CTIS 264 Final

Syed Abdullah Hassan

21801072

1.1 Radix Sort [1]

a.

List: 82, 901, 100, 12, 150, 77, 55, 23.

The Radix sorts works staring from the least significant digit of the numbers and then sorts the list according to it. It moves up from there and sorts the previously sorted list by the next significant digits and so on (Counting sort is used inside Radix Sort) [1].

So:
Define 10 queues, each represents a digit from 0-9. Then insert the number in each of the queue based on the number's least significant digit.

Then group the numbers from queue 0 to 9 based on the order they have been inserted.

100, 150, 901, 82, 12, 23, 55, 77

Then insert the numbers again into the queue based on the next least significant digit (tens in this case).

100, 901, 12, 23, 150, 55, 77, 82

Repeat for the next significant digit (hundred)

012, 023, 055, 077, 082, 100, 150, 901 (zero has been put to make it clearer)

The list is sorted in ascending order.

b.

Pseudo-code [3]:

1. Find max number from the array
2. exp ← 1
3. while max /exp > 0
4. do
5. array ← CountingSort(array, exp)
6. exp = exp * 10

c.

Radix sort uses Transform and Conquer design technique. Each iteration of the sort transforms the given array of numbers according to a significant digit. Then it uses that transformed array as input for the next iteration of sorting.

d.

The Counting Sort is called inside the Radix sort. The basic operation for counting sort is copying one element from array1 to array2 [3].

e.

$\sum_{i=0}^{n} i + i + \log_2 i = \frac{n(n+1)}{2} + \frac{n(n+1)}{2} + \frac{n(n+1)}{2} = O(n)$

f.

Radix sort has Ω(n), θ(n), O(n) for the best, average, and worst case. Although they are the same, but the speed of the algorithm also depends upon the range of the numbers being used. If the numbers have more digits, it will have more iterations.

g.

$\Theta(\log_2 n)$

h. Done

i.

#	Number of Elements	Running Time(s)	Dataset
1	1000	3.82373	Ordered
2	10,000	45.67326	Ordered
3	1000	3.99821	Random
4	10,000	29.11182	Random
5	1000	3.81830	Reverse
6	10,000	28.84407	Reverse

j.

*analysis taken from the bonus assignment

1. Selection Sort

Being a simple sorting algorithm, it works extremely well for small files. It has the time efficiency: Ω(n^2), θ(n^2), and O(n^2). Each item in this algorithm is moved at most no more than once [4]. Because of O(n^2) time complexity, it is inefficient in sorting large lists of data. However, selection sort is still the algorithm that does the least amount of data moving. It might be a better choice in data sets where the data is considerably larger than the keys [8].

#	**Number of Elements**	**Running Time(s)**	**Dataset Order**
1	1000	0.0676	Ordered
2	10,000	16.8130	Ordered
3	1000	0.2668	Random
4	10,000	16.7893	Random
5	1000	0.2714	Reverse
6	10,000	17.0534	Reverse

2. Bubble Sort

Bubble sort is an algorithm that repeats; it is comparing each pair of adjacent items and swapping them if they are in the wrong order. It repeats until no swaps are required, indicating that the list is sorted [13][23]. It has a O (n2) Time complexity means that its efficiency decreases as list becomes bigger [12][24]. It works at Ω(n) in its best case. Bubble sort is a good choice for sorting small arrays.

#	Number of Elements	Running Time(s)	Dataset Order
1	1000	0.0000	Ordered
2	10,000	0.0066	Ordered
3	1000	0.4306	Random
4	10,000	32.1357	Random
5	1000	0.7128	Reverse
6	10,000	44.2858	Reverse

Radix sort has better best, average, and worst-case efficiency compared to quick sort, which has O(n^2) while radix sort has O(n). In the case of bubble sort, the best-case efficiency is the same. However, Radix sort has better efficiency in average and worst case compared to bubble sort.

1.2 Shell Sort [2]

a.

List: 12, 34, 54, 2, 3

Shell sort finds gap, which is n/2 in the case we will be discussing. One by one we select elements to the right of the gap and place them at their appropriate position (swap them).

In this case the difference will be 2. So, we will swap 12 with 54, 34 with 2 and 3 will remain in the position.

List: 12, 2, 54, 34, 3

Now the gap is reduced to 1 and repeat.

List: 2, 3, 12, 34, 54

Now gap is reduced to 0 and sorting stops, and the array is sorted.

b. [2]

```
procedure shellSort()
  A : array of items

  while interval < A.length /3 do:
    interval = interval * 3 + 1
  end while

  while interval > 0 do:

    for outer = interval; outer < A.length; outer ++ do:

    valueToInsert = A[outer]
    inner = outer;

      while inner > interval -1 && A[inner - interval] >= valueToInsert do:
        A[inner] = A[inner - interval]
        inner = inner - interval
      end while

    A[inner] = valueToInsert

    end for

  interval = (interval -1) /3;

  end while

end procedure
```

c. Shell sort uses decrease and conquer design technique just like insertion sort.

d. The swapping is of the values is the basic operation of Shell Sort.

e. $\sum_{i=0}^{n} \log_2 i + \sum_{i=0}^{n} \log_2 i + n = \frac{n(n+1)}{2} + \frac{n(n+1)}{2} + \frac{n(n+1)}{2} = O((\text{nlog}_2\, n)^2)$

f. Shell sort has $\Omega(\text{nlog}_2\, n)$, $\theta((\text{nlog}_2\, n)^2)$, $O((\text{nlog}_2\, n)^2)$ for the best, average, and worst case.

g. $\Theta((n\log_2 n)^2)$

2.0

a.

Critical Path Method is a way to recognizes the critical flow of operations within a program. Critical programs are the ones without which the program cannot complete by the deadline. It basically finds out the best way to do a set of tasks in which some might rely on each other.

Example:

The following are the steps to make an omelet:

1. Wisk eggs into a bowl
2. Add salt and pepper to the eggs
3. Heat oil in the frying pan
4. Fry onions
5. Add the egg to the frying pan
6. Fill it with cheese
7. Serve it

Steps 1, 3, 5, 7 are critical. However, adding salt and pepper alongside onions will make it taste better but it is not necessary to make it.

Therefore, the CPM to make an omelet is 1->3->5->7.

b. [4]

Dijkstra's Algorithm allows you to calculate the shortest path CPM for weighted Graphs. The algorithm finds the solution using the following steps:

1. Set initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the node C (current).
3. For each neighbor N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

c.

Dijkstra's Algorithm uses a graph data structure. It uses a parent class from which it selects which vertices to select according to their weights. The algorithm inputs the vertices and edges then outputs the shortest path possible between the given edges. Extra steps would have a higher weight so Dijkstra's Algorithm go through the critical steps to complete the graph without creating a cycle.

d.

Dijkstra's Algorithm uses greedy algorithm design technique. It solves single source shortest path problem. It computes the shortest path from one source node to all other remaining nodes of the graph.

I. References

[1] "Radix Sort," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/radix-sort/. [Accessed 1 Jun 2020].

[2] "Shell Sort," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/shellsort/. [Accessed 01 Jun 2020].

[3] "Radix Sort Algorithm," [Online]. Available: https://www.programiz.com/dsa/radix-sort. [Accessed 01 Jun 2020].

[4] "CodinGame," [Online]. Available: https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm. [Accessed 1 Jun 2020].