Algorithm Analysis - Introduction

Tanjina Helaly
CSI 207: Algorithms
Department of Computer Science and Engineering
University of Asia Pacific

What is Algorithm?

From Dictionary

- A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

What is Algorithm?

- A set of steps to complete a task.

- **Task**: You need to buy a TV.
  - Algorithm 1:
    - Take a transport and go to market.
    - Find the/a store that sells TV.
    - Choose the TV you want to buy
    - Pay the price.
    - Bring the TV to your home.
  - Algorithm 2:
    - Browse an Online shop
    - Choose the TV you want to buy
    - Check the price.
    - If the price matches
      - Pay the price
      - Deliver to your Address
    - Else
      - Start from beginning.

What is Algorithm – In Computer Science?

- **An *algorithm is any well-defined computational procedure that***
  - ***Takes*** some value, or set of values, as ***input and***
  - ***produces some value, or set of values, as output.***

- *An* ***algorithm*** *is thus* ***a sequence of computational steps*** *that* ***transform*** *the* **input** into the **output**.

Origin of the word "Algorithm"

- Can be traced back to the 9th century
- Abdullah Muhammad bin Musa al-Khwarizmi,
  - Persian scientist, astronomer and mathematician
  - Known as "The father of Algebra"
  - The word "Algorithm" came from his name.
- In 12th century, one of his book was translated in Latin
  - There his name was written as "Algoritmi" in Latin, which means "the decimal number system"
- In English, the word "**algorithm**" emerged in the 19th century.

When did we start using algorithm?

- From the beginning of Human life, we are using algorithm at every step of our life.

- But on paper – during the beginning of modern mathematics.
  - Euclid, Nicomachus of Gerasa expressed their ideas in step by step actions.
    - Euclids Life Span –**Mid 4thcentury BC – Mid 3rd century BC.**

Where do we need Algorithm?

- Algorithms is everywhere.
  - Starts with very basic mathematical/calculation algorithms
  - To many modern technologies (BUZZ words)
    - Artificial intelligence
    - Machine learning
    - Neural Network
    - Data Science
    - Blockchain
    - Molecular biology.

Where do we need Algorithm? – Example

- Internet - Finding route, **Search (Google)**
- Google Map
- E-commerce
  - Customer pattern, market basket algorithm
- Weather pattern
- **Sorting** – Insertion Sort, Merge Sort, Bubble Sort….
- Find Min(Shortest path), Max(Maximize Profit)
- Scheduling
- The Human Genome
  - Identifying all the 100,000 genes in human DNA
  - Cross matching

Why do we need to learn about Algorithm ?

- Identifying **how good** an algorithm is?

- **When** to apply an algorithm?
  - This will save your time and make your program faster by applying the right one.
  - Example (BFS vs. DFS)

- How to **design** new algorithm?
  - Analyze their **correctness** and **efficiency**

What makes an Algorithm good?

- Correctness
- Efficiency
  - Run faster

A Simple Problem

- Assume you are asked to write an algorithm to find the GCD of 2 integers.
  - Algorithm 1: Using Prime factor
    - Calculate the common prime factors of the 2 numbers and multiply the factors.
  - Algorithm 2(Euclidean method): Using Divisor

Algorithm 1: Using Prime factor

- Calculate Prime Factors of number n1
- Calculate Prime Factors of number n2
- Identify the common factors
- Multiply the factors and return

- Example: n1 = 8, n2 = 36
  - 8= 2x2x2
  - 36 = 2x2x3x3
  - Common factors 2,2
  - GCD = 2x2 = 4

Algorithm 2(Euclidean method)

- Algorithm
  - While n1 is not divided by n2
    - Calculate remainder r = n1 mod n2
    - n1=n2
    - n2=r
  - Return n2
- Example: n2 = 8, n1= 36
  - Iteration 1
    - n1 is not divided by n2
    - r= 36%8 = 4
  - Iteration 2
    - n1(8) is divided by n2(4)
    - Return 4 as gcd.

Which one will you choose for your program?

Algorithm 1: Using Prime factor

- Example: n1 = 8, n2 = 36
  - 8= 2x2x2 *Needs ~3 divisions & 3 checks for prime*
  - 36 = 2x2x3x3 *Needs ~4 divisions & 4 checks for prime*
  - Common factors 2,2 *Needs ~3 checks for common prime*
  - GCD = 2x2 = 4 *Needs 1 multiplication*

Total ~18 operations

Algorithm 2(Euclidean method)

- Example: n2 = 8, n1= 36
  - Iteration 1
    - n1 is not divided by n2
    - r= 36%8 = 4
  - Iteration 2
    - n1(8) is divided by n2(4)
    - Return 4 as gcd.

Needs 1 check and 1 division

Needs 1 check

Total ~3 operations

Algorithm of Leap Year

LeapYear1()

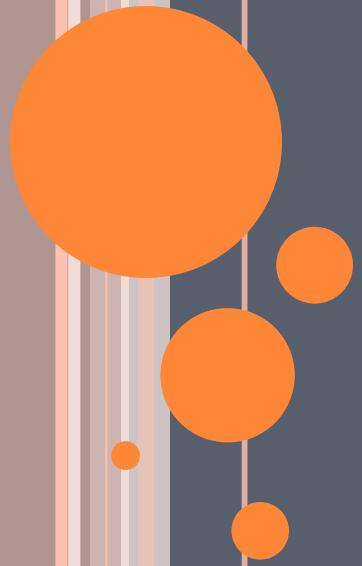
If n is not divisible by 4 then
    n is not a leap year
Elseif n is not divisible by 100 then
    It is a leap year
Elseif it is not divisible by 400 then
    It is not a leap year
Else
    It is a leap year.
Endif

LeapYear2()

If n is divisible by 400 then
    n is not a leap year
Elseif n is divisible by 100 then
    It is a leap year
Elseif it is divisible by 4 then
    n is a leap year
Else
    n is not a leap year.
Endif

Leap Year divisible by 4 and 400 but not 100

Algorithm Analysis



Algorithm Analysis

- ***Analyzing an algorithm is***
  - ***predicting the resources*** that the algorithm requires.
    - Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, **but most often it is computational time** that we want to measure.
    - **Question: How do we measure time?**

Algorithm Analysis

- **Why do we need analysis:**
  - Generally, from **several** candidate algorithms -> identify **a** most efficient one.
  - Such analysis may indicate **more than one viable** candidate,
  - but we can often **discard several inferior** algorithms in the process.

Let's take a look at a problem

- Assume there are 50 students in your section and your teacher is storing the ID of the students who have enrolled for CSE 207. You are asked to find if a particular student has enrolled in the course or not.

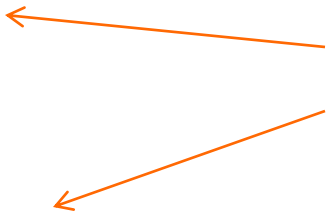
  Enrolled Students (Count = 21)

  1,3, 4,6, 7, 10,15, 17,18, 19,21, 22, 25,30, 33,34, 35,44, 46,49

Solutions?

- **Solution 1:**

```
for (i =0; i<21; i++)
    if array[i] == key (id we are looking for)
        return true;
    return false;
```



Maximum how many times are we searching?

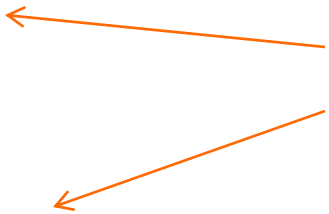
- **Solution 2:**
-

```
for (i =0; i<21; i++)
    mid = mid point of the array
    if array[mid] == key
        return mid
    else if key<array[mid]
        search in lower half of the array (upto mid-1)
    else
        search in the upper half (index starting from mid+ 1)
```

Generalized Solutions?

- **Solution 1:**

```
for (i =0; i<n; i++)
    if array[i] == key (id we are looking for)
        return true;
    return false;
```



Maximum how many times are we searching?

- **Solution 2:**
-

```
for (i =0; i<n; i++)
    mid = mid point of the array
    if array[mid] == key
        return mid
    else if key<array[mid]
        search in lower half of the array (upto mid-1)
    else
        search in the upper half (index starting from mid+ 1)
```

The classical problem that maps to this scenario.

- Find if a specific integer appear in an array.

```
for i = 0 to array length - 1
    If X = array[i] then
      return true
return false
```

Analysis of the Algorithm

(find how much time it will take to run.)

Analysis of Algorithm – RAM Model (Random-Access Machine)

- Instructions are executed one after another, with **no concurrent** operations.
- Each such instruction **takes a constant amount** of time.
- The **total time** an algorithm takes is the summation of time taken by each instruction.

Analysis of Algorithm – RAM Model (Random-Access Machine)

- The RAM model contains **instructions** commonly found in real computers:
  - Arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
  - Data movement (load, store, copy), and
  - Control (conditional and unconditional branch, subroutine call and return).

RAM Model – Time Complexity

Line#	Instruction	Cost	# Times this line executes
1	for i = 0 to array length - 1	c1	22
2	if X = array[i] then	c2	21
3	return i	c3	1
4	return -1	c4	0

- T = 22c1+21c2+c3

RAM Model – Time Complexity

Line#	Instruction	Cost	# Times this line executes
1	for i = 0 to array length - 1	c1	n+1
2	if X = array[i] then	c2	n
3	return i	c3	1
4	return -1	c4	0

- For array size = n,
- Total time T(n) = c1*(n+1) + c2*n + c3
  = (n+1)c1 + nc2 +c3
  = (c1+c2)n +(c1+c3)
  = cn +d [Linear to size of the input]
  = O(n) [Order of n]

RAM Model – Time Complexity

- Now think about some scenario
  - The value we are looking for is at the beginning of the Array
    - Known as **Best Case** as minimum time is required to execute.
  - The value we are looking for is not in the Array
    - Known as **Worst Case** as maximum time is required to execute.

RAM Model – Time Complexity

- **Average case:**
  - the amount of some computational resource (typically time) used by the algorithm, averaged over all possible inputs.
  - Similar to **worst case for input size = ½ of original input.**

RAM Model – Time Complexity

Line#	Instruction	Cost	Best	Worst
1	for i = 0 to array length - 1	c1	c1	(n+1)c1
2	if X = array[i] then	c2	c2	nc2
3	return i	c3	1	0
4	return -1	c4	0	1

- So
  - T_{best} = c1 + c2 + c3 -> **constant time**
  - T_{worst} = c1 *(n+1) + c2*n + c3 -> **Linear function of n**
    = (c1+c2)*n + (c1+c3)
    = an+b
  - $T_{average}$ = an/2+b = a1n+b -> **Liner function of n.**

So, what is algorithm analysis?

- To analyze an algorithm means:
  - **developing** a formula for predicting ***how fast*** an algorithm is, based on the **size of the input** (**time complexity**), and/or
  - **developing** a formula for predicting ***how much memory*** an algorithm requires, based on **the size of the input** (**space complexity**)
- Usually **time** is our biggest concern

Try these – Find the Best/Worst case time complexity

- **ArraySum**

```
int sumArray(int[] ar, int l){
    int sum = 0;
    for(int i=0; i<l;i++){
            sum += ar[i];
    }
    return sum;
}
```

Complexity (Best & Worst),
$T(n) = an + b = O(n)$

- **ArrayMax**

```
int maxArray(int[] ar, int l){
    int max = ar[0];
    for(int i=1; i<l;i++){
        if(max<ar[i])
        max = ar[i];
    }
   return max;
}
```

Complexity (Best & Worst),
$T(n) = an + b = O(n)$

Try these – Find the Best/Worst case time complexity

- **Matrix Sum**

```
int sumSquareMatrix(int[][] array, int n){
    int sum=0;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
                    sum+=array[i][j];
        }
    }
    return sum;
}
```

Complexity (Best & Worst),
$T(n) = an^2 + b = O(n^2)$

- **Find Element**

```
int findItemInArray(int arr[], int n, int key){
    for (int i =n; i>=0; i--)  {
      if (arr[i] == key)
        return i;
    }
    return -1;
}
```

Complexity.
Best, $T(n) = c = O(1)$
Worst, $T(n) = an^2 + b = O(n^2)$

More Examples

```
for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
        print(i*j)
```

Complexity,
$T(n) = an^2 = O(n^2)$

More Examples

```
for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        print(i*j)
```

i	#times inner loop
1	1
2	2
3	3
…	…
n	n

Time Complexity,

$T(n) = 0+1+2+3+\ldots\ldots+n$

$= n(n+1)/2 = an^2 + bn + c$

$= O(?)$

Try These.

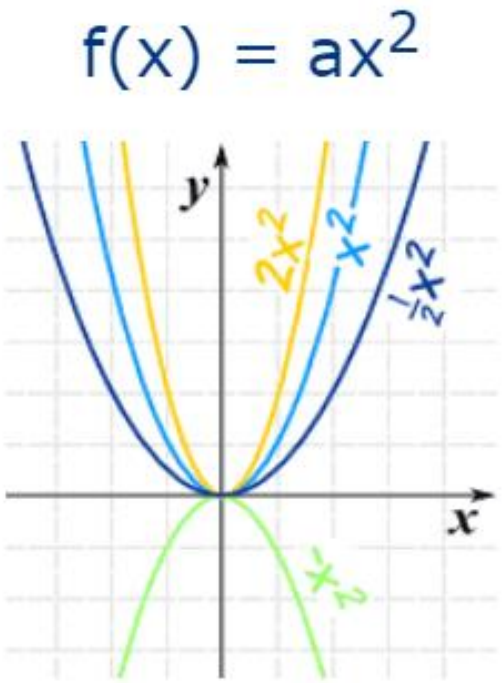
	Algo1	Algo2
Algorithm	for(i=1; i<n; i=i+2) print(i)	for(i=0; i<n; i+=2) print(i)
T(n)		
O(n)		

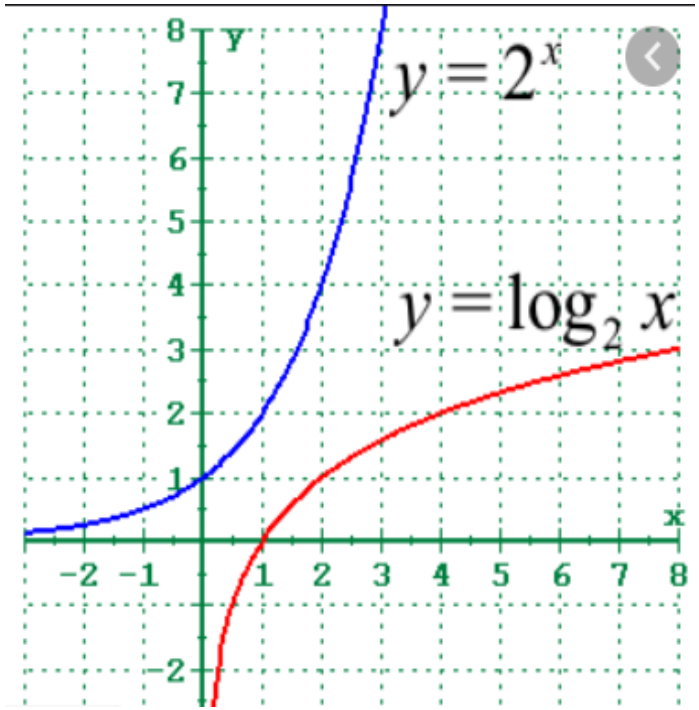
Different functions

- *Constant functions,* $f(n) = c$
- *Logarithmic functions,* $f(n) = \log n$
- *Linear functions,* $f(n) = n$
- *Superlinear functions,* $f(n) = n \lg n$
- *Quadratic functions,* $f(n) = n^2$
- *Cubic functions,* $f(n) = n^3$
- *Exponential functions,* $f(n) = c^n$
- *Factorial functions,* $f(n) = n!$
- *Comparison of Time taken for the above equations:*

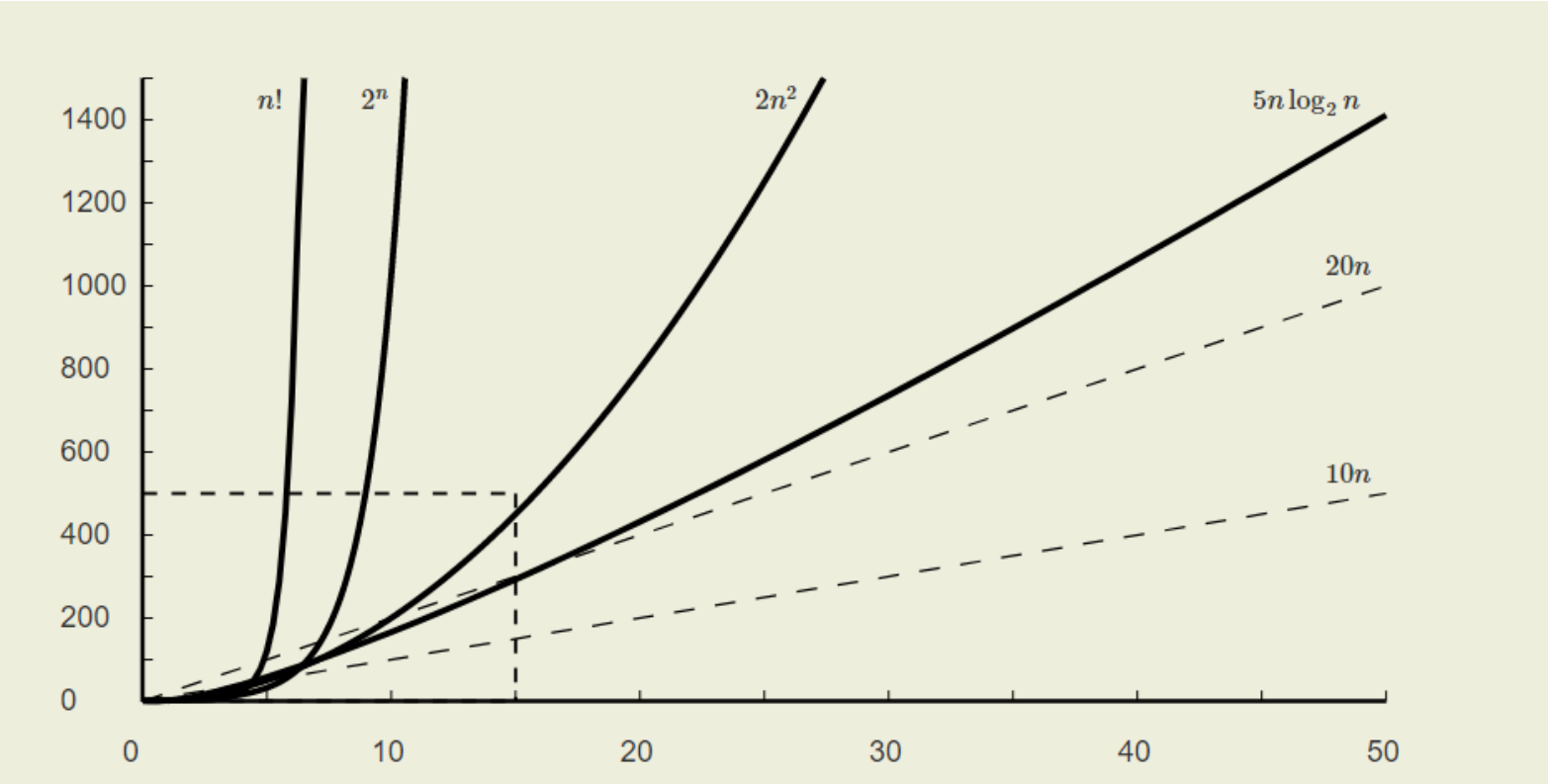
$$n! \geq 2^n \geq n^3 \geq n^2 \geq n \log n \geq n \geq \log n \geq c$$

Graphs of quadratic and logarithmic functions





Comparisons of some functions



Comparison of Growth rate

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

Another Comparison

- Let's compare 2 sorting algorithms; merge sort and insertion sort. Time complexity of the algorithms are as below.

- $T(n)_{merge\_sort} = cn\ lgn$ (Assume $50n\ lgn$)
  - faster than insertion sort
- $T(n)_{insertion\_sort} = cn^2$ (Assume $2n^2$)

Another Comparison

- Assume 2 computers are running 2 sorting algorithm

  - Computer A
    - **Faster computer** (1000 times faster than Computer B)
    - Execute 10 billion instructions per second
    - Running insertion sort – **slower algorithm**

  - Computer B
    - **Slower computer**
    - Executes 10 million instruction per second
    - Running merge sort - **faster algorithm**

Another Comparison

- To Sort 10 million data
  - Computer A will take (faster m/c running slower sort)

$$\frac{2.(10^7)^2 \, instructions}{10^{10} \, instructions/\sec} = 20000 \sec(more\ than\ 5.5\ hours)$$

  - Computer B will take (slower m/c running faster sort)

$$\frac{50.10^7 \lg 10^7 \, instructions}{10^7 \, instructions/\sec} \approx 1163 \sec(less\ than\ 20 \min)$$

- By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs 17 times faster than computer A.

Insertion Sort (Home Work)

Insertion Sort Algorithm

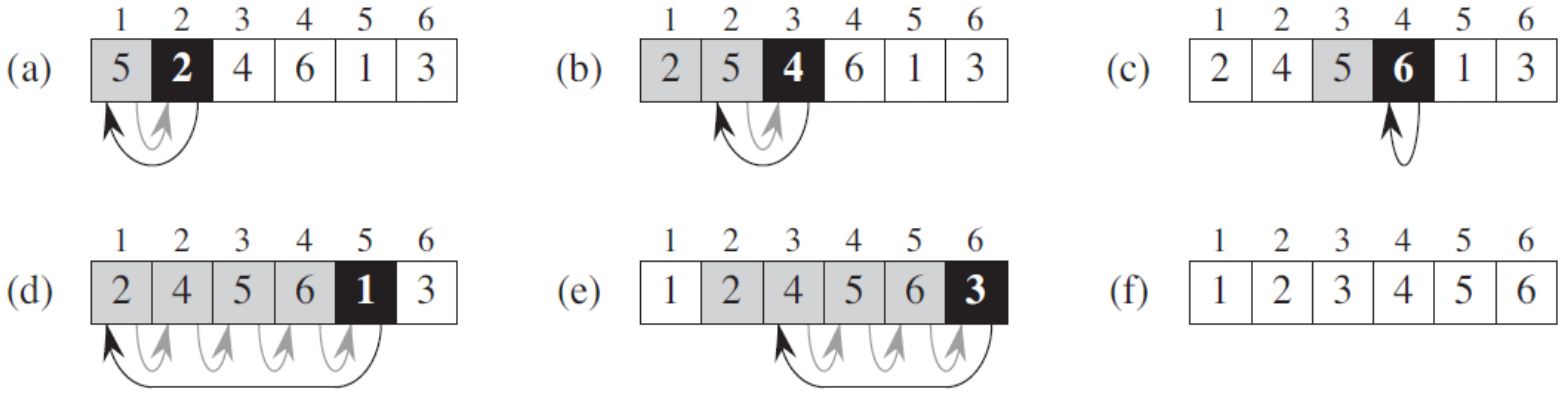


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Insertion Sort Algorithm

```
INSERTION-SORT(A)
    for j=2 to A.length
            key = A[j];
            // Insert A[j] into sorted sequence A[1…j-1]
            i=j-1;
            while i>0 and A[i]>key
                    A[i+1] = A[i];
                    i--;
            A[i+1] = key
```

Time Complexity – Insertion Sort

Line#	Instruction	Cost	Times	Best	Worst
1	for j=2 to A.length	c1	n	nc1	nc1
2	key = A[j];	c2	n-1	c2(n-1)	c2(n-1)
3	i=j-1;	c3	n-1	c3(n-1)	c3(n-1)
4	while i>0 and A[j]>key	c4	$\sum_{j=2}^{n} t_j$	c4(n-1)	$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
5	a[i+1] = a[i];	c5	$\sum_{j=2}^{n} (t_j - 1)$	0	$\sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$
6	i--;	c6	$\sum_{j=2}^{n} (t_j - 1)$	0	$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$
7	A[i+1] = key	c7	n-1	c7(n-1)	c7(n-1)

tj → the number of times the **while** loop test in line 4 executed for that value of j

- $So, running\ time\ T(n) = c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} t_j + c5\sum_{j=2}^{n}(t_j - 1) + c6\sum_{j=2}^{n}(t_j - 1) + c7(n-1)$

Best Case

- *Running time,*

$T(n) = c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} t_j +$
$c5\sum_{j=2}^{n}(t_j-1) + c6\sum_{j=2}^{n}(t_j-1) + c7(n-1)$

- **Best Case:**
  - List is already sorted. As a result the following will occur
    - While loop will check the condition **once** and return false as A[i]<key. Hence $t_j = 1$
      - So, $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1 = n - 1$
    - Also line 5 and 6 won't execute
  - So,
  - $T(n) = c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1)$
    $= (c1+c2+c3+c4+c7)n - (c2+c3+c4+c7)$
    $= an + b$ -> Linear Function

Worst Case

- $T(n) = c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} t_j +$ $c5\sum_{j=2}^{n}(t_j-1) + c6\sum_{j=2}^{n}(t_j-1) + c7(n-1)$
- List is sorted in reverse order. As a result the following will occur
  - While loop will check A[j] with entire subarray A[1… j-1].
  - So, $t_j = j$ for $j = 2,3,\ldots,n$
  - $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
  - $\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$

Worst Case

- So,

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4\left(\frac{n(n+1)}{2} - 1\right)$$
$$+ c5\left(\frac{n(n-1)}{2}\right) + c6\left(\frac{n(n-1)}{2}\right) + c7(n-1)$$

$= (c4/2+c5/2+c6/2)n^2 + (c1+c2+c3+c4/2-c5/2-c6/2+c7)n - (c2+c3+c4+c7)$

$= an^2 + bn + c$ -> Quadratic Function

Growth rate

Order Of Growth

- To ease analysis and focus on the important features we will do one more abstraction
  - Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.

Order Of Growth – Insertion Sort

- Worst-case running time is $an^2 + bn + c$.
- Now do the simplification
  - Drop lower-order terms$\Rightarrow an^2$.
  - Ignore constant coefficient $\Rightarrow n^2$.
- But we cannot say that the worst-case running time *T (n) equals* n^2.
  - It *grows like* n^2. *But it doesn't equal* n^2.
  - We say *that the order of growth* is n^2.
- Usually the algorithm with smaller order of growth is considered to be more efficient.

References

- Chapter 1+2 (Cormen)