Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

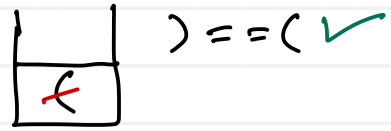
Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

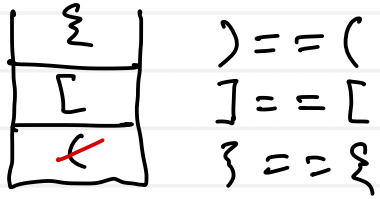
Valid Parantheses

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

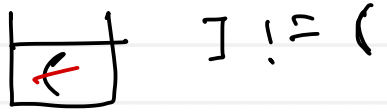
Example 1:
Input: s = "()"
Output: true



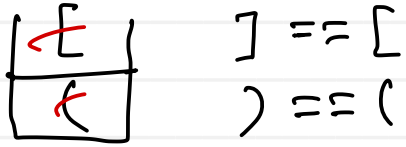
Example 2:
Input: s = "()[]{}"
Output: true



Example 3:
Input: s = "(]"
Output: false



Example 4:
Input: s = "([])"
Output: true

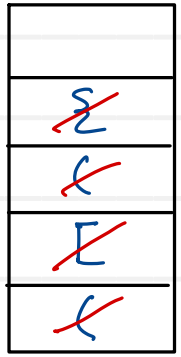


1. create stack for parenthesis
2. traverse string from left to right
   2.1 if opening parenthesis
       push it on stack
   2.2 if closing parenthesis
       if stack is not empty,
       pop opening from stack &
       compare with closing, if they
       are matching continue,
       otherwise return false.
3. if stack is not empty, return false
4. if stack is empty, return true

Parenthesis balancing using stack

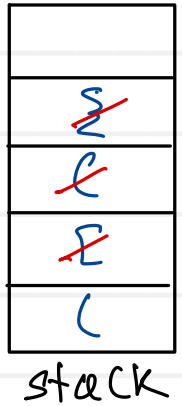
5+([9-4]*(8-{6/2}))

] == [
} == {
) == (
) == (



5+([9-4]*(8-{6/2]))

] == [
} == {
] != (



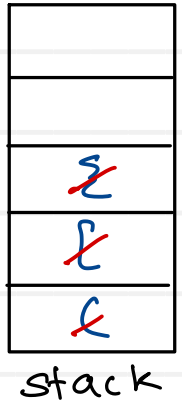
opening ([{
0 1 2

closing)] }
0 1 2

string
↓
indexOf( )
↓
returns index of char
returns -1 if char not found

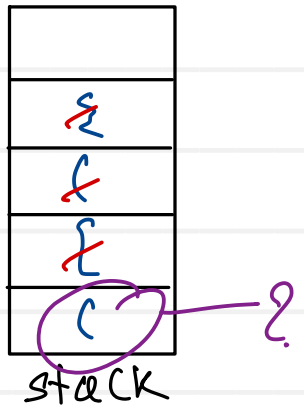
5+([9-4]*8-{6/2}))

] == [
} == {
) == (
) == {



5+([9-4]*(8-{6/2})

] == [
} == {
) == (



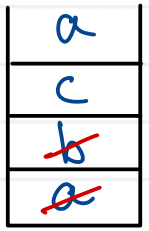
Remove all adjacent duplicates in string

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

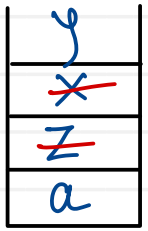
We repeatedly make duplicate removals on s until we no longer can.

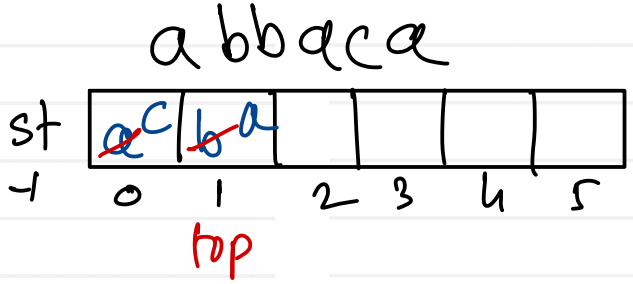
Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

Example 1:
Input: s = "abbaca"
Output: "ca"



Example 2:
Input: s = "azxxzy"
Output: "ay"






```
String removeDuplicate(String s){
    int n = s.length();
    char[] st = new char[n];
    int top = -1;
    for(int i=0; i<n; i++){
        char ch = s.charAt(i);
        if(top > -1 && ch == st[top])
            top--;
        else{
            top++;
            st[top] = ch;
    } }
    return new String(st, 0, top+1);
}
```

Algorithm

Program : set of rules / instructions to processor / CPU
Algorithm : set of instructions to human (programmer)
- step by step solution of given problem

- Algorithms are programming language independent.
- Algorithms can be written in any human understandable language.
- Algorithms can be used as a templates

Algorithm ⟶ Program / function
(Template) (Implementation)

Find sum of array elements
step 1 : create sum & initialize to 0
step 2 : traverse array for 0 to N-1 index
step 3 : Add every element of array in sum
step 4 : return / print sum

e.g. searching, sorting
↓ ↓
linear / binary selection / bubble / quick

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity
- It is done to finding time and space requirements of the algorithm
    1. Time - time required to execute the algorithm (ns, us, ms, s)
    2. Space - space required to execute the algorithm inside memory (bytes, kb, mb)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
    - time is dependent on type of machine (CPU), number of processes running at that time
    - space is dependent on type of machine (architecture), data types

growth wrt size of input (n)

- Approximate time and space analysis of the algorithm is always done
- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"
- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input
- This behaviour of the algorithm is observed in three different cases
    1. Best case
    2. Average case
    3. Worst case

Notations used to represent time/space complexity

$O()$ (upper bound)

$\Omega()$ (lower bound)

$\Theta()$ (Avg/tight bound)

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Print 1D array on console

```
void print1DArray(int arr[], int n)
{
    for(i=0; i<n ; i++)
        sysout(arr[i]);
}
```

No. of iterations = n

Time ∝ iterators

Time ∝ 1 * n

$T(n) = O(n)$

2. Print 2D array on console

```
void print2DArray(arr[][], m, n){
    for(i=0; i<m ; i++){
        for(j=0; j<n ;j++){
            sysout(arr[i][j]);
        }
    }
}
```

No. of iterations (outer loop) = m

No. of iterations (inner loop) = n

Total iterations = m * n

Time ∝ m * n

$T(m, n) = O(m * n)$

if $m \approx n$

itr = n^2

time ∝ n^2

$T(n) = O(n^2)$

Time complexity

3. Add two numbers

```
void add(int n1, int n2) {
    int sum = n1 + n2;
    return sum;
}
```

- irrespective of input values, algorithm takes constant time all the time.
- Here we are having constant time requirement, it is denoted as

$$T(n) = O(1)$$

4. Print table of given number

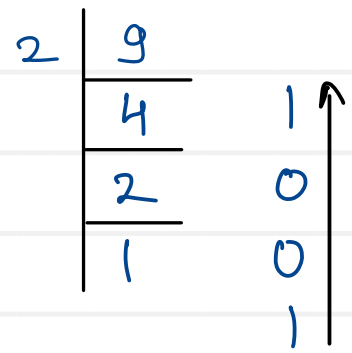
```
void printTable(int num) {
    for(i=1; i<=10; i++)
        sysout(i * num);
}
```

- loop will iterate fixed no. of time i.e 10
- constant time requirement

$$T(n) = O(1)$$

Time complexity

5. Print binary of decimal number



$(9)_{10} = (1001)_2$

```
void printbinary(int n) {
    while (n > 0) {
        sysout(n % 2);
        n = n / 2
    }
}
```

n	n > 0	n % 2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$n = 9, 4, 2, 1$

$= n, n/2, n/4 \ldots$

$= n/2^0, n/2^1, n/2^2 \ldots n/2^{itr}$

$1 = n/2^{itr}$

$2^{itr} = n$

$\log 2^{itr} = \log n$

$itr \log 2 = \log n$

$itr = \frac{\log n}{\log 2}$

Time $\propto$ itr

Time $\propto \frac{\log n}{\log 2}$

$T(n) = O(\log n)$

Time complexity

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), …., O(2^n), ……

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms of log n

```
for(i=0; i<n; i++) → O(n)
for(i=n; i>0; i--) → O(n)
for(i=0; i<n; i+=20) → O(n)
for(i=n; i>0; i/=2) → O(log n)
for(i=1; i<n; i*=2) → O(log n)
```

n = 9, 4, 2, 1
i = 1, 2, 4, 8

```
for(i=1; i<=10; i++) → O(1)
```

```
for(i=0; i<n; i++) → n
  for(j=0; j<n; j++) → n
                          * → O(n^2)
```

```
for(i=0; i<n; i++); → n
for(j=0; j<n; j++); → n
                        + = 2n → O(n)
```

```
for(i=0; i<n; i++) → n
  for(j=n; j>0; j/=2) → log n
                          * → O(n log n)
```

Time complexity

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j+n/2<=n; j++)      -> n
    for(k=2; k<=n; k=k*2)      -> log n
```

Total itr = $n * n * \log n$

$= n^2 \log n$

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j<=n; j=2*j)        -> log n
    for(k=1; k<=n, k=k*2)      -> log n
```

total itr = $n * \log n * \log n$

$= n \log^2 n$

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

(space required to store actual input)

(space required to process input)

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

input variable = arr

processing variables = n, key, i

Auxiliary space = 3 unit

constant space all the time

$S(n) = O(1)$

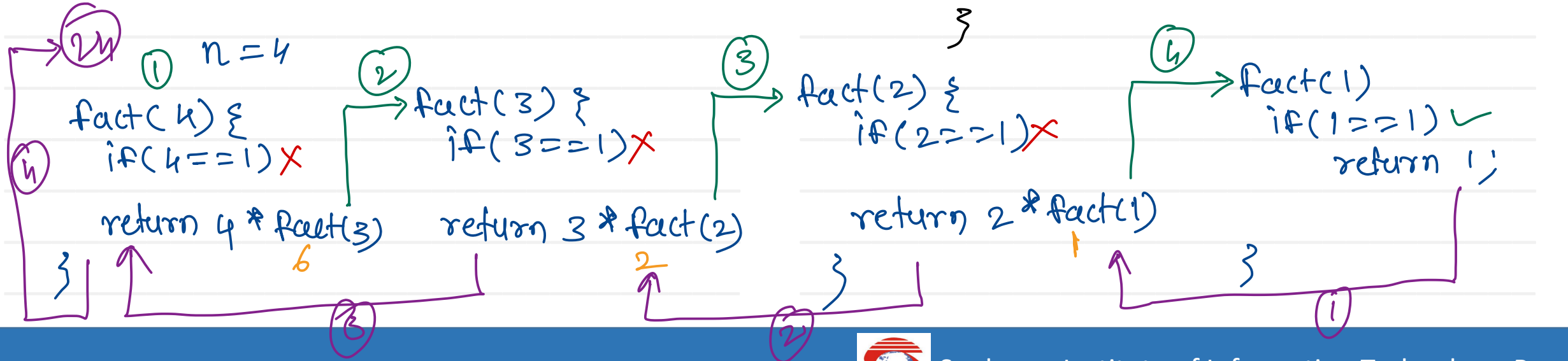
Recursion

- Calling function within itself
- we can use recursion
  - if we know formula/process in terms of itself
  - if we know terminating condition

e.g. $n! = n * (n-1)!$

$0! = 1! = 1$

```
int fact(int n) {
    if(n==1)
        return 1;
    return n * fact(n-1);
}
```



Algorithm analysis

Iterative

- loops are used

```
int fact(int num) {
    int f=1;
    for(int i=1; i<=num; i++)
        f *= i;
    return f;
}
```

n

Time ∝ no. of iterations of the loop

Time ∝ n

$T(n) = O(n)$

$S(n) = O(1)$

Recursive

- recursion is used

```
int rfact(int num) {
  if(num==1)
     return 1;
  return num * rfact(num-1);
}
```

Time ∝ No. of recursive calls

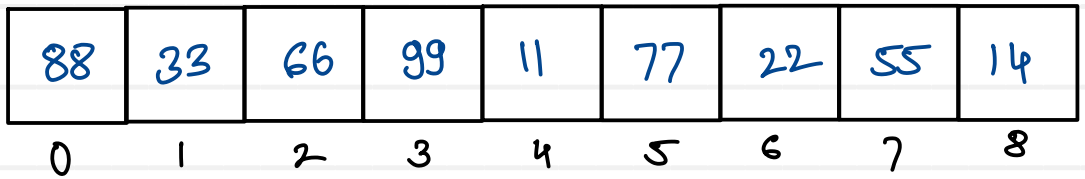
Time ∝ n

$T(n) = O(n)$

$S(n) = O(n)$

Linear search (random data)

1. decide/take key from user
2. traverse collection of data from one end to another
3. compare key with data of collection
   - 3.1 if key is matching
     return index/true
   - 3.2 if key is not matching
     return -1/false



Key == arr[i]

77

Key

i = 0, 1, 2, 3, 4, 5

Key is found

89

Key

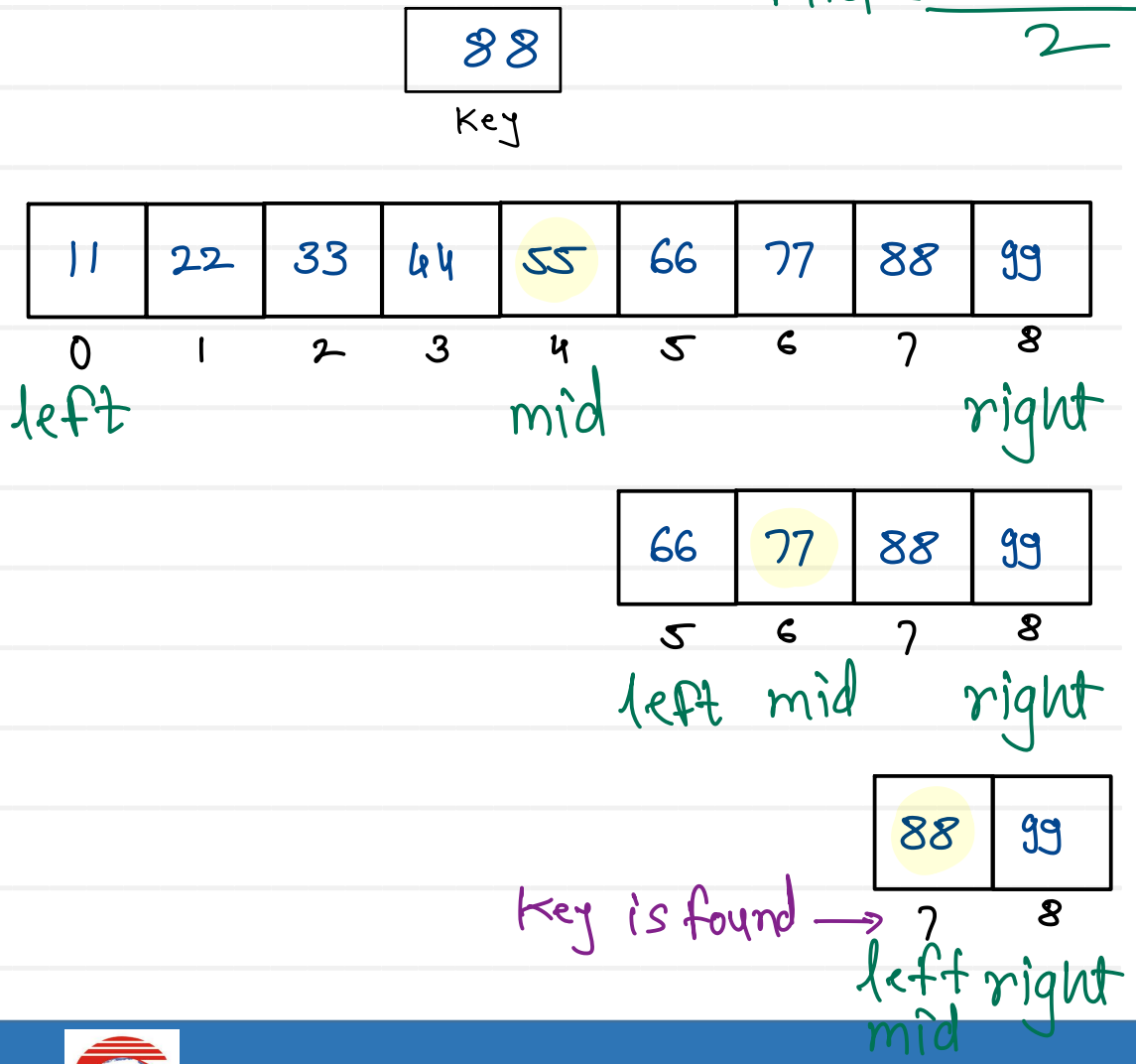
i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Key is not found

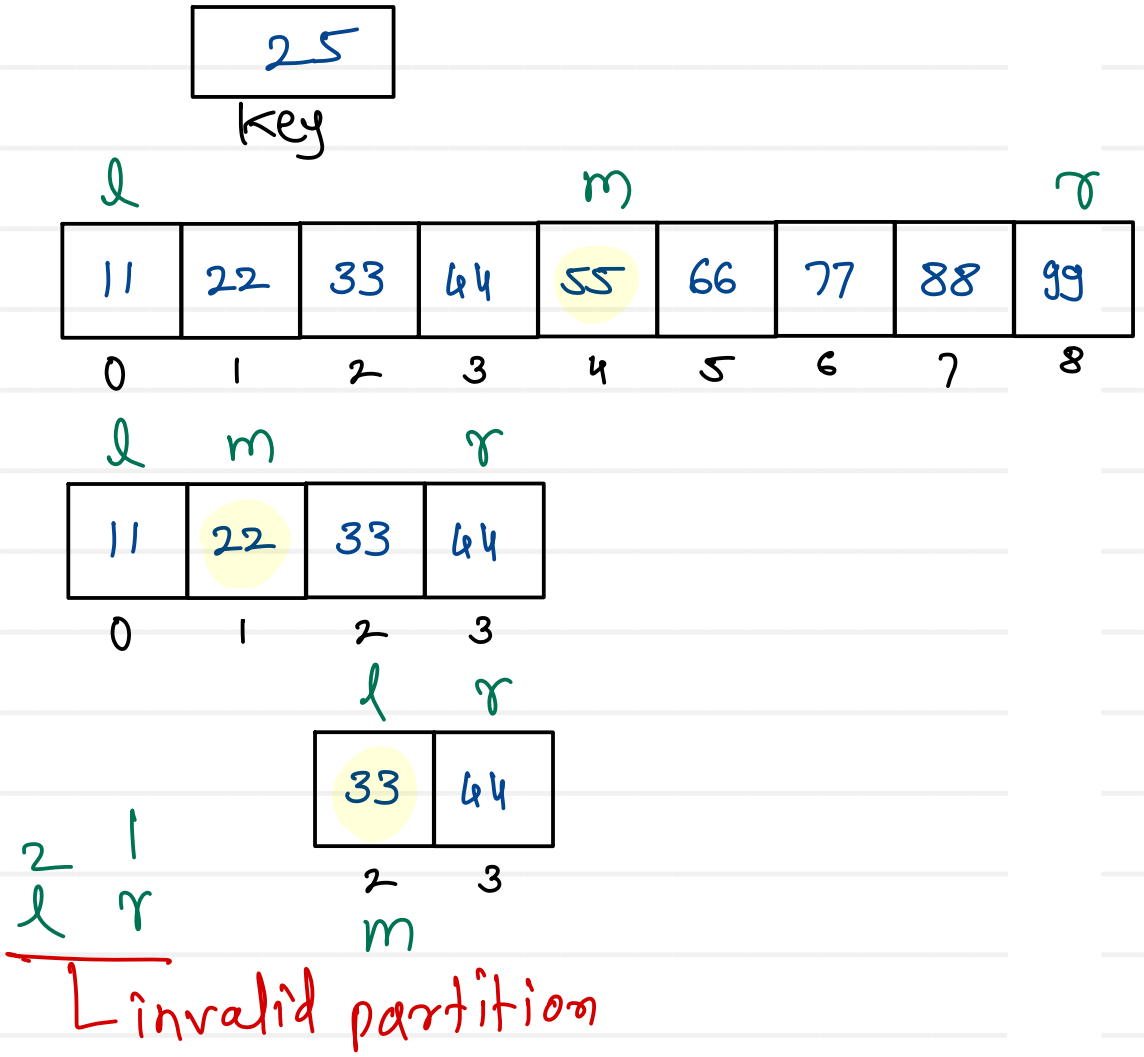
Binary search

1. take key from user
2. divide array into two parts
   (find middle element)
3. compare middle element with key
3.1 if key is matching
   return index (mid)
3.2 if key is less than middle element
   search key in left partition
3.3 if key is greater than middle element
   search key in right partition
3.4 if key is not matching
   return -1

mid = (left + right) / 2



Binary search



valid partition : left <= right
invalid partition : left > right

left partition : left → mid −1
right partition : mid+1 → right

```
l=0 , r=8 , m;
while(l <= r) {
    m = (l+r) /2;
    if( key == arr[m]
        return m;
    else if( key < arr[m])
        right = m-1;
    else
        left = m+1;
}
return -1;
```

11	22	33	44	55	66	77	88	99
0	1	2	3	4	5	6	7	8

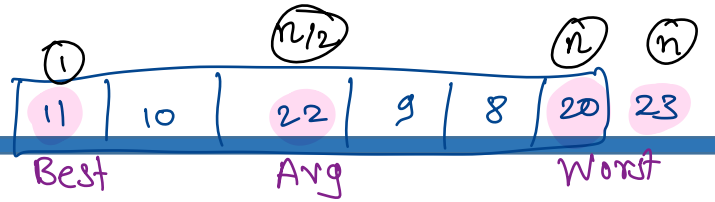
key = 88

l	r	l <= r	m
0	8	T	4
5	8	T	6
7	8	T	7

key = 25

0	8	T	4
0	3	T	1
2	3	T	2
2	1	F	

Searching algorithms analysis



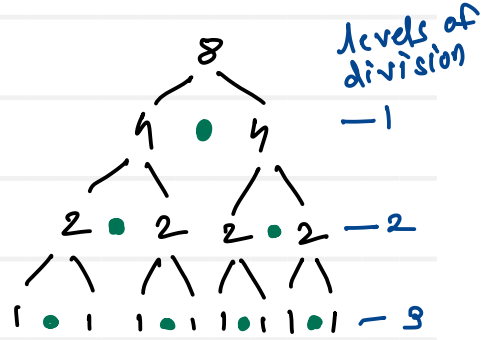
- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search
   - Best case - if key is found at few initial locations → O(1)
   - Average case - if key is found at middle locations → O(n)
   - Worst case - if key is found at last few locations / key is not found → O(n)

S(n)=O(1)

2. Binary search
   - Best case - if key is found at first few levels → O(1)
   - Average case - if key is found at middle levels → O(log n)
   - Worst case - if key is found at last level / not found → O(log n)

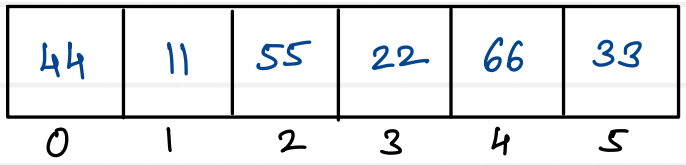
S(n)=O(1)



$2^3 = 8 \Rightarrow 2^l = n$ $\quad l = \frac{\log n}{\log 2}$

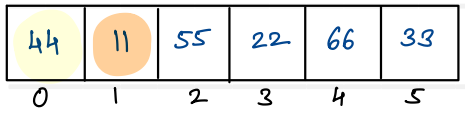
Selection sort

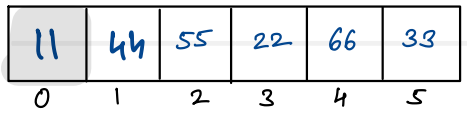
1. Select one position of the array
2. Find smallest element out of remaining elements
3. Swap selected position element and smallest element
4. Repeat above steps until array is sorted (N-1)



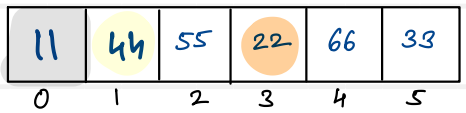
i minIndex j
3 3 4
5
6

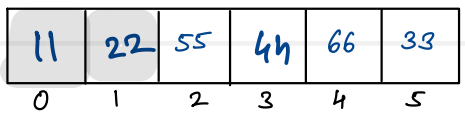
Pass 1



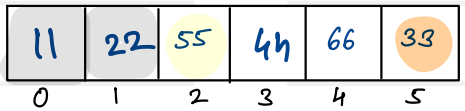


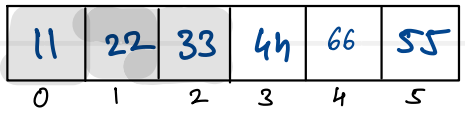
Pass 2



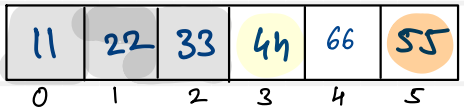


Pass 3

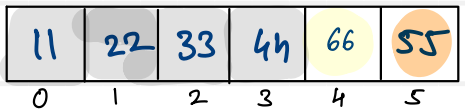


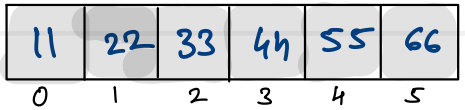


Pass 4



Pass 5

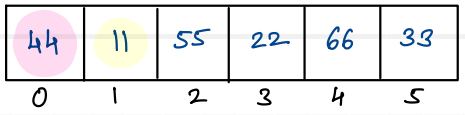




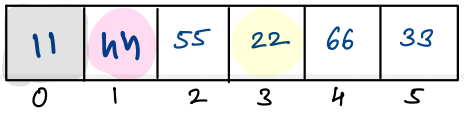
to select position : $i = 0 \rightarrow N-2$ $i < N-1$

to find minimum: $j = i+1 \rightarrow N-1$ $j < N$

Selection sort



i	minIndex	j
0	0	1
	1	2
		3
		4
		5
		6



i	minIndex	j
1	1	2
	3	3
		4
		5
		6

```
minIndex = i
for( j=i+1 ; j<N ; j++)
    if( arr[j] < arr[minIndex])
        minIndex = j;
```

Mathematical polynomial:

Degree
↳ highest power in polynomial

- degree term is always higher growing term in that polynomial.

n	n^2
1	1
10	100
100	10000
1000	1000000

No. of passes = N-1

Pass	comp
1	N-1
2	N-2
⋮	⋮
N-1	1

Total comp = $1+2+3 \cdots + (n-2) + (n-1)$ (n)

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

$$\text{Time} \propto \frac{n^2-n}{2}$$

$$T(n) = O(n^2)$$ – Best – Avg – worst

$$S(n) = O(1)$$

Bubble sort

1. Compare all pairs of consecutive elements of the array one by one
2. If left element is greater than right element , then swap both
3. Repeat above steps until array is sorted (N-1)

No. of elements = n
No. of passes = n-1

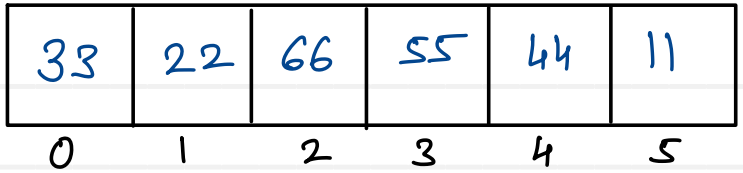
pass	comps
1	n-1
2	n-2
3	:
4	2
5	1

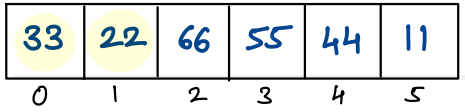
Total comps = 1+2+3---+n
$= \frac{n(n+1)}{2}$

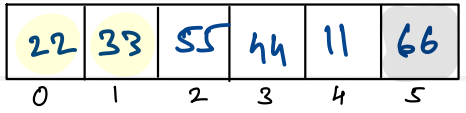
Time $\propto \frac{1}{2}(n^2+n)$

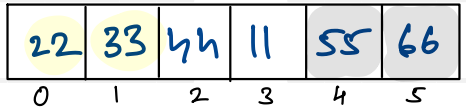
$T(n) = O(n^2)$ Avg Worst

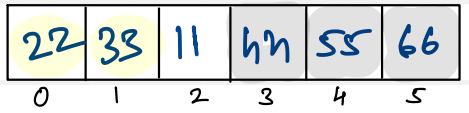
Bubble sort

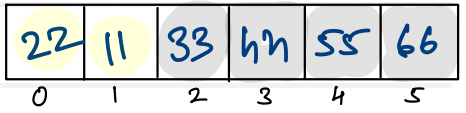

33 22 66 55 44 11
0 1 2 3 4 5

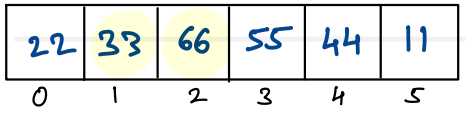

33 22 66 55 44 11
0 1 2 3 4 5

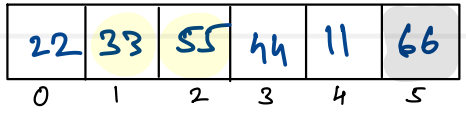

22 33 55 44 11 66
0 1 2 3 4 5

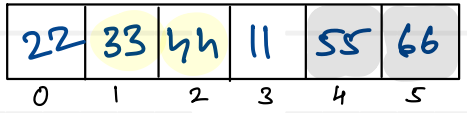

22 33 44 11 55 66
0 1 2 3 4 5

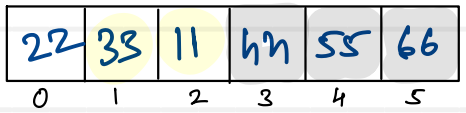

22 33 11 44 55 66
0 1 2 3 4 5

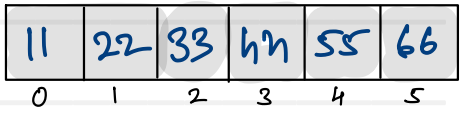

22 11 33 44 55 66
0 1 2 3 4 5

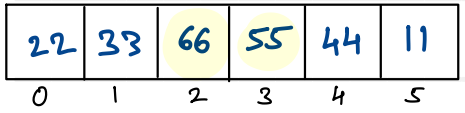

22 33 66 55 44 11
0 1 2 3 4 5

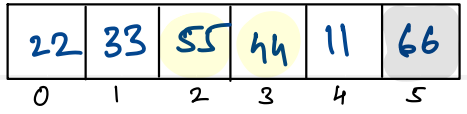

22 33 55 44 11 66
0 1 2 3 4 5

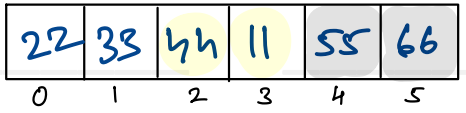

22 33 44 11 55 66
0 1 2 3 4 5

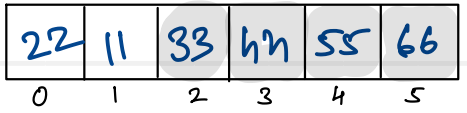

22 33 11 44 55 66
0 1 2 3 4 5

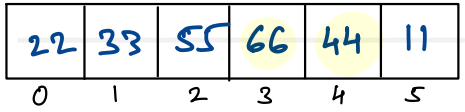

11 22 33 44 55 66
0 1 2 3 4 5


22 33 66 55 44 11
0 1 2 3 4 5

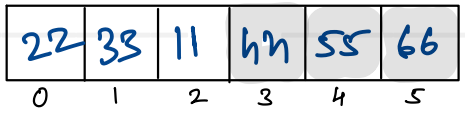

22 33 55 44 11 66
0 1 2 3 4 5

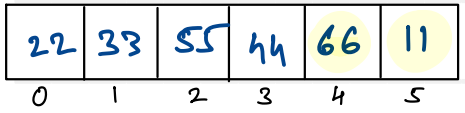

22 33 44 11 55 66
0 1 2 3 4 5

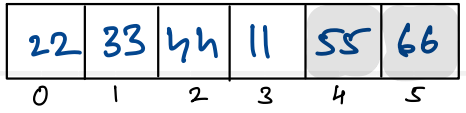

22 11 33 44 55 66
0 1 2 3 4 5

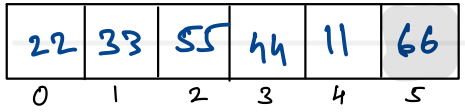

22 33 55 66 44 11
0 1 2 3 4 5


22 33 44 55 11 66
0 1 2 3 4 5


22 33 11 44 55 66
0 1 2 3 4 5


22 33 55 44 66 11
0 1 2 3 4 5


22 33 44 11 55 66
0 1 2 3 4 5


22 33 55 44 11 66
0 1 2 3 4 5

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com