1 Introduction to Algorithms

An algorithm is a sequence of unambiguous instructions for solving a problem i.e. for obtaining a required output for any legitimate(valid) input in finite time.

1.1 Performance/Analysis of Algorithms

It refers to the memory and time representation of the program.

Methods of Analysis:

- Analytical
- Experimental

Any algorithm is analysed on the following criteria:

- Space Complexity
- Time Complexity

1.1.1 Space Complexity

It is the amount of memory required for a program to completion. It has 3 categories:

- Instruction Space (Compiled Program)
- Data Space (Space needed by var/const)
- Environment Stack Space (Recursive calls)

$\boxed{\text{Denoted by: } C + S_p}$

Sample Questions

1. Sum of array without recursion

```
1 int sum(int a[],int n)
2 {
3     int sum = 0;
4     for(int i = 0;i < n;i++)
5         sum = sum + a[i];
6     return sum;
7 }
```

Space Complexity: $6x$ bytes[a]

Reason: Line 1 occupies x bytes for pointer a and x bytes for integer n. Line 3 occupies x bytes for sum and x bytes for allocating 0. Line 4 will occupy x bytes for allocating integer i. In Line 6 space will reserved for returning data.

[a] where x is bytes occupied by int

2. Sum of array with recursion

```
1 int sum(int a[],int n)
2 {
3     if(n > 0)
4         return sum(a,n-1) + a[n-1];
5     return 0;
6 }
```

Space Complexity: $3x \times (n + 1)$ bytes[a]

Reason: Line 1 occupies x bytes for pointer a and x bytes for integer n. Line 4 will execute n times and each time space is reserved for pointer a and $n - 1$ thus giving $3x \times n$. During the last case of $n = 0$, Line 5 will be executed returing 0 thus occupying x bytes.

[a] where x is bytes occupied by int and n is the size of array

3. Linear Search using Recursion

```
1 int search(int a[],int n,int n)
2 {
3       if(n < 1) return -1;
4       if(a[n-1] == x) return x-1;
5       search(a,n-1,x);
6 }
```

Space Complexity: $8x \times (n+1)$ bytes[a]

Reason: Line 5 occupies 8 bytes for everytime it's executed and similar to previous question we get $n+1$ total executions.

[a] where x is bytes occupied by int and n is the size of array

1.1.2 Time Complexity

There are 2 approaches to estimate time:

- Operation Counter
- Step Counter

Sample Questions

1. Finding max element position

```
1 int maxpos(int a[],int n) {
2     int pos = 0;
3     for(int i = 0;i < n;i++)
4         if(a[pos] > a[i]) pos = i; return pos;
```

Time Complexity: $n-1$

Reason: Line 4 is executed $n-1$ times.

2. Polynomial Evaluation

```
int polyeval(int coeff[],int n,int x) {
    int y = 1, value = coeff[0];
    for(int i = 1;i <= n;i++) {
        y = y * x;
```

```
5        value += y * coeff[i];
6    }
7    return value;
8 }
```

$\boxed{\text{Time Complexity: } 2n \text{ Multiplication \& } n \text{ Addition}}$

Reason: Line 4 & 5 shows multiplication is done $2n$ times and addition operation is done n times.

3. Polynomial Evaluation using Horner's Algorithm

```
int horner(int coeff[],int n,int x) {
    int value = coeff[n];
    for(int i = 1;i <= n;i++)
        value = value * x + coeff[n-i];
    return value;
}
```

$\boxed{\text{Time Complexity: } n \text{ Multiplication \& } n \text{ Addition}}$

Reason: Let's assume a polynomial $3x^2 + 3x + 1$. By previous method we were simply substituting x into the equation. In case of Horner's Algorithm we simplified the expression i.e. $3x^2+3x+1 = x\underbrace{(3x+3)}_{\text{evaluated first}}+1$.

4. Rank Sorting

```
void rank(int a[],int n,int r[]) {
    for(int i=0;i < n;i++) r[i] = 0;
    for(int i=1;i < n;i++)
        for(int j = 0;j < i;j++)
            if(a[j] <= api])
                r[i]++;
            else
                r[j]++;
}

void rearrange(int a[],int n,int r[]) {
    int *n = new int[n];
```

```
13      for(int i = 0;i < n;i++) u[r[i]] = a[i];
14      for(int i = 0;i < n;i++) a[i] = u[i];
15      delete u[];
16  }
```

$\boxed{\text{Time Complexity: } \frac{n(n-1)}{2} + 2n}$

Reason: As in Line 3 iterable i goes from 1 to $n-1$ the iterable j in Line 4 goes from 0 to $i-1$ for every value of i i.e. if $i = 1$ then $j : 0 \to 0$, if $i = 2$ then $j : 0 \to 1 \ldots$ if $i = n-1$ then $j : 0 \to n-2$ which can be simplified as sum of number of iterations of j:

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

And from Line 13 & 14 we get the $2n$ operations.

5. Selection Sort

```
void SelectionSort(int a[],int n) {
    for(int size = n;size > 1;size--) {
        int j = max(a,size);
        std::swap(a[j],a[size-1]);
    }
}

int max(int a[],int n) {
    int pos = 0;
    for(inr i = 1;i < n;i++)
        if(a[pos] < a[i])
            pos = i;
    return pos;
}
```

$\boxed{\text{Time Complexity: } \frac{n(n-1)}{2} + 3(n-1)}$

Reason: The $3(n-1)$ factor comes from swapping $n-1$ times while the former follows same pattern as precious Q.4.

6. Transpose with Step-Counter method

```
1 void transpose(int **n,int r) {
2     for(int i = 0;i < r;i++)
3         for(int j = i+1;j < r;j++)
4             std::swap(a[i][j],a[j][i]);
5 }
```

$\boxed{\text{Time Complexity: } \frac{(r-1)r}{2} + \frac{r(r+1)}{2} + (r+1)}$

Reason: Let's count the steps, time taken, total execution time with a table.

Line	t	ν	$\nu \times t$
1	0	0	0
2	0	0	0
3	1	$r+1$	$r+1$
4	1	$\frac{r(r+1)}{2}$	$\frac{r(r+1)}{2}$ [a]
5	1	$\frac{(r-1)r}{2}$	$\frac{(r-1)r}{2}$
Total Time			r^2+r+1

[a] r+1 because of exit loop condition