

Recitation 3 (HW2)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 475: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Partition in QuickSort

Recall the partition function we learn in QuickSort algorithm:

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo    // place for swapping
    for j := lo to hi - 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
```

This method is called '**Lomuto Partition**' which is linear-time and in-place.

Partition in QuickSort

We introduce another partition method which is also linear-time and in-place.

The idea is:

Set two indices at both ends of the array, then move toward each other until they detect an inversion:

Partition in QuickSort

We introduce another partition method which is also linear-time and in-place.

The idea is:

Set two indices at both ends of the array, then move toward each other until they detect an inversion:

Element on the left index is larger than the pivot, while element on the right index is smaller than the pivot.

Partition in QuickSort

We introduce another partition method which is also linear-time and in-place.

The idea is:

Set two indices at both ends of the array, then move toward each other until they detect an inversion:

Element on the left index is larger than the pivot, while element on the right index is smaller than the pivot.

Then swap the two inverted elements, and repeat the above process.

Partition in QuickSort

We introduce another partition method (Hoare partitioning scheme) which is also linear-time and in-place.

The idea is:

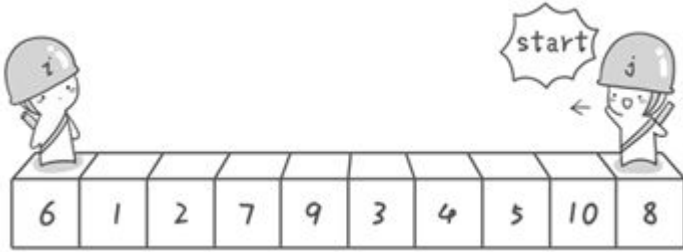
Set two indices at both ends of the array, then move toward each other until they detect an inversion:

Element on the left index is larger than the pivot, while element on the right index is smaller than the pivot.

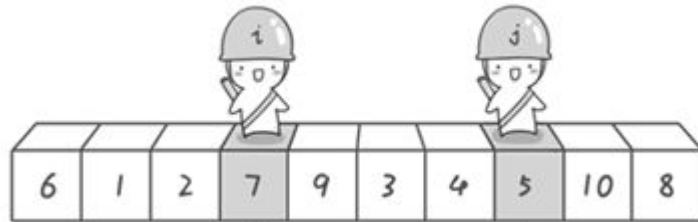
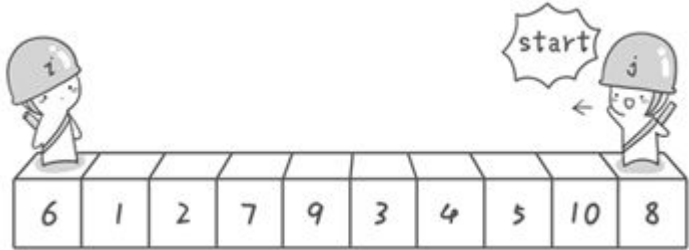
Then swap the two inverted elements, and repeat the above process.

When two indices meet, swap the pivot with the index and returns the final index.

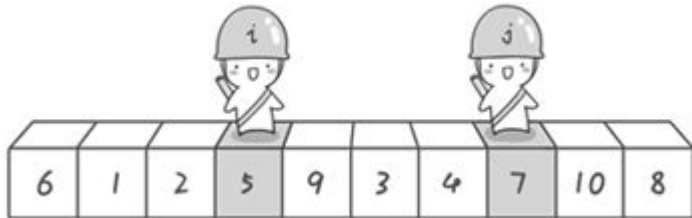
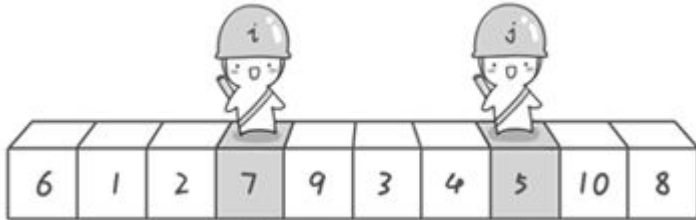
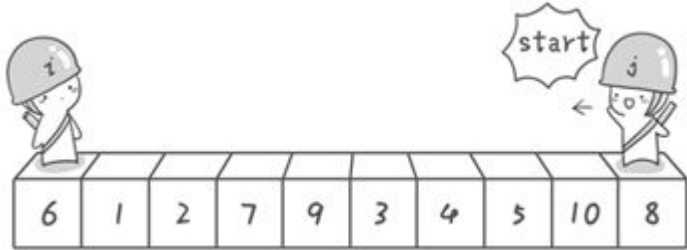
Partition in QuickSort



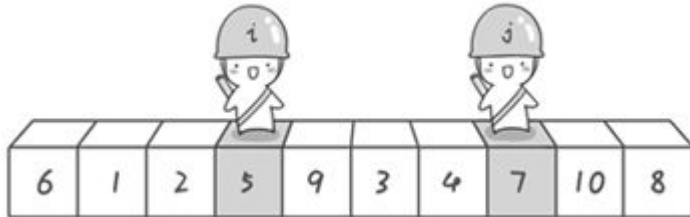
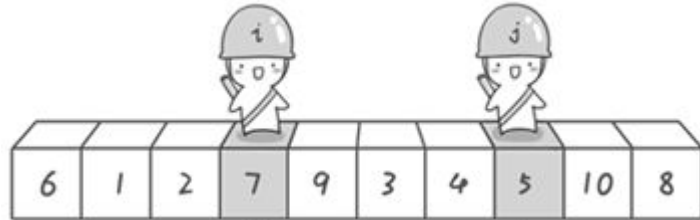
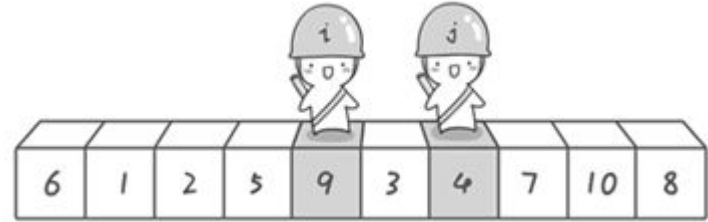
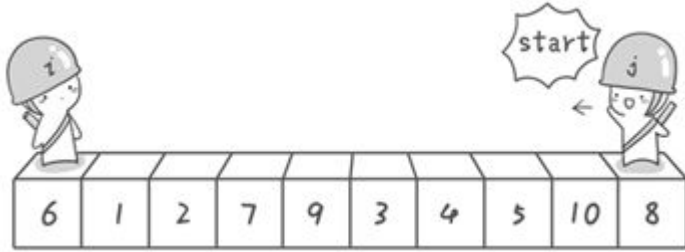
Partition in QuickSort



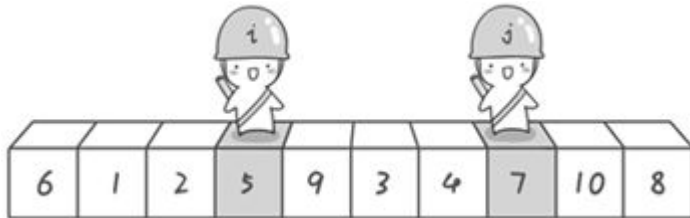
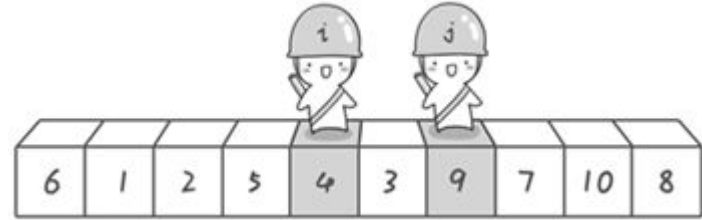
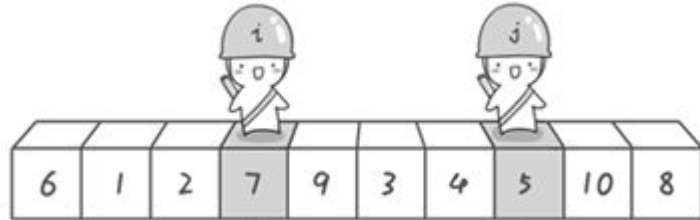
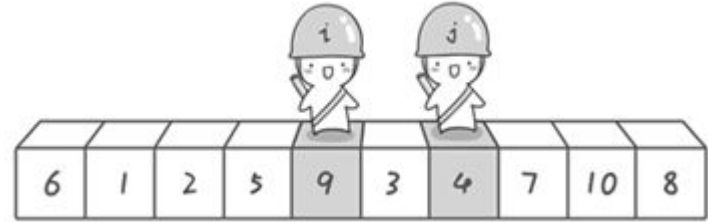
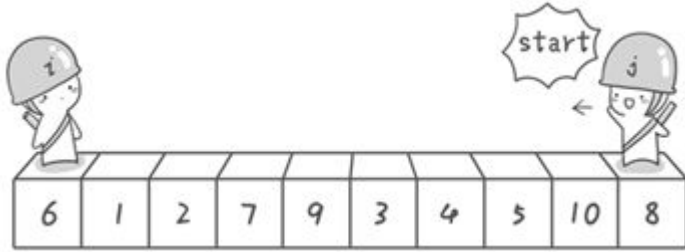
Partition in QuickSort



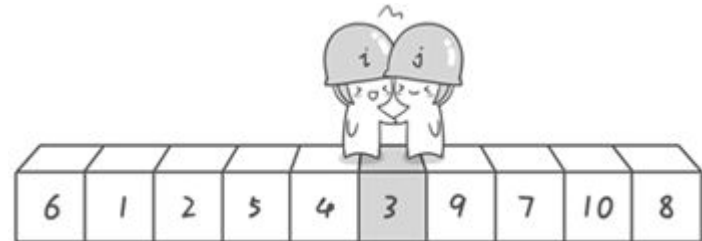
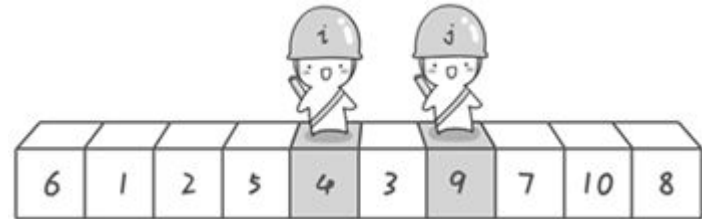
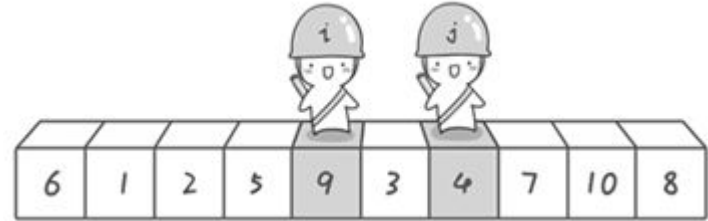
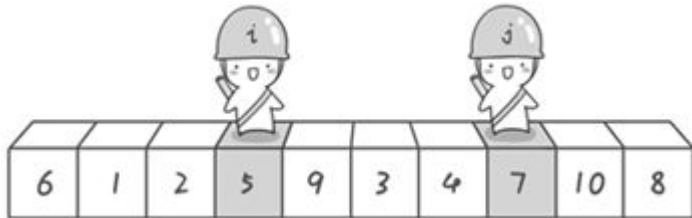
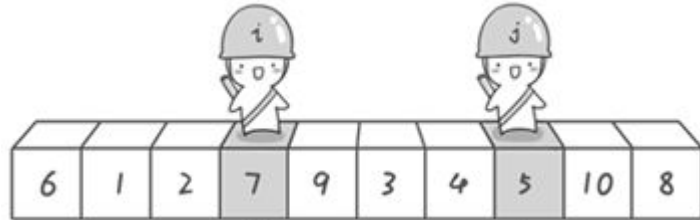
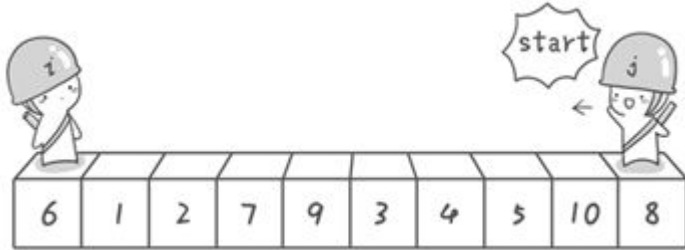
Partition in QuickSort



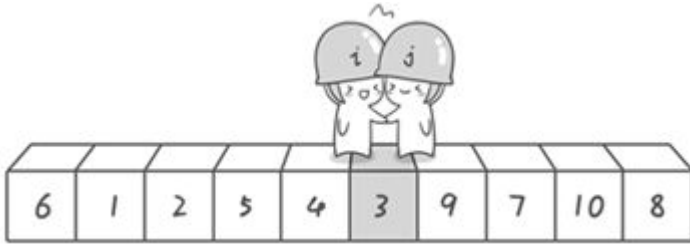
Partition in QuickSort



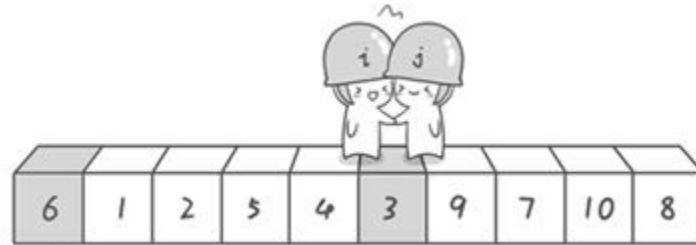
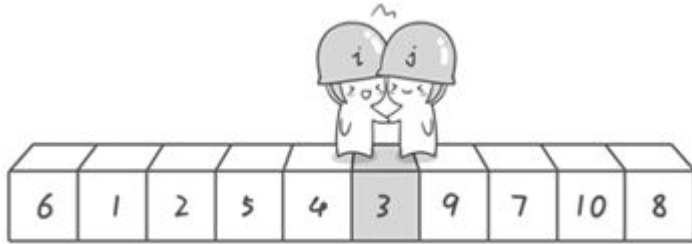
Partition in QuickSort



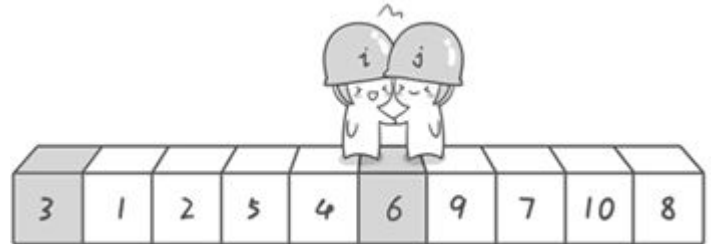
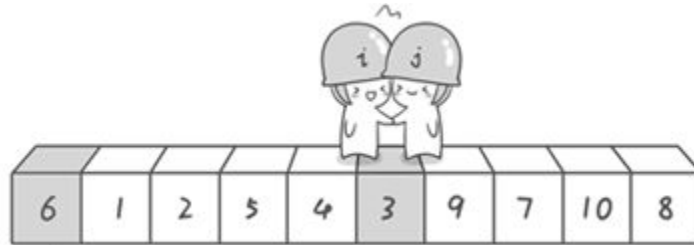
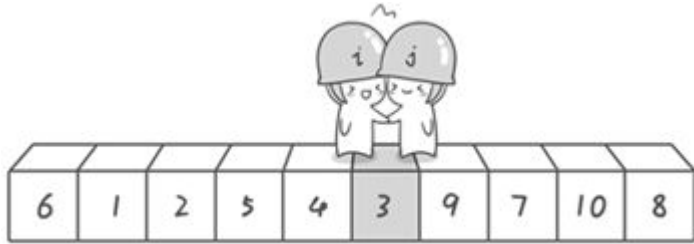
Partition in QuickSort



Partition in QuickSort



Partition in QuickSort



Partition in QuickSort

```
15 quickSort(array, 0, array.length - 1);
16
17 void quickSort(int[] array, int start, int end) {
18     // base case
19     if (start > end) return;
20
21     // set the start as the pivot
22     int pivot = array[start];
23
24     int i = start, j = end;
25     while (i != j) {
26         while (i < j && array[j] >= pivot) {
27             j--;
28         }
29         while (i < j && array[i] <= pivot) {
30             i++;
31         }
32         if (i < j) {
33             swap(array[i], array[j]);
34         }
35     }
36
37     array[start] = array[i];
38     array[i] = pivot;
39
40     quickSort(array, start, i - 1);
41     quickSort(array, i + 1, end);
42 }
```


Partition in QuickSort (Reference)

Hoare partition algorithm

Generally, the first item or the element is assumed to be the initial pivot element. Some choose the middle element and even the last element.

It is a linear algorithm.

It is relatively faster.

It is slightly difficult to understand and to implement.

It doesn't fix the pivot element in the correct position.

Lomuto partition algorithm

Generally, a random element of the array is located and picked and then exchanged with the first or the last element to give initial pivot values. In the aforementioned algorithm, the last element of the list is considered as the initial pivot element.

It is also a linear algorithm.

It is slower.

It is easy to understand and easy to implement.

It fixes the pivot element in the correct position.

Problem 1

Problem 1 (5+5+13 points)

Let $A[1, \dots, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called a **reverse pair** for A .

- (a) List the five reverse pairs of the array $A = [2, 3, 8, 6, 1]$.
- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.
- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithm satisfies the required time complexity bound.

Problem 1

- (a) List the five reverse pairs of the array $A = [2, 3, 8, 6, 1]$.

Problem 1

(a) List the five reverse pairs of the array $A = [2, 3, 8, 6, 1]$.

Assume A is 1-indexed.

(1,5) (2,5) (3,4) (3,5) (4,5)

According to the definition, **reverse pairs** should consists of index instead values.

Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

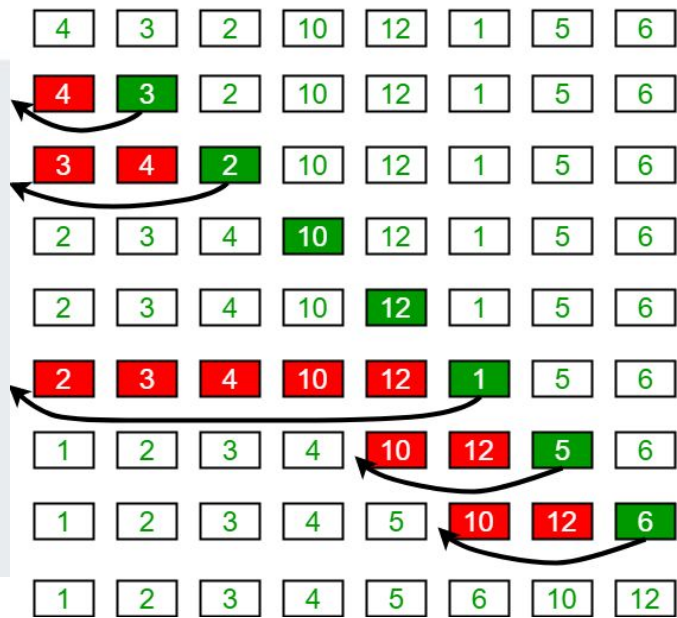
Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

Recall the pseudo-code of INSERTION-SORT:

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Insertion Sort Execution Example

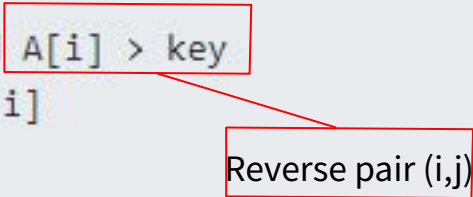


Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

Recall the pseudo-code of INSERTION-SORT:

```
for j = 2 to A.length
  key = A[j]
  // Insert A[j] into the sorted sequence A[1..j - 1].
  i = j - 1
  while i > 0 and A[i] > key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = key
```



Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

```
for j = 2 to A.length
  key = A[j]
  // Insert A[j] into the sorted sequence A[1..j - 1].
  i = j - 1
  while i > 0 and A[i] > key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = key
```

When the inner while-loop iterates once, two elements of one reverse pair swap, leading to the number of reverse pairs decreasing by one.

Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Also, after the INSERTION-SORT, the array is increasingly sorted, so the number of reverse pairs is 0.

Problem 1

- (b) What is the relationship between the running time of INSERTION_SORT and the number of reverse pairs in the input array? Justify your answer.

Define $T(n)$ as the running time of INSERTION-SORT

K as the number of reverse pairs

Thus, the outer j -loop runs for n times, and the inner while-loop runs for K times (making the number of reverse pairs from K to 0).

$$T(n) = \theta(n + K)$$

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

Recall the pseudo-code of MERGE-SORT:

```
MERGE-SORT( $A, p, r$ )
```

```
1  if  $p < r$ 
```

```
2    then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
```

```
3    MERGE-SORT( $A, p, q$ )
```

```
4    MERGE-SORT( $A, q+1, r$ )
```

```
5    MERGE( $A, p, q, r$ )
```

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.

Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

```
MERGE_SORT( $A, p, r$ )
```

```
1  if  $p < r$ 
```

```
2    then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
```

```
3    MERGE_SORT( $A, p, q$ )
```

```
4    MERGE_SORT( $A, q+1, r$ )
```

```
5    MERGE( $A, p, q, r$ )
```

In order to keep the running time $\theta(n \log n)$, we maintain the structure of MERGE-SORT while only modifying MERGE function (rename as **MERGE'**).

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

What is the time complexity of **MERGE**'?

Problem 1

- (c) Modify `MERGE_SORT` in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

What is the time complexity of **MERGE**'?

It should still be $\theta(n)$ as before, in order to keep the overall running time $\theta(n \log n)$.

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

What is the function of **MERGE**'?

Problem 1

- (c) Modify `MERGE_SORT` in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

What is the function of **MERGE**'?

Recall the `MERGE-SORT`, the two recursive functions separately sorted the left half of array and the right half of array.

Meanwhile, the function of `MERGE` is to merge the sorted left half of array with the sorted right half of array into a sorted array.

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.

Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

What is the function of **MERGE'**?

Similarly, here the two recursive functions separately sort and count the number of reverse pairs in the left half of array and the right half of array.

Meanwhile, the function of **MERGE'** is to merge left with right and count the number of reverse pairs (i,j) where index **i** is in the left and index **j** is in the right.

Total reverse pairs = pairs from *left* + pairs from *right* + **pairs from left and right**

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.

Fully explain your algorithm and also justify why your algorithm satisfies the required time complexity bound.

What is the func

Similarly, here th
reverse pairs in t

Meanwhile, the f
reverse pairs (i,j

Total number = l

MERGE-SORT(A, p, r)

1 if $p < r$

2 then $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 $X = \text{MERGE_SORT}(A, p, q)$

4 $Y = \text{MERGE_SORT}(A, q+1, r)$

5 $Z = \text{MERGE}(A, p, q, r)$

return $X+Y+Z$

id count the number of
y.

t and count the number of
the right.

en left and right

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

Where should we modify the function MERGE'?

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

Where should we modify the function MERGE'?

Recall the condition is $L[i] > R[j]$ where L is left half and R is right half, then (i,j) is a reverse pair.

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

How to count the number of reverse pairs?

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.
Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

How to count the number of reverse pairs?

Hint: The left half L and right half R are already sorted.

Problem 1

- (c) Modify MERGE_SORT in order to give an algorithm which determines the number of reverse pairs in an array of n numbers in $\Theta(n \log n)$ run time.

Fully explain your algorithm and also justify why your algorithms satisfies the required time complexity bound.

How to count the number of reverse pairs?

L R

[start...mid, mid+1...end]

If $L[i] \leq R[j]$, then

$L[i] > R[mid+1...j-1] \Rightarrow$ how many reverse pairs?

Problem 1

- (c) Modify MERGE_SORT in order to give an array of n numbers in $\Theta(n \log n)$ runtime. Fully explain your algorithm and also justify the time complexity bound.

How to count the number of reverse pairs in an array

L R

[start...mid, mid+1...end]

If $L[i] \leq R[j]$, then

$L[i] > R[mid+1...j-1] \Rightarrow$ how many reverse pairs? $\Rightarrow j - (mid+1)$

```
1 int count = mergeSort(array, 0, array.length - 1, new int[array.length]);
2
3 int mergeSort(int[] array, int start, int end, int[] tmp) {
4     // base case
5     if (start >= end) return 0;
6
7     int mid = start + (end - start) / 2;
8
9     int left = mergeSort(array, start, mid, tmp);
10    int right = mergeSort(array, mid + 1, end, tmp);
11    int cnt = merge(array, start, mid, end, tmp);
12
13    return left + right + cnt;
14 }
15
16 int merge(int[] array, int start, int mid, int end, int[] tmp) {
17     for (int i = start; i <= end; ++i) {
18         tmp[i] = array[i];
19     }
20
21     int i = start, j = mid + 1, k = start, cnt = 0;
22     while (i <= mid) {
23         if (j > end || (tmp[i] <= tmp[j])) {
24             cnt += j - (mid + 1);
25             array[k++] = tmp[i++];
26         } else {
27             array[k++] = tmp[j++];
28         }
29     }
30
31     return cnt;
32 }
```

reverse pairs in
complexity

Problem 2

Problem 2 (5+8+8 points)

Recall that MERGE_SORT splits the input array into two halves of almost equal sizes, and after each half is sorted recursively, both of them are merged into a sorted array.

Let's now consider a variant of MERGE_SORT, where instead of splitting the array into two equal size parts, we split it into four parts of almost equal sizes, recursively sort out each part, and at the end, merge all four parts together into a sorted final array.

In what follows, we want to find the time complexity of this variant of MERGE_SORT.

- (a) Define $T(n)$ as the worst-case running time of this variant on any input of size n . Find a recursive formula for $T(n)$, i.e. write $T(n)$ in terms of $T(k)$ for some $1 \leq k < n$.
- (b) Draw the corresponding recursion tree and find the explicit answer for $T(n)$.
- (c) Prove your result in part (b) formally using strong induction.
For full credit, your answer must use $\Theta(\cdot)$ notation, i.e., you should obtain both an upper bound and a lower bound.

Problem 2

- (a) Define $T(n)$ as the worst-case running time of this variant on any input of size n .
Find a recursive formula for $T(n)$, i.e. write $T(n)$ in terms of $T(k)$ for some $1 \leq k < n$.

Recall MergeSort($A[1 \dots n]$):

1. MergeSort($A[1 \dots n/2]$)
2. MergeSort($A[n/2+1 \dots n]$)
3. Merge($A[1 \dots n/2]$, $A[n/2+1 \dots n]$)

Problem 2

- (a) Define $T(n)$ as the worst-case running time of this variant on any input of size n . Find a recursive formula for $T(n)$, i.e. write $T(n)$ in terms of $T(k)$ for some $1 \leq k < n$.

Variant MergeSort($A[1 \dots n]$):

1. MergeSort($A[1 \dots n/4]$)
2. MergeSort($A[n/4+1 \dots 2n/4]$)
3. MergeSort($A[2n/4+1 \dots 3n/4]$)
4. MergeSort($A[3n/4+1 \dots n]$)
5. Merge($A[1 \dots n/4], A[n/4+1 \dots 2n/4], A[2n/4+1 \dots 3n/4], A[3n/4+1 \dots n]$)

Problem 2

- (a) Define $T(n)$ as the worst-case running time of this variant on any input of size n . Find a recursive formula for $T(n)$, i.e. write $T(n)$ in terms of $T(k)$ for some $1 \leq k < n$.

Variant MergeSort($A[1 \dots n]$): $\Rightarrow T(n)$

1. MergeSort($A[1 \dots n/4]$) $\Rightarrow T(n/4)$
2. MergeSort($A[n/4+1 \dots 2n/4]$) $\Rightarrow T(n/4)$
3. MergeSort($A[2n/4+1 \dots 3n/4]$) $\Rightarrow T(n/4)$
4. MergeSort($A[3n/4+1 \dots n]$) $\Rightarrow T(n/4)$
5. Merge($A[1 \dots n/4], A[n/4+1 \dots 2n/4], A[2n/4+1 \dots 3n/4], A[3n/4+1 \dots n]$) $\Rightarrow \theta(n)$

Problem 2

- (a) Define $T(n)$ as the worst-case running time of this variant on any input of size n . Find a recursive formula for $T(n)$, i.e. write $T(n)$ in terms of $T(k)$ for some $1 \leq k < n$.

Variant MergeSort($A[1 \dots n]$):

$$T(n) = 4T(n/4) + \theta(n)$$

=>

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

$$T(1) = C, C > 0$$

Problem 2

(b) Draw the corresponding recursion tree and find the explicit answer for $T(n)$.

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

$$T(1) = C, C > 0$$

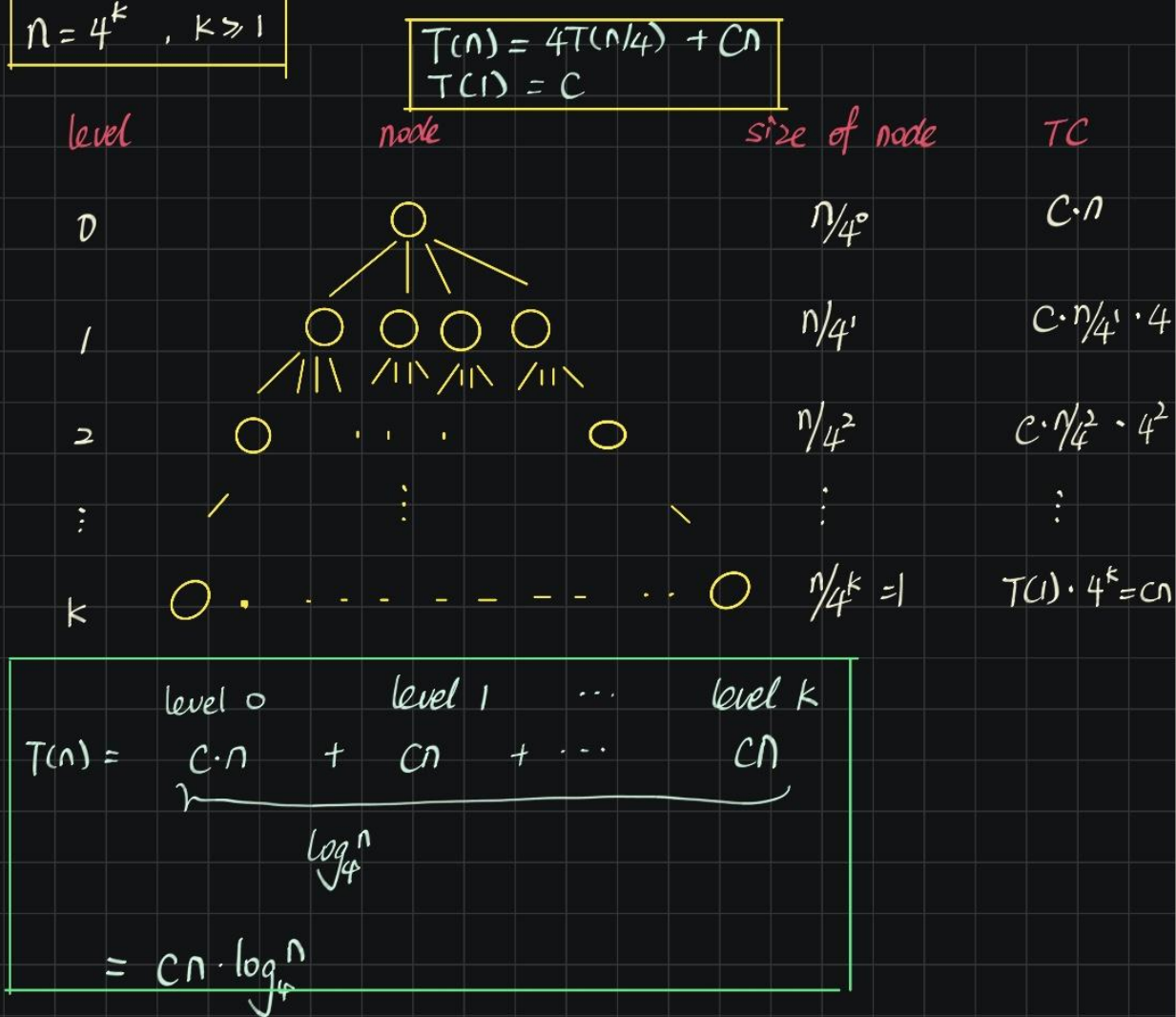
Problem 2

(b) Draw the corresponding recursion tree

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

$$T(1) = C, C > 0$$

$$\Rightarrow T(n) = \theta(n \log_4(n))$$



Problem 2

(c) Prove your result in part (b) formally using **strong induction**.

$$T(n) = \theta(n \log_4(n))$$

=> obtain both an **upper** bound and a **lower** bound.

$$\Rightarrow T(n) = O(n \log_4(n)) \text{ and } T(n) = \Omega(n \log_4(n))$$

Problem 2

(c) Prove your result in part (b) formally using **strong induction**.

Prove: $T(n) = O(n \log_4(n))$

We need to show that there exists a constant $C > 0$ s.t. $T(n) \leq C n \log_4(n)$ for large values of n .

Problem 2

(c) Prove your result in part (b) for

Prove: $T(n) = O(n \log_4(n))$

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

$$T(1) = 1$$

$$T(n) \leq C n \log_4(n)$$

(1) Base Case ($n=4$): choose C s.t

$$T(4) \leq C \cdot 4 \cdot \log_4^4$$

$$\boxed{T(4) = 4T(1) + 4 = 8}$$

$$\Rightarrow 8 \leq 4 \cdot C$$

$$2 \leq C$$

(2) Inductive step:

• Assumption: (*) holds for $n < k$ \Rightarrow

• Conclusion: prove (*) holds for $n=k$

$$n = k/4$$

$$T(k/4) \leq C \cdot k/4 \cdot \log_4^{(k/4)}$$

$$T(k) = 4T(k/4) + k$$

$$\leq 4 \cdot C \cdot k/4 \cdot \log_4^{(k/4)} + k$$

$$\leq kC (\log_4^k - \log_4^4) + k$$

$$\leq kC \log_4^k + \underbrace{k(1-C)}_{\leq 0}$$

$$\leq Ck \log_4^k$$

$$\boxed{C \geq 2}$$

Problem 2

(c) Prove your result in part (b) formally using **strong induction**.

Prove: $T(n) = \Omega(n \log_4(n))$

We need to show that there exists a constant $C > 0$ s.t. $T(n) \geq C n \log_4(n)$ for large values of n .

Problem 2

(c) Prove your result in part (c)

Prove: $T(n) = \Omega(n \log_4(n))$

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

$$T(1) = 1$$

$$T(n) \geq C n \log_4(n)$$

(1) Base Case ($n=4$): choose C s.t

$$T(4) \geq C \cdot 4 \cdot \log_4^4$$

$$T(4) = 4T(1) + 4 = 8$$

$$\Rightarrow 8 \geq 4 \cdot C \\ 2 \geq C$$

(2) Inductive step:

• Assumption: (*) holds for $n < k$ \Rightarrow

• Conclusion: prove (*) holds for $n=k$

$$n = k/4$$

$$T(k/4) \geq C \cdot k/4 \cdot \log_4^{(k/4)}$$

$$T(k) = 4T(k/4) + k$$

$$\geq 4 \cdot C \cdot k/4 \cdot \log_4^{(k/4)} + k$$

$$= kC (\log_4^k - \log_4^4) + k$$

$$= kC \log_4^k + \underbrace{k(1-C)}$$

If $0 < C \leq 1$, $k(1-C) \geq 0$.

$$T(n) \geq kC \cdot \log_4^k$$

Problem 3

Problem 3 (32 points)

For each of the following recursions, draw the recursion tree and find the height of the tree, the running time of each layer, and the sum of running times for all layers. Then use this information to find the explicit answer for $T(n)$.

For full credit, your answers must use $\Theta(\cdot)$ notation, i.e., you should obtain both an upper bound and a lower bound.

(a) $T(n) = T(n - 1) + n.$

(b) $T(n) = 2T(n/4) + \sqrt{n}.$

You may assume that n is a power of 4, i.e., $n = 4^k$ for some positive integer k .

(c) $T(n) = 9T(n/3) + n^2.$

You may assume that n is a power of 3, i.e., $n = 3^k$ for some positive integer k .

(d) $T(n) = T(n/2) + 1.$

You may assume that n is a power of 2, i.e., $n = 2^k$ for some positive integer k .

Problem 3

(a) $T(n) = T(n - 1) + n.$

Problem 3

(a) $T(n) = T(n-1) + n.$

height

$n \dots \dots | \cdot n = n$

$n-1 \dots \dots | \cdot (n-1) = n-1$

\vdots

$=n$ $x \dots \dots | \cdot x = x$

\vdots

$2 \dots \dots | \cdot 2 = 2$

$1 \dots \dots | \cdot T(1) = 1$

sum = $| \cdot T(1) + \sum_{x=2}^n x$

$= 1 + 2 + \dots + n-1 + n = \frac{n(n+1)}{2}$

Thus, $T(n) = \Theta(n^2)$

Problem 3

(b) $T(n) = 2T(n/4) + \sqrt{n}$.

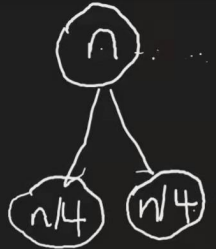
You may assume that n is a power of 4, i.e., $n = 4^k$ for some positive integer k .

Problem 3

(b) $T(n) = 2T(n/4) + \sqrt{n}$.

You may assume that n is a power of 4.

Recursion tree diagram and calculations:



1. $\sqrt{n} = \sqrt{n}$

2. $2 \cdot \sqrt{n/4} = \sqrt{n}$

height

$\log_4^n + 1$

$n/4^x$

$2^x \cdot \sqrt{n/4^x} = \sqrt{n}$

① $2^{\log_4^n} \cdot T(1) = \sqrt{n}$

sum = $2^{\log_4^n} \cdot T(1) + \sum_{x=0}^{\log_4^n - 1} \sqrt{n}$

$= (\log_4^n + 1) \cdot \sqrt{n}$

Thus, $T(n) = \Theta(\sqrt{n} \cdot \log n)$

Problem 3

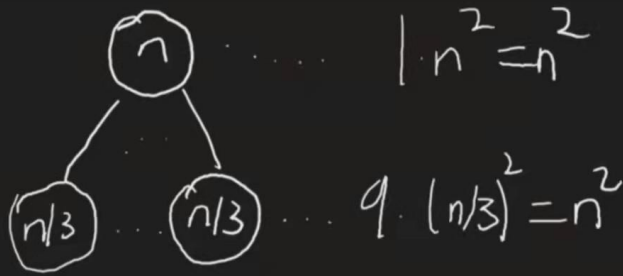
(c) $T(n) = 9T(n/3) + n^2$.

You may assume that n is a power of 3, i.e., $n = 3^k$ for some positive integer k .

Problem 3

(c) $T(n) = 9T(n/3) + n^2$.

You may assume that n is a power of 3, i.e., $n = 3^k$ for some non-negative integer k .



Recursion tree diagram showing the work at each level. The root node is n , and it branches into two children, both labeled $n/3$. The work at the root is n^2 , and the work at the children is $9 \cdot (n/3)^2 = n^2$.

Recursion tree diagram showing the work at each level. The root node is n , and it branches into two children, both labeled $n/3$. The work at the root is n^2 , and the work at the children is $9 \cdot (n/3)^2 = n^2$.

$$= (\log_3 n + 1) \cdot n^2$$

Thus, $T(n) = \Theta(n^2 \log n)$

Problem 3

(d) $T(n) = T(n/2) + 1$.

You may assume that n is a power of 2, i.e., $n = 2^k$ for some positive integer k .

Problem 3

(d) $T(n) = T(n/2) + 1.$

You may assume that n is

positive integer k .

height

$$\begin{aligned}
 & \begin{array}{c} \textcircled{n} \\ | \\ \textcircled{n/2} \\ | \\ \textcircled{n/4} \\ | \\ \textcircled{n/8} \\ | \\ \textcircled{n/16} \\ | \\ \textcircled{n/32} \\ | \\ \textcircled{n/64} \\ | \\ \textcircled{n/128} \\ | \\ \textcircled{n/256} \\ | \\ \textcircled{n/512} \\ | \\ \textcircled{n/1024} \\ | \\ \textcircled{n/2048} \\ | \\ \textcircled{n/4096} \\ | \\ \textcircled{n/8192} \\ | \\ \textcircled{n/16384} \\ | \\ \textcircled{n/32768} \\ | \\ \textcircled{n/65536} \\ | \\ \textcircled{n/131072} \\ | \\ \textcircled{n/262144} \\ | \\ \textcircled{n/524288} \\ | \\ \textcircled{n/1048576} \\ | \\ \textcircled{n/2097152} \\ | \\ \textcircled{n/4194304} \\ | \\ \textcircled{n/8388608} \\ | \\ \textcircled{n/16777216} \\ | \\ \textcircled{n/33554432} \\ | \\ \textcircled{n/67108864} \\ | \\ \textcircled{n/134217728} \\ | \\ \textcircled{n/268435456} \\ | \\ \textcircled{n/536870912} \\ | \\ \textcircled{n/1073741824} \\ | \\ \textcircled{n/2147483648} \\ | \\ \textcircled{n/4294967296} \\ | \\ \textcircled{n/8589934592} \\ | \\ \textcircled{n/17179869184} \\ | \\ \textcircled{n/34359738368} \\ | \\ \textcircled{n/68719476736} \\ | \\ \textcircled{n/137438953472} \\ | \\ \textcircled{n/274877906944} \\ | \\ \textcircled{n/549755813888} \\ | \\ \textcircled{n/1099511627776} \\ | \\ \textcircled{n/2199023255552} \\ | \\ \textcircled{n/4398046511104} \\ | \\ \textcircled{n/8796093022208} \\ | \\ \textcircled{n/17592186044416} \\ | \\ \textcircled{n/35184372088832} \\ | \\ \textcircled{n/70368744177664} \\ | \\ \textcircled{n/140737488355328} \\ | \\ \textcircled{n/281474976710656} \\ | \\ \textcircled{n/562949953421312} \\ | \\ \textcircled{n/1125899906842624} \\ | \\ \textcircled{n/2251799813685248} \\ | \\ \textcircled{n/4503599627370496} \\ | \\ \textcircled{n/9007199254740992} \\ | \\ \textcircled{n/18014398509481984} \\ | \\ \textcircled{n/36028797018963968} \\ | \\ \textcircled{n/72057594037927936} \\ | \\ \textcircled{n/144115188075855872} \\ | \\ \textcircled{n/288230376151711744} \\ | \\ \textcircled{n/576460752303423488} \\ | \\ \textcircled{n/1152921504606846976} \\ | \\ \textcircled{n/2305843009213693952} \\ | \\ \textcircled{n/4611686018427387904} \\ | \\ \textcircled{n/9223372036854775808} \\ | \\ \textcircled{n/18446744073709551616} \\ | \\ \textcircled{n/36893488147419103232} \\ | \\ \textcircled{n/73786976294838206464} \\ | \\ \textcircled{n/147573952589676412928} \\ | \\ \textcircled{n/295147905179352825856} \\ | \\ \textcircled{n/590295810358705651712} \\ | \\ \textcircled{n/1180591620717411303424} \\ | \\ \textcircled{n/2361183241434822606848} \\ | \\ \textcircled{n/4722366482869645213696} \\ | \\ \textcircled{n/9444732965739290427392} \\ | \\ \textcircled{n/18889465931478580854784} \\ | \\ \textcircled{n/37778931862957161709568} \\ | \\ \textcircled{n/75557863725914323419136} \\ | \\ \textcircled{n/151115727451828646838272} \\ | \\ \textcircled{n/302231454903657293676544} \\ | \\ \textcircled{n/604462909807314587353088} \\ | \\ \textcircled{n/1208925819614629174706176} \\ | \\ \textcircled{n/2417851639229258349412352} \\ | \\ \textcircled{n/4835703278458516698824704} \\ | \\ \textcircled{n/9671406556917033397649408} \\ | \\ \textcircled{n/19342813113834066795298816} \\ | \\ \textcircled{n/38685626227668133590597632} \\ | \\ \textcircled{n/77371252455336267181195264} \\ | \\ \textcircled{n/154742504910672534362390528} \\ | \\ \textcircled{n/309485009821345068724781056} \\ | \\ \textcircled{n/618970019642690137449562112} \\ | \\ \textcircled{n/1237940039285380274899124224} \\ | \\ \textcircled{n/2475880078570760549798248448} \\ | \\ \textcircled{n/4951760157141521099596496896} \\ | \\ \textcircled{n/9903520314283042199192993792} \\ | \\ \textcircled{n/19807040628566084398385987584} \\ | \\ \textcircled{n/39614081257132168796771975168} \\ | \\ \textcircled{n/79228162514264337593543950336} \\ | \\ \textcircled{n/158456325028528675187087900672} \\ | \\ \textcircled{n/316912650057057350374175801344} \\ | \\ \textcircled{n/633825300114114700748351602688} \\ | \\ \textcircled{n/1267650600228229401496703205376} \\ | \\ \textcircled{n/2535301200456458802993406410752} \\ | \\ \textcircled{n/5070602400912917605986812821504} \\ | \\ \textcircled{n/10141204801825835211973625643008} \\ | \\ \textcircled{n/20282409603651670423947251286016} \\ | \\ \textcircled{n/40564819207303340847894502572032} \\ | \\ \textcircled{n/81129638414606681695789005144064} \\ | \\ \textcircled{n/162259276829213363391578010288128} \\ | \\ \textcircled{n/324518553658426726783156020576256} \\ | \\ \textcircled{n/649037107316853453566312041152512} \\ | \\ \textcircled{n/1298074214633706907132624082305024} \\ | \\ \textcircled{n/2596148429267413814265248164610048} \\ | \\ \textcircled{n/5192296858534827628530496329220096} \\ | \\ \textcircled{n/10384593717069655257060992658440192} \\ | \\ \textcircled{n/20769187434139310514121985316880384} \\ | \\ \textcircled{n/41538374868278621028243970633760768} \\ | \\ \textcircled{n/83076749736557242056487941267521536} \\ | \\ \textcircled{n/166153499473114484112975882535043072} \\ | \\ \textcircled{n/332306998946228968225951765070086144} \\ | \\ \textcircled{n/664613997892457936451903530140172288} \\ | \\ \textcircled{n/1329227995784915872903807060280344576} \\ | \\ \textcircled{n/2658455991569831745807614120560689152} \\ | \\ \textcircled{n/5316911983139663491615228241121378304} \\ | \\ \textcircled{n/10633823966279326983230456482242756608} \\ | \\ \textcircled{n/21267647932558653966460912964485513216} \\ | \\ \textcircled{n/42535295865117307932921825928971026432} \\ | \\ \textcircled{n/85070591730234615865843651857942052864} \\ | \\ \textcircled{n/170141183460469231731687303715884105728} \\ | \\ \textcircled{n/340282366920938463463374607431768211456} \\ | \\ \textcircled{n/680564733841876926926749214863536422912} \\ | \\ \textcircled{n/1361129467683753853853498429727072845824} \\ | \\ \textcircled{n/2722258935367507707706996859454145691648} \\ | \\ \textcircled{n/5444517870735015415413993718908291383296} \\ | \\ \textcircled{n/10889035741470030830827987437816582766592} \\ | \\ \textcircled{n/21778071482940061661655974875633165533184} \\ | \\ \textcircled{n/43556142965880123323311949751266331066368} \\ | \\ \textcircled{n/87112285931760246646623899502532662132736} \\ | \\ \textcircled{n/174224571863520493293247799005065324265472} \\ | \\ \textcircled{n/348449143727040986586495598010130648530944} \\ | \\ \textcircled{n/696898287454081973172991196020261297061888} \\ | \\ \textcircled{n/1393796574908163946345982392040522594123776} \\ | \\ \textcircled{n/2787593149816327892691964784081045188247552} \\ | \\ \textcircled{n/5575186299632655785383929568162090376495104} \\ | \\ \textcircled{n/11150372599265311570767859136324180752990208} \\ | \\ \textcircled{n/22300745198530623141535718272648361505980416} \\ | \\ \textcircled{n/44601490397061246283071436545296723011960832} \\ | \\ \textcircled{n/89202980794122492566142873090593446023921664} \\ | \\ \textcircled{n/178405961588244985132285746181186892047843328} \\ | \\ \textcircled{n/356811923176489970264571492362373784095686656} \\ | \\ \textcircled{n/713623846352979940529142984724747568191373312} \\ | \\ \textcircled{n/1427247692705959881058285969449495136382746624} \\ | \\ \textcircled{n/2854495385411919762116571938898990272765493248} \\ | \\ \textcircled{n/5708990770823839524233143877797980545530986496} \\ | \\ \textcircled{n/11417981541647679048466287755595961091061972992} \\ | \\ \textcircled{n/22835963083295358096932575511191922182123945984} \\ | \\ \textcircled{n/45671926166590716193865151022383844364247891968} \\ | \\ \textcircled{n/91343852333181432387730302044767688728495783936} \\ | \\ \textcircled{n/182687704666362864775460604089535377456991567872} \\ | \\ \textcircled{n/365375409332725729550921208179070754913983135744} \\ | \\ \textcircled{n/730750818665451459101842416358141509827966271488} \\ | \\ \textcircled{n/1461501637330902918203684832716283019655932542976} \\ | \\ \textcircled{n/2923003274661805836407369665432566039311865085952} \\ | \\ \textcircled{n/5846006549323611672814739330865132078623730171904} \\ | \\ \textcircled{n/11692013098647223345629478661730264157247460343808} \\ | \\ \textcircled{n/23384026197294446691258957323460528314494920687616} \\ | \\ \textcircled{n/46768052394588893382517914646921056628989841375232} \\ | \\ \textcircled{n/93536104789177786765035829293842113257979682750464} \\ | \\ \textcircled{n/187072209578355573530071658587684226515959365500928} \\ | \\ \textcircled{n/374144419156711147060143317175368453031918731001856} \\ | \\ \textcircled{n/748288838313422294120286634350736906063837462003712} \\ | \\ \textcircled{n/1496577676626844588240573268701473812127674924007424} \\ | \\ \textcircled{n/2993155353253689176481146537402947624255349848014848} \\ | \\ \textcircled{n/5986310706507378352962293074805895248510699696029696} \\ | \\ \textcircled{n/11972621413014756705924586149611790497021399392059392} \\ | \\ \textcircled{n/23945242826029513411849172299223580994042798784118784} \\ | \\ \textcircled{n/47890485652059026823698344598447161988085597568237568} \\ | \\ \textcircled{n/95780971304118053647396689196894323976171195136475136} \\ | \\ \textcircled{n/191561942608236107294793378393788647952342390272950272} \\ | \\ \textcircled{n/383123885216472214589586756787577295904684780545900544} \\ | \\ \textcircled{n/766247770432944429179173513575154591809369561091801088} \\ | \\ \textcircled{n/1532495540865888858358347027150309183618739122183602176} \\ | \\ \textcircled{n/3064991081731777716716694054300618367237478244367204352} \\ | \\ \textcircled{n/6129982163463555433433388108601236734474956488734408704} \\ | \\ \textcircled{n/12259964326927110866866776217202473468949912977468817408} \\ | \\ \textcircled{n/24519928653854221733733552434404946937899825954937634816} \\ | \\ \textcircled{n/49039857307708443467467104868809893875799651909875269632} \\ | \\ \textcircled{n/98079714615416886934934209737619787751599303819750539264} \\ | \\ \textcircled{n/196159429230833773869868419475239575503198607639501078528} \\ | \\ \textcircled{n/392318858461667547739736838950479151006397215279002157056} \\ | \\ \textcircled{n/784637716923335095479473677900958302012794430558004314112} \\ | \\ \textcircled{n/1569275433846670190958947355801916604025588861116008628224} \\ | \\ \textcircled{n/3138550867693340381917894711603833208051177722232017256448} \\ | \\ \textcircled{n/6277101735386680763835789423207666416102355444464034512896} \\ | \\ \textcircled{n/12554203470773361527671578846415332832204710888928069025792} \\ | \\ \textcircled{n/25108406941546723055343157692830665664409421777856138051584} \\ | \\ \textcircled{n/50216813883093446110686315385661331328818843555712276103168} \\ | \\ \textcircled{n/100433627766186892221372630771322662657637687111424552206336} \\ | \\ \textcircled{n/200867255532373784442745261542645325315275374222849104412672} \\ | \\ \textcircled{n/401734511064747568885490523085290650630550748445698208825344} \\ | \\ \textcircled{n/803469022129495137770981046170581301261101496891396417650688} \\ | \\ \textcircled{n/1606938044258990275541962092341162602522202993782792835301376} \\ | \\ \textcircled{n/3213876088517980551083924184682325205044405987565585670602752} \\ | \\ \textcircled{n/6427752177035961102167848369364650410088811975131171341205504} \\ | \\ \textcircled{n/12855504354071922204335696738729300820177623950262342682411008} \\ | \\ \textcircled{n/25711008708143844408671393477458601640355247900524685364822016} \\ | \\ \textcircled{n/51422017416287688817342786954917203280710495801049370729644032} \\ | \\ \textcircled{n/102844034832575377634685573909834406561420991602098741459288064} \\ | \\ \textcircled{n/205688069665150755269371147819668813122841983204197482918576128} \\ | \\ \textcircled{n/411376139330301510538742295639337626245683966408394965837152256} \\ | \\ \textcircled{n/822752278660603021077484591278675252491367932816789931674304512} \\ | \\ \textcircled{n/1645504557321206042154969182557350504982735865633579863348609024} \\ | \\ \textcircled{n/3291009114642412084309938365114701009965471731267159726697218048} \\ | \\ \textcircled{n/6582018229284824168619876730229402019930943462534319453394436096} \\ | \\ \textcircled{n/13164036458569648337239753460458804039861886925068638906788872192} \\ | \\ \textcircled{n/26328072917139296674479506920917608079723773850137277813577744384} \\ | \\ \textcircled{n/52656145834278593348959013841835216159447547700274555627155488768} \\ | \\ \textcircled{n/105312291668557186697918027683670432318895095400549111254310977536} \\ | \\ \textcircled{n/210624583337114373395836055367340864637790190801098222508621955072} \\ | \\ \textcircled{n/421249166674228746791672110734681729275580381602196445017243910144} \\ | \\ \textcircled{n/842498333348457493583344221469363458551160763204392890034487820288} \\ | \\ \textcircled{n/1684996666696914987166688442938726917102321526408785780068975640576} \\ | \\ \textcircled{n/3369993333393829974333376885877453834204643052817571560137951281152} \\ | \\ \textcircled{n/6739986666787659948666753771754907668409286105635143120275902562304} \\ | \\ \textcircled{n/13479973333575319897333507543509815336818572211270286240551805124608} \\ | \\ \textcircled{n/26959946667150639794667015087019630673637144422540572481103610249216} \\ | \\ \textcircled{n/53919893334301279589334030174039261347274288845081144962207220498432} \\ | \\ \textcircled{n/107839786668602559178668060348078522694548577690162289924414440996864} \\ | \\ \textcircled{n/215679573337205118357336120696157045389097155380324579848828881993728} \\ | \\ \textcircled{n/431359146674410236714672241392314090778194310760649159697657763987456} \\ | \\ \textcircled{n/862718293348820473429344482784628181556388621521298319395315527974912} \\ | \\ \textcircled{n/1725436586697640946858688965569256363112777243042596638790631055949824} \\ | \\ \textcircled{n/3450873173395281893717377931138512726225554486085193277581262111899648} \\ | \\ \textcircled{n/6901746346790563787434755862277025452451108972170386555162524223799296} \\ | \\ \textcircled{n/13803492693581127574869511724554050904902217944340773110325048447598592} \\ | \\ \textcircled{n/27606985387162255149739023449108101809804435888681546220650096895197184} \\ | \\ \textcircled{n/55213970774324510299478046898216203619608871777363092441300193790394368} \\ | \\ \textcircled{n/110427941548649020598956093796432407239217743554726184882600387580788736} \\ | \\ \textcircled{n/220855883097298041197912187592864814478435487109452369765200775161577472} \\ | \\ \textcircled{n/441711766194596082395824375185729628956870974218904739530401550323154944} \\ | \\ \textcircled{n/883423532389192164791648750371459257913741948437809479060803100646309888} \\ | \\ \textcircled{n/1766847064778384329583297500742918515827483896875618958121606201292619776} \\ | \\ \textcircled{n/3533694129556768659166595001485837031654967793751237916243212402585239552} \\ | \\ \textcircled{n/7067388259113537318333190002971674063309935587502475832486424805170479104} \\ | \\ \textcircled{n/14134776518227074636666380005943348126619871175004951664972849610340958208} \\ | \\ \textcircled{n/28269553036454149273332760011886696253239742350009903329945699220681916416} \\ | \\ \textcircled{n/56539106072908298546665520023773392506479484700019806659891398441363832832} \\ | \\ \textcircled{n/113078212145816597093331040047546785012958969400039613319782796882727665664} \\ | \\ \textcircled{n/226156424291633194186662080095093570025917938800079226639565593765455331328} \\ | \\ \textcircled{n/452312848583266388373324160190187140051835877600158453279131187530910662656} \\ | \\ \textcircled{n/904625697166532776746648320380374280103671755200316906558262375061821325312} \\ | \\ \textcircled{n/1809251394333065553493296640760748560207343510400633813116524750123642650624} \\ | \\ \textcircled{n/36185027886661311069865932815214971204146870208012$$

Problem 3

You may find that the running time of each layer is same in all four problems.

This is a special case when drawing the recursion tree.

In following lectures, you will meet more general cases and learn how to solve them.

Problem 4

Problem 4 (10 points)

Recall that we showed the stability of `INSERTION_SORT`.

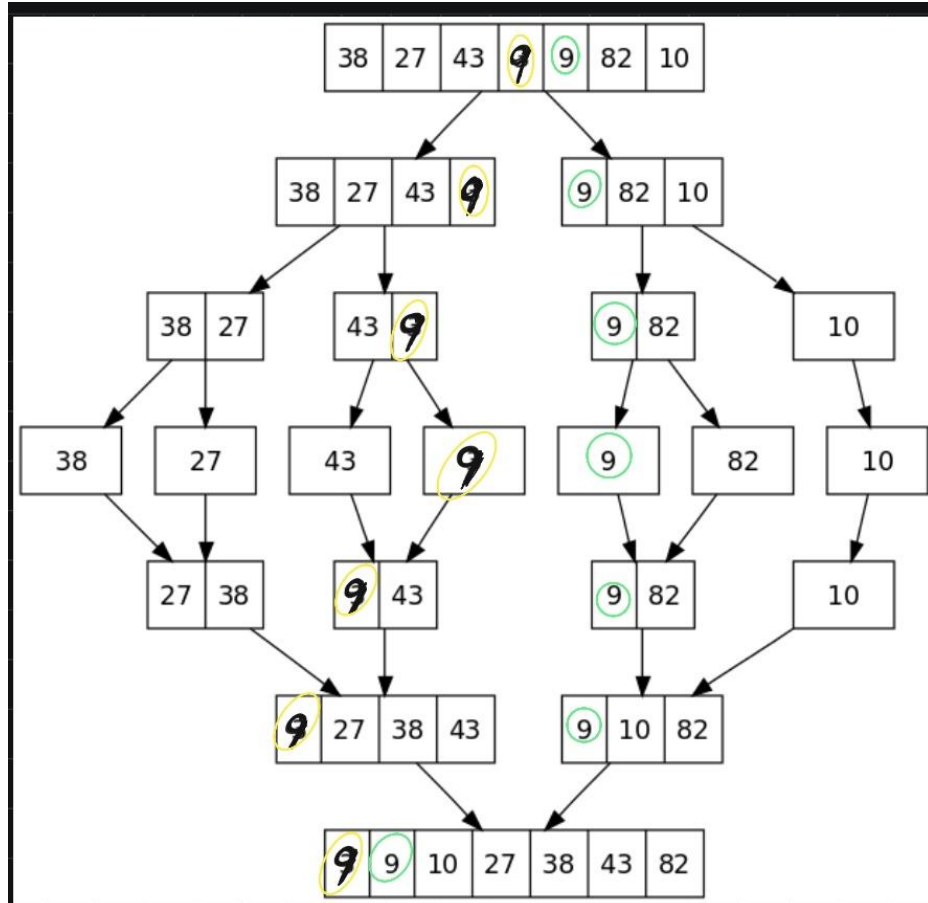
Show that `MERGE_SORT` is a stable sorting algorithm. For that, you should show that the stability of the input is preserved under all the steps undertaken by `MERGE_SORT`.

Problem 4

Show that Merge sort is
a stable sorting algorithm.

Recall MergeSort($A[1\dots n]$):

1. MergeSort($A[1\dots n/2]$)
2. MergeSort($A[n/2+1\dots n]$)
3. Merge($A[1\dots n/2]$, $A[n/2+1\dots n]$)



merge
level 1
level 2
level 2
level 3

Problem 4

Prove: Merge sort is a stable sorting algorithm.

=> Use Strong Induction

MergeSort($A[1\dots n]$):

1. MergeSort($A[1\dots n/2]$)
2. MergeSort($A[n/2+1\dots n]$)
3. Merge($A[1\dots n/2]$, $A[n/2+1\dots n]$)

```
1 Merge(B[1...k], C[1...m]):
2     define D[1...k+m]
3     i = 1
4     j = 1
5     for l = 1 to k+m:
6         if j > m or (i <= k and B[i] <= C[j]):
7             D[l] = B[i]
8             i = i + 1
9         else:
10            D[l] = c[j]
11            j = j + 1
12     return D
```

Problem 4

(I)Base Case: check the algorithm preserves the stability for inputs of size $n=1$:

After we call the `mergeSort(A[1...1])`, the array $A[1...1]$ is stable. The only element in A is unchanged and stable itself.

Problem 4

(II) Inductive Step:

Assumption: assume that the algorithm preserves the stability for inputs of size $1, 2, \dots, k$ where $k < n$.

Problem 4

(II) Inductive Step:

Conclusion: prove that the algorithm preserves the stability for inputs of n:

```
1  Merge(B[1...k], C[1...m]):
2      define D[1...k+m]
3      i = 1
4      j = 1
5      for l = 1 to k+m:
6          if j > m or (i <= k and B[i] <= C[j]):
7              D[l] = B[i]
8              i = i + 1
9          else:
10             D[l] = c[j]
11             j = j + 1
12     return D
```

Problem 4

We know that the algorithm preserves the stability for inputs of size $n/2$ (the recursive parts), then $B[1 \dots n/2]$ and $C[1 \dots n/2]$ are all sorted with stability.

Then we call `merge(B[1...n/2],C[1...n/2])`; inside of the for-loop in the merge function, whenever there is $B[i] \leq C[j]$, $B[i]$ will always come to the front of $C[j]$ with its original relative orders ($B[i]$ comes from left part, $C[j]$ comes from right part).

Then after merging B and C , the sorted array D also preserves the stability as A .

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Although you may find this problem meaningless,
it is designed to form your mind of divide-and-conquer.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

1. Define the base case.

That is, when the current problem is small enough, it will be quite easy to quickly compute the result. (usually $O(1)$)

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

1. Define the base case.

That is, when the current problem is small enough, compute the result. (usually $O(1)$)

MERGE-SORT(A, p, r)

1 if $p < r$

2 then $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q+1, r$)

5 MERGE(A, p, q, r)

quickly

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

2. Otherwise, divide the current problem into some similar(or same) problems with smaller size, and use recursion to solve them.

After calling all recursions, we know these smaller problems have been solved while we don't need to consider how.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

2. Otherwise, divide the current problem into two smaller problems, and use recursion to solve them.

After calling all recursions, we know the answer. We don't need to consider how.

MERGE-SORT(A, p, r)

1 if $p < r$

2 then $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q+1, r$)

5 MERGE(A, p, q, r)

elements with

solved while

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

3. Find a method to make use of the result of all recursions in order to compute the answer of the current problem.

Usually the most difficult part of divide-and-conquer

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

The idea of divide-and-conquer consists of three steps:

3. Find a method to make use of the recursive answer of the current problem.

MERGE-SORT(A, p, r)

compute the

1 if $p < r$

2 then $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q+1, r$)

5 MERGE(A, p, q, r)

Usually the most difficult part of divide-and-conquer is the merge step.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

When the array consists of only one element, the largest number is itself.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

When the array consists of only one element, the largest number is itself.

Recursions:

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

When the array consists of only one element, the largest number is itself.

Recursions:

Recursively call the left half and right half of the array and get two results.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

When the array consists of only one element, the largest number is itself.

Recursions:

Recursively call the left half and right half of the array and get two results.

Compute the result:

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

Base Case:

When the array consists of only one element, the largest number is itself.

Recursions:

Recursively call the left half and right half of the array and get two results.

Compute the result:

We now have the maximum of left and maximum of right. So the maximum of this array is the larger one between them.

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

```
1 Find_max(A, l, r):  
2     if l == r:  
3         return A[l];  
4     mid = (l + r)/2  
5     val1 = Find_max(A, l, mid)  
6     val2 = Find_max(A, mid+1, r)  
7     return max(val1, val2)  
8  
9 Find_max(A, 1, n)
```

Problem 5

Problem 5 (14 points)

Let $A[1 \dots n]$ be an array consisting of n distinct numbers. Develop a divide-and-conquer algorithm to find the largest number among the elements of A . Write down the pseudo-code for your algorithm.

What is the time complexity of this algorithm?

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

$$T(1) = C$$

$$C > 0$$

Bonus Problem 1

Recall the variant of `SELECTION_SORT` discussed in Homework 1 Problem 5. In each iteration, we needed to find the maximum and the second maximum elements among k given elements. Note that this is feasible by using at most $2k - 3$ comparisons.

- (a) Can you improve the number of comparisons used?
- (b) What is the minimum number of comparisons you can use?
- (c) Can you use at most $k + \log_2 k - 2$ comparisons?
- (d) Show how improving the number of comparisons used affects the time complexity of the variant algorithm.

Bonus Problem 1

(a-c) Find the maximum and the second maximum elements among k given elements, within $k + \log_2(k) - 2$ comparisons.

Bonus Problem 1

$$k + \log_2(k) - 2$$

3, 11, 5, 7, 14, 1, 13, 8

11 [3] 7 [5] 14 [1] 13 [8]

11 [3, 7] 14 [1, 13]

14 [1, 13, 11]

max = 14

Find sec-max : [1, 13, 11]

Total comparison = 7 + 2 = 9

comparisons

4

2

1

$$4 + 2 + 1 = 7$$

2

$A[1 \dots k]$:

Find max : $k/2 + k/4 + \dots + 1 = k - 1$

sec-max : $\log k - 1$

Total = $k - 1 + \log k - 1 = k + \log k - 2$

Bonus Problem 1

(d) Show how improving the number of comparisons used affects the **time complexity** of the variant algorithm.

Comparisons: $k + \log k - 2$ comparisons to find the maximum and the second among k elements.

Swap operations: in each iteration, we need to make at most **2 swaps** to place the maximum and the second maximum at the end of the remaining elements. This gives us $2 * (n/2) = n$ total swappings.

Bonus Problem 1

(d) Show how improving the number of comparisons used affects the **time complexity** of the variant algorithm.

$$\text{Total comparisons} = \sum_{k=2}^n (k + \log k - 2), k = 2, 4 \dots n$$

$$\sum_{k=2}^n (k + \log k - 2)$$

$$= \frac{n^2}{4} + \log\left(\frac{n!}{2}\right)$$

The total running time becomes:

$$T(n) = C_1 * \sum_{k=2}^n (k + \log k - 2) + C_2 * n = C_1 * \left(\frac{n^2}{4} + \log\left(\frac{n!}{2}\right)\right) + C_2 * n,$$

for some constants $C_1, C_2 > 0$.

$$T(n) = O(n^2)$$

Q & A

Thank you