SCJ2013 Data Structure & Algorithms

Quick Sort

Nor Bahiah Hj Ahmad & Dayang Norhayati A. Jawawi

Quick Sort Operation

- Quick sort is similar with merge sort in using **divide** and **conquer** technique.

- Differences of Quick sort and Merge sort :

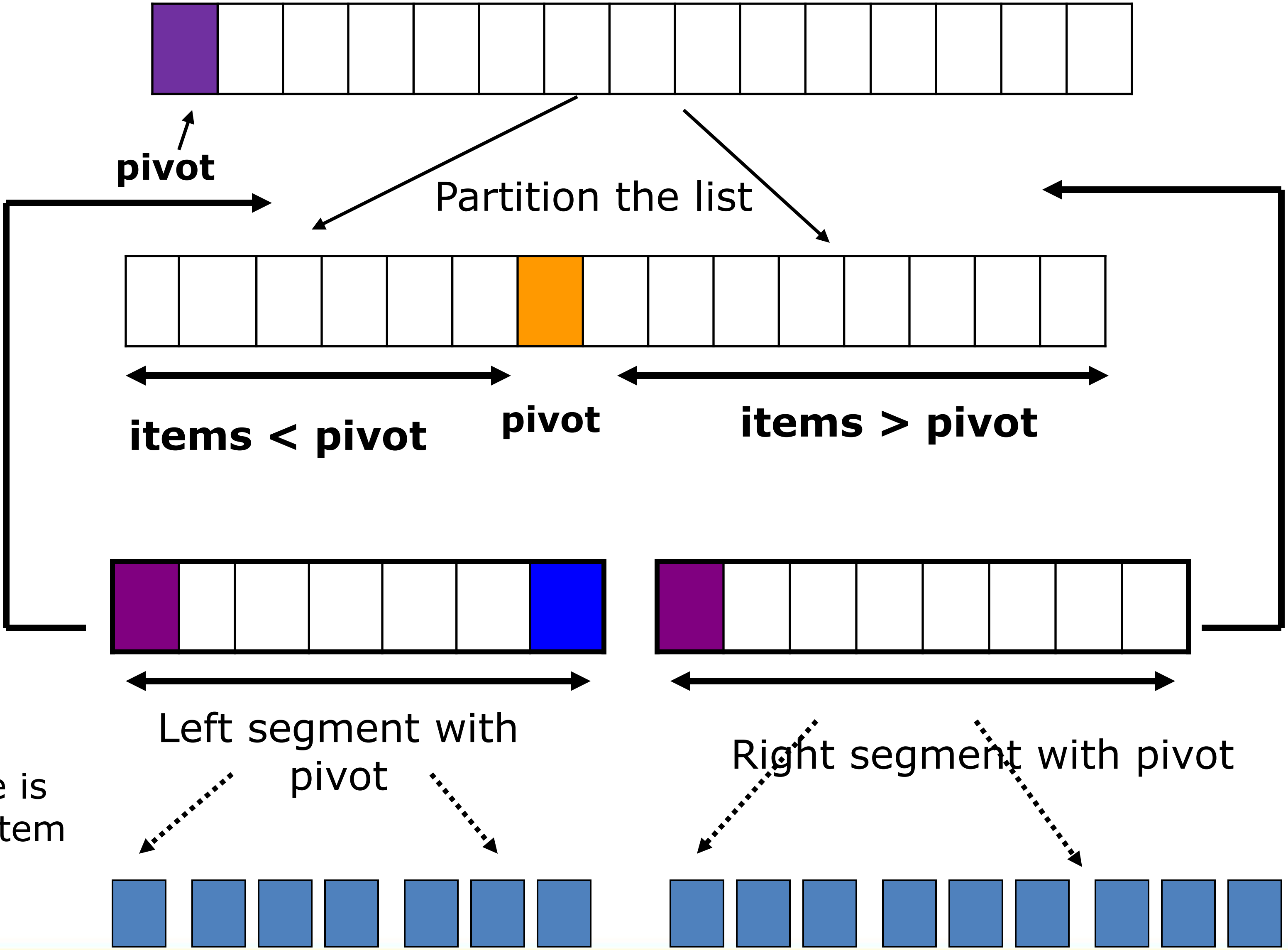
Quick Sort	**Merge Sort**
Partition the list based on the ***pivot* value**	Partition the list by dividing the list into **two**
No merge operation is needed since when there is only one item left in the list to be sorted, all other items are already in sorted position.	**Merge operation** is needed to sort and merge the item in the left and right segment.

Quicksort

- A **divide-and-conquer** algorithm

- Strategy
  - Choose a pivot (first element in the array)
  - Partition the array about the pivot
    - items < pivot
    - items >= pivot
    - Pivot is now in correct sorted position
  - Sort the left section again until there is one item left
  - Sort the right section again until there is one item left

Quick Sort Process



Partition process is repeated until there is only one item left in the list.

Quick Sort Implementation

- **`quickSort()`** function – a recursive function that will partition the list into several sub lists until there is one item left in the sub list.

`quickSort()` function

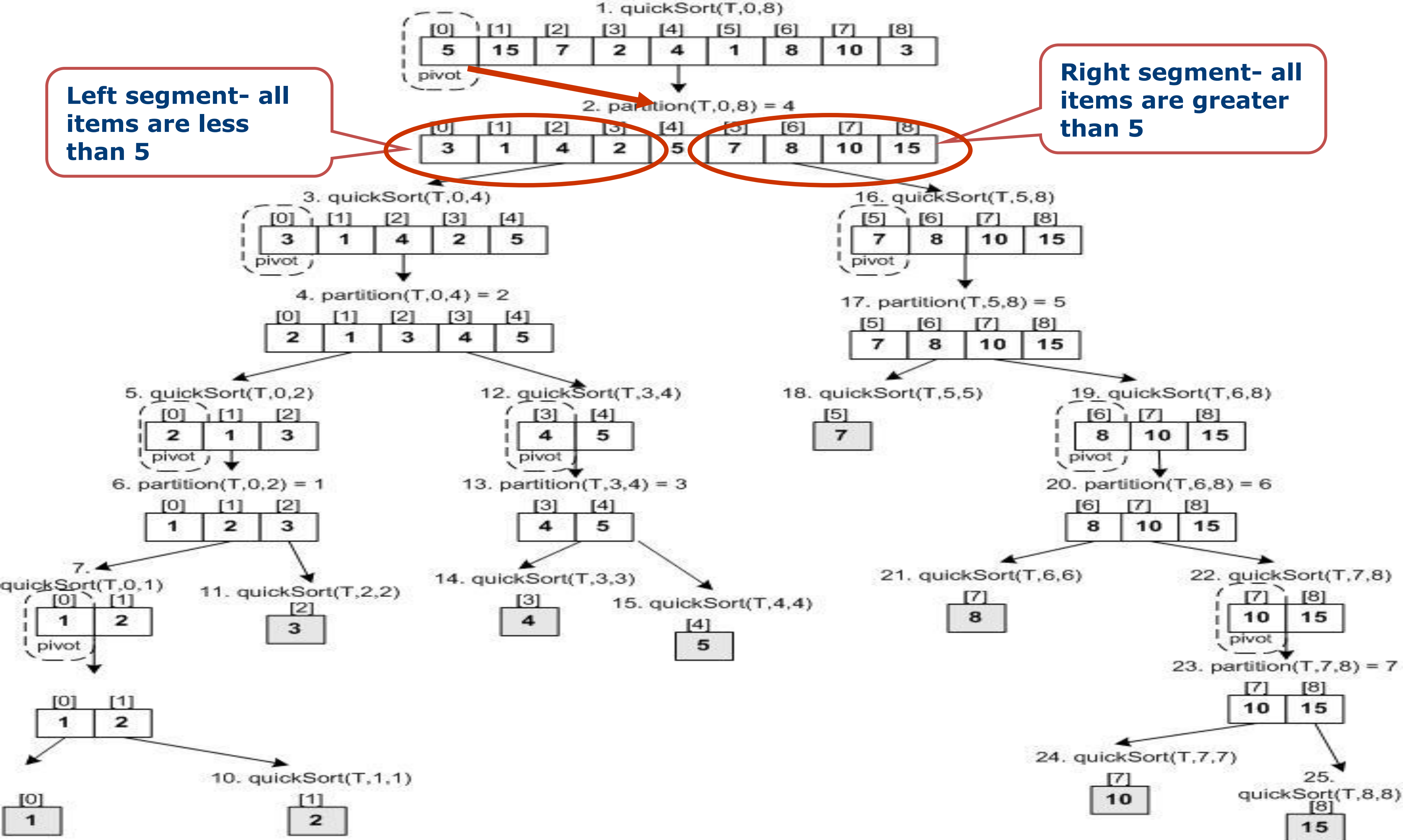
```
void quickSort (dataType arrayT[],
                  int first , int last)
{
     int cut;
     if (first<last){
        cut = partition(T, first,last);
        quickSort(T, first,cut);
        quickSort (T, cut+1, last);
     }
}
```

Recursive function that will partition the list into several sub lists until there is one item left in the sub list

Identify pivot or cutting point & **rearrange** the list based on **pivot value**

cut the list into 2 sub lists based on **cut value**

quickSort [5 15 7 2 4 1 8 10 3]



Left segment- all items are less than 5

Right segment- all items are greater than 5

Quick Sort Implementation

`partition()` function – **organize** the data so that the

- items with values **less than pivot or equal to pivot** will be on the **left** of the pivot,
- while the values at the **right** pivot contains items that are **greater**.

partition() function

```
int partition(int T[], int first,int last)
{
      int pivot, temp;
      int loop, cutPoint, bottom, top;
      pivot=T[first];
      bottom=first; top= last;
      loop=1;     //always TRUE
      while (loop) {
          while (T[top]>pivot){
           // find smaller value than
           // pivot from top array
                top--;
          }
          while(T[bottom]<pivot){
           //find larger value than
           //pivot from bottom
                 bottom++;
          }
```

Identify pivot

From top
Find value < pivot
& **skip** value > pivot

From bottom
Find value > pivot
& **skip** value < pivot

partition() function

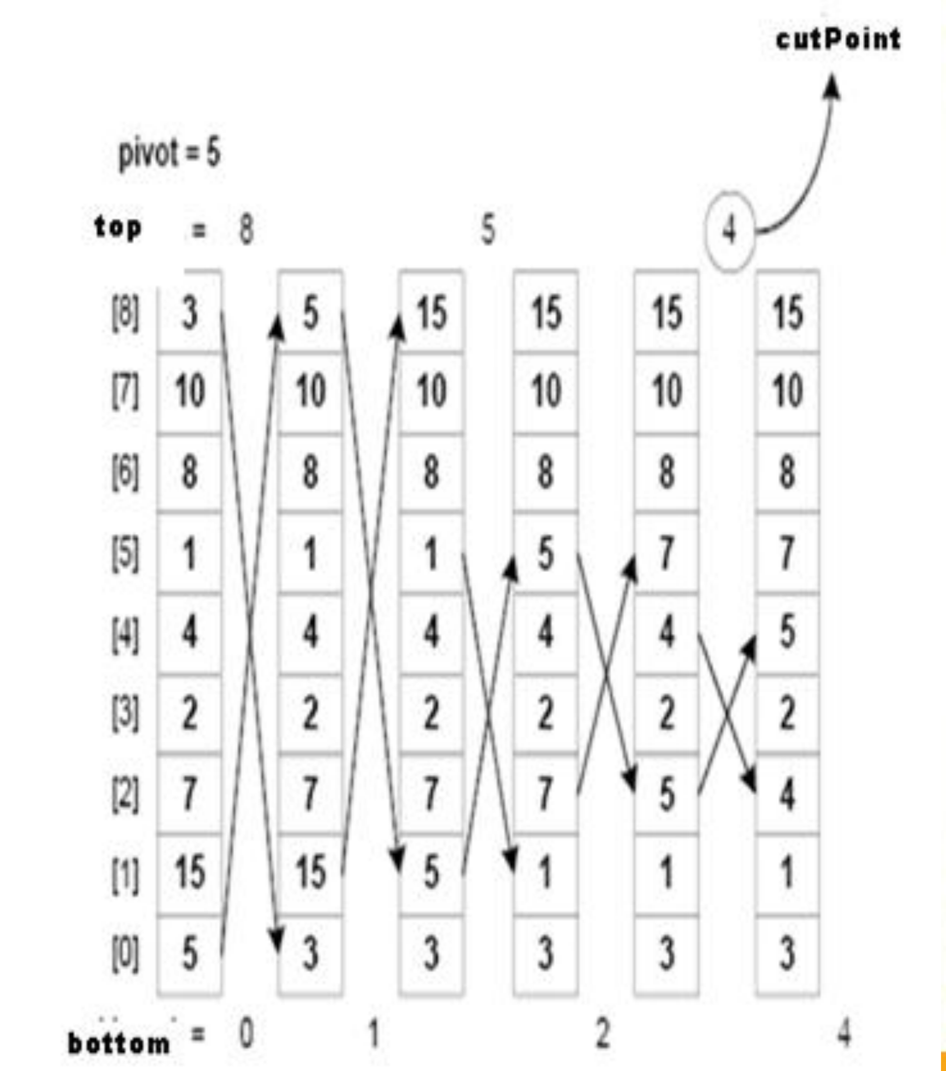
```
   if (bottom<top) {
         // change pivot place
         temp=T[bottom];
         T[bottom]=T[top];
         T[top]=temp;
   }
   else {
          loop=0; //loop false
          cutPoint = top;
   }//end if
  }// end while
  return cutPoint;
}//end function
```

Swap values disorder at top & bottom position

Stop loop

Return cut value

Partition process for array: [5 15 7 2 4 1 8 10 3]



cutPoint

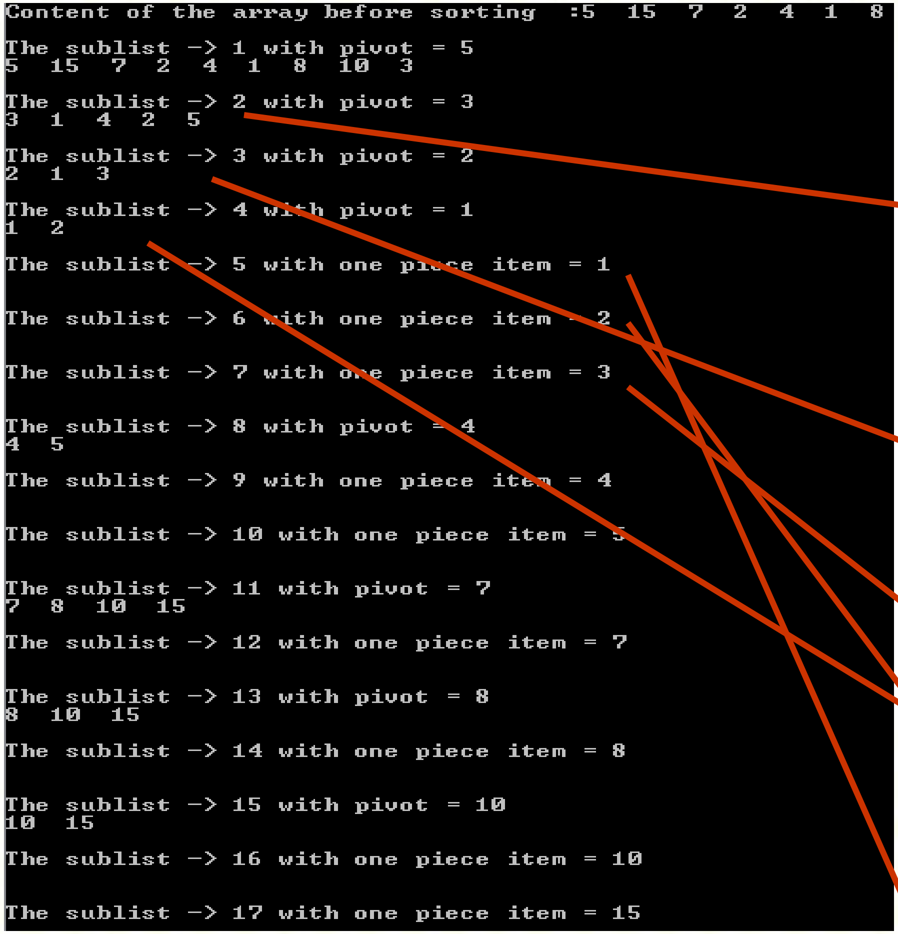
pivot = 5

top = 8 5 4

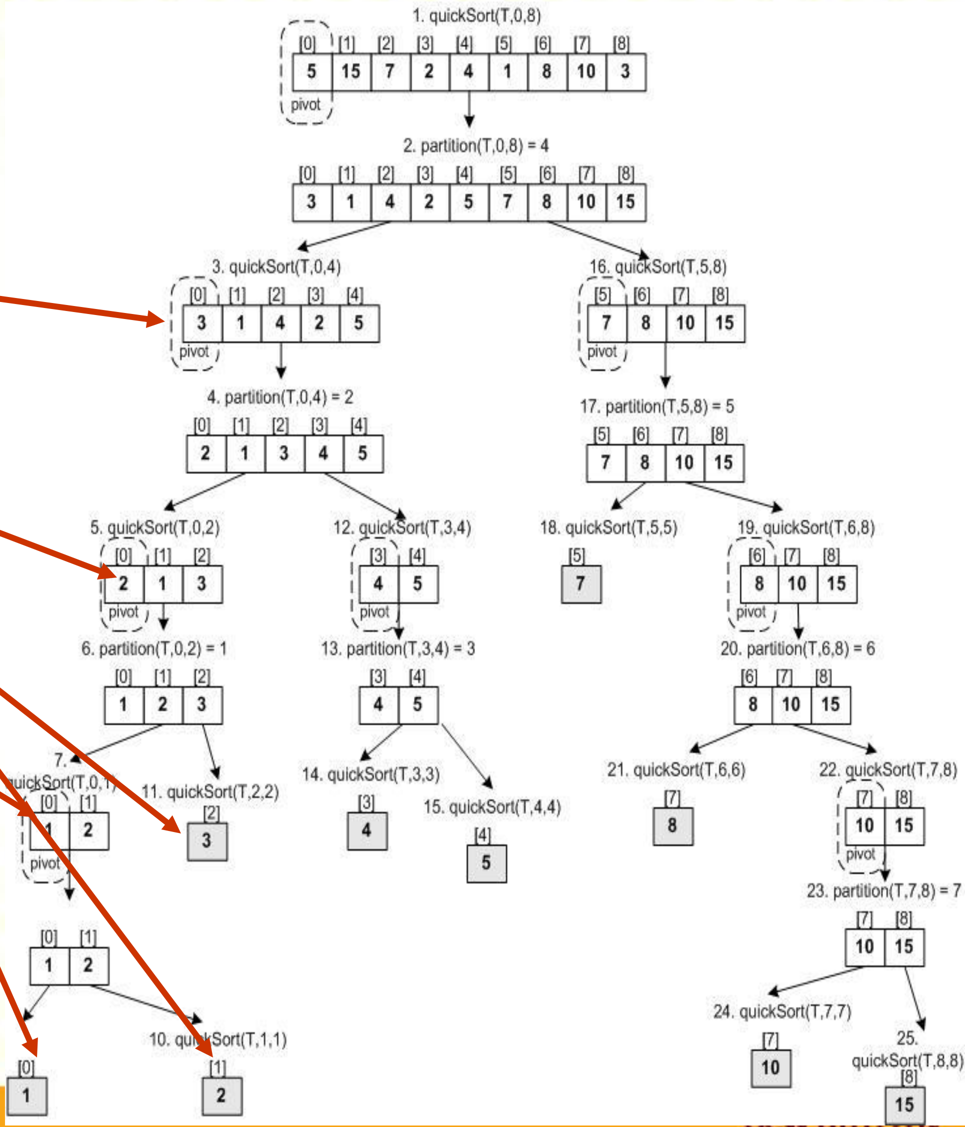
[8]	3	5	15	15	15	15
[7]	10	10	10	10	10	10
[6]	8	8	8	8	8	8
[5]	1	1	1	5	7	7
[4]	4	4	4	4	4	5
[3]	2	2	2	2	2	2
[2]	7	7	7	7	5	4
[1]	15	15	5	1	1	1
[0]	5	3	3	3	3	3

bottom = 0 1 2 4

quickSort[5 15 7 2 4 1 8 10 3]



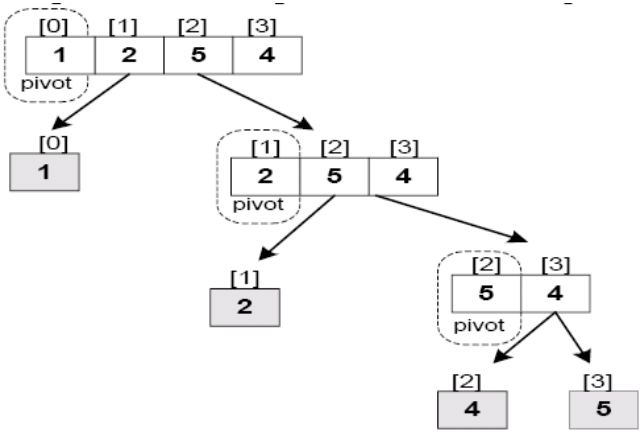
```
Content of the array before sorting  :5  15  7  2  4  1  8
The sublist -> 1 with pivot = 5
5  15  7  2  4  1  8  10  3
The sublist -> 2 with pivot = 3
3  1  4  2  5
The sublist -> 3 with pivot = 2
2  1  3
The sublist -> 4 with pivot = 1
1  2
The sublist -> 5 with one piece item = 1

The sublist -> 6 with one piece item = 2

The sublist -> 7 with one piece item = 3

The sublist -> 8 with pivot = 4
4  5
The sublist -> 9 with one piece item = 4

The sublist -> 10 with one piece item = 5

The sublist -> 11 with pivot = 7
7  8  10  15
The sublist -> 12 with one piece item = 7

The sublist -> 13 with pivot = 8
8  10  15
The sublist -> 14 with one piece item = 8

The sublist -> 15 with pivot = 10
10  15
The sublist -> 16 with one piece item = 10

The sublist -> 17 with one piece item = 15
```



Quick Sort Analysis

- The efficiency of quick sort depends on the **pivot value**.
- This class chose the first element in the array as pivot value.
- However, pivot can also be chosen at **random**, or **from the last element** in the array.
- The **worse case** for quick sort occur when the **smallest** item or the **largest** item always be chosen as **pivot** value causing the left partition and the right partition not balance.

Example of worse case quick sort: sorted array [1 2 5 4] causing imbalance partition.



Quick Sort Analysis

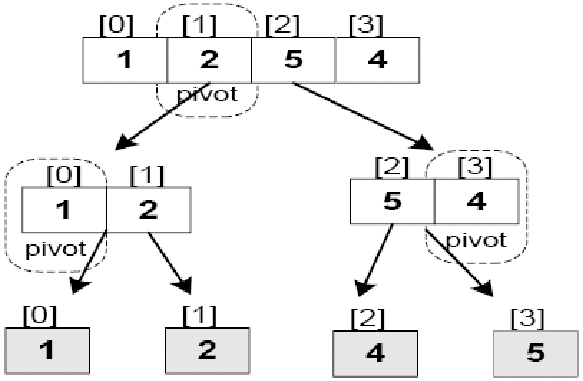
- The **best case** for quick sort happen when the list is partition into **balance segment**.
- Must chose the right pivot that can put other items in balance situation.
- The number of comparisons in partition process for base case situation is as follows:

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + 16\frac{n}{16} + \ldots\ldots\ x\frac{n}{x}$$

Quick Sort Analysis

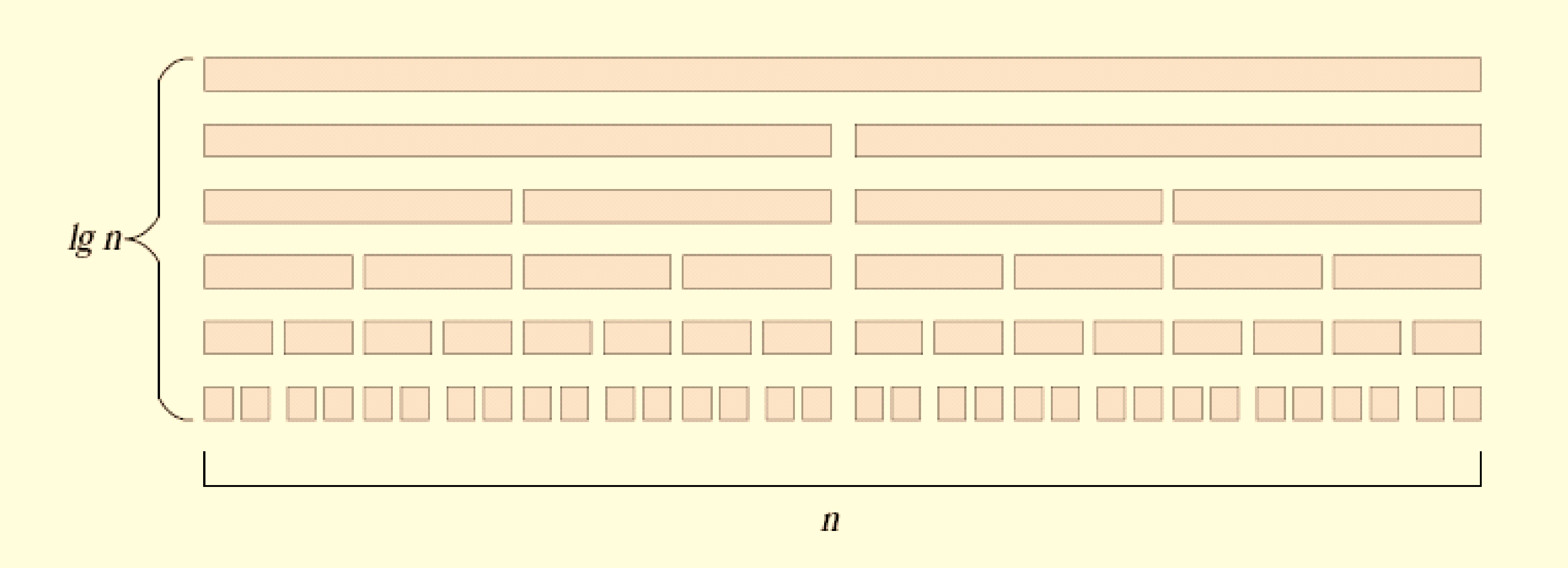
The **best case** for quick sort happen when the **left segment and the right segment is balanced** (have the same size) with value *x* ≈ lg *n* .

Example of best case quick sort: array[1 2 5 4].



Quick Sort Analysis

The number of steps to get the balance segment while partitioning the array is *lg n and the number of comparisons depend on the size list, n.*



$$: n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + 16\frac{n}{16} + \ldots\ldots\ x\frac{n}{x}$$

Quicksort

- Analysis
  - Average case: O(n * $\log_2 n$)
  - Worst case: O(n^2)
    - When the array is already sorted and the smallest item is chosen as the pivot
  - Quicksort is usually extremely fast in practice
  - Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

Summary

- Un-optimized selection sort, bubble sort, and insertion sort are all O(n^2) algorithms

- Quicksort and Mergesort are two very fast recursive sorting algorithms

References

1. Frank M. Carano, Janet J Prichard. “*Data Abstraction and problem solving with C++” Walls and Mirrors*. 5th edition (2007). Addision Wesley.
2. Nor Bahiah et al. *Struktur data & algoritma menggunakan C++. Penerbit UTM, 2005.*