Lecture (02)

Analysis of Non-recursive Algorithms

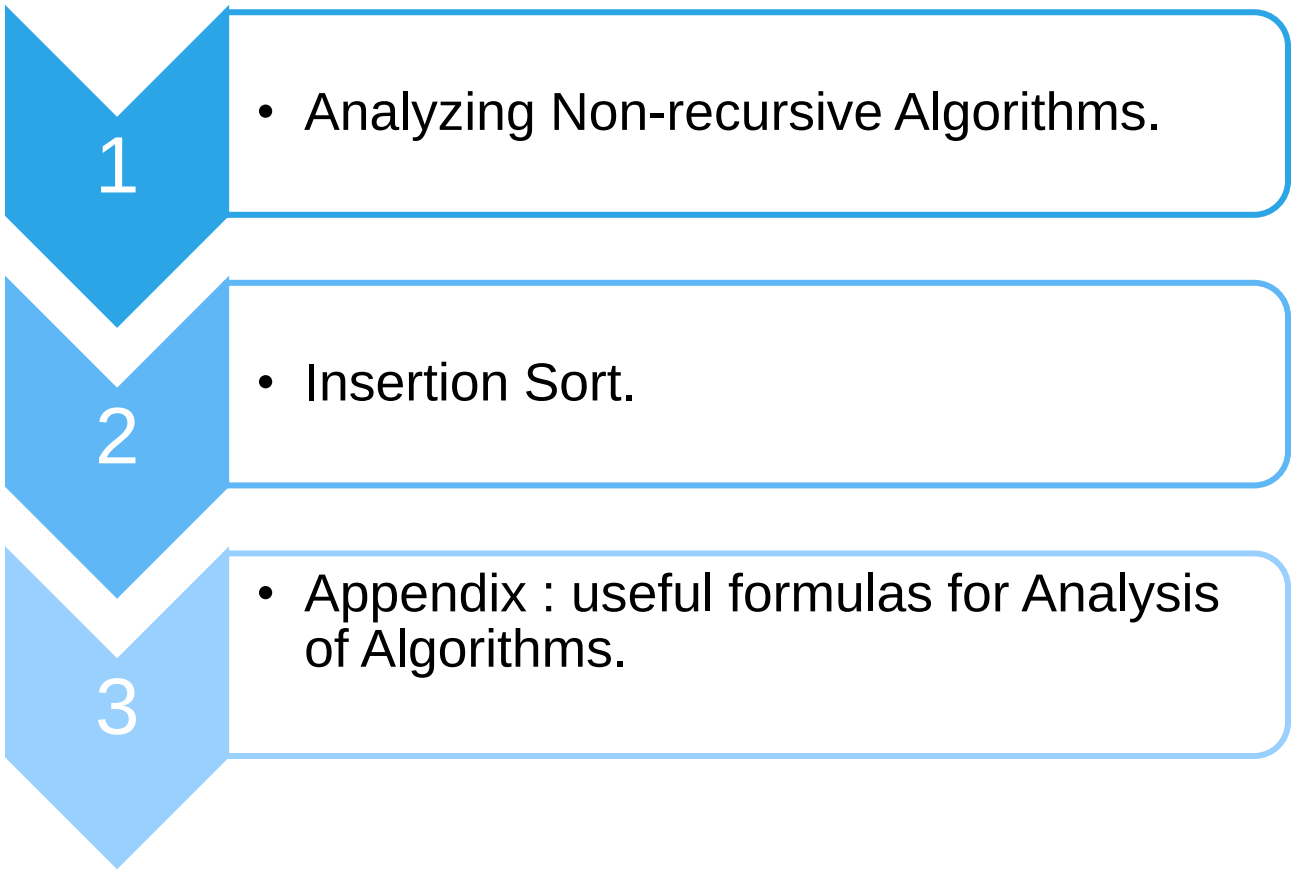
Dr. Ensaf Hussein
Assoc. Prof. of Artificial Intelligence

NU : Fall 2024
CIT645: Formal Methods and Computer Algorithms

DESIGN AND ANALYSIS OF ALGORITHMS

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost

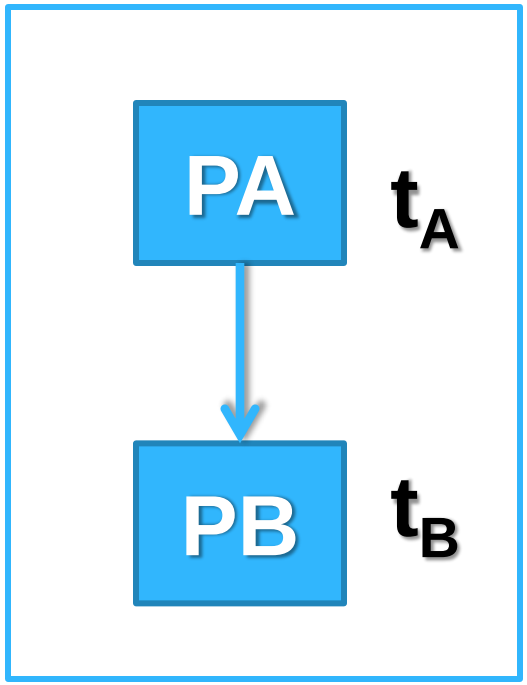

1
• Analyzing Non-recursive Algorithms.
2
• Insertion Sort.
3
• Appendix : useful formulas for Analysis of Algorithms.

ANALYZING NON-RECURSIVE ALGORITHMS

ANALYSIS ALGORITHM CONTROL STRUCTURES

- Sequencing
- If-then-else
- For loop
- While loop
- ~~Recursion~~

SEQUENCING



$t_A + t_B = \max(t_A, t_B)$

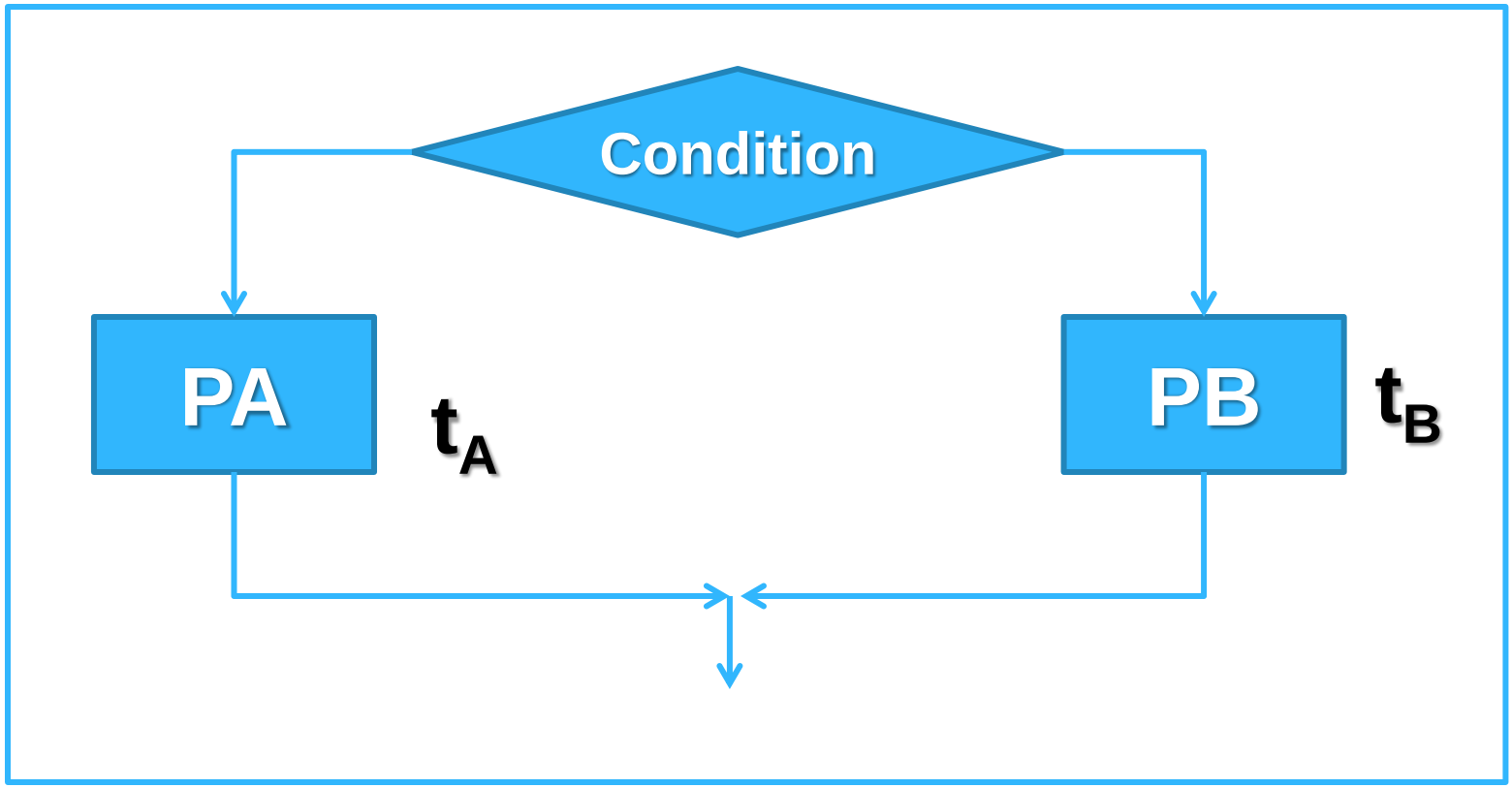
EX:

If $t_A = \theta(n)$, $t_B = \theta(n^2)$

➔ $t_A + t_B = \max(t_A , t_B)$

$= \theta(n^2)$

IF-THEN-ELSE



$$t_A/t_B = \max(t_A,t_B)$$

FOR LOOP

```
for i←1 to n
        {


        P(i)

                        }
```

$$\sum_{i=1}^{n} t_i$$

```
for i←1 to n
    {
     for j←1 to n
         {

         P(ij)

         }
    }
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij}$$

WHILE LOOP

```
While (n>0)
    {
      ----------

      ----------

     Set n←n-1

    }
```

$= \theta(n)$

```
While (n>0)
    {
      ----------

      ----------

     Set n ← n/2

    }
```

$= \theta(\log n)$

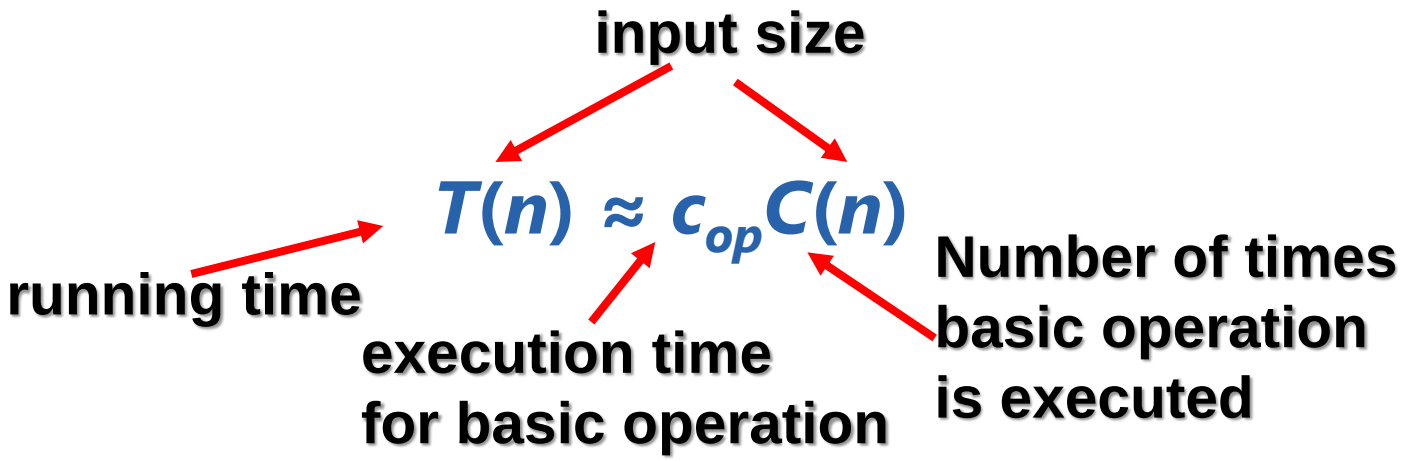
```
Set x← 0
Set y← n
While (x<y)
    {
      ----------

      ----------

    Set x ←x+1
    Set y ← y-1
       }
```

$= \theta(n)$

THEORETICAL ANALYSIS OF TIME EFFICIENCY

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

Basic operation: the operation that contributes most towards the running time of the algorithm



GENERAL PLAN

How to find $T(n) \approx c_{op}C(n)$??

- Decide on a parameter indicating an **input's size**
- Does C(n) depends **only on n** or does it also **depend on type** of input?
- Identify the **basic operation**
- **Set up sum** expressing the # of times basic operation is executed.
- **Find closed form for the sum or at least establish its order of growth**

ALGORITHMS TECHNIQUES

- Brute force(e.x. Sequential search)
- Decrease and Conquer (e.x. Insertion Sort)
- Divide and Conquer (e.x. Merge – Quick Sort)
- Transform and Conquer (Heap Sort)
- Dynamic Programming
- Greedy Technique.
- Approximation Algorithms
- NP Complete problems

Analysis of Non-recursive Algorithms

ALGORITHM MaxElement(A[0..n-1])
//Determines largest element
maxval <- A[0]
for i <- 1 **to** n-1 **do**
    if A[i] > maxval
        maxval <- A[i]
return maxval

Input size: n

Basic operation: > or <-

$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$

$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 - 1 + 1$
$= n - 1 \in \Theta(n)$

Analysis of Non-recursive Examples

ALGORITHM UniqueElements(A[0..n-1])

//Determines whether all elements are //distinct

for i <- 0 **to** n-2 **do**

    for j <- i+1 **to** n-1 **do**

        if A[i] = A[j]

            return false

return true

Input size: n

Basic operation: A[i] = A[j]

UniqueElements (contd.)

```
for i <- 0 to n-2 do
    for j <- i+1 to n-1 do
        if A[i] = A[j]
            return false
return true
```

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$$

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \approx \tfrac{1}{2}n^2$$

$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$

•

$$\begin{aligned} C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \end{aligned}$$

Analysis of Nonrecursive (contd.)

ALGORITHM MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])

//Output: C = AB

```
for i <- 0 to n-1 do
    for j <- 0 to n-1 do
        C[i, j] = 0.0
        for k <- 0 to n-1 do
            C[i, j] = C[i, j] + A[i, k]×B[k, j]
return C
```

Input size: n

Basic operation: ×

M(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$

Analysis of Nonrecursive (contd.)

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Input size: n

Basic operation: ×

M(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$$

If we took into account the time spent in the additions too

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

T(n) ≈ c_mM(n)+c_aA(n)

= c_mn3+$c_a$$n^3$

= (c_m+c_a)n^3

Analysis of Nonrecursive (contd.)

ALGORITHM Binary(n)

// Output: # of digits in binary

// representation of n

count <- 1

while n > 1 **do**

count <- count+1

n <- $\left\lfloor \frac{n}{2} \right\rfloor$

return count

Input size: n

Basic operation: $\lfloor - \rfloor$

C(n) = ?

Start with n:

After 1 iteration: $\lfloor n/2 \rfloor$
After 2 iterations: $\lfloor n/4 \rfloor$
After kk iterations: $\lfloor n/2^k \rfloor$

The loop stops when $\lfloor n/2^k \rfloor \leq 1$, which implies:
$n/2^k \leq 1 \implies n \leq 2^k \implies k \geq \log_2(n)$

Therefore, the number of iterations k is approximately $\lceil \log_2(n) \rceil$.

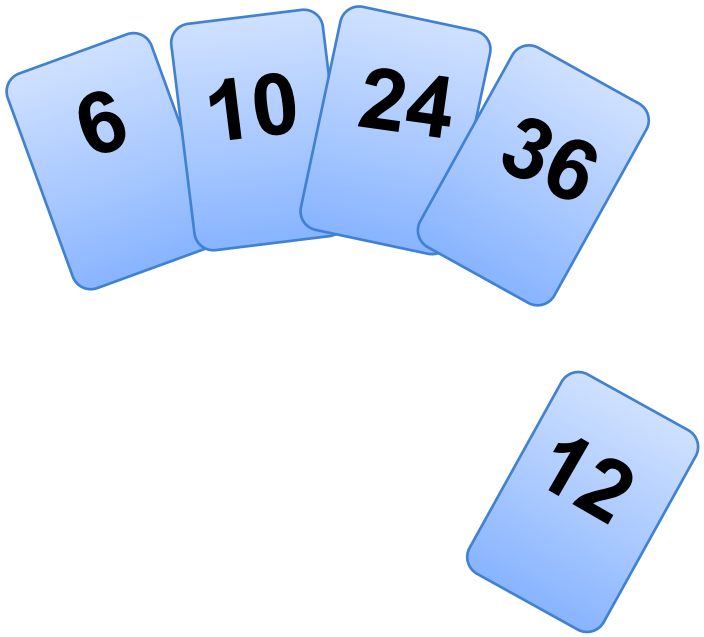
INSERTION SORT

INSERTION SORT

Idea: like sorting a hand of playing cards

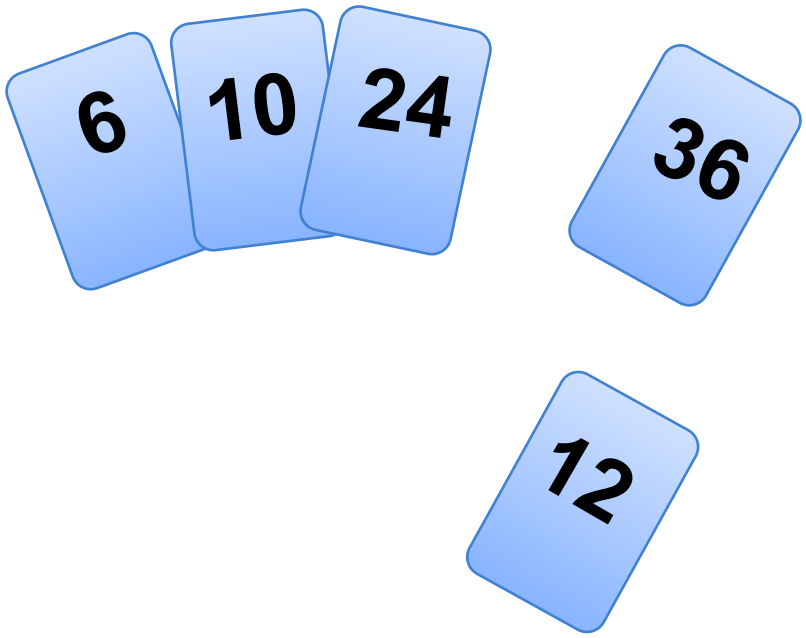
- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

INSERTION SORT

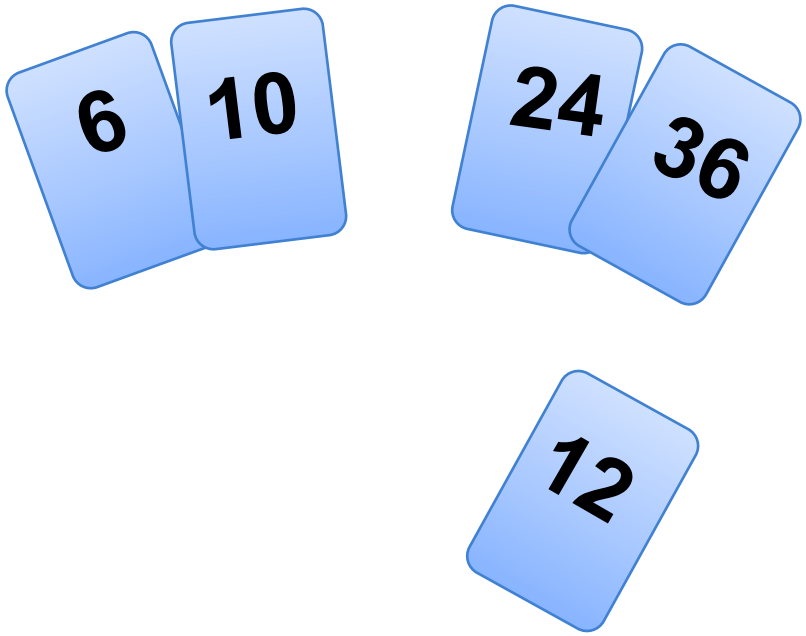


To insert 12, we need to make room for it by moving first 36 and then 24.

INSERTION SORT



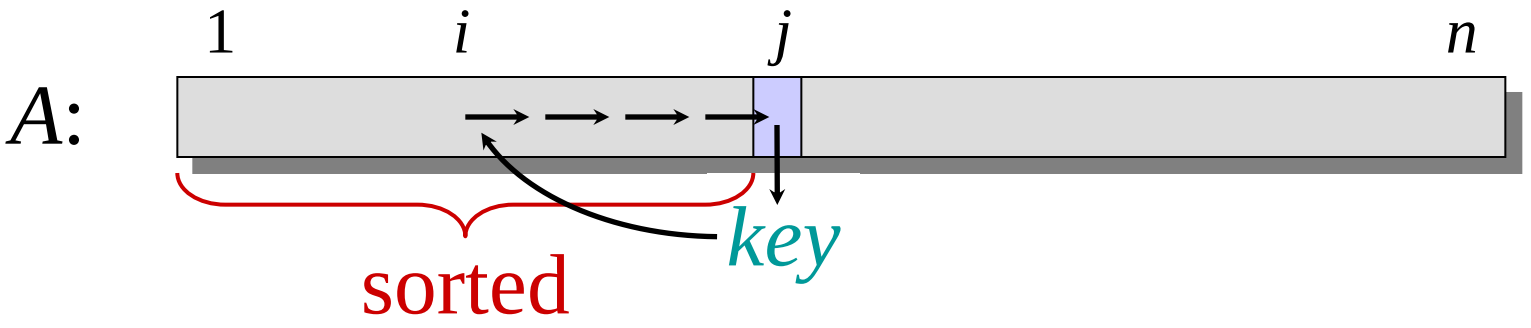
INSERTION SORT



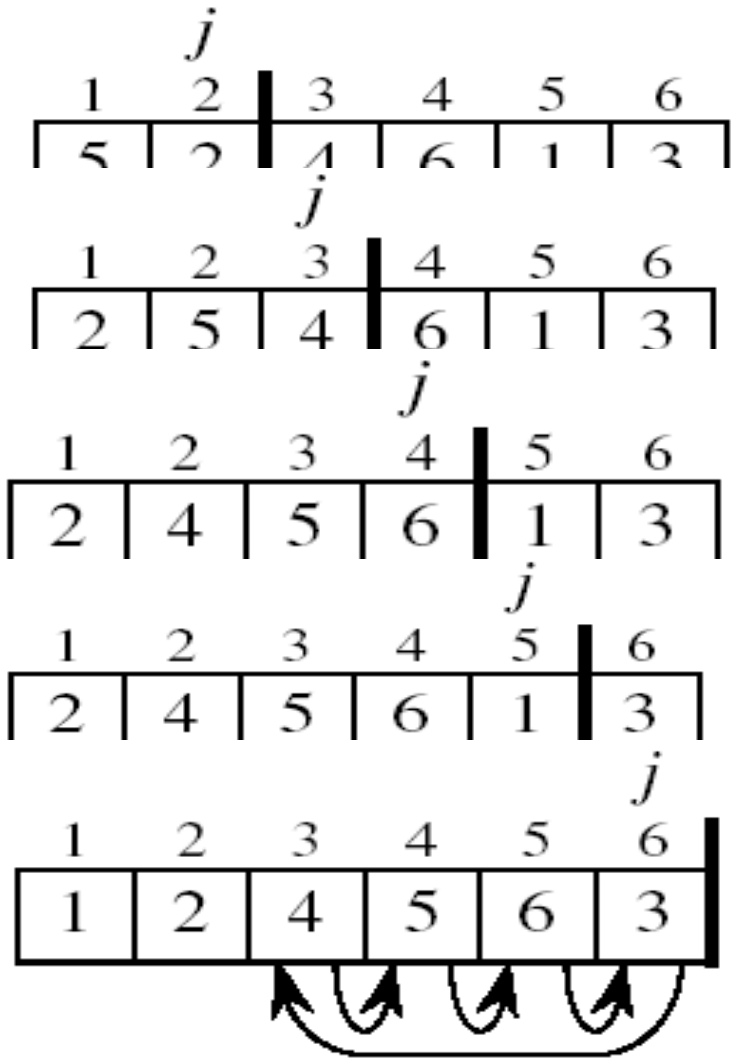
Insertion sort

"pseudocode"

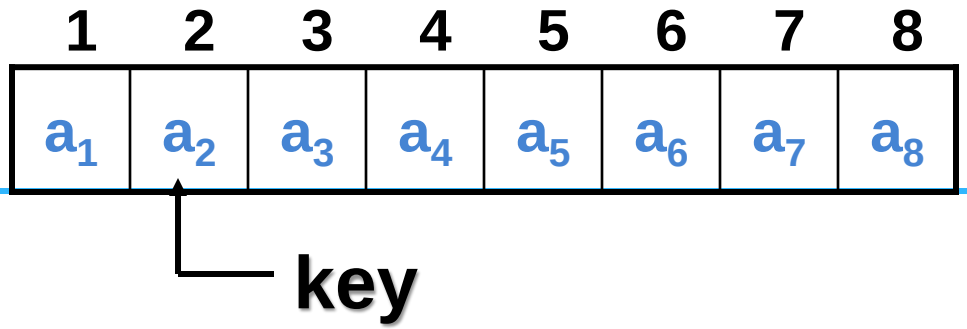
```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



INSERTION SORT



INSERTION-SORT



INSERTION-SORT(*A*)

```
for j ← 2 to n
        key ← A[ j ]
        //Insert A[ j ] into the sorted sequence A[1 . . j-1]
        i ← j - 1
        while i > 0 and A[i] > key
                A[i + 1] ← A[i]
                i ← i - 1
        A[i + 1] ← key
```

ANALYSIS OF INSERTION SORT

INSERTION-SORT*(A)*

```
for j ← 2 to n do
        key ← A[ j ]
        i ← j - 1
        while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i – 1
        A[i + 1] ← key
```

Assume t_j is # of times the while statement is executed at iteration j

BEST CASE ANALYSIS

```
while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i – 1
```

The array is already sorted

- A[i] ≤ key upon the first time the **while** loop test is run
- $t_j = 1$

Then: $$T(n) = \sum_{j=2}^{n} 1$$

Best case of T(n) ∈ Θ(n)

WORST CASE ANALYSIS

```
while i > 0 and A[i] > key
        A[i + 1] ← A[i]
        i ← i – 1
```

The array is reversed

- Always A[i] > key
- Have to compare key with all elements to the left of the j-th position ⇒ compare with j-1 elements
- **⇒ $t_j = j$**

Then: $$T(n) = \sum_{j=2}^{n} j$$

WORST CASE ANALYSIS

Using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

T (n) = $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

Best case of T(n) $\in \Theta(n^2)$

INSERTION SORT - SUMMARY

Advantages

- Good running time for "almost sorted" arrays $\Theta(n)$

Disadvantages

- $\Theta(n^2)$ running time in worst and average case

APPENDIX: USEFUL MATHEMATICAL FORMULAS FOR THE ANALYSIS OF ALGORITHMS

APPENDIX: IMPORTANT SUMMATION FORMULAS

1. $\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$

(l,u are integer limits, l≤ u); $\sum_{i=1}^{n} 1 = n$

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

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

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1} (a \neq 1) = \sum_{i=0}^{n} 2^j = 2^{n+1}\text{-}1$

APPENDIX: IMPORTANT SUMMATION FORMULAS

6. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \ \ldots + \frac{1}{n} \approx \ln n + \ \gamma$

$$, where\ \gamma \ \approx 0.5772 (Euler's\ Constant)$$

7. $\sum_{i=1}^{n} i2^i = 1 * 2 + 2 * 2^2 + \ \ldots + n2^n = (n-1)2^{n+1} + 2$

8. $\sum_{i=1}^{n} ia^i = 1 * 2 + 2 * 2^2 + \ \ldots + \ a\frac{d}{da}\left(\frac{a^{n+1}-1}{a-1}\right)$

9. $\sum_{i=1}^{n} \lg i \ \approx n \lg n$

10. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$

11. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

APPENDIX: PROPERTIES OF LOGARITHMS

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_b(m^n) = n \cdot \log_b(m)$
4. $\log_b(mn) = \log_b(m) + \log_b(n)$
5. $\log_b(m/n) = \log_b(m) - \log_b(n)$
6. $a^{\log_b m} = m^{\log_b a}$
7. $\log_a m = \frac{\log_b m}{\log_b a} = \log_a b \log_b m$