Searching and Sorting Algorithms

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

Today Objectives

- Introduce searching and sorting algorithms
- Describe how to perform case analysis for searching and sorting algorithms.
- Describe the efficiency of sorting and searching algorithms.

Searching

Searching

- Searching is a common task in computer programming.
- Searching is the process of looking for a specific element in a database.

Searching

Context

- In this class, we will study searching algorithms and perform demos for numerical arrays.
- Many algorithms and data structures are devoted to searching but, we will study only two approaches: **linear search** and **binary search**.

Linear Search

10	5	1	9	13	2	25	41	2	13
25									

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

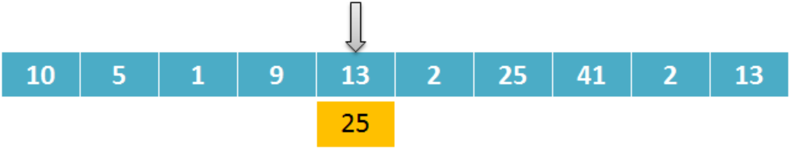
- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

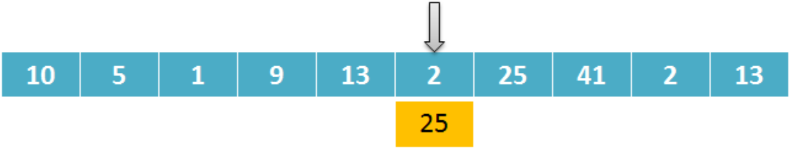
Linear Search



Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search



Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Linear Search

- compare the **key** element with each element in the array or the list,
- continue the process until the key matches an element in the list or the list is exhausted without a match being found
- if a match is made, it returns the **index** of the element in the array that matches the key. If it is not the case, it returns -1.

Linear Search

Iterative linear search:

- For each item in the list:
  - if that item has the desired value,
    - stop the search and return the item's location.
- return not found.

Linear Search

Iterative linear search:

- For each item in the list:
  - if that item has the desired value,
    - stop the search and return the item's location.
- return not found.

Recursive linear search RecurSearch(value, list):

- if the list is empty, return not found;
- else,
  - if that item has the desired value,
    - stop the search and return the item's location.
  - else
    - return RecurSearch(value, remainder of the list)

Linear Search

```
int search (int a[], int x, int n) {
    for (int i=0; i<n; i++)
        if (a[i] == x)   return i;
    return -1;
}
```

Linear Search

```
int search (int a[], int x, int n) {
    for (int i=0; i<n; i++)
        if (a[i] == x)   return i;
    return -1;
}
```

```
int search (int a[], int x) {
    if (isempty(a)) return -1;
    else

        if (a[1] == x)  return 1;
        else
           return search(remain(a,1),x);
}
```

Linear Search

- **In the worst scenario**, we have to search all the elements in the array. If there are n elements in the array, we need n operations.
- **In the best scenario search**, we need only one operation to find the key element. The first element in the array matches the key.
- **In average**, we need $\frac{n}{2}$ operations (if middle of the array)to finish the searching process.

Linear Search is not really efficient. Binary Search is a better option for searching arrays.

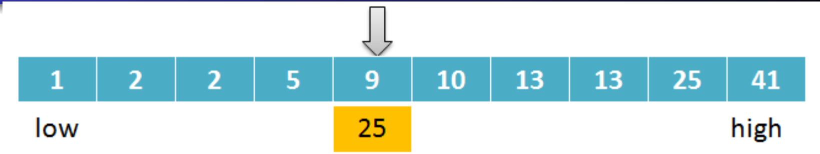
Binary Search

1	2	2	5	9	10	13	13	25	41
25									

Binary Search

- The array is supposed to be sorted beforehand.
- A binary search begins by comparing **the middle element** of the array with the key element. If a match is made, it returns the value.
- If the key value is less or more than the middle element,
  - the search continues the lower or upper half of the array respectively.
  - a new middle element is selected while eliminating the other half from consideration.

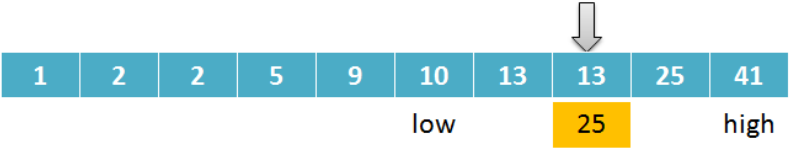
Binary Search



Binary Search

- The array is supposed to be sorted beforehand.
- A binary search begins by comparing **the middle element** of the array with the key element. If a match is made, it returns the value.
- If the key value is less or more than the middle element,
  - the search continues the lower or upper half of the array respectively.
  - a new middle element is selected while eliminating the other half from consideration.

Binary Search



Binary Search

- The array is supposed to be sorted beforehand.
- A binary search begins by comparing **the middle element** of the array with the key element. If a match is made, it returns the value.
- If the key value is less or more than the middle element,
  - the search continues the lower or upper half of the array respectively.
  - a new middle element is selected while eliminating the other half from consideration.

Binary Search

Binary Search

- The array is supposed to be sorted beforehand.
- A binary search begins by comparing **the middle element** of the array with the key element. If a match is made, it returns the value.
- If the key value is less or more than the middle element,
  - the search continues the lower or upper half of the array respectively.
  - a new middle element is selected while eliminating the other half from consideration.

Binary Search

Binary Search

- The array is supposed to be sorted beforehand.
- A binary search begins by comparing **the middle element** of the array with the key element. If a match is made, it returns the value.
- If the key value is less or more than the middle element,
  - the search continues the lower or upper half of the array respectively.
  - a new middle element is selected while eliminating the other half from consideration.

Binary Search

Iterative binary search

```
int bsearch(int a[], int sz, int x){
  int low = 0, high = sz -1;
  while(low <= high) {
    int mid = (low+high)/2;
    if(x < a[mid])
      high = mid - 1;
    else if(x > a[mid])
      low = mid + 1;
    else
      return a[mid];
  }
  return -1;
}
```

Binary Search

Recursive binary search

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

Linear Search vs Binary Search

- Binary search is more efficient. The complexity of linear search is $O(n)$ while the complexity of binary search is $O(logn)$.
- If we have 1 billions elements in the array:
  - Worst case for linear search: 1 billion comparisons
  - Worst case for binary search: 30 comparisons
- Linear search can work for any array; however, binary search requires sorted arrays.

Linear Search vs Binary Search

- Binary search is more efficient. The complexity of linear search is $O(n)$ while the complexity of binary search is $O(logn)$.
- If we have 1 billions elements in the array:
  - Worst case for linear search: 1 billion comparisons
  - Worst case for binary search: 30 comparisons
- Linear search can work for any array; however, binary search requires sorted arrays.

$\rightarrow$ Sorting algorithms for indexing or grouping elements are needed.

Applications of Searching Algorithms

- It's easy to search or sort a number/a string of characters, but how about images, videos, documents, web?
- How to find customers with the same behaviors?
  - Content-based Information Retrieval - CBIR: e.g. Google image (Reverse image search)
  - Complex structures i.e. graph or tree for indexing and searching

Applications of Searching Algorithms

How to find the patients infected with Covid-19 (F0) by MOH in Vietnam?

Applications of Searching Algorithms

How to find the patients infected with Covid-19 (F0) by MOH in Vietnam?

$\rightarrow$ Apply the principles of binary search to reduce the test cost and time

Sorting

Principle

A sorting algorithm is an algorithm that puts elements of a list in a certain order. For numerical values, we often sort them in **ascending** or **descending order**.

Sorting

Principle

A sorting algorithm is an algorithm that puts elements of a list in a certain order. For numerical values, we often sort them in **ascending** or **descending order**.

- Numbers are said to be in ascending order when they are arranged from the smallest to the largest number. Example: 2, 3, 5, 8, 13, 15, 21, 23.
- Descending order indicates that numbers are arranged from the largest to the smallest number. Example: 23, 21, 15, 13, 8, 5, 3, 2.

Sorting

- Sorting data is one of the most important computing applications. For complex data such as image, voice, video, text, document, sorting this data requires advance algorithms.
- In this lecture, we explore the simplest known sorting algorithms for numbers:
  - Elementary sorting: *Selection Sort, Insertion Sort, Bubble Sort*.
  - Advance sorting: *Quick Sort, Merge Sort*.

Visualize sorting algorithms:

- http://math.hws.edu/eck/js/sorting/xSortLab.html
- https://www.toptal.com/developers/sorting-algorithms

Elementary Sorting

Problematics

Given an array of n elements denoted by $a_0, a_1, a_2, ..., a_{n-1}$, the objective is to sort this sequence in **ascending order** such as:

$$a_0 < a_1 < a_2 < ... < a_{n-1}.$$

In this lecture, we focus on sorting arrays in ascending order in our samples.

Elementary Sorting

Algorithms are different from each other, but two criteria should be considered:

Elementary Sorting

Algorithms are different from each other, but two criteria should be considered:

- **Computational complexity**: An efficient sorting algorithm should have low complexity. Given the size of the list of n elements, for typical serial sorting algorithms, good behavior is $O(nlogn)$, with parallel sort in $O(log^2 n)$, and bad behavior is $O(n^2)$.

Elementary Sorting

Algorithms are different from each other, but two criteria should be considered:

- **Computational complexity**: An efficient sorting algorithm should have low complexity. Given the size of the list of n elements, for typical serial sorting algorithms, good behavior is $O(nlogn)$, with parallel sort in $O(log^2 n)$, and bad behavior is $O(n^2)$.
- **Memory consumption**: it concerns a program consuming computer resources. Cheap memory usage is preferred.

Selection Sort

Principle

- The algorithm divides the input list into two parts: the sublist of elements already sorted and the unsorted sublist of elements remaining to be sorted.

Selection Sort

Principle

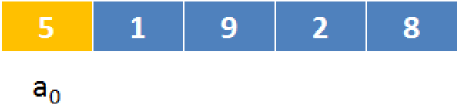
- The algorithm divides the input list into two parts: the sublist of elements already sorted and the unsorted sublist of elements remaining to be sorted.
- The algorithm proceeds by:
  - find the **smallest element** in the unsorted sublist
  - swap this element with the **leftmost unsorted element**, it equivalents to move this element from the unsorted sublist to the sorted one,
  - continue to proceed with all elements in the **unsorted sublist**.

Selection Sort

5	1	9	2	8

for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort



for $i \leftarrow 0$ **to** $n-1$ **do**

  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$

  swap a_i and $a_{idx_{min}}$

end for

Selection Sort

5	1	9	2	8

a_0

$a_{min} = a_1 = 1$, swap a_0 and a_1

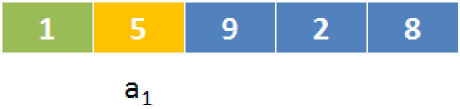
for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	5	9	2	8

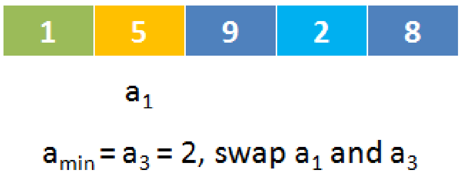
for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort



for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort



for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	9	5	8

for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	9	5	8

a_2

for $i \leftarrow 0$ **to** $n-1$ **do**

  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$

  swap a_i and $a_{idx_{min}}$

end for

Selection Sort

$a_{min} = a_3 = 5$, swap a_2 and a_3

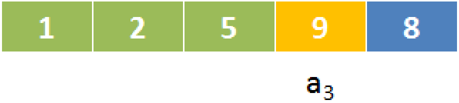
for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	5	9	8

for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort



for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort

$a_{min} = a_4 = 5$, swap a_3 and a_4

for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	5	8	9

for $i \leftarrow 0$ **to** $n-1$ **do**
    $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
    swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	5	8	9
				a_4

for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort

1	2	5	8	9

for $i \leftarrow 0$ **to** $n-1$ **do**
  $idx_{min} \leftarrow \arg\min_{k=i,..,n-1} a_k$
  swap a_i and $a_{idx_{min}}$
end for

Selection Sort

C/C++ Code

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

Selection Sort

Complexity

Count operations inside the loop

- first iteration makes n-1 comparisons, second does n-2, and so on
- one swap per iteration

Total operations:

$$n - 1 + n - 2 + n - 3 + \ldots + 2 + 1 = n\,(n-1)\,/\,2$$

Thus the complexity of Selection Sort is $O(n^2)$

Insertion Sort

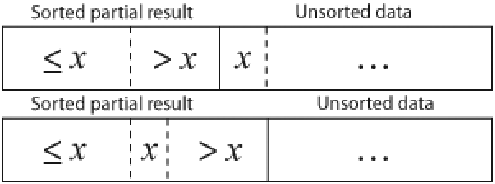
Principle

- Insertion Sort algorithm iterates between the sorted part and the unsorted part.

Insertion Sort

Principle

- Insertion Sort algorithm iterates between the sorted part and the unsorted part.
- The algorithm proceeds by:
  - remove one element from the unsorted part
  - find the location it belongs within the sorted list and inserts it there.
  - repeat until no elements remain in the **unsorted sublist**.



Insertion Sort

5	1	9	2	8

for $i \leftarrow 0$ **to** $n-1$ **do**
  $j \leftarrow i$
  while $j > 0$ && $a[j-1] > a[j]$ **do**
    swap $a[j-1]$ and $a[j]$
    $j \leftarrow j - 1$
  end while
end for

Insertion Sort

5	1	9	2	8

a_0

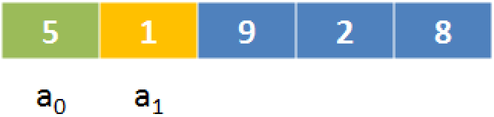
```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

5	1	9	2	8

```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort




```
for  i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

5	1	9	2	8
a_0	a_1			

$a_0 > a_1$, insert a_1 before a_0

for $i \leftarrow 0$ **to** $n-1$ **do**
    $j \leftarrow i$
    while $j > 0$ && $a[j-1] > a[j]$ **do**
        swap $a[j-1]$ and $a[j]$
        $j \leftarrow j-1$
    end while
end for

Insertion Sort

1	5	9	2	8

```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

for $i \leftarrow 0$ **to** $n-1$ **do**
  $j \leftarrow i$
  while $j > 0$ && $a[j-1] > a[j]$ **do**
    swap $a[j-1]$ and $a[j]$
    $j \leftarrow j-1$
  end while
end for

Insertion Sort

$a_1 < a_2$, no movement requires

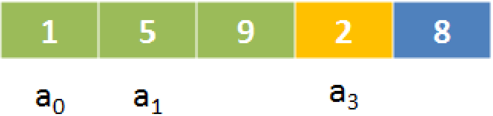
for $i \leftarrow 0$ **to** $n - 1$ **do**
    $j \leftarrow i$
    while $j > 0$ && $a[j-1] > a[j]$ **do**
        swap $a[j-1]$ and $a[j]$
        $j \leftarrow j - 1$
    end while
end for

Insertion Sort

1	5	9	2	8

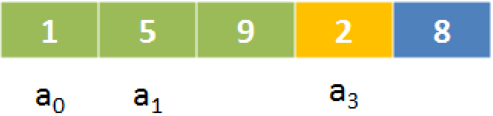
for $i \leftarrow 0$ **to** $n-1$ **do**
    $j \leftarrow i$
    while $j > 0$ && $a[j-1] > a[j]$ **do**
        swap $a[j-1]$ and $a[j]$
        $j \leftarrow j - 1$
    end while
end for

Insertion Sort



for $i \leftarrow 0$ **to** $n-1$ **do**
  $j \leftarrow i$
  while $j > 0$ && $a[j-1] > a[j]$ **do**
    swap $a[j-1]$ and $a[j]$
    $j \leftarrow j-1$
  end while
end for

Insertion Sort



$a_0 < a_3 < a_1$, insert a_3 between a_0 and a_1

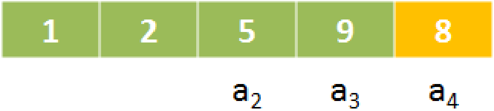
```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

1	2	5	9	8

```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort




```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

1	2	5	9	8
		a_2	a_3	a_4

$a_2 < a_4 < a_3$, insert a_4 between a_2 and a_3

```
for i ← 0 to n − 1 do
    j ← i
    while j > 0 && a[j − 1] > a[j] do
        swap a[j − 1] and a[j]
        j ← j − 1
    end while
end for
```

Insertion Sort

1	2	5	8	9

for $i \leftarrow 0$ **to** $n - 1$ **do**
  $j \leftarrow i$
  while $j > 0$ && $a[j-1] > a[j]$ **do**
    swap $a[j-1]$ and $a[j]$
    $j \leftarrow j - 1$
  end while
end for

Insertion Sort

C/C++ Code

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

Insertion Sort

Complexity

Count operations inside the loop

- first iteration does 1 comparisons, second does ≤ 2, third ≤ 3 and so on
- last iteration optentially follows with n-1 comparisons

Total operations:

$$n -1 + n -2 + n-3 + ... + 2 + 1 = n\ (n-1)\ /\ 2$$

Thus the complexity of Insertion Sort is $O(n^2)$. However what are the complexities for the best and the worst?

Bubble Sort

Principle

Bubble Sort algorithm proceeds by:

- compare each pair of **adjacent elements** and swaps them if they are in the wrong order.
- pass through the list and repeat until no swaps are needed.

Bubble Sort

5	1	9	2	8

repeat
  swapped $\leftarrow$ false
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $a[i-1] > a[i]$ **then**
      swap($a[i-1]$, $a[i]$)
      swapped $\leftarrow$ true
    end if
  end for
until swapped = false

Bubble Sort

5	1	9	2	8
a_0	a_1			

repeat
    swapped $\leftarrow$ false
    for $i \leftarrow 1$ **to** $n-1$ **do**
        if $a[i-1] > a[i]$ **then**
            swap($a[i-1]$, $a[i]$)
            swapped $\leftarrow$ true
        end if
    end for
until swapped = false

Bubble Sort

5	1	9	2	8
a_0	a_1			

$a_0 > a_1$, swap a_0 and a_1

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	9	2	8

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	9	2	8
	a_1	a_2		

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

$a_1 < a_2$, no swap requires

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	9	2	8

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	9	2	8
		a_2	a_3	

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

$a_2 > a_3$, swap a_2 and a_3

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	9	8

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	9	8
			a_3	a_4

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	9	8
			a_3	a_4

$a_3 > a_4$, swap a_3 and a_4

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	8	9

repeat
    swapped $\leftarrow$ false
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        if $a[i-1] > a[i]$ **then**
            swap($a[i-1]$, $a[i]$)
            swapped $\leftarrow$ true
        end if
    end for
until swapped $=$ false

Bubble Sort

1	5	2	8	9
a_0	a_1			

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	8	9
a_0	a_1			

$a_0 < a_1$, no swap requires

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	5	2	8	9

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort


```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

$a_1 > a_2$, swap a_1 and a_2

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	2	5	8	9

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort


```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

$a_2 < a_3$, no swap requires

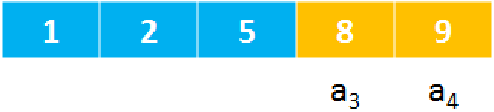
repeat
  swapped $\leftarrow$ false
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $a[i-1] > a[i]$ **then**
      swap($a[i-1]$, $a[i]$)
      swapped $\leftarrow$ true
    end if
  end for
until swapped $=$ false

Bubble Sort

1	2	5	8	9

repeat
  swapped ← false
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $a[i-1] > a[i]$ **then**
      swap($a[i-1]$, $a[i]$)
      swapped ← true
    end if
  end for
until swapped = false

Bubble Sort




```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	2	5	8	9
			a_3	a_4

$a_3 < a_4$, no swap requires

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	2	5	8	9

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

1	2	5	8	9

no further swap is needed

```
repeat
    swapped ← false
    for i ← 1 to n − 1 do
        if a[i − 1] > a[i] then
            swap( a[i − 1], a[i] )
            swapped ← true
        end if
    end for
until swapped = false
```

Bubble Sort

C/C++ Code

```
int bubble(int a[], int n) {
        int swapped = 1;
        while (swapped == 1){
                swapped = 0;
                for (int i = 1; i <=n;i++)
                        if (a[i-1] > a[i]){
                                swap(&a[i-1],&a[i])
                                swapped = 1;
                        }
        }
        return 0;
}
```

Bubble Sort

Complexity

Count operations inside the loop

- first iteration does n-1 comparisons and n-1 swaps,
- second does n-2 comparisons and n-2 swaps,
- .... (n-1)th iteration does one comparison and one swap.

Total operations:

$$2(n -1 + n -2 + n\text{-}3 + \ldots + 2 + 1) = n\ (n\text{-}1)$$

Thus the complexity of Bubble Sort is $O(n^2)$.

Conclusion

- Selection Sort, Insertion Sort, and Bubble Sort have a complexity of $O(n^2)$ in the worst case where the array is in descending order. The best case is that the array is already sorted in the right order.
- Since the complexity is too high; sorting algorithms are sensitive to the size of array n. If n is too big, the cost is very expensive.
- Sorting algorithms have to be improved to accelerate running time.

Efficient Sorting

The previous algorithms have a high complexity $O(n^2)$; many efficient sorting algorithms are proposed while improving the running cost (average complexity $O(nlogn)$).
The most common are:

- Merge Sort
- Quick Sort

Efficient Sorting

The most common strategy is to use **Recursive** and **Divide and Conquer** algorithms

- Divide: If the input size is too large to deal with straightforwardly, divide the data into two or more disjoint subsets.
- Recur: Use divide and conquer to solve the subproblems associated with the data subsets.
- Conquer: Take the solutions to the sub-problems and "merge" these solutions into a solution for the original problem.

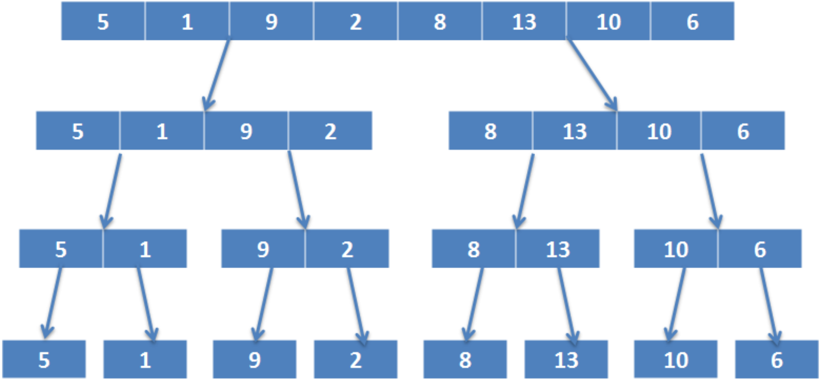
Merge Sort

Principle

Merge sort is a divide and conquer algorithm which can proceed by:

- **Divide**: divide the unsorted array into n sub-arrays.
- **Conquer**: each sub-array contains one element and an array of one element is considered sorted.
- **Recur**: merge sub-arrays repeatedly to produce new sorted sub-array until only one sub-array remains.
- the last sub-array will be the sorted array.

Merge Sort

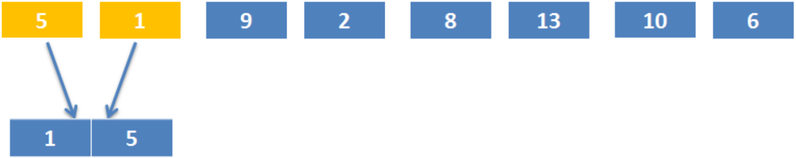


Divide the unsorted array into 1-element sub-arrays.

Merge Sort

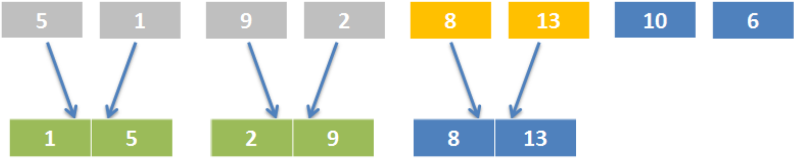
Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort



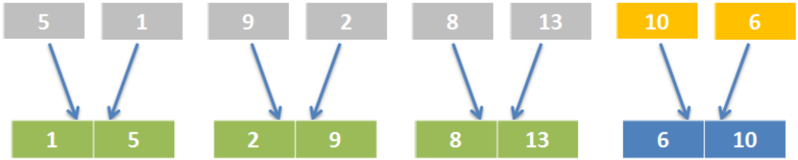
Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort



Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

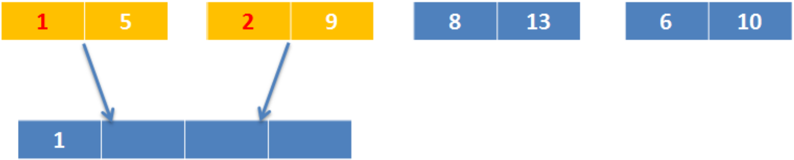


Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

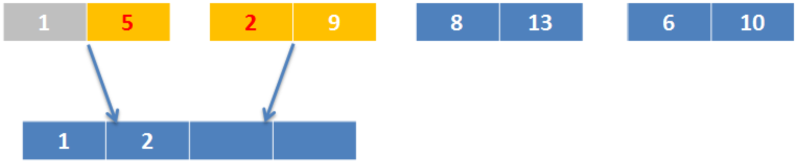
Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort



Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

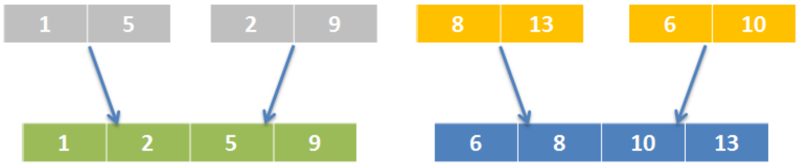


Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort



Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

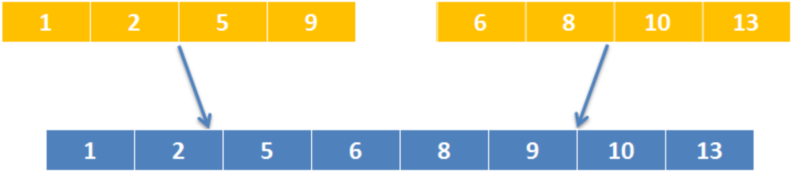
Merge Sort

1	2	5	9

6	8	10	13

Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort



Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

1	2	5	6	8	9	10	13

Merge sub-arrays repeatedly to produce new sorted sub-arrays until only one sub-array remains.

Merge Sort

Merge Soft algorithm can be written as follows :

mergeSort (a, p, r)

```
if (p > r) then
    q ← (p+r)/2;
    mergeSort (a, p, q)
    mergeSort (a, q+1, r)
    merge (a, p, q, r)
end if
```

where the merge is a function allowing to combination of sub-arrays.

Merge Sort

Q. How much memory does mergesort require?
A. Too much!

- Original input array = n.
- Auxiliary array for merging = n.
- Local variables: constant.
- Function call stack: log n
- Total = 2n + log n.

Q: How much memory do other sorting algorithms require?
A: n + k variable declaration for selection sort, insertion sort, and selection sort.

Quick Sort

Principle

Quick sort can be considered as a divide and conquer algorithm which can proceed by:

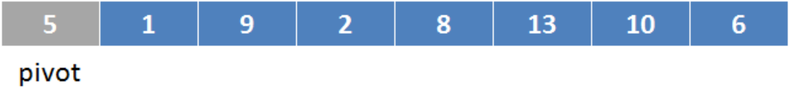
- Select an element randomly, called a pivot, from the array.
- **Conquer** : arrange the array so that all elements with values less than the pivot come before the pivot (lower part), while all elements with values greater than the pivot come after it (higher part).
- **Divide**: the array is now divided into lower and higher parts.
- **Recur**: apply recursively and separately the above steps to these two parts.

Quick Sort

5	1	9	2	8	13	10	6

Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



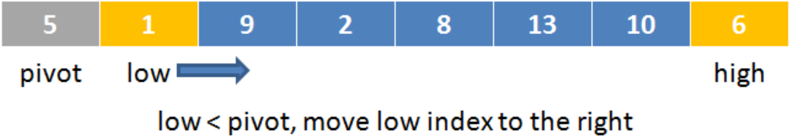
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

5	1	9	2	8	13	10	6
pivot	low						high

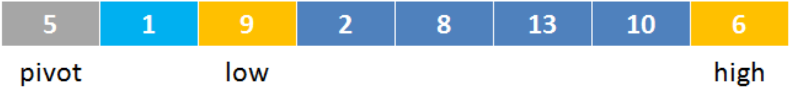
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



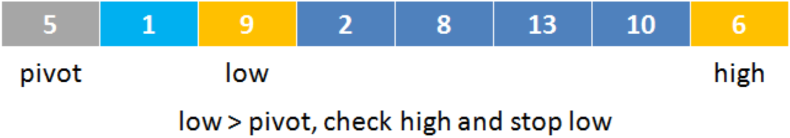
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



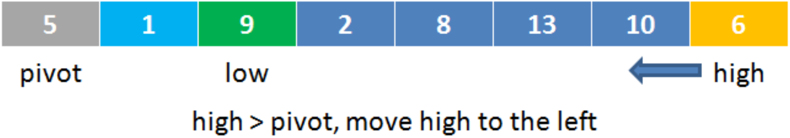
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



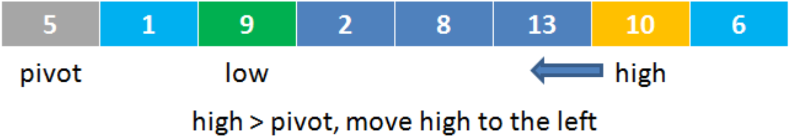
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

5	1	9	2	8	13	10	6
pivot		low				high	

Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



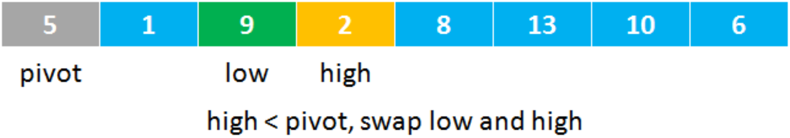
Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

5	1	9	2	8	13	10	6
pivot		low	high				

Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort



Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

2	1	5	9	8	13	10	6

Choose a pivot and arrange elements respectively into lower and higher parts

Quick Sort

- At the end of this step, we have two new arrays to be sorted.
- Each array will be sorted using Quick Sort algorithm.
- The base case for recusive calls is where each array has one element or two elements.

Quick Sort

Quick Soft algorithm can be written as following:

quickSort (a, low, high)

1: **if** (low $<$ high) **then**
2:     p $\leftarrow$ partition(a, low, high)
3:     quicksort(a, low, p - 1)
4:     quicksort(a, p $+$ 1, high)
5: **end if**

Quick Sort

partition (a, low, high)

```
pivot ← a[high]
i ← lo w
for j ← low to high -1 do
    if a[j] ≥ pivot then
        swap (a[i],a[j])
        i := i + 1
    end if
end for
swap (a[i],a[j])
return i
```

Conclusion

Complexity Comparison

Algorithm	Best	Average	Worst	Space
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	O(n)
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)