SORTING AND SEARCHING

DATA STRUCTURES AND ALGORITHMS

SORTING AND SEARCHING

Sorting Algorithms

- Sorting Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort
  - Counting sort

- Search Algorithms
  - Linear search
  - Binary search
    - Non recursive

SORTING: BUBBLE SORT

ALGORITHMS AND DATA STRUCTURES

BUBBLE SORT

- Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Pseudocode

- Complexity
  - Best-Case O(n)
  - Worst Case O(n2)
  - Average Case O(n2)
  - Space complexity O(1)
  - Stable - yes

```
bubblesort(data[],n)
    for  I = 0 to n22
        for j  = n-1  down to i+1
            swap elements in positions j and j-1  if they are out of order;
```

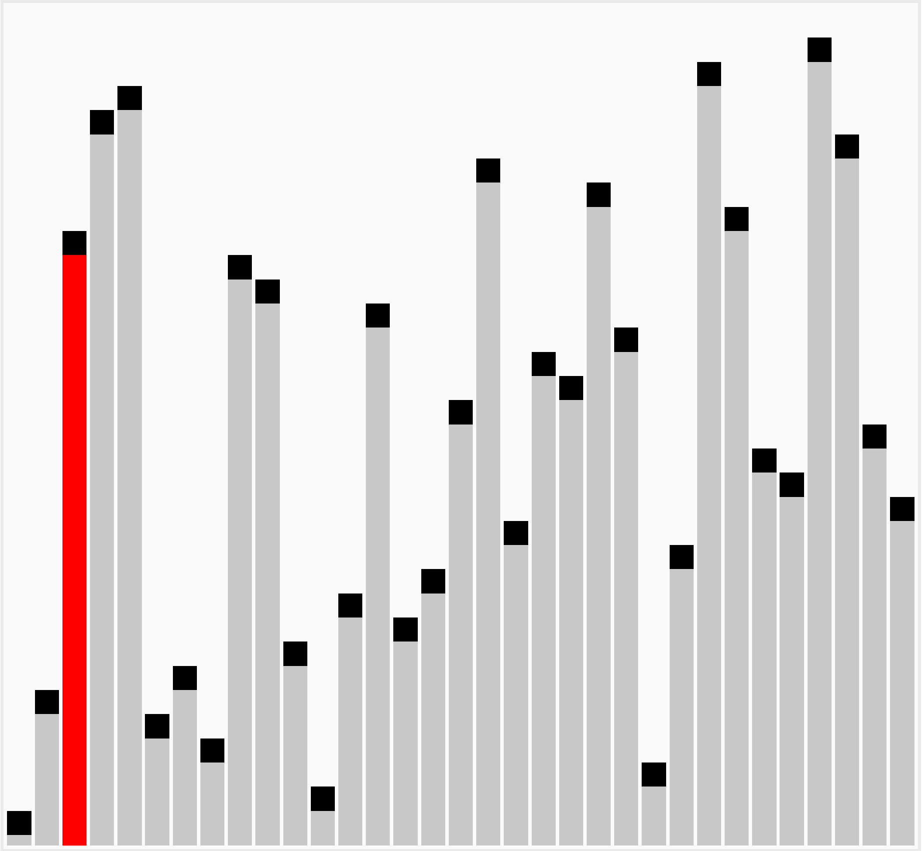
BUBBLE SORT

Work Principles

1st Compare		2nd Compare	3rd Compare	4th Compare	5th Compare	
17	0	17	17	17	17	**5**
25	1	25	25	25	5	17
8	2	8	8	5	25	25
5	3	5	5	8	8	8
49	4	12	12	12	12	12
12	5	49	49	49	49	49

BUBBLE SORT

```
void bubble_sort(int a[], int length){

    int temp;
    for (int i = 0; i < length-1; i++)
    {
        for (int j = 0; j < length-1-i; j++)
        {
            if(a[j] < a[j+1]){
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

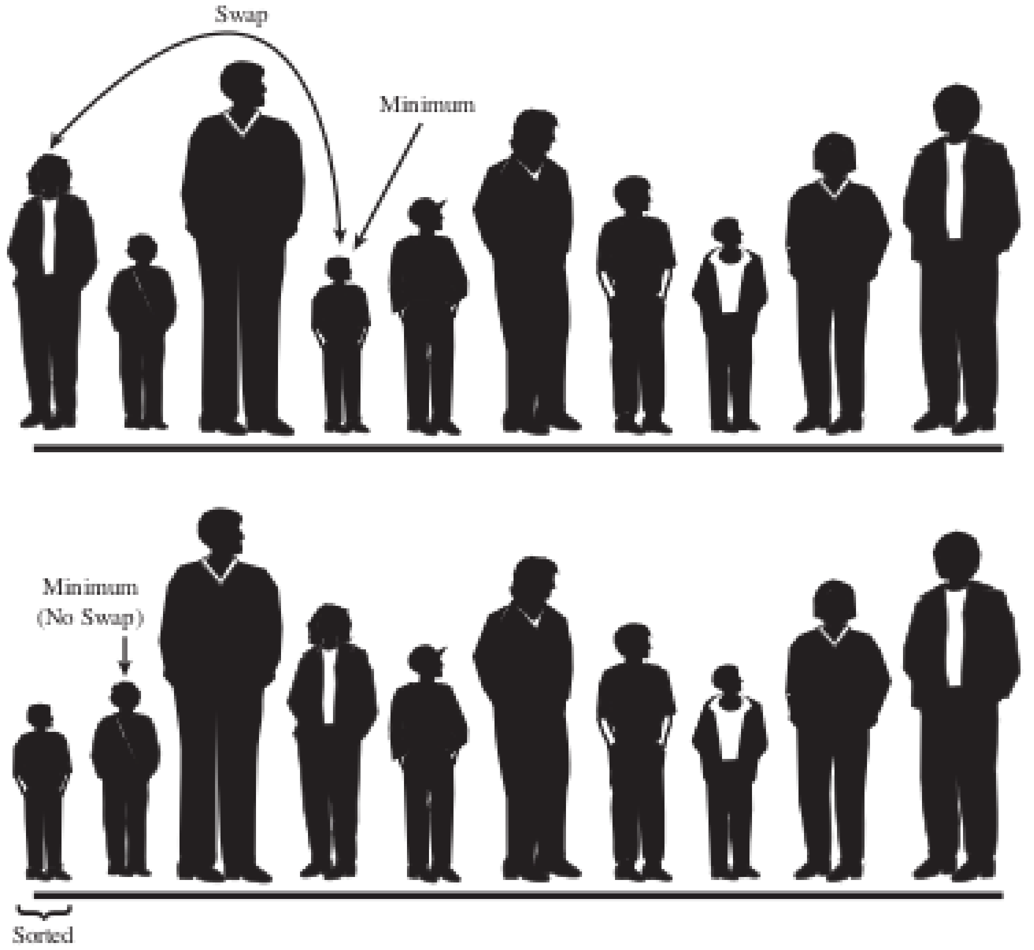


SORTING: SELECTION SORT

ALGORITHMS AND DATA STRUCTURES

SELECTION SORT

- selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.
- Complexity
  - Best-Case O(n)
  - Worst Case O(n2)
  - Average Case O(n2)
  - Space complexity O(1)
  - Stable - yes

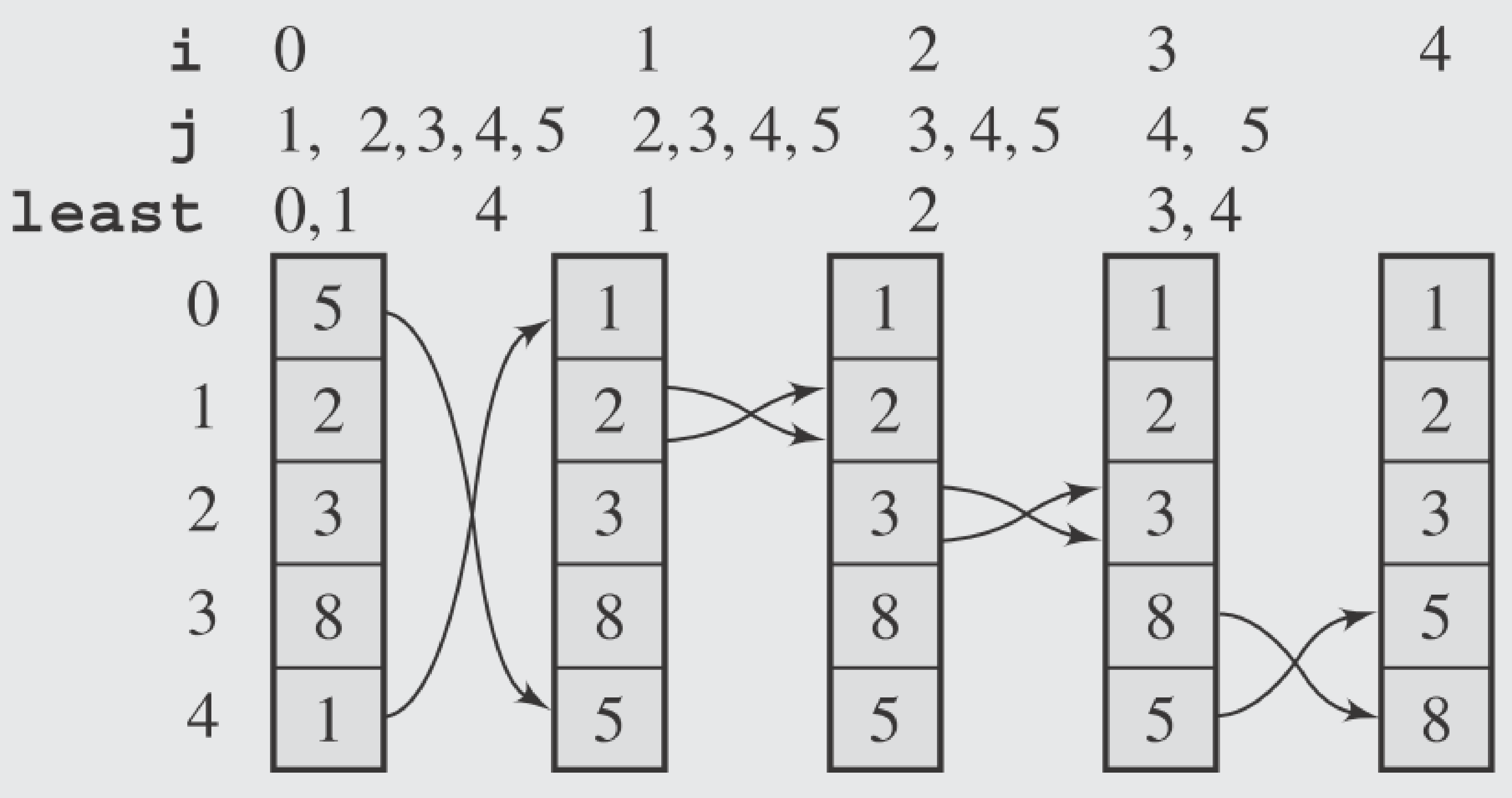


Pseudocode

selectionsort(data[ ],n)
for I = 0 to n-2
select the smallest element among data[i], . . . , data[n-1];
swap it with data[i];

SELECTION SORT

Work Principles



SELECTION SORT

```
void selection_sort(int a[], int length){

    int temp, min;
    for (int i = 0; i < length; i++)
    {
        min = i;
    for (int j = i; j < length; j++)
    {
        if(a[j] < a[min])
            min = j;
    }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
    }
}
```

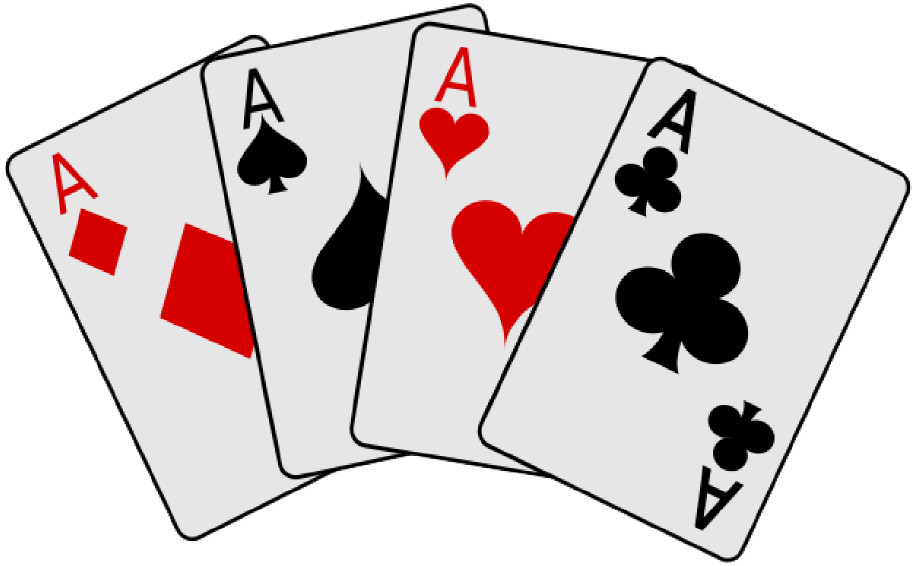
SORTING: INSERTION SORT

ALGORITHMS AND DATA STRUCTURES

INSERTION SORT

- Properties

  - Innsertion sort takes advantage of presorting. More efficient in practice than most other simple quadratic (O(n2)) algorithms such as selection sort or bubble sort
  - Efficient for small data sets.
  - O(n + d), where d is the number of inversions.
  - It is stable sort. It does not change the relative order of elements with equal keys.
  - It is in-place sorting. It only requires a constant amount O(1) of additional memory space.

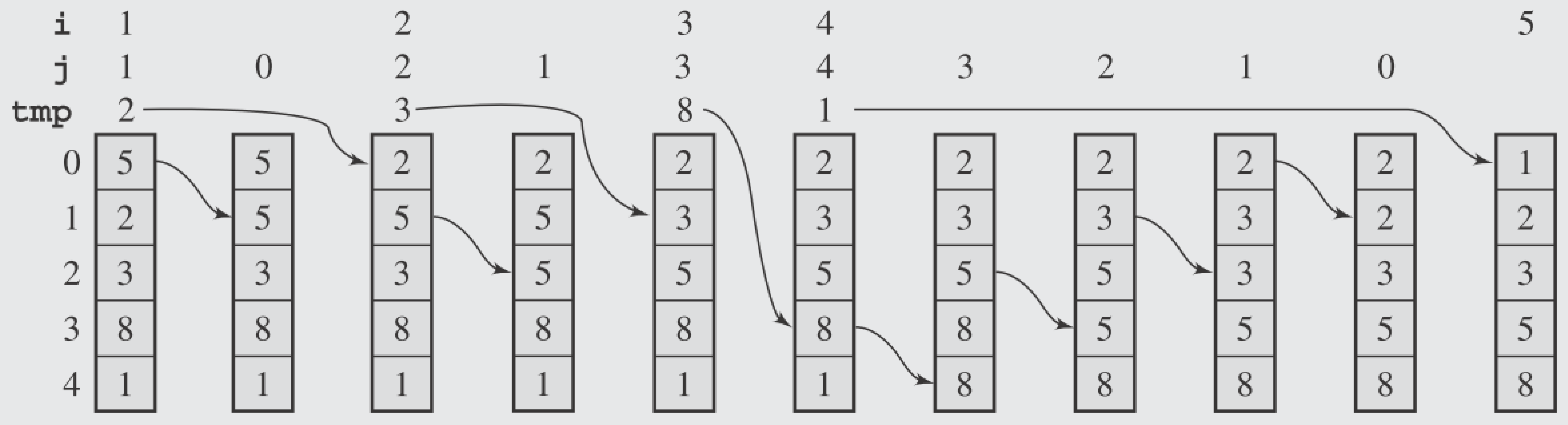


Pseudocode

insertionsort(data[ ],n)
    for I = 1 to n-1
        move all elements data[j] greater than data[i] by one position;
        place data[i] in its proper position;

INSERTION SORT

Work Principles



INSERTION SORT

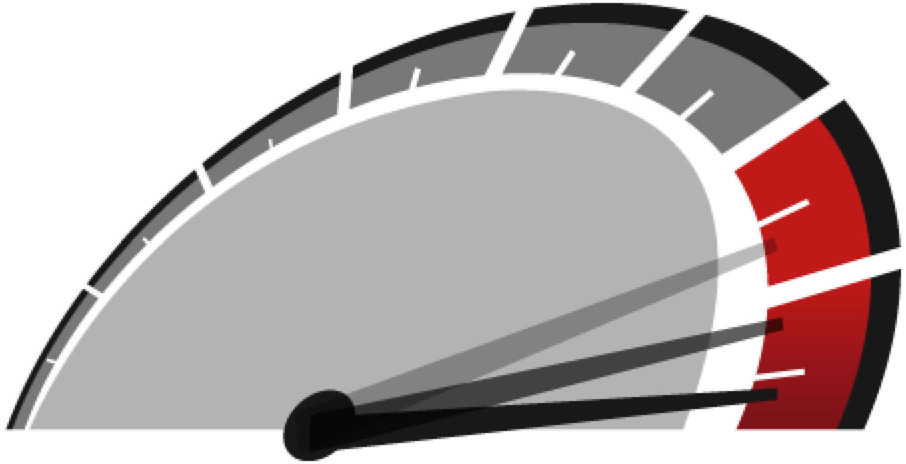
```
void insertion_sort(int a[], int length){

    int temp, j;
    for (int i = 0; i < length; i++)
    {
        j = i;
        while (j > 0 && a[j] <a[j-1])
        {
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
            j--;
        }
    }
}
```

SORTING: QUICK SORT

ALGORITHMS AND DATA STRUCTURES

QUICK SORT

- Properties
  - Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways.
  - The key process in quick Sort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.
- Complexity
  - Best Case O(nlog n)
  - Worst-case O(n2)
  - Average-Case O(n log n)
  - Space complexity log n,
  - Not stable

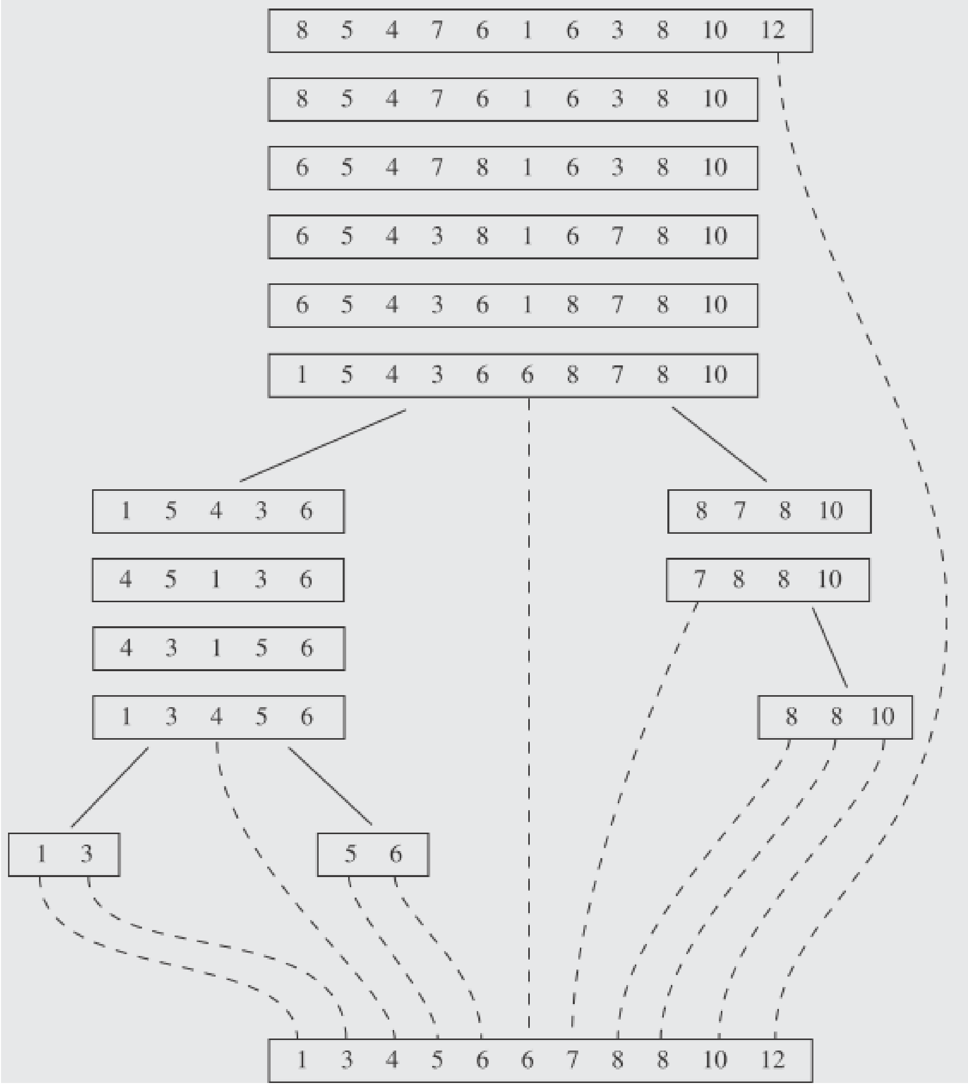


Pseudo code

```
quicksort(array[])
 if length(array)  >   1
    choose bound; // partition array into subarray1 and subarray2
    while there are elements left in array
      include     element either in subarray  = {el: el ≤ bound}
            or in subarray2 = {el:  el ≥ bound};
    quicksort(subarray1));
    quicksort(subarray2);
```

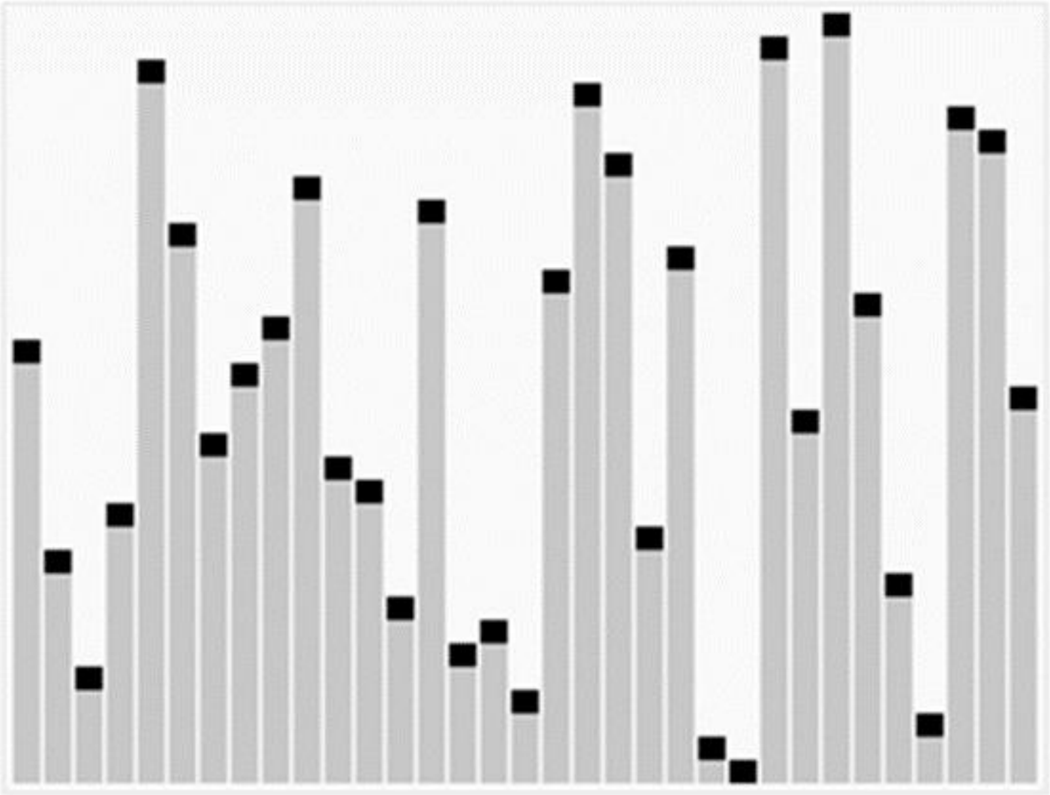
QUICK SORT

Work Principles



QUICK SORT

```
void quick_sort(int a[], int left, int right){
int i = left;
int j = right;
int temp;
int pivot = a[left + (right-left)/2];
while (i <= j){
    while(a[i] < pivot)
        i++;
    while (a[j] > pivot)
        j--;
    if(i <= j){
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i++;
        j--;
    }
}
if(i < right)
    quick_sort(a,i,right);
if(j > left)
    quick_sort(a,left, j);
}
```



SORTING: MERGE SORT

ALGORITHMS AND DATA STRUCTURES

MERGE SORT

- Properties
  - Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.Complexity
  - Best Case O(n log n)
  - Worst-case O(n log n)
  - Average-Case O(n log n)
  - Space complexity O(n)
  - Stable

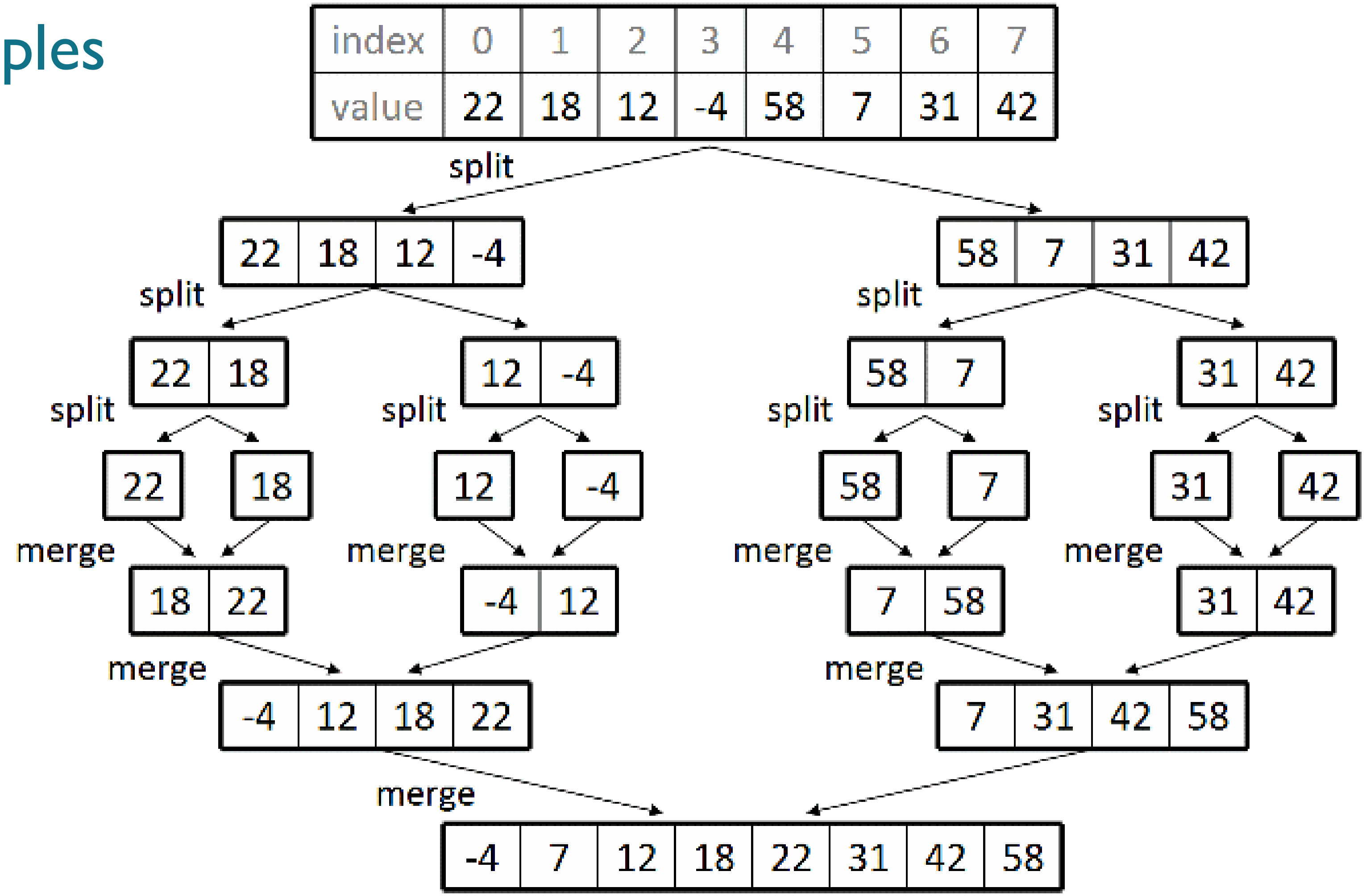
Pseudo code

mergesort(data[])
if data have at least two elements
mergesort(left half of data);
mergesort(right half of data);
merge(both halves into a sorted list);

merge(array1[ ], array2[ ], array3[ ])
i1, i2, i3 are properly initialized;
while both array2 and array3 contain elements
if array2[i2] < array3[i3]
array1[i1++] = array2[i2++];
else array1[i1++] = array3[i3++];
load into array1 the remaining elements of either array2 or array3;

MERGE SORT

Work Principles



MERGE SORT

```
void merge_sort(int a[], int left, int right){

    if(right > left){

        int middle = left + (right-left)/2;
        merge_sort(a, left, middle);
        merge_sort(a, middle+1, right);
        merge(a, left, middle, right);

    }
}
```

MERGE SORT

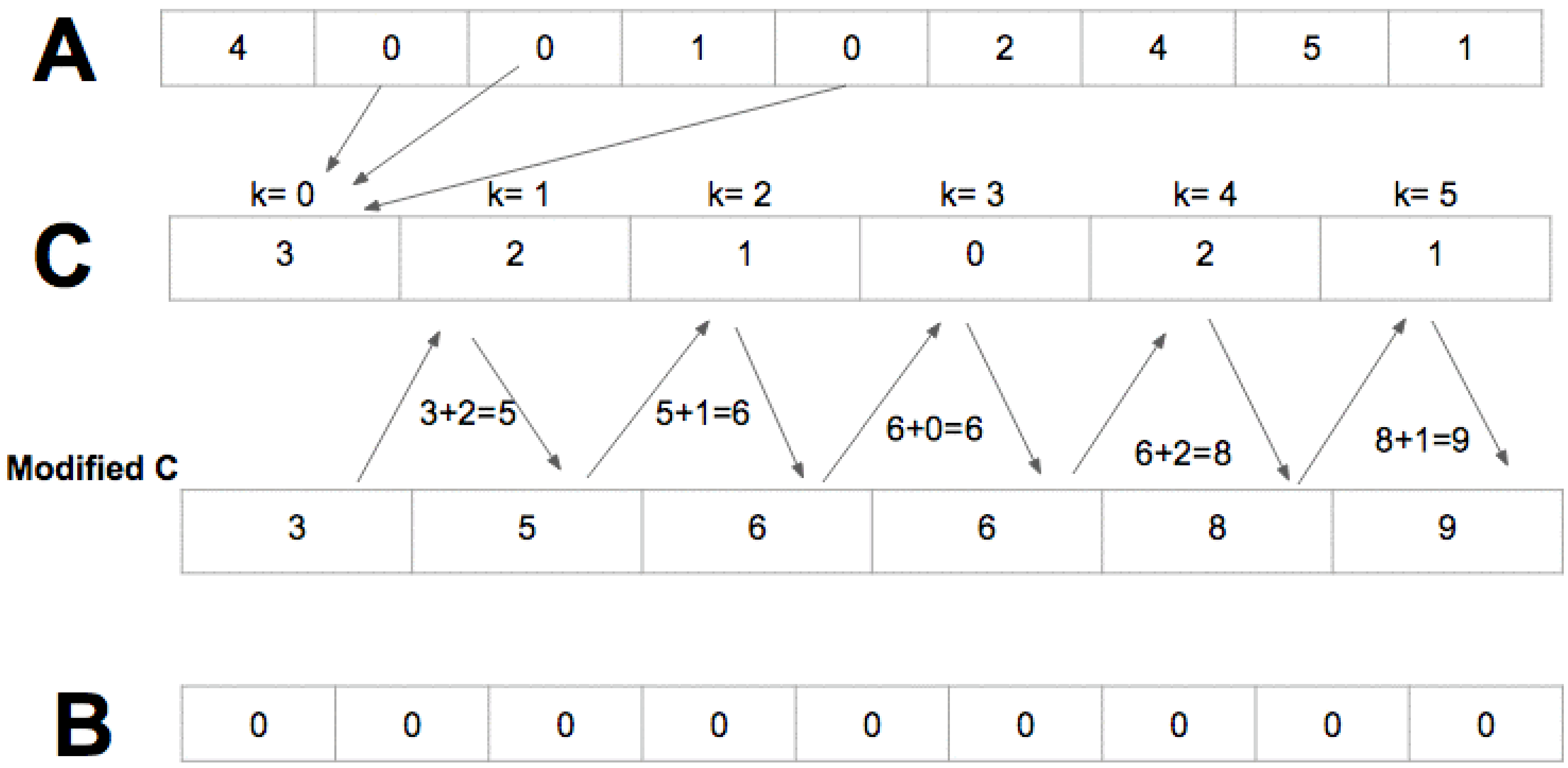
```
void merge(int a[], int left, int middle, int right){

    int n1 = middle - left + 1;
    int n2 = right - middle;
    int *L = new int[n1];
    int *R = new int[n2];
    for(int i = 0; i < n1; i++)
        L[i] = a[left+i];
    for(int j = 0; j < n2; j++)
        R[j] = a[middle+j+1];
    int i,j,k;
    i = j = 0;
    k = left;

    while (i < n1 && j < n2){
        if(L[i] <= R[j])
            a[k++] = L[i++];
        else
            a[k++] = R[j++];
    }
    while (i < n1)
        a[k++] = L[i++];
    while (j < n2)
        a[k++] = R[j++];
    delete[] R;
    delete[] L;
}
```

SORTING: COUNTING SORT

ALGORITHMS AND DATA STRUCTURES

COUNTING SORT

Work Principles



COUNTING SORT

```
void counting_sort(int a[], int length){

    int min, max, idx;
    min = max = a[0];
    for (int i = 0; i < length; i++){
        max = a[i] > max ? a[i] : max;
        min = a[i] < min ? a[i] : min;
    }
    int b = max - min + 1;
    int *bucket = new int[b];
    memset(bucket, 0, b * sizeof(int));
    for (int i = 0; i < length; i++)
        bucket[a[i] - min]++;
    idx = 0;
    for (int i = min; i <= max; i++){
        for (int j = 0; j < bucket[i - min]; j++){
                a[idx++] = i;
        }
    }
    delete[] bucket;
}
```

SEARCH ALGORITHMS: BINARY SEARCH

ALGORITHMS AND DATA STRUCTURES

BINARY SEARCH

```
int binary_search(int a[], int left, int right, int value){

    while (left < right)
    {
        int middle = left + (right - left)/2;
        if(value < a[middle])
            right = middle - 1;
        else if(value > a[middle])
            left = middle + 1;
        else
            return middle;
    }
    return -1;
}
```

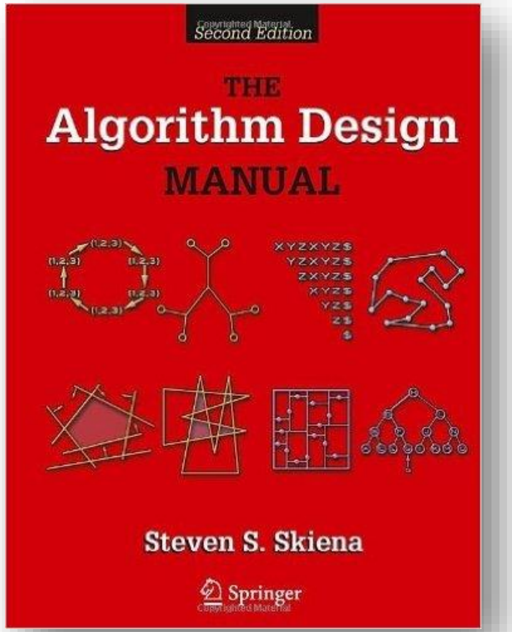
BINARY SEARCH: RECURSIVE VERSION

```
int binary_search_r(int a[], int left, int right, int value){

    if(left > right)
        return -1;
    int middle = left + (right- left)/2;
    if(value < middle)
        binary_search_r(a,left, middle-1,value);

    else if( value > middle)
        binary_search_r(a, middle + 1, right, value);

    else
        return middle;
}
```

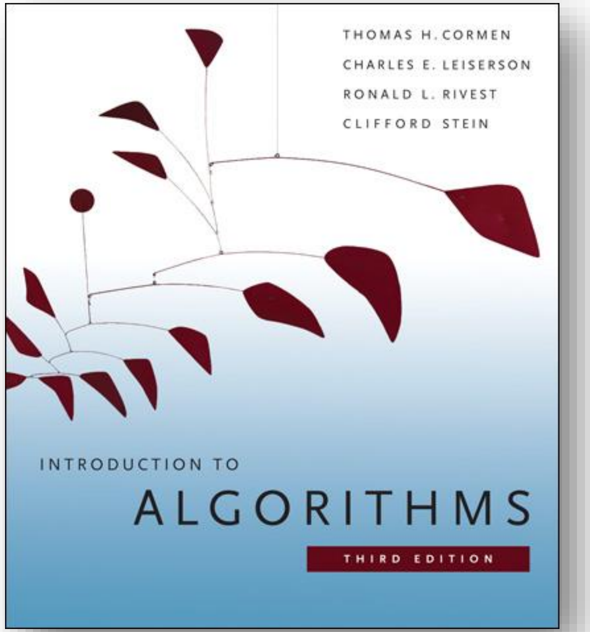
OVERVIEW

Algorithm	Best-case	Worst-case	Average-case	Space Complexity	Stable?
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$\log n$ best, n avg	Usually not*
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting Sort	$O(k + n)$	$O(k + n)$	$O(k + n)$	$O(k + n)$	Yes

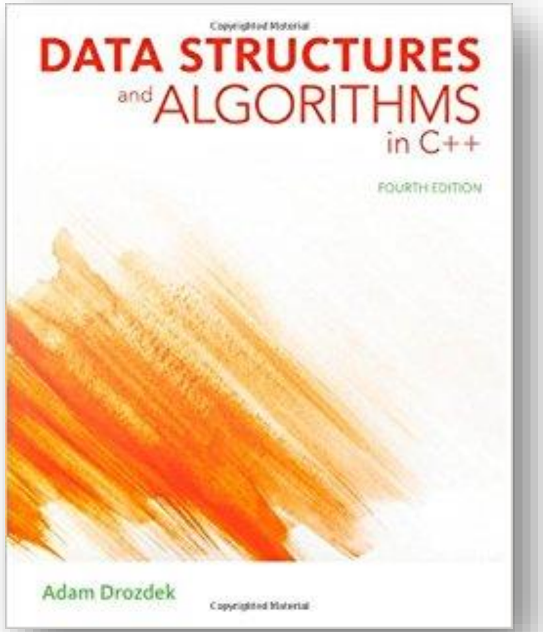
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 3: Sorting and Searching
Page: 103



Thomas H. Cormen
Introduction to Algorithms
Chapter II: Sorting and Orders Statistics
Page: 147



Adam Drozdek
Data structures and Algorithms in C++
Chapter 9: sorting
Page: 491