Sorting Algorithms (Part II)

Outline

- Recursion
- Sorting Algorithms using Recursions
  - Merge Sort
  - Quick Sort

Recursion

- Recursion is a method of solving a computational problem by breaking it down into smaller instances of the same problem.

- Recursion solves such recursive problems by using recursive functions that call themselves until a base case is reached.

Recursion

- A recursive function is defined in terms of
  - Recursive case(s)
    - The case for which the solution is expressed in terms of a smaller version of itself.
  - Base case(s)
    - The case that stops the recursion and returns a result.

Examples of Recursive Algorithms: Compute Factorials

- Factorial
  - The factorial of a positive integer is the product of all positive integers less than or equal to it.
  - E.g., $4! = 4 \times 3 \times 2 \times 1 = 4 \times 3!$
  - Recursion
    - Recursive case: $n! = n(n-1)!, n > 0$
    - Base case: $1! = 1$

Examples of Recursive Algorithms: Compute Factorials

```
unsigned long long factorial(unsigned int n) {
    if (n == 1) {
       // Base case: factorial of 0 and 1 is 1
        return 1;
    } else {
       // Recursive case: n! = n * (n-1)!
       return n * factorial(n - 1);
    }
}
```

factorial(4) = 4 * factorial(3)

↳ 3 * factorial(2)

↳ 2 * factorial(1)

↑

base case

factorial(1)=1

Examples of Recursive Algorithms: Binary Search

- Binary search for sorted arrays
  - Probe the middle element of the array
  - If the value is smaller than the middle one, search the left part of the array.
  - Otherwise, search the right part of the array.
  - Repeat the above until the value is found as the middle element, or the size of the array reduces to zero.

0 … $mid-1$	mid	$mid+1$ … $n-1$
$\leq a[mid]$	$a[mid]$	$\geq a[mid]$

Binary Search using Recursion

```
int binarysearch(int a[], int x, int low, int high){
    int mid = (low + high)/2;
    if (low > high)
      return -1;
    else if (a[mid] == x)
      return mid;
    else if (a[mid] < x)
      return binarysearch(a, x, mid+1, high);
    Else // a[mid] > x
      return binarysearch(a, x, low, mid-1);
}
```

base cases

recursive cases

0 ... $mid-1$	mid	$mid+1$ ... $n-1$
$\leq a[mid]$	$a[mid]$	$\geq a[mid]$

Direct vs Indirect Recursive Function

Type	**Definition**	**Example**

Direct recursive function	A function calls itself.	`void func_A(int n){`
		`    if (n>0)`
		`        func_A(n-1);`
		`}`

Indirect recursive function	Multiple functions call each other in a cycle, eventually leading back to the original function.	A calls B, and B calls A.
		
		`void func_A(int n){`
		`    if (n>0)`
		`        func_B(n-1);`
		`}`
		
		`void func_B(int n){`
		`    if (n>0)`
		`        func_A(n-1);`
		`}`

Divide-and-Conquer

- Recursion is a fundamental technique in the design and implementation of divide-and-conquer algorithms.

- Divide-and-conquer is a problem-solving strategy where a problem is divided into smaller subproblems, solved independently, and then combined to solve the original problem.

Divide-and-Conquer

- Divide-and-Conquer consists of three steps:
  - Divide the problem into smaller subproblems;
  - Conquer the sub-problems by solving them recursively or iteratively;
  - Combine the solutions to solve the original problem.

Recursion v.s. Iteration

- Both recursion and iteration are used to solve problems involving repetitive tasks or processes.
- They break down larger problems into smaller, more manageable sub-problems and repeat operations until a specific condition is met.
  - In recursion, a function repeatedly calls itself.
  - In iteration, a set of instructions is executed repeatedly using loops.
- Anything that can be computed iteratively can also be computed recursively, and vice versa.

Example: Recursion v.s. Iteration
Summing Elements of an Array

Iteration

```
int iterativeSum(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += a[i];
    }
    return sum;
}
```

Recursion

```
int recursiveSum(int a[], int n) {
    if (n <= 0)
        return 0;
    else
        return recursiveSum(a, n - 1) + a[n - 1];
}
```

Recursion v.s. Iteration

Q: Is the recursive version usually faster?

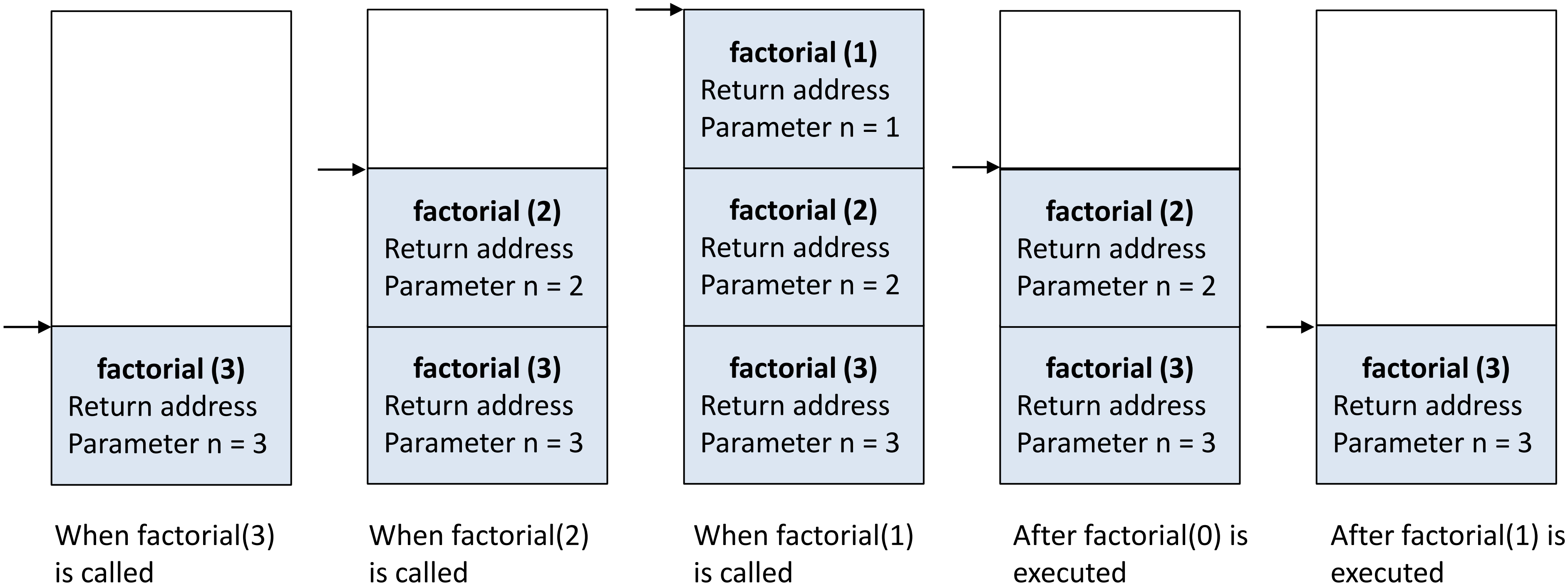
A: No. It's usually **slower** due to the overhead of maintaining the call stack.

Recursion v.s. Iteration

Q: Does the recursive version usually use less memory?

A: No. It usually uses **more memory** (for the call stack).

Call Stack for Recursion



Stack frame (also called activation record) usually includes necessary information about the function call such as argument passed to the function, return address back to the caller function and local variables.

Stack pointer

Recursion

Q: Then **why** use recursion?

A: Sometimes it is much **simpler to write** the recursive version for many divide-and-conquer algorithms.

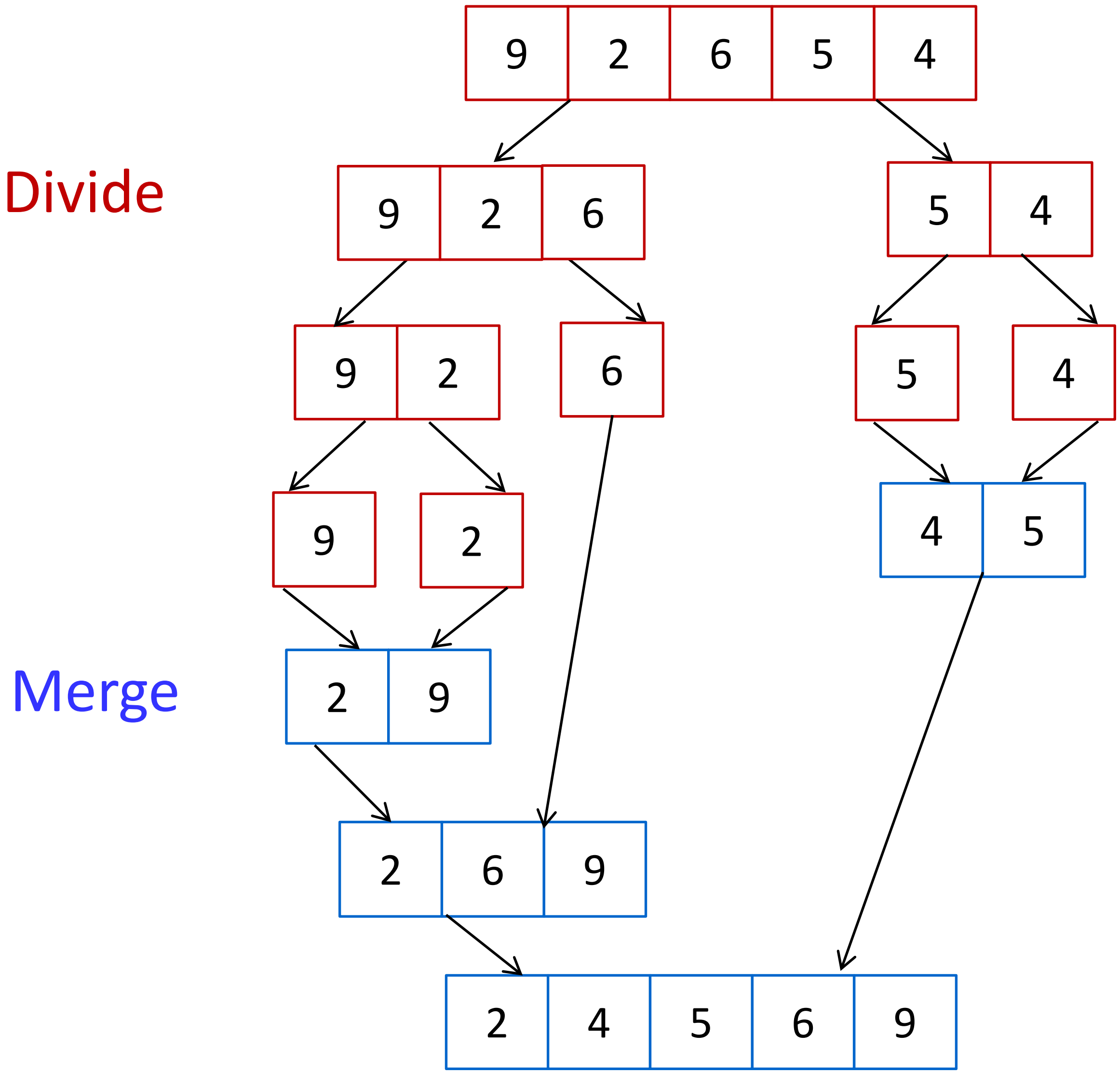
Choosing Between Recursion and Iteration

- **Nature of the Problem:**
  - Some problems (divide-and-conquer problems) are naturally suited for recursion (e.g., tree traversal).
  - Others are more naturally solved using iteration (e.g., summing elements in an array).
- **Performance Considerations:**
  - Recursion might have higher overhead due to maintaining the call stack.
  - Iteration might be preferred in cases where efficiency is crucial.
- **Code Readability and Maintainability:**
  - Recursive solutions can be more elegant and easier to understand for certain problems (divide-and-conquer problems).
  - Iterative solutions may be more readable for simple sequential operations.

Merge Sort

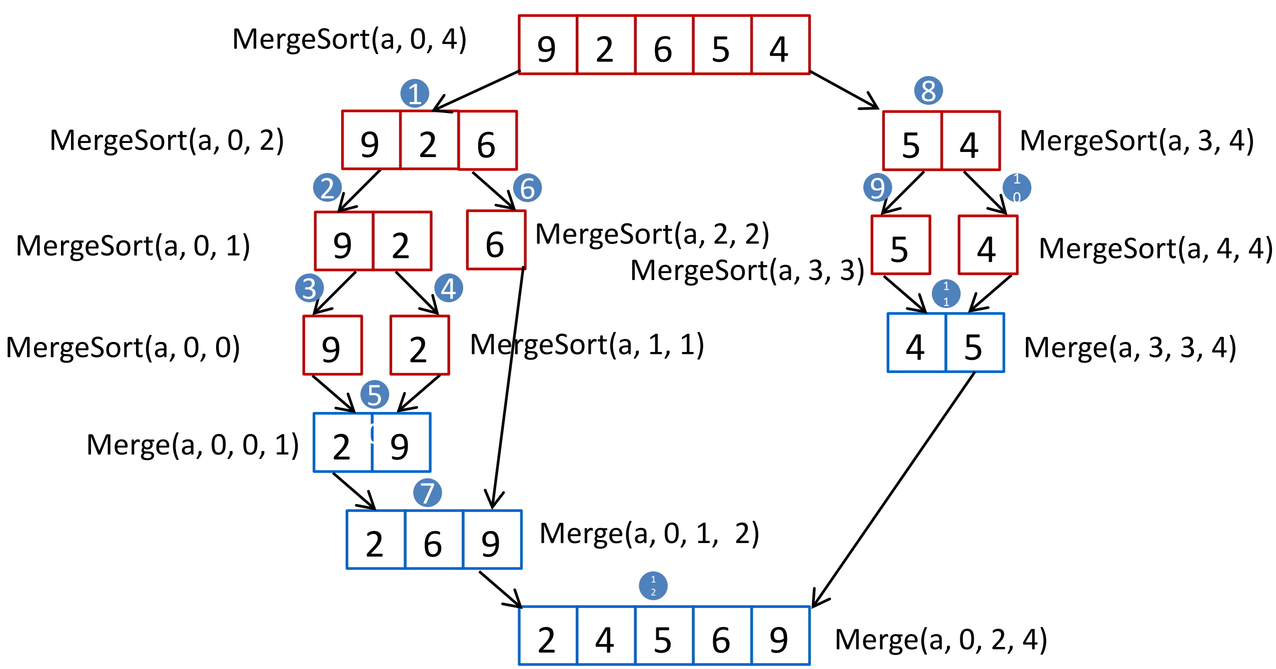
- Main idea: Divide and merge each sub-sequence in order.
  1. Recursively divide the sequence into two nearly equal halves until single elements
  2. Merge them back together in order.

Example: Merge Sort



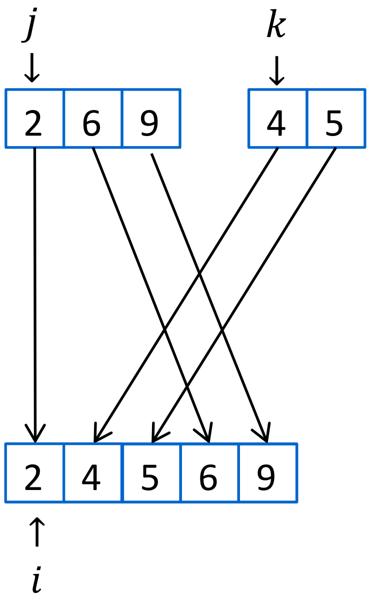
Merge Sort

```
void MergeSort(int a[], int left, int right){
   if (left < right){ // array size > 1
        unsigned const middle = (left+right)/2;
        MergeSort(a,left,middle);
        MergeSort(a,middle+1,right);
        Merge(a,left,middle,right);
   }
}
```



Merge Sort: Merge Function

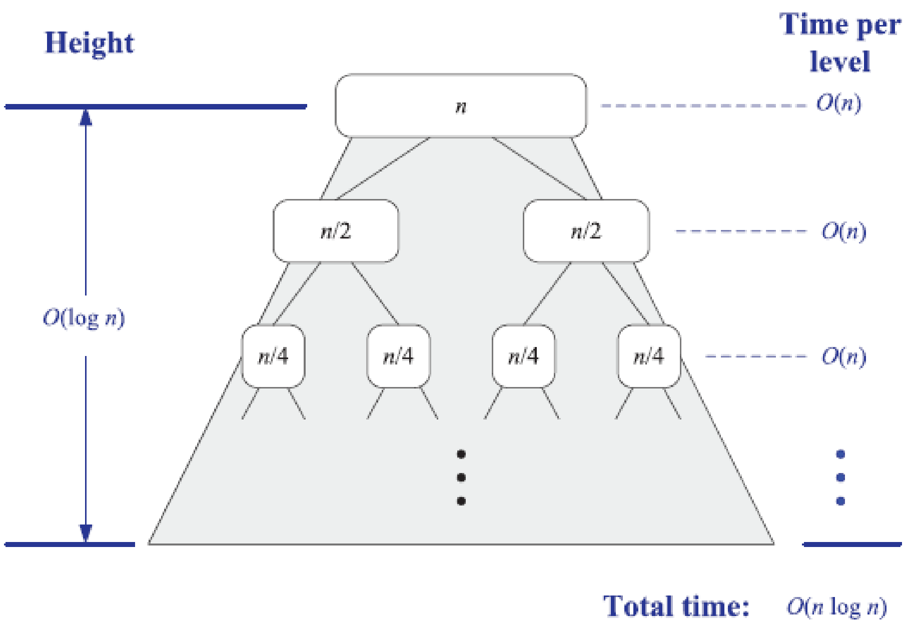
```
void Merge(int array[], int left, int middle, int right){
   int* temp = new int [right-left+1];
   unsigned i = 0;            // counter for the temp array
   unsigned j = left;         // counter for left array
   unsigned k = middle + 1; // counter for right array
   while (j<=middle && k <=right)
     if (array[j] <= array[k])
          temp[i++] = array[j++];
     else
          temp[i++] = array[k++];
   while (j <= middle)
          temp[i++] = array[j++];
   while (k <= right)
          temp[i++] = array[k++];
   for (i=left; i <= right; ++i)
        array[i] = temp[i-left];
   delete [] temp;
}
```



Complexity of Merge Sort

- Time complexity of `Merge()`:
  - $O(n)$ where n is the total number of elements being merged

- Time complexity of `MergeSort()`:
  - $O(n \log n)$



Quick Sort

- Main idea: Divide and sort each sub-sequence based on a pivoted value recursively.

1. Select a pivot element p.
   - Can be arbitrarily chosen.
   - E.g., 1st element can be chosen.

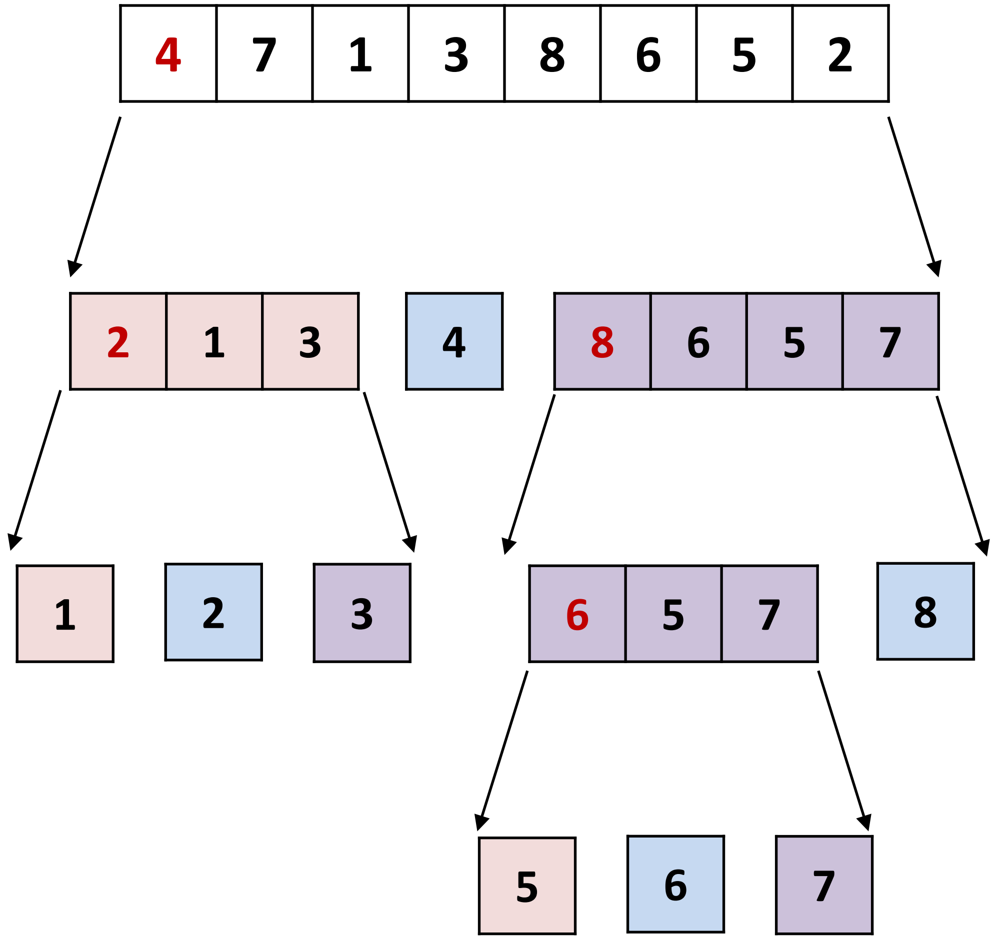
Pivot	Remaining elements

2. Partition the remaining elements in 2 parts: S and L
   a) For each $i \in S, i \leq p$
   b) For each $i \in L, i > p$
   c) Pivot is at in its final sorted position.

Left partition S (Element<Pivot)	**Pivot**	Right partition L (Element≥Pivot)

3. Recursively apply the same process to the unsorted subarrays S and L.

Example: Quick Sort



Quick Sort: Partition

- Initial array before partition

P	**R**	**R**	**R**	**R**	**R**	**R**	**R**	**R**	**R**	**R**	**R**

- As we traverse the array, it will be arranged in the following form
  - If $C \geq P$, then leave it at the same place.
  - If $C < P$, then swap it with the first larger element.

P	**S**	**S**	**S**	**L**	**L**	**L**	**L**	**C**	**R**	**R**	**R**

If $C < P$

- At the end, swap the pivot with the last smaller element.

P	**S**	**S**	**S**	**S**	**S**	**L**	**L**	**L**	**L**	**L**	**L**

$\longrightarrow$

S	**S**	**S**	**S**	**S**	**P**	**L**	**L**	**L**	**L**	**L**	**L**

P: Pivot   S: Smaller   L: Larger   C: Current   R: Remaining

Example: Partition in Quick Sort

Steps	Explanation	Illustration
0	Initial array	4 7 1 3 8 6 5 2
1	Since 7 > 4, leave as it is	4 7 1 3 8 6 5 2 → 4 7 1 3 8 6 5 2
2	Since 1 < 4, swap with 1st larger element.	4 7 1 3 8 6 5 2 → 4 1 7 3 8 6 5 2
3	Since 3 < 4, swap with 1st larger element.	4 1 7 3 8 6 5 2 → 4 1 3 7 8 6 5 2
4	Since 8 > 4, leave as it is	4 1 3 7 8 6 5 2 → 4 1 3 7 8 6 5 2

Pivot Smaller Larger Current Remaining

Example: Partition in Quick Sort

Steps	Explanation	Illustration
6	Since 6 $>$ 4, leave as it is	4 1 3 7 8 6 5 2 → 4 1 3 7 8 6 5 2
7	Since 5 $>$ 4, leave as it is	4 1 3 7 8 6 5 2 → 4 1 3 7 8 6 5 2
8	Since 2 $<$ 4, swap with 1st larger element.	4 1 3 7 8 6 5 2 → 4 1 3 2 8 6 5 7
9	Swap the pivot 4 with the last smaller element 2	4 1 3 2 8 6 5 7 → 2 1 3 4 8 6 5 7

Pivot Smaller Larger Current Remaining

Quick Sort

```
void QuickSort(int a[], int left, int right){
    if(left < right){ // array size > 1
        int i = Partition(a, left, right); // index of the pivot
        QuickSort(a,left, i-1);
        QuickSort(a,i+1, right);
    }
}
```

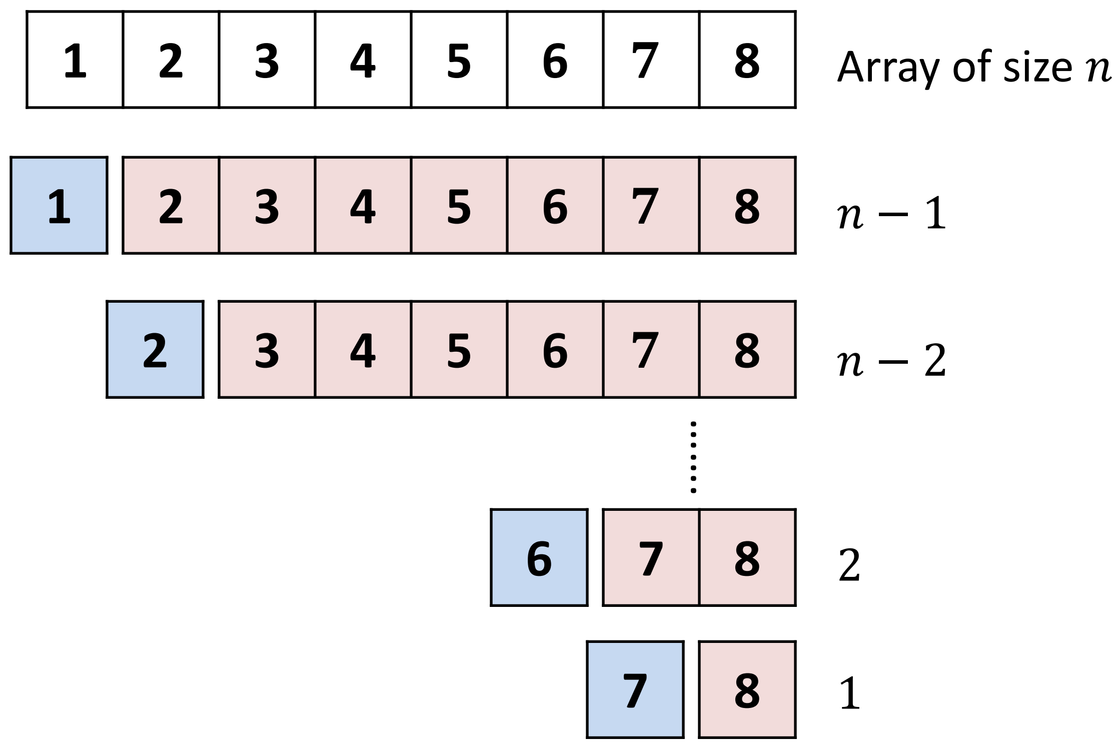
```
unsigned Partition(int a[], int i, int j){
   int p=a[i]; // 1st element as the pivot
   int h=i; // the position of the 1st larger element
   for(int k=i+1;k<=j;++k){ //k: position of the current element
     if(a[k]<p){
        ++h;
        Swap(a[k],a[h]); // swap with the 1st larger element
     }
     // else: don't do anything, keep the item as it is
    }

   Swap(a[h],a[i]); // Move pivot to its correct position
   return h;
}
```

Time Complexity of Quick Sort

- Time complexity of `Partition()`:
  - $O(n)$, n is the number of elements in the subarray being partitioned.
- Time complexity of `QuickSort()`:
  - Worst case:
    - Array is already sorted. Pivot is always the max/min.
    - Time complexity: $O(n^2)$.
  - Best case:
    - Each round divides the two parts into nearly equal size.
    - Time complexity: $O(n\log_2 n)$.

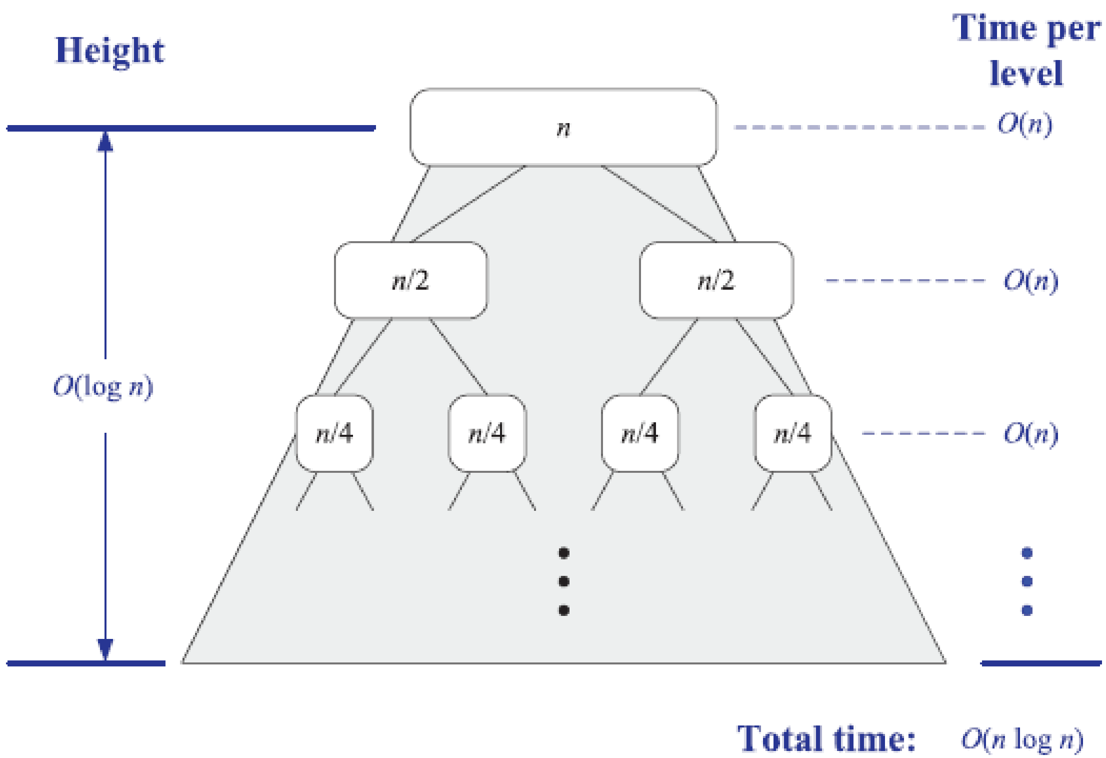
Time Complexity of Quick Sort

- Worst Case:
  - $(n-1)+(n-2)+(n-3)+\ldots+1 \;=\; O(n^2)$.



Time Complexity of Quick Sort

- Best Case:
  - $O(n\log_2 n)$.



Randomized Quick Sort

```
void RandomQuickSort(int a[], int left, int right){
    if(left < right){
        int i = RandomPartition(a, left, right);
        RandomQuickSort(a,left, i-1);
        RandomQuickSort(a,i+1, right);
    }
}
```

```
unsigned RandomPartition(int a[], int i, int j){
   int r = rand() % (j-i)+i+1;
   Swap(a[i],a[r]); //swap the randomly chosen element with the 1st element
   int p=a[i];
   int h=i;
   for(int k=i+1;k<=j;++k)
     if(a[k]<p){
        ++h;
        Swap(a[k],a[h]);
     }
   Swap(a[h],a[i]);
   return h;
}
```

- By randomly choosing the pivot, the split of the input array is expected to be reasonably well balanced on average.
- Average time complexity: $O(n \log n)$

Comparison: Merge Sort vs Quick Sort

	Time Complexity	**Stability**	**In-place**	**Adaptability**
Merge Sort	Consistent $O(n\log_2 n)$	Yes	No	No
Quick Sort	Average case $O(n\log_2 n)$	No	Yes	Yes (using random pivot)

Summary

- Recursion
- Sorting Algorithms using recursions
  - Merge Sort
  - Quick Sort

References

- M. T. Goodrich, R. Mamassia, D. M. Mount, Data Structures and Algorithms in C++, 2nd Ed., Wiley.