1.7

Performance Analysis and Measurement

1.7 Performance Evaluation

- Two aspects:
  - **Space Complexity**
    - How much memory space is used?
  - **Time Complexity**
    - How much execution time is needed?
- Two approaches:
  - **Performance Analysis**
    - machine independent
    - a prior estimate
  - **Performance Measurement**
    - machine dependent
    - a posterior measure

1.7.1 Uses Of Performance Analysis

- Determine practicality of algorithm
- Predict run time on large instance
- Compare algorithms with different complexity
  - e.g., $O(n)$ v.s. $O(n^2)$

1.7.1

Performance Analysis

- Space complexity : $S(P) = C + S_P(I)$
- C is a **fixed** part:
  - Independent of the size of input and output.
  - Space for instruction and static variables, fixed-size structured variables, constants.
- $S_P(I)$ is a **variable** part:
  - Depends on the specific problem instance.
  - Space of referenced variable and recursion stack space (**Instance Characteristics**).

20

Instance Characteristics (I)

- Commonly used characteristics (I) include the size of the **input** and **output** of the problem.
- We shall concentrate solely on estimating the 2nd part, $S_P(I)$.
- Ex1. sorting(A[], n)
  Then I= number of integers = n.
- Ex 2. Summation of 1 to n, i.e., 1+2+3+…+n
  Then I= value of n = n.

21

Space Complexity: Simple Function

```
float Abc(float a, b, c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- I = a, b, c
- C = space for the program + space for variables a, b, c, Abc = constant
- $S_{Abc}(I) = 0$
- $S(Abc) = C + S_{Abc}(I) =$ constant

22

Space Complexity : Iterative Summation

```
float Sum(float *A, const int n)
{ float s = 0;
  for(int i=0; i<n; i++)
      s += A[i];
  return s;
}
```

- $I = n$ (number of elements to be summed)
- C= constant
- $S_{Sum}(I) = 0$ (A stores only the address of array)
- $S(Sum) = C + S_{Sum}(I) =$ constant

23

Space Complexity : Recursive Summation

```
float Rsum(float *A, const int n)
{
  if (n<=0) return A[0];
  else return (Rsum(A, n-1) + A[n-1]);
}
```

- $I = n$ (number of elements to be summed)
- C= constant
- Each recursive call "Rsum" requires $4(1 + 1 + 1) =$ 12 bytes.
- Number of calls: $Rsum(A, n) \rightarrow Rsum(A, n-1) \rightarrow \ldots \rightarrow Rsum(A, 0) \Rightarrow n + 1$ calls
- $S(Rsum) = C + S_{Rsum}(n) =$ const $+ 12\ (n + 1)$

24

1.7.1.2 Time Complexity

$$T(P) = C + T_P(I)$$

- C is a **constant**:
  - Compile time.
- $T_P(I)$ is **variable**:
  - Execution time.

25

Performance Analysis

- How to evaluate $T_P(I)$?
  - Count every Add, Sub, Multiply, … etc.
  - Practically infeasible because each instruction takes different running time at different machine.
- Use "**program step**" to estimate $T_P(I)$
  - "program step" = a statement whose execution time is ***independent*** of instance characteristics(I).

abc=a+b+b*c; → one program step
a=2; → one program step

26

Time Complexity : Iterative Summation

- $I = n$ (number of elements to be summed)
- $T_{Sum}(I) = 1 + n + 1 + n + 1 = 2n + 3$
- $T(Sum) = C + T_{Sum}(n)$ = const+$(2n + 3)$

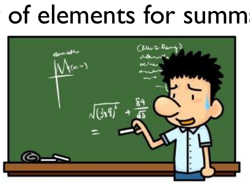
```
float Sum(float *A, const int n)
{ float s = 0;              // 1 step
  for(int i=0; i<n; i++)    // n+1 steps
      s += A[i];            // n steps
  return s;                 // 1 step
}
```

27

Time Complexity : Recursive Summation

```
float Rsum(float *A, const int n)
{
  if (n<=0)                          // 1 step
    return A[0];                     // 1 step
  else return(Rsum(A,n-1)+A[n-1]); // 1 step
}
```

- I = n (number of elements for summation)
- $T_{Rsum}(n)$ = ?



28

Time Complexity : Recursive Summation

```
float Rsum(float *A, const int n)
{
  if (n<=0)                              // 1 step
    return A[0];                         // 1 step
  else return (Rsum(A, n-1) + A[n-1]);// 1 step
}
```

- $I = n$ (number of elements for summation)
- $T_{Rsum}(0) = 2$
- $T_{Rsum(n)} = 2 + T_{Rsum(n-1)}$
  $= 2 + (2 + T_{Rsum}(n-2))$
  $= \ldots$
  $= 2n + T_{Rsum}(0) = 2n + 2$

29

Time Complexity : Matrix Addition

```
void Add(int **a, int **b, int **c, int m, int n)
{
  for(int i=0; i<m; i++)     // m+1 steps
   for(int j=0; j<n; j++)    // m*(n+1) steps
      c[i][j] = a[i][j]+b[i][j]; // m*n steps
}
```

- $I = m(rows), n\,(columns)$
- $T_{Add(I)} = (m+1) + m(n+1) + mn$
  $= 2mn + 2m + 1$
- $T(Add) = C + T_{Add}(I)$
  $= const + (2mn + 2m + 1)$

30

Observation on Step Counts

- In the previous examples :
  $T_{Sum}(n) = 2n + 3$ steps
  $T_{Rsum}(n) = 2n + 2$ steps
- So, Rsum is faster than Sum?
  - **No!**
  - ∵The execution time of each step is different.
- "**Growth Rate**" is more critical
  - "*How the execution time changes in the instance characteristics?*"

31

Program Growth Rate

- In the Sum program, $T_{Sum}(n) = 2n + 3$ means when n is tenfold $(10X)$, the execution time $T_{Sum}(n)$ is tenfold $(10X)$.
- We say that Sum program runs in **linear** time.
- $T_{Rsum}(n) = 2n + 2$ also runs in **linear** time.
- We say $T_{Sum}(n)$ and $T_{Rsum}(n)$ have the same growth rate, and are equal in time complexity!

32

1.7.1.3

Asymptotic Notation

- To make meaningful (but inexact) statements about the time and space complexities of a program.
  - Predict the growth rate.
- Two programs with time complexity
  - P1: $c_1n^2 + c_2n$
  - P2: c_3n
  - Which one runs faster?

33

1.7.1.3

Asymptotic Notation

- Scenario 1: $c_1 = 1, c_2 = 2$, and $c_3 = 100$
  - $P1(n^2 + 2n) \leq P2(100n)$ for $n \leq 98$.
- Scenario 2: $c_1 = 1, c_2 = 2$, and $c_3 = 1000$
  - $P1(n^2 + 2n) \leq P2(1000n)$ for $n \leq 998$.
- No matter what values c_1, c_2 and c_3 are, there will be an n beyond which $c_1n^2 + c_2n > c_3n$
- Therefore, we should compare the complexity for a ***sufficiently large value*** of n

34

Notation: Big-O (O)

- Definition:
  $f(n) = O(g(n))$ iff there exist c, $n_0 > 0$ such that $f(n) \le cg(n)$ for all $n \ge n_0$.
- Ex1. $3n + 2 = O(n)$
  - $3n + 2 \le 4n$ for all $n \ge 2$
- Ex2. $100n + 6 = O(n)$
  - $100n + 6 \le 101n$ for all $n \ge 6$
- Ex3. $10n^2 + 4n + 2 = O(n^2)$
  - $10n^2 + 4n + 2 \le 11\,n^2$ for all $n \ge 5$

The ***upper bound*** or ***worst-case running time***

Notation: Omega (Ω)

- Definition: $f(n) = \Omega(g(n))$ iff there exist $c, n_0 > 0$ such that $f(n) \ge cg(n)$ for all all $n \ge n_0$.
- Ex1. $3n + 2 = \Omega(n)$
  - since $3n + 2 \ge 3n$ for all $n \ge 1$
- Ex2. $100n + 6 = \Omega(n)$
  - since $100n + 6 \ge 100\,n$ for all $n \ge 1$
- Ex3. $10n^2 + 4n + 2 = \Omega(n^2)$
  - since $10n^2 + 4n + 2 \ge n^2$ for all $n \ge 1$

The ***lower bound*** or ***best-case running time***

36

Notation: Theta (Θ)

- Definition: $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Ex1. $3n + 2 = \Theta(n)$
- Ex2. $100n + 6 = \Theta(n)$
- Ex3. $10n^2 + 4n + 2 = \Theta(n^2)$

The ***tight bound*** or ***average-case running time***

37

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0, a_m > 0$, then $f(n) = O(n^m)$.

- $3n + 2 = O(n)$
- $100n + 6 = O(n)$
- $10n^2 + 4n + 2 = O(n^2)$
- $6n^4 + 1000\, n^3 + n^2 = O(n^4)$

- ***Leading constants*** and ***lower-order terms*** do not matter.

38

Theorem 1.2 Proof

$$\begin{aligned} f(n) &= a_m n^m + \ldots + a_1 n + a_0 \\ &\le |a_m| n^m + \ldots + |a_1| n + |a_0| \\ &\le n^m(|a_m| + \ldots + |a_1| + |a_0|) \\ &\le n^m c \ \text{ for } n \ge 1 \end{aligned}$$

So, $f(n) = O(n^m)$

39

Quiz

- n^2 - 10n – 6 = O(?)
- n + log n = O(?)
- n + n log n = O(?)
- n^2 + log n = O(?)
- $2^n + n^{10000}$ = O(?)
- $n^4 + 1000\, n^3 + n^2 = O(n^4)$, True or false?
- $n^4 + 1000\, n^3 + n^2 = O(n^5)$, True or false?

40

Naming Common Functions

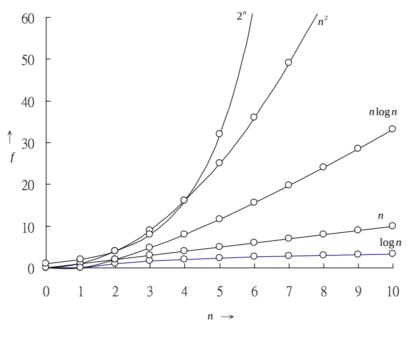
Complexity	Naming
O(1)	Constant time
O(log n)	Logarithmic time
O(n log n)	O(log n) ≤ . ≤ O(n^2)
O(n^2)	Quadratic time
O(n^3)	Cubic time
O(n^{100})	Polynomial time
O(2^n)	Exponential time

When n is large enough, the latter terms take more time than the former ones.

43

1.7.1 F1.4

Plot of Common Function Values



44

1.7.1 T1.8

Execution Time Comparison

	f (n)						
n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 µs	.03 µs	.1 µs	1 µs	10 µs	10s	1µs
20	.02 µs	.09 µs	.4 µs	8 µs	160 µs	2.84h	1ms
30	.03 µs	.15 µs	.9 µs	27 µs	810 µs	6.83d	1s
40	.04 µs	.21 µs	1.6 µs	64 µs	2.56ms	121d	18m
50	.05 µs	.28 µs	2.5 µs	125 µs	6.25ms	3.1y	13d
100	.10 µs	.66 µs	10 µs	1ms	100ms	3171y	$4*10^{13}$y
10^3	1 µs	9.96 µs	1 ms	1s	16.67m	$3.17*10^{13}$y	$32*10^{283}$y
10^4	10 µs	130 µs	100 ms	16.67m	115.7d	$3.17*10^{23}$y	…
10^5	100 µs	1.66 ms	10s	11.57d	3171y	$3.17*10^{33}$y	…
10^6	1ms	19.92ms	16.67m	31.71y	$3.17*10^7$y	$3.17*10^{43}$y	…

µs = microsecond = 10^{-6} second; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years;

45

Compute Execution Time in Big-O

- Two approaches to compute the time complexity of a program in big-O
- Approach 1:
  Step1: Compute the total step-count.
  Step2: Take big-O using theorem 1.2.
- Approach 2:
  Step1: Take big-O on each step.
  Step2: Sum up the big-O of all steps.

46

Rule of Sum

- If $f_1(n) = O(g_1(n))$, and $f_2(n)=O(g_2(n))$, then **$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.**
  - Ex. $f_1(n) = O(n)$, $f_2(n) = O(n^2)$
    Then $f_1(n) + f_2(n) = O(n^2)$.
  - Ex. $f_1(n) = O(n)$, $f_2(n) = O(n)$
    Then $f_1(n) + f_2(n) = O(n)$.
- Good for computing the time complexity of a sequential program.

47

Rule of Product

```
for (i=0; i<n; i++) {          // O(n)
    for (j=0; j<n; j++)        // O(n)
        sum := sum + 1;        // O(1)
}
```

$f(n) = O(n \cdot n \cdot 1) = O(n^2)$.

- If $f_1(n) = O(g_1(n))$, and $f_2(n)=O(g_2(n))$, then **$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.**
  - Ex. $f_1(n) = O(n)$, $f_2(n) = O(n)$
    Then $f_1(n) \cdot f_2(n) = O(n^2)$.
- Applicable to **nested loops.**

48

Complexity of Binary Search

```
int BinarySearch(int *A, const int x, const int n)
{ int left=0, right=n-1;

  while (left <= right) ──────────────────> O(?)
  { // more integers to check
    int middle = (left+right)/2; ──────────> O(1)

    if (x < A[middle])  right = middle-1; ──> O(1)

    else if (x > A[middle])  left = middle+1 -> O(1)

    else return middle; ──────────────────> O(1)
  } // end of while
  return -1; // not found
}
```

49

Complexity of Binary Search

- Analysis of the while loop:
  - Iteration 1: n values to be searched
  - Iteration 2: n/2 left for searching
  - Iteration 3: n/4 left for searching
  - …
  - Iteration k+1: $n/(2^k)$ left for searching

  When $n/(2^k) = 1$, searching must finish.
  i.e. $n = 2^k \Rightarrow k = \log_2 n$
- Hence, **worst-case exe time** of binary search is $O(\log_2 n)$.

50

1.7.2 Performance Measurement

- Obtain actual space and time requirement when running a program.
- How to do time measurement in code?
  - Method 1: Use clock(), measured in clock ticks
  - Method 2: Use time(), measured in seconds
- To time a short program, it is necessary to repeat it many times, and then take the average.

51

Performance Measurement

Method 1: Use clock(), measured in clock ticks

```
#include <time.h>

void main()
{
  clock_t start = clock();
  // main body of program comes here!
  clock_t stop = clock();
  double duration = ((double) (stop-start))
                   / CLOCKS_PER_SEC;
}
```

52

Performance Measurement

Method 2: Use time(), measured in seconds

```
#include <time.h>

void main()
{
  time_t start = time(NULL);

  // main body of program comes here!

  time_t stop = time(NULL);

  double duration = (double) difftime(stop,start);
}
```

53