

Algorithms & Complexity - Introduction

Yanjun Ma
`yma@computing.dcu.ie`

CA313@Dublin City University (2009-2010)

October 22, 2009

Outline of the lectures

- Introduction to Complexity Theory
 - How to compute the formal complexity of a program?
 - Time complexity classes
 - The class P . Polynomial-time reducibility
- The class NP . Cook's theorem
- Heuristics for NP complete problems: Genetic algorithms, Tabu search, Beam search
- Heuristics for NP complete problems: Ant algorithms, Simulated annealing
- Space complexity classes
- Relations between time and space

Introduction to Complexity Theory

- Growth of Functions (the $O(f)$)
- Complexity Classes
- The class P (e.g. Closure problems) and Corresponding Turing Machines
- Boolean Satisfiability

Quantifiers

Two types of Quantifiers

- Universal quantifier: $\forall x$ “for all x ”
- Existential quantifier: $\exists x$ “there exists an x such that”

Scoping: “Every man loves a woman”

- $\forall x \exists y \text{ loves}(x, y)$: Becks loves Posh, Tony loves Cherie, Tom loves Katie...
- $\exists y \forall x \text{ loves}(x, y)$: Becks loves Posh, Tony loves Posh, Tom loves Posh...

Time complexity: an informal definition

Definition (Time complexity)

The *(time) complexity* of a program (for a given input) is the number of elementary instructions that this program executes. **This number is computed with respect to the size n of the input data.**

Note

We thus make the assumption that each elementary instruction takes the same amount of time (because it is probably one processor instruction, which is true for example for addition and multiplication between integers).

Space complexity: an informal definition

Definition (Space complexity)

The (*space*) *complexity* of a program (for a given input) is the number of elementary objects that this program needs to store during its execution. **This number is computed with respect to the size n of the input data.**

Note

We thus make the assumption that each elementary object needs the same amount of space (which is almost true for the basic types: one machine word).

Complexity: why bother?

Estimation/Prediction

When you write/run a program, you need to be able to predict its needs, its requirements.

Usual requirements

- execution time
- memory space

Quantities to estimate

- execution time \Rightarrow time complexity
- memory space \Rightarrow space complexity

Complexity: why bother?

It is pointless to run a program that requires:

- 6TeraB of RAM on a desktop machine;
- 10,000 years to run...

You do not want to wait for an hour:

- for the result of your query on Google;
- when you are checking your bank account online;
- when you are opening a picture file on Photoshop;
- etc.

⇒ It is important to write efficient algorithms!

Complexity: why bother?

It is *very* important that you know what you can and cannot do to improve (i.e. optimize) your algorithms:

- It is not wise to run an algorithm with $O(n^2)$ complexity if there is an algorithm with $O(\ln(n))$ doing the same job...
- Do not try to optimize an algorithm for a problem if we *formally* know there is no efficient (i.e. polynomial) algorithm...
- \Rightarrow You will need to know how to compute and estimate the complexity of algorithms if you want to be able to say that.

“Speed” of Programming Languages

You will read things like “C is faster than Java”, “Java is faster than Python”, etc.

More important than the programming language are:

- (i) the formal complexity of the algorithm;
- (ii) the programmer (i.e. simplicity, quality of the program, etc.)

Note

An algorithm with complexity $O(n \times \ln(n))$ in Python will probably be faster than an algorithm with complexity $O(n^2)$ in C...

A formal definition

Definition (Time complexity)

We will note $T(A, n)$ the number of elementary instructions that an algorithm A executes when the size of the input data is n , or $T(A)$ when unambiguous.

Example

```
// note: x is an unsorted array
int findMin(int[] x) {
    int k = 0; int n = x.length;
    for (int i = 1; i < n; i++) {
        if (x[i] < x[k]) {
            k = i;
        }
    }
    return k;
}
```

Computing the complexity

Sequence of instructions

The number of elementary instructions executed in a sequence of instructions is the sum of the number of elementary instructions executed in each instruction:

$$T(A_1; A_2; A_3) = T(A_1) + T(A_2) + T(A_3)$$

Loop of instructions

The number of elementary instructions executed in a loop is (at most) the number of iterations multiplied by the number of elementary instructions executed in the body of the loop:

$$T(\text{for}(\text{int } i = 1; i < n; i++)\{B\}) = n \times T(B)$$

Example

```
// note: x is an unsorted array
int findMin(int[] x) {
    int k = 0; int n = x.length;
    for (int i = 1; i < n; i++) {
        if (x[i] < x[k]) {
            k = i;
        }
    }
    return k;
}
```

$$T(\text{findMin}, n) = 2 + T(\text{findMinLoop}) \times n + 1$$

$$T(\text{findMinLoop}) = 2$$

$$T(\text{findMin}, n) = 3 + 2 \times n$$

The O

In order to speak more easily about the complexities algorithms, they are usually grouped into “families” (logarithmic, polynomial, etc.).

We will write $g(n) \in O(f(n))$, where g is usually a combination of polynomial, exponential and logarithmic functions. We write:

$$g \in O(f)$$

if and only if:

$$\exists c > 0, \exists d > 0, \forall n \in N, g(n) \leq c \times f(n) + d$$

The O - Some simple rules

$\forall f, g$, if $f \in O(g)$ and $f' \in O(g')$, then:

$$f + f' \in O(g + g')$$

$$f \times f' \in O(g \times g')$$

Consider the polynomial $f(n) = 31n^2 + 17n + 3$, and prove that $f(n) \in O(n^2)$

The O - Examples

$$f(n) = 2n + 3; f(n) \in O(n)$$

$$f(n) = 6n^2 + 235; f(n) \in O(n^2)$$

$$f(n) = 6n + 567\ln(n); f(n) \in O(n)$$

$$f(n) = 6n \times \ln(n); f(n) \in O(n \times \ln(n))$$

$$f(n) = 2\exp(n) + n\ln(n) + n^456; f(n) \in O(\exp(n))$$

Average vs. worst-case complexity

Definition (Worst-case complexity)

The worst-case complexity is the complexity of an algorithm when the input is the worst possible with respect to complexity.

Definition (Average complexity)

The average complexity is the complexity of an algorithm that is averaged over all the possible inputs (assuming a uniform distribution over the inputs).

Average vs. worst-case complexity

We assume that the complexity of the algorithm is $T(i)$ for an input i . The set of possible inputs of size n is denoted I_n .

Worst-case complexity

$$T_{wc}(n) = \max_{i \in I_n} T(i).$$

Average complexity

$$T_{av}(n) = \frac{1}{|I_n|} \times \sum_{i \in I_n} T(i),$$

where $|I_n|$ denotes the *size* of I_n , i.e. the number of possible inputs of size n .

A hierarchy of common formal complexities

- constant: $O(1)$
- logarithmic: $O(\ln(n))$
- linear: $O(n)$
- linearithmic: $O(n \times \ln(n))$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^p), p \in \mathbb{N}$
- exponential: $O(\exp(n))$

A hierarchy of common formal complexities: some examples

- constant: $O(1)$. Print the first element in a list.
- logarithmic: $O(\log_2(n))$. Search a number in a sorted list of size n (**Binary Search**).
- linear: $O(n)$. Search a number in an unsorted list of size n .
- linearithmic: $O(n \times \log_2(n))$. Sorting a list of size n (**Quick Sort Algorithm**).
- quadratic: $O(n^2)$. Multiplication of matrices (lower bound).
- cubic: $O(n^3)$. Parsing.
- polynomial: $O(n^p), p \in \mathbb{N}$. Shortest path in a weighted graph.
- exponential: $O(\exp(n))$. Travelling Salesman Problem (TSP).

Example: Binary Search

```
// note: x is a sorted array
int BinarySearch(int [] x, int y) {
    int high = x.length; int low = 0, mid;
    while (low <= high) {
        mid=(low+high)/2;
        // shrink the search interval on the right
        if(x[mid] < y) low = mid+1;
        // shrink the search interval on the left
        else if(x[mid] > y) high = mid-1;
        else return mid;
    }
}
```

Example: Binary Search (cont.)

```
// note: x is a sorted array
int BinarySearch(int [] x, int y, int low, int high) {
    int mid = -1;
    if (low <= high) {
        mid=(low+high)/2;
        // shrink the search interval on the right
        if(x[mid] < y)
            mid = BinarySearch(x, y, mid+1, high);
        // shrink the search interval on the left
        else if(x[mid] > y)
            mid = BinarySearch(x, y, low, mid-1);
    }
    else return mid;
}
```

Binary Search: Worst-case Complexity

Constructing a Binary Tree with n nodes

- Empty tree if $n = 0$
- If $n \neq 0$:
 - root node = $(n - 1)/2$
 - left subtree corresponds to $x[0] \dots x[\frac{n-1}{2} - 1]$
 - right subtree corresponds to $x[\frac{n-1}{2} + 1] \dots x[n]$

Worst-case Complexity

- The number of comparisons in BinarySearch \leq the depth of the tree + 1
- The depth of the binary tree is $\log_2(n + 1) - 1$
- The worst case complexity $\log_2(n + 1)$

Binary Search: Average Complexity

Constructing a Complete Binary Tree with $n = 2^h - 1$ nodes

- The depth is $\log_2(n + 1) - 1 = h - 1$
- depth 0: 1 comparison; depth 1: 2 comparisons; \dots ; depth i : $i + 1$ comparisons...
- Assuming the probability of searching each node is equal, i.e. $p_i = 1/n$, the average number of comparisons are:

Binary Search: Average Complexity (cont.)

$$\begin{aligned} AC &= \sum_{i=0}^{n-1} p_i (1 \times 1 + 2 \times 2^1 + 3 \times 2^2 + \cdots + (h-1) \times 2^{h-2} + h \times 2^{h-1}) \\ &= \frac{1}{n} (1 \times 1 + 2 \times 2^1 + 3 \times 2^2 + \cdots + (h-1) \times 2^{h-2} + h \times 2^{h-1}) \\ &= \frac{1}{n} ((h-1) \times 2^h + 1) \text{ proof by induction} \\ &= \frac{1}{n} (h \times 2^h - 2^h + 1) \quad h = \log_2(n+1) \\ &= \frac{1}{n} ((n+1) \log_2(n+1) - n) \\ &= \frac{n+1}{n} \log_2(n+1) \\ &\approx \log_2 n + 1 - 1 \end{aligned}$$

A hierarchy of common formal complexities: remarks

From a practical point of view

Polynomial algorithms are (usually) considered efficient. Everything slower than that is (usually) considered inefficient.

However...

- Linear and quadratic algorithms \Rightarrow OK.
- Cubic algorithms \Rightarrow OK in some cases.
- Everything above \Rightarrow NOT OK.

Notes on formal complexity

In practice:

- Complexity ignores constant factors. A problem whose complexity is $10^{1000}n$ is polynomial, but is completely impractical in practice. A problem whose complexity is $10^{-10000}2^n$ is exponential, but can be practical for not too big values of n .
- Complexity ignores the size of the exponents in polynomials. A problem with time n^{1000} is polynomial, yet impractical...

Complexity of algorithms vs. complexity of problems

Definition (Decision Problems)

A (binary) *decision problem* is a question in some formal system with a yes-or-no answer.

The problem itself is distinct from the methods used to solve it, called decision procedures or algorithms. A decision problem which can be solved by some algorithm is called decidable.

Example

Given two (natural) numbers x and y , does x divide y ?

This is a yes-or-no question, and its answer depends on the values of x and y . An algorithm for this decision problem would tell how to determine whether x divides y , given x and y .

Complexity of algorithms vs. complexity of problems

Definition

Complexity of decision problems The complexity of a (decidable) decision problem is the one of the *most efficient algorithm* for that decision problem.

⇒ It is thus more difficult to compute the complexity of a problem...

Decision problems. Examples

Example

- Is x prime?
- Is this list empty?
- Is this matrix symmetric?
- etc.

Some definitions

Definition (Algorithm)

An algorithm is a sequence of (elementary) instructions that makes possible to go from a well defined state to another well defined state.

Definition (Algorithm (formal))

An algorithm is a word (on some alphabet) which can be read and executed by a Turing Machine.