Sorting Algorithms

Sorting – The Task

- **Given an array**

  `x[0], x[1], … , x[size-1]`

  reorder entries so that

  `x[0] <= x[1] <=  . . . <= x[size-1]`

  Here, List is in non-decreasing order.

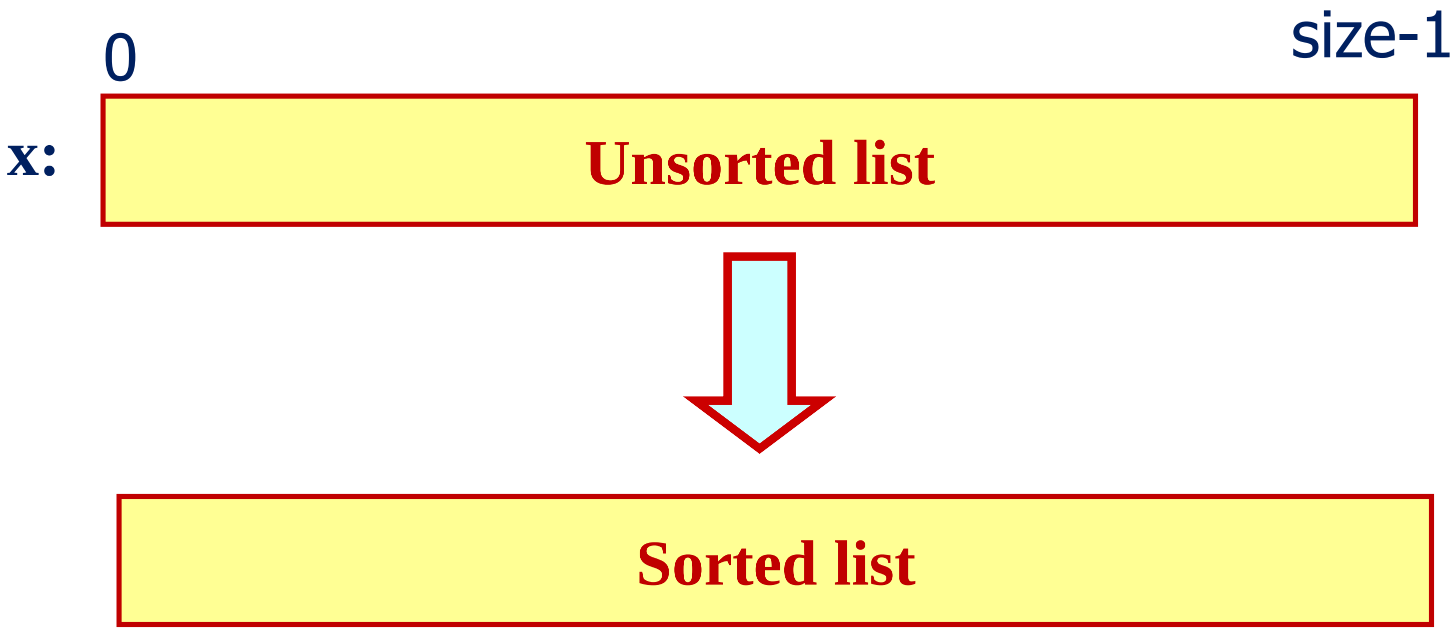
- **We can also sort a list of elements in non-increasing order.**

Sorting – Example

- **Original list:**
  - **10, 30, 20, 80, 70, 10, 60, 40, 70**
- **Sorted in non-decreasing order:**
  - **10, 10, 20, 30, 40, 60, 70, 70, 80**
- **Sorted in non-increasing order:**
  - **80, 70, 70, 60, 40, 30, 20, 10, 10**

Sorting Problem

- **What do we want :**
  - **- Data to be sorted in order**



Issues in Sorting

Many issues are there in sorting techniques

- How to rearrange a given set of data?
- Which data structures are more suitable to store data prior to their sorting?
- How fast the sorting can be achieved?
- How sorting can be done in a memory constraint situation?
- How to sort various types of data?

Sorting Algorithms

Sorting by Comparison

- Basic operation involved in this type of sorting technique is comparison. A data item is compared with other items in the list of items in order to find its place in the sorted list.
  - Insertion
  - Selection
  - Exchange
  - Enumeration

Sorting by Comparison

Sorting by comparison – Insertion:

- From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or to a different list.

e.g.: Insertion sort

Sorting by comparison – Selection:

- First the smallest (or largest) item is located and it is separated from the rest; then the next smallest (or next largest) is selected and so on until all item are separated.

e.g.: Selection sort, Heap sort

Sorting by Comparison

Sorting by comparison – Exchange:

- If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.

e.g.: Bubble sort, Shell Sort, Quick Sort

Sorting by comparison – Enumeration:

- Two or more input lists are merged into an output list and while merging the items, an input list is chosen following the required sorting order.

e.g.: Merge sort

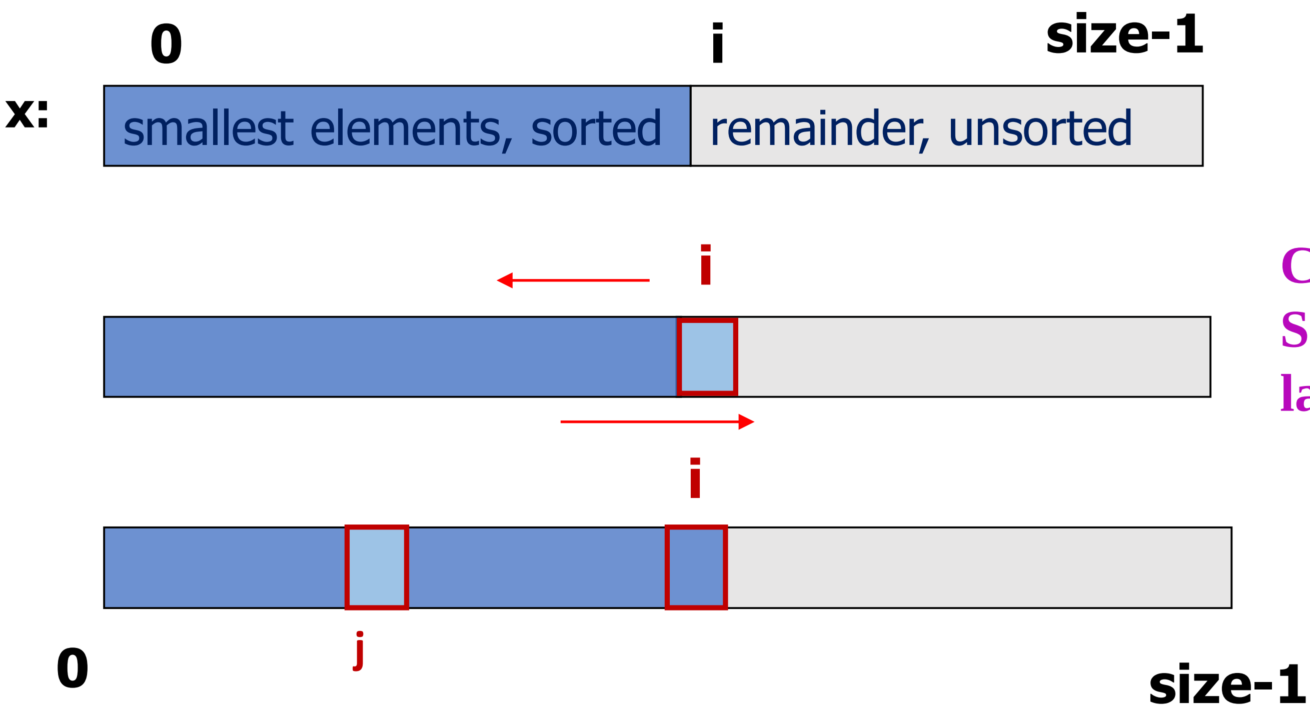
Sorting by Distribution

- No key comparison takes place
- All items under sorting are distributed over an auxiliary storage space based on the constituent element in each and then grouped them together to get the sorted list.
- Distributions of items based on the following choices
  - ✓ **Radix** - An item is placed in a space decided by the bases (or radix) of its components with which it is composed of.
  - ✓ **Counting -** Items are sorted based on their relative counts.
  - ✓ **Hashing** - Items are hashed, that is, dispersed into a list based on a hash function.

Insertion Sort

Insertion Sort

General situation :



Compare and Shift till x[i] is larger.

Insertion Sort

```
void insertionSort (int list[], int size)
{
        int i,j,item;

        for (i=1; i<size; i++)
        {
                item = list[i] ;
/* Move elements of list[0..i-1], that are greater than
item, to one position ahead of their current position */

                for (j=i-1; (j>=0)&& (list[j] > item); j--)
                        list[j+1] = list[j];
                list[j+1] = item ;
        }
}
```

Insertion Sort

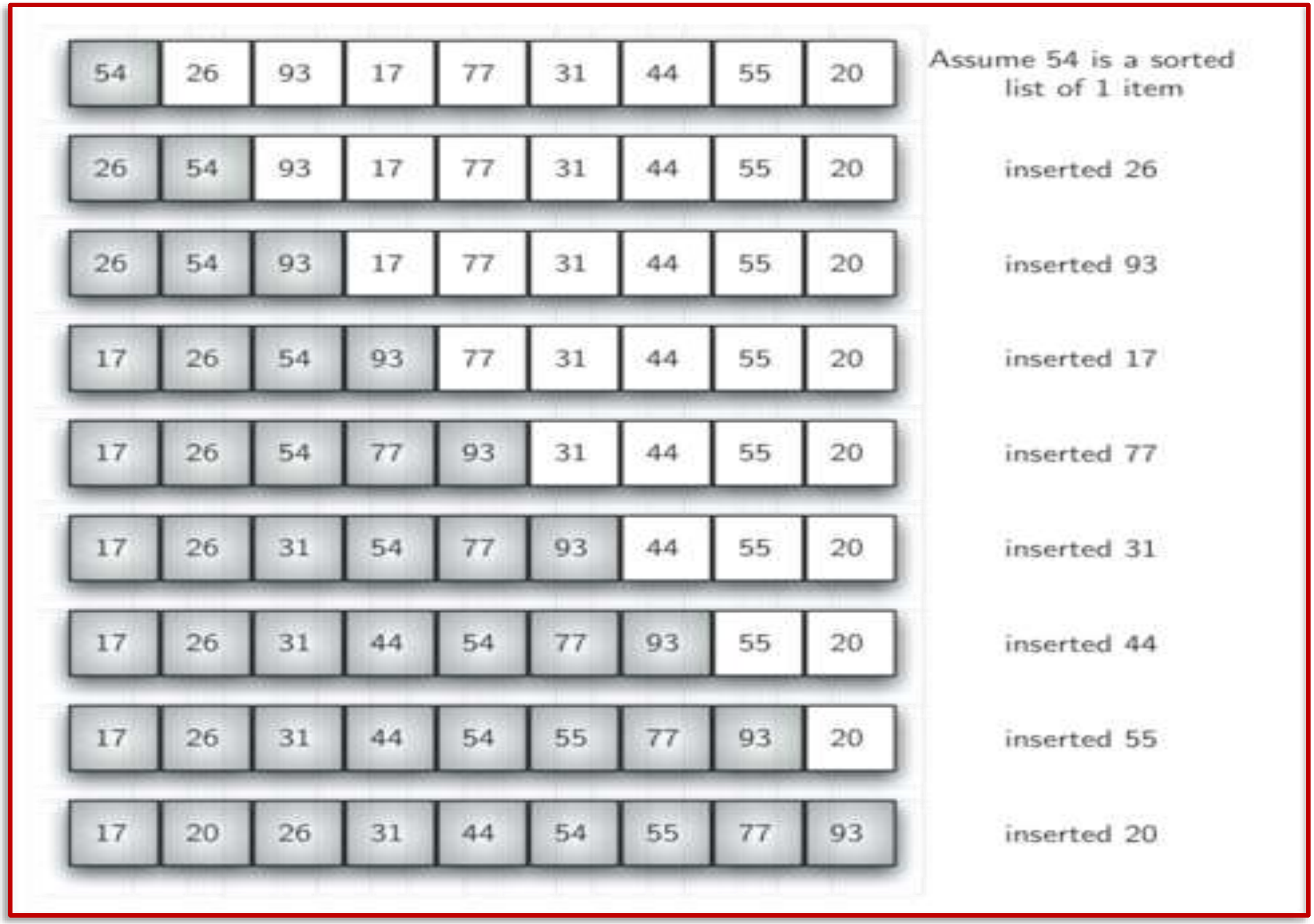
```
int main()
{
    int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};

    int i;
    for(i=0;i<12;i++)
       printf("%d ",x[i]);
    printf("\n");

    insertionSort(x,12);

    for(i=0;i<12;i++)
       printf("%d ",x[i]);
    printf("\n");
}
```

OUTPUT

-45 89 -65 87 0 3 -23 19 56 21 76 -50

-65 -50 -45 -23 0 3 19 21 56 76 87 89

Insertion Sort - Example



Insertion Sort: Complexity Analysis

Case 1: If the input list is already in sorted order

Number of comparisons: Number of comparison in each iteration is 1.

$$C(n) = 1 + 1 + 1 + \cdots + 1 \quad \text{upto (n-1)}^{\text{th}} \text{ iteration.}$$

Number of movement: No data movement takes place in any iteration.

$$M(n) = 0$$

Insertion Sort: Complexity analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons: Number of comparison in each iteration is 1.

$$C(n) = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Number of movement: Number of movements takes place in any i^{th} iteration is i.

$$M(n) = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Insertion Sort: Complexity analysis

Case 3: If the input list is in random order

- Let p_j be the probability that the key will go to the j^{th} location $(1 \leq j \leq i+1)$. Then the number of comparisons will be $j \cdot p_j$.

- The average number of comparisons in the $(\mathrm{i}+1)^{th}$ iteration is

$$A_{i+1} = \sum_{j=1}^{i+1} j \cdot p_j$$

- Assume that all keys are distinct and all permutations of keys are equally likely.

$$p_1 = p_2 = p_3 = \cdots = p_{i+1} = \frac{1}{i+1}$$

Insertion Sort: Complexity analysis

Case 3: Number of comparisons

- Therefore, the average number of comparisons in the (i + $1)^{th}$ iteration

$$A_{i+1} = \frac{1}{i+1}\sum_{j=1}^{i+1} j = \frac{1}{i+1} \cdot \frac{(i+1)\cdot(i+2)}{2} = \frac{i+2}{2}$$

- Total number of comparisons for all $(n-1)$ iterations is

$$C(n) = \sum_{i=0}^{n-1} A_{i+1} = \frac{1}{2} \cdot \frac{n(n-1)}{2} + (n-1)$$

Insertion Sort: Complexity analysis

Case 3: Number of Movements

- On the average, number of movements in the i^{th} iteration

$$M_i = \frac{i + (i-1) + (i-2) + \cdots + 2 + 1}{i} = \frac{i+1}{2}$$

- Total number of movements

$$M(n) = \sum_{i=1}^{n-1} M_i = \frac{1}{2} \cdot \frac{n(n-1)}{2} + \frac{n-1}{2}$$

Insertion Sort: Summary of Complexity Analysis

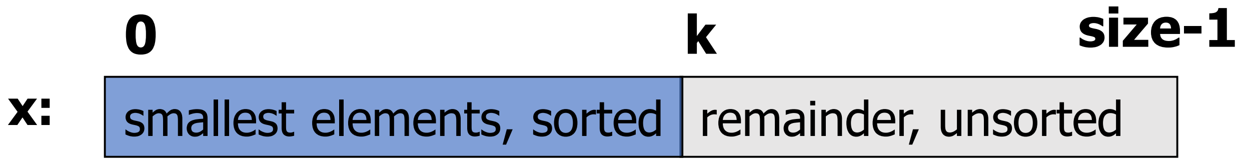
Case	Comparisons	Movement	Memory	Remarks
Case 1	$\mathrm{C}(n) = (n-1)$	$M(n) = 0$	$S(n) = n$	Input list is in sorted order
Case 2	$\mathrm{C}(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{2}$	$S(n) = n$	Input list is sorted in reverse order
Case 3	$\mathrm{C}(n) = \frac{(n-1)(n+4)}{4}$	$M(n) = \frac{(n-1)(n+2)}{4}$	$S(n) = n$	Input list is in random order

Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\,(n-1)$	$T(n) = O(n)$	Best case
Case 2	$T(n) = c\,n(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = c\frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Average case

Selection Sort

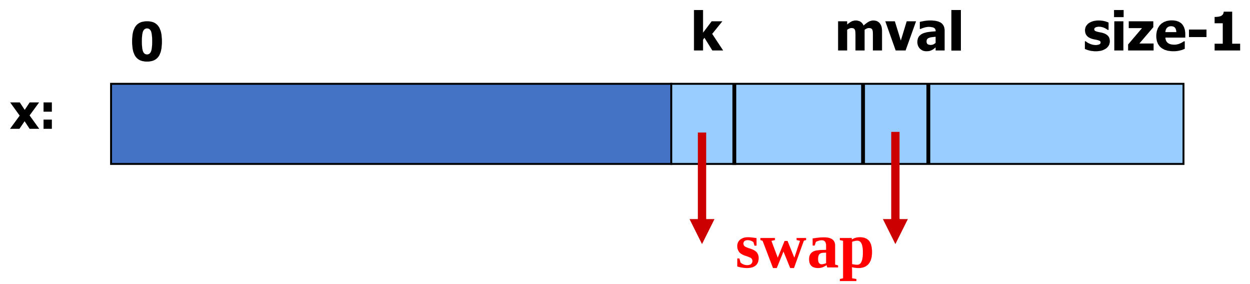
Selection Sort

General situation :



Steps :

- **Find smallest element, mval, in x[k…size-1]**
- **Swap smallest element with x[k], then increase k.**



Selection Sort

```
/* Yield location of smallest element in
x[k .. size-1];*/

int findMinLloc (int x[ ], int k, int size)
{
    int j, pos;        /* x[pos] is the smallest
element found so far */
      pos = k;
      for (j=k+1; j<size; j++)
            if (x[j] < x[pos])
                  pos = j;
      return pos;
}
```

Selection Sort

```
/* The main sorting function */
/* Sort x[0..size-1] in non-decreasing order */

int selectionSort (int x[], int size)
{   int k, m;
        for (k=0; k<size-1; k++)
      {
                m = findMinLoc(x, k, size);
                temp = a[k];
                a[k] = a[m];
                a[m] = temp;
        }
}
```

Selection Sort - Example

x:

3	12	-5	6	142	21	-17	45

x:

-17	12	-5	6	142	21	3	45

x:

-17	-5	12	6	142	21	3	45

x:

-17	-5	3	6	142	21	12	45

x:

-17	-5	3	6	142	21	12	45

x:

-17	-5	3	6	12	21	142	45

x:

-17	-5	3	6	12	21	142	45

x:

-17	-5	3	6	12	21	45	142

x:

-17	-5	3	6	12	21	45	142

Selection Sort: Complexity Analysis

Case 1: If the input list is already in sorted order

Number of comparisons:

$$C(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$$

Number of movement: no data movement takes place in any iteration.

$$M(n) = 0$$

Selection Sort: Complexity Analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons:

$$c(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$$

Number of movements:

$$M(n) = \frac{3}{2}(n - 1)$$

Selection Sort: Complexity Analysis

Case 3: If the input list is in random order

Number of comparisons:

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

- Let p_i be the probability that the i^{th} smallest element is in the i^{th} position. Number of total swap operations $= (1 - p_i) \times (n-1)$

where $p_1 = p_2 = p_3 = \cdots = p_n = \frac{1}{n}$

- **Total number of movements**

$$M(n) = \left(1 - \frac{1}{n}\right) \times (n-1) \times 3 = \frac{3(n-1)(n-1)}{n}$$

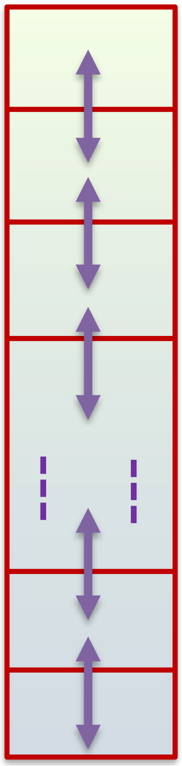
Selection Sort: Summary of Complexity analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$c(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 0$	Input list is in sorted order
Case 2	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{3(n-1)}{2}$	$S(n) = 0$	Input list is sorted in reverse order
Case 3	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{3(n-1)^2}{n}$	$S(n) = 0$	Input list is in random order

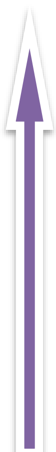
Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = \frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best case
Case 2	$T(n) = \frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) \approx \frac{(n-1)(2n+3)}{2}$ (Taking $n-1 \approx n$)	$T(n) = O(n^2)$	Average case

Bubble Sort

Bubble Sort



In every iteration heaviest element drops at the bottom.



The bottom moves upward.

The sorting process proceeds in several passes.

- In every pass we go on comparing neighbouring pairs, and swap them if out of order.
- In every pass, the largest of the elements under considering will *bubble* to the top (i.e., the right).

Bubble Sort

How the passes proceed?

- **In pass 1, we consider index 0 to n-1.**
- **In pass 2, we consider index 0 to n-2.**
- **In pass 3, we consider index 0 to n-3.**
- **……**
- **……**
- **In pass n-1, we consider index 0 to 1.**

Bubble Sort - Example

Pass: 1

x:	3	12	-5	6	72	21	-7	45

x:	3	12	-5	6	72	21	-7	45

x:	3	-5	12	6	72	21	-7	45

x:	3	-5	6	12	72	21	-7	45

x:	3	-5	6	12	72	21	-7	45

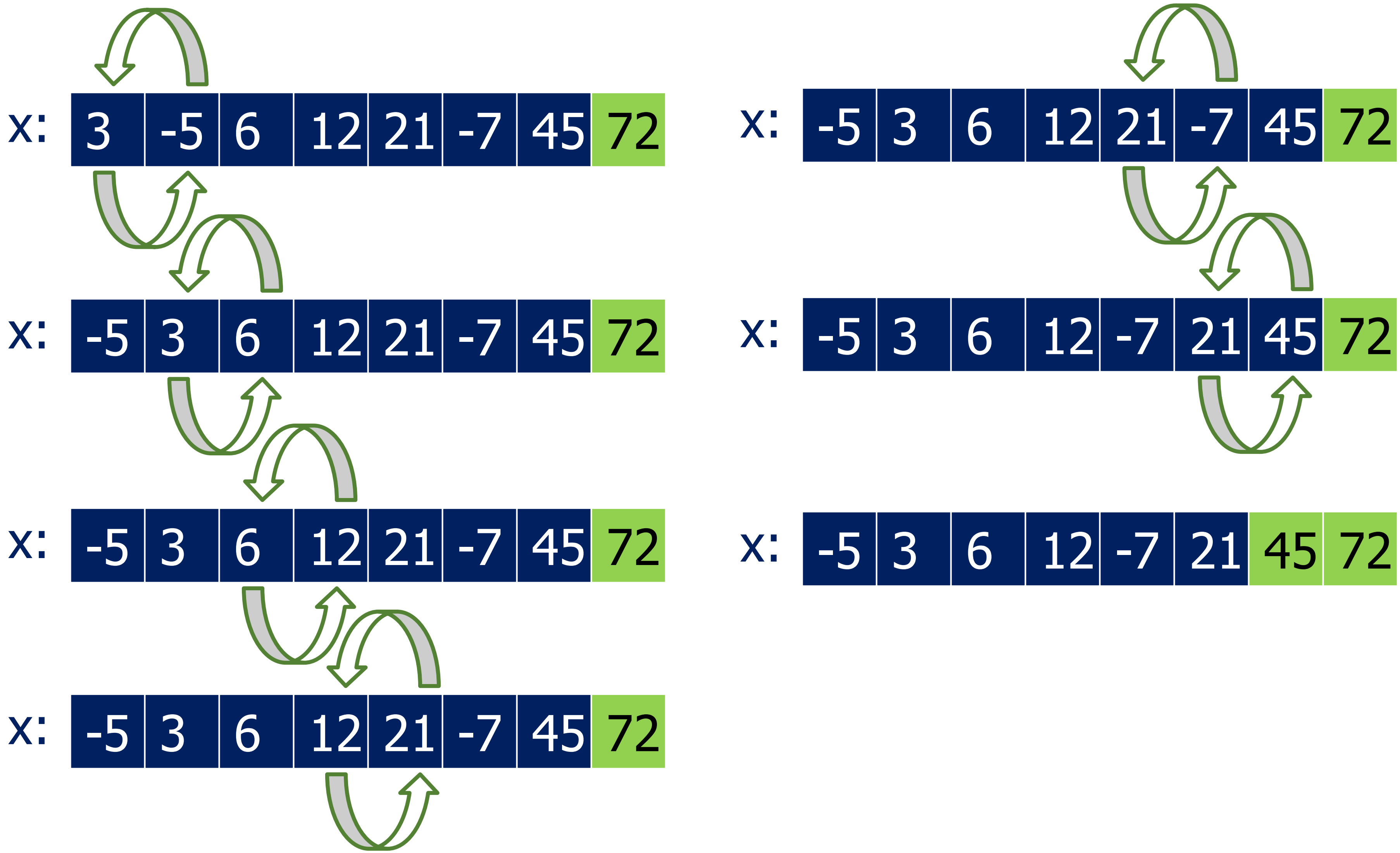
x:	3	-5	6	12	21	72	-7	45

x:	3	-5	6	12	21	-7	72	45

x:	3	-5	6	12	21	-7	45	72

Bubble Sort - Example

Pass: 2



Bubble Sort

```
void swap(int *x, int *y)
{
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void bubble_sort(int x[], int n)
{
      int i,j;
      for (i=n-1; i>0; i--)
         for (j=0; j<i; j++)
            if (x[j] > x[j+1])
               swap(&x[j],&x[j+1]);
}
```

Bubble Sort

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
     printf("%d ",x[i]);
  printf("\n");
  bubble_sort(x,12);
  for(i=0;i<12;i++)
     printf("%d ",x[i]);
  printf("\n");
}
```

OUTPUT

-45 89 -65 87 0 3 -23 19 56 21 76 -50

-65 -50 -45 -23 0 3 19 21 56 76 87 89

Bubble Sort: Complexity analysis

Case 1: If the input list is already in sorted order

Number of comparisons:

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

Number of movements:

$$M(n) = 0$$

Bubble Sort: Complexity analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons:

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

Number of movements:

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

Bubble Sort: Complexity analysis

Case 3: If the input list is in random order

Number of comparisons:

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

Number of movements:

- Let p_j be the probability that the largest element is in the unsorted part is in j^{th} $(1 \leq j \leq n - i + 1)$ location.
- The average number of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} (\overline{n - i + 1} - j) \cdot p_j$$

Bubble Sort: Complexity analysis

Case 3: If the input list is in random order

Number of movements:

- $p_1 = p_2 = p_{n-i+1} = \frac{1}{n-i+1}$
- Therefore, the average number of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} \frac{1}{n-i+1} \cdot (\overline{n-i+1} - j) = \frac{n-i}{2}$$

- The average number of movements

$$M(n) = \sum_{i=1}^{n-1} \frac{n-i}{2} = \frac{n(n-1)}{4}$$

Bubble Sort: Summary of Complexity analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$c(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 0$	Input list is in sorted order
Case 2	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{2}$	$S(n) = 0$	Input list is sorted in reverse order
Case 3	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{4}$	$S(n) = 0$	Input list is in random order

Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best case
Case 2	$T(n) = cn(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = \frac{3}{4}n(n-1)$	$T(n) = O(n^2)$	Average case

Bubble Sort

How do you make best case with (n-1) comparisons only?

- By maintaining a variable flag, to check if there has been any swaps in a given pass.
- If not, the array is already sorted.

Bubble Sort

```
void bubble_sort(int x[], int n)
{
  int i,j;
  int flag = 0;
   for (i=n-1; i>0; i--)
   {
     for (j=0; j<i; j++)
     if (x[j] > x[j+1])
     {
       swap(&x[j],&x[j+1]);
       flag = 1;
     }
     if (flag == 0) return;
   }
}
```

What is Divide and Conquer?

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots$
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for $\boldsymbol{S}_1, \boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Efficient Sorting algorithms

Two of the most popular sorting algorithms are based on divide-and-conquer approach.

- Quick sort
- Merge sort

Basic concept of divide-and-conquer method:

```
sort (list)
{
    if the list has length greater than 1
    {
        Partition the list into lowlist and highlist;
        sort (lowlist);
        sort (highlist);
        combine (lowlist, highlist);
    }
}
```

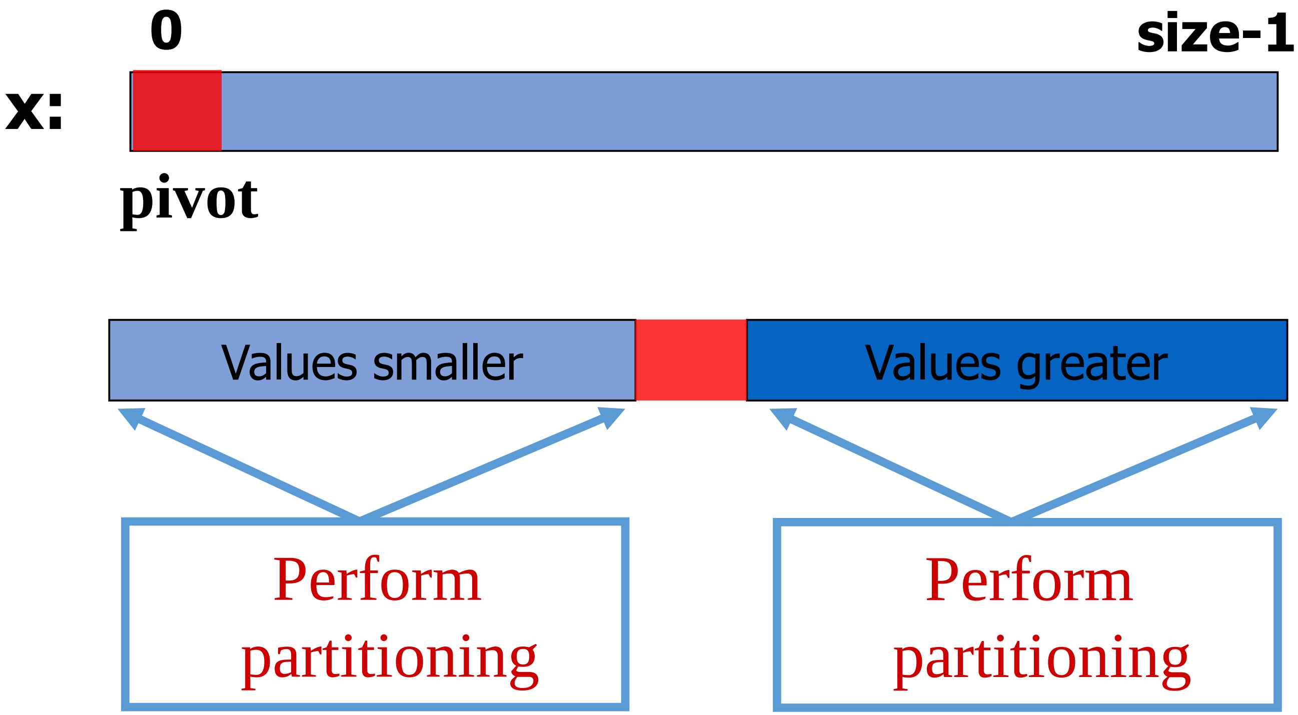
Quick Sort

Quick Sort – How it Works?

At every step, we select a *pivot element* in the list (usually the first element).

- We put the pivot element in the *final position* of the sorted list.
- All the elements less than or equal to the pivot element are to the left.
- All the elements greater than the pivot element are to the right.

Quick Sort Partitioning



Quick Sort

```
#include <stdio.h>
void quickSort( int[], int, int);
int partition( int[], int, int);
void main()
{
   int i,a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};
   printf("\n\nUnsorted array is: ");
   for(i = 0; i < 9; ++i)
      printf(" %d ", a[i]);
   quickSort( a, 0, 8);
   printf("\n\nSorted array is: ");
   for(i = 0; i < 9; ++i)
      printf(" %d ", a[i]);
}
void quickSort( int a[], int l, int r)
{
   int j;
   if( l < r ) {  // divide and conquer
   j = partition( a, l, r);
   quickSort( a, l, j-1);
   quickSort( a, j+1, r);
   }
}
```

Quick Sort

```
int partition( int a[], int l, int r)
{
   int pivot, i, j, t;
   pivot = a[l];
   i = l;
   j = r+1;
   while( 1) {
          do {
              ++i;
             } while(a[i]<=pivot && i<=r);
          do {
              --j;
             } while( a[j] > pivot );
          if( i >= j ) break;
          t = a[i];
          a[i] = a[j];
          a[j] = t;
   }
   t = a[l];
   a[l] = a[j];
   a[j] = t;
   return j;
}
```

Example

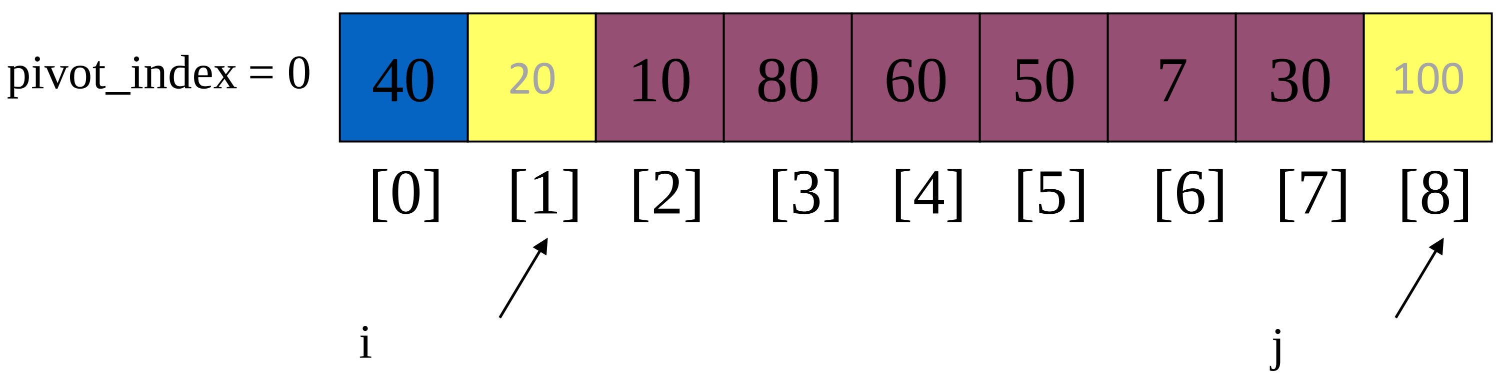
We are given array of n integers to sort:

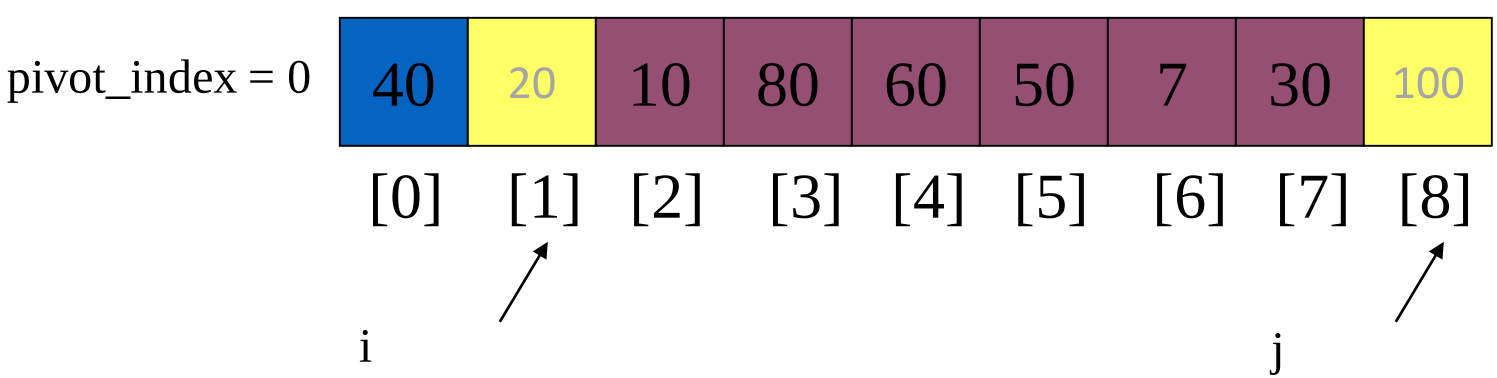
40	20	10	80	60	50	7	30	100

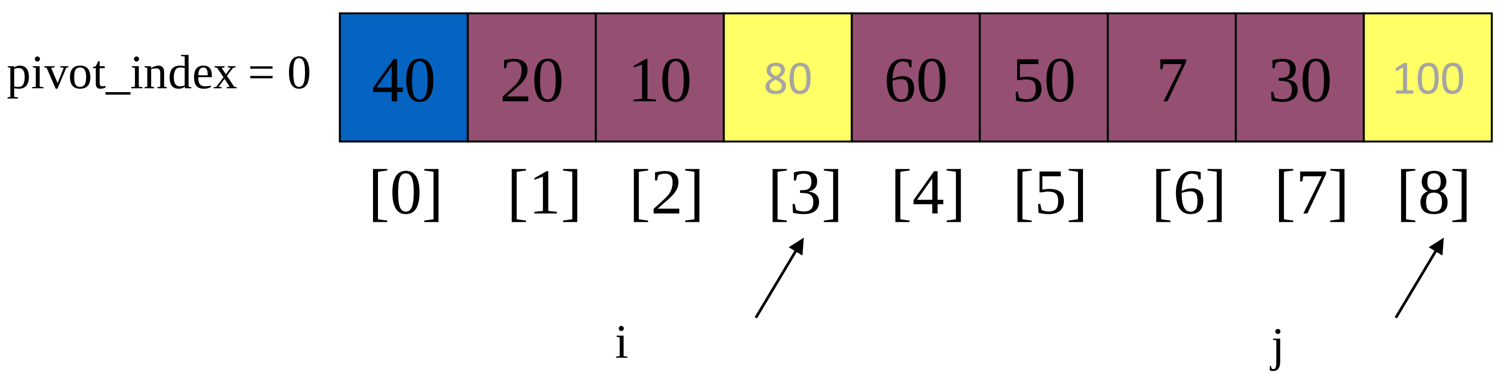
Pick Pivot Element

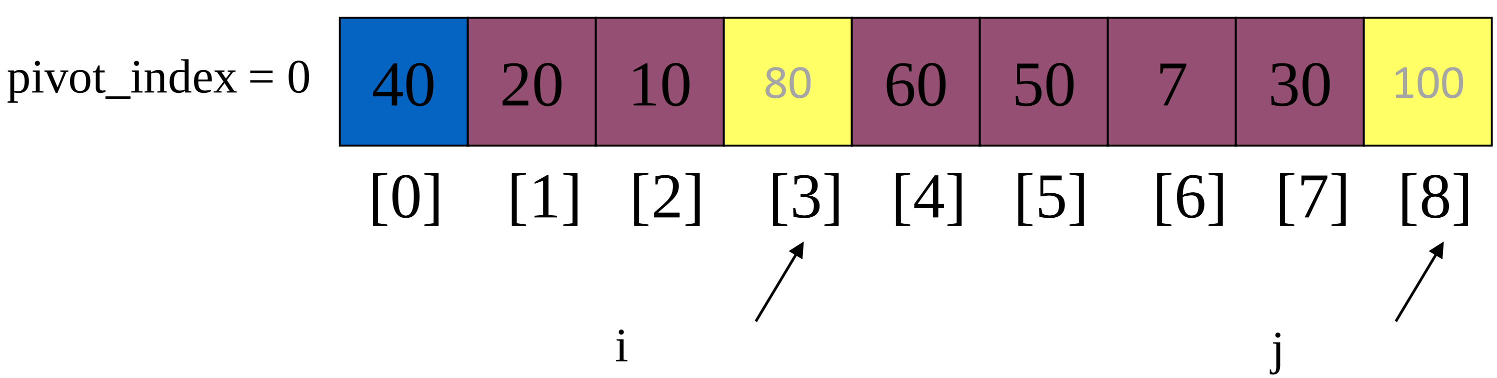
There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

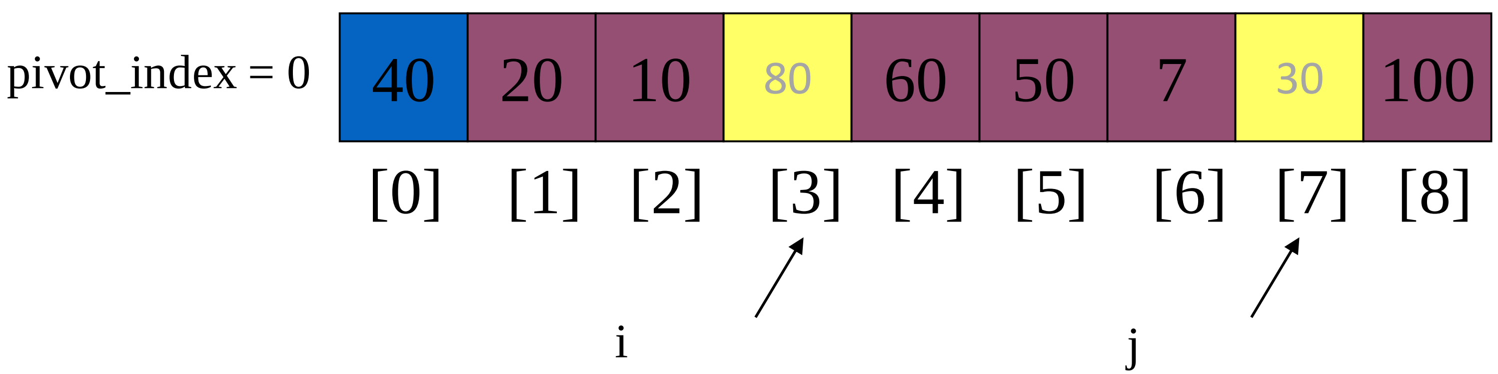

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

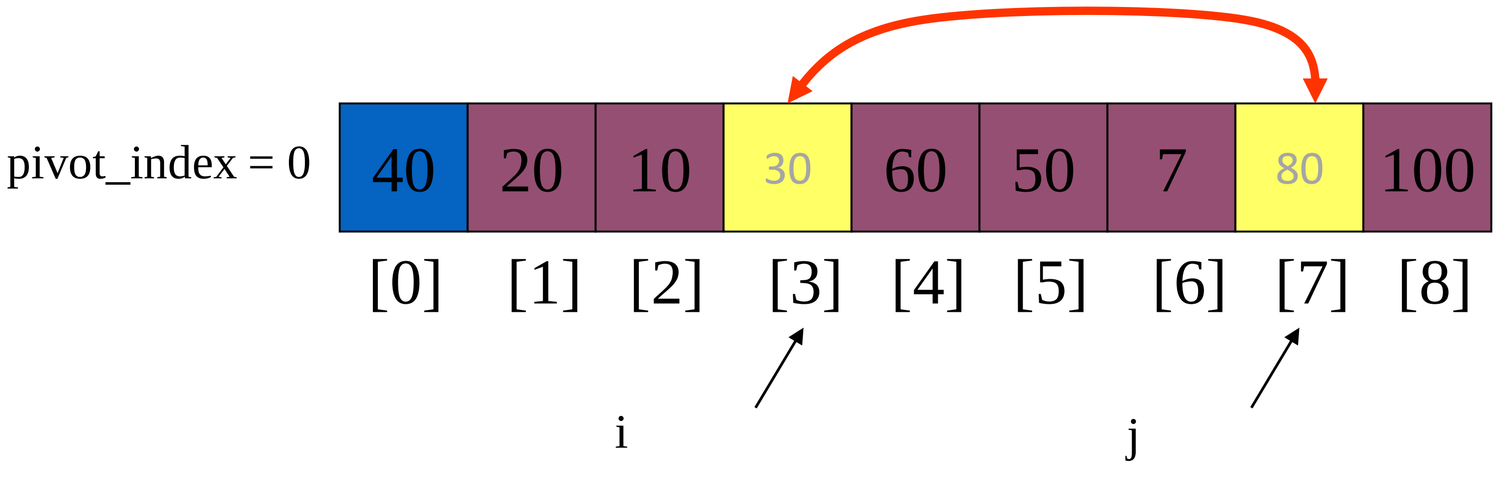

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

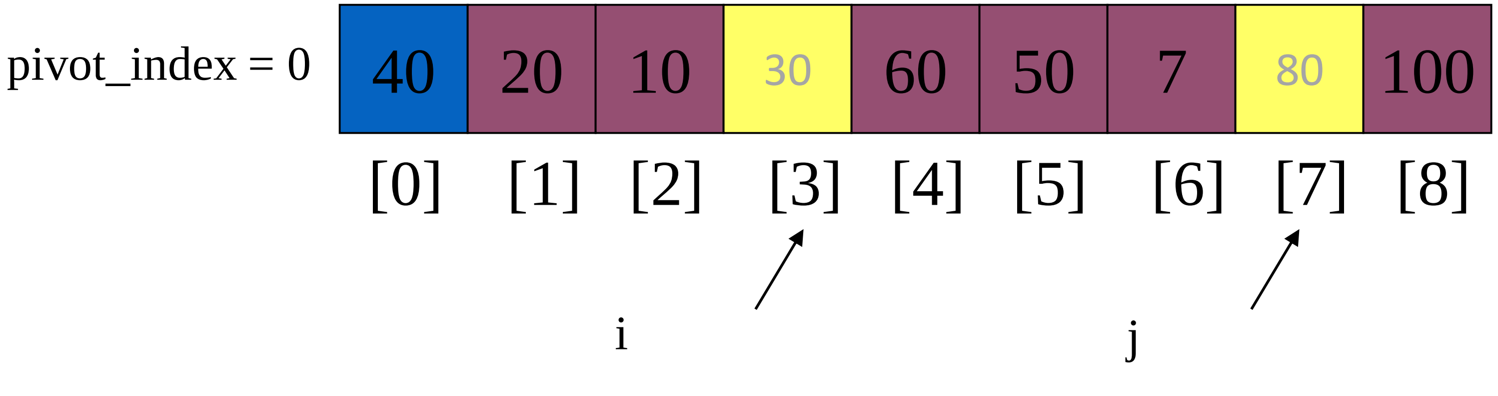

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

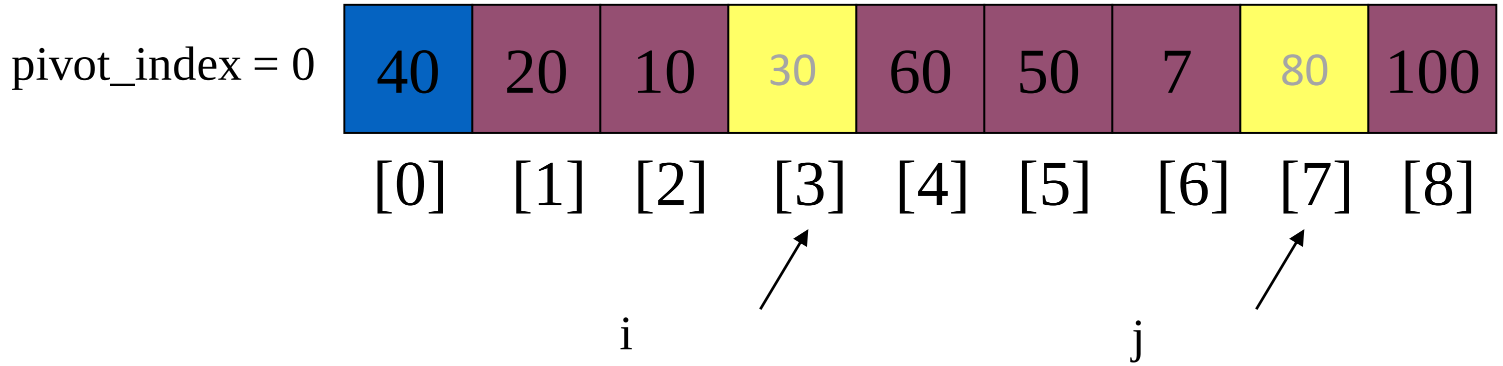


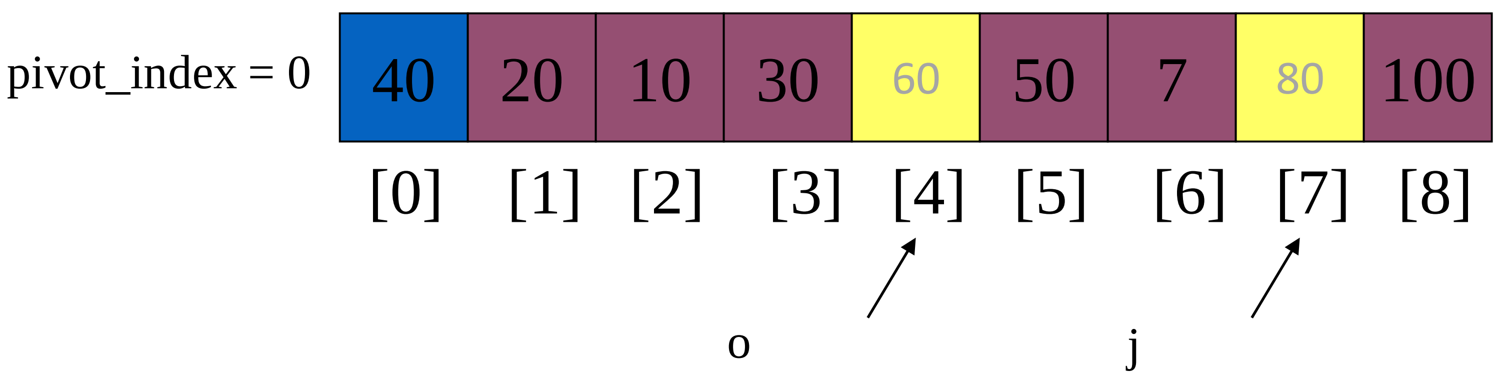
pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

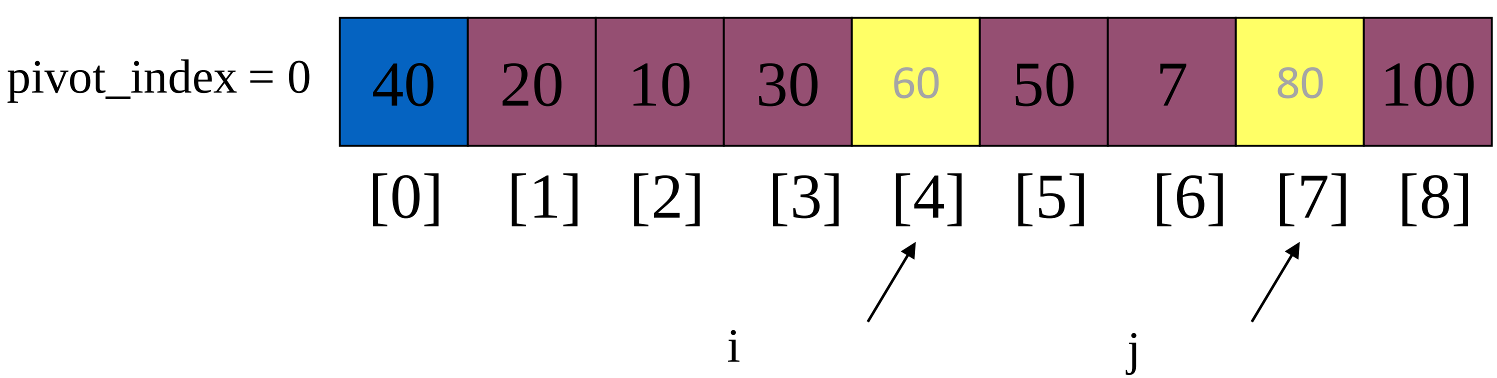

pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

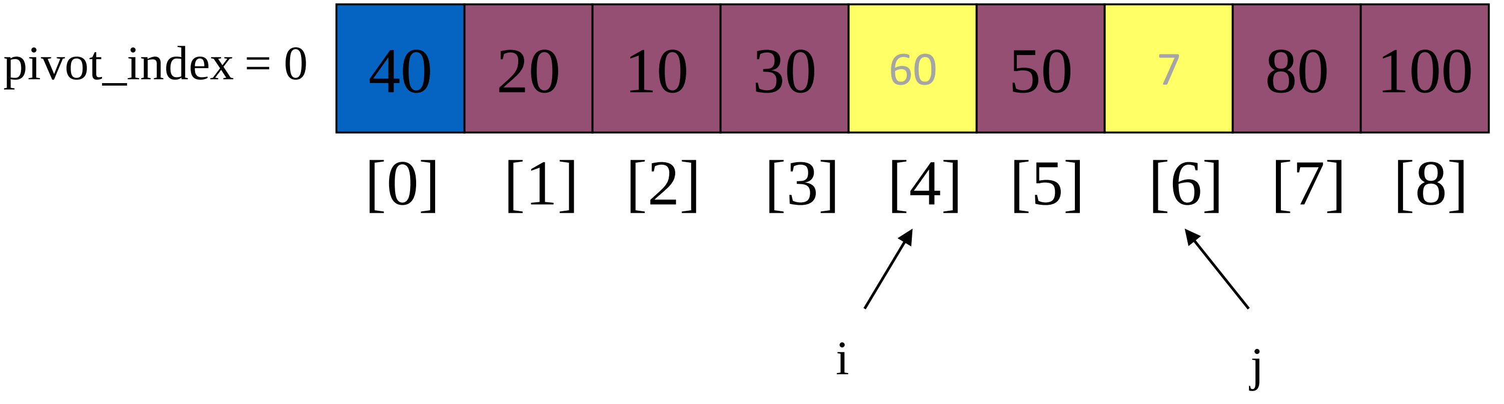

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j


pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

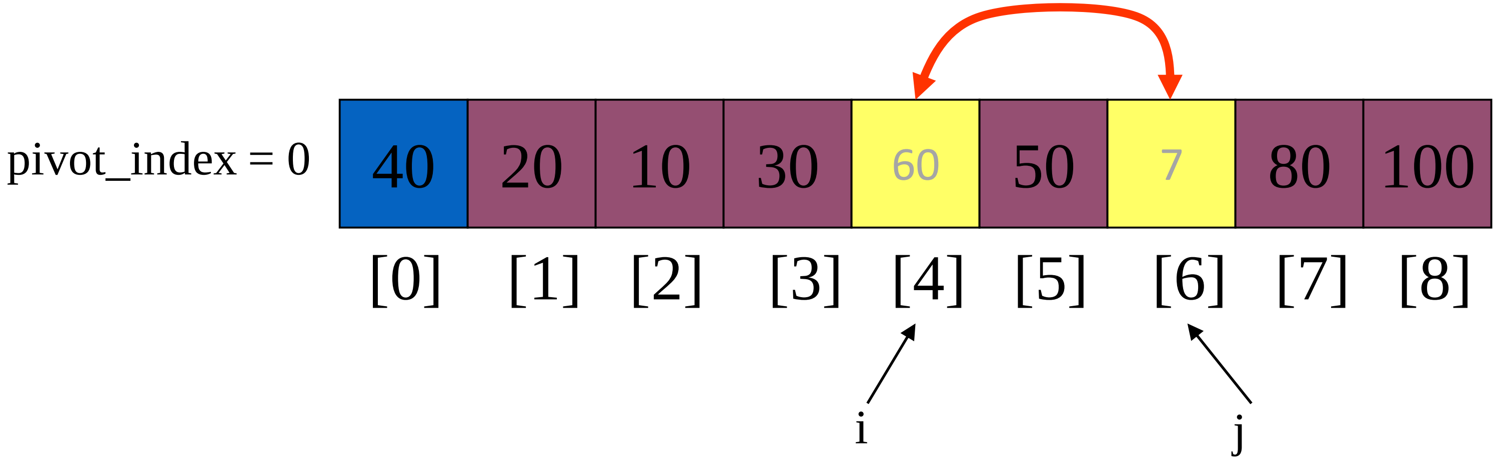

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

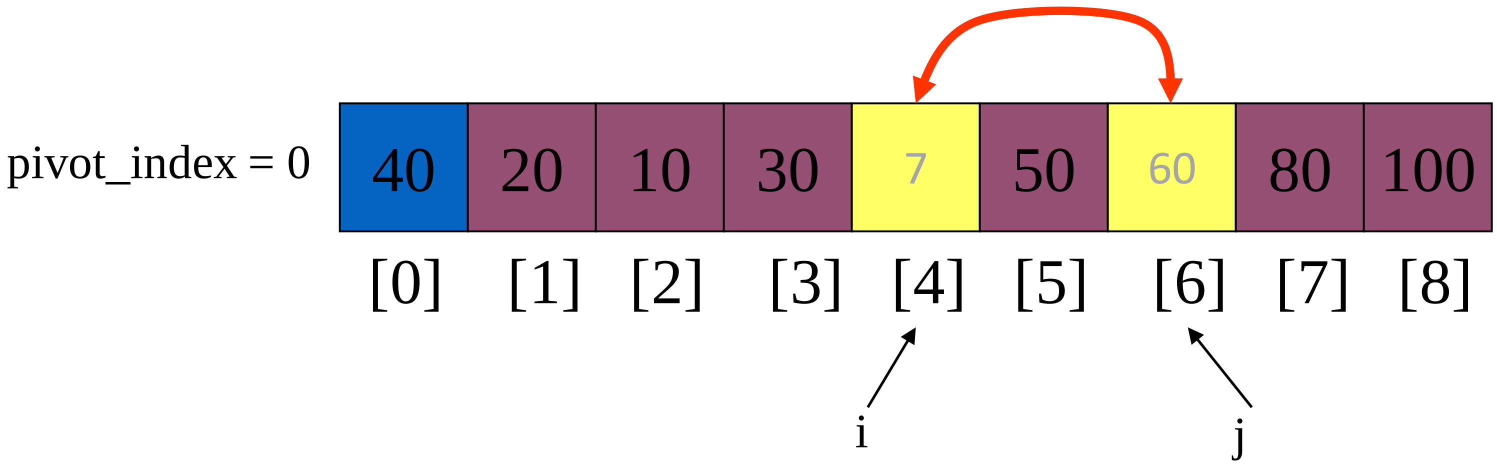

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
o
j

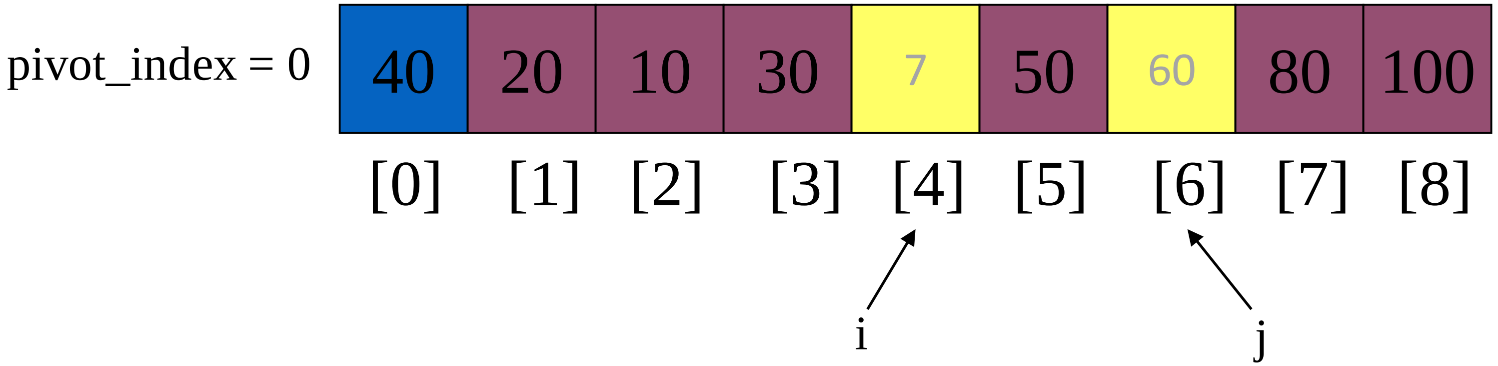

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

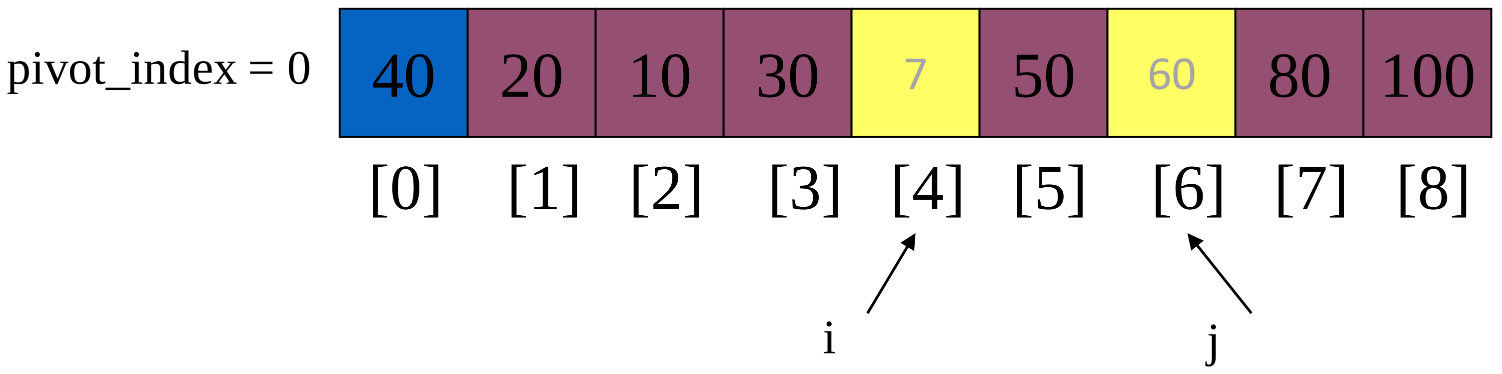


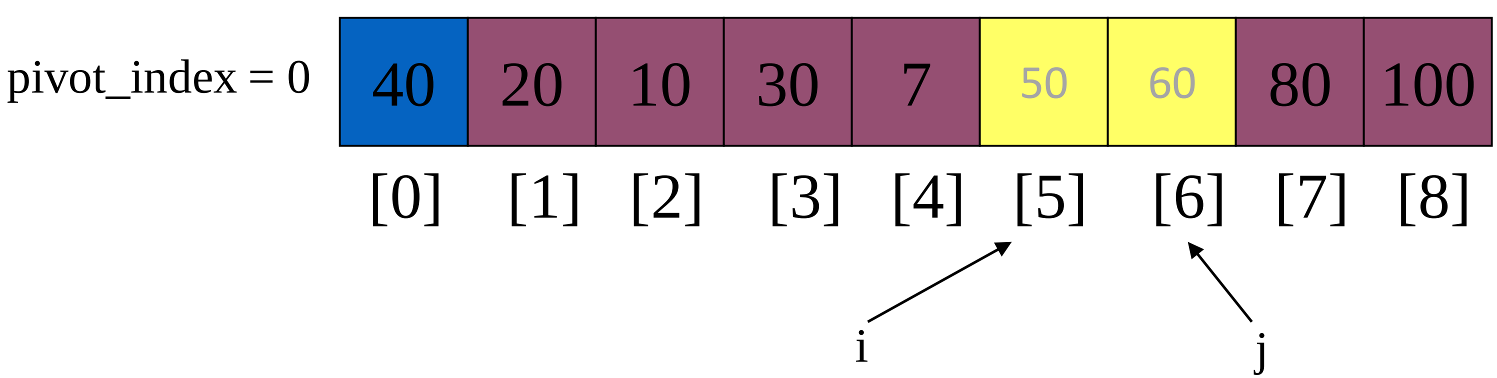
pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

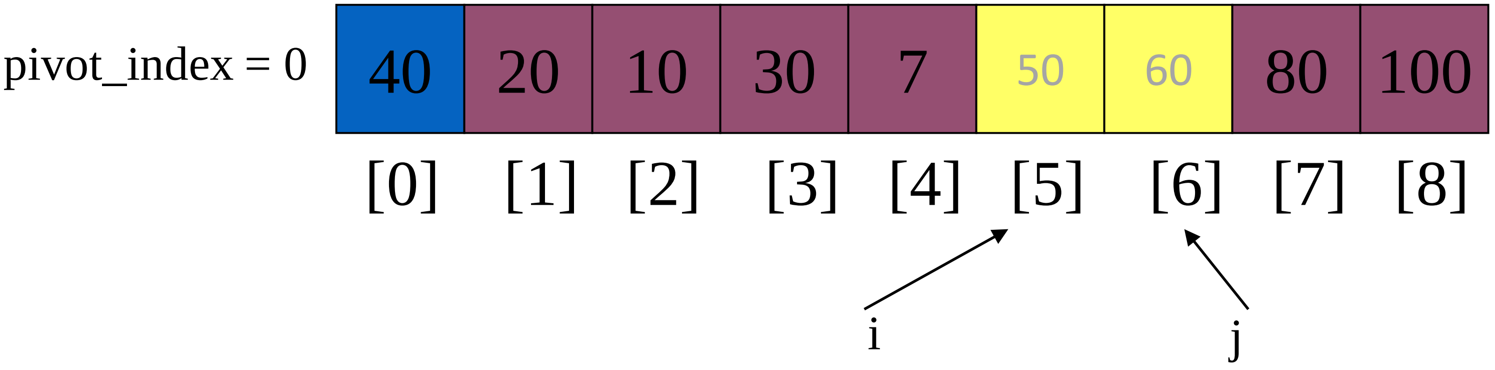

pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j


pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

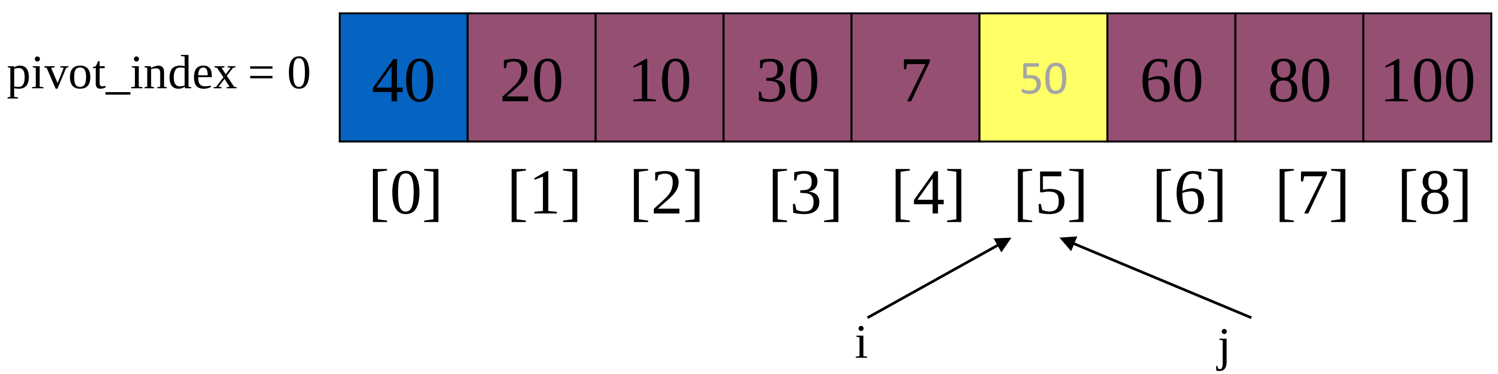

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

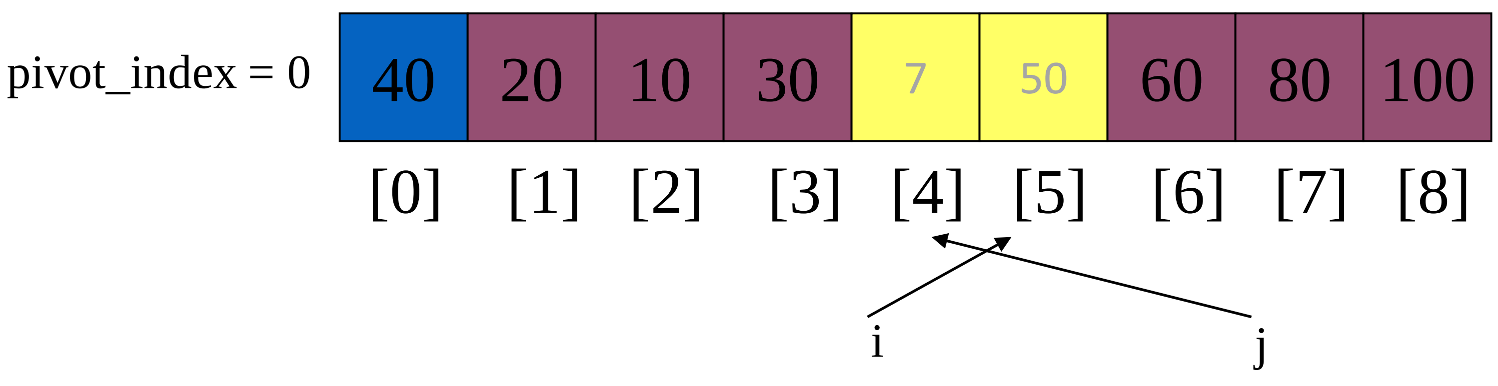

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

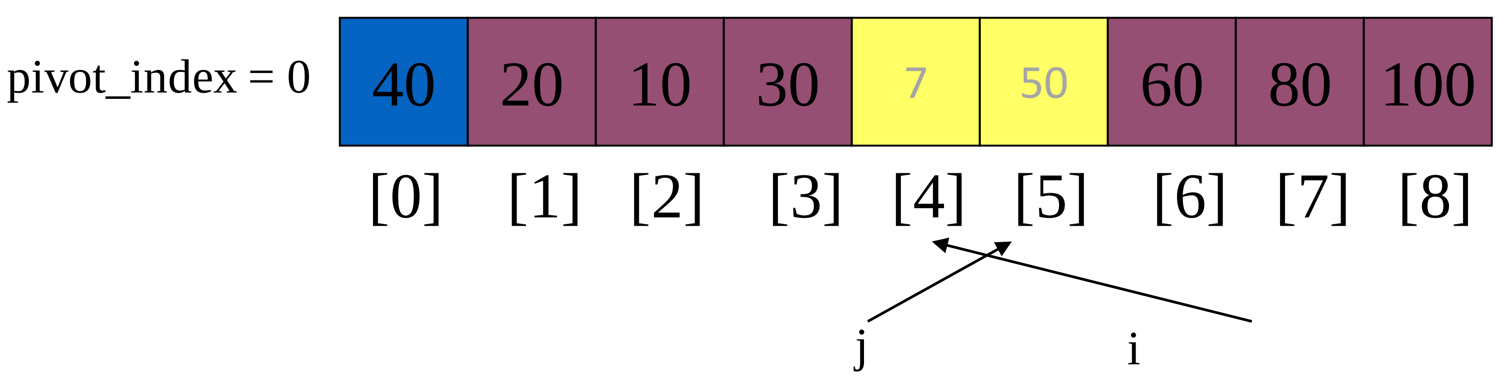

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j



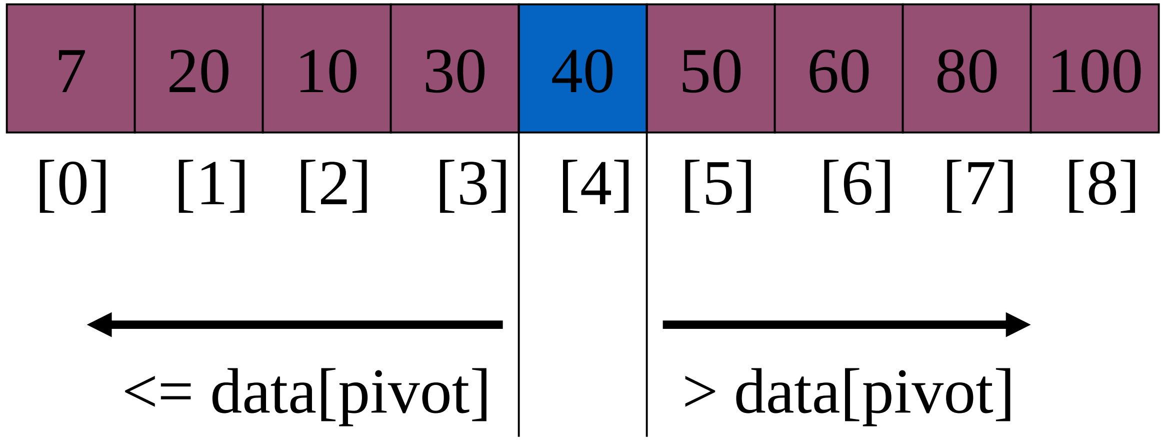
pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j


pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

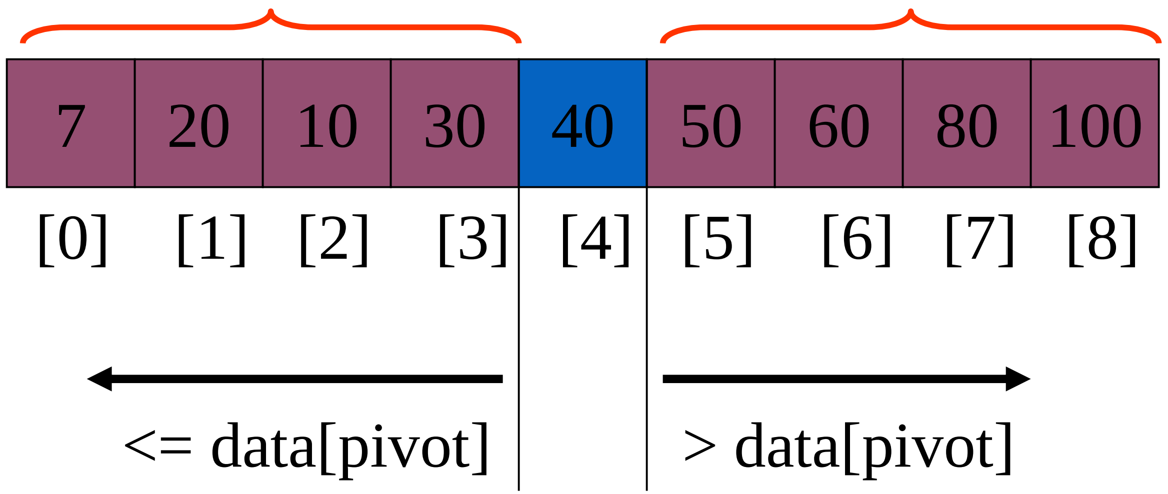

pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j


pivot_index = 0
40
20
10
30
7
50
60
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
j
i

Partition Result



Recursion: Quicksort Sub-arrays



Quick Sort: Complexity analysis

Memory requirement:

Size of the stack is:

$$S(n) = \lfloor log_2 n \rfloor + 1$$

Number of comparisons:

- Let, $T(n)$ represents total time to sort n elements and $P(n)$ represents the time for perform a partition of a list of n elements.

$$T(n) = P(n) + T(n_l) + T(n_r), \quad \text{with } T(1) = T(0) = 0$$

where, n_l = number of elements in the left sub list

n_r = number of elements in the right sub list and $0 \leq n_l, n_r < n$

Quick Sort: Complexity analysis

$$C(n) = \underbrace{n}_{\textit{Partition}} + \underbrace{C(k-1)}_{\substack{\textit{Sort left subarray}\\ \textit{by recursive call}\\ \textit{to quicksort}}} + \underbrace{C(n-k)}_{\substack{\textit{Sort right subarray}\\ \textit{by recursive call}\\ \textit{to quicksort}}}, \quad C(0) = C(1) = 0$$

(k = final position of pivot element)

Quick Sort: Complexity analysis

Case 1: Elements in the list are in ascending order

Number of comparisons:

$$C(n) = n - 1 + C(n-1), \text{ with } C(1) = C(0) = 0$$

$$C(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

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

Number of movements:

$$M(n) = 0$$

Quick Sort: Complexity analysis

Case 2: Elements in the list are in reverse order

Number of comparisons:

$$C(n) = n - 1 + C(n-1), \text{ with } C(1) = C(0) = 0$$

$$C(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

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

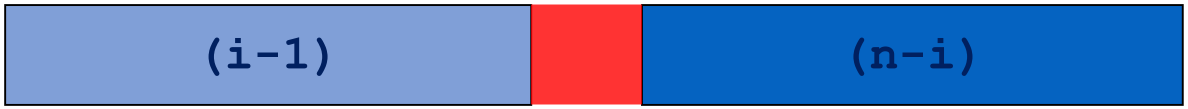
Number of movements:

$$M(n) = \begin{cases} \dfrac{n-1}{2}, if\ n\ is\ odd \\ \dfrac{n}{2}, if\ n\ is\ even \end{cases}$$

Quick Sort: Complexity analysis

Case 3: Elements in the list are in random order

i^{th} location



Number of comparisons:

$$C(n) = n + C(k-1) + C(n-k), \qquad C(0) = C(1) = 0$$

all values of k are equally likely. We must average over all k.

$$C(n) = \sum_{k=1}^{n} \frac{1}{n}[n + C(k-1) + C(n-k),] \quad with\ C(1) = C(0) = 0$$

s

Quick Sort: Complexity analysis

$$C(n) = n + \sum_{k=1}^{n} \frac{1}{n}[C(k-1) + C(n-k)]$$

$$C(n) = n + \frac{1}{n}\sum_{k=1}^{n} C(k-1) + \frac{1}{n}\sum_{k=1}^{n} C(n-k)$$

Note:

$\sum_{k=1}^{n} C(k-1) = \sum_{i=0}^{n-1} C(i)$, by substituting i = k−1.

$\sum_{k=1}^{n} C(n-k) = \sum_{i=0}^{n-1} C(i)$, by substituting i = n-k.

$$C(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} C(i)$$

Quick Sort: Complexity analysis

$$C(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} C(i)$$

$$nC(n) = n^2 + 2\sum_{i=0}^{n-1} C(i) \ldots\ldots (1)$$

Writing down the same recurrence with n−1 replacing n, we get

$$(n-1)C(n-1) = (n-1)^2 + 2\sum_{i=0}^{n-2} C(i) \ldots\ldots\ldots\ldots (2)$$

(1) – (2)

$$nC(n) - (n-1)C(n-1) = n^2 - (n-1)^2 + 2C(n-1)$$

$$nC(n) = 2n - 1 + (n+1)C(n-1)$$

1.39n lg(n)/6 ≈ 0.23 n lg(n) exchanges.

Quick Sort: Complexity analysis

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

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

Quick Sort: Complexity analysis

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

"Telescoping" $\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2c}{n+1}$ to get the explicit form:

$$\frac{T(n)}{n+1} + \frac{T(n-1)}{n} + \frac{T(n-2)}{n-1} + \ldots + \frac{T(2)}{3} + \frac{T(1)}{2}$$

$$- \frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} - \ldots - \frac{T(2)}{3} - \frac{T(1)}{2} - \frac{T(0)}{1}$$

$$= \frac{2c}{n+1} + \frac{2c}{n} + \ldots + \frac{2c}{3} + \frac{2c}{2},$$ or

---

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \frac{1}{n+1}\right) \approx 2c(H_{n+1} - 1) \approx c' \log n$$

($H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$ is the n^{th} harmonic number).

Quick Sort: Complexity analysis

Case 3: Elements in the list are in random order

Number of movements:

$$M(n) = \frac{1}{n}\sum_{i=1}^{n}[(i-1) + M(i-1) + M(n-i)]$$

$$M(n) = \frac{n-1}{2} + \frac{2}{n}\sum_{i=1}^{n-1} M(i)$$

$$M(n) = 2(n+1)(log_e n + 0.577) - 4n$$

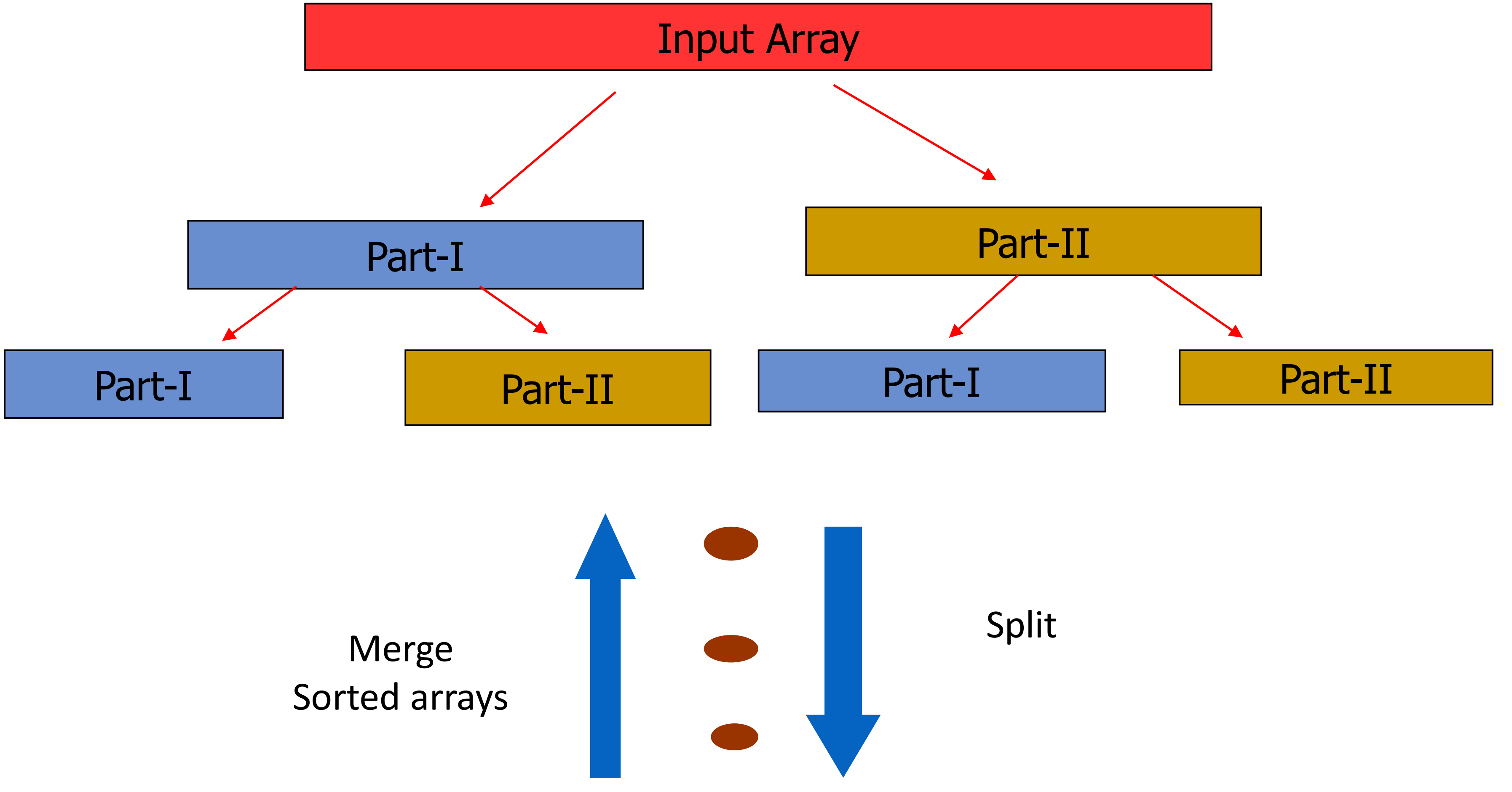
Quick Sort: Summary of Complexity analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$c(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 1$	Input list is in sorted order
Case 2	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \left\lfloor\frac{n}{2}\right\rfloor$	$S(n) = 1$	Input list is sorted in reverse order
Case 3	$C(n) = 2(n+1)(log_e n + 0.577) - 4n$	$M(n) = 2(n+1)(log_e n$	$S(n) = \lceil log_2 n \rceil + 1$	Input list is in random order

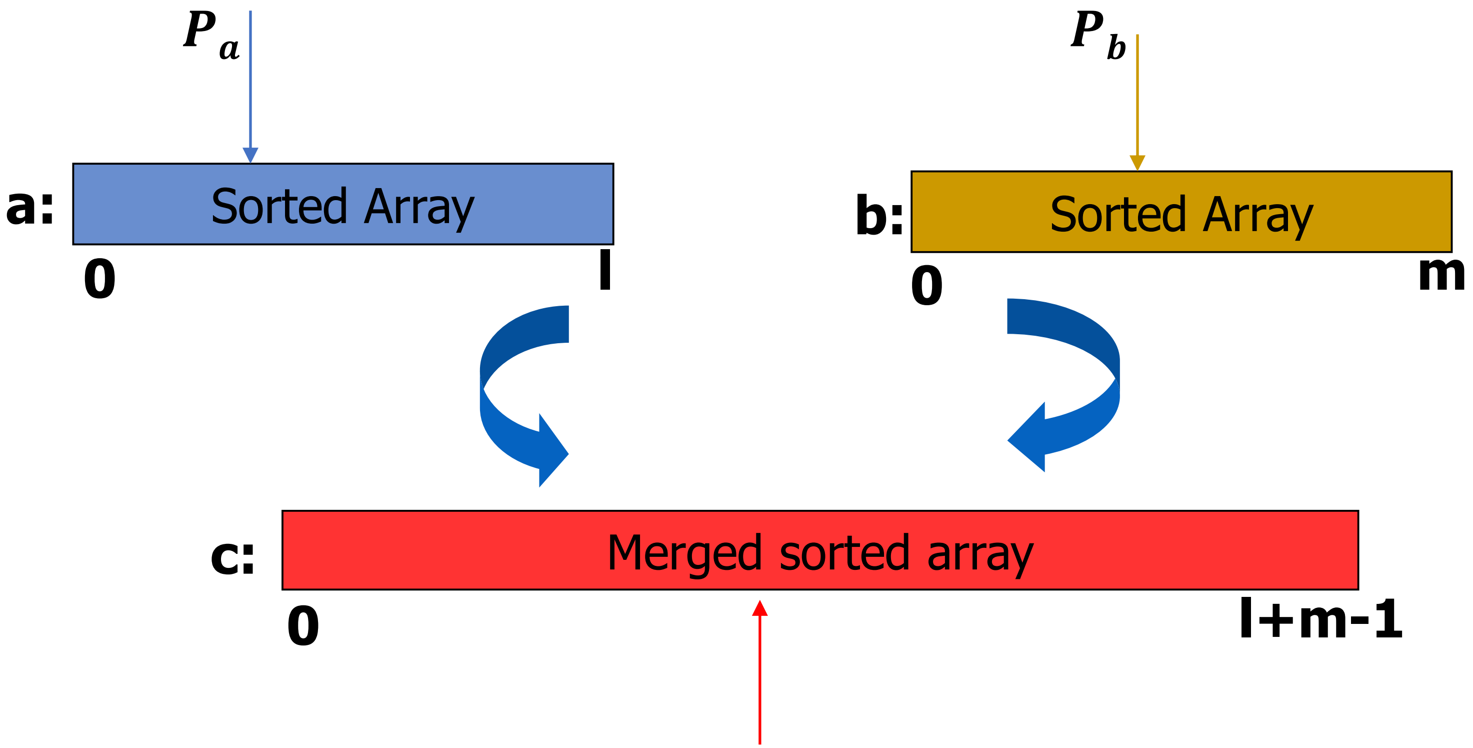
Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Worst case
Case 2	$T(n) = c\left(\frac{n(n-1)}{2} + \left\lfloor\frac{n}{2}\right\rfloor\right)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = 4c(n+1)(log_e n + 0.577) - 8cn$ $T(n) = 2c[nlog_2 n - n + 1]$	$T(n) = O(nlog_2 n)$	Best / Average case

Merge Sort

Merge Sort – How it Works?

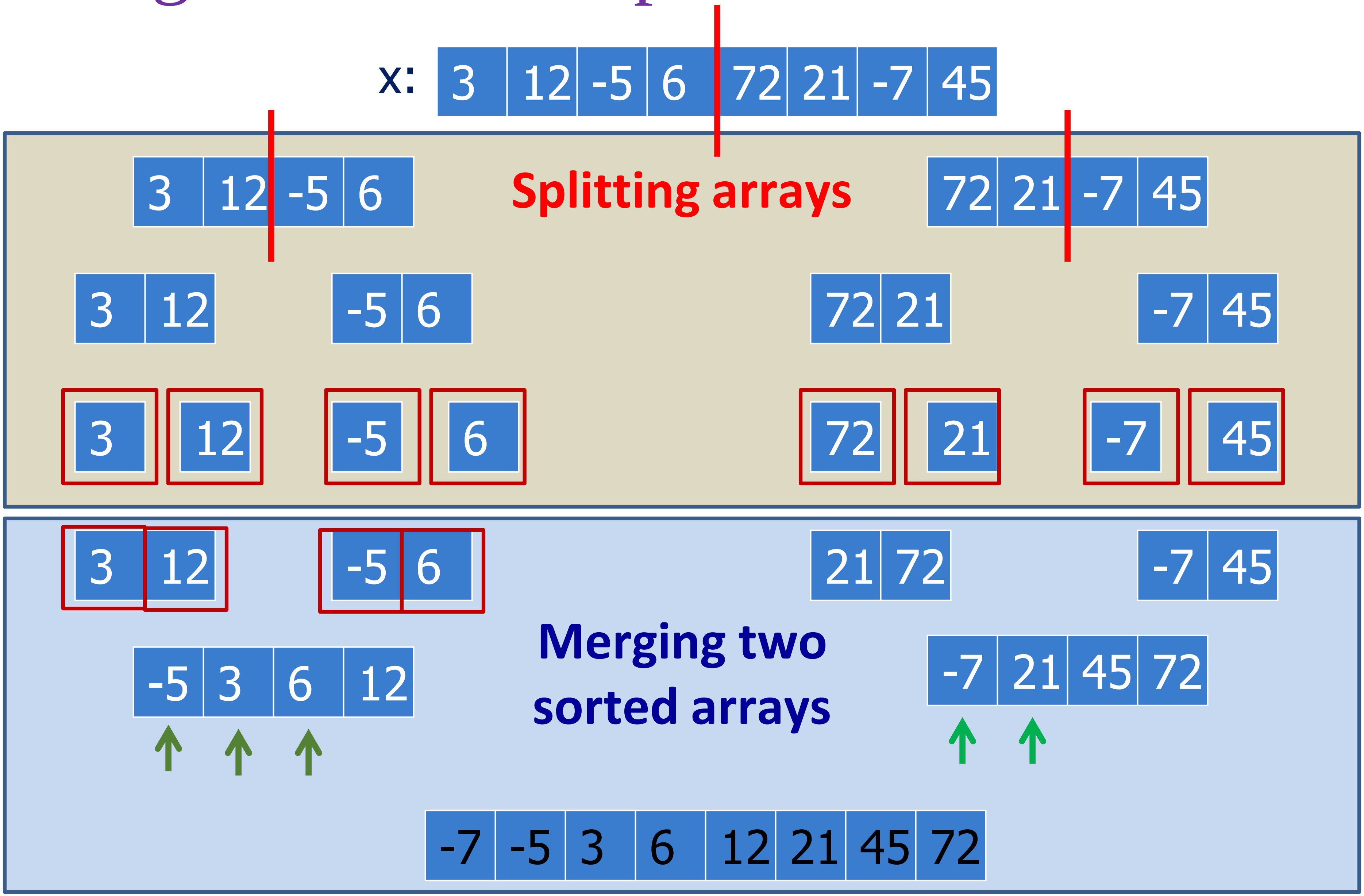


Merging two Sorted arrays



Move and copy elements pointed by P_a if its value is smaller than the element pointed by P_b in $(l + m - 1)$ operations and otherwise.

Merge Sort – Example



Merge Sort Program

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
    int a[30],n,i;
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter array elements:");

    for(i=0;i<n;i++)
      scanf("%d",&a[i]);

    mergesort(a,0,n-1);

    printf("\nSorted array is :");
    for(i=0;i<n;i++)
      printf("%d ",a[i]);

    return 0;
}
```

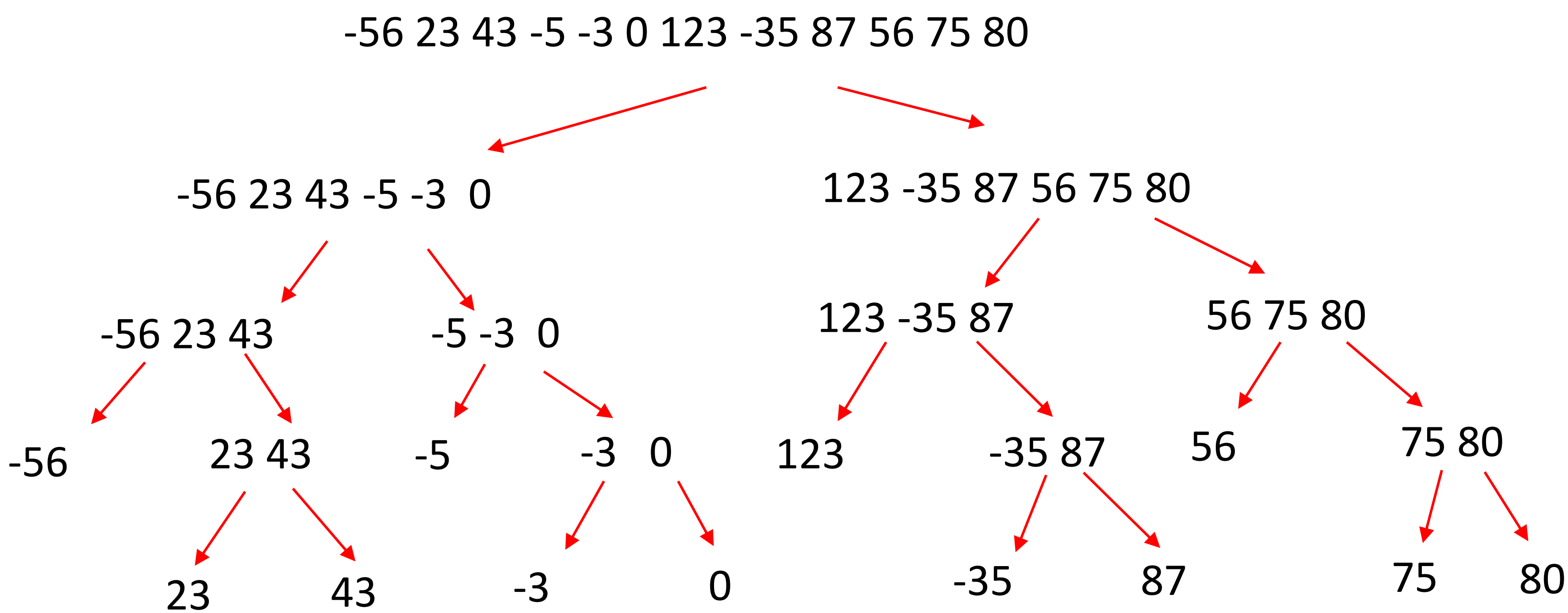
Merge Sort Program

```
void mergesort(int a[],int i,int j)
{
    int mid;

    if(i<j) {
    mid=(i+j)/2;
   /* left recursion */
    mergesort(a,i,mid);
   /* right recursion */
    mergesort(a,mid+1,j);
 /* merging of two sorted sub-arrays */
    merge(a,i,mid,mid+1,j);
    }
}
```

Merge Sort Program

```
void merge(int a[],int i1,int i2,int j1,int j2)
{
    int temp[50]; //array used for merging
    int i=i1,j=j1,k=0;

    while(i<=i2 && j<=j2) //while elements in both lists
    {
      if(a[i]<a[j])
         temp[k++]=a[i++];
      else
         temp[k++]=a[j++];
     }

    while(i<=i2) //copy remaining elements of the first list
       temp[k++]=a[i++];

    while(j<=j2) //copy remaining elements of the second list
       temp[k++]=a[j++];

    for(i=i1,j=0;i<=j2;i++,j++)
    a[i]=temp[j]; //Transfer elements from temp[] back to a[]
}
```

Merge Sort – Splitting Trace



Output: -56 -35 -5 -3 0 23 43 56 75 80 87 123

Space Complexity??

Worst Case: O(n.log(n))

Merge Sort: Complexity analysis

Time Complexity:

$$T(n) = D(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + C(n) \qquad if\ n > 1$$

$$T(n) = c_1 \qquad if\ n = 1$$

- For simplicity of calculation

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) = T\left(\frac{n}{2}\right)$$

$$T(n) = c_1 \qquad if\ n = 1$$

$$T(n) = c_2 + 2T\left(\frac{n}{2}\right) + (n-1) \qquad if\ n > 1$$

$$T(n) = n \cdot c_1 + nlog_2 n + (c_2 - 1)(n-1) \qquad Assuming\ n = 2^k$$

Quick Sort vs. Merge Sort

- **Quick sort**
  - **hard division, easy combination**
  - partition in the divide step of the divide-and-conquer framework
  - hence combine step does nothing
- **Merge sort**
  - **easy division, hard combination**
  - merge in the combine step
  - the divide step in this framework does one simple calculation only

Quick Sort vs. Merge Sort

Both the algorithms divide the problem into two sub problems.

- **Merge sort:**
  - two sub problems are of almost equal size always.

- **Quick sort:**
  - an equal sub division is not guaranteed.

- This difference between the two sorting methods appears as the deciding factor of their run time performances.