Analysis of Algorithms

Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine **how much in the way of resources**, such as **time** or **space**, the algorithm will require.

- **Space Complexity** → memory and storage are very cheap nowadays. ×
- **Time Complexity** ✓ Different platforms → different time. Absolute time is hard to measure as it depends on many factors.

Example: moving between university buildings: it depends on who are walking, which way he/she use, etc. time is not good measurement. Number of steps is a better one.

Example:

- Consider the problem of summing $\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$

Come up with an algorithm to solve this problem.

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

Counting Basic Operations

- A **basic operation** of an algorithm is the most significant contributor to its total time requirement.

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	**3**

How to calculate the time complexity?

- Measure execution time. × Algorithm for small data size will take small time comparing to a large data.
- Calculate time required for an algorithm in terms of the size of input data. × Does not work as the same algorithm over the same data will not take the same time.

Run summing code 2 times and compare time

- Determine order of **growth** of an algorithm with respect to the size of input data. ✓

Order of time or **growth of time:**

Go back to summing result

n,	A,	B,	C
1)	7183,	7183,	820
10)	2052,	4105,	102
100) ,	7183,	155974,	1026
1000) ,	66700,	2983004,	3079
10000) ,	411484,	149256917,	2052
100000) ,	1903500,	13209223813,	1027

Linear growth

Quadratic growth

Constant growth

In term of **time complexity**, we say that algorithm **C** is better than **A** and **B**

Types of Time Complexity

- Best case analysis ✗ too optimistic
- Average case analysis ✗ too complex (statistical methods)
- Worst case analysis ✓ it will not exceed this

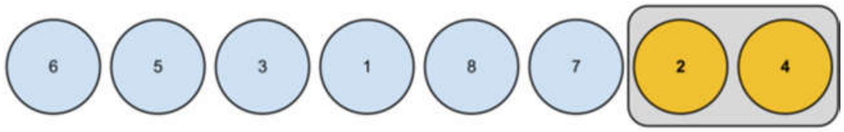
RAM model of computation

We assume that:

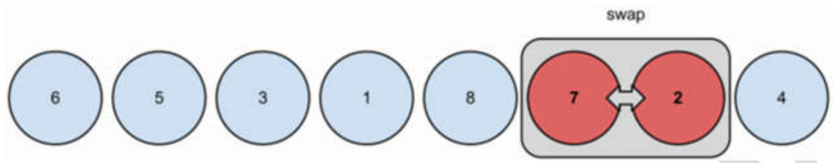
- We have infinite memory
- Each operation (+,-,*,/,=) takes 1 unit of time
- Each memory access takes 1 unit of time
- All data is in the RAM

Bubble Sort:

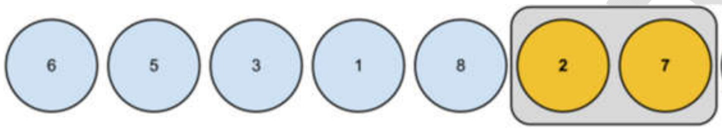
1. Each two adjacent elements are compared:

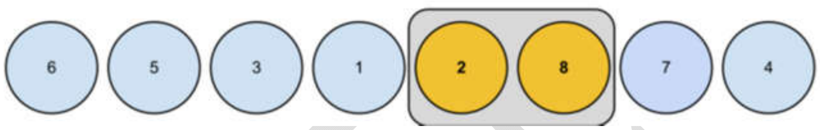


2. Swap with larger elements:

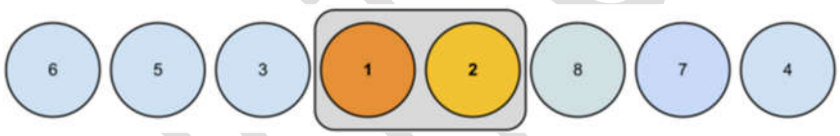


3. Move forward and swap with each larger item:

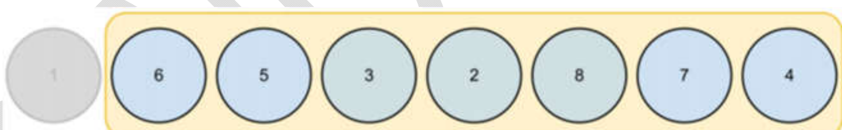




4. If there is a lighter element, then this item begins to bubble to the surface:



5. Finally the smallest element is on its place:



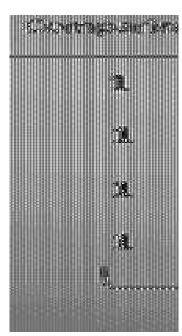
Make a demo using the following data set

12	8	7	5	2

Worst case analysis

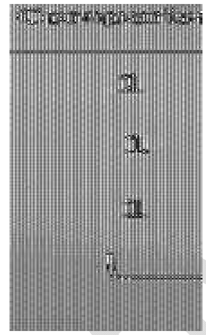
After 1st round:

8	7	5	2	12



After 2nd round:

7	5	2	8	12



For whole sorting algorithm: **16+12+8+4** for a data size of 5 elements:

= 4 (4 + 3 + 2 + 1) = 4 (n-1 + n-2 + …. + 2 + 1) = 4 (n-1*n/2) =

2 * n * (n-1) ➔ **$pn^2 + qn + r$** → **p**, **q**, and **r** are some constant.

Implement and test effectiveness of bubble sort algorithm

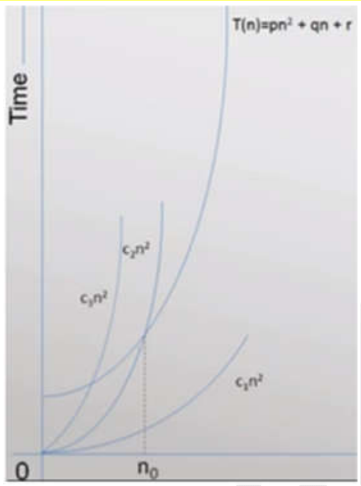
for (int i = 0; i < arr.length-1; i++) {	i=0	j=n-1	n-1
for (int j = 0; j <arr.length-i-1 ; j++) {	i=1	j=n-2	n-2
if(arr[j+1]<arr[j]){	:	:	:
temp = arr[j];	:	:	:
arr[j] = arr[j+1];	i=n-1	j=0	1
arr[j+1] = temp;			
}			
}			
}			

The Big-O Notation

Assume the order of time of an algorithm is a **quadratic** time as displayed in the graph. Our job is to find an **upper bond** for this function **T(n)**. Consider a function c_1n^2 ← never over take **T(n)**

C_2n^2 such that its greater than **T(n)** for $n>n_0$. In this case we say that C_2n^2 is an upper bond of **T(n)**

But we can come up with many functions satisfy this condition. We need to be precise.



OBSERVATIONS:

$\forall\, n > n_0$

$c_2n^2 \geq pn^2 + qn + r$

Big Oh $\mathbf{O(n^2)}$: **f(n)**: there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0 \leq f(n) \leq cn^2}$ for all $\mathbf{n \geq n_0}$

In general

$\mathbf{O(g(n))}$: **f(n)**: there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0 \leq f(n) \leq cg(n)}$ for all $\mathbf{n \geq n_0}$

Example 1:

$\mathbf{5n^2 + 6 \in O(n^2)}$??? ✓

Find $\mathbf{cn^2}$ → $c=6$ and $n_0=3$

→ $c=5.1$ $n_0=8$

Example 2:

$\mathbf{5n + 6 \in O(n^2)}$??? ✓

Find $\mathbf{cn^2}$ → $c=11$ and $n_0=1$

Example 3:

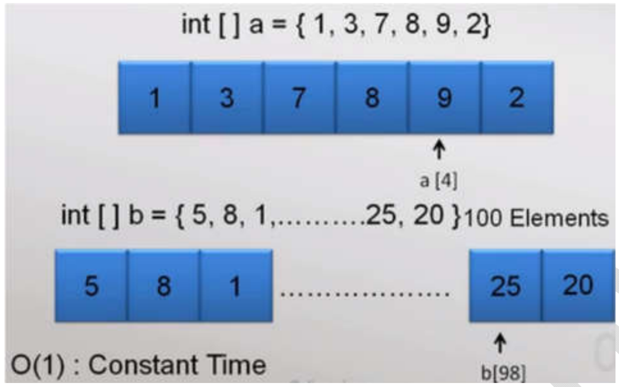
$\mathbf{n^3 + 2n^2 + 4n + 8 \in O(n^2)}$??? ✗

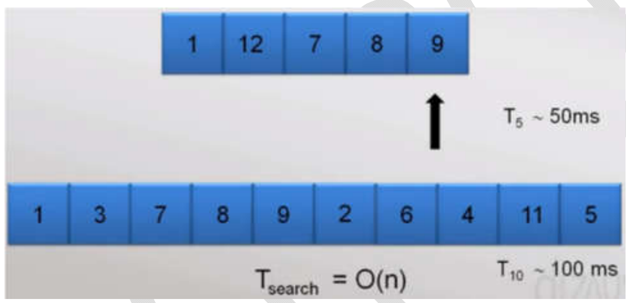
Find $cn^2 \geq n^3 + 2n^2 + 4n + 8$??? ✗

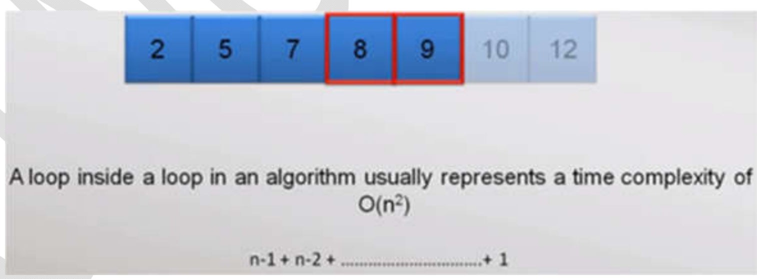
$$a_m n^m + a_{m-1}n^{m-1} - - - - - - - - - - - + a_0 \in O(n^m)$$

$$\log n \leq \sqrt{n} \leq n \leq n\log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

What does it mean?

Array element access: 

Array element search: 

Bubble sort algorithm: 

Asymptotic Analysis

Asymptotic (مقارب) analysis measures the efficiency of an algorithm as the input size becomes large.

It is actually an **estimation** technique. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

- The critical resource for a program is -most often- **running time**.
- The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
  - ***cn*** (for ***c*** any positive constant) ➔ **linear** growth rate or running time.
  - n^2 ➔ **quadratic** growth rate
  - 2^n ➔ **exponential** growth rate.

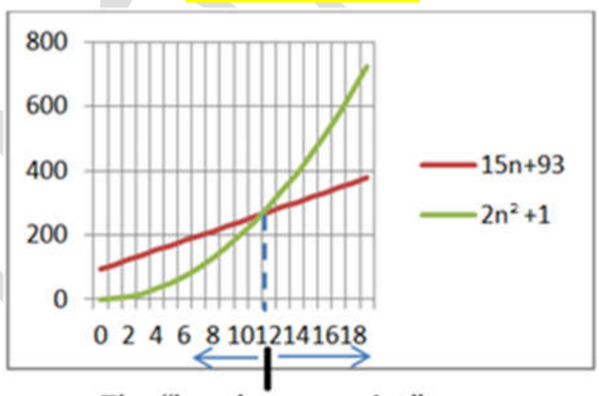
Worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well.

Example:

Assume: Algorithm A: time = **15n + 93**

Algorithm B: time = $\mathbf{2n^2 + 1}$ **which is faster?**

Graph using Excel



The "break-even point"

We are interested for large **n**

*** For sufficiently large n, algorithm A is faster**

*** In the long run constants do not mater.**

Upper bound for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have. ➔ **big-O notation**.

For **T(*n*)** a non-negatively valued function, **T(*n*)** is in set **O(*f*(*n*))** if there exist two positive constants ***c*** and n_0 such that **T(*n*) ≤ *cf*(*n*)** for all $n > n_0$.

- Prove that **15n + 93** is **O(n)**

  We must show +ve **c** and n_0 such that **15n + 93 ≤ c(n)** for **n ≥ n_0**

  <provided **n= 93**> ➔ **15n+n** ➔ **16n ≤ cn** → <provided **c = 16**>

  So for **c=16** and **n_0 = 93** ➔ **// proved**

  Graph using Excel

- Prove that **$2n^2+1 = O(n^2)$**

  Must show +ve **c**, n_0 such that **$2n^2+1 \le c(n^2)$ for $n \ge n_0$**

  $2n^2+1$ <provided n=1>

  $2n^2+ n^2$ ➔ $3n^2$ <provided c=3>

  $2n^2+1 \le 3n^2$

  So, **c=3 , n_0=1** // proved

  Graph using Excel

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1n^2 + c_2n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1n^2 + c_2n \le c_1n^2 + c_2n^2 \le (c_1 + c_2)n^2$ for all $n > 1$. So, $\mathbf{T}(n) \le cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the second definition.

The **lower bound** for an algorithm is denoted by the symbol **Ω**, pronounced “big-Omega” or just “Omega.”

For **T(*n*)** a non-negatively valued function, **T(*n*)** is in set **Ω(*g*(*n*))** if there exist two positive constants ***c*** and n_0 such that **T(*n*) ≥ *cg*(*n*)** for all $n > n_0$.

- Prove that **15n+93** is **Ω(n)**

  We must show +ve **c** and n_0 such that **15n+93 ≥ c(n)** for **n ≥ n_0**

  <because 93 is +ve> ≥ **c(n)** ➔ <provided **c=15**> ← so any **n_0 > 0** will do

  So **c=15, n_0=1** // proved

  Graph using Excel

- Prove that **$2n^2+1$** is **$\Omega(n^2)$**

  Must show +ve **c** and n_0 such that **$2n^2+1 \ge cn^2$** for **$n \ge n_0$**

  <because 1 is +ve>

So **c=2, n_0=1** // proved

Graph using Excel

Example 3.7 Assume $\mathbf{T}(n) = c_1n^2 + c_2n$ for c_1 and $c_2 > 0$. Then,

$$c_1n^2 + c_2n \geq c_1n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

When the **upper** and **lower bounds** are the same within a constant factor, we indicate this by using **Θ (big-Theta)** notation.

T(n) = **Θ(g(n)) iff** T(n) = **O(g(n))** **and** T(n) = **Ω (g(n))**

Example: Because the **sequential search algorithm** is both in **O(*n*)** and in **Ω(*n*)** in the average case, we say it is **Θ(*n*)** in the average case.

Simplifying Rules

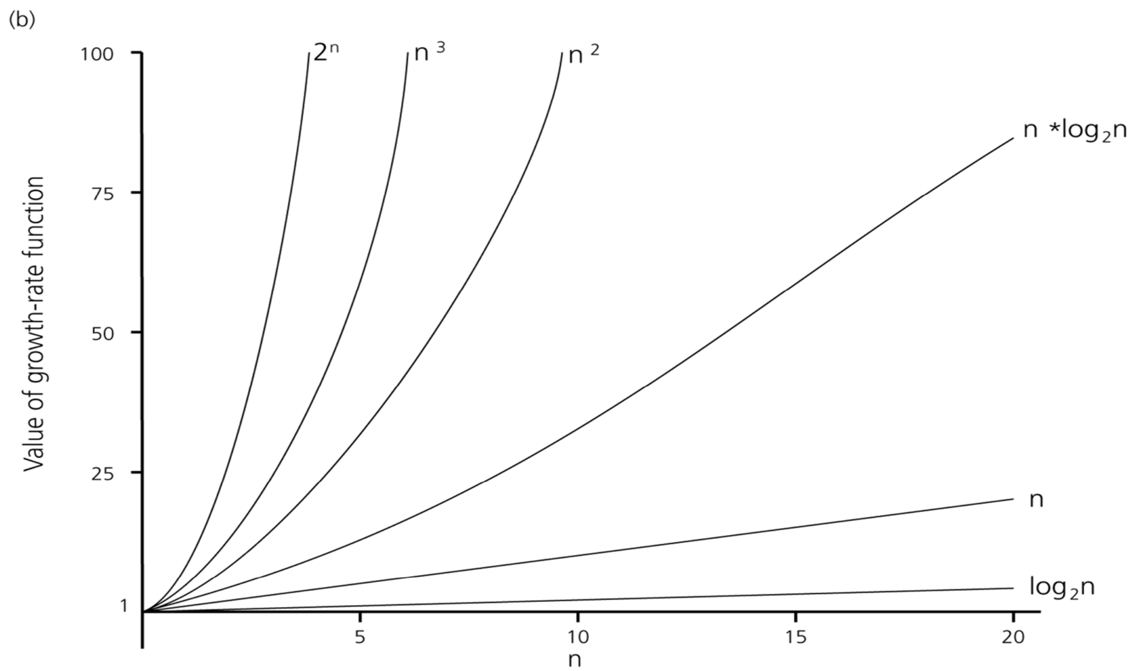
1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

- **Rule (2)** is that you can ignore any multiplicative constants.
- **Rule (3)** says that given two parts of a program run in sequence, you need to consider only the more expensive part.
- **Rule (4)** is used to analyze simple loops in programs.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function.

Order of growth of some common functions:

$$O(1) \le O(\log_2 n) \le O(n) \le O(n \log_2 n) \le O(n^2) \le O(n^3) \le O(2^n)$$



If the problem size is always small, you can probably ignore an algorithm's efficiency

Limitations of big-O analysis:

- Overestimate.
- Analysis assumes infinite memory.
- Not appropriate for small amounts of input.
- The constant implied by the Big-Oh may be too large to be ignored (**2*N* log *N*** *vs.* **1000*N***)

Analyzing Algorithm Examples

General Rules of analyzing algorithm code:

Rule 1 — *for* loops:

The running time of a **for** loop is at most the running time of the statements inside the **for** loop (including tests) **times** the number of iterations.

Rule 2 — Nested loops:

Analyze these **inside out**. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

Rule 3 — Consecutive Statements:

These just add (which means that the maximum is the one that counts.

Rule 4 — *if/else*:

```
if( condition )
    S1
else
    S2
```

The running time of an **if/else** statement is never more than the running time of the **test** plus the larger of the running times of **S1** and **S2**.

Rule 5 — *methods call*:

If there are method calls, these must be analyzed first.

Sorting Algorithm

1- Bubble Sort (revision) ➔ O(n^2)

```
public static void bubble(int[] arr){
  int temp;
  for (int i = 0; i < arr.length-1; i++) {
    for (int j = 0; j <arr.length-i-1 ; j++) {
      if(arr[j+1]<arr[j]){
        temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
}
```

2- **Selection Sort (revision) ➔ O(n^2)**: named selection because every time we select the smallest item.

```
public static void selection (int[] arr){
   int temp, minIndex;
   for (int i = 0; i < arr.length-1; i++) {
      minIndex = i;
      for (int j = i+1; j <arr.length ; j++) {
         if(arr[j]<arr[minIndex]){
            minIndex=j;
         }
      }
      if(i!= minIndex){
         temp = arr[i];
         arr[i] = arr[minIndex];
         arr[minIndex] = temp;
      }
   }
}
```

3- Insertion sort ➔ O(n^2):

```
public static void insertion (int[] arr){
   int j, temp, current;
   for (int i = 1; i < arr.length; i++) {
     current = arr[i];
      j=i-1;
      while (j>=0 && arr[j]>current){
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1]=current;
   }
}
```

O(n^2) sorting algorithms comparison:

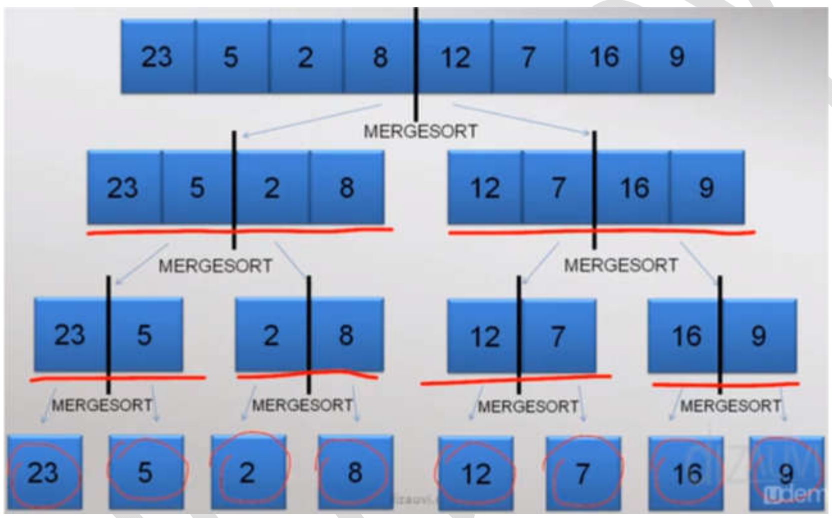
(run demo @ http://www.sorting-algorithms.com/)

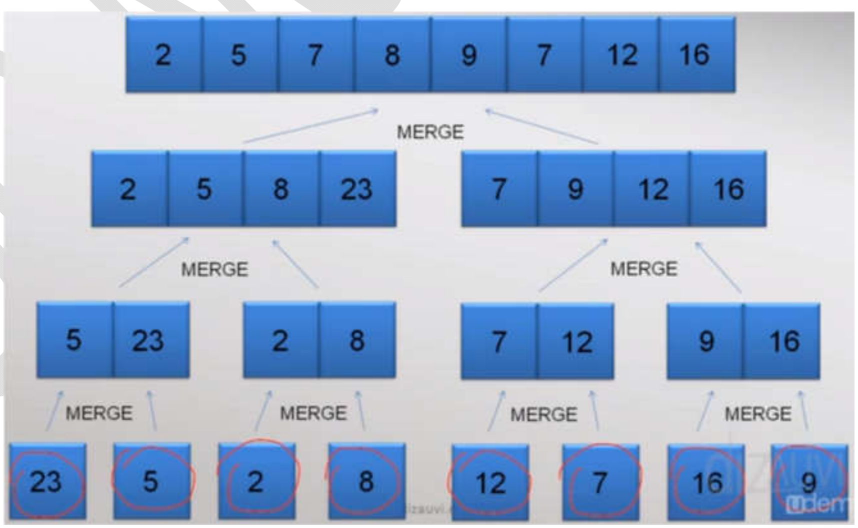
Bubble Sort	Selection Sort	Insertion Sort
Very inefficient	• Better than bubble sort • Running time is independent of ordering of elements	• Relatively good for small lists • Relatively good for partially sorted lists

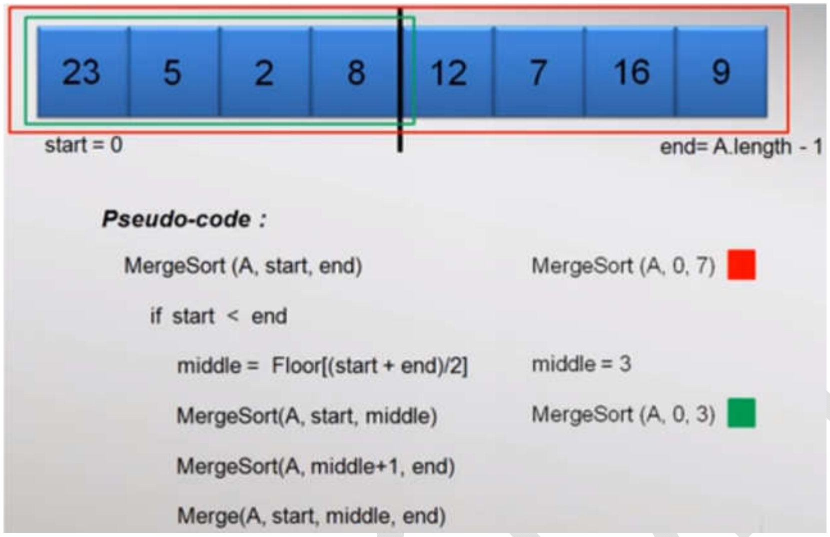
Merge sort: recursive algorithm

Merge: take 2 sorted arrays and merge them together into one.

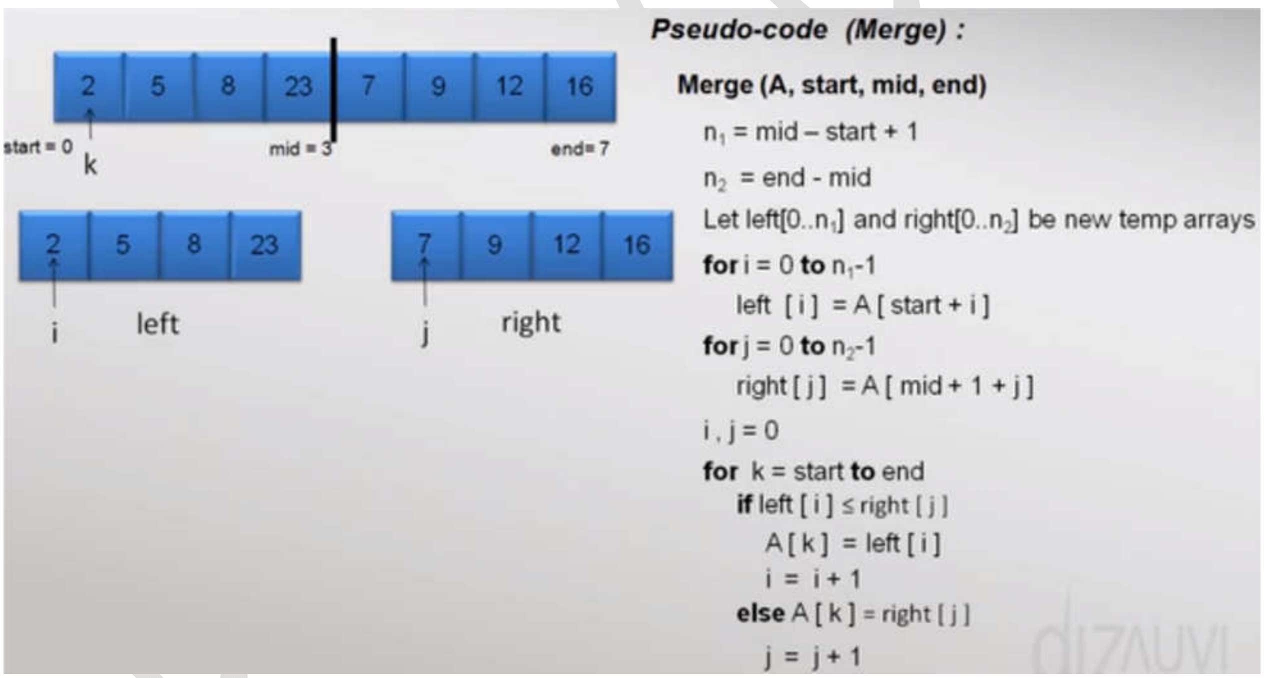
Example:







Pseudo code:

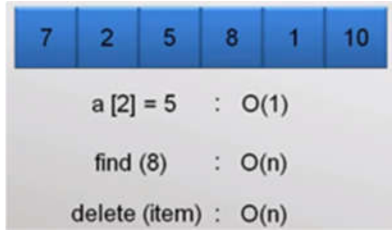


Make sure of array boundaries

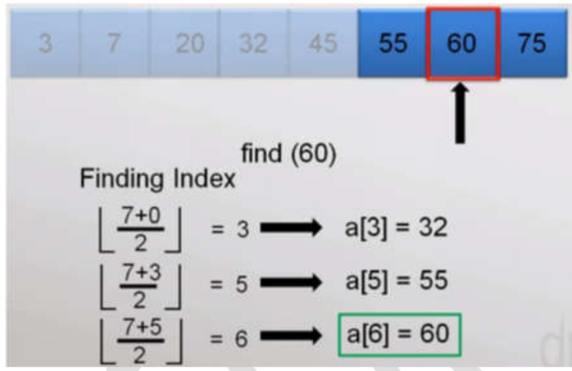
H.W: implement merge sort your own

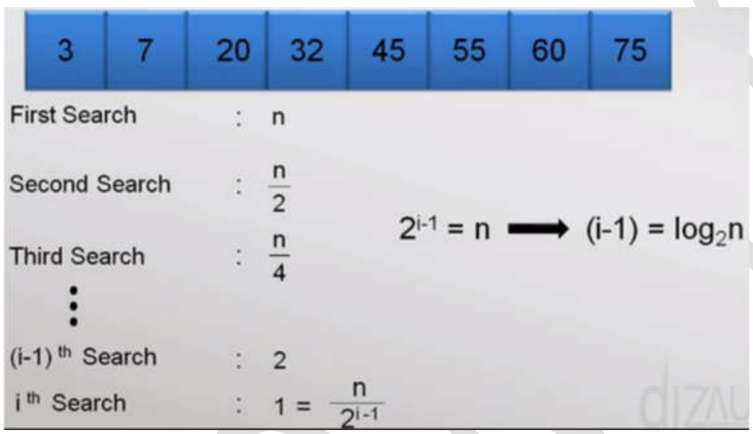
Searching elements in an array:

Case 1: unordered array:



Case 2: ordered array: -Binary search-





find (item) = $O(\log_2 n)$

n	$\log_2 n$
2	1
1024	10
1048576 (Million)	20
1099511627776 (Trillion)	40

Inserting and deleting items from ordered array