2^n)$ for any k>0

(b) $lgn = o(n)$
Explanation ⇒ Let's apply limits to prove this complexity ⇒

$$\lim_{n\to\infty} \frac{lgn}{n}$$

Now, We know that lg n = $\frac{lnn}{ln2}$

Taking derivative ⇒ (lg n)' = $(\frac{lnn}{ln2})' \Rightarrow \frac{1}{nln2}$

Continuing with the derivation & using L'Hospital's rule
⇒

$$\lim_{n\to\infty} \frac{(lgn)'}{(n)'}$$

⇒

$$\lim_{n\to\infty} \frac{1}{nln2} = 0$$

Therefore, $lgn = o(n)$

(c) $n = o(n^3)$
Explanation $\Rightarrow$ Let's apply limits to prove this complexity
$\Rightarrow$

$$\lim_{n\to\infty} \frac{n}{n^3}$$

$\Rightarrow$

$$\lim_{n\to\infty} \frac{n'}{(n^3)'}$$

$\Rightarrow$

$$\lim_{n\to\infty} \frac{1}{3n^2}$$

Therefore, $n = o(n^3)$
