Brian Malyszka 4643400

Computer Science I – Exercise: Sorting

Before starting the exercise, go through the full slides, simulations, codes, and run time analysis. Then start doing the exercise.

1)Show the contents of the array below being sorted using Insertion Sort at the end of each loop iteration.

Initial	2	8	3	6	5	1	4	7
	2	8	3	6	5	1	4	7
	2	8	3	6	5	1	4	7
	2	3	8	6	5	1	4	7
	2	3	6	8	5	1	4	7
	2	3	5	6	8	1	4	7
	1	2	3	5	6	8	4	7
Sorted	1	2	3	4	5	6	8	7
	1	2	3	4	5	6	7	8

2) Show the contents of the array below being sorted using Selection Sort at the end of each loop iteration. As shown in class, please run the algorithm by placing the smallest item in place first.

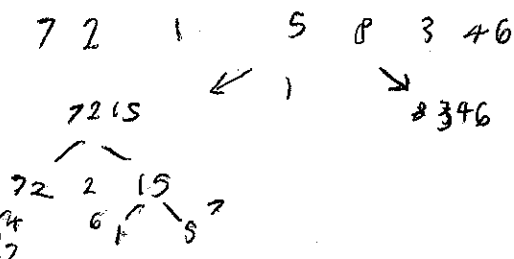
Initial	6	2	8	1	3	7	5	4
	1	2	8	6	3	7	5	4
	1	2	8	6	3	7	5	4
	1	2	3	6	8	7	5	4
	1	2	3	4	8	7	5	6
	1	2	3	4	5	7	8	6
	1	2	3	4	5	6	8	7
Sorted	1	2	3	4	5	6	7	8

3) Show the contents of the array below being sorted using Bubble Sort at the end of each loop iteration. As shown in class, please run the algorithm by placing the largest item in place first.

Initial	4	2	6	5	7	1	8	3
	2	4	5	6	1	7	3	8
	2	4	5	1	6	3	7	8
	2	4	1	5	3	6	7	8
	2	1	4	3	5	6	7	8
	1	2	3	4	5	6	7	8
	1	2	3	4	5	6	7	8
Sorted	1	2	3	4	5	6	7	8

4) When Merge Sort is run on an array of size 8, the merge function gets called 7 times. Consider running Merge Sort on the array below. What would the contents of the array be right before the 7th call to the Merge function?

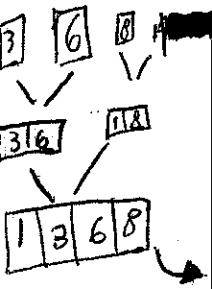
Initial	7	2	1	5	8	3	4	6
Before 7th Merge	2	7	1	5	8	3	4	6



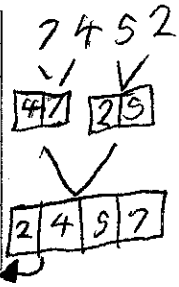
5) Show the result of running Partition (as shown in class on Friday) on the array below using the leftmost element as the pivot element. Show what the array looks like after each swap.

Initial	5	2	1	7	8	3	4	6
	5	2	1	4	8	3	7	6
	5	2	1	4	3	8	7	6
After Partition	3	2	1	4	5	8	4	6

6) Show the contents of the array below after each merge occurs in the process of Merge-Sorting the array below:



Initial	3	6	8	1	7	4	5	2
	3	6	8	1	7	4	5	2
	3	6	1	8	7	4	5	2
	1	3	6	8	7	4	5	2
	1	3	6	8	4	7	5	2
	1	3	6	8	4	7	2	5
	1	3	6	8	2	4	5	7
Last	1	2	3	4	5	6	7	8



7) Here is the code for the partition function (used by Quick Sort). Explain the purpose of each line of code.

```
int partition(int* vals, int low, int high) {

  int lowpos = low; // remember the index of pivot
  low++; // set low to just right of pivot

  while (low <= high) { // loop, tests if low has passed high

    while (low <= high && vals[low] <= vals[lowpos]) low++; // loop to increase low when vals[low] is less than pivot
    while (high >= low && vals[high] > vals[lowpos]) high--; // same as line above but for high

    if (low < high) // test if low has passed high or not
      swap(&vals[low], &vals[high]); // swap low and high values
  }

  swap(&vals[lowpos], &vals[high]); // swap pivot and high value
  return high; // return partition
}
```

8) Explain, why in worst case scenario the quick sort algorithm runs more slowly than Merge Sort

In the partition function of quicksort, it has 2 nested loops and therefore has $O(n^2)$. This means that in worst case, quick sort is slower than the $O(n \log_2 n)$ of merge sort.

9) In practice, quick sort runs slightly faster than Merge Sort. This is because the partition function can be run "in place" while the merge function can not. More clearly explain what it means to run the partition function "in place".

This function runs "in place" because it sorts the original array without using up any more memory.

10. You are trying to write a code for selection sort and you come-up with the following code. However, there is a bug in the code. Identify that bug and explain why that is a bug and edit that part of the code to correct it. Later, analyze the run-time of the updated code:

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx, temp;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = 0; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;


       temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;

    }
}
```

min 5
4 8 7 6
min=8
i=5

j can be set to (i+1) as at index 0 is the minimum value already and we do not need to check it.

1st: n steps
2nd: n-1 steps
3rd: n-2 steps
4th: n-3 steps
last: 1 step

$T(n) = n + (n-1) + (n-2) + n-3 + \dots + 3 + 2 + 1$

∴ which is equivalent to $\sum_{i=1}^{n} i$

which is equivalent to $\frac{n(n+1)}{2}$

$\therefore O(n^2)$

11) Explain the steps to come-up with the recurrence relation for merge sort and solve the recurrence relation to get the run-time of merge sort.

in merge there are 2 mergeSort calls and 1 merge call, each mergeSort splits the array in half $(n/2)$ and merge has $O(n)$

$\therefore T(n) = 2T(n/2) + n$

$T(n/2) = 2T(n/4) + n/2$

$T(n) = 2(2T(n/4) + n/2) + n$

$= 4T(n/4) + 2n$

$T(n/4) = 2T(n/8) + n/4$

$T(n) = 4(2T(n/8) + n/4) + 2n$

$= 8T(n/8) + 3n$ pattern!

$T(n) = 2^k T(n/2^k) + kn$

we know $T(1) = 1$ ∴

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$

$\therefore T(n) = 2^{\log_2 n} T(1) + (\log_2 n) n$

$= n \cdot 1 + n \log_2 n$

$\therefore O(n \log_2 n)$