Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

Algorithm Analysis.

Authors: Juan Perea
Isabel Segura Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Algorithm Analysis

Most of the information has been sourced from the books [1, 2].

Introduction

In mathematics and computer science, an **algorithm** is an effective method for solving a problem, expressed as a pre-written set of well-defined, ordered and finite instructions and rules. We can use a more-or-less formal language, such as natural language, pseudocode, or a programming language like Java, to express these algorithms.

For example, if we want to calculate the summation of the first N integers, we start with a result value of 0, then loop from 1 to N, and for each value i, we add i to the result. In Java, we write this as:

```
static long func1(int n) {
        long sum=0;
        for (int i=1; i¡=n; ++i) sum += i;
        return sum;
}
```

We are used to deal with algorithms in our life: if we want *to look for a word in a dictionary*, we usually open the dictionary by the middle, then search again in the first or the second half depending on the alphabetical order, and so on, or we can look at every page in order until we find the page that contains our word, which is obviously less efficient.

In Mathematics, we can use (at least) two methods for *calculating the list of prime numbers (any number that can only be divided by 1 and itself) that are less than N*: we can iterate over odd numbers (and, for each candidate, test it against all prime numbers less tham or equal its square root), or we can use the more complex but more efficient Sieve of Eratosthenes algorithm (see the Wikipedia article[1]). The following list of steps describes the Eratosthenes' algorithm:

[1] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

1. Create an array of integers containing the values from 2 to n: (2, 3, 4, ..., n).
2. Set a variable x to 2 (which is the first firme number)
3. Traverse the array from x to N, setting all multiples of x to 0.
4. The following number after x and greater than 0, is the next prime. Then, set the variable x to this number.
5. Repeat the 3-4 steps until x^2 is greater than n.

At the end, the non-cero elements in the array are the prime numbers from 0 to n.

We will use these examples in this chapter to illustrate the concepts shown.

Algorithm analysis

Most algorithms are designed to deal with problems that have variable sized inputs: this input size can determine the amount of resources the algorithm consumes until it completes. Typically, an algorithm can consume two types of resources for its completion: *time*, and storage (*memory* or disk space). Algorithm analysis tries to determine the amount of resources needed to execute it, in the form of a function related to the input size of the algorithm.

Taking a look at the first example mentioned in the introduction, analyzing the summation algorithm would result in counting the number of times each simple instruction is executed:

```
static long func1(int n) {
        long sum=0; → 1
        for (int i=1; i≤n; i++) → i=1:1 ; i≤n:n+1 ; i++:n
                sum += i; → n
        return sum; → 1
}
```

Thus, given n, this algorithm will execute a total of

$$1 + 1 + (n + 1) + n + n + 1 = 3n + 4 \tag{1}$$

single instructions; if c is the time it takes to execute the longest instruction, then the running time of or algorithm will be, at most, c(3n+4). Let us suppose c=1 millisecond; then the running time will be (3n+4) milliseconds.

Similarly, we can easily see that searching time in a dictionary is always limited, whichever the chosen algorithm, by a multiple of the dictionary size. Being N the number of pages in the dictionary, if we iterate through all the pages, it is obvious that we will need to look at the whole set of N pages if the searched word is in the last one. But if we divide the dictionary in halfs, we will

only need to open the dictionary a few times; i n fact, doubling the dictionary size will result in one page looked more. We can then infer that running time will be limited by a multiple of log2N. This is not relevant if a dictionary had a few pages, but becomes an important issue for huge dictionaries.

When we estimate running time, we have to be aware that we can be lucky and find the searched word in the page we open the dictionary by at the first try (best case), but we can also be unlucky and not find the word until the last page (worst case), so maybe we want to study an average case.

When calculating prime numbers, there are no best and worst case scenarios; running time will only depend on N. But, in this case, running time won't be limited by a multiple of N, as the bigger the number we are checking, the longer it will take to check if it's prime (checking if an odd number i is prime might need $\sqrt{i}/2$ iterations, so running time for each i will be limited by a multiple of $\sqrt{i}$). Therefore, the overall running time will be limited by a multiple of $\sqrt{1}+\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{7}+\sqrt{11}+\ldots\sqrt{N}$, which is limited by a multiple of NN.

Performance versus Memory

Though memory space (or disk space, if we are talking about, for example, a database) is an almost unlimited resource in modern computing, it can still be an issue for some algorithms. For example, an algorithm for a computer that plays chess might need to store all the possible moves it has studied.

Sometimes, the same problem can be solved by an algorithm that penalizes performance and an algorithm that penalizes memory space. For example, the Sieve of Eratosthenes algorithm achieves a better performance, but it needs to keep an array of booleans, which has a size that is multiple of the problem size, while the other algorithm will need no extra storage.

Another way to increment performance by using more memory is to change the way data are stored. For example, back to our dictionary, we redistribute the words and use a function to calculate the page number in which a word must be inserted or searched for, (this is called a hash function). This way, searching time will be constant, whichever the dictionary size, but memory space (the number of pages in the dictionary) will grow considerably.

As an example, we can use the following function: Let us assign a value to each letter (A=0, B=1, ..., Z=25), and let us use the first three letters of each word (L1, L2 and L3). Then, the page number will be:

$$PageNumber = 26^2 * L1 + 26 * L2 + L3 \tag{2}$$

So if we want to store (or look for) 'ABBEY', we will find it in:

$$PageNumber = 26^2 *'A' + 26 *'B' +'B' = 26^2 * 0 + 26 * 1 + 1 = 0 + 26 + 1 = 27 \tag{3}$$

This algorithm is much faster, but the storage penalty is really important: such dictionary would have $263 = 17576$ pages, most of which will be empty, and perhaps other pages will not have enough space to hold all the words they should.

Asymptotic analysis

When analyzing algorithms, we use asymptotic notations to show in a mathematical way how resource consumption scales with input size. The aim of asymptotic analysis is to focus on the shape of the function curve, and to define functions as simple as possible that limit either as an upper bound (big O), as a lower bound (big Omega), and as both (big Theta). In this chapter, we will focus on upper bound limits, and just mention lower bound limits.

Upper bound limit big O

We have seen in the previous examples how algorithm efficiency for big input sizes can be measured in terms of a limiting simple function. Such function, that describes the growing behaviour of an algorithm for input sizes tending to infinity is known as asymptotic notation or 'big O' notation.

The mathematical definition of big o is a s follows:

- Consider a function f(N) which is non-negative for all integers N¿=0 . "f(N) is big o g(N)", or "f(N) O(g(N))", of "f(N) has order of g(N) complexity", if there exists an integer n0, and a constant c¿0, such that, for all integers n¿=n0, then f(n)¡=cg(n). In a more comprehensive way, this definition says that for values of n greater than a given value n0, g(N) is proportional to f(N), or worse. In other words, it says that g(N) grows at least as fast as f(N) for big values of N.

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.

[2] M. Weiss, *Data structures and problem solving using Java*. Addison Wesley Publishing Company, 2002.