Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

```
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=1 ; i<=10 ; i++) → O(1)
for ( i=n; i>0 ; i/=2) → O(log n)
for ( i=1;  i<n ; i*=2) → O(log n)
```

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

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

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

Linear search

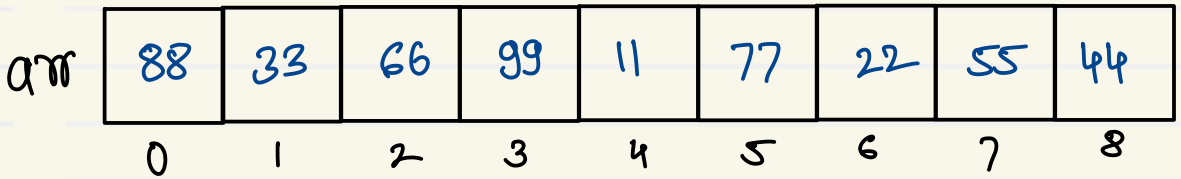
```
int linearSearch(arr, key){
    for(i=0; i<arr.length; i++){
        if(key == arr[i])
            return i;
    return -1;
}
```

n – length of array

$T(n) = O(n)$

constant space (key, i, n)

$S(n) = O(1)$



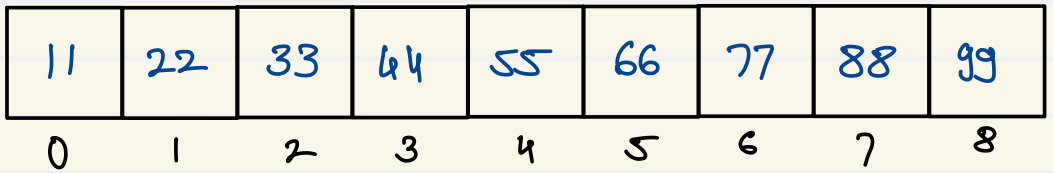
key = 88 → best case → O(1)

key = 11 → Avg case → n/2 → O(n)

key = 44/100 → worst case → n → O(n)

Binary search

```
int binarySearch(arr, key){
    l = 0 , r = arr.length-1, m;
    while (l <= r) {
        m = (l+r)/2;
        if( key == arr[m])
            return m;
        else if(key < arr[m])
            r = m-1;
        else
            l = m+1;
    }
    return -1;
}
```

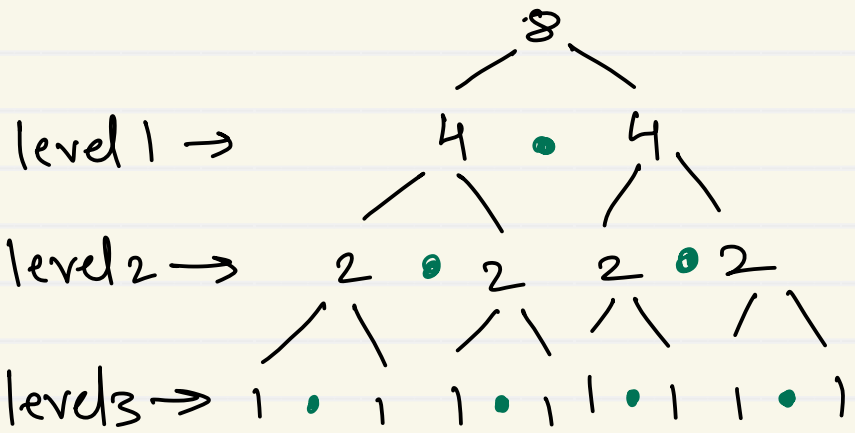


Key = 88

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

Key = 25

l	r	l <= r	m	
0	8	T	4	<
0	3	T	1	>
2	3	T	2	<
2	1	F		



n – no. of elements
l – no. of levels

$$2^l = n$$

processing variable -> l, r, m, key

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

$$2^l = n$$

$$\log 2^l = \log n$$

$$l \log 2 = \log n$$

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

Time $\propto l$

Time $\propto \frac{1}{\log 2} \log n$

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

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

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)

- calling function itself is called as recursion.

- while writing recursive functions consider
  - explain process/formula in terms of itself
  - find terminating / base condition.

e.g. $n! = n * (n-1)!$ $\quad 0! = 1$

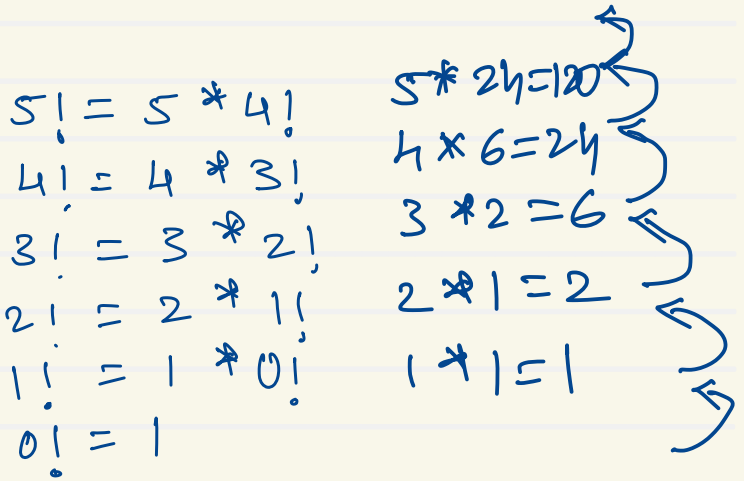
$x^y = x * x^{y-1}$ $\quad x^0 = 1$

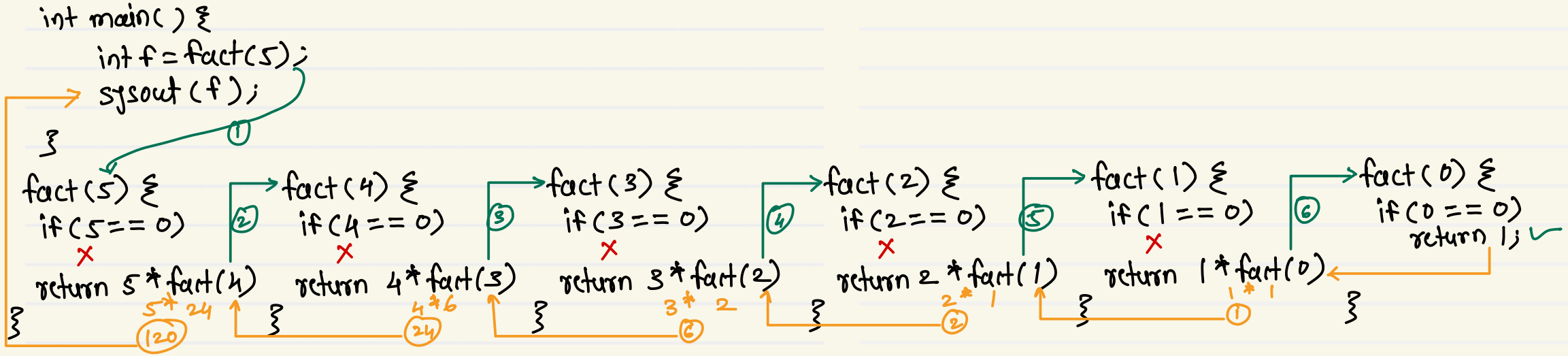
$T_n = T_{n-1} + T_{n-2}$ $\quad T_1 = T_2 = 1$

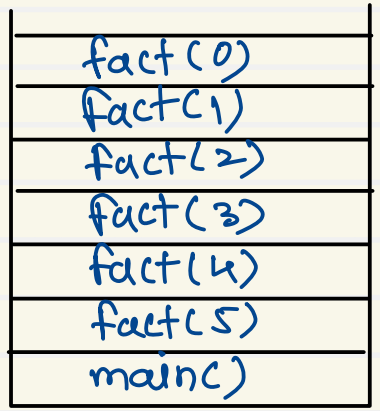
(1, 1, 2, 3, 5, 8, 13 - - - -)

- for every recursive call, function activation record / stack frame is created on stack seperately

```
int factorial(int n) {
    //base condition
    if(n==0)
        return 1;
    //explain formula in terms of itself
    return n * factorial(n-1);
}
```




int main( ) {
int f = fact(5);
sysout(f);
}
fact(5) {
if(5==0)
return 5 * fact(4)
}
fact(4) {
if(4==0)
return 4 * fact(3)
}
fact(3) {
if(3==0)
return 3 * fact(2)
}
fact(2) {
if(2==0)
return 2 * fact(1)
}
fact(1) {
if(1==0)
return 1 * fact(0)
}
fact(0) {
if(0==0)
return 1;
}
1
2
3
4
5
6
5 * 24
120
4 * 6
24
3 * 2
6
2 * 1
2
1 * 1
1


fact(0)
fact(1)
fact(2)
fact(3)
fact(4)
fact(5)
main()

Iterative approach

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

Time $\propto$ No. of iterations of the loop

Time $\propto$ n

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

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

Recursive approach

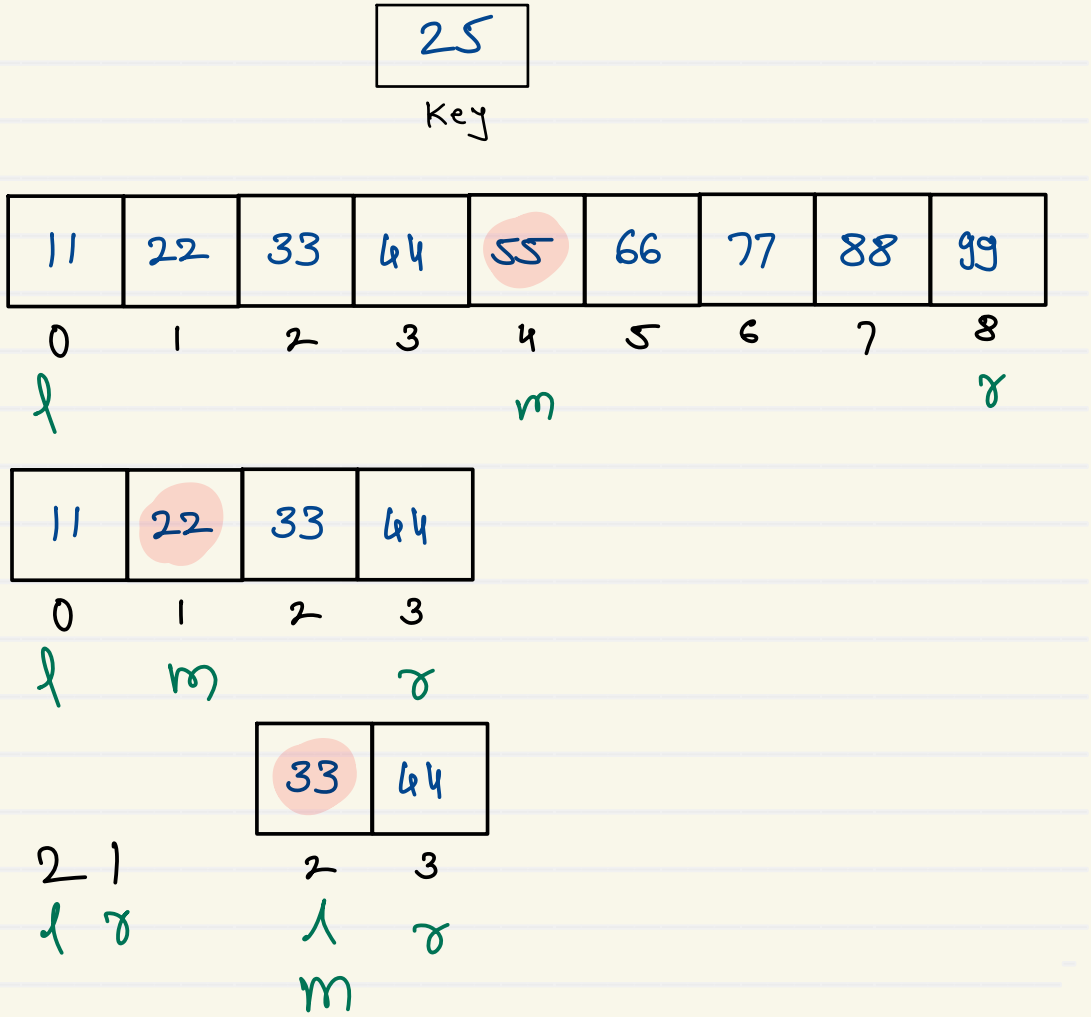
```
int factorial(int n) {
    if(n==0)
        return 1;
    return n * factorial(n-1);
}
```

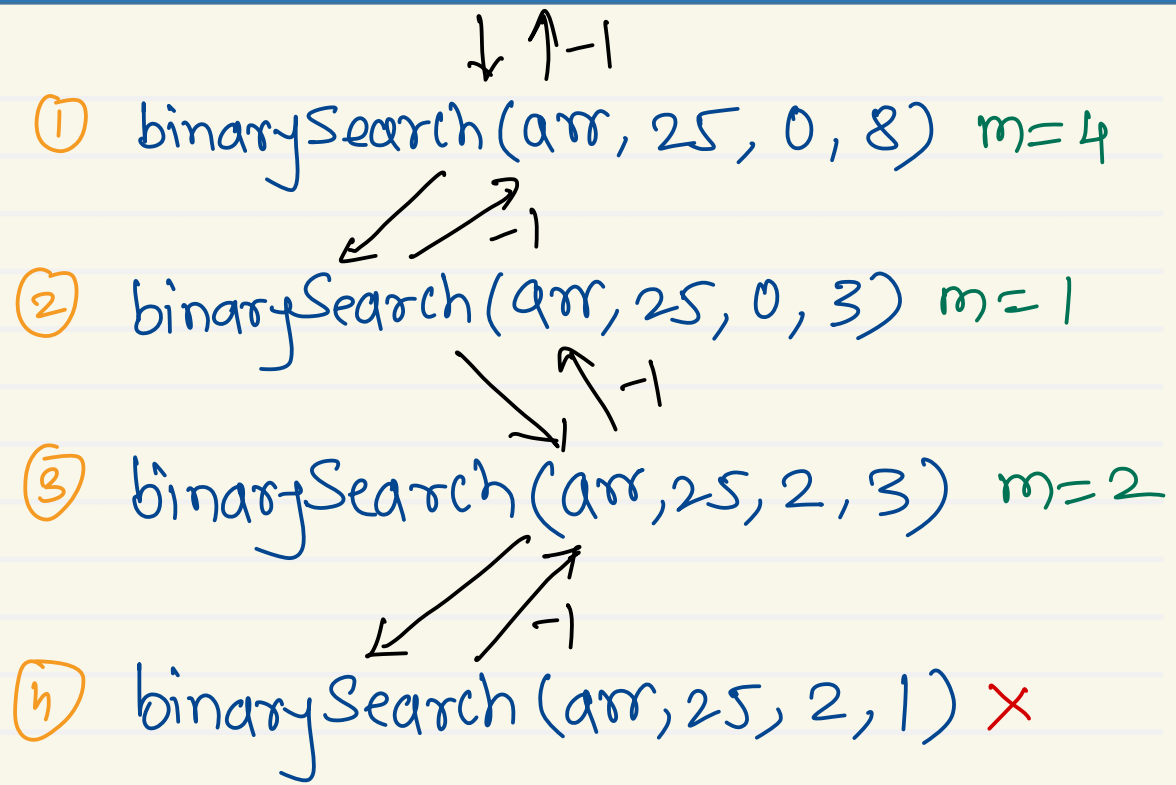
Time $\propto$ no. of recursive calls

Time $\propto$ n

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

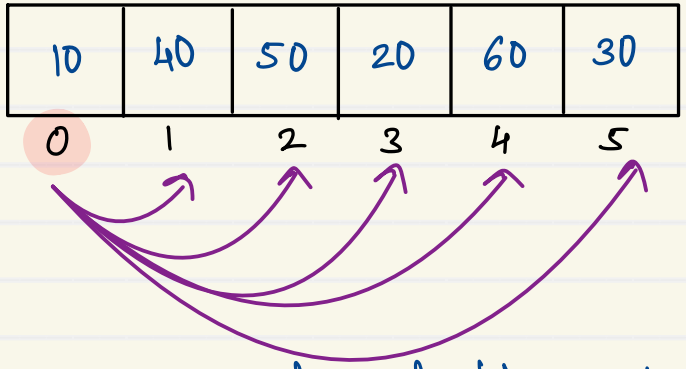
$$S(n) = O(n)$$ ← space required to create FARs

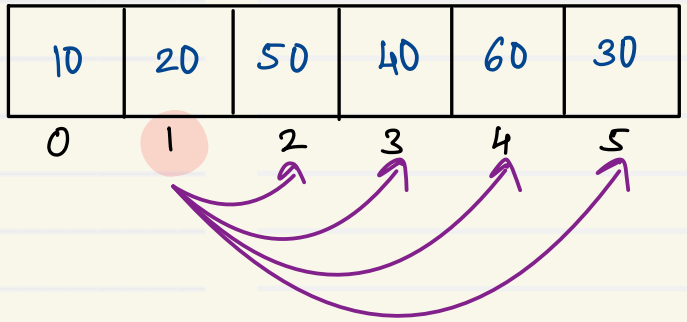


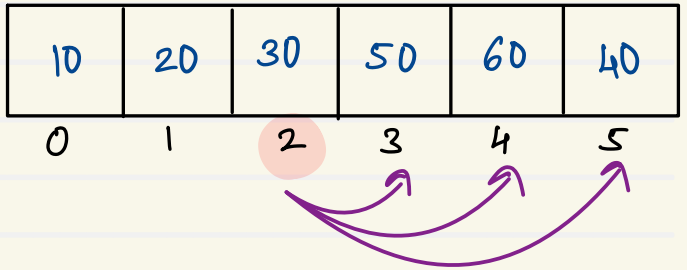


$T(n) = O(\log n)$

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







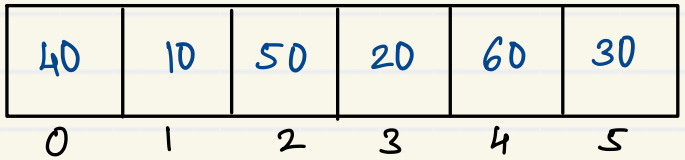

```
void selectionSort(arr, N){
    for(i=0; i<N-1; i++){
        for(j=i+1; j<N; j++){
            if(arr[i] > arr[j]){
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

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)

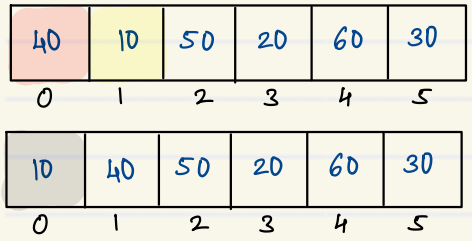
N = no. of elements

N-1 = no. of passes

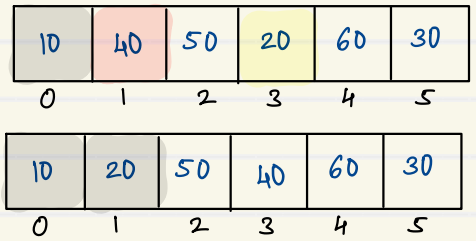


i	minIndex	j
3	3	4
		5
		6 ✗

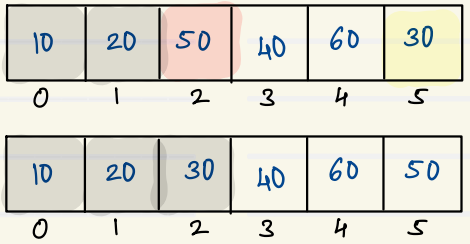
Pass 1



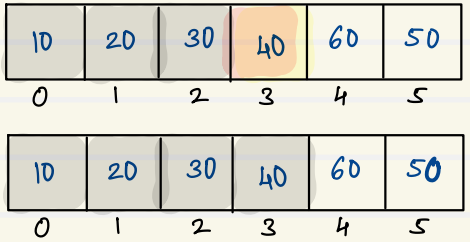
Pass 2



Pass 3



Pass 4

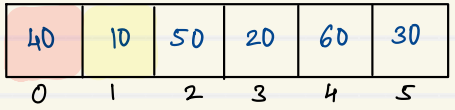


Pass 5

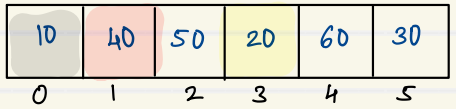
to select position of array : $i = 0 \rightarrow N-2$ $(i < N-1)$

to find smallest out of remaining elements : $j = i+1 \rightarrow N-1$ $(j < N)$

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 ✗

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

n – no. of elements

passes	comps
1	n-1
2	n-2
3	n-3
⋮	⋮
	2
n-1	1

Total comps = $1+2+3+\ldots\ldots+(n-2)+\underbrace{(n-1)}_{n}$

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

$= \frac{n^2+n}{2}$

Time $\propto$ comps

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

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

$S(n) = O(1)$ ← in place sorting algo.

Mathematical polynomials:
functions

Degree: Highest power of a variable

- while writing time / space complexities consider only degree term

e.g. $\text{Time} \propto \frac{1}{2}(n^2+n)$

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

- because degree term is highest growing term in polynomial

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