COMPSCI 105

Principles of Computer Science

Performance Analysis

Contents

Introduction

Efficiency

Running time of different algorithms

Big-O

Examples

Choosing between algorithms

```
public long fibonacciA(int n){
      if (n<=2)
            return 1;
      else
            return fibonacciA(n-1) + fibonacciA(n-2);
}
```

```
public long fibonacciB(int n){
      if (n<=2)
            return 1;
      else{
            long[] result = new long[n+1];
            results[1] = 1;
            results[2] = 1;
            for(int i=3; i<=n; i++)
                  result[i] = result[i-1] + result[i-2];
            return result[n];
      }
}
```


Measuring the time required

How long does it take to download a file from the Internet?

Suppose:

- 2000 ms to set up an initial connection
- 0.5 ms to down each byte

Time taken for a file of size *n* bytes is:

- $T(n) = 2000 + 0.5n$

Large files

- *T(1000000) = 502000ms = 8 minutes + 22 seconds*
- *T(2000000) = 1002000 = 16 minutes + 42 seconds*

Estimating time taken

Looking at large files

- The connection time is insignificant
- The important factor is the size of the file

How long does it take to download a file?

- Depends on the size of the file
- Double the size means we (more of less) double the time
- Linear time

We are interested in the rate at which the time increases as the size of the problem (or data) increases.

How do we analyse running time?

We look at algorithms

- Not dependant on the hardware
- Not dependant on the programming language

Count the number of instructions executed

- Gives us a way to compare efficiency of different algorithms

Example: an algorithm that manipulates *n* strings

- Algorithm A requires time proportional to $3n^2$
- Algorithm B requires time proportional to 7n
- Which would you choose?

Estimate only

- Describe the way the time changes as the problem gets bigger

Example

Summing the first *n* integers

```
int sum = 0;
for (int i = 0; i < n; i++) {
        sum += n;
}
```

```
int sum = 0;
```

- Assignment is done only once

```
for (int i = 0; i < n; i++)
```

- One assignment to initialise i
- Conditional test performed n+1 times
- Increment performed n times

```
sum += n;
```

- Assignment is performed n times (inside the loop)

Calculation

```
int sum = 0;
for (int i = 0; i < n; i++) {
        sum += n;
}
```

Given input of size n, the number of operations:

- = 1 + 1 + (n-1) + n + n
- = 3n + 3

Big problem sizes, and general descriptions

- Ignore the smaller terms = 3n
- Ignore the constants = n

O(n)

- "order n"

Example

Counting the unique elements in an array

4	6	0	3	3	9	7	6	7	1
0	1	2	3	4	5	6	7	8	9

This array has 7 unique values:

- 0, 1, 3, 4, 6, 7 and 9

Algorithm

Count the last element in the array with a given value

For each element in the array

- increment the count if there are no later elements the same

```
int numUniques = 0;
boolean isLastCopy;

for (int i = 0; i < nums.length; i++) {
        isLastCopy = true;
        for (int j = i+1; j < nums.length; j++)
                if (nums[i] == nums[j])
                        isLastCopy = false;
        if (isLastCopy)
                numUniques++;
}
```


Analysis

Number of times each statement is executed

```
int numUniques = 0;
```

- executed only once

```
boolean isLastCopy;
```

- executed only once

```
for (int i = 0; i < nums.length; i++)
```

- initialisation executed only once
- conditional test executed n+1 times
- increment executed n times

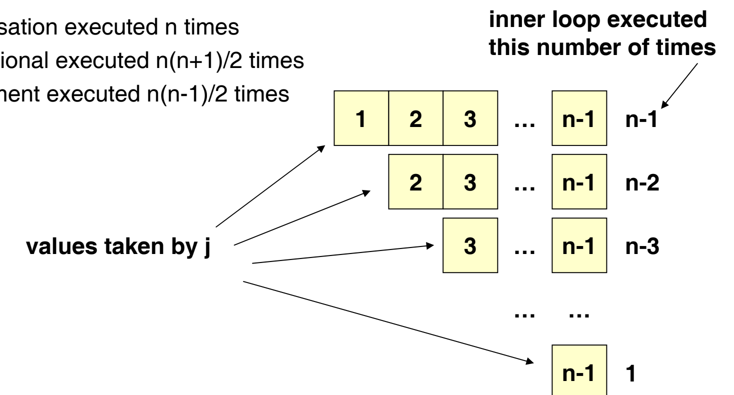
Analysis

```
        isLastCopy = true;
```

- executed n times

```
        for (int j = i+1; j < nums.length; j++)
```

- initialisation executed n times
- conditional executed n(n+1)/2 times
- increment executed n(n-1)/2 times



Analysis

```
if (nums[i] == nums[j])
```

- executed n(n-1)/2 times

```
isLastCopy = false;
```

- may be executed up to n(n-1)/2 times (worst case)

```
if (isLastCopy)
```

- executed n times

```
numUniques++;
```

- may be executed up to n times (worst case)

Analysis

In total: 1 + 1 + 1 + n+1 + n + n + n + n(n+1)/2 + n(n-1)/2 + n(n-1)/2 + n(n-1)/2 + n + n

$$= 2n^2 + 5n + 5$$

When n gets very large

- 5 doesn't matter
- 5n doesn't matter
- Ignore the constant multiplier 2
- = $O(n^2)$

Big-O (big-oh)

Rules

- Ignore low-order terms in the function
- Ignore any constants in the high-order terms

Examples

- *f(n) = n + log(n)*
- O(n)

- $f(n) = n^4 + 1000n^2 + 34.5n + 12$
- $O(n^4)$

- *f(n) = 3n + 1000*
- O(n)

Exercises

For each of the following, determine the big-oh value:

(a) f(n) = 45;

(b) $f(n) = 7n^4 + 20000$

(c) $f(n) = 3n^2 + n\log(n)$

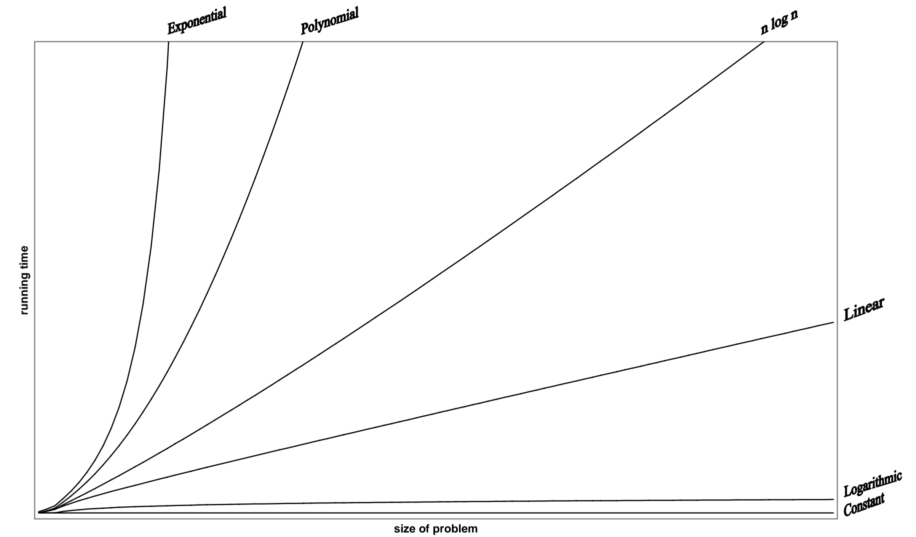
(d) f(n) = n + 34

(e) f(n) = nlog(n) + 34n + 6;

Common growth rates

Common name	Relationship between running time and problem size	Big-oh expression
constant	$time \propto k$	O(1)
logarithmic	$time \propto \log(n)$	O(log(*n*))
linear	$time \propto n$	O(*n*)
n log n	$time \propto n \log(n)$	O(*n* log(*n*))
polynomial	$time \propto n^k$	O(n^k)
exponential	$time \propto k^n$	O(k^n)

Graph of common running time



Example: O(1)

Get the first element in an array

```
public Object getFirstElement(Object[] o) {
       return o[0];
}
```

Number of instructions executed is always 1

- Size of the array does not matter

Example: O(log(n))

Binary search

```
private int bSearch(int[] array, int first, int last, int value)  {
   int index;
   if (first > last)
      index = -1;
   else {
      int mid = (first + last) / 2;
      if (value == array[mid])
         index = mid;
      else if (value < array[mid])
         index = bSearch(array, first, mid - 1, value);
      else
         index = bSearch(array, mid + 1, last, value);
   }
   return index;
}
```

Each time we halve the problem

- If n is 256, then we require a maximum of 8 calls
- log(n)

Example: O(n)

Finding if an element exists in an array

```
private boolean contains(int[] array, int value)  {
   boolean exists = false;

   for (int i = 0; i < array.length; i++) {
      if (array[i] == value) {
         exists = true;
      }
   }

   return exists;
}
```

We look at every element in the array

- Increasing the array increases the number of instructions
- Linear search

Example: O(n^2)

Printing out pairs of values

```
public void printPairs (int n) {
   for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++)
         System.out.println(i + "," + j);
   }
}
```

Problem size is n

- Running time is n * n
- If we double n, we quadruple the running time

Exercise

What is the big-O for each of the following algorithms:

- Determine if two Arrays are the same or not
- Print out a chess board pattern which is n squares wide and n squares high
- Find the maximum element in an array
- Generate the reverse of a String
- Generate all the permutations of a String
- Copy the contents of one array to another array
- Swap two elements in an array

Caution

Many factors to consider when choosing algorithm

- Simplicity
- Efficiency
- Memory requirements

Consider following running times:

- $T_1(n) = 100n$ O(n)
- $T_2(n) = n\log(n)$ O(nlog(n))

Which is better?

- Linear is better in theory, but in this case, only when log(n) >100
- When n > 100,000,000,000,000,000,000,000,000,000,000

Sometimes the constant factors really *do* matter :)