Data Structures
Chapter 1

1. Recursion
2. **Performance Analysis**
   - Space Complexity
   - Time Complexity
   - Step Count
3. Asymptotic Analysis

Performance Analysis

- The program we write should
  1. meet the specification.
  2. work correctly.
  3. be documented properly.
  4. run effectively
  5. be readable.

  6. **use the storage effectively – space**
  7. **run timely – time**

  space & time complexity

  The **space complexity** of a program is the amount of **memory** that it needs to run to completion.

  The **time complexity** of a program is the amount of computer **time** that it needs to run to completion.

Performance Analysis

- **Space complexity:**

- Fixed space requirements : **c**
  - that do not depend on input size, simple or fixed-size variables

- Variable space requirements : $S_{p(I)}$
  - that depend on the instance I, stack, variable

The total space requirement for the program P:

$$\boldsymbol{S(P) = c + S_p(I)}$$

where **c** is a constant for fixed space and variable space for the instance I.

We are concerned about only $\boldsymbol{S_p(I)}$, but **not c**. **Why?**
Because we usually **compare** the algorithms of the programs.

Performance Analysis

- Space complexity: $S(P) = c + S_p(I)$
- Example: $S_{sum}(n)$ = ?

Program sum

```
float sum(float list[], int n) {
  float total = 0;
  for (int i=0; i<n; i++)
    total += list[i];
  return total;
}
```

$\boldsymbol{S_{sum}(n)} = 0$ since the C/C++ passes list[] by its address.

Performance Analysis

- **Space complexity:** $S(P) = c + S_p(I)$

- **Example:** $S_{rsum}(n) = ?$

Program rsum	Program rsum
`float rsum(float list[], int n) {` `  if (n)` `    return rsum(list, n-1) + list[n-1];` `  return 0;` `}`	`float rsum(float list[], int n) {` `  if (n)` `    return rsum(list, n-1) + list[n];` `  return 0;` `}`

Performance Analysis

- Space complexity: $S(P) = c + S_p(I)$

- **Example: $S_{rsum}(n)$ = ?**

Program rsum

```
float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}
```

The variable space requirement are for **two** parameters and **one** return address are saved in the system stack **per recursive call**:

$$sizeof(n) + list[]\,address + return\,address = 12$$

assuming 32 bit address

Performance Analysis

- Space complexity: $S(P) = c + S_p(I)$

- **Example: S_{rsum}(n=MAX_SIZE) = ?**

Program rsum

```
float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}
```

The variable space requirement are for **two** parameters and **one** return address are saved in the system stack **per recursive call**:

$$sizeof(n) + list[]\,address + return\,address = 12$$

$$\boldsymbol{S_{sum}(n) = 12 * n}$$

Performance Analysis

- **Time complexity:** The time taken by the program P:
  - $\boldsymbol{T(P) = compile\ time\ c + execution\ time\ T_p(n)}$
- Similarly, we are concerned about only $\boldsymbol{T_p(n)}$, but not c.
- **Example:** $Tp(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$
  - where n – number of execution, c for constant time for operation

We are not concerned about this, but …

- **Program step:** a meaningful program segment whose execution time is independent of the instance characteristics.
- **Example:**
  - $a = 2;$ ⇨ *1 step!!*
  - $a = 2 * b + 3 * c/d - e + f/g/a/b/c\,;$ ⇨ *1 step!!*

Performance Analysis

- **Example:** How many **program steps** required?

Program sum	2n+3
`float sum(float list[], int n) {`	
`  float total = 0;`	1
`  for (int i=0; i<n; i++)`	n+1
`    total += list[i];`	n
`  return total;`	1
`}`	

Performance Analysis

- **Example:** How many **program steps** required?

<table>
<tr><th>Program rsum</th><th>2n+2</th></tr>
<tr><td><pre>float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}</pre></td><td><pre>
n+1
n
1
</pre></td></tr>
</table>

Performance Analysis

- **Comparison:**

Program sum

```
float sum(float list[], int n) {
  float total = 0;
  for (int i=0; i<n; i++)
    total += list[i];
  return total;
}
```

Program rsum

```
float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}
```

2n + 3 > 2n + 2

sum > rsum

$T_{iterative} > T_{recursive}$

Performance Analysis

- **Example:** How many **program steps** required?

Program sum of matrix	
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],` `int c[][MAX_SIZE}, int rows, int cols) {` `for(int i=0; i<rows; i++)` `for(int j=0; j<cols; j++)` `c[i][j] = a[i][j] + b[i][j];` `}`	 `rows + 1` `rows * (cols+1)` `rows * cols`

step count = **2 rows*cols + 2 rows + 1**

Step Count Example 1:

- **What is the exact number of times sum++ executed?**

	Step count
`int sum = 0;`	1
`for (int i = 1; i <= n*n; i++)`	n * n + 1
`    for (int j = 1; j <= i; j++)`	2 + 3 + … + n*n+1
`        sum++;`	?

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Step Count Example 2:

- **What is the exact number of times sum++ executed?**

	Step count
`int sum = 0;`	1
`for (int i = 1; i <= n; i++)`	n + 1
`    for (int j = n; j >= i; j--)`	(n+1) + (n) + (n-1) + … + 2
`        sum++;`	?

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Step Count Example 3:

- **What is the exact number of times sum++ executed?**

	Step count
`int sum = 0;` `while (n >= 1) {` `    sum++;` `    n /= 2;` `}`	n / 2^k = 1

```
We have to find the smallest k such that n / 2^k = 1
```

```
n / 2^k = 1
n = 2^k
log(n) = log(2^k)
log(n) = k
```

Step Count Example 4:

- Compute the following series:

a) $1 + 2 + 3 + \ldots + 9 + 10 =$

b) $1 + 2 + 3 + \ldots + (N - 1) + N =$

c) $1 + 2 + 4 + \ldots + 32 =$

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Data Structures
Chapter 1

1. Recursion
2. **Performance Analysis**
   - Space Complexity
   - Time Complexity
   - Step Count
3. Asymptotic Analysis