Ch4. Performance Measurement

SNU
IDB Lab.

Preview of Chapters

- Chapter 2
  - How to analyze the space and time complexities of program
- Chapter 3
  - Review asymptotic notations such as O, Ω, Θ, o for simplifying the performance analysis
- Chapter 4
  - Show how to measure the actual run time of a program by using a clocking method

Bird's eye view

- When you try to market your code
  - Memory requirements is easy to figure out
  - Running time requires need of experiments

- In this chapter
  - How to perform such an experiment for run time
  - Factors affecting running time
    - The number and type of operations
    - The memory access pattern for the data & instructions in your program

Table of Contents

Introduction

- Performance measurement
  - Obtaining the actual space and time requirements of a program
  - Dependent on the particular compiler and the specific computer
  - Space requirements can be measured easily through the compiler and the analytical method
  - Time requirements can be measured by Java method *System.currentTimeMills()*

- System.currentTimeMills()
  - Returning the present time in millisecs since midnight (GMT), January 1, 1970

- With the system clock, if we want to measure the worst case time requirements for sorting
  - Need to decide the size of input data
  - Need to determine the data that exhibit the worst case behavior

Table of Contents

Choosing Instance Size

- Want to measure the time of insertion sort for the array of n objects
  - In theory, we know $f(n) = \Theta(n^2)$
  - We can determine the quadratic function with three values of n

$$f(n) = an^2 + bn + c$$

- In practice, we need the times for more than three values of n
  - Asymptotic analysis tells us the behavior only for sufficiently large values of n
  - Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve

- Reasonable choice of the input size
  - N= 100, 200, …. , 1000
  - N= 500, 1000, 1500, …. 5000

Insertion Sort

- N = 5, input sequence = (2, 3, 1, 5, 4)
- Making a sorted array using insertion
- Best case: n-1 comparisons
- Worst case: (n-1)*n / 2 comparisons

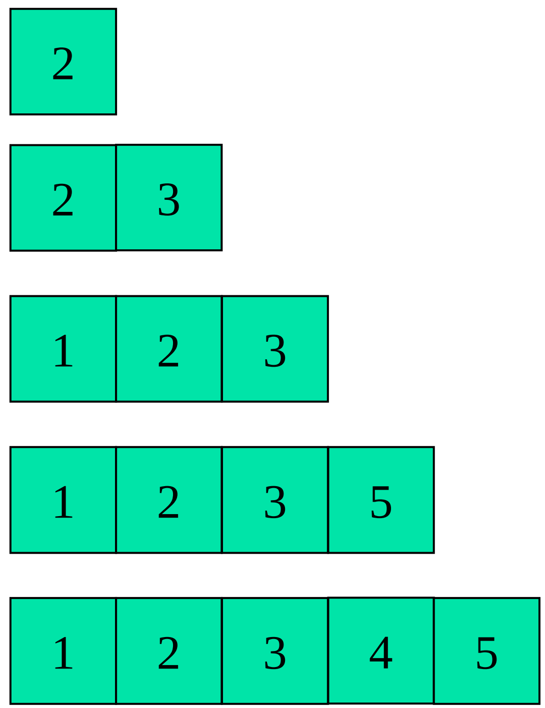


Table of Contents

Developing the Test Data

- Worst case or Best case
  - Easy to generate the test data
- Average case
  - Difficult to generate the test data
- If we cannot develop the test data showing the complexity
  - Pick the least (maximum, average) measured time from randomly generated data as an estimate of the best (worst, average) behavior

Table of Contents

Setting Up the experiment: Program 4.1

```
public static void main(String [] args){
    int step = 10;
    System.out.println("The worst-case times, in milliseconds, are");
    System.out.println("n \telapsed time");
    for (int n = 0; n <= 1000; n += step)  {
      Integer [] a = new Integer[n];    // create element array
      for (int i = 0; i < n; i++)          // initialize array
           a[i] = new Integer(n - i);
      long startTime = System.currentTimeMillis() ;
      InsertionSort2.insertionSort(a); // sort the elements
      long elapsedTime = System.currentTimeMillis() - startTime;
      System.out.println(n + "\t" + elapsedTime);
      if (n == 100) step = 100;
    }
}
```

Execution Times using program 4.1

n	Time	n	Time
0	0	100	0
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	60
50	0	600	50
60	0	700	0
70	50	800	60
80	0	900	50
90	0	1000	110

Times are in milliseconds

Experiment Accuracy

- Accuracy of measurements
  - When n is small, measured time can be inaccurate because of an error tolerance
  - Error tolerance of *System.currentTimeMills()*

$$t-100 \ \leq \ t \ \leq \ t+100 \ \ (\mathrm{t}=\mathrm{ms})$$

- To improve the accuracy upto 10%
  - Elapsed time should be 1000 msecs
  - For different data sizes
    - Repeat the program upto 1000 msecs
    - Measure the average

Insertion Sort Exp with 10% accuracy (1)

```
public static void main(String [] args) {
int step = 10;  // intially data size 10, 20,…
System.out.println("The worst-case times, in milliseconds, are");
System.out.println("n repetitions elapsed time time/sort");
for (int n = 0; n <= 1000; n += step)
  { Integer [] a = new Integer[n]; // create element array
    long startTime = System.currentTimeMillis() ;
    long counter = 0;
    do {   counter++;
           for (int i = 0; i < n; i++)   a[i] = new Integer(n - i); // initialize array
           InsertionSort2.insertionSort(a); // sort the elements }
    while(System.currentTimeMillis( ) - startTime < 1000);  // keep going upto  1000 msecs
    long elapsedTime = System.currentTimeMillis() - startTime;
    System.out.println (n + " " + counter + " " + elapsedTime + " " + ((float) elapsedTime)/counter
    if (n == 100) step = 100; // after 100, data size → 100, 200, 300, …
  }
}
```

Insertion Sort Exp with 10% accuracy (2)

n	Repetitions	Total Time	Time per Sort
0	11273	1050	0.09
10	8842	1050	0.12
20	6891	1040	0.15
30	5126	1040	0.20
40	3890	1050	0.27
50	3093	1040	0.34
60	2426	1040	0.43
70	1928	1050	0.54
80	1577	1040	0.66
90	1309	1040	0.79
100	1109	1050	0.95

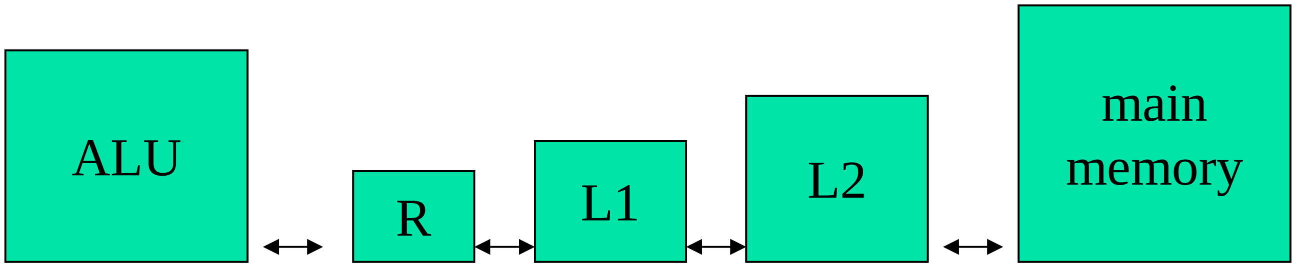
Times are in milliseconds

* fixed given time * fixed given data

Table of Contents

A Simple Computer Model (1/2)

- Consider a simple computer model



ALU: Arithmetic Logical Unit
R: Register
L1: Level 1 Cache
L2: Level 2 Cache

A Simple Computer Model (2/2)

- Cycle of our model
  - Time needed to load data
    - 2 cycles, L1→R
    - 10 cycles, L2→L1→R
    - 100 cycles, main memory→ L2→L1→R
  - Add
    - 1 cycle, R → ALU
  - Store
    - 1 cycle, write operation in memory

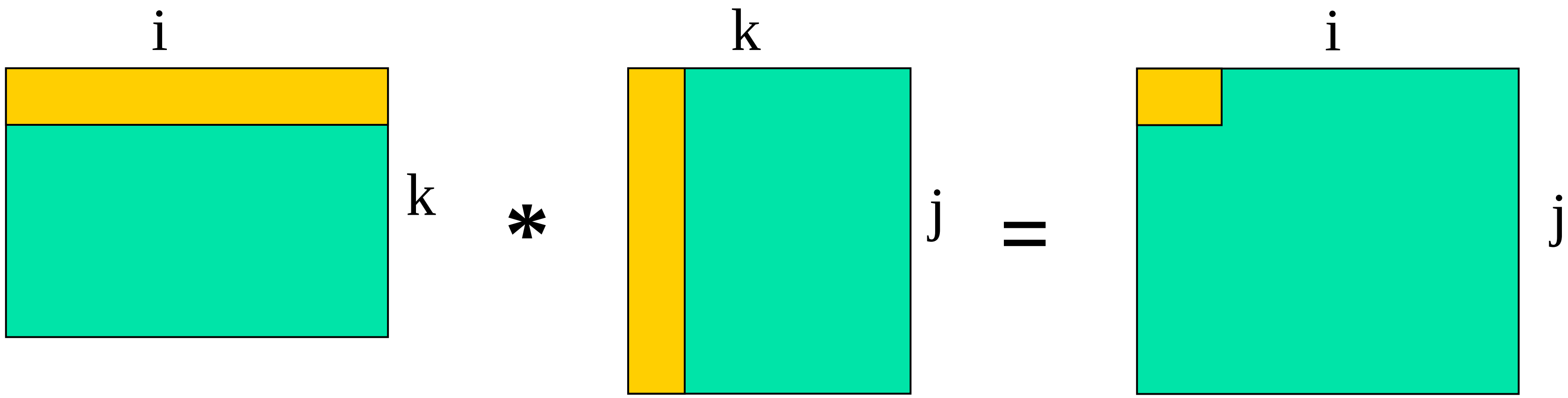
Effect of Cache Misses on Run Time

- Compiling "a = b + c"
  - load b; load c; add; store a
  - add, store
    - 1 cycle each
  - load
    - No cache miss → 2*2 cycles → total 4 cycles
    - Every cache miss → 100*2 cycles → total 202 cycles

- Run time depends on cache miss!

Matrix multiplication (1/5)

- Matrix multiplication



- Rows of Matrix are stored adjacently in memory

Matrix multiplication (2/5)

- Multiplication of two square matrices

$$c[i][j] = \sum_{k=1}^{n} a[i][k] * b[k][j]$$

$$(1 \le i \le m,\ 1 \le j \le p)$$

- Position of elements in the memory
  - Same row → adjacent
  - Same column → apart

Matrix multiplication (3/5)

```
public static void fastSquareMultiply(int [][]a, int [][] b,int [][] c, int n)
{  for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)     c[i][j] = 0;
     for (int i = 0; i < n; i++)
          for (int j = 0; j < n; j++)
               for (int k = 0; k < n; k++)   c[i][j] += a[i][k] * b[k][j];
}
```

What if we exchange j and k

- A(3,3) * B(3,2) = C(3,2)

$$a_{11}a_{12}a_{13} \quad b_{11}b_{12}$$

$$a_{21}\ a_{22}a_{23} \quad b_{21}b_{22}$$

$$a_{31}a_{32}a_{33} \quad b_{31}b_{32}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}$$

Matrix multiplication (4/5)

- For loop order
  - ijk order
    - Elements of a, c are accessed by row
    - Elements of b are accessed by column
    - Probability of cache miss
  - ikj order
    - All elements of a, b, c are accessed by row
    - It will take less time

Matrix multiplication (5/5)

- Run times for matrix multiplication

n	ijk order	ikj order
500	15.3	13.7
1000	127.9	110.5
2000	1059.1	886.5

- ikj order takes 10% less time

Observation

- Matrix multiplication
  - Knowledge about computer architecture
  - Memory access pattern of matrix multiplication
  - Simple idea about data positioning →
    a very fundamental data structure technique
  - Performance vs. Data structure

Summary

- When you try to market your code
  - Memory requirements is easy to figure out
  - Running time requires need of experiments

- In this chapter
  - How to perform such an experiment for run time
  - Factors affecting running time
    - The number and type of operations
    - The memory access pattern for the data & instructions in your program