Brute Force Technique

- **Brute force** is a straightforward approach to solve a problem, usually directly based on the problem statement and definitions of the concepts involved.
- Brute force is applicable to a very wide variety of problems.
- For some important problems (e.g., sorting, searching, matrix multiplication, string matching) the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
- Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
- A brute-force algorithm can serve an important theoretical or educational purpose.

Selection Sort

- Scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.

- Then, scan the list, starting with the second element, to find the smallest among the last n-1 elements and exchange it with the second element, putting the second smallest element in its final position.

- Generally, on the i^{th} pass through the list, the algorithm searches for the smallest item among the last n-i elements and swaps it with A_i

- After **n-1 passes**, the list is sorted.

Selection Sort Algorithm

ALGORITHM ***SelectionSort*****(A[0...n-1])**

```
//Sorts a given array by selection sort
//Input: An array A[0…n-1] of orderable elements
//Output: Array A[0…n-1] sorted in ascending order
for i ← 0 to n-2 do
        min ← i
        for j ← i + 1 to n-1 do
                if A[j] < A[min]   min ← j
        swap A[i] and A[min]
```

Analysis of Selection Sort Algorithm

- **Input Size metric:**
  Number of elements n.

- **Basic operation:**
  Key comparison

- The number of times the basic operation is executed **depends only on the array size**.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

Example: Sort {89,45,68,90,29,34,17} in ascending order using Selection sort.

Input: | 89 45 68 90 29 34 17
17 | 45 68 90 29 34 89
17 29 | 68 90 45 34 89
17 29 34 | 90 45 68 89
17 29 34 45 | 90 68 89
17 29 34 45 68 | 90 89
Output: 17 29 34 45 68 89 | 90

Bubble Sort

- Compare adjacent elements of the list and exchange them if they are out of order. This is done repeatedly till the end of the list.
- After first pass, the largest element will be placed in the last position in the list. The next pass bubbles up the second largest element, and so on.
- After n-1 passes, the list is sorted.
- Pass i ($0 \le i \le n-2$) of the bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \;?\leftrightarrow\; A_{j+1}, \ldots, A_{n-i-1} \;|\; A_{n-i} \le \ldots \le A_{n-1}$$

in their final positions

Bubble Sort Algorithm

ALGORITHM ***BubbleSort*****(A[0...n-1])**

```
//Sorts a given array by bubble sort
//Input: An array A[0...n-1] of orderable elements
//Output: Array A[0...n-1] sorted in ascending order
for i ← 0 to n-2 do
    for j ← 0 to n – 2 - i do
        if A[j + 1] < A[j]  swap A[i] and A[j + 1]
```

Analysis of Bubble Sort Algorithm

- **Input Size metric:**
  Number of elements n.
- **Basic operation:**
  Key comparison

- The number of times the basic operation is executed **depends only on the array size**.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} [(n - 2 - i) - 0 + 1] = \sum_{i=0}^{n-2} (n - 1 - i)$$

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

Example: Sort {89, 68, 45, 20} in ascending order using Bubble sort.

89 ↔ 68 45 20
68 89 ↔ 45 20
68 45 89 ↔ 20
68 45 20 89
PASS 0

68 ↔ 45 20 89
45 68 ↔ 20 89
45 20 68 89
PASS 1

45 ↔ 20 68 89
PASS 2

20 45 68 89

Sequential Search

ALGORITHM SequentialSearch(A[0...n], k)

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0...n-1] whose value is
//            equal to K or -1 if no such element is found
A[n] ← K
i ← 0
while i<n and A[i] ≠ K    do
      i ← i + 1
if i < n return i
else return -1
```

Brute-Force String Matching

- Given a string of n characters called the text and a string of m characters (m ≤ n) called the pattern, find a substring of the text that matches the pattern.
- Find i, the index of the leftmost character of the first matching substring in the text such that

  $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

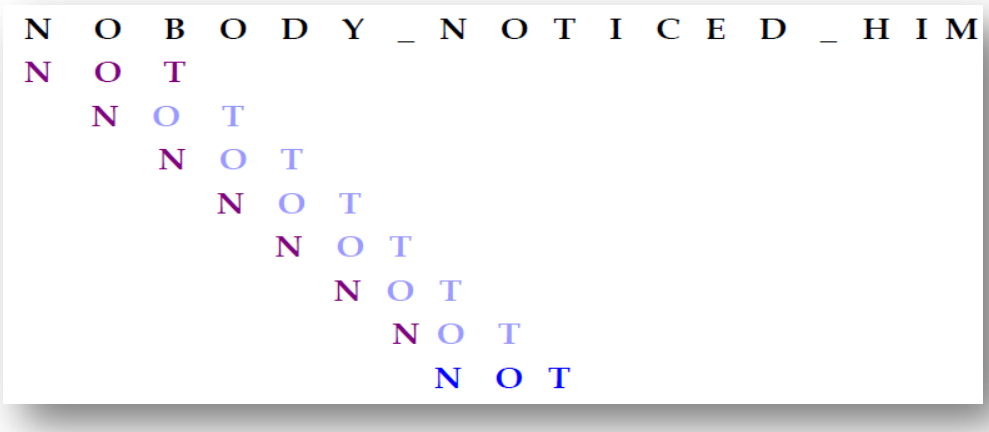
- Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.
- If a mismatch pair is encountered, then shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text.

- Note that the last position in the text which can still be a beginning of a matching substring is **n – m** (provided text's positions are indexed from 0 to n – 1).

ALGORITHM *BruteForceStringMatch*(T[0...n-1], P[0...m-1])

```
//Implements brute-force string matching
//Input: An array T[0...n-1] of n characters representing
// a text and an array P[0...m-1] of m characters
//representing a pattern
//Output: The index of the first character in the text that
//starts a matching substring or -1 if the search is
//unsuccessful
for i ← 0 to n – m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return -1
```

Example: Apply Brute-Force String Matching algorithm to search for the pattern "NOT" in the text "NOBODY_NOTICED_HIM". Also, find the number of character comparisons.



- **Number of character comparisons = 12**

Analysis of Brute-Force String Matching Algorithm

- **Input Size metric:**

  Number of characters in the text i.e., n.

- **Basic operation:**

  Key comparison

- The number of times the basic operation is executed **depends not only on the array size but also on pattern of input**.

$$C_{worst}(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} [(m - 1) - 0 + 1] = \sum_{i=0}^{n-m} m.$$

$$= m \sum_{i=0}^{n-m} 1 = m\,((n - m) - 0 + 1) = mn - m^2 + m$$

$$\approx mn \in \Theta(mn)$$

$$C_{best}(n) = \Theta(m)$$

Divide-and-Conquer Technique

Divide – and - conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

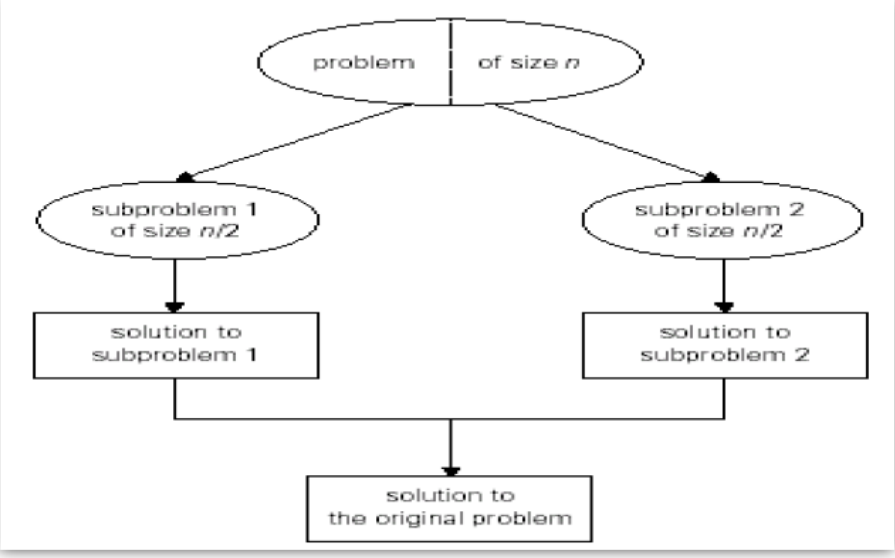


Figure: *Divide – and – Conquer Technique (typical case).*

- Not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution.
- Divide-and-Conquer technique is **ideally suited for *parallel computations***, in which each subproblem can be solved simultaneously by its own processor. Later all solutions can be merged to get solution to original problem. Thus, the ***execution speed of a program*** which is based on this technique can be ***improved significantly***.

- In the most typical case of divide-and-conquer, a problem's instance of size n is divided into ***two instances of size n/2***.
- Generally, an instance of **size *n*** can be divided into **b instances of size *n/b***, with ***a* of them needing to be solved**. (Here, a and b are constants; $a \geq 1$ and $b > 1$).
- Assuming that size n is a power of b, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n) \quad \ldots\ldots\ldots\ldots (1)$$

(General Divide-and-Conquer Recurrence)

where, $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Master Theorem:

If f(n) $\in \Theta(n^d)$ with $d \geq 0$ in recurrence Equation (1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Problems: Find the order of growth of the following recurrences using Master theorem:

1. $T(n) = 4T(n/2) + n$, $T(1) = 1$

Soln: Here, a=4, b=2, d=1.

since, $a > b^d$ i.e., $4 > 2^1$, $T(n) \in \Theta(n^{\log_b a})$

Therefore, $T(n) \in \Theta(n^{\log_2 4})$

$T(n) \in \mathbf{\Theta(n^2)}$

2. $T(n) = 4T(n/2) + n^2, T(1) = 1$

Soln: Here, a=4, b=2, d=2.

since, $a = b^d$ i.e., $4 = 2^2$, $T(n) \in \Theta(n^d \log n)$

Therefore, $T(n) \in \mathbf{\Theta(n^2 \log n)}$

3. $T(n) = 4T(n/2) + n^3, T(1) = 1$

Soln: Here, a=4, b=2, d=3.

since, $a < b^d$ i.e., $4 < 2^3$, $T(n) \in \Theta(n^d)$

Therefore, $T(n) \in \mathbf{\Theta(n^3)}$

Mergesort

- The strategy behind Merge Sort is to *change the problem of sorting into the problem of merging two sorted sub-arrays into one*.
- If the two halves of the array were sorted, then merging them carefully could complete the sort of the entire array.
- Mergesort sorts a given array A[0…n-1] by dividing it into two halves A[0… $\lfloor n/2 \rfloor$-1] and A[$\lfloor n/2 \rfloor$…n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- The merging of two sorted arrays can be done as follows:
  - Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
  - The elements pointed to are compared, and the smaller of them is added to a new array being constructed.
  - After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
  - This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Mergesort Algorithm

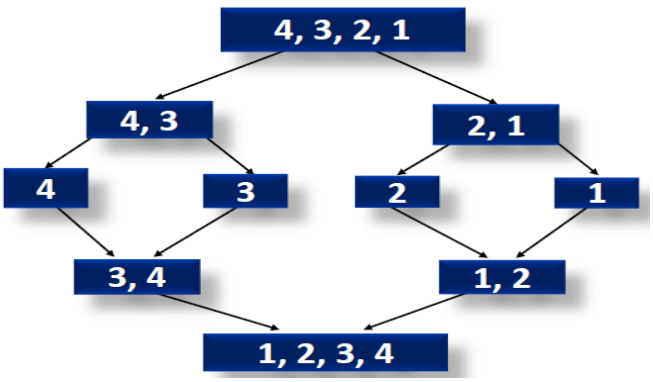
ALGORITHM Mergesort(A[0...*n*-1])

```
//Sorts array A[0...n-1] by recursive mergesort
//Input: An array A[0...n-1] of orderable elements
//Output: Array A[0...n-1] sorted in nondecreasing order
if n > 1
    copy A[0 . . . n/2 - 1] to B[0 . . . n/2 - 1]
    copy A[ n/2 . . . n - 1] to C[0 . . . n/2 – 1]
    Mergesort(B[0 . . . n/2 – 1])
    Mergesort(C[0 . . . n/2 – 1])
    Merge(B, C, A)
```

ALGORITHM Merge(B[0 . . . p-1], C[0 . . . q-1], A[0 . . . p + q - 1])

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 . . . p-1] and C[0 . . . q-1] both sorted
//Output: Sorted array A[0 . . . p + q - 1] of the elements of B & C
i ← 0, j ← 0, k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j . . . q – 1] to A[k . . . p + q – 1]
else copy B[i . . . p – 1] to A[k . . . p + q – 1]
```

Example1: Apply Mergersort to sort the list {4, 3, 2, 1} in ascending order.



Analysis of Mergesort Algorithm

- Assuming that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is

  $$\mathbf{C(n) = 2C(n/2) + C_{merge}(n) \quad for \quad n > 1, \quad C(1) = 0.}$$

  where, $\mathbf{C_{merge}(n)}$ is the number of key comparisons performed during the merging stage.

- For the worst case (Ex: smaller elements may come from the alternating arrays), $C_{merge}(n) = n - 1$, and we have the recurrence

  $$\mathbf{C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad for \; n > 1, \; C_{worst}(1) = 0.}$$

- Hence, according to the Master Theorem,

  $\mathbf{C_{worst}(n) \in \Theta(n \log n)}$ (Here, a=2, b=2, d=1)

  (Since, $a = b^d$, $C_{worst}(n) \in \Theta(n^d \log n)$)

Quicksort

- Unlike Mergesort, which divides its input's elements according to their position in the array, Quicksort divides them according to their value.
- It rearranges elements of a given array A[0…n-1] to achieve its ***partition***, a situation where all the elements before some position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s]:

  $$\underbrace{\mathbf{A[0] \ldots A[s-1]}}_{\text{all are} \le A[s]} \quad \mathbf{A[s]} \quad \underbrace{\mathbf{A[s+1] \ldots A[n-1]}}_{\text{all are} \ge A[s]}$$

- After a partition has been achieved, A[s] ***will be in its final position*** in the sorted array.
- We can continue sorting the two subarrays of the elements preceding and following A[s] independently by using same method.

Quicksort Algorithm

ALGORITHM *Quicksort(A[l..r])*

```
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n - 1], defined by its left and right indices l and r
//Output: The subarray A[l..r] sorted in nondecreasing
//order
if l < r
        s ← Partition(A[l...r])   //s is a split position
        Quicksort(A[l...s - 1])
        Quicksort(A[s + 1...r])
```

- A partition of subarray A[l…r] (0 ≤ l < r ≤ n-1) can be achieved by the following algorithm.

Pseudocode implementing the Partition Procedure:

ALGORITHM *Partition(A[l..r])*

```
//Partitions a subarray by using its first element as a pivot
//Input: subarray A[l..r] of A[0..n - 1], defined by its left
//and right indices l and r (l< r)
//Output: A partition of A[l..r], with the split position
//returned as  this function's value
        p ← A[l]
        i ← l; j ← r+1
        repeat
                repeat   i ← i + 1  until  A[i] ≥ p  or i ≥ r
                repeat   j ← j – 1   until  A[j] ≤ p
                swap(A[i], A[j ])
   until   i  ≥ j
    swap(A[i],A[j])       // undo last swap when i ≥ j
        swap(A[l], A[j ])
        return j
```

- First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding rule, we call this element the ***pivot***.
    - we use the simplest strategy of selecting the subarray's first element: p=A[*l*].

Procedure to achieve partition:

- The elements in the subarray are rearranged to achieve partition by using an efficient method based on **left-to-right scan** and **right-to-left scan**, each comparing the subarray's element with the pivot.

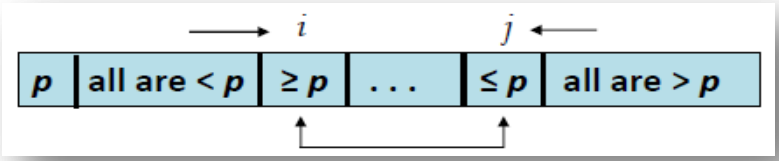
Left-to-Right Scan:

- The left-to-right scan, denoted by index *i,* starts with the second element.
- Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the **first element greater than or equal to the pivot**.

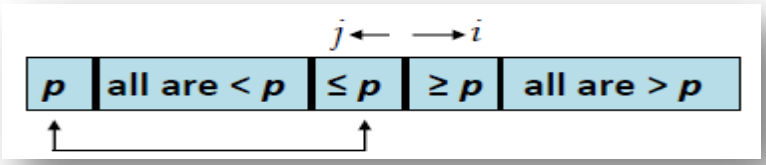
Right-to-Left Scan:

- The right-to-left scan, denoted by index *j,* starts with the last element of the subarray.
- Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the **first element smaller than or equal to the pivot**.

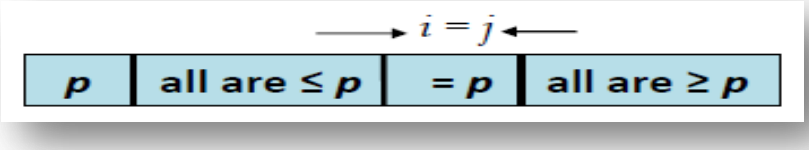
- After both scans stop, three situations may arise, depending on **whether or not the scanning indices have crossed**:
  1. If scanning indices *i* and *j* have not crossed, i.e., *i* < *j*, we simply exchange A[*i*] and A[*j*] and resume the scans by incrementing *i* and decrementing *j*, respectively:



2. If the scanning indices have crossed over, i.e., *i* > *j*, we will have partitioned the array after exchanging the pivot with A[*j*]:

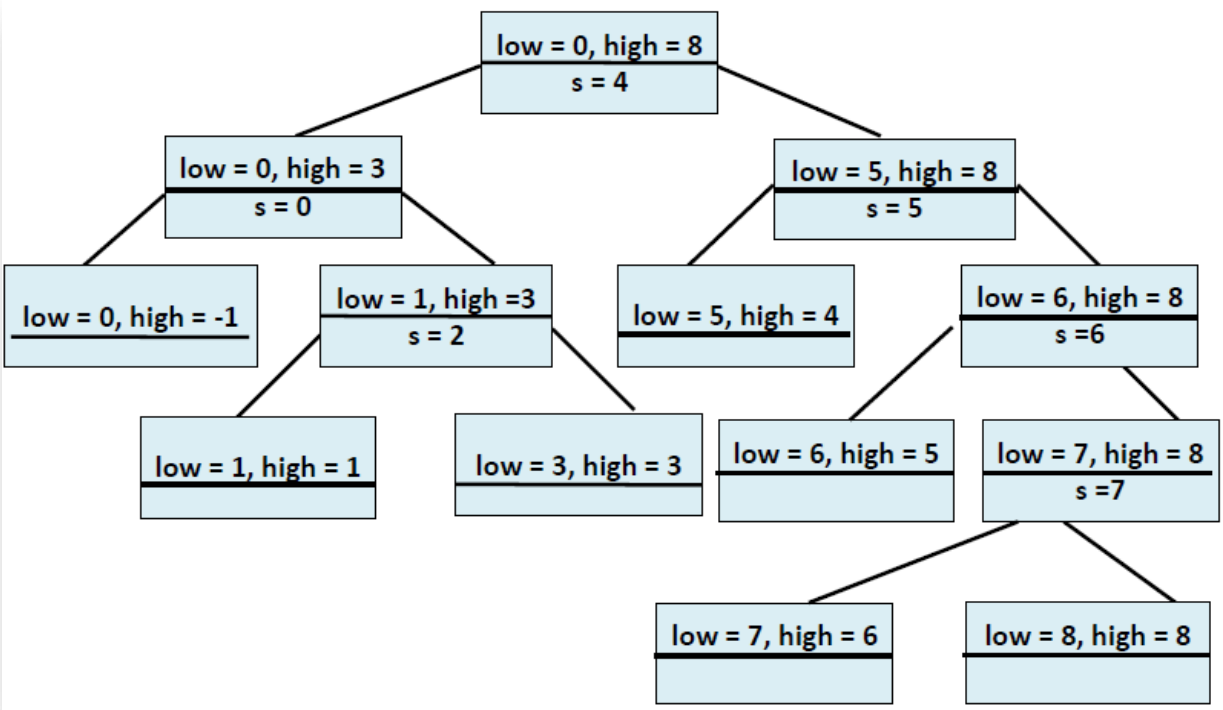


3. Finally, if the scanning indices stop while pointing to the same element, i.e., *i*=*j*, the value they are pointing to must be equal to *p*. Thus we have the array partitioned, with the split position *s* = *i* = *j* :



- We can combine the case i=j with the case of crossed-over indices (*i*>*j*) by exchanging the pivot with A[*j*] whenever *i* ≥ *j*.

Example: Draw the tree structure of the recursive calls made to sort the list {40, 20, 10, 80, 60, 50, 7, 30, 100} in ascending order using Quicksort.



Quick Sort Best case Efficiency

- The number of key comparisons made before a partition is achieved, is ***n*+1 if the scanning indices cross over**.
- The number of key comparisons made before a partition is achieved is ***n* if the scanning indices coincide**.
- If all the splits happen in the middle of corresponding subarrays, we will have the best case.
- Therefore, the number of key comparisons in the best case will satisfy the recurrence:

$$C_{best}(n) = 2\,C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0$$

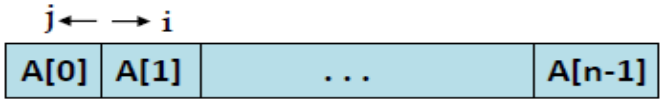
- According to the Master Theorem,

$$C_{best}(n) \in \Theta(n \log_2 n) \quad [\text{ Here, } d=1,\ a=2,\ b=2,\ a=b^d\ ]$$

QuickSort Worst case analysis

- In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, while the size of the other will be just one less than the size of the subarray being partitioned. This situation arises in particular, for **increasing arrays (already Sorted)**.

- If A[0 . . . *n*-1] is a strictly increasing array and we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0:



- So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will find itself with the strictly increasing array A[1 . . . *n*-1] to sort.

- This sorting continues until the last one A[*n*-2] . . . *n*-1] has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n + 1) + n + \ldots + 3 = \frac{(n+1)(n+2)}{2} - 3$$

$$C_{worst}(n) \in \Theta(n^2)$$

QuickSort Average Case analysis

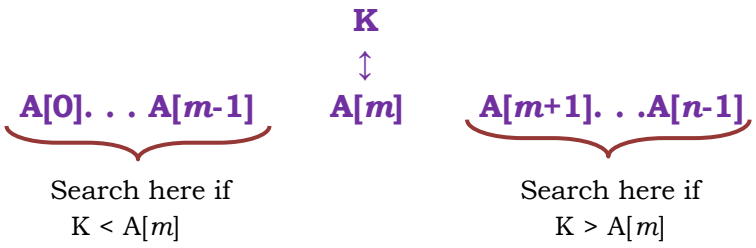
- Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n.

$$C_{avg}(n) \approx 1.38 n \log_2 n$$

- Thus, on the average, quicksort makes only 38 % more comparisons than in the best case.

Binary Search

- Binary Search is an efficient algorithm for searching in a sorted array.
- The basic algorithm is to find the middle element of the list, compare it against the key, decide which half of the list must contain the key, and repeat with that half.
- It works by comparing a search key K with the array's middle element $A[m]$.
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$ and for the second half if $K > A[m]$:



Binary Search Algorithm:

ALGORITHM ***BinarySearch*****(A[0 . . . *n*-1], K)**

```
//Implements nonrecursive binary search
//Input: An array A[0 . . . n-1] sorted in ascending order
//       and a search key K
//Output: An index of the array's element that is equal
//        to K or -1 if there is no such element
l ← 0; r ← n -1
while l ≤ r do
      m ← (l + r)/2
      if K = A[m] return m
      else if K < A[m] r ← m – 1
      else l ← m + 1
return -1
```

Analysis of Binary Search Algorithm

Worst-Case Analysis:

- For simplicity we assume that after one comparison of K with A[m], the algorithm can determine whether K is smaller, equal to, or larger than A[m].
- The worst-case inputs include all arrays that do not contain a given search key.
- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \; C_{worst}(1) = 1$$

- Assuming that $n = 2^k$, the solution to the above recurrence obtained is

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1$$

Average-Case Analysis:

- The average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n$$

********** **END OF UNIT - 2** **********