tSorting & Selection Exercises

Ibrahim Albluwi, Sufyan Almajali, DANA AL-TAHER, Abdulrahman Abudabaseh

Exercise 1 (Tracing & Analysis)

A. For each of the following pairs of sub-arrays, count the number of *data moves* and *data compares* performed by the **`MERGE`** function in Merge Sort. Assume that each sub array is of size `n/2`.

Express your answer using **tilde** notation.

```
1. [n n n n n n n n … n]      [1 2 3 4 5 6 7 8 … n/2]
2. [1 2 3 4 5 6 7 8 … n/2]    [1 2 3 4 5 6 7 8 … n/2]
```

B. For each of the following arrays, count the number of *swaps* and *data compares* performed by the **`PARTITION`** function in Quicksort. Assume that the pivot is the first element and that the size of the array is `n`.

Express your answer using **tilde** notation.

```
1. [1 2 3 4 5 6 7 8 9 10 11 12 … n]
2. [1 2 2 2 2 2 … 2 0 0 0 0 0 … 0]    // number of 2's = number of 0's
3. [1 2 1 2 1 2 … 1 2]
```

C. Consider the array a[] which was partitioned only once during the execution of Quicksort. The array contents look as follows *after* the the **`PARTITION`** function finished executing:

`a = [1, 5, 7, 2, 9, 12, 11, 10]` Which elements could have possibly been the pivot?

Exercise 2 (Analysis)

A. Given an array of size **`n`** that is a power of 2:

1. How many times will the **`MERGE`** function be called in Merge Sort?
2. How many times will the **`PARTITION`** function be called in Quick Sort?

Provide your answer for the best case and the worst case and express your answer in tilde notation.

B. Consider a variant of Quicksort that uses Quick-Select to find the median in each partitioning step and uses it as the pivot. Analyze the number of *data compares* performed by Quicksort in the best case and in the worst case.

C. Consider an algorithm named Randomized-Sort that uses the Quick-Select algorithm to sort *n* elements, where Quick-Select is called to find the smallest element in the array, then is called again to find the 2nd smallest, then the 3rd, etc. Whenever the k^{th} smallest element is found, it is placed in its correct position in a new array. Randomized-Sort at the end copies back all of the elements that are in the sorted array into the original array.

Analyze the best case and worst case running time of this algorithm

Exercise 3 (Design)

A. Consider the following Quicksort function:

```
1   QUICKSORT(a[], first, last)
2         if (first >= last)
3               return
4
5         p = PARTITION(a, first, last)
6         QUICK-SORT (a, first, p - 1)
7         QUICK-SORT (a, p + 1, last)
```

Modify the Quicksort algorithm, such that it guarantees that the **`PARTITION`** function will always produce two non-empty partitions if the partitioned range has 8 elements or more.

Note the following:

- You are not allowed to change lines **`5`**, **`6`** or **`7`**. Your modifications must all happen before these lines.
- Assume that the **`PARTITION`** function uses the first element in the range as the pivot.
- Assume that the array has no duplicate elements.

B. Design a new divide and conquer algorithm named **`RangeQuickSort.`** Given an array **`a[]`** and a range of values **`[A ... B]`**, the algorithm sorts ***only*** the values of the array that fall within the given range. Assume that the values **`A`** and **`B`** exist in the array.

Example. `a[] = {14, 98, 16, 12, 15,5, 88, 2, 20, 33}`

Calling `RangeQuickSort(a, 0, 9, 14, 20)` should return array with the values in the range 14 to 20 sorted:

```
a[] = {5, 12, 2, 14, 15, 16, 20, 88, 33, 98}
```

Notes.

- The values to the left of the range and the values to the right of the range could be of any order and do not have to be sorted.
- Your algorithm must be more efficient than sorting the whole array.

Exercise 4 (Sorting & Recurrence Equations)

A. Assume that the **MERGE** function in Merge Sort was modified such that it returns (without doing anything) if `a[mid] <= a[mid+1]`.

Fill out the following table regarding the number of data *compares* done by Merge Sort (the whole algorithm, not only the **MERGE** function).

	Best Case	**Worst Case**
Provide an example Array of size **8**		
Describe the running time using a **recurrence equation**		
Running Time in Tilde (~) notation for an array of size ***n***		

B. Consider the following code for a modified version of the Quick Sort algorithm:

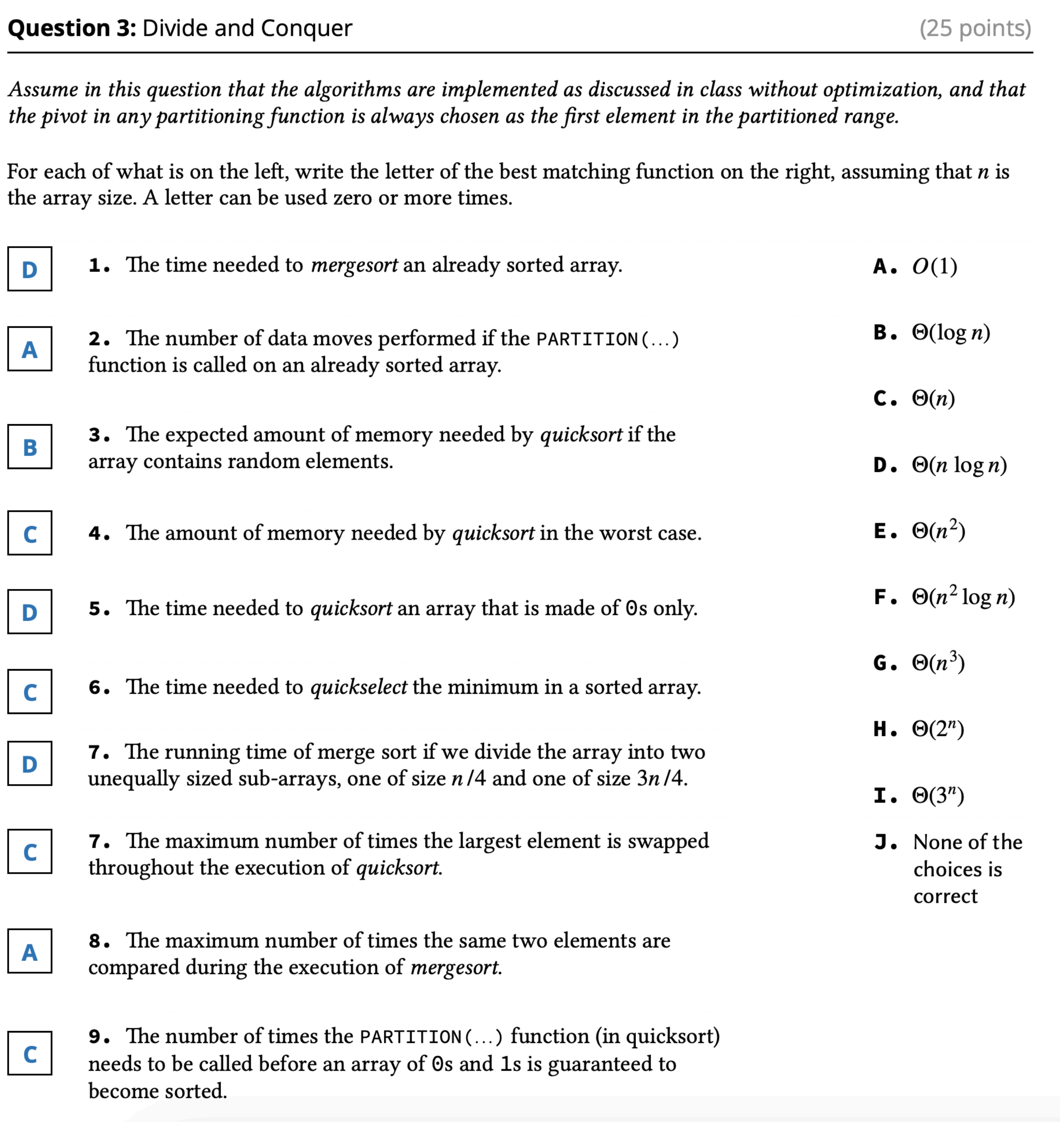
```
QUICKSORT(a[], first, last)
      if (first >= last)
            return

      p = PARTITION(a, first, last)
      SORT-A (a, first, p - 1)
      SORT-B (a, p + 1, last)
```

Fill out the following table below assuming **`SORT-A(…)`** and **`SORT-B(…)`** are replaced with the corresponding options.

- *Assume that **`PARTITION`** picks the first element in the sorted range as the pivot.*
- *Remember that the running time of insertion sort is Θ(n^2) in the worst case and Θ(n) in the best case, while selection sort runs in Θ(n^2).*

Type of Sorted Data	**SORT-A**	**SORT-B**	**Recurrence Equation**
Sorted (ascending)	Insertion Sort	Quick Sort	
Random (pivot is the median)	Merge Sort	Quick Sort	
Random (pivot is the median)	Selection Sort	Quick Sort	
Sorted (ascending)	Quick Sort	Insertion Sort	

A screenshot from the midterm of Fall 2023



Question 3: Divide and Conquer (25 points)

Assume in this question that the algorithms are implemented as discussed in class without optimization, and that the pivot in any partitioning function is always chosen as the first element in the partitioned range.

For each of what is on the left, write the letter of the best matching function on the right, assuming that n is the array size. A letter can be used zero or more times.

D	**1.** The time needed to *mergesort* an already sorted array.
A	**2.** The number of data moves performed if the `PARTITION(...)` function is called on an already sorted array.
B	**3.** The expected amount of memory needed by *quicksort* if the array contains random elements.
C	**4.** The amount of memory needed by *quicksort* in the worst case.
D	**5.** The time needed to *quicksort* an array that is made of `0`s only.
C	**6.** The time needed to *quickselect* the minimum in a sorted array.
D	**7.** The running time of merge sort if we divide the array into two unequally sized sub-arrays, one of size $n/4$ and one of size $3n/4$.
C	**7.** The maximum number of times the largest element is swapped throughout the execution of *quicksort*.
A	**8.** The maximum number of times the same two elements are compared during the execution of *mergesort*.
C	**9.** The number of times the `PARTITION(...)` function (in quicksort) needs to be called before an array of `0`s and `1`s is guaranteed to become sorted.

A. $O(1)$

B. $\Theta(\log n)$

C. $\Theta(n)$

D. $\Theta(n \log n)$

E. $\Theta(n^2)$

F. $\Theta(n^2 \log n)$

G. $\Theta(n^3)$

H. $\Theta(2^n)$

I. $\Theta(3^n)$

J. None of the choices is correct

Solutions

1.A

Arrays	Data Compares	Data Moves
1. `[n n n n n n n n … n]` `[1 2 3 4 5 6 7 8 … n/2]`	**½ n**	**2n**
1. `[1 2 3 4 5 6 7 8 … n/2]` `[1 2 3 4 5 6 7 8 … n/2]`	**n-1**	**2n**

1.B

Arrays	Data Compares	Swaps
2. `[1 2 3 4 5 6 7 8 9 10 11 12 … n]`	**~n**	**1**
2. `[1 2 2 2 2 2 … 2 0 0 0 0 0 … 0]` `// number of 2's = number of 0's`	**~n**	**~n/2**
1. `[1 2 1 2 1 2 … 1 2]`	**~n**	**~n/4**

1.C

1 and 9

2.A

	Best Case	Worst Case
1	`~n, exactly n-1`	`~n, exactly n-1`
2	`~n, exactly n-1`	`~n, exactly n-1`

2.B Best case = $n \log n$ and worst case = n^2

2.C Best case = n^2 and worst case = n^3

3.A The idea is to force the partition function to pick an element (as the pivot) that is not the maximum and not the minimum (since these are the cases that lead to an empty partition). The following is one way that avoids picking the max or min as the pivot.

```
QUICKSORT(a[], first, last)
      if (first >= last):
            return

      if (last - first > 8):
            x = a[first], y = a[first + 1], z = a[first + 2]
            if (NOT(y < x AND y < z) AND NOT(y > x AND y > z))
                  // y is not min and not max
                  SWAP(a[first], a[first + 1])
            else if (NOT(z < x AND z < y) AND NOT(z > x AND z > y))
                  // z is not min and not max
                  SWAP(a[first], a[first + 2])
            // otherwise a[first] is not the min and not the max

      p = PARTITION(a, first, last)
      QUICK-SORT (a, first, p - 1)
      QUICK-SORT (a, p + 1, last)
```

3.B

```
RangeQuickSort(data[], first, last, A, B):
    if (last <= first)
        return;

    p = Partition(data, first, last)
    if (data[p] <= A):
        RangeQuickSort(data, p + 1, last, A, B)
    else if (data[p] >= B):
        RangeQuickSort(data, first, p - 1, A, B)
    else:
        RangeQuickSort(data, first, p - 1, A, B)
        RangeQuickSort(data, p + 1, last,  A, B)
```

4.A

	Best Case	**Worst Case**
Provide an example Array of size **8**	[1 2 3 4 5 6 7 8]	[1 2 1 4 1 3 1 5]
Describe the running time using a **recurrence equation**	T(n) = 2T(n/2) + 1	T(n) = 2T(n/2) + n
Running Time in Tilde (~) notation for an array of size ***n***	~n	$\sim n\log_2(n)$

4.B

Type of Sorted Data	SORT-A	SORT-B	Recurrence Equation
Sorted (ascending)	Insertion Sort	Quick Sort	`T(n) = T(n-1) + O(n)`
Random (pivot is the median)	Merge Sort	Quick Sort	`T(n) = T(n/2) + O(nlogn)`
Random (pivot is the median)	Selection Sort	Quick Sort	$T(n) = T(n/2) + O(n^2)$
Sorted (ascending)	Quick Sort	Insertion Sort	`T(n) = O(n)`