8 : Sorting and Searching Algorithms

IT3206 – Data Structures and Algorithms

Level II - Semester 3

Overview

- This section will illustrate several sorting and searching algorithms and will discuss the implementations for each algorithm.

- This section also discusses the time complexities of each sorting and searching algorithm.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
    - Explain selected searching and sorting Algorithms.
    - Demonstrate implementations of selected searching and sorting algorithms.
    - Analyze the time complexities of selected searching and sorting algorithms.

List of subtopics

1. Introduction to iterative and, Divide and Conquer Methodology
2. Sorting
   i. Iterative Method
      a) Bubble sort
      b) Selection sort
      c) Insertion sort
   ii. Divide and Conquer Method
      a) Merge sort
      b) Quick sort
      c) Radix Sort
      d) Heap Sort
3. Searching algorithms
   i. Linear search
   ii. Binary search
   iii. Interpolation search

8.1 Introduction to iterative and, Divide and Conquer Methodology

Iterative Method

- An iterative algorithm executes steps in iterations.
- Repetitive structure is used in the iterative methodology.

Divide and Conquer Method

- In divide and conquer methodology, the problem is break into several subproblems and solve the subproblems.
- Then combine these solutions to create the solution for the original problem.

8.2 Sorting

- a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

  - The output is in a certain order (increasing or decreasing).
  - The output is a permutation, or reordering, of the input

8.2.1 Iterative Method

- The Iterative method uses repetition structure.
- Looping statements are used in the iterative method such as :
    - for loop
    - while loop
    - do while loop
    - repeat until etc.
- The following basic algorithms are discussed in this section.
    - Bubble sort (sorting by exchange)
    - Selection sort (sorting by selecting)
    - Insertion sort (sorting by insertion)

8.2.1 Iterative Method - Sorting by Exchange

- Eg : Bubble sort
- One of the simplest sorting is algorithm is known as bubble sort. The algorithm as follows.
- Beginning at the last element in the list
- Compare each element with the previous element in the list. If an element is less than its predecessor, swap these two element.
- To completely sort the list, you need to perform this process n-1 times on a list of length n

8.2.1.1 Bubble sort

- *Bubble sort* is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. While simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes. Its average case and worst case are both $O(n^2)$.

- Bubble sort is an instance of a sorting by exchange category.

8.2.1.1 Bubble sort

- There are two main methods of exchanging the data elements in the bubble sort algorithm
  - from first data element to last data element
  - from last data element to first data element

8.2.1.1 Bubble sort

- Following is a demonstration of how bubble sorting works.

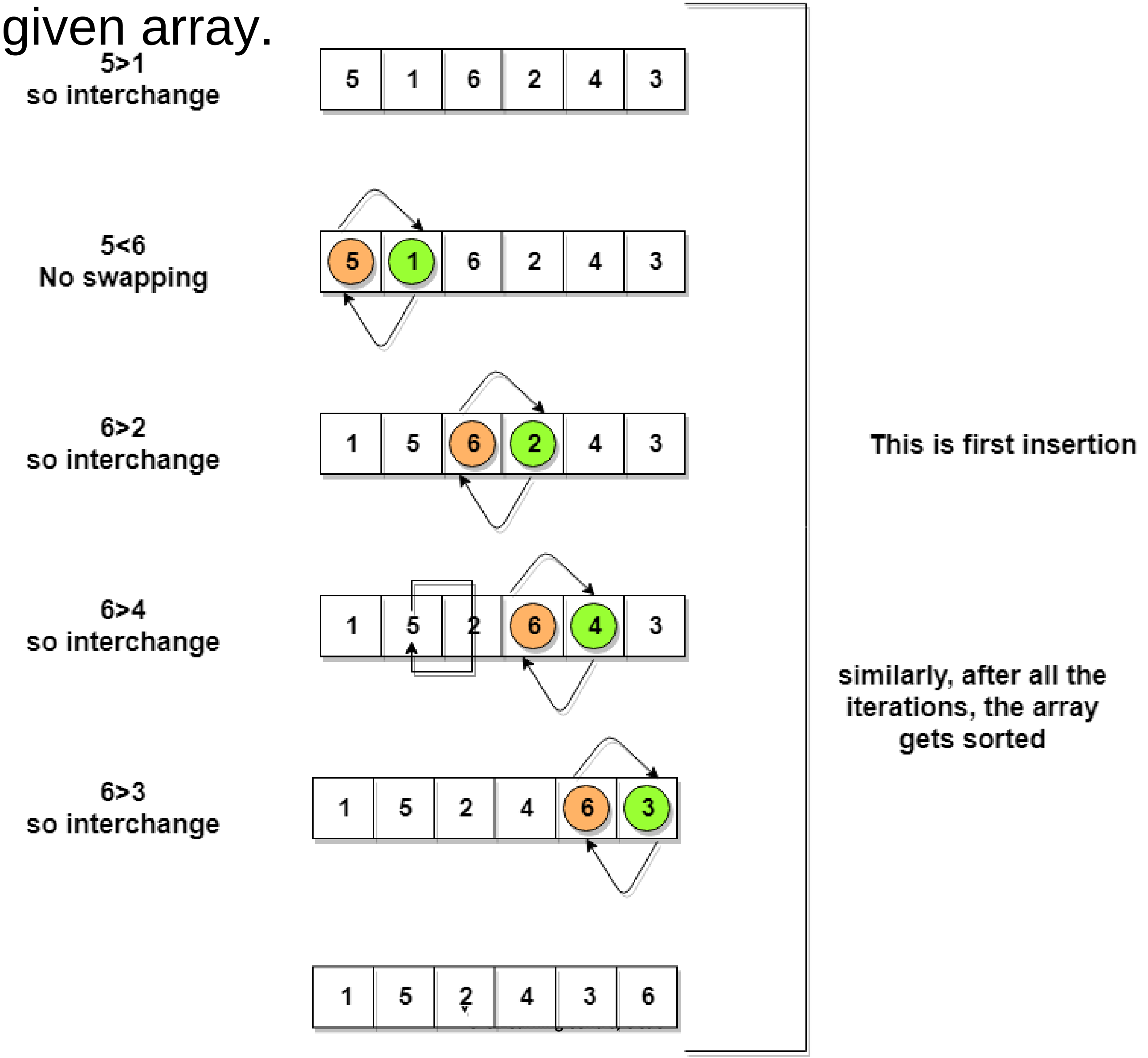
Example 01

6 5 3 1 8 7 2 4

https://commons.wikimedia.org/wiki/File:Bubble-sort.gif

Example 02

- Let's consider an array with values (5, 1, 6, 2, 4, 3)
- Below, we have a pictorial representation of how bubble sort will sort the given array.



8.2.1.1 Bubble sort

- The pseudocode for the bubble sort algorithm is given below:

```
begin BubbleSort(list)

   for all elements of list (iterate through all the data elements)
      if list[i] > list[i+1] (check whether the two data elements are in correct order)
         swap(list[i], list[i+1]) (if the data elements are not in the correct order, swap the two data elements)
      end if
   end for

   return list

end BubbleSort
```

8.2.1.1 Bubble sort

Let's consider an array with values (5 1 4 2 8).

First pass

- (**5 1** 4 2 8) –> (1 5 4 2 8)
  - The algorithm compares the first two elements (5 and 1).
  - Elements are not in the correct order (5 > 1)
    - Swaps element 5 with element 1
- (1 **5 4** 2 8) –> (1 4 5 2 8)
  - The algorithm compares the next two elements (5 and 1).
  - Elements are not in the correct order (5 > 4)
    - Swaps element 5 with element 4
- (1 4 **5 2** 8) –> (1 4 2 5 8)
  - The algorithm compares the next two elements (5 and 2).
  - Elements are not in the correct order (5 > 2)
    - Swaps element 5 with element 2
- (1 4 2 **5 8**) –> (1 4 2 5 8)
  - The algorithm compares the next two elements (5 and 8).
  - Elements are in the correct order (5 < 8)

End of first pass (All data elements are iterated)

8.2.1.1 Bubble sort

Second pass

- (**1 4** 2 5 8) –> (1 4 2 5 8)
- (1 **4 2** 5 8) –> (1 2 4 5 8),
  - Elements are not in the correct order (4 > 2)
    - Swaps element 4 with element 2
- (1 2 **4 5** 8) –> (1 2 4 5 8)

End of the second pass (All data elements are iterated)

- All data elements are in sorted order (1 2 4 5 8).

8.2.1.1 Bubble sort

Eg :

- (a)Run through the bubble sort algorithm by hand on the list :44,55,12,42,94,18,06,67
- (b) Write a Java program to implementing the bubble sort algorithm on an array.
- The basic idea underlying the bubble sort is to pass through the file sequentially several times.
- Each pass consists of comparing each element in the file with its predecessor [x[i] and x[i-1] and interchanging the two elements if they are not proper order.

8.2.1.1 Bubble sort

- The following comparisons are made on the first pass
- 44,55,12,42,94,18,06,67
- Data[7] with Data[6] (67,06) No interchange
- Data[6] with Data[5] (06,18) interchange
- Data[5] with Data[4] (06,94) interchange
- Data[4] with Data[3] (06,42) interchange
- Data[3] with Data[2] (06,12 interchange
- Data[2] with Data[1] (06,55) interchange
- Data[1] with Data[0] (06,44) interchange
- after the first pass, the smallest element is in its proper positions.
- **06,44,55,12,42,94,18,67**

8.2.1.1 Bubble sort

- Original -> 44,55,12,42,94,18,06,67
- Pass 1 -> **06**,44,55,12,42,94,18,67
- Pass 2 -> **06,12**,44,55,18,42,94,67
- Pass 3 -> **06,12,18**,44,55,42,67,94
- Pass 4 -> **06,12,18,42**,44,55,67,94
- Pass 5 -> **06,12,18,42,44**,55,67,94
- Pass 6 -> **06,12,18,42,44,55**,67,94
- Pass 7 -> **06,12,18,42,44,55,67**,94
- Sorted file->**06,12,18,42,44,55,67,94**

8.2.1.1 Bubble sort

- Java code for bubble sort algorithm

```
static void bubbleSort(int[] arr) {

        int n = arr.length;
        int temp = 0;

         for(int i=0; i < n; i++){
                 for(int j=1; j < (n-i); j++){

                         if(arr[j-1] > arr[j]){
                                 temp = arr[j-1];
                                 arr[j-1] = arr[j];
                                 arr[j] = temp;
                         }

                 }
         }
```

8.2.1.1 Bubble sort

Time complexity analysis cont.

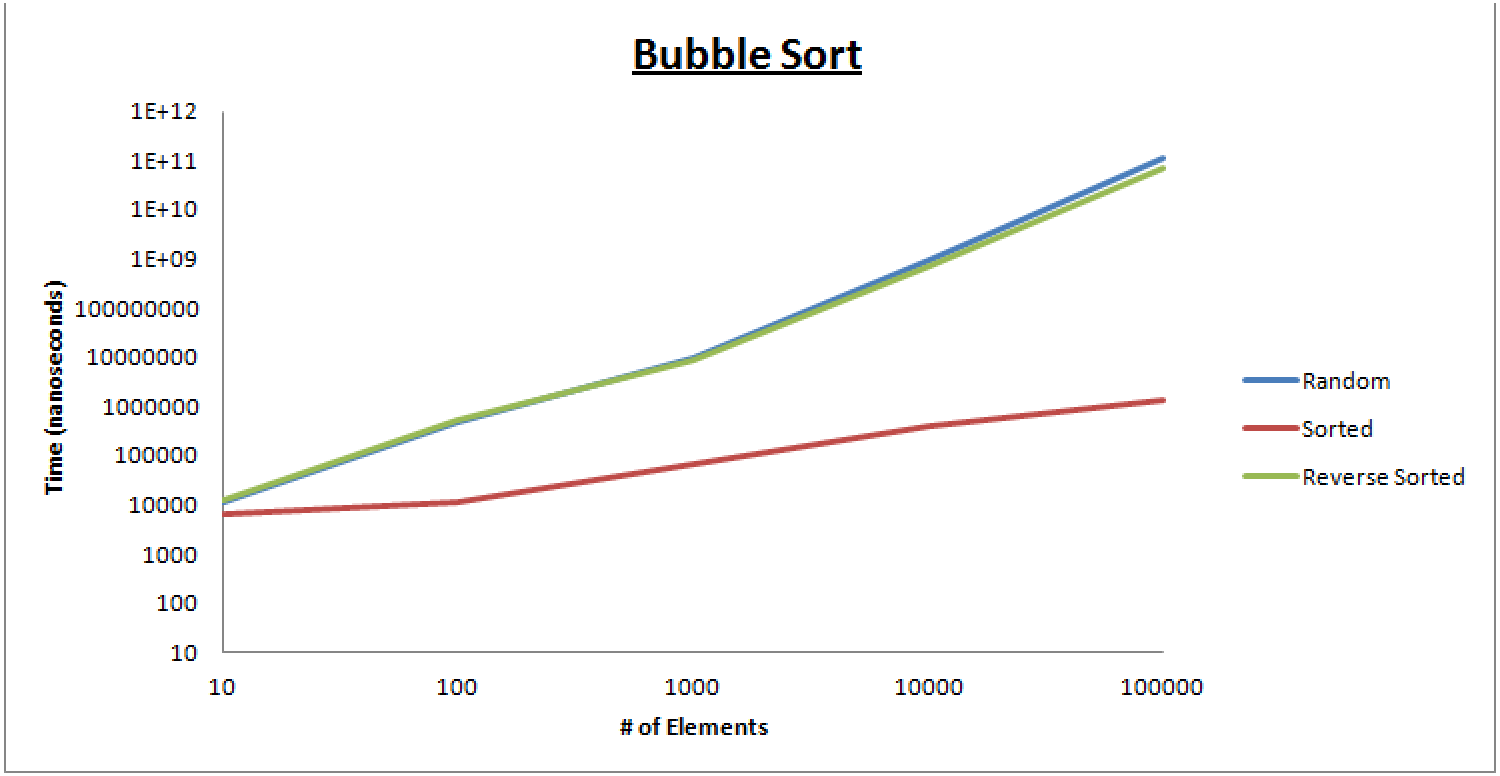
- Bubble sort employs two loops: an inner loop and an outer loop. The inner loop performs O(N) comparisons deterministically. In the worst-case scenario, the outer loop runs O(N) times. As a result, the worst-case time complexity of bubble sort is O(N x N) = $\mathcal{O}(N^2)$
- We need to do N iterations. In each iteration, we do the comparison, and we perform swapping if required. Given an array of size N, the first iteration performs (N - 1) comparisons. The second iteration performs (N - 2) comparisons. In this way, the total number of comparison will be:

$$(N-1)+(N-2)+(N-3)+\ldots\ldots+3+2+1=\frac{N(N-1)}{2}=\mathcal{O}(N^2)$$

- Time complexity of the bubble sort would be $\mathcal{O}(N^2)$

8.2.1.1 Bubble sort

- Analysis of time (nanoseconds) with respect to the number of data elements (n)



8.2.1.2 Selection sort

- In terms of an array A, the selection sort finds the smallest element in the array and exchanges it with A[0]. Then, ignoring A[0], the sort finds the next smallest and swaps it with A[1] and so on.

8.2.1.2 Selection sort

- Scan the list and put the smallest number in the first position.
- Disregard the first position, which is now the smallest number, and put the second smallest number in the second position.
- Proceed in this manner until reaching the end of the list.

- Selection sort is an instance of a sorting by selection category.

8.2.1.2 Selection sort

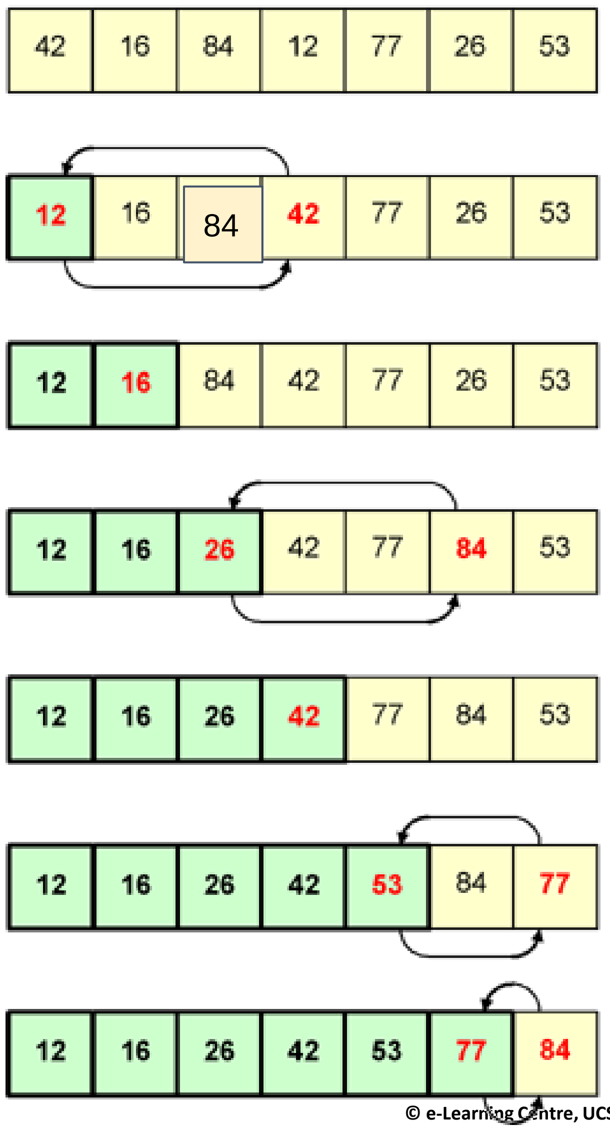
- Following is a demonstration of how Selection sorting works.

Example 01

8.2.1.2 Selection sort

Example 02

- The following is an illustration of how Selection sorting works.



The array, before the selection sort operation begins.

The smallest number (**12**) is swapped into the first element in the structure.

In the data that remains, **16** is the smallest; and it does not need to be moved.

26 is the next smallest number, and it is swapped into the third position.

42 is the next smallest number; it is already in the correct position.

53 is the smallest number in the data that remains; and it is swapped to the appropriate position.

Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*

8.2.1.2 Selection sort

- Pseudocode for the selection sort algorithm

```
Begin SelectionSort(list, n)

  for i = 1 to n - 1
    min = i    (take 1st element of the unsorted list as min)

    for j = i+1 to n (iterate through all the elements of the list)
      if list[j] < list[min] then (if j data item is less than min)
        min = j (update min to j data item)
      end if
    end for

    if indexMin != i  then (if 1st element of the unsorted list and min is different)
      swap list[min] and list[i] (swap 1st element of unsorted list with min)
    end if

  end for
end selectionSort
```

8.2.1.2 Selection sort

Let's consider an array with values (5 1 4 2 8).

- (5 1 4 2 8) –> (1 5 4 2 8)
  - take 1st element of the unsorted list as min = 5
  - find and update the min of the unsorted list = 1
  - swap 1st element of the unsorted list with min
    - Swaps element 5 with element 1
- (1 5 4 2 8) –> (1 2 4 5 8)
  - take 1st element of the unsorted list as min = 5
  - find and update the min of the unsorted list = 2
  - swap 1st element of the unsorted list with min
    - Swaps element 5 with element 2
- (1 2 4 5 8) –> (1 2 4 5 8)
  - take 1st element of the unsorted list as min = 4
  - find and update the min of the unsorted list = 4
  - 1st element and the min is the same
- (1 2 4 5 8) –> (1 2 4 5 8)
  - take 1st element of the unsorted list as min = 5
  - find and update the min of the unsorted list = 5
  - 1st element and the min is the same

End of the iteration

8.2.1.2 Selection sort

- Java code for selection sort algorithm

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n; i++) {
            min_idx = i;
            for (j = i+1; j < n; j++) {
                    if (arr[j] < arr[min_idx] )
                            min_idx = j;
                            }
            int temp = arr[min_dx];
            arr[min_dx] = arr[i];
            arr[i] = temp;
   }
}
```

8.2.1.2 Selection sort

Time complexity analysis

- The implementation has two loops.
- The outer loop which picks the values one by one from the list is executed n times where n is the total number of values in the list.
- The inner loop, which compares the value from the outer loop with the rest of the values, is also executed n times where n is the total number of elements in the list.
- Therefore, the number of executions is (n * n), which can also be expressed as O(n2).

8.2.1.2 Selection sort

Time complexity analysis cont.

```
void selectionSort(int arr[], int n)
{
   Line 1 - int i, j, min_idx;
   Line 2 - for (i = 0; i < n; i++) {
   Line 3 -          min_idx = i;
   Line 4 -          for (j = i+1; j < n; j++) {
   Line 5 -                   if (arr[j] < arr[min_idx] )
   Line 6 -                            min_idx = j;}
   Line 7 -          int temp = arr[min_dx];
   Line 8 -          arr[min_dx] = arr[i];
   Line 9 -          arr[i] = temp;
  }
}
```

8.2.1.2 Selection sort

Time complexity analysis cont.

Line 1: COST = C1, TIME = 1, where C1 is some constant

Line 2: COST = C2, TIME = n+1, where C2 is some constant

Line 3: COST = C3, TIME = n, where C3 is some constant

Line 4: COST = C4, TIME = $(n^2-n) / 2 + n$, where C4 is some constant

Line 5: COST = C5, TIME = $(n^2-n) / 2$, where C5 is some constant

Line 6: COST = C6, TIME = $(n^2-n) / 2$, where C6 is some constant

Line 7: COST = C7, TIME = n, where C7 is some constant

Line 8: COST = C8, TIME = n, where C5 is some constant

Line 9: COST = C9, TIME = n, where C9 is some constant

Runtime = $(C1 *1) + (C2 *(n+1)) + (C3 *n) + (C4 * ((n^2-n)/2) + n) + (C5 * (n^2-n) / 2) + (C6 * (n^2-n) / 2) + (C7 * n)+ (C8 * n)+ (C9 * n)$

Where U,V, and W are constants

$= U +Vn + Wn^2$

$= O(n^2)$

8.2.1.3 Insertion sort

- Insertion Sort orders a list in the same way we would order a hand of playing cards.
- Compare the first two numbers, placing the smallest one in the first position.
- Compare the third number to the second number. If the third number is larger, then the first three numbers are in order. If not, then swap them. Now compare the numbers in positions one and two and swap them if necessary.
- Proceed in this manner until reaching the end of the list.

- Insertion sort is an instance of a sorting by insertion category.

8.2.1.3 Insertion sort

- Following is a demonstration of how Insertion sorting works.

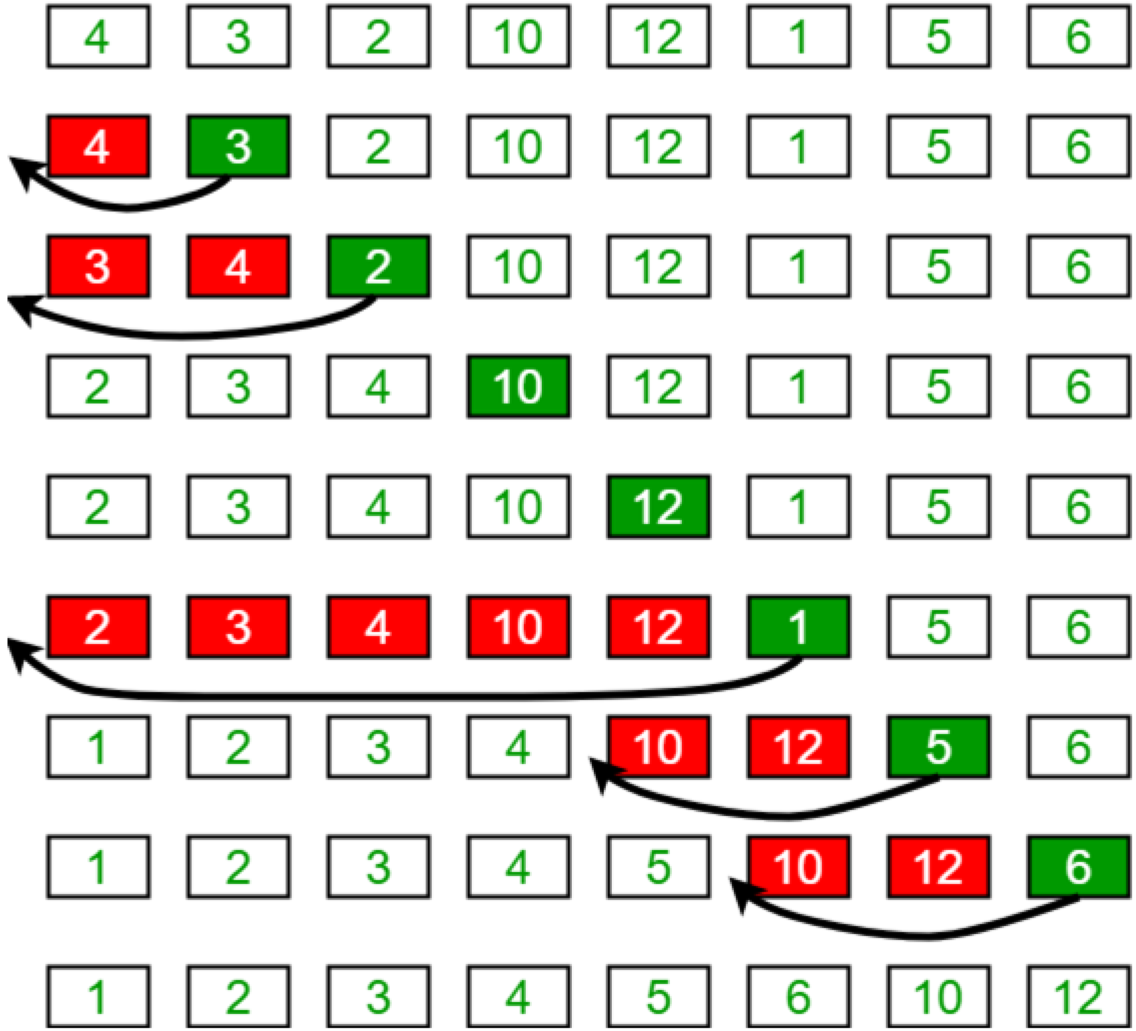
Example 01

6 5 3 1 8 7 2 4

8.2.1.3 Insertion sort

Example 02

- Following is a illustration of how Insertion sorting works.



8.2.1.3 Insertion sort

- Pseudocode for insertion sort algorithm

```
Begin insertionSort(A)

  for i = 1 to n
        key ← A [i]
        j ← i – 1

         while j > = 0 and A[j] > key (compare the adjacent elements and if not in the correct order)
               A[j+1] ← A[j] (replace j+1 indexed element with j indexed element)
               j ← j – 1
        End while

        A[j+1] ← key (replace j+1 indexed element with 'key' element)

  End for

End insertionSort
```

8.2.1.3 Insertion sort

Let's consider an array with values (5 1 4 2 8).

- (5 1 4 2 8) –> (1 5 4 2 8)
  - compare first two elements of the list (5>1)
  - not in the correct order
    - remove lower value element (1) and insert into the correct order position (1, 5)
- (1 5 4 2 8) –> (1 4 5 2 8)
  - compare next two elements of the list (5>4)
  - not in the correct order
    - remove lower value element (4) and insert into the correct order position (1, 4, 5)
- (1 4 5 2 8) –> (1 2 4 5 8)
  - compare next two elements of the list (5>2)
  - not in the correct order
    - remove lower value element (2) and insert into the correct order position (1, 2, 4, 5)
- (1 2 4 5 8) –> (1 2 4 5 8)
  - compare next two elements of the list (8>5)
  - elements are in correct order

End of the iteration

8.2.1.3 Insertion sort

- Java Code for insertion sort algorithm

```
public static void insertionSort(int array[]) {
        int n = array.length;

        for (int i = 1; i < n; i++) {
            int key = array[i];
            int j = i-1;

            while ( (j > -1) && ( array [j] > key ) ) {
                array [j+1] = array [j];
                j--;
            }

            array[j+1] = key;
        }
```

8.2.1.3 Insertion sort

Time complexity analysis

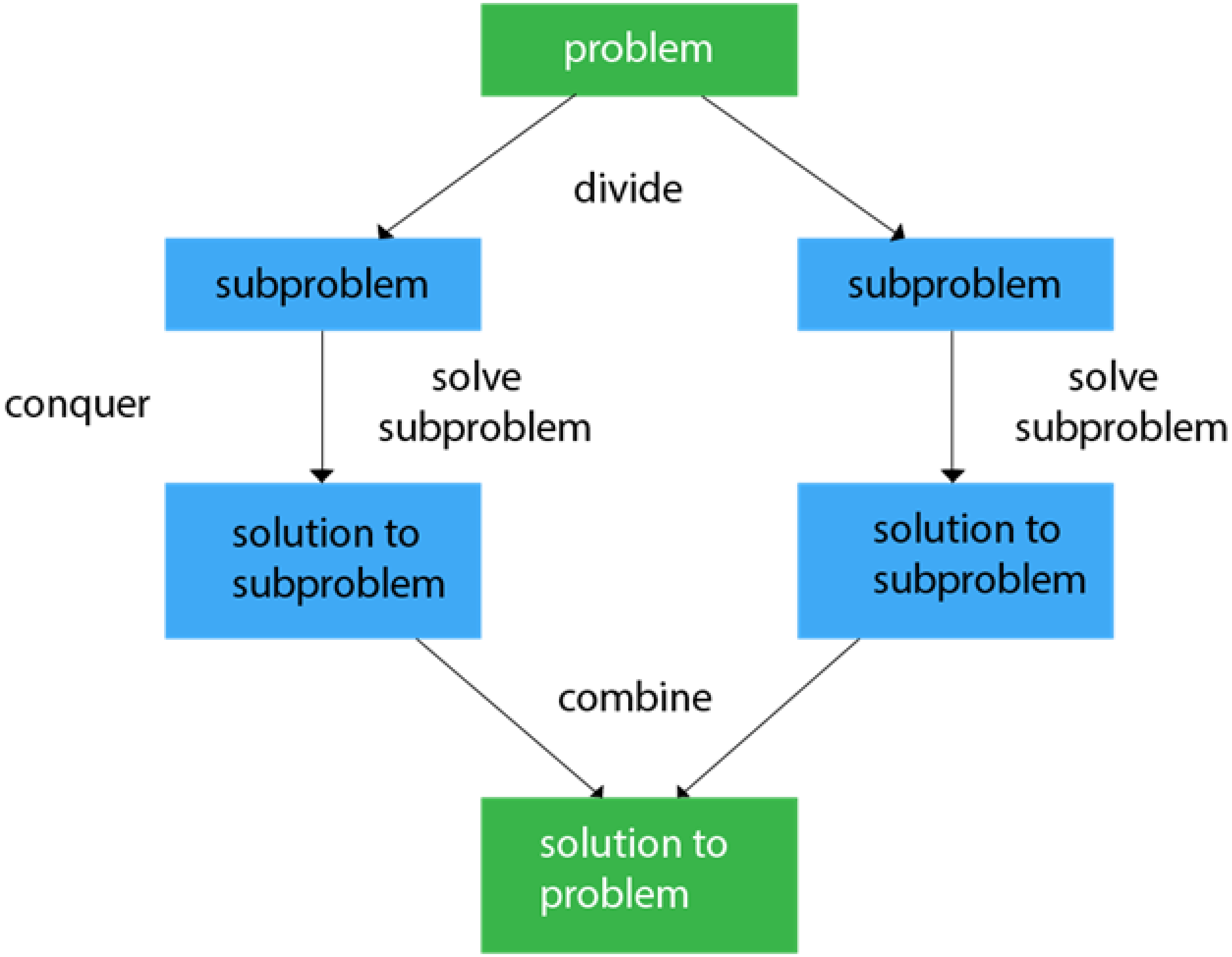
- The two nested loops are an indication that we are dealing with quadratic effort, meaning with time complexity of $O(n^2)$.
- This is the case if both the outer and the inner loop count up to a value that increases linearly with the number of elements.

8.2.2 Divide and Conquer Method

- In divide and conquer method, the original problem is divided into smaller subproblems and find the solutions for those subproblems.
- Then the solutions are combined to find the solution for the original problem.
- The divide-and-conquer paradigm involves three steps at each level of the recursion:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.

8.2.2 Divide and Conquer Method

- Following is a demonstration of how divide and conquer method works.



8.2.2.1 Merge sort

- *Merge sort* takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(*n* log *n*).

8.2.2.1 Merge sort

- Following is a demonstration of how merge sorting works.

Example 01

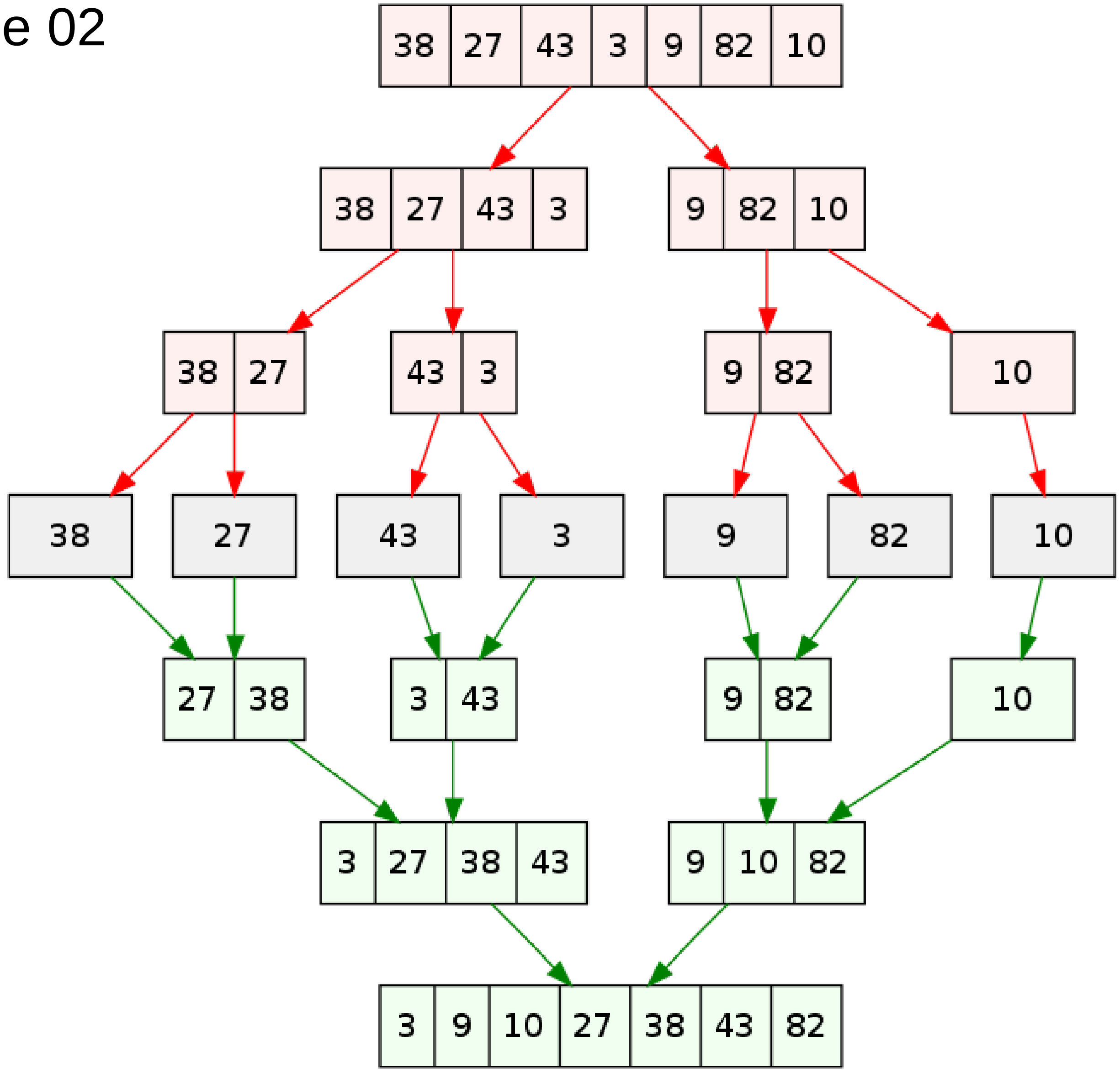
6 5 3 1 8 7 2 4

https://en.wikipedia.org/wiki/Merge_sort

8.2.2.1 Merge sort

- Following is a illustration of how merge sorting works.

Example 02



8.2.2.1 Merge sort

- Pseudocode for merge sort algorithm

```
function mergeSort( array[] data ):

    // base case:
    if len(data) < 2:
        return data

    // recursive case:

    mid_index = floor( length(data)/2 )

    left_half = data[:mid_index]
    right_half = data[mid_index:]

    left_half = mergeSort(left_half)
    right_half = mergeSort(right_half)

    merge(left_half, right_half, data)

    return data
}
```

8.2.2.1 Merge sort

- Java code for merge sort algorithm

```
public static void mergeSort(int[] a, int n) {
    if (n < 2) {
        return;
    }
    int mid = n / 2;
    int[] l = new int[mid];
    int[] r = new int[n - mid];

    for (int i = 0; i < mid; i++) {
        l[i] = a[i];
    }
    for (int i = mid; i < n; i++) {
        r[i - mid] = a[i];
    }
    mergeSort(l, mid);
    mergeSort(r, n - mid);

    merge(a, l, r, mid, n - mid);
}
```

8.2.2.1 Merge sort

Time complexity analysis

- To merge the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.
- Hence the total time for merge sort will become n(log n + 1), which gives us a time complexity of O(n*log n).

8.2.2.2 Quick sort

- Choose an element out of the list as a pivot. A good process to select a pivot is to compare the first, middle, and last elements and choose the middle value.
- Compare every other element in the list to the pivot and create two lists, one list where every element is smaller than the pivot and one where every element is larger.
- Now split each of these lists into smaller lists.
- Continue in this way until the small lists have only one or two elements and we can sort them with at most one comparison each.

8.2.2.2 Quick sort

- Quick sort is one of the fastest sorting by exchange algorithm.
- Given an array of n elements, in quick sort:

  If array only contains one element,

  - return array.

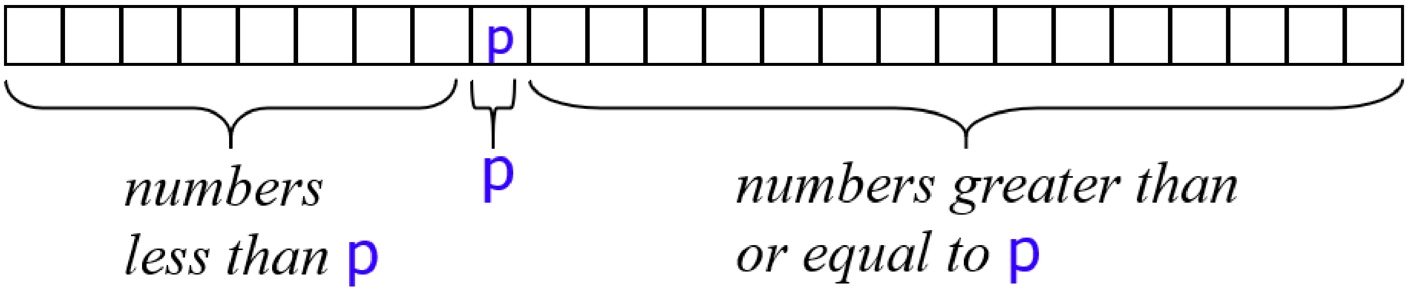
  Else

    - Pick one element to use as pivot.
    - Partition elements into two sub-arrays:
      - Elements less than or equal to pivot
      - Elements greater than pivot
    - Quicksort two sub-arrays
    - Return results

8.2.2.2 Quick sort

Partitioning - quick sort algorithm

- Choose any number from data elements to use it as a pivot (p) to partition the elements of the array such that the resulting array consists of:
  - One sub-array that contains elements >= pivot
  - Another sub-array that contains elements < pivot



8.2.2.2 Quick sort

There are many ways to pick the pivot value:

- Always pick first element as pivot.
- Always pick last element as pivot.
- Pick a random element as pivot.
- Pick median as pivot.

8.2.2.2 Quick sort

Following are the basic steps in quick sort algorithm

- Step 1 − Pick the pivot value
- Step 2 − partition the array using pivot value
- Step 3 − quicksort left partition recursively
- Step 4 − quicksort right partition recursively

8.2.2.2 Quick sort

- Following is a demonstration of how quick sorting works.

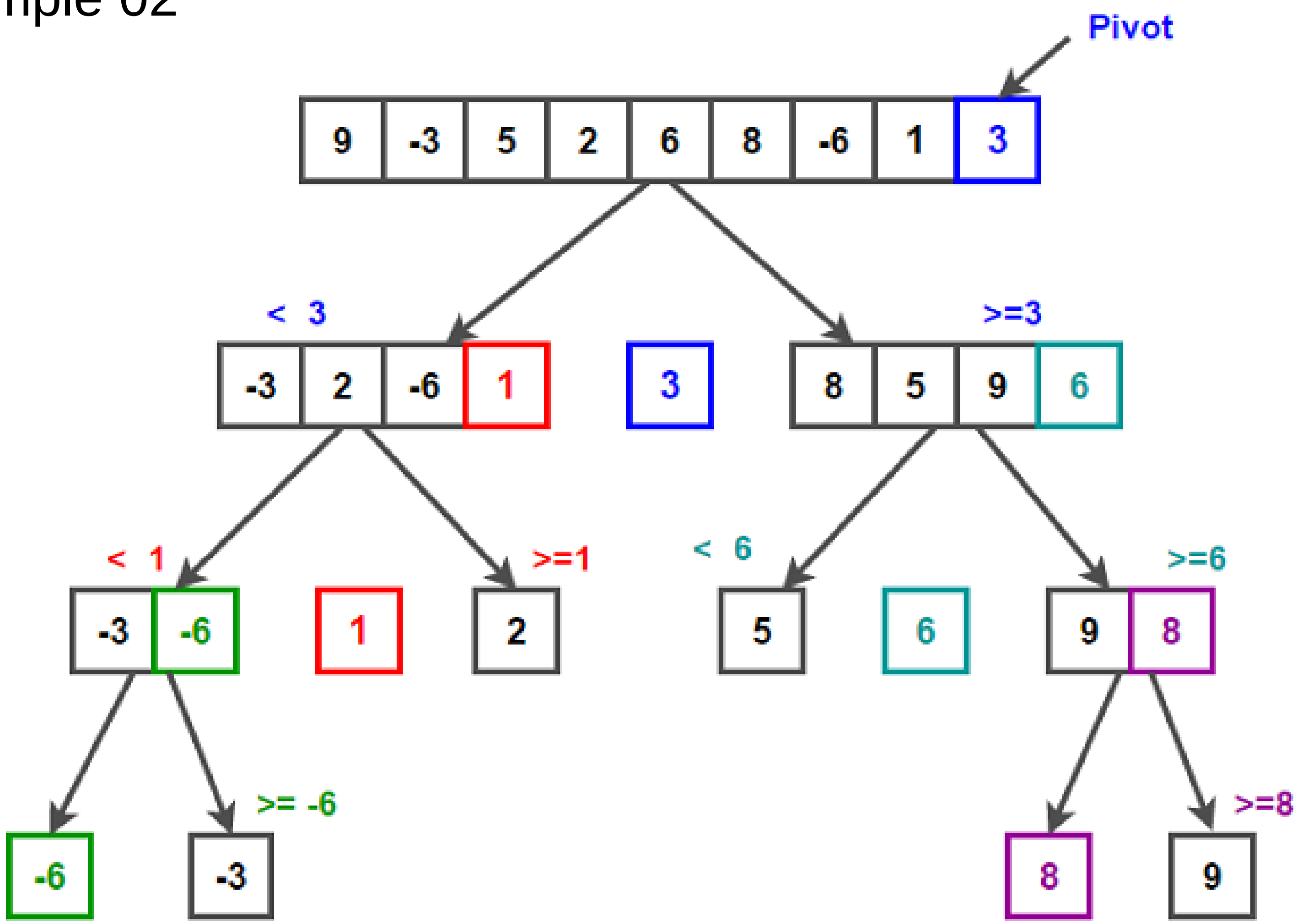
Example 01

6 5 3 1 8 7 2 4

8.2.2.2 Quick sort

- Following is a illustration of how quick sorting works.

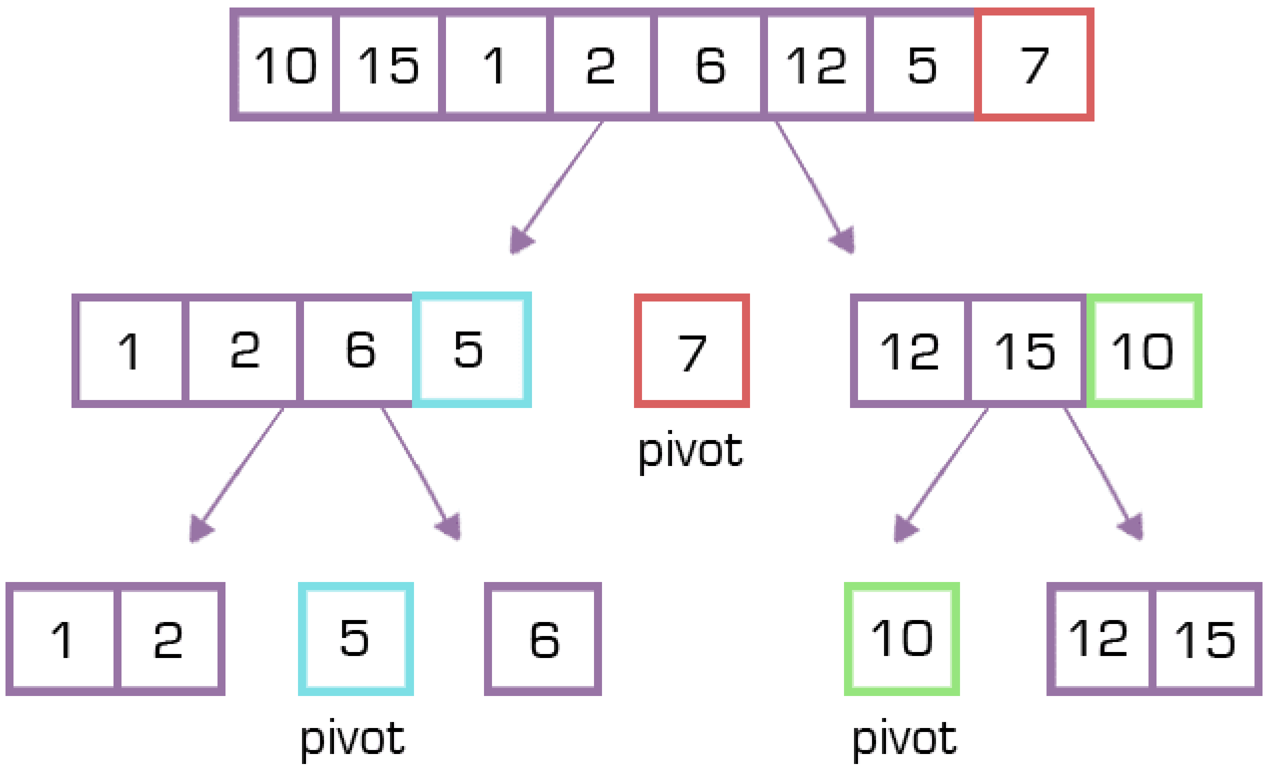
Example 02



8.2.2.2 Quick sort

- Following is a illustration of how quick sorting works.

Example 03



8.2.2.2 Quick sort

- Pseudocode for quick sort algorithm - partitioning function

```
partition(arr, beg, end)
  set end as pivotIndex
  pIndex = beg - 1

  for i = beg to end-1
        if arr[i] < pivot
          swap arr[i] and arr[pIndex]
          pIndex++

  swap pivot and arr[pIndex+1]
return pIndex +
```

8.2.2.2 Quick sort

- Pseudocode for quick sort algorithm - sorting function

```
quickSort(arr, beg, end)
  if (beg < end)
    pivotIndex = partition(arr,beg, end)
    quickSort(arr, beg, pivotIndex)
    quickSort(arr, pivotIndex + 1, end)
```

8.2.2.2 Quick sort

- Java code for quick sort algorithm - partitioning function

```
static int partition(int[] arr, int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for(int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

8.2.2.2 Quick sort

- Java code for quick sort algorithm - sorting function

```
static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

8.2.2.2 Quick sort

Time complexity analysis

- For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.
- And if keep on getting unbalanced subarrays, then the running time is the worst case, which is O(n2).
- Where as if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as O(n*log n).

8.2.2.3 Radix Sort

- *Radix sort* is an algorithm that sorts a list of fixed-size numbers of length k in $O(n \cdot k)$ time by treating them as bit strings. We first sort the list by the least significant bit while preserving their relative order using a stable sort. Then we sort them by the next bit, and so on from right to left, and the list will end up sorted. Most often, the counting sort algorithm is used to accomplish the bitwise sorting, since the number of values a bit can have is small.

8.2.2.3 Radix Sort

- Suppose, there is an array of 6 elements.
- First, the algorithm sort the elements based on the value of the unit place.
- Then, sort the elements based on the value of the tenth place. Next, the hundred place.
- The comparisons are made among the digits of the number from LSB to MSB.

8.2.2.3 Radix sort

- Following is a demonstration of how radix sorting works using binary numbers.

Example 01 - (2 0 5 1 7 3 4 6)

2	0 1 0	0 1 0	0 0 0	0 0 0	0
0	0 0 0	0 0 0	1 0 0	0 0 1	1
5	1 0 1	1 0 0	1 0 1	0 1 0	2
1	0 0 1	1 1 0	0 0 1	0 1 1	3
7	1 1 1	1 0 1	0 1 0	1 0 0	4
3	0 1 1	0 0 1	1 1 0	1 0 1	5
4	1 0 0	1 1 1	1 1 1	1 1 0	6
6	1 1 0	0 1 1	0 1 1	1 1 1	7

input element list

binary representation of elements

sort the elements based on LSB to MSB

final element list - sorted

8.2.2.3 Radix sort

- Following is a demonstration of how radix sorting works using decimal numbers

Example 02 - (32 224 16 15 31 169 123 252)

0 3 2	0 3 1	0 1 5	0 1 5
2 2 4	0 3 2	0 1 6	0 1 6
0 1 6	2 5 2	1 2 3	0 3 1
0 1 5	1 2 3	2 2 4	0 3 2
0 3 1	2 2 4	0 3 1	1 2 3
1 6 9	0 1 5	0 3 2	1 6 9
1 2 3	0 1 6	2 5 2	2 2 4
2 5 2	1 6 9	1 6 9	2 5 2

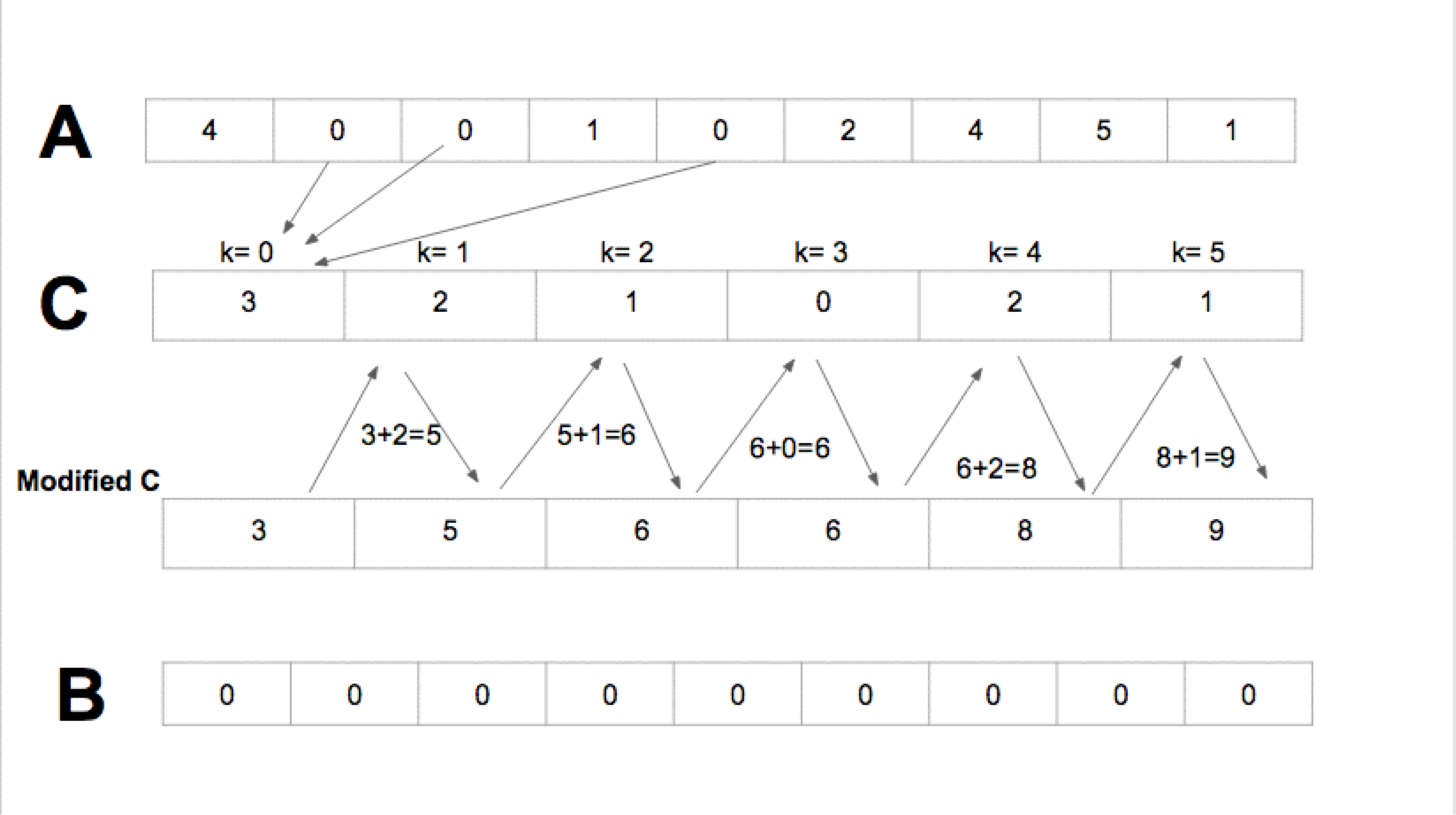
input element list

sort the elements based on LSB to MSB

8.2.2.3 Radix sort

- Following is a demonstration of how radix sorting works.

Example 03



8.2.2.3 Radix sort

- Pseudocode for radix sort algorithm

```
function radixSort (L: list of unsorted items)
   largest := max(L)
   exponent := floor(log10(largest))
   bucket := empty list
   index := 0

   for i to exponent+1 do:
     bucket := empty list

     for j to length(L) do:
       number := L[j]
      digit := i+1

      while(digit--) do:
        temp := number % 10
```

8.2.2.3 Radix sort

- Pseudocode for radix sort algorithm cont.

```
        number := floor(number-temp/10)
      digit := temp
      if bucket[digit] exists then
        bucket[digit] := bucket[digit]
      if bucket[digit] is undefined then
        bucket[digit] := empty list
      add L[j] to bucket[digit]
    reset index to 0
    for digit to length(bucket) do:
      if bucket[digit] exists then
        for j to length(bucket[digit])
          L[index++] := bucket[digit][j]
  return L
end-function
```

8.2.2.3 Radix sort

Time complexity analysis

- The complexity is $O((n+b)* \log_b(maxx))$ where b is the base for representing numbers and maxx is the maximum element of the input array.
- If $maxx <= n^c$,then the complexity can be written as $O(n*\log_b (n))$.

8.2.2.4 Heap Sort

- *Heapsort* is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes $O(\log n)$ time, instead of $O(n)$ for a linear scan as in simple selection sort. This allows Heapsort to run in $O(n \log n)$ time.

8.2.2.4 Heap Sort

- A heap is a complete binary tree.
- Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array.
- Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.
- Heap sort basically recursively performs two main operations:
  - Build a heap H, using the elements of array.
  - Repeatedly delete the root element of the heap formed in 1st phase.

8.2.2.4 Heap sort

- Following is a demonstration of how heap sorting works.

Example 01

10	4	8	5	12	2	6	11	3	9	7	1

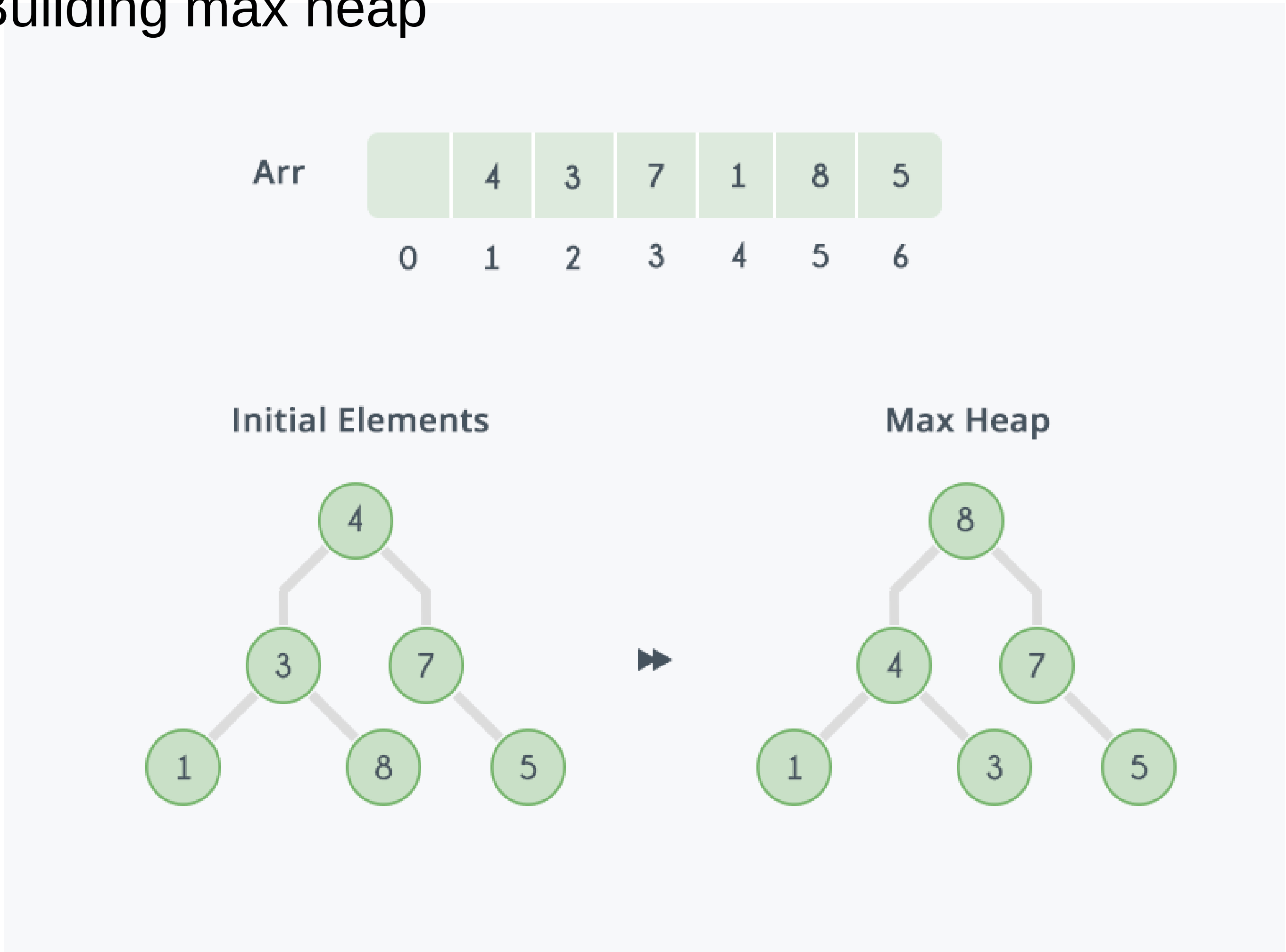
https://commons.wikimedia.org/wiki/File:Heap_sort_example.gif

8.2.2.4 Heap sort

- Following is a illustration of how heap sorting works.

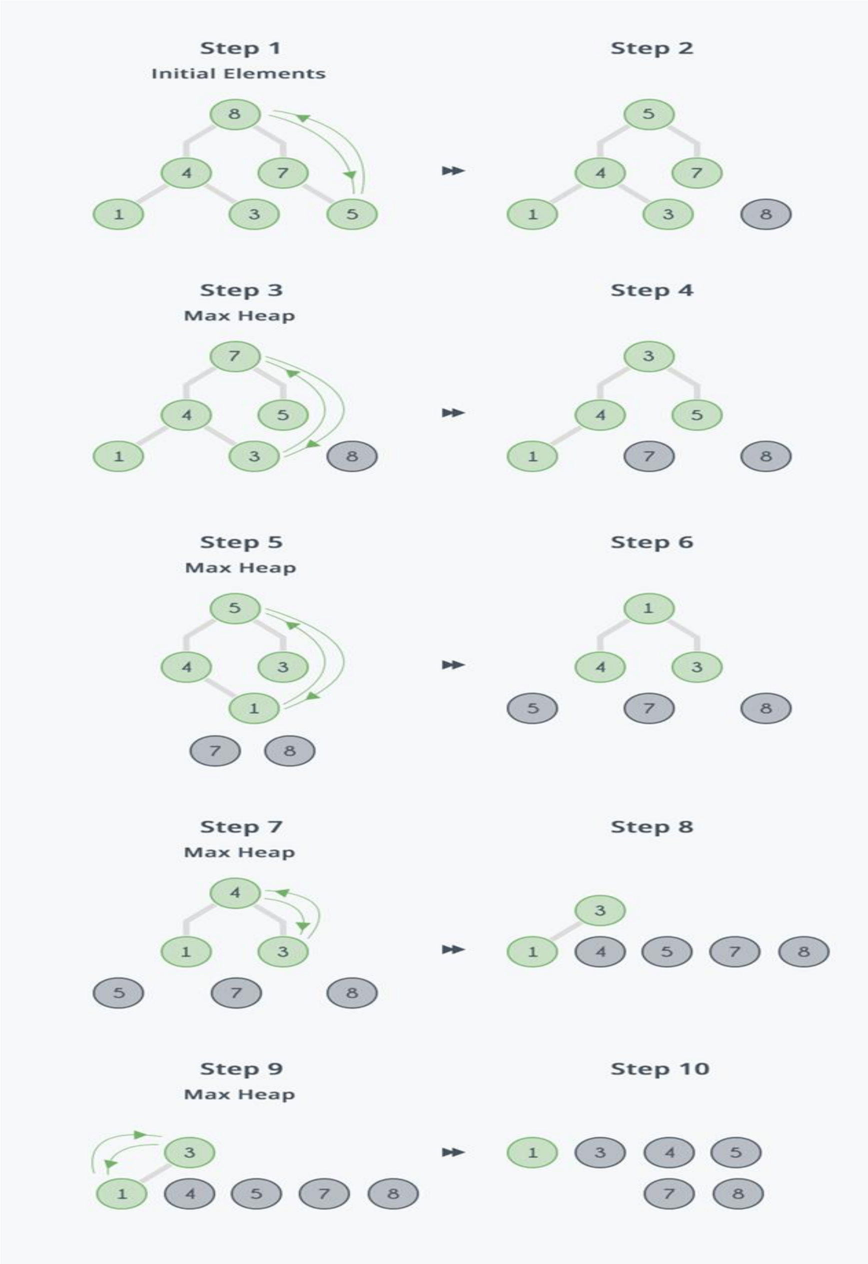
Example 02 - (4 3 7 1 8 5)

- Building max heap



8.2.2.4 Heap sort

- Sorting the array



8.2.2.4 Heap Sort

- Pseudocode for heap sort algorithm

```
def heap_sort(array):
    length = len(array)
    array = build_heap(array)

    for i in range(length-1, 0, -1):
        largest = array[0]
        array[0] = array[i]
        array[i] = largest

        heapify(array[:i], 0)

    return array
```

8.2.2.4 Heap Sort

- Java code for heap sort algorithm

```
public void Heapsort(int arr[])
   {
      int n = arr.length;

      for (int i = n / 2 - 1; i >= 0; i--)
         heapify(arr, n, i);

      for (int i = n - 1; i > 0; i--) {
         int temp = arr[0];
         arr[0] = arr[i];
         arr[i] = temp;

         heapify(arr, i, 0);
      }
   }
```

8.2.2.4 Heap sort

Time complexity analysis

- heapsort runs in O(N*logN) time. Although it may be slightly slower than quicksort, an advantage over quicksort is that it is less sensitive to the initial distribution of data.
- Certain arrangements of key values can reduce quicksort to slow O(N2) time, whereas heapsort runs in O(N*logN) time no matter how the data is distributed.

8.3 Searching algorithms

- Searching algorithms are closely related to the concept of dictionaries.
- Dictionaries are data structures that support search, insert, and delete operations.
- One of the most effective representations is a hash table. Typically, a simple function is applied to the key to determine its place in the dictionary. Another efficient search algorithms on sorted tables is binary search.
- If the dictionary is not sorted then heuristic methods of dynamic reorganization of the dictionary are of great value. One of the simplest are cache-based methods: several recently used keys are stored a special data structure that permits fast search (for example is always sorted). For example keeping the last N recently found values at the top of the table (or list) dramatically improved performance. Other cache-based approach are also possible. In the simplest form the cache can be merged with the dictionary:
  - **move-to-front method** : A *heuristic* that moves the target of a *search* to the *head* of a *list* so it is found faster next time.
  - **transposition method** : Search an *array* or *list* by checking items one at a time. If the value is found, swap it with its predecessor so it is found faster next time.

8.3.1 Linear search

- When the input array is not sorted, we have little choice but to do a linear sequential search, which steps through the array sequentially until a match is found.
- The complexity of the algorithm is analyzed in three ways.
  - First, we provide the cost of an unsuccessful search.
  - Then, we give the worst-case cost of a successful search.
  - Finally, we find the average cost of a successful search.
- Analyzing successful and unsuccessful searches separately is typical.
- Unsuccessful searches usually are more time consuming than are successful searches.
- For sequential searching, the analysis is straightforward.

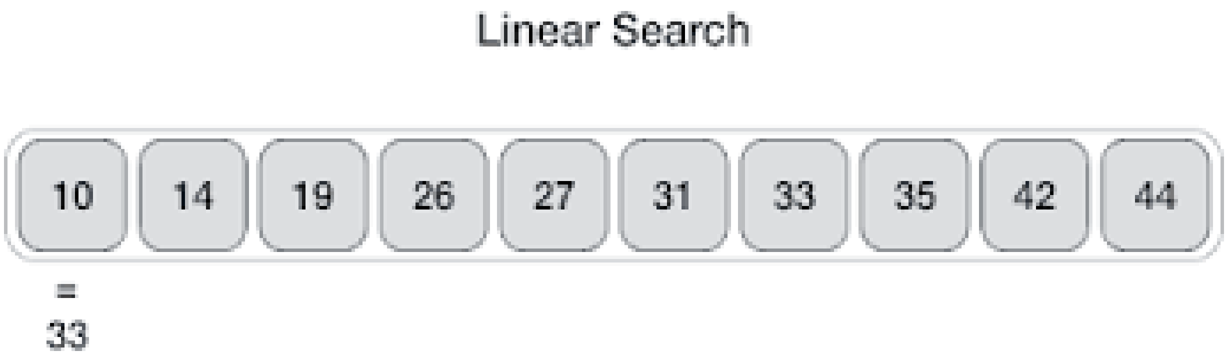
8.3.1 Linear search

- An unsuccessful search requires the examination of every item in the array, so the time will be O(N) (N is the number of items in the array).
- In the worst case, a successful search, too, requires the examination of every item in the array because we might not find a match until the last item.
- Thus the worst-case running time for a successful search is also linear.
- On average, however, we search only half of the array.
- That is, for every successful search in position *i*, there is a corresponding successful search in position N-1-i (assuming we start numbering from 0).
- However, N/2 is still O(N).

8.3.1 Linear search

- Following is a demonstration of how linear search works.

Example 01



https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm

8.3.2 Binary search

- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

8.3.2 Binary search

- Following is a demonstration of how binary search works.

Example 01

Search for 47

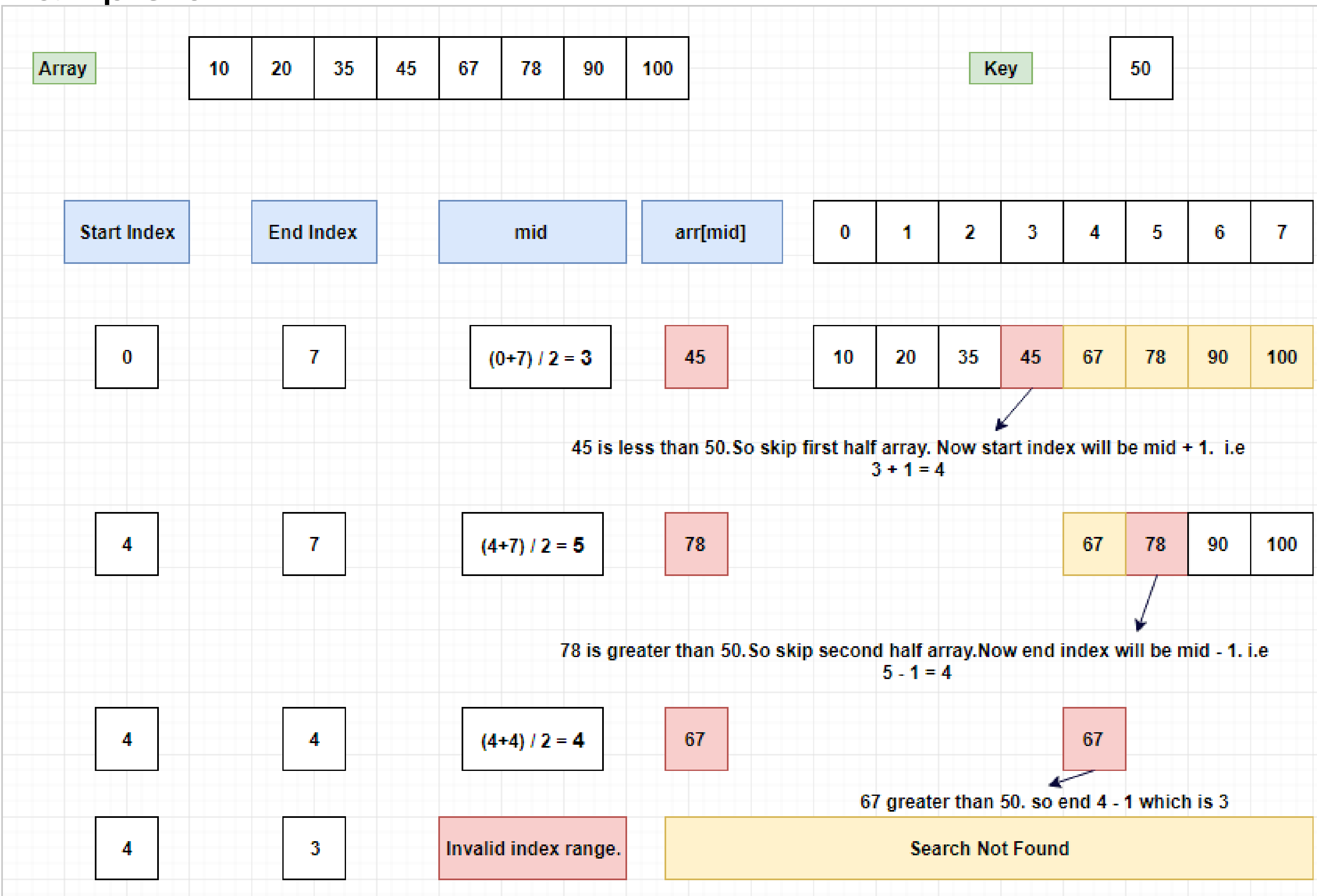
0	4	7	10	14	23	45	47	53

https://brilliant.org/wiki/binary-search/

8.3.2 Binary search

- Following is a illustration of how binary search works.

Example 02



8.3.2 Binary search

- Pseudocode for binary search - iterative approach

```
binarySearch(arr, size)
  loop until beg is not equal to end
        midIndex = (beg + end)/2
        if (item == arr[midIndex] )
            return midIndex
        else if (item > arr[midIndex] )
            beg = midIndex + 1
        else
            end = midIndex - 1
```

8.3.2 Binary search

- Pseudocode for binary search - recursive approach

```
binarySearch(arr, item, beg, end)
    if beg<=end
        midIndex = (beg + end) / 2
        if item == arr[midIndex]
            return midIndex
        else if item < arr[midIndex]
            return binarySearch(arr, item, midIndex + 1, end)
        else
            return binarySearch(arr, item, beg, midIndex - 1)

    return -1
```

8.3.2 Binary search

- Java code for binary search - iterative approach

```
public static void binarySearch(int arr[], int first, int last, int key){
  int mid = (first + last)/2;
  while( first <= last ){
    if ( arr[mid] < key ){
      first = mid + 1;
    }else if ( arr[mid] == key ){
      System.out.println("Element is found at index: " + mid);
      break;
    }else{
       last = mid - 1;
    }
    mid = (first + last)/2;
  }
  if ( first > last ){
    System.out.println("Element is not found!");
  }
}
```

8.3.2 Binary search

- Java code for binary search - recursive approach

```
public static int binarySearch(int arr[], int first, int last, int key){
    if (last>=first){
        int mid = first + (last - first)/2;
        if (arr[mid] == key){
        return mid;
        }
        if (arr[mid] > key){
        return binarySearch(arr, first, mid-1, key);//search in left subarray
        }else{
        return binarySearch(arr, mid+1, last, key);//search in right subarray
        }
    }
    return -1;
  }
```

8.3.2 Binary search

Time complexity analysis

- The time complexity of the binary search algorithm is O(log n).
- The best-case time complexity would be O(1) when the central index would directly match the desired value.
- The worst-case scenario could be the values at either extremity of the list or values not in the list.

8.3.3 Interpolation search

- The Interpolation Search is an improvement over Binary Search.
- Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.
- If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method,

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

A = list

Lo = Lowest index of the list

Hi = Highest index of the list

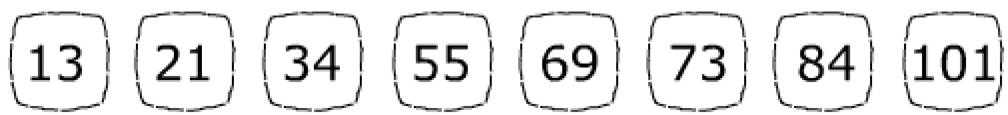
A[n] = Value stored at index n in the list

8.3.3 Interpolation search

- Following is a illustration of how interpolation search works.

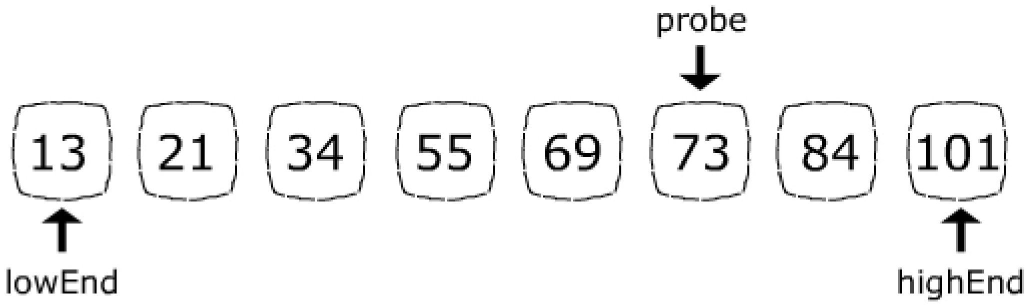
Example 01

Suppose we want search number 84 in the following array

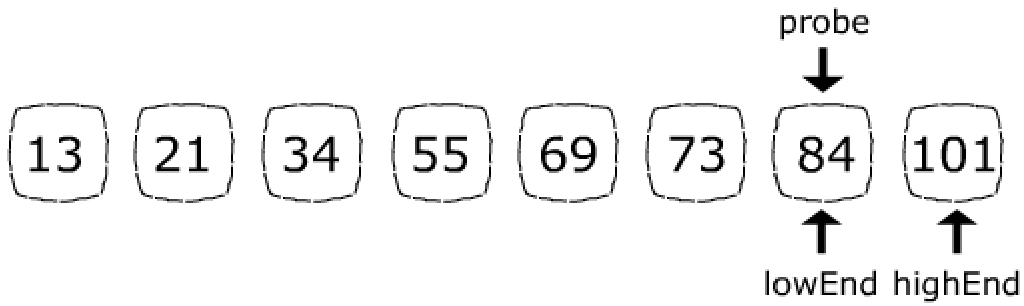


8.3.3 Interpolation search

- The array's length is 8, so initially Hi = 7 and Lo = 0
- In the first step, the probe position formula will result in probe = 5:



8.3.3 Interpolation search



- Because 84 is greater than 73 (current probe), the next step will abandon the left side of the array by assigning Lo= probe + 1.
- Now the search space consists of only 84 and 101. The probe position formula will set probe = 6 which is exactly the 84's index:
- Since the target (84) is found, index 6 will return.

8.3.3 Interpolation search

- Pseudocode for interpolation search

```
A → Array list
N → Size of A
X → Target Value

Procedure Interpolation_Search()

 Set Lo  →  0
 Set Mid → -1
 Set Hi  →  N-1

 While X does not match

   if Lo equals to Hi OR A[Lo] equals to A[Hi]
     EXIT: Failure, Target not found
   end if
```

8.3.3 Interpolation search

- Pseudocode for interpolation search cont.

```
Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

    if A[Mid] = X
      EXIT: Success, Target found at Mid
    else
      if A[Mid] < X
        Set Lo to Mid+1
      else if A[Mid] > X
        Set Hi to Mid-1
      end if
    end if
  End While

End Procedure
```

8.3.3 Interpolation search

- Java code for interpolation search

```
public static int interpolationSearch(int arr[], int lo,
                                      int hi, int x)
  {
    int pos;

    // Since array is sorted, an element
    // present in array must be in range
    // defined by corner
    if (lo <= hi && x >= arr[lo] && x <= arr[hi]) {

      // Probing the position with keeping
      // uniform distribution in mind.
      pos = lo
            + (((hi - lo) / (arr[hi] - arr[lo]))
               * (x - arr[lo]));
```

8.3.3 Interpolation search

- Java code for interpolation search cont.

```
        // Condition of target found
        if (arr[pos] == x)
            return pos;

        // If x is larger, x is in right sub array
        if (arr[pos] < x)
            return interpolationSearch(arr, pos + 1, hi, x);

        // If x is smaller, x is in left sub array
        if (arr[pos] > x)
            return interpolationSearch(arr, lo, pos - 1, x);
    }
    return -1;
}
```

8.3.3 Interpolation search

- A static searching method that is sometimes faster, however, is an interpolation search, which has better Big-Oh performance on average than binary search but has limited practicality and a bad worst case.
- For an interpolation search to be practical, two assumptions must be satisfied:

    1. Each access must be very expensive compared to a typical instruction.

        For example, the array might be on a disk instead of in memory, and each comparison requires a disk access.

    2. The data must not only be sorted, it must also be fairly uniformly distributed.

        For example, a phone book is fairly uniformly distributed. If the input items are {1, 2, 4, 8, 16, ... }, the distribution is not uniform.

8.3.3 Interpolation search

- The interpolation search requires that we spend more time to make an accurate guess regarding where the item might be. The binary search always uses the midpoint.
- Interpolation search has a better Big-Oh bound on average than does binary search, but has limited practicality and a bad worst case.

Summary

8.2.1	The bubble sort is the least efficient, but the simplest, sort.
8.2.2	The radix sort is about as fast as quicksort but uses twice as much memory.
8.3	Running time of binary search algorithm is O(log n)