# Introduction to Algorithm

## Definition:

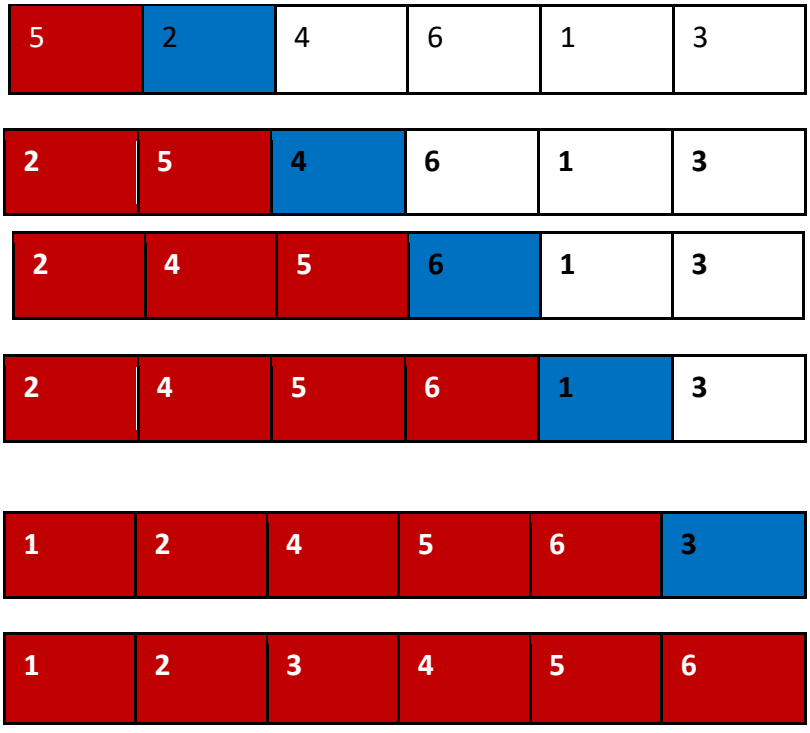
- ✓ Algorithm is any well-defined computational procedure.
- ✓ It takes some value, or set of values as input and produces some values or set of values as output.
- ✓ It is a tool for solving computation problem.
- ✓ Example: Sorting algorithm for sorting problem.

## Insertion Sort:

- ✓ Insertion sort is a sorting algorithm
- ✓ Input: a sequence of n numbers- $a_1, a_2, a_3 \ldots, a_n$
- ✓ Output: Reordering of input sequences ( $a_1', a_2', a_3' \ldots, a_n'$ ) such that $a_1' \le a_2' \le a_3' \ldots \le a_n'$

## INSERTION-SORT(A)

1. for j = 2 to A.length
2. key = A[j]
3. //Inset A[j] into the sorted sequence A[1 … j-1]
4. i = j - 1
5. while i > 0 and A[i] > key
6. A[i+1] = A[i]
7. i = i - 1
8. A[i+1] = key



## Insertion Sort (Running Time)

| INSERTION-SORT(A) | Cost | Time |
|---|---|---|
| 1. for j = 2 to A.length | $C_1$ | n |
| 2. key = A[j] | $C_2$ | n-1 |
| 3. //Inset A[j] into the sorted sequence A[1 … j-1] | $C_3$ | n-1 |
| 4. i = j – 1 | 0 | n-1 |
| 5. while i > 0 and A[i] > key | $C_5$ | $\sum_{j=2}^{n} t_j$ |
| 6. A[i+1] = A[i] | $C_6$ | $\sum_{j=2}^{n} t_j - 1$ |
| 7. i = i – 1 | $C_7$ | $\sum_{j=2}^{n} t_j - 1$ |
| 8. A[i+1] = key | $C_8$ | n-1 |

## Running Time & Space Complexity:

- ✓ Computer is fast but it is not infinitely fast.
- ✓ If computer were infinitely fast any correct algorithm would do.
- ✓ As computer is not infinitely fast, so algorithm should be designed to finish within expected time.
- ✓ Memory of computer is not unlimited and not free rather computer memory is fast, limited and costly.
- ✓ Algorithm should be designed that should use expected extra memory.

## Kinds of Runtime Analysis

- ✓ **Worst-case: (usually)**
  - $T(n)$ = maximum time of algorithm on any input of size $n$.
- ✓ **Average-case: (sometimes)**
  - $T(n)$ = expected time of algorithm over all inputs of size $n$.
  - Need assumption of statistical distribution of inputs.
- ✓ **Best-case:**
  - Cheat with a slow algorithm that works fast on *some* input.

## Asymptotic Notation - O-notation

ORDER: $1 < logn < \sqrt{n} < n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

DEF: $O(g(n)) = \{ f(n) :$ there exist positive constants $c$ and $n_0$ such that $0 \le f(n) \le c\ g(n)$ for all $n \ge n_0 \}$

## Basic manipulations:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 10n^2 – 5n + 1 = o(n^3)$

## Asymptotic Notation - Ω-notation

ORDER: $1 < logn < \sqrt{n} < n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

DEF: $\Omega(g(n)) = \{ f(n) :$ there exist positive constants $c$ and $n_0$ such that $0 \le c\ g(n) \le f(n)$ for all $n \ge n_0 \}$

### Basic manipulations:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 10n^2 - 5n + 1 = \Omega(n^3)$

## Asymptotic Notation - Θ-notation

ORDER: $1 < logn < \sqrt{n} < n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

DEF: $\Theta(g(n)) = \{ f(n) :$ there exist positive constants $c_1, c_2$ and $n_0$ such that $0 \le c_1\ g(n) \le f(n) \le c_2\ g(n)$ for all $n \ge n_0 \}$

### Basic manipulations:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 10n^2 - 5n + 1 = \Theta(n^3)$

## Asymptotic Notation(Example)

- ✓ $2n + 3$
- ✓ $5n^2 + 7n + 2$
- ✓ $n^2 \log n + n$
- ✓ $n!$
- ✓ $\log n!$

### Analysis of Insertion Sort:

| INSERTION-SORT(A) | Cost | Time |
|---|---|---|
| for j = 2 to A.length | $C_1$ | n |
| key = A[j] | $C_2$ | n-1 |
| //Inset A[j] into the sorted sequence A[1 … j-1] | $C_3$ | n-1 |
| i = j – 1 | 0 | n-1 |
| while i > 0 and A[i] > key | $C_5$ | $\sum_{j=2}^{n} t_j$ |
| A[i+1] = A[i] | $C_6$ | $\sum_{j=2}^{n} t_j - 1$ |
| i = i – 1 | $C_7$ | $\sum_{j=2}^{n} t_j - 1$ |
| A[i+1] = key | $C_8$ | n-1 |

$$\mathbf{T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)}$$

## Best Case Analysis:

- The array is already sorted
  - A[i] ≤ key upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$
  $= an + b = \Theta(n)$

$$\mathbf{T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)}$$

## Worst Case Analysis

- The array is in reverse sorted order
  - Always A[i] > key in while loop test
    Have to compare key with all elements to the left of the j-th position
    $\Rightarrow$ compare with j-1 elements $\Rightarrow t_j = j$

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ quadratic function of n

$T(n) = \Theta(n^2)$ order of growth in $n^2$

$$\boldsymbol{T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)}$$

## Insertion Sort – Summary

- Advantages
  - Good running time for "almost sorted" arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges

## Divide and Conquer:

- ✓ Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data ***S*** in two or more disjoint subsets $S_1$, $S_2$, …
  - Recur: solve the sub problems recursively
  - Conquer: combine the solutions for $S_1$, $S_2$, …, into a solution for ***S***
- ✓ The base case for the recursion are sub problems of constant size
- ✓ Analysis can be done using **recurrence equations**

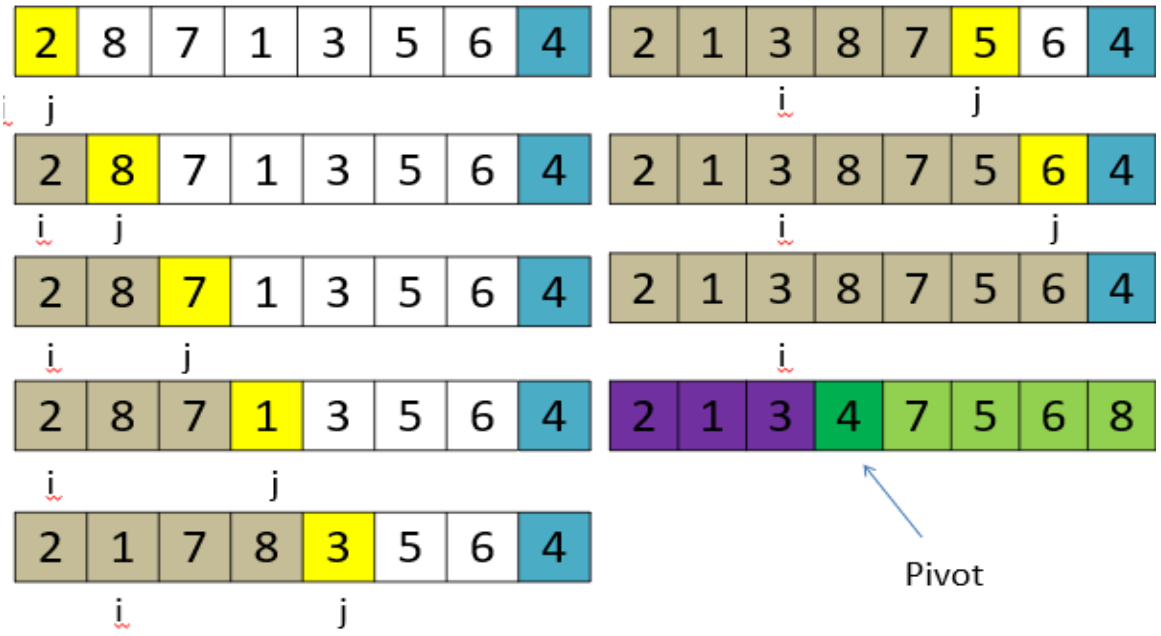
# Quick Sort

**QUICKSORT(A, p, r)**

1. if p < r
2. q = PARTITION(A, p, r)
3. QUICKSORT(A, p, q – 1)
4. QUICKSORT(A, q + 1, r)

**PARTITION(A, p, r)**

1. x = A[r]
2. i = p – 1
3. for j = p to r - 1
4. if A[j] <= x
5. i = i+1
6. exchange A[i] with A[j]
7. exchange A[I + 1] with A[r]
8. return i+1;



# Divide and Conquer(Merge Sort)

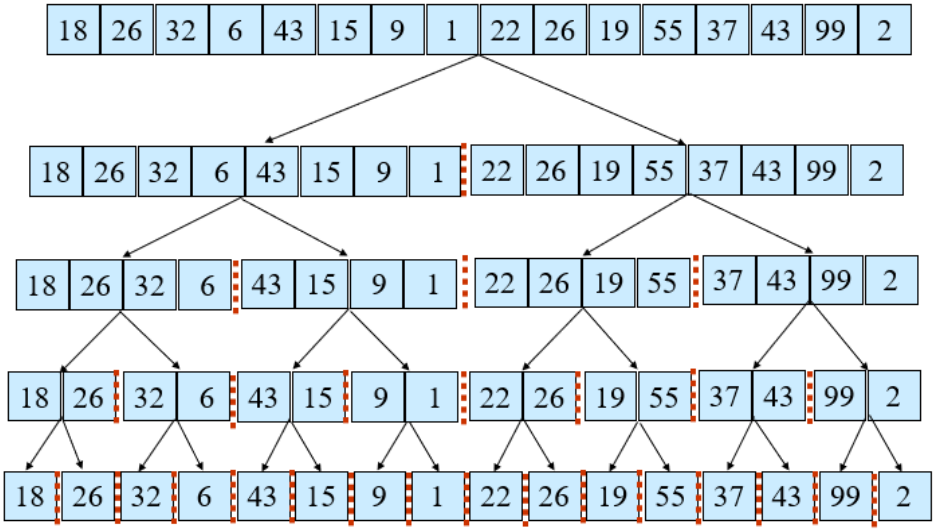
## Divide and Conquer

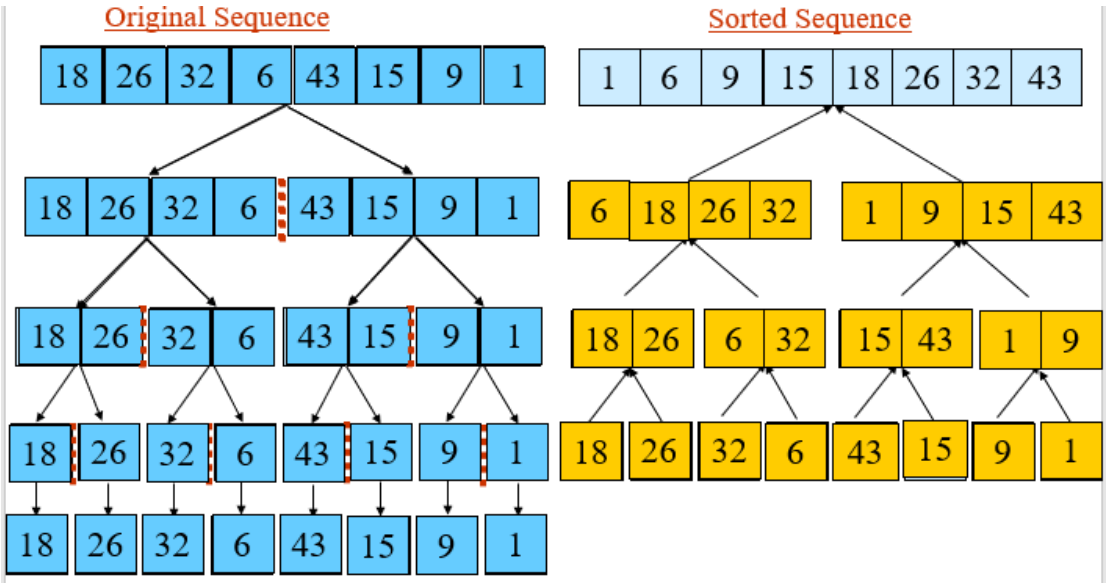
- **Recursive in structure**
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

## An Example:

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine*:** Merge the two sorted subsequences to produce the sorted answer.



## Merge-Sort (A, p, r)

**INPUT:** a sequence of $n$ numbers stored in array A
**OUTPUT:** an ordered sequence of $n$ numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1   if p < r
2      then q ← ⌊(p+r)/2⌋
3           MergeSort (A, p, q)
4           MergeSort (A, q+1, r)
5           Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort($A$, 1, $n$)

## Procedure Merge:

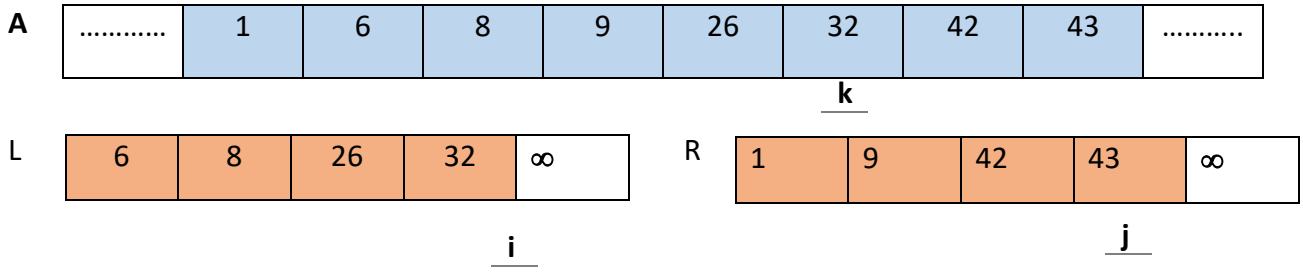
```
Merge(A, p, q, r)
1   n1 ← q – p + 1
2   n2 ← r – q
3   for i ← 1 to n1
4       do L[i] ← A[p + i – 1]
5   for j ← 1 to n2
6       do R[j] ← A[q + j]
7   L[n1+1] ← ∞
8   R[n2+1] ← ∞

9   i ← 1
10  j ← 1
11  for k ←p to r
12      do if L[i] ≤ R[j]
13           then A[k] ← L[i]
14                i ← i + 1
15           else A[k] ← R[j]
16                j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.
Output: Merged sorted subarray in $A[p..r]$.

**Sentinels**, to avoid having to check if either subarray is fully copied at each step.

## Merge – Example:



## Analysis of Merge Sort

- Running time ***T*(*n*)** of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging $n$ elements takes $\Theta(n)$

Total :

$T(n) = \Theta(1)$ if $n = 1$

$T(n) = 2T(n/2) + \Theta(n)$ if $n > 1$ $\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS, Chapter 4)

# Methods for Solving Recurrences

- Iteration method
- Substitution method
- Recursion tree method
- Master method

# The Iteration Method:

✓ Convert the recurrence into a summation and try to bound it using known series
  - Iterate the recurrence until the initial condition is reached.
  - Use back-substitution to express the recurrence in terms of *n* and the initial (boundary) condition.

## T(n) = c + T(n/2)

$T(n) = c + T(n/2)$
$= c + c + T(n/4)$
$= c + c + c + T(n/8)$

$T(n/2) = c + T(n/4)$
$T(n/4) = c + T(n/8)$

Assume $n = 2^k$

$T(n) = \underbrace{c + c + \ldots + c}_{K \text{ times}} + T(1)$
$= c\lg n + T(1)$
$= \Theta(\lg n)$

## Iteration Method – Example:

**T(n) = n + 2T(n/2)** **Assume: $n = 2^k$**

$T(n) = n + 2T(n/2)$
$= n + 2(n/2 + 2T(n/4))$
$= n + n + 4T(n/4)$
$= n + n + 4(n/4 + 2T(n/8))$
$= n + n + n + 8T(n/8)$
$\ldots = in + 2^iT(n/2^i)$
$= kn + 2^kT(1)$
$= n\lg n + nT(1) = \Theta(n\lg n)$

$T(n/2) = n/2 + 2T(n/4)$

# The substitution method:

✓ Guess a solution
✓ Use induction to prove that the solution works

- **Guess a solution**
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation

  $T(n) \le d\,g(n)$, for some $d > 0$ and $n \ge n_0$
  - Induction hypothesis: $T(k) \le d\,g(k)$ for all $k < n$
- **Prove the induction goal**
  - Use the induction hypothesis to find some values of the constants d and $n_0$ for which the induction goal holds

## Example: Binary Search

$$T(n) = c + T(n/2)$$

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le d \lg(n/2)$
- Proof of induction goal:

$$T(n) = T(n/2) + c \le d \lg(n/2) + c$$
$$= d \lg n - d + c \le d \lg n$$
$$\text{if: } -d + c \le 0,\ d \ge c$$

- Base case?

## Example 2

$$T(n) = T(n-1) + n$$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le c\,n^2$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$$T(n) = T(n-1) + n \le c\,(n-1)^2 + n$$
$$= cn^2 - (2cn - c - n) \le cn^2$$
$$\text{if: } 2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$$

  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

## Example 3

$$T(n) = 2T(n/2) + n$$

- Guess: $T(n) = O(n \lg n)$
  - Induction goal: $T(n) \le cn \lg n$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le cn/2 \lg(n/2)$
- Proof of induction goal:

$$T(n) = 2T(n/2) + n \le 2c\,(n/2)\lg(n/2) + n$$
$$= cn \lg n - cn + n \le cn \lg n$$
$$\text{if: } -cn + n \le 0 \Rightarrow c \ge 1$$

- Base case?

# The recursion-tree method

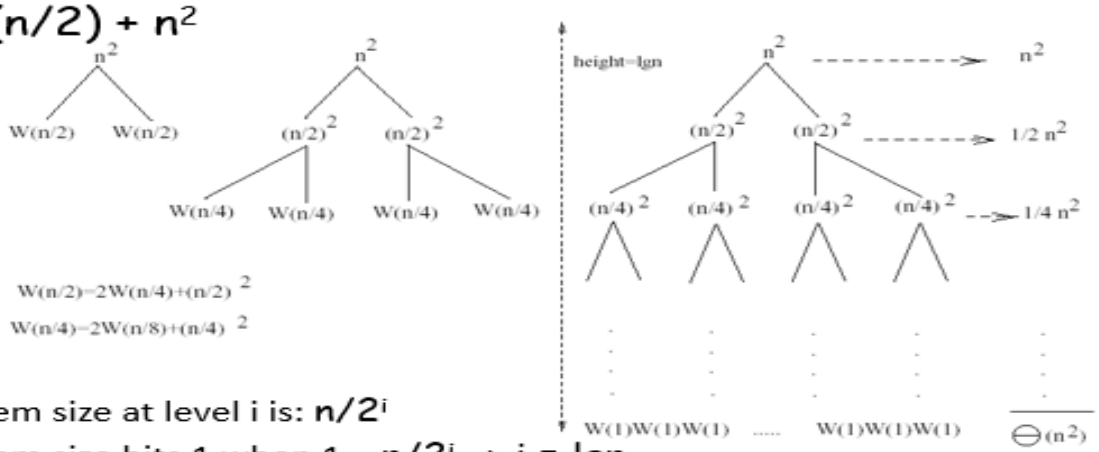
Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

– Used to "guess" a solution for the recurrence

## Example 1

$W(n) = 2W(n/2) + n^2$




- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level i = $2^i$
- Total cost:

$$W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \le n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-1/2} + O(n) = 2n^2$$

$\Rightarrow W(n) = O(n^2)$

12

## Example 2 - Substitution

$T(n) = 3T(n/4) + cn^2$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le dn^2$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/4) \le d\,(n/4)^2$
- Proof of induction goal:

$T(n) = 3T(n/4) + cn^2$

$\le 3d\,(n/4)^2 + cn^2$

$= (3/16)\,d\,n^2 + cn^2$

$\le d\,n^2$ if: $d \ge (16/13)c$

- Therefore: $T(n) = O(n^2)$

## Example 3 - Substitution

$W(n) = W(n/3) + W(2n/3) + O(n)$

- Guess: $W(n) = O(n \lg n)$
  - Induction goal: $W(n) \le dn\lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $W(k) \le d\,k\lg k$ for any K $< n$ $(n/3, 2n/3)$
- Proof of induction goal:

Try it out as an exercise!!

- $T(n) = O(n \lg n)$

1

# Master's method:

- "Cookbook" for solving recurrences of the form:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where, a ≥ 1, b > 1, and $f(n) = \Theta(n^k \log^p n)$

case 1: if $\log_b^a > k$ then T(n) = $\Theta(n^{\log_b^a})$

case 2: if $k = \log_b^a$ then

(i) if p > -1 then T(n) = $\Theta(n^k \log^{p+1} n)$

(ii) if p = -1 then T(n) = $\Theta(n^k \log\log n)$

(iii) if p < -1 then T(n) = $\Theta(n^k)$

case 3: if $k < \log_b^a$ then

i) if p >= 0 then $\Theta(n^k \log^p n)$ ii) if p < 0 then $\Theta(n^k)$

## Why $n^{\log_b a}$?

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = \underbrace{aT\left(\frac{n}{b}\right)}_{\underbrace{a^2T\left(\frac{n}{b^2}\right)}_{a^3T\left(\frac{n}{b^3}\right)}}$$

$$\vdots$$

$$T(n) = a^iT\left(\frac{n}{b^i}\right) \quad \forall i$$

- Assume $n = b^k \Rightarrow k = \log_b n$
- At the end of iteration i = k:

$$T(n) = a^{\log_b n}T\left(\frac{b^i}{b^i}\right) = a^{\log_b n}T(1) = \Theta\left(a^{\log_b n}\right) = \Theta\left(n^{\log_b a}\right)$$

- Case 1:
  - If f(n) is dominated by $n^{\log_b a}$:
    - $T(n) = \Theta(n^{\log_b n})$
- Case 3:
  - If f(n) dominates $n^{\log_b a}$:
    - $T(n) = \Theta(f(n))$
- Case 2:
  - If $f(n) = \Theta(n^{\log_b a})$:
    - $T(n) = \Theta(n^{\log_b a} \log n)$

# Example: T(n) = 2T(n/2) + n

a = 2, b = 2, $\log_2 2 = 1$

Compare $n^{\log_2 2}$ with f(n) = n

$\Rightarrow$ $f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow$ $T(n) = \Theta(n \lg n)$

$T(n) = 2T(n/2) + n^2$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow$ f(n) Case 3 $\Rightarrow$ verify regularity cond.

$a\ f(n/b) \leq c\ f(n)$

$\Leftrightarrow 2\ n^2/4 \leq c\ n^2 \Rightarrow c = ½$ is a solution ($c<1$)

$\Rightarrow T(n) = \Theta(n^2)$

$T(n) = 2T(n/2) + \sqrt{n}$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^{1/2}$

$\Rightarrow f(n) = O(n^{1-\varepsilon})$ Case 1

$\Rightarrow T(n) = \Theta(n)$

$T(n) = 3T(n/4) + n\lg n$

$a = 3, b = 4, \log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

f(n) = Case 3

Check regularity condition:

$3*(n/4)\lg(n/4) \leq (3/4)n\lg n = c*f(n), c=3/4$

$\Rightarrow T(n) = \Theta(n\lg n)$

$T(n) = 2T(n/2) + n\lg n$

$a = 2, b = 2, \log_2 2 = 1$

- Compare n with $f(n) = n\lg n$
  - seems like case 3 should apply

# Heap Sort

**Goals:** To explore the implementation, testing and performance of heap sort algorithm

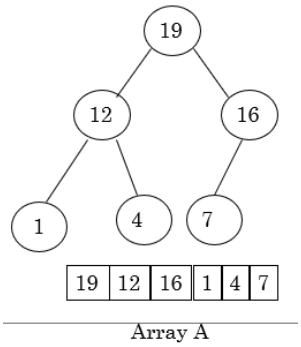
## HEAP:

A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

- Complete Binary tree
- Heap Order

It is implemented as an array where each node in the tree corresponds to an element of the array.

- The binary heap data structures is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.




- The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed

PARENT (*i*)
return floor(*i*/2)
LEFT (*i*)
return 2*i*
RIGHT (*i*)
return 2*i* + 1

- For every node *v*, other than the root, the key stored in *v* is greater or equal (smaller or equal for max heap) than the key stored in the parent of *v*.
- In this case the maximum value is stored in the root

## Definition:

Max Heap:

Store data in ascending order

Has property of

A[Parent(i)] ≥ A[i]

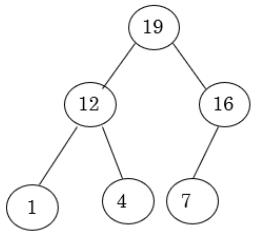
Min Heap:

Store data in descending order

Has property of

A[Parent(i)] ≤ A[i]

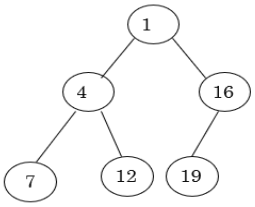
MAX HEAP EXAMPLE




| 19 | 12 | 16 | 1 | 4 | 7 |
|---|---|---|---|---|---|

Array A

MIN HEAP EXAMPLE




| 1 | 4 | 16 | 7 | 12 | 19 |
|---|---|---|---|---|---|

Array A

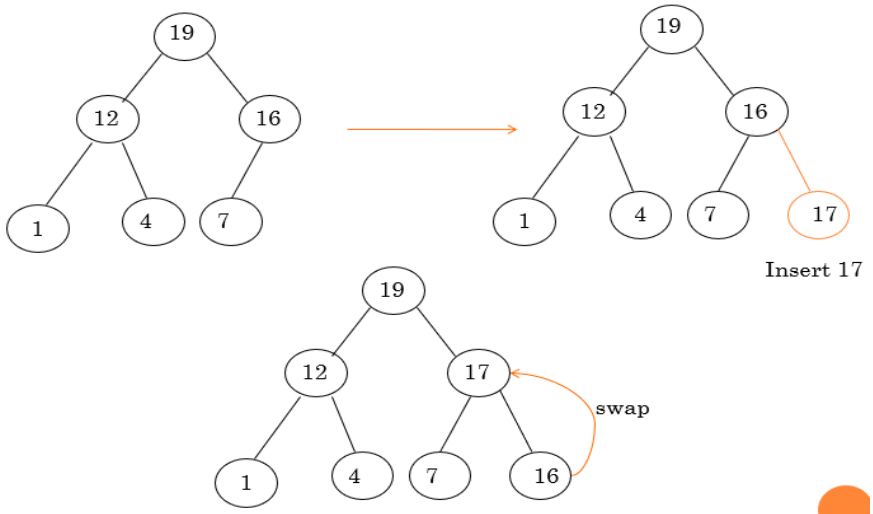
# Insertion:

- Algorithm
  1. Add the new element to the next available position at the lowest level
  2. Restore the max-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

OR

Restore the min-heap property if violated

- General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



Deletion:

- Delete max
  - Copy the last number to the root (overwrite the maximum element stored there).
  - Restore the max heap property by percolate down.
  - Delete min
  - Copy the last number to the root (overwrite the minimum element stored there).

Restore the min heap property by percolate down

## Heap Sort:

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap

## Procedures on Heap:

- Heapify
- Build Heap
- Heap Sort

Heapify:

- Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
l ← left(i)
r ← right(i)
if l <= heapsize[A] and A[l] > A[i]
   then largest ←l
   else largest ← i
if r <= heapsize[A] and A[r] > A[largest]
   then largest ← r
if largest != i
   then swap A[i] ←→ A[largest]
      Heapify(A, largest)
}
```

**BUILD HEAP:**

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify'

on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A)
{
heapsize[A] ←length[A]
for i ←|length[A]/2  //down to 1
   do Heapify(A, i)
}
```

Heap Sort Algorithm:

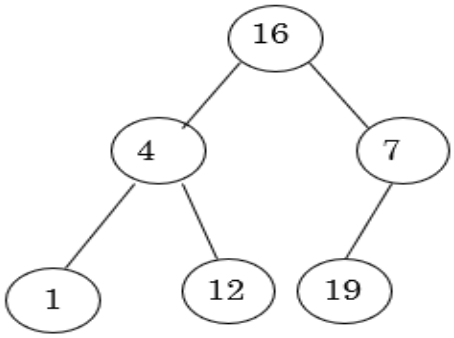
- The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

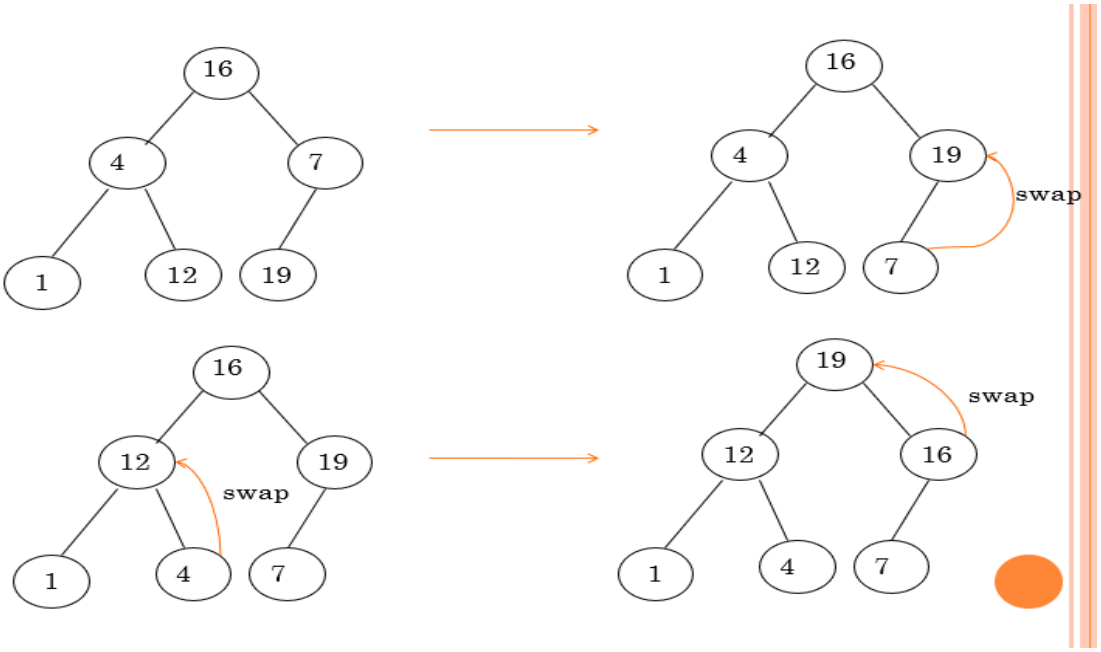
```
Heapsort(A)
{
Buildheap(A)
for i ← length[A] //down to 2
   do swap A[1] ←→ A[i]
   heapsize[A] ← heapsize[A] - 1
   Heapify(A, 1)
}
```

**Example:** Convert the following array to a heap

| 16 | 4 | 7 | 1 | 12 | 19 |
|---|---|---|---|---|---|

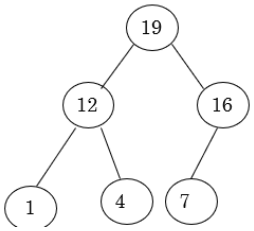
Picture **the array as a complete binary tree:**

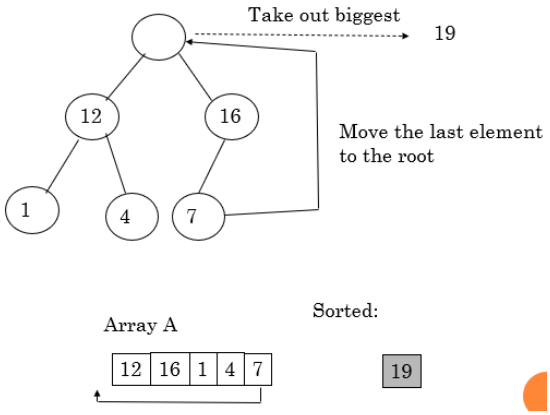


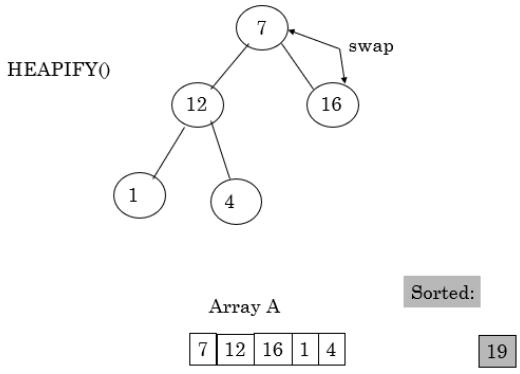


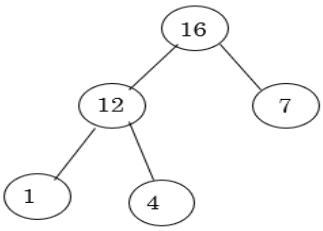

- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

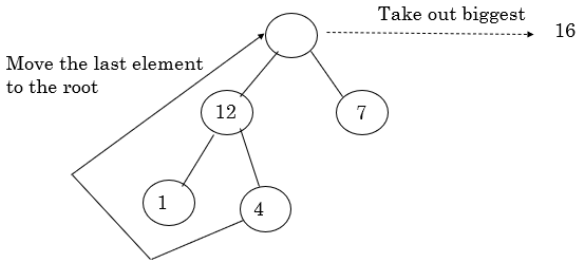
- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap






Array A

| 16 | 12 | 7 | 1 | 4 |
|---|---|---|---|---|

Sorted:

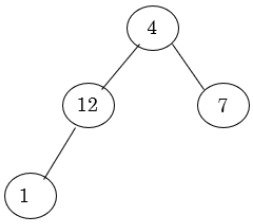
| 19 |
|---|

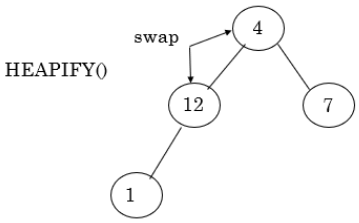
Array A

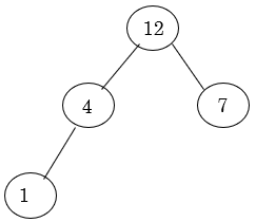
| 12 | 7 | 1 | 4 |
|---|---|---|---|

Sorted:

| 16 | 19 |
|---|---|




Array A

| 4 | 12 | 7 | 1 |
|---|---|---|---|

Sorted:

| 16 | 19 |
|---|---|

Array A

| 4 | 12 | 7 | 1 |
|---|---|---|---|

Sorted:

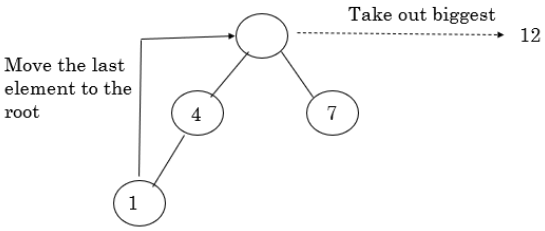
| 16 | 19 |
|---|---|

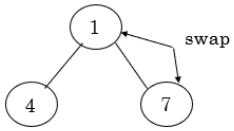
Array A

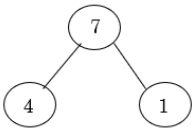
| 12 | 4 | 7 | 1 |
|---|---|---|---|

Sorted:

| 16 | 19 |
|---|---|




Array A

| 4 | 7 | 1 |
|---|---|---|

Sorted:

| 12 | 16 | 19 |
|---|---|---|

Array A

| 1 | 4 | 7 |
|---|---|---|

Sorted:

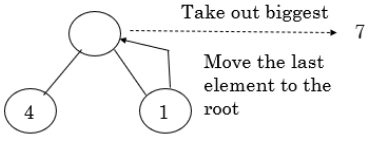
| 12 | 16 | 19 |
|---|---|---|

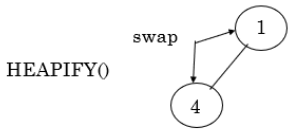
Array A

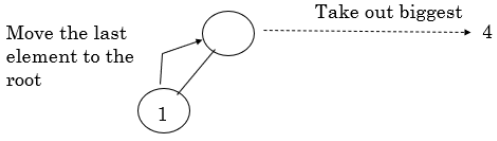
| 7 | 4 | 1 |
|---|---|---|

Sorted:

| 12 | 16 | 19 |
|---|---|---|




Array A

| 1 | 4 |
|---|---|

Sorted:

| 7 | 12 | 16 | 19 |
|---|---|---|---|

Array A

| 4 | 1 |
|---|---|

Sorted:

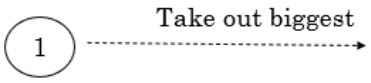
| 7 | 12 | 16 | 19 |
|---|---|---|---|

Array A

| 1 |
|---|

Sorted:

| 4 | 7 | 12 | 16 | 19 |
|---|---|---|---|---|




Sorted:

| 1 | 4 | 7 | 12 | 16 | 19 |
|---|---|---|---|---|---|

Array A

Sorted:

| 1 | 4 | 7 | 12 | 16 | 19 |
|---|---|---|---|---|---|

## TIME ANALYSIS

- Build Heap Algorithm will run in O(n) time
- There are $n$-1 calls to Heapify each call requires O(log $n$) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)