Algorithm efficiency

TEKNOTURF

Efficiency of an algorithm is based on:

- CPU time usage
- Memory usage
- Disk usage
- Network usage

Algorithm efficiency

TEKNOTURF

To find a better solution to a problem, we may use algorithms.
To choose the correct algorithm, we need to analyze the efficiency of the algorithm.

How do we analyze the algorithm?

Analysis of the algorithms are based on 3 things:

1. **Quality** and **accuracy** of the algorithm
2. How much time do the algorithms take to perform the operations (**time complexity**)
3. How much memory it requires (**Space Complexity**)

Time complexity

TEKNOTURF

Time complexity of an algorithm signifies the total time required by the algorithm to complete its execution with provided resources

It is measured in terms of number of operations rather than computer time; because computer time depends on the hardware, processor, etc.

Time for most algorithms depends on the amount of data or size of the problem. If the volume of data is doubled, the time needed for the computations is also doubled.

Time needed is proportional to the amount of data

Algorithm's performance may vary with different types of input data.
Hence, we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size

Cases in Algorithm Analysis

TEKNOTURF

Three cases to analyze an algorithm:

Worst Case Analysis:

- The worst case analysis calculates upper bound on running time of an algorithm
- It is the maximum length of running time for any input of size n
- We usually concentrate on finding only the worst-case running time

Average Case Analysis:

- Average case complexity is the complexity of an algorithm that is averaged over all possible inputs
- It is the average running time for any input of size n

Best Case Analysis:

- Best case analysis calculates lower bound on running time of an algorithm
- We must know the case that causes minimum number of operations to be executed

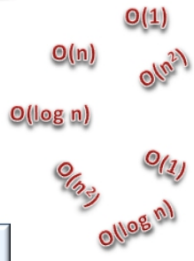
Big Oh Notation

TEKNOTURF

- The most common metric for calculating time complexity is Big O notation
- It expresses the amount of time required by the algorithm
- It can be denoted by the symbol 'O'
- Big Oh denotes "fewer than or the same as" <expression> iterations

Time complexity for different operations can be denoted as:

O(1) : An algorithm which executes on constant time period
O(n) : An algorithm which executes in linear time period
$O(n^2)$: An algorithm which executes in quadratic time period
O(log n) : An algorithm which executes in a logarithmic time period



n? Big Oh removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity

Determining the Time complexity

TEKNOTURF

```
statement;
```

The running time of the statement will not change in relation to N. Then its Time Complexity will be **Constant. (**O(1))

```
for(i=0; i < N; i++)
{
    statement;
}
```

The running time of the loop is directly proportional to N. When N doubles, so does the running time. So the time complexity for this algorithm will be **Linear** . (O(n))

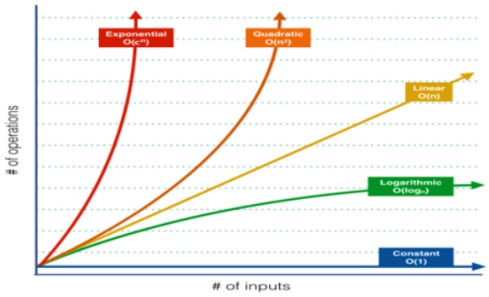
```
for(i=0; i < N; i++) {
for(j=0; j < N; j++) {
    statement;
} }
```

The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N. The time complexity for this code will be **Quadratic** . ($O(n^2)$)

```
while(low <= high) {
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1; else break;
}
```

This is an algorithm to break a set of numbers into halves. The running time of the algorithm is proportional to the number of times N can be divided by 2. This algorithm will have a **Logarithmic** Time Complexity. (O(log n))

Measure



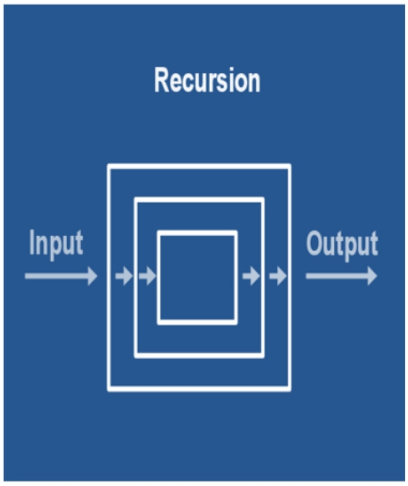
General order that we may consider:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n!) < O(n^2)$

Recursion

It is an algorithm which calls itself with "smaller" input values, and obtains the result for the current input by applying simple operations to the returned value for the smaller inputs.

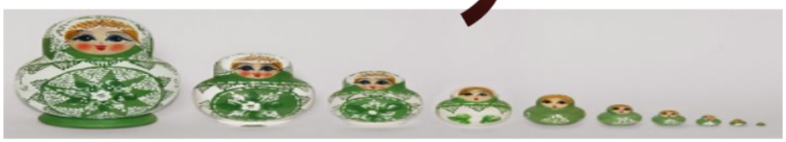
Solves difficult problems easily and provides readability but needs lot of memory



Recursion – Base case

Function call ends when condition meets base value

if (answer is known) provide the answer	base case
else make a recursive call to solve a **smaller** version of the same problem	recursive case



Factorial using recursion

How to find Factorial of 6 ?

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

```
public int  factorial(int n)
   {
          if (n <= 1)
                return 1;
          else
                n*factorial(n-1);

   }
```

Exit / Base condition

Recursive call

Iteration Vs. Recursion

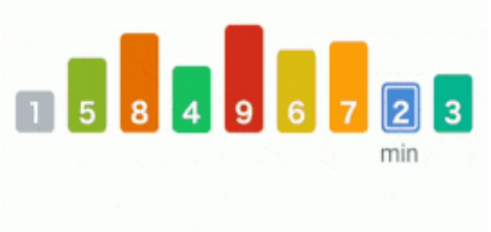
ITERATION

- It is a process of executing statements repeatedly, until some specific condition is specified
- Iteration involves four clear cut steps, initialization, condition, execution and updating
- Any recursive problem can be solved iteratively
- Iterative computer part of a problem is more efficient in terms of memory utilization and execution speed

RECURSIVE

- Recursion is a technique of defining anything in terms of itself
- There must be an exclusive if statement inside the recursive function specifying stopping condition
- Not all problems has recursive solution
- Recursion is generally a worst option to go for simple program or problems not recursive in nature

Searching and Sorting



In computer science, a searching algorithm is an algorithm that retrieves information stored within some data structure.

A sorting algorithm is an algorithm that puts elements of a list in a certain order

Searching

TEKNOTURF

The appropriate search algorithm often depends on the data structure being searched

Search algorithms can be classified based on their mechanism of searching

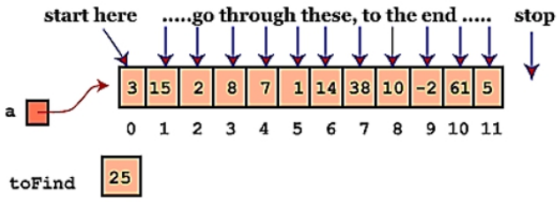
What are the most popular searching methods?

- Sequential / Linear Search
- Binary Search

Linear Search

TEKNOTURF

Linear search is a very simple search algorithm.
It searches for data by comparing each item in a list one after the other.



returns -1
for this example, because 25 is not found in the given array

```
int Search(int a[], int size, int toFind)
{
    int index , retVal = -1;
    for( index=0; index < size; index++)
    {
        if ( toFind == a[index] )
        {
            retVal = index;
            break;
        }
    }
    return retVal;
}
```

Binary Search

Binary Searching Procedure

TEKNOTURF

- This search algorithm works on the principle of divide and conquer
- It is an efficient algorithm for finding an item from an ordered list (sorted) of items

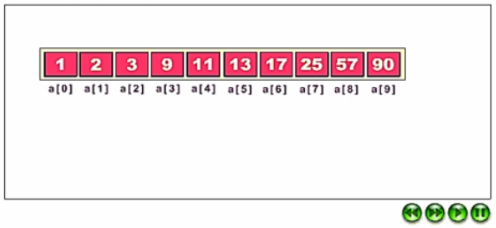
Binary search compares the target value to the middle element of the array

If the value is matched, then it returns the value

If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array

Repeat this procedure on the lower (or upper) half of the array.

Binary search



Binary Search is useful when there are large numbers of elements in an array

Binary Search

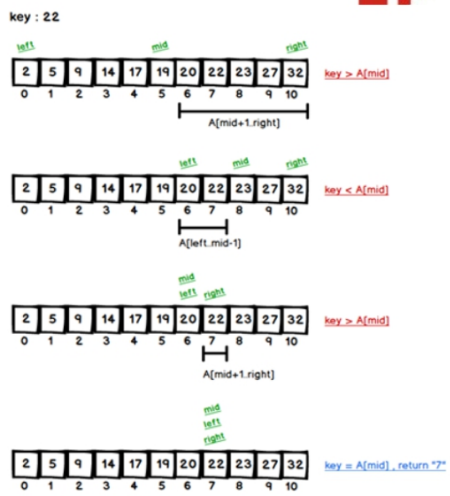
```
int binarySearch(int A[], int size, int key)
{
   int left = 0, right = size-1;
   while (left <= right) {
      mid = ( left + right)/2;
      if( key == A[mid]
           return mid;
      else if ( key < A[mid])
           right = mid – 1;
       else
           left = mid + 1;
   }
  return -1;
}
```

Returns index when key is found

When search value is less than the mid value

When search value is greater than the mid value

Returns -1 when key is not found



Sorting

TEKNOTURF

Sorting is a process that organizes a collection of data into either ascending or descending order

Types of Sorting

TEKNOTURF

Insertion Sort

Bubble Sort

Quick Sort

Merge Sort

Heap Sort

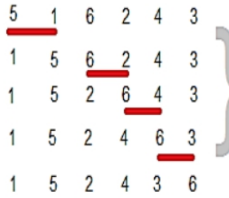
Bubble sort

Bubble Sort takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Why is it called Bubble sort?

With each iteration, the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface

Lets take this Array.



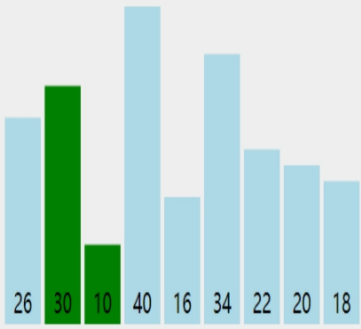
Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

Bubble sort

Bubble Sort Algorithm

```
bubbleSort(A,n)
    for i = 0 to n-1
        for j = 0 to n-i-1
            if A[j] >A[ j+1 ]
                swap a[j] <-> A[j+1]
```



Insertion Sort

It is a simple Sorting algorithm which sorts the array by shifting elements one by one

It is efficient for smaller data sets, but very inefficient for larger lists

How Insertion Sort works?

Step 1	12 **3** 1 5 8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3 12 **1** 5 8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1 3 12 **5** 8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1 3 5 12 **8**	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
	0 1 3 8 12	Sorted Array in Ascending Order

Insertion Sort

TEKNOTURF

Insertion Sort Algorithm

```
insertionSort(A: an array of items)
        for i = 1 to length(A)-1
                item = A[i]
                j = i
                while j> 0 and  A[j – 1] > item
                        A[j] = A[j – 1]
                        j = j – 1
                A[j] = item
```

6 5 3 1 8 7 2 4

Quick Sort

TEKNOTURF

Quick Sort sorts any list very quickly

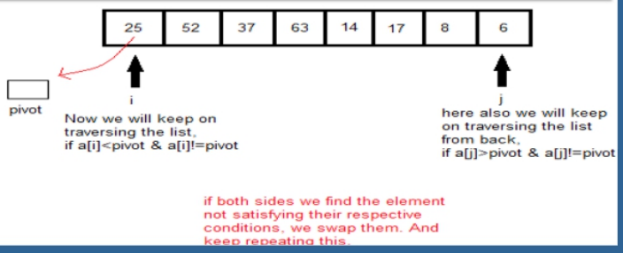
It is based on the rule of **Divide and Conquer**

This algorithm divides the list into three main parts :

- Elements less than the Pivot element
- Pivot element
- Elements greater than the pivot element

Strategy

- Choose a pivot (first element in the array)
- Partition the array about the pivot
  - items < pivot
  - items >= pivot
  - Pivot is now in correct sorted position
- Sort the left section again until there is one item left
- Sort the right section again until there is one item left



Quick Sort

TEKNOTURF

Quick Sort Algorithm

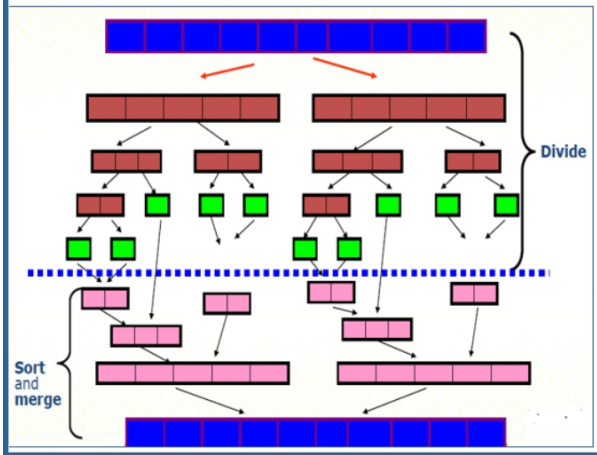
```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q – 1)
4      QUICKSORT(A, q + 1, r)
```

```
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
```

6 5 3 1 8 7 2 4

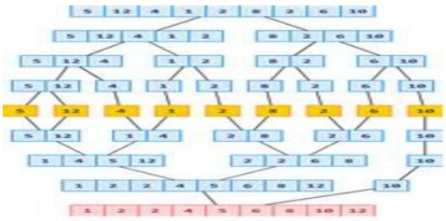
Merge Sort

Merge sort applies divide and conquer strategy



Three main steps in Merge Sort algorithm:

- **Divide** an array into halves until only one piece left
- **Sort** each half
- **Merge** the sorted halves into one sorted array



Merge Sort

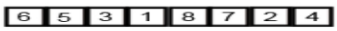
Merge Sort Algorithm

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure
```




```
procedure merge( var a as array, var b as array )

   var c as array

   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while
```

Heap Sort

Heap Sort algorithm is the best sorting method in-place with no quadratic worst-case scenarios

This algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array.

The heap is reconstructed after each removal

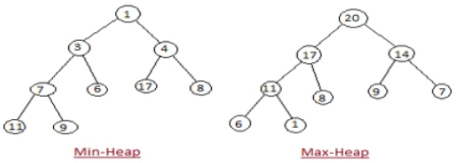
Heap is a special ***complete binary tree*** -based data structure that satisfies the special heap properties.

Heap Property : All nodes are either *greater than or equal to* or *less than or equal to* each of its children

Heap Sort

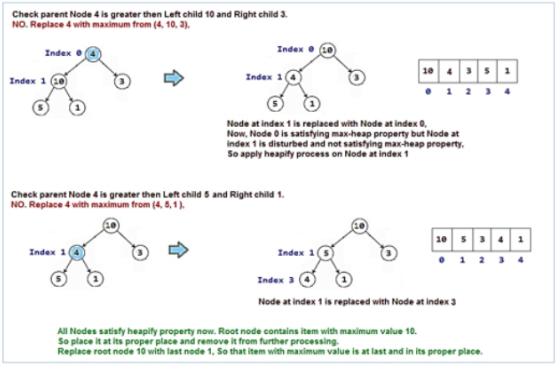
Max-Heap - If the parent nodes are greater than their child nodes

Min-Heap - if the parent nodes are smaller than their child nodes



What is heapifying process?

Heapify picks the largest or smallest child node and compares it to its parent node. If the parent node is larger or smaller than its child node, heapify quits, otherwise it swaps the parent node with the largest child node.



Heap Sort

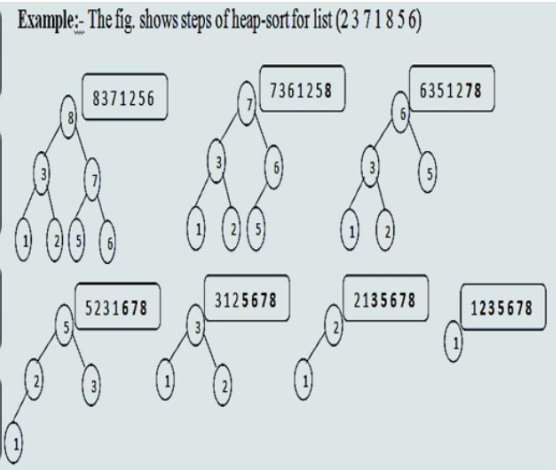
How heap sort works?

The first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap) on the unsorted list

Once the heap is built, the first element of the Heap is either largest (Max-Heap) or smallest (Min-Heap). Next, it places the first element of the heap into the sorted array.

Then it again makes the heap using the remaining elements and picks the first element of the heap and puts it into the array.

It repeats this process until the array becomes completely sorted.



Heap Sort

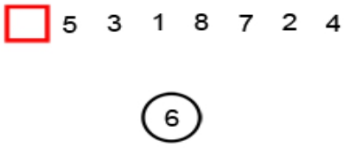
TEKNOTURF

Heap Sort Algorithm

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Sorting Comparison

TEKNOTURF

Sorting Algorithm Complexity

	Worst Case	Average Case	Best Case
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$