جامعة بيرزيت

BIRZEIT UNIVERSITY

COMP232

Data Structure

Lectures Note: Algorithms

Prepared by: **Dr. Mamoun Nawahdah**

2016/2017

Math Review

1. $\log(nm) = \log n + \log m.$
2. $\log(n/m) = \log n - \log m.$
3. $\log(n^r) = r \log n.$
4. $\log_a n = \log_b n / \log_b a.$

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

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6}.$$

$$\sum_{i=1}^{\log n} n = n \log n.$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ for } a \neq 1.$$

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n},$$

and

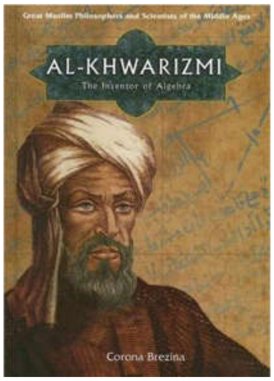
$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1.$$

Finally,

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}.$$

What is an Algorithm?



Born: Uzbekistan
Died: 850 AD, Baghdad, Iraq

Definition:

- **Algorithm** is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will terminate in a defined end-state.

Euclid's Algorithm (300$_{BC}$)

- Used to find Greatest common divisor **(GCD)** of two positive integers.
- GCD of two numbers, the largest number that divides both of them without leaving a remainder.

Euclid's Algorithm:

- Consider two positive integers '**m**' and '**n**', such that **m>n**
- **Step1**: Divide **m** by **n**, and let the reminder be **r**.
- **Step2**: if **r=0**, the algorithm ends, **n** is the GCD.
- **Step3**: Set, **m→n**, **n→r** , go back to **step 1** .

Implement this iteratively and recursively

```
public static int iteratively (int m, int n){
   int r = m % n;
   while (r != 0) {
      m = n;
      n = r;
      r = m % n;
   }
   return n;
}
```

```
public static int recursively(int m, int n) {
      if (n==0)
         return m;
      return recursively(n, m % n);
   }
```

Why Algorithms?

- Gives an idea (estimate) of running time.
- Help us decide on hardware requirements.
- What is feasible vs. what is impossible.
- Improvement is a never ending process.

Correctness of an Algorithm:

- Must be proved (mathematically)

  Step1: statement to be proven.

  Step2: List all assumptions.

  Step3: Chain of reasoning from assumptions to the statement.

- Another way is to check for **incorrectness** of an algorithm.

  Step1: give a set of data for which the algorithm does not work.

  Step2: usually consider small data sets.

  Step3: Especially consider borderline cases.

Recursion

Definition:

- A function that calls itself is said to be recursive.
- A function **f1** is also recursive if it calls a function **f2**, which under some circumstances calls **f1**, creating a cycle in the sequence of calls.
- The ability to invoke itself enables a recursive function to be repeated with different parameter values.
- You can use recursion as an alternative to iteration (looping).

The Nature of Recursion:

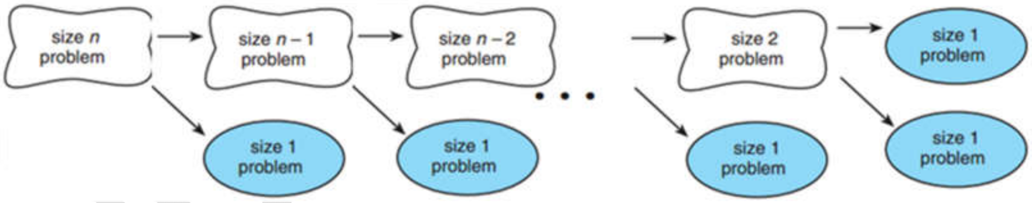
Problems that lend themselves to a recursive solution have the following characteristics:

- One or more simple cases of the problem have a straightforward, non-recursive solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying this redefinition process every time the recursive function is called, eventually the problem is reduced entirely to the simple case(s), which are relatively easy to solve.

The recursive algorithms will generally consist of an "**if** statement" with the following form:

```
if this is a simple case
        solve it
else
        redefine the problem using recursion
```

Illustration:



Example:

Solve the problem of multiplying **6** by **3**, assuming **we only know addition**:

- **Simple case**: any number multiplied by 1 gives us the original number.
- The problem can be split into the two problems:

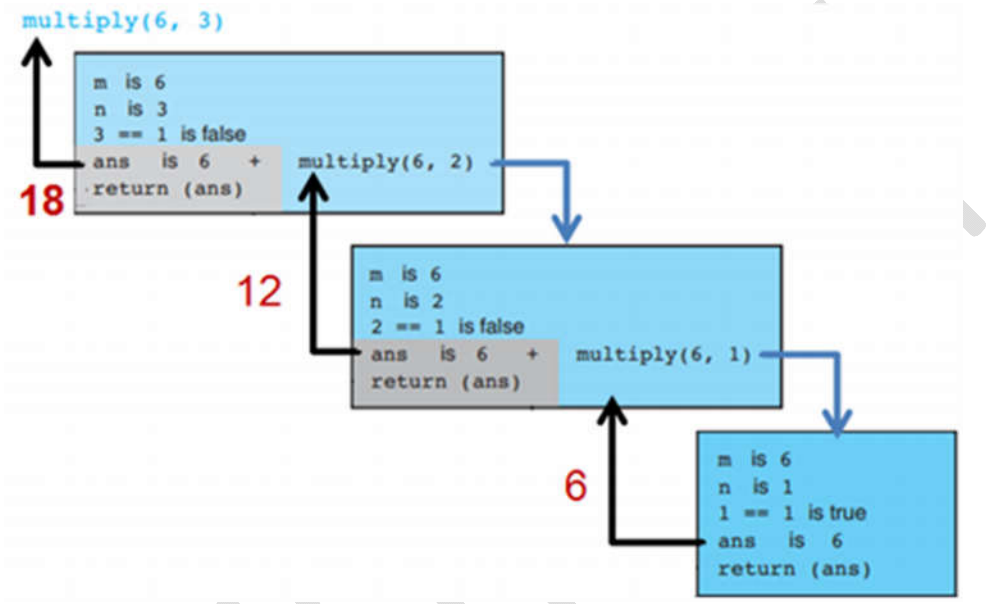
1. Multiply 6 by 2.
1.1 Multiply 6 by 1.
1.2 Add (Multiply 6 by 1) to the result of problem 1.1.
2. Add (Multiply 6 by 1) to the result of problem 1.

Implement this recursively

Tracing a Recursive Function:

- Tracing an algorithm's execution provides us with valuable insight into how that algorithm works.
- By drawing an **activation frame** corresponding to each call of the function.
- An activation frame shows the parameter values for each call and summarizes the execution of the call.

multiply(6, 3):

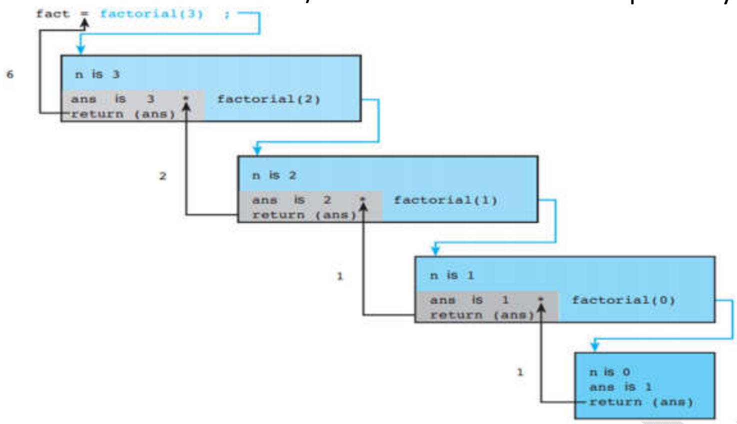


Recursive Mathematical Functions:

- Many mathematical functions can be defined recursively.
- An example is the factorial of n (**n!**):
  - **0!** is 1
  - **n!** is **n * (n 1)!** , for n> 0
- Thus **4!** is **4 *3!**, which means 4 *3 *2 *1, or 24.

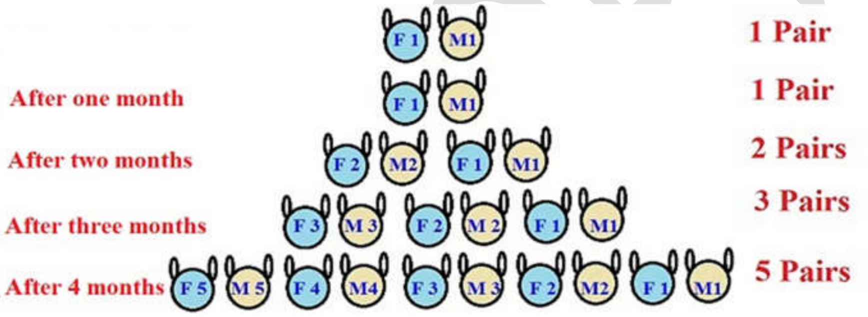
Implement this iteratively and recursively

Tracing the recursive function



Fibonacci Numbers:

Leonardo **Bonacci** (1170 –1250)



- **Problem:**
  - How many pairs of rabbits are alive in month **n**?
  - Recurrence relation:

rabbit(n) = rabbit(n-1) + rabbit(n-2)

❖ The Fibonacci sequence is defined as:

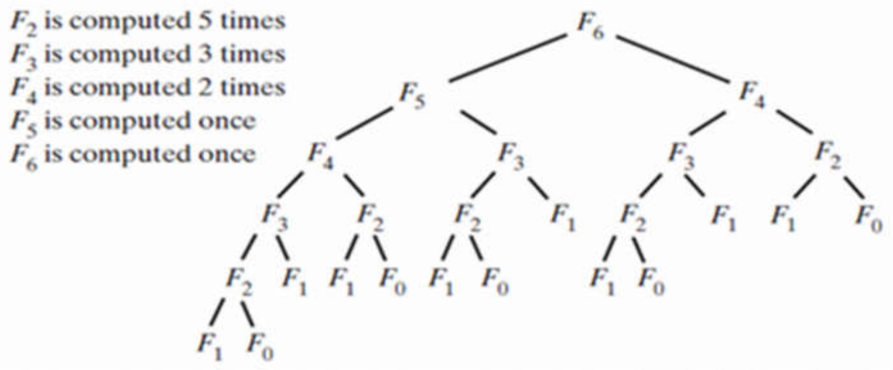
- Fibonacci **0** is 1
- Fibonacci **1** is 1
- Fibonacci **n** is **Fibonacci n 2 + Fibonacci n 1**, for n>1

Implement this recursively

Poor Solution to a Simple Problem:

```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Why is this inefficient? Try F_6



Self-Check:

- Write and test a recursive function that returns the value of the following recursive definition:
  - **f(x) = 0** **if x = 0**
  - **f(x) = f(x - 1) + 2** **otherwise**

What set of numbers is generated by this definition?

Design Guidelines:

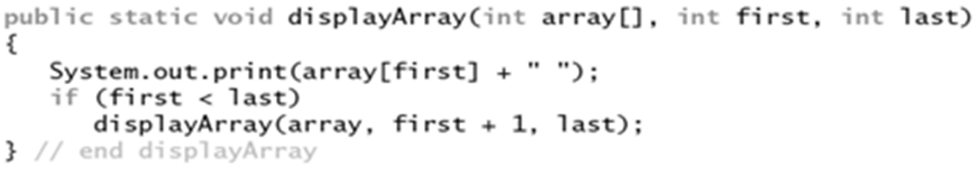
- Method must be given an **input value**.
- Method definition must contain **logic** that involves this input, leads to different cases.
- One or more cases should provide solution that does not require recursion.
  - else **infinite recursion**
- One or more cases must include a recursive invocation.

Stack of Activation Records:

- Each call to a method generates an activation record.
- Recursive method **uses more memory** than an iterative method.
  - Each recursive call generates an activation record.
- If recursive call generates too many activation records, could cause **stack overflow**.

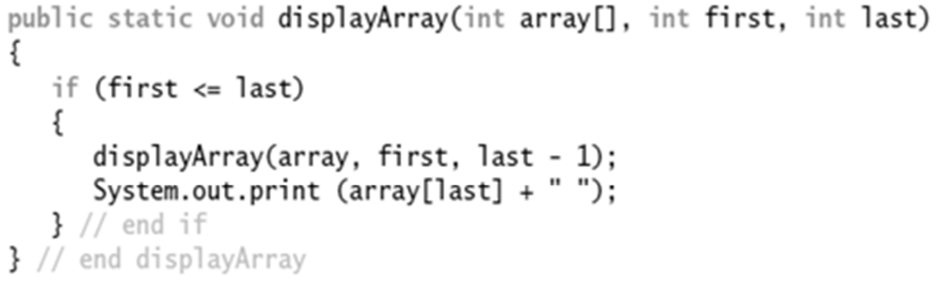
Recursively Processing an Array:

Starting with array[first]:



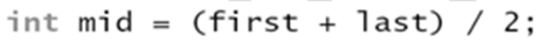
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

Starting with array[last]:

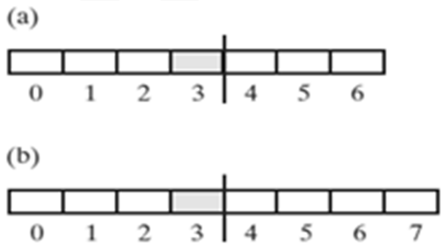


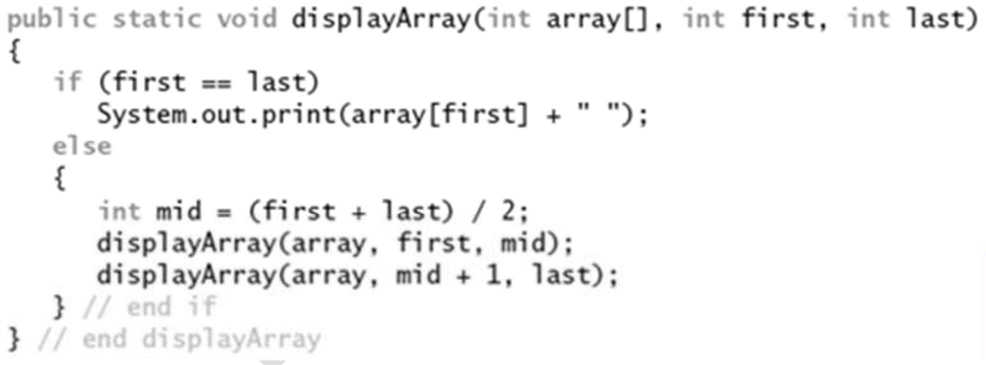
```
public static void displayArray(int array[], int first, int last)
{
   if (first <= last)
   {
      displayArray(array, first, last - 1);
      System.out.print (array[last] + " ");
   } // end if
} // end displayArray
```

Processing array from middle:



```
int mid = (first + last) / 2;
```



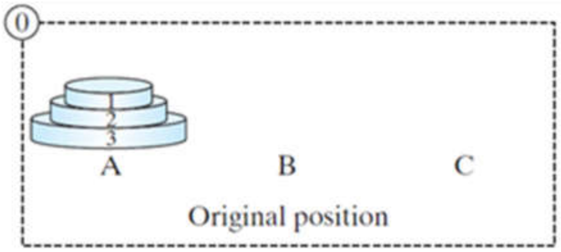


```
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Consider
first + (last – first) / 2
Why?

Tower of Hanoi

Simple Solution to a Difficult Problem:



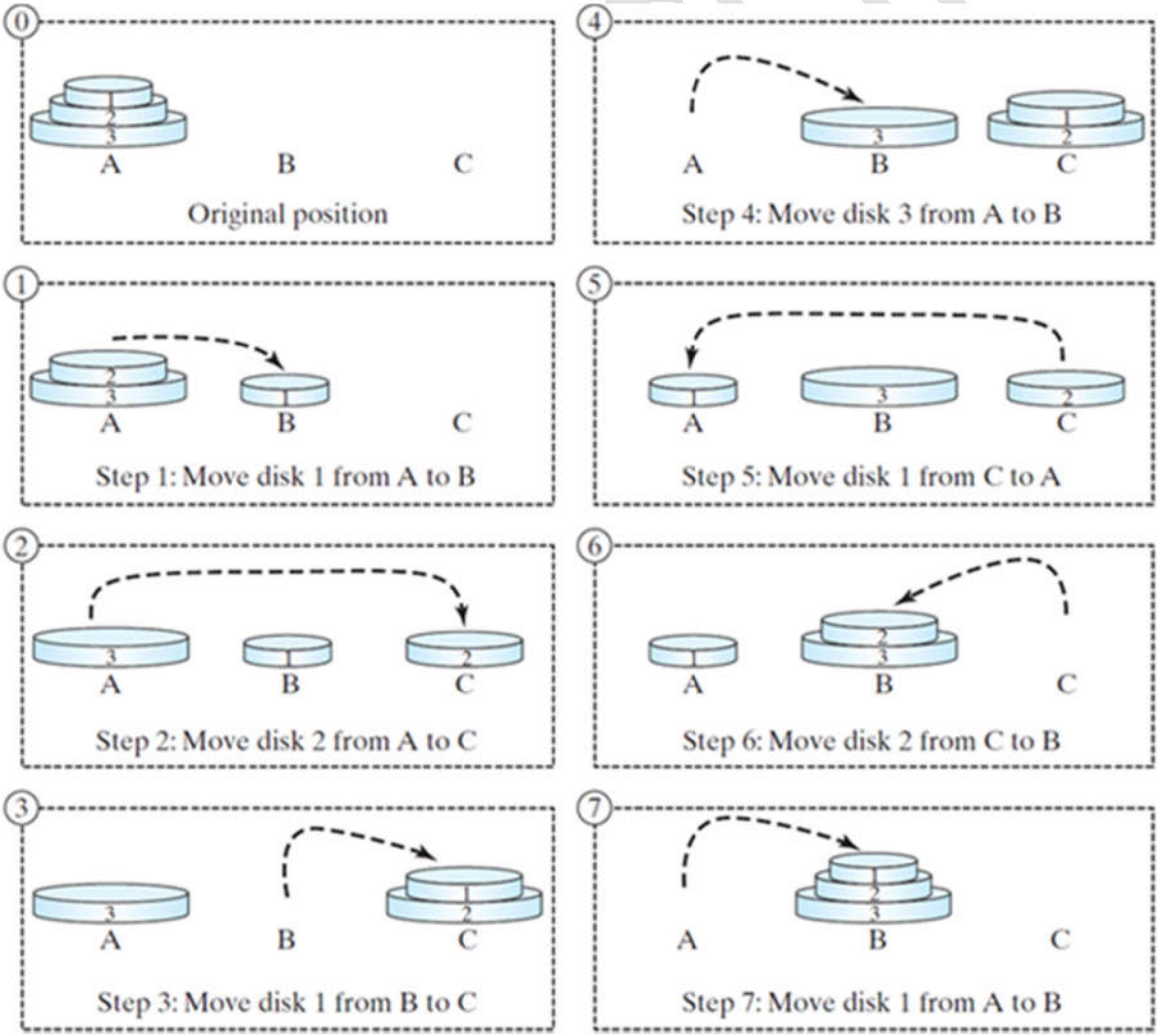
Original position

Rules:

- Move one disk at a time. Each disk moved must be topmost disk.
- No disk may rest on top of a disk smaller than itself.
- You can store disks on the 2nd pole temporarily, as long as you observe the previous two rules.

Tower of Hanoi flash @ https://www.mathsisfun.com/games/towerofhanoi.html

Sequence of moves for solving the Towers of Hanoi problem with three disks:



Original position

Step 4: Move disk 3 from A to B

Step 1: Move disk 1 from A to B

Step 5: Move disk 1 from C to A

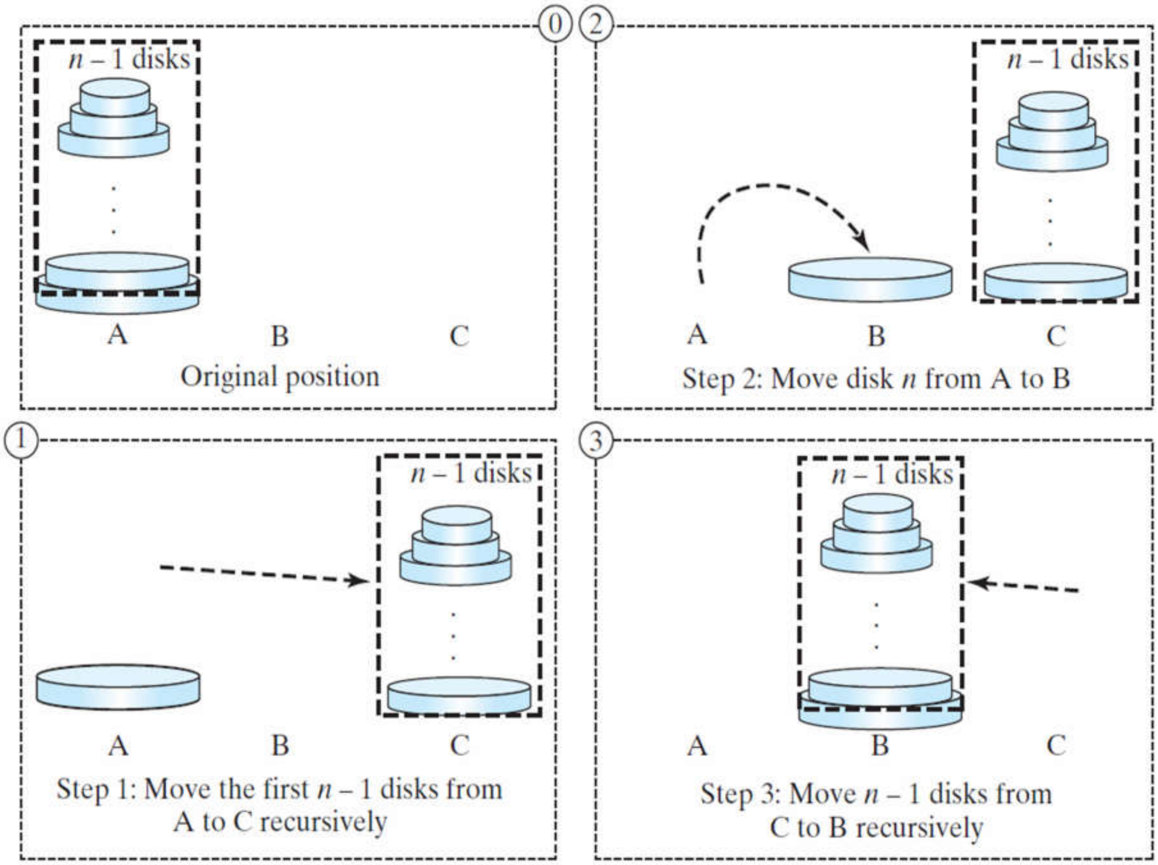
Step 2: Move disk 2 from A to C

Step 6: Move disk 2 from C to B

Step 3: Move disk 1 from B to C

Step 7: Move disk 1 from A to B

The Tower of Hanoi problem can be decomposed into three sub-problems.



Original position

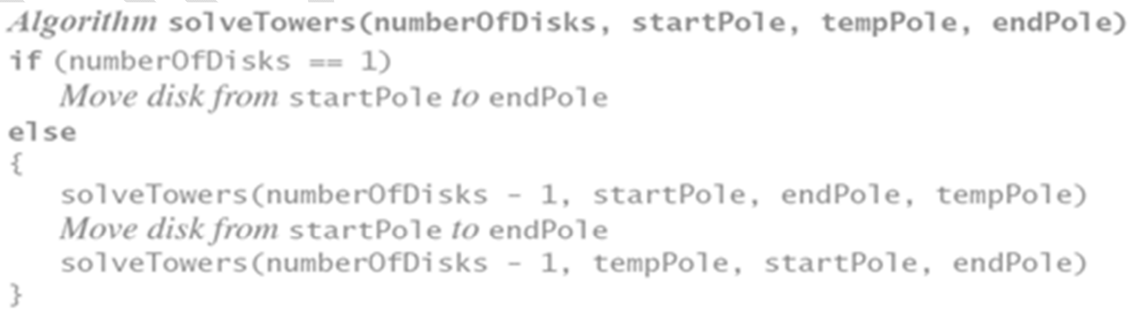
Step 2: Move disk *n* from A to B

Step 1: Move the first *n* – 1 disks from A to C recursively

Step 3: Move *n* – 1 disks from C to B recursively

- Move the first **n-1** disks from **A** to **C** with the assistance of tower **B**.
- Move disk **n** from **A** to **B**.
- Move **n-1** disks from **C** to **B** with the assistance of tower **A**.

Solutions:



```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

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.

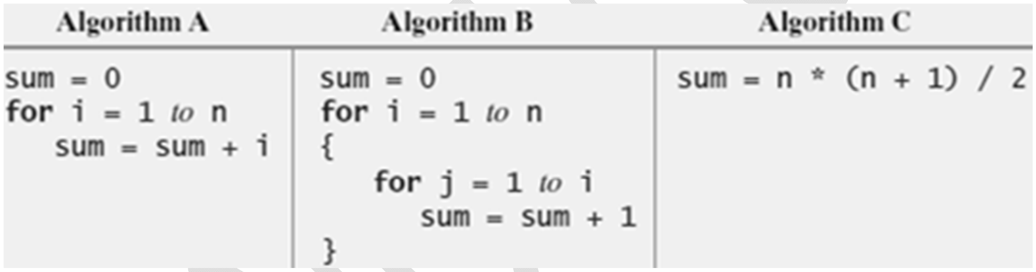
- **Space Complexity ➔** memory and storage are very cheap nowadays. ✗
- **Time Complexity** ✓ Different platforms ➔ different time. Absolute time is hard to measure as it depends on many factors.

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 $\sum_{k=1}^{n} k = 1+2+3+...+n$

Come up with an algorithm to solve this problem.

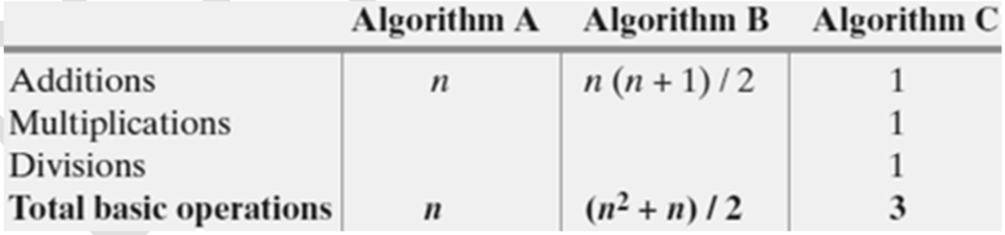


Algorithm A	Algorithm B	Algorithm C

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

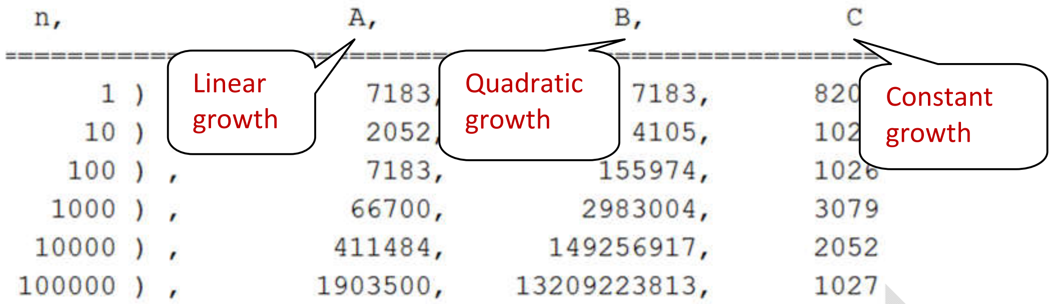
- Measure execution time. ✗ Algorithm for small data size will take small time comparing to a large data.
- Calculate time required for an algorithm in terms of the size of input data. ✗ Does not work as the same algorithm over the same data will not take the same time.

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

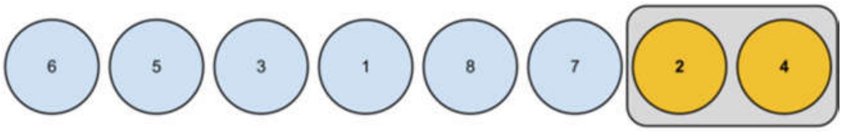
RAM model of computation

We assume that:

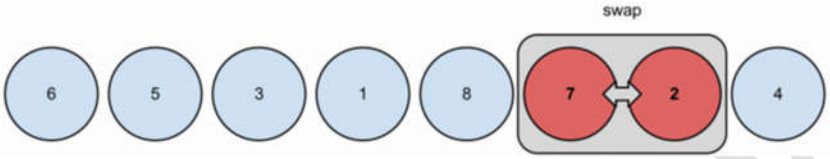
- We have infinite memory
- Each operation (+,-,*,/,=) takes 1 unit of time
- Each memory access takes 1 unit of time
- All data is in the RAM

Bubble Sort:

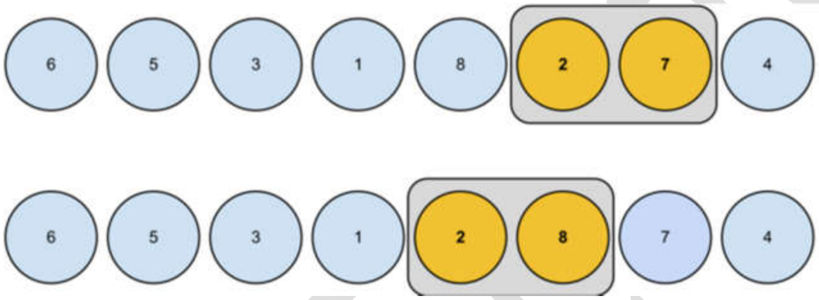
1. Each two adjacent elements are compared:



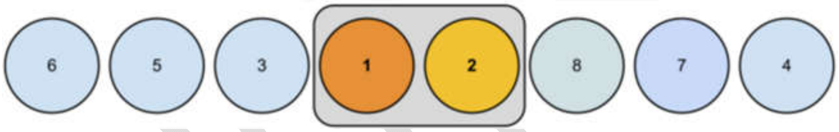
2. Swap with larger elements:



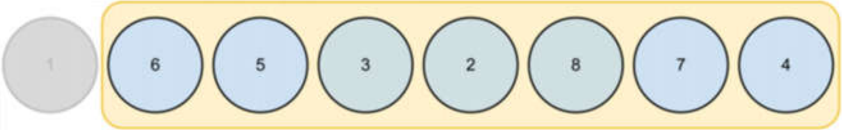
3. Move forward and swap with each larger item:



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:



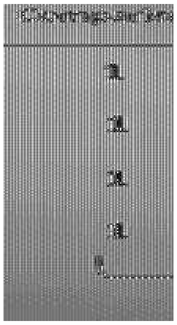
Make a demo using the following data set

12	8	7	5	2

Worst case analysis

After 1st round:

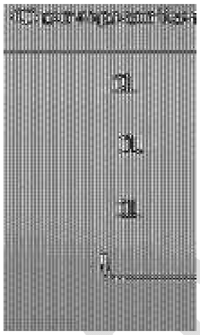
8	7	5	2	12



Comparisons
1
1
1
1

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

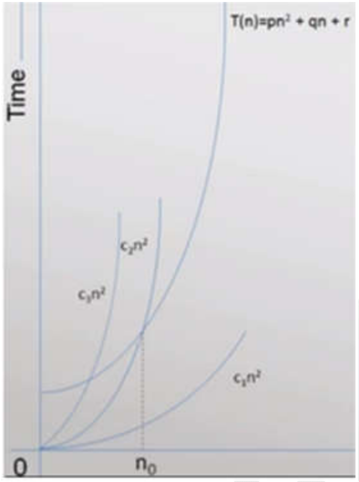
for (**int** i = 0; i < arr.**length**-1; i++) {
  for (**int** j = 0; j <arr.**length**-i-1 ; j++) {
    if(arr[j+1]<arr[j]){
      temp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = temp;
    }
  }
}
	i=0
	i=1
	:
	:
	i=n-1
		j=n-1
		j=n-2
		:
		:
		j=0
			n-1
			n-2
			:
			:
			1

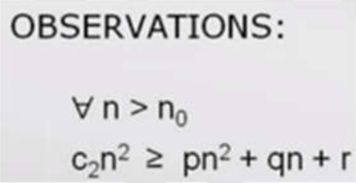
The Big-O Notation

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 $\mathbf{c_1n^2}$ 🡐 never over take **T(n)**

$\mathbf{C_2n^2}$ such that its greater than **T(n)** for $\mathbf{n>n_0}$. In this case we say that $\mathbf{C_2n^2}$ 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:

$\mathbf{5n^2 + 6} \in \mathbf{O(n^2)}$??? ✓

Find $\mathbf{cn^2}$ ➔ c=6 and n_0=3

➔ c=5.1 n_0=8

Example 2:

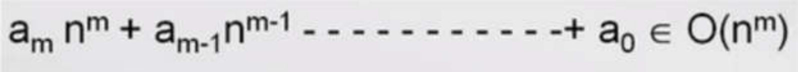
$\mathbf{5n + 6} \in \mathbf{O(n^2)}$??? ✓

Find $\mathbf{cn^2}$ ➔ c=11 and n_0=1

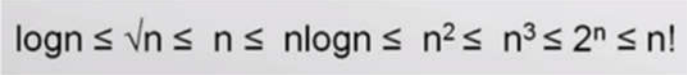
Example 3:

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

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



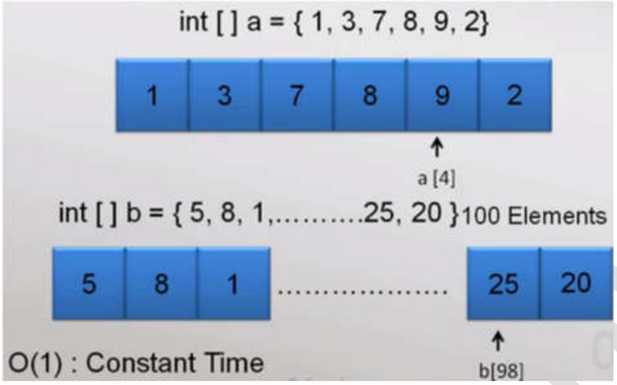
$$a_m\, n^m + a_{m-1}n^{m-1} \text{ - - - - - - - - - - - -+ } a_0 \in O(n^m)$$



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

What does it mean?

Array element access:

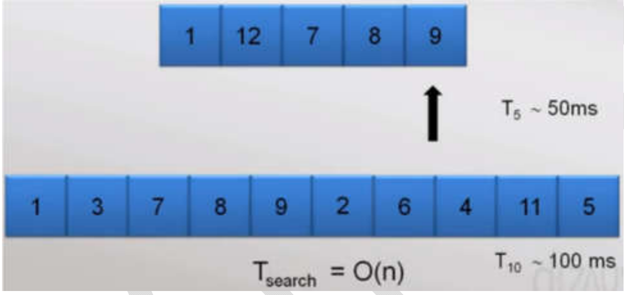


int [ ] a = { 1, 3, 7, 8, 9, 2}

int [ ] b = { 5, 8, 1,..........25, 20 }100 Elements

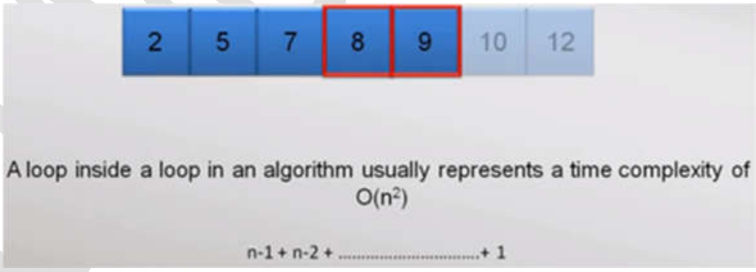
O(1) : Constant Time

Array element search:



T_{search} = O(n)

Bubble sort algorithm:



A loop inside a loop in an algorithm usually represents a time complexity of $O(n^2)$

n-1 + n-2 + ...+ 1

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.

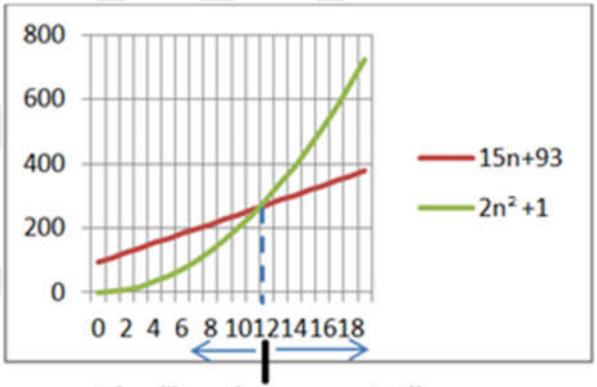
Worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well.

Example:

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

Algorithm B: time = **$2n^2 + 1$** **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 **$15n + 93 \le c(n)$** for **$n \ge n_0$**

  <provided **n= 93**> ➔ **15n+n** ➔ **$16n \le cn$** → <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

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 **$15n+93 \ge c(n)$** for **$n \ge n_0$**

  <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

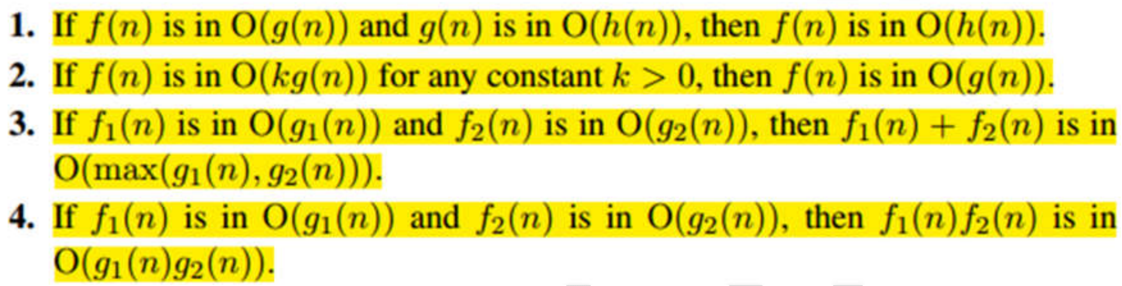
  Graph using Excel

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



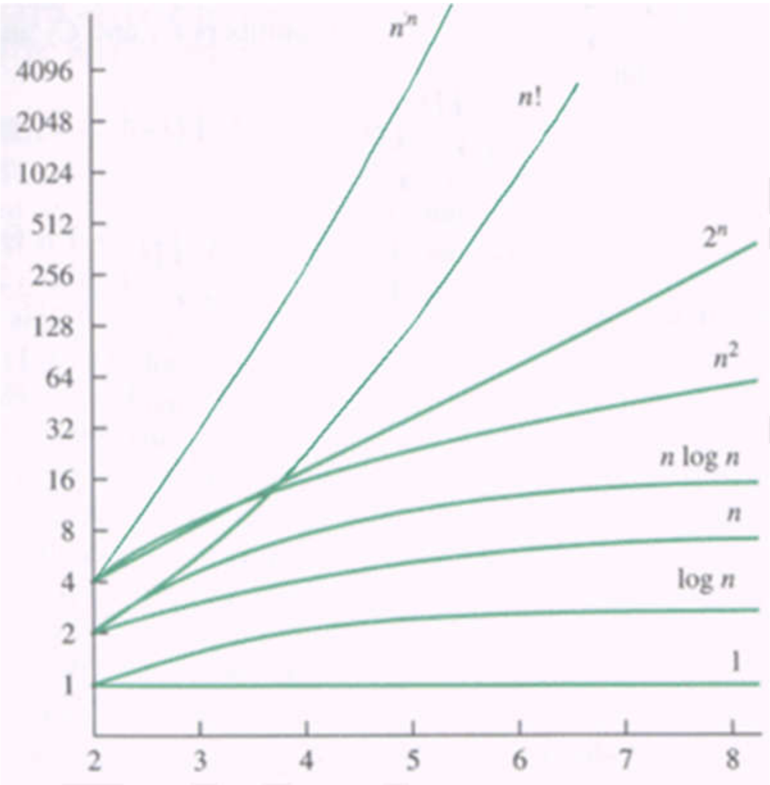
1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.
2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.
3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then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

- **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:

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



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)

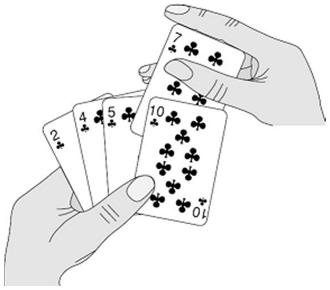
```
public static void bubble(int[] arr){
  int temp;
  for (int i = 0; i < arr.length-1; i++) {
    for (int j = 0; j <arr.length-i-1 ; j++) {
      if(arr[j+1]<arr[j]){
        temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
}
```

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;
   }
}
```



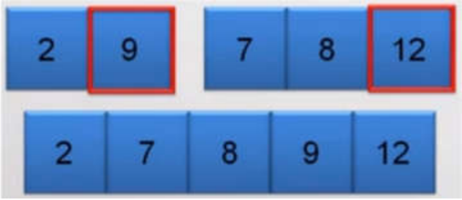
O(n^2) sorting algorithms comparison:

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

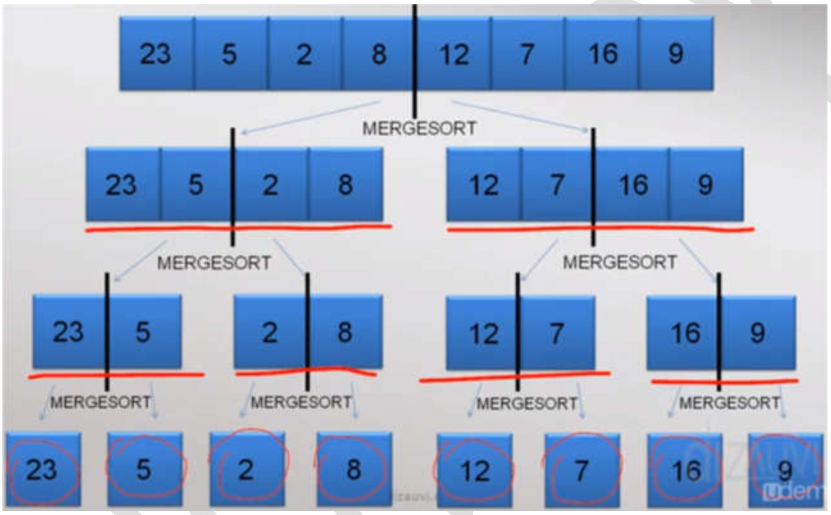
Bubble Sort	Selection Sort	Insertion Sort
Very inefficient	• Better than bubble sort • Running time is independent of ordering of elements	• Relatively good for small lists • Relatively good for partially sorted lists

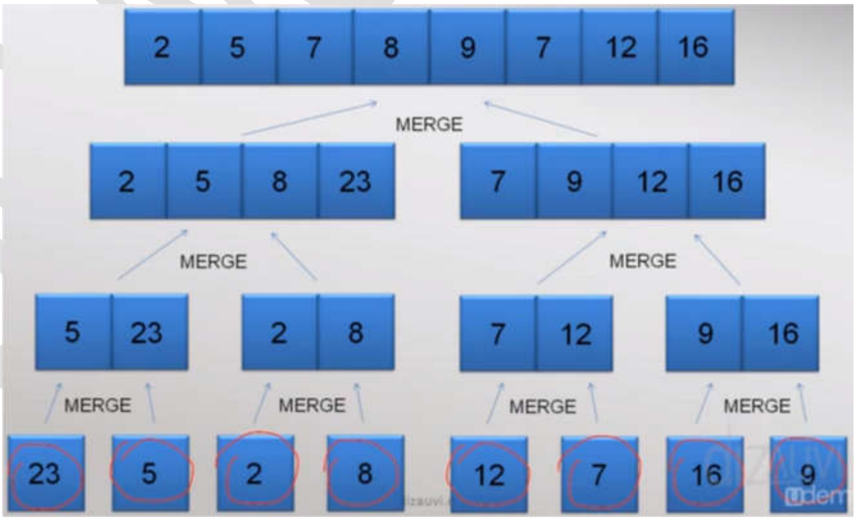
Merge sort: recursive algorithm

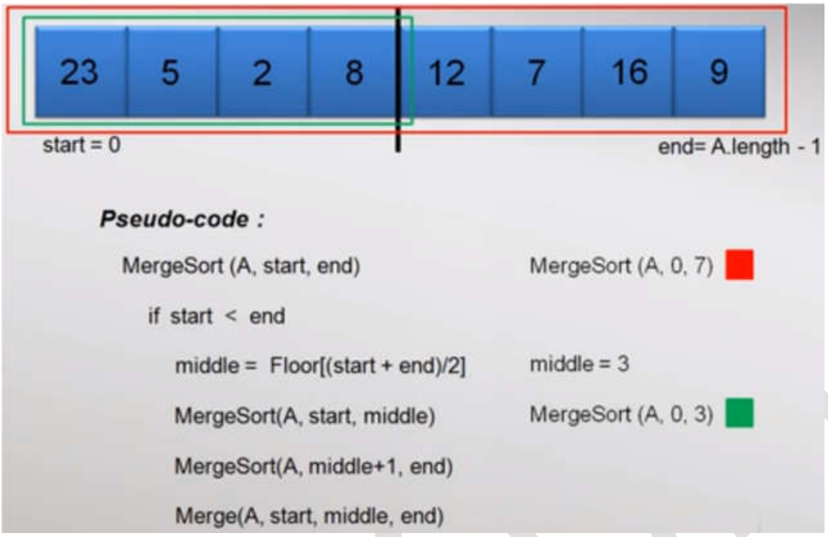
Merge: take 2 sorted arrays and merge them together into one.



Example:



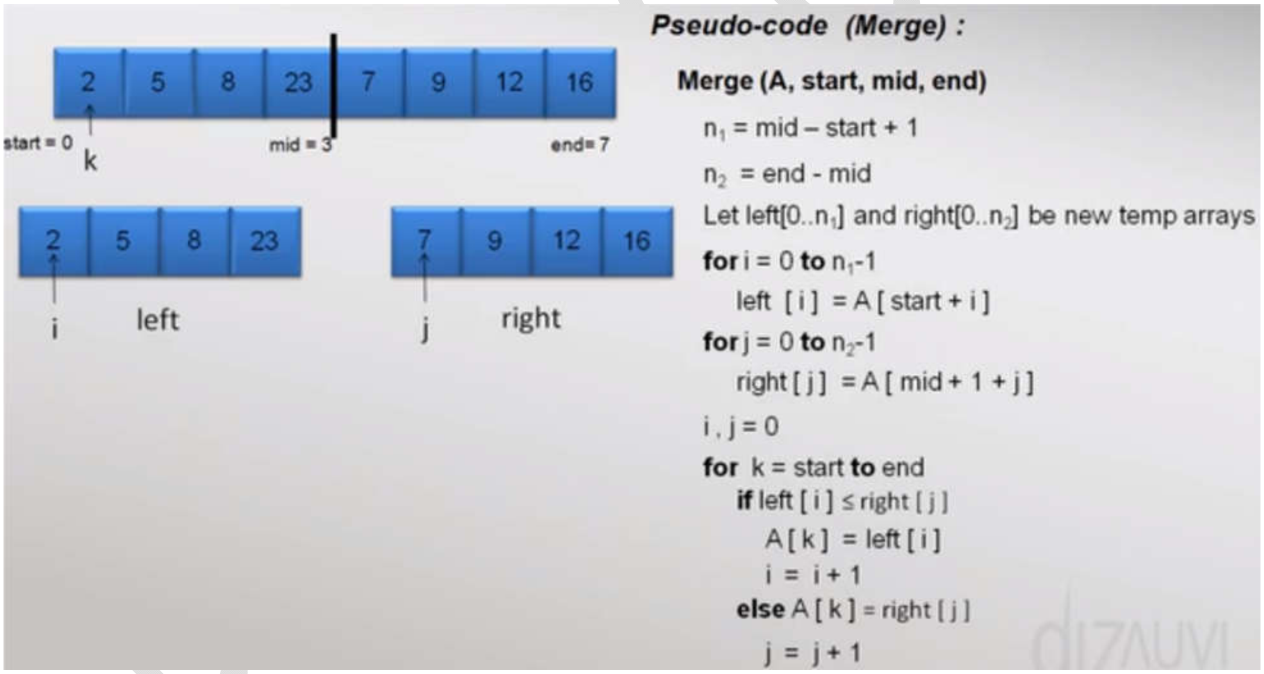




Pseudo-code :

```
MergeSort (A, start, end)                      MergeSort (A, 0, 7) 🟥
    if start < end
        middle = Floor[(start + end)/2]        middle = 3
        MergeSort(A, start, middle)            MergeSort (A, 0, 3) 🟩
        MergeSort(A, middle+1, end)
        Merge(A, start, middle, end)
```

Pseudo code:



Pseudo-code (Merge) :

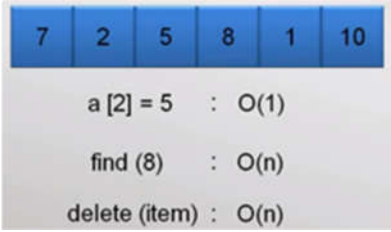
```
Merge (A, start, mid, end)
    $n_1$ = mid – start + 1
    $n_2$ = end - mid
    Let left[0..$n_1$] and right[0..$n_2$] be new temp arrays
    for i = 0 to $n_1$-1
        left [ i ] = A [ start + i ]
    for j = 0 to $n_2$-1
        right [ j ] = A [ mid + 1 + j ]
    i , j = 0
    for k = start to end
        if left [ i ] $\le$ right [ j ]
            A [ k ] = left [ i ]
            i = i + 1
        else A [ k ] = right [ j ]
            j = j + 1
```

Make sure of array boundaries

H.W: implement merge sort your own

Searching elements in an array:

Case 1: unordered array:



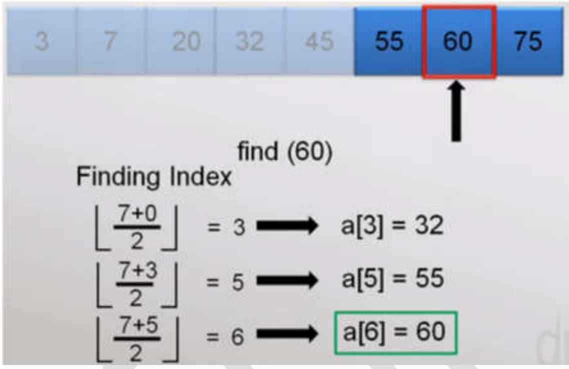
7	2	5	8	1	10

a [2] = 5 : O(1)

find (8) : O(n)

delete (item) : O(n)

Case 2: ordered array: -Binary search-



3	7	20	32	45	55	60	75

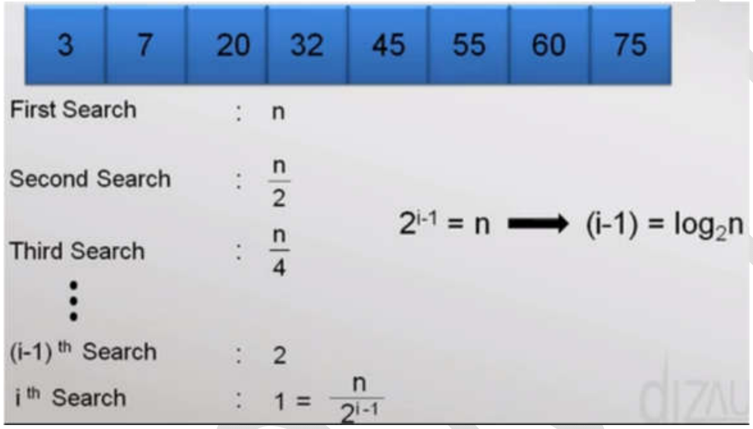
find (60)

Finding Index

$$\left\lfloor \frac{7+0}{2} \right\rfloor = 3 \longrightarrow a[3] = 32$$

$$\left\lfloor \frac{7+3}{2} \right\rfloor = 5 \longrightarrow a[5] = 55$$

$$\left\lfloor \frac{7+5}{2} \right\rfloor = 6 \longrightarrow \boxed{a[6] = 60}$$



3	7	20	32	45	55	60	75

First Search : n

Second Search : $\frac{n}{2}$

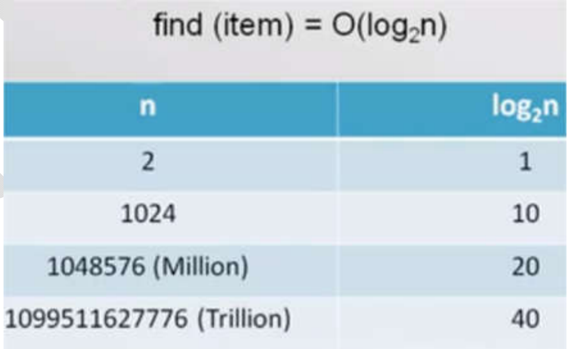
Third Search : $\frac{n}{4}$

⋮

$(i-1)^{th}$ Search : 2

i^{th} Search : $1 = \frac{n}{2^{i-1}}$

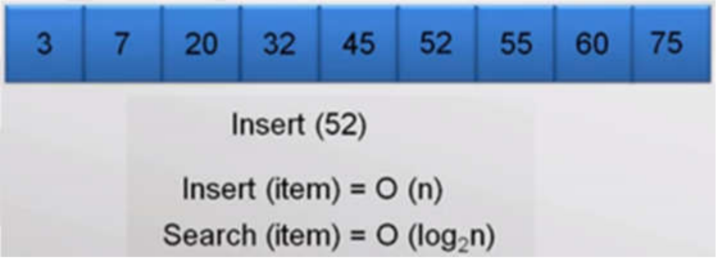
$$2^{i-1} = n \longrightarrow (i-1) = \log_2 n$$



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

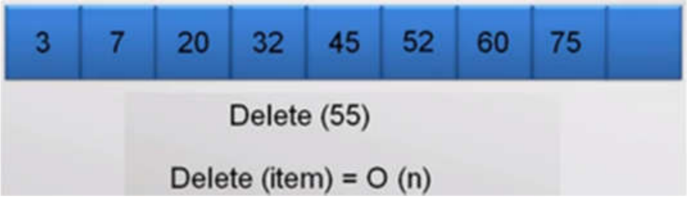


3	7	20	32	45	52	55	60	75

Insert (52)

Insert (item) = O (n)

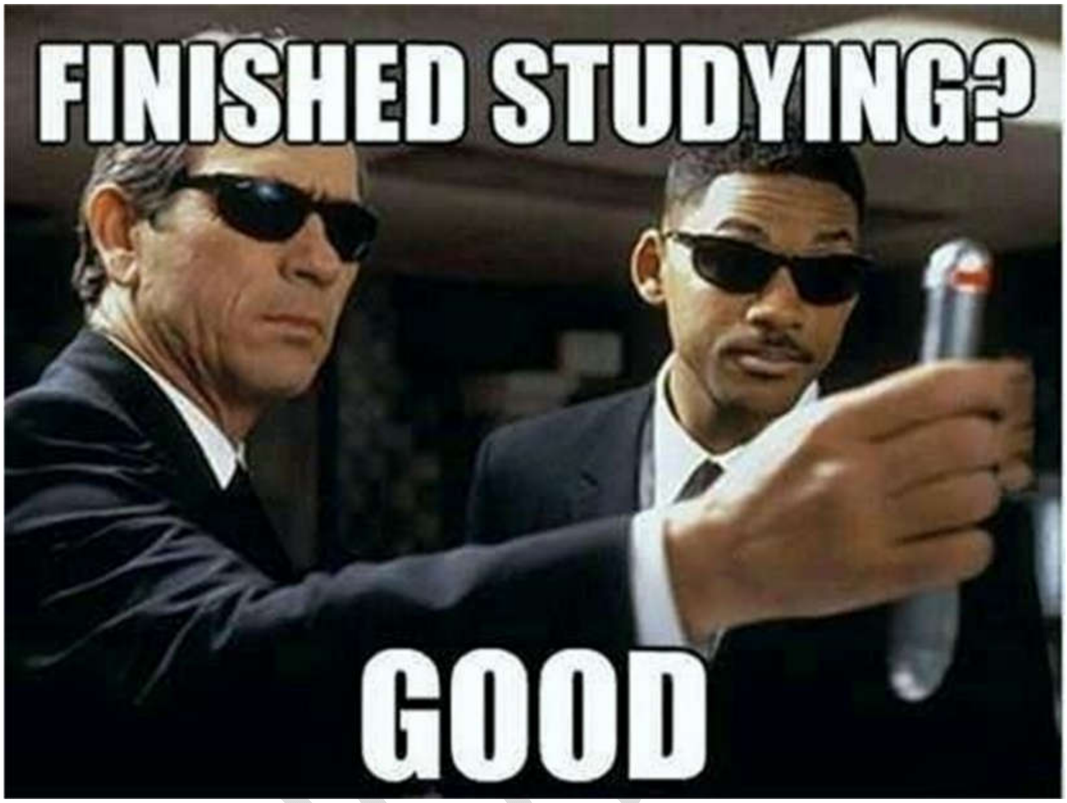
Search (item) = O $(\log_2 n)$



3	7	20	32	45	52	60	75	

Delete (55)

Delete (item) = O (n)


FINISHED STUDYING?
GOOD