CMSC 351 - Introduction to Algorithms
Spring 2012
Lecture 2

Instructor: MohammadTaghi Hajiaghayi
Scribe: Rajesh Chitnis

1 Introduction

In this lecture we look at Analysis of Algorithms and the O notation.

2 Analysis of Algorithms

We care about the efficiency of algorithms. A good measure is running time which can predict the behavior of an algorithm without implementing it on a specific computer. Again we usually consider running times of the small parts of programs which are the **heart** of the programs and not everything (like number of multiplications, additions, etc). We define certain parameters and certain measures that are the most important for the analysis (an approximation of the real world).

The main feature (as you have seen in previous classes) is to ignore the constant factors and concentrate on the behavior of the algorithm when the **size of the input** goes to infinity. The size is usually defined as a measure of amount of space required to store the input and usually will be denoted by n.

Given a problem and a definition of size, we want to find an expression that gives the running time of the algorithm relative to the size usually in the **worst case**. The best case is usually ruled out since it is not representative. The average case might be a good choice but is usually very hard to measure (even the definition of average is not clear when we have different parameters, also mathematically difficult to analyze). Also it is hard to predict what will happen in practice (say in a airplane) and thus considering worst-case scenario is safer.

3 The Big-O Notation

Definition 1 *A function* $g(n)$ *is* $O(f(n))$ *for another function* $f(n)$*, if there exist constants* c *and* N *such that for all* $n \geq N$*, we have* $g(n) \leq c.f(n)$

In other words, for large enough n, the function $g(n)$ is no more than a constant times the function $f(n)$. The O-notation is like $\leq$ only from the above, though we usually try to find the tightest bound.

Example: $5n^2 + 15 = O(n^2)$ since $5n^2 + 15 \leq 16n^2$ for $n \geq 4$. Also it is $O(n^3)$ since $5n^2 + 15 \leq n^3$ for all $n \geq 6$.

We always write $O(n)$ instead of $O(5n+4)$. We also write $O(\log n)$ without specifying the base of the logarithm. So $O(1)$ means constant, $O(n)$ is linear, $O(n^2)$ is quadratic and so on. Though in general showing $g(n) = O(f(n))$ is not easy, it will be easy for almost all algorithms in this class.

4 Polynomial vs Exponential

Theorem 1 $n^c = O(a^n)$ *for all constants* $c > 0$ *and* $a > 1$.

Note that $(\log_a m)^c = O(a^{\log_a m}) = O(m)$. We call $(\log_a m)^c$ as "polylog" in m. So we prefer polynomials over exponentials (like 2^n) and polylog over polynomials.

5 Rules

We can add and multiply using the following rules (but we cannot subtract or divide). Proofs follow from the definition and are given in the book [1].

1. If $f(n) = O(s(n))$ then $c.f(n) = O(s(n))$ for any constant $c > 0$.
2. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n) + g(n) = O(s(n) + r(n))$.
3. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n).g(n) = O(s(n).r(n))$.

6 Other Notation

We often try to use (tight) upper bounds on running times of algorithms which means there is an algorithm with at most this running time. However often we need to say that there is no algorithm that can achieve a better running time. Of course this is much harder since we should model **every** algorithm. The notation for lower bounds is Ω.

Definition 2 *A function* $T(n)$ *is* $\Omega(g(n))$ *for another function* $g(n)$*, if there exist constants* c *and* N *such that for all* $n \geq N$*, we have* $T(n) \geq c.g(n)$.

Examples are $n^2 = \Omega(n^2 - 100)$ and $n = \Omega(n^{0.9})$. The O-notation corresponds to $\leq$ and the Ω-notation corresponds to $\geq$. What about $=$? We have the following notation:

Definition 3 *A function* $f(n)$ *is* $\theta(g(n))$ *for another function* $f(n)$*, if* $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

Example is $5n \log_2 n - 10 = \theta(n \log_2 n)$. The constants in O and Ω in the above definition need not be the same. Also note that if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$.

What about strict inequalities, i.e., $<$ and $>$ instead of $\leq$ and $\geq$?

Definition 4 *We say* $f(n) = o(g(n))$*, i.e.,* $f(n)$ *is little-oh of* $g(n)$*, if* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*. Similarly we say that* $f(n) = \omega(g(n))$ *if* $g(n) = o(f(n))$*.*

For example, $\frac{n}{\log n} = o(n)$ but $\frac{n}{10} \neq o(n)$.

Theorem 2 $n^c = o(a^n)$ *for all constants* $c > 0$ *and* $a > 1$*.*

Corollary 1 $(\log_a n)^c = o(n)$

Note that we usually ignore the constants in O-notation but sometimes in practice even the constant matters, e.g., in Google (see Chapter 3 of the book [1] and Wikipedia article on "Big O notation"). Sometimes we use $\tilde{O}$ to hide logarithms, e.g., $O(n \log n) = \tilde{O}(n)$.

7 Time and Space Complexity

We analyze an algorithm by counting the number of major steps the algorithm performs. For example, in sorting we count the number of comparisons. In the celebrity problem, we counted the number of questions asked. In a sense we say since we ignore the constant factors, all other operations are only a constant factor of the number of major steps. In this case we say **time complexity** or the **running time** of the algorithm is in $O(f(n))$.

The **space complexity** of an algorithms indicates the amount of temporary storage (not including the input and output) for running the algorithm. An $O(n)$ space algorithm requires a constant amount of memory per input primitive. An $O(1)$ space algorithm needs a constant amount of memory irrespective of the size of the input. Given the current amount of storage in disks that we have, we focus mainly on running time and not space complexity (although it is important for Google).

8 Some Mathematical techniques for Computing Running Times

Now we consider some mathematical techniques such as recursion, summation, etc. for computing the running time of algorithms. The proofs are often by induction.

Theorem 3 $S_1(n) = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$.

Proof: We have seen the proof of this theorem in Lecture 1. ∎

Theorem 4 $S_2(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

Proof: Proof is by induction and is given in the book [1]. There is also another way to prove this theorem without using induction. ■

Theorem 5 $F(n) = \sum_{i=0}^{n} 2^i = 1 + 2 + \ldots + 2^n = 2^{n+1} - 1$.

Proof: There is a simple proof by induction. Alternatively we can try to compare $F(n)$ with another expression involving $F(n)$. Note that $2F(n) = 2 + 4 + 8 + \ldots + 2^{n+1}$. Take the difference between $2F(n)$ and $F(n)$ to obtain $F(n) = 2^{n+1} - 1$. ■

Theorem 6 $G(n) = \sum_{i=1}^{n}(i \times 2^i) = (1 \times 2^1) + (2 \times 2^2) + \ldots + (n \times 2^n) = (n-1)2^{n+1} + 2$.

Proof: There is a simple proof by induction. Alternatively we can try to compare $G(n)$ with another expression involving $G(n)$. Note that $2G(n) = (1\times 2^2)+(2\times 2^3)+(3\times 2^4)+\ldots+(n\times 2^{n+1})$. Take the difference between $2G(n)$ and $G(n)$ to obtain $G(n) = n2^{n+1} - \left(2^1+2^2+\ldots+2^n\right) = n2^{n+1} - (F(n)-1) = n2^{n+1} - (2^{n+1}-2) = (n-1)2^{n+1} + 2$. ■

9 Further Reading

It is also good to know the following facts given on pages 53-54 of the book [1]. For completeness we give them below:

1. Arithmetic Series:
   - $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$
   - If $a_n = a_{n-1} + c$ for some constant c, then $a_1 + a_2 + \ldots + a_n = \frac{n(a_1+a_n)}{2}$
2. Geometric Series:
   - $1 + 2 + \ldots + 2^n = 2^{n+1} - 1$
   - If $a_n = c.a_{n-1}$ for some constant $c \neq 1$, then $a_1 + a_2 + \ldots + a_n = a_1\frac{c^n-1}{c-1}$
   - If $a_n = c.a_{n-1}$ for some constant $1 > c > 0$, then $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c}$
3. Sum of Squares: $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$
4. Harmonic Series: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \ln(n) + \gamma + O(\frac{1}{n})$ where $\gamma = 0.577\ldots$ is the Euler's constant.
5. Logarithm Formulas:
   - $\log_b a = \frac{1}{\log_a b}$

- $\log_a x = \frac{\log_b x}{\log_b a}$
- $b^{\log_b x} = x$
- $b^{\log_a x} = x^{\log_a b}$

6. Sum of Logarithms: $\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 = \theta(n \log n)$
7. Summation by Integral: For a monotone increasing continuous function f, we have $\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)\,dx$
8. Stirling's Approximation: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O(\frac{1}{n})\right)$. Thus $\log_2 n! = \theta(n \log n)$

References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*