Data Structures
Spring 2009

Kishore Kothapalli

Chapter 2

Asymptotic Analysis

2.1 A Gentle Introduction

Recall from the previous chapter that efficiency of the operations performed is an important concern in the field of computing. It is also important to know what are the parameters that one has to consider in evaluating the efficiency of the proposed operations. To use standard notions, from now on, we use the term “algorithm” to refer the method of performing operations. The word algorithm is attributed to the Arabian mathematican al-Khowrazimi, because of his fundamental contributions.

The experiments of the past week would lead us to beleive that the time taken by an algorithm is the parameter of interest in this direction. While it is true that the time taken is of good relevance, other parameters such as space, cost, are also applicable. In the rest of this chapter, we will however focus on the time aspect. It can be noted that the same notions extend to the case of the space consumption.

2.2 Measuring the Time Taken

Notice that measuring the time taken by an algorithm is a very difficuly task. Apart from the programming challenges, one finds that the actual time is subject to variations due to the input used, machine used, the system details such as the operating system, compiler, etc., occassionally the program, among others. Hence, using absolute times is not as helpful. For instance, if we report that binary search takes 2 micro second on an input array of size 1 M entries, it does not indicate what will be time on a 2 M sized array, will the time be always 2 micro second for any input of 1 M, what if the machine configuration is changed to something different, and such. Therefore, one uses a qualitative metric in analysis of algorithms.

The qualitative metric used is developed so that the time taken is measured on an abstract machine called the Random Access Machine. The random access machine model is shown in Figure **??**. According to this model, there is a processor with a limited amount of read/write memory cells and a limited number of registers. The (abstract) machine has a program counter (PC) and supports programming constructs such as recursion, looping, jumping, and branching. Moreover, the machine has a standard instruction set that includes:

- Arithmetic Operations: Add, subtract, multiply, divide
- Logical Operations: AND, OR, EX-OR, NOT, NAND, NOR
- Conditional operations: $=, >, <, \geq, \leq$
- Shift Operations: left shift and right shift by a constant positions,

- Boolean Operations:
- Memory Access Operations: Load/Store.

It is important to note that while in principle each of these operations can take varying number of machine clock cycles to be executed, we will assume that each of the above takes precisely one unit of time to execute. This simplification is justified by the fact that the unit of time may be taken as the maximum time required by any of these operations. For this reason, this model is sometimes called as the unit cost model also.

Further, the time taken is measured as a function of the size of the input. This lets us interpolate the time taken for an input of a given size. For instance, we say that algorithm A has a runtime of $2n^2 \log n + 1.5n$, where n is the size of the input. The size of the input is also measured as the number of (binary) bits required to represent the input. If the input TODO.

The question now we have to address is how to measure the time taken as a function of the input. This can be done for example by counting the number of operations involved in the algorithm. The example below illustrates this point.

```
Algorithm SumIntegers($A$)
    1. // $A$ is an array of $n$ integers. Find their sum.
    2. int $i, sum = 0$;
    3. for $i = 1$ to $n$ do
    4.   sum = sum + $A[i]$;
    5. end-for
End Algorithm.
```

In the above algorithm, step 1 is a comment and hence consumes no operations. Step 2 declares two integers i ans sum. Each of these shall be taken to consume one unit of time. Step 3 is a loop construct for n iterations. Here, we assume that it takes 2 unit operations for perform the increment of i and the comparison of i with n to decide the next instruction. There is of course one unit operation, independent of the number of loop iterations, to initialize i. Step 4 takes one unit operation per loop iteration. Step 5 is again taken to be consuming no operations. So, the total time taken is $0 + 2 + (1 + n \cdot (2 + 1)) = 3n + 3$.

Another example is given below. TODO

Recursive functions

We thus arrive at the following guidelines in measuring the time taken by a program on the RAM.

- Simple Statement : Each simple statement that does a single unit time operation takes a unit amount of time.
- Conditional statement: In a conditional statement of the form `if` (condition) `then` (statement1) `else` statement 2, the time taken will be the time for evaluating the condition plus the maximum time between statement1 and statement2.
- Loop statement : In a loop statement of the form `for` (loop init., condition, increment) statement; the time taken shall be the product of the number of iterations and the time taken by the statement plus the time for loop condition and increment evaluation. In the case of a nested loop, there is a nested product. Similar rules apply for other loop constructs such as `while`.

2.3 Towards Asymptotic Analysis

Consider the two examples given earlier, it feels that it is a bit of an overkill to count each and every operation. Morerover, since the actual time is machine dependent, one is often intersted in the growth characteristics of the runtime. To this end, we introduce a technique called asymptotic analysis. The central theme of asymptotic analysis is to capture the relative growth rates of functions. We introduce the following definitions.

Definition 2.3.1 (Big-Oh) *Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$, we say that $f(n) \in O(g(n))$ if there exists positive constants c, n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.*

The point to note from the above definition is that we are considering the growth of $f(n)$ with respect to $c \cdot g(n)$ for values of n beyond a fixed value n_0. This allows us to write that $f(n) = 100n^2 + 1000n \in O(g(n))$ for $g(n) = 2n + 3$. While in principle $f(n)$ is larger than $g(n)$ for small values of n, the idea is that for large values of n, n^2 grows much faster than $1000n$. So that the growth rate is of the order of n^2. Some more examples are given below.

- $\log^k n \in O(n)$ for any constant $k > 0$.
- If $f(n)$ is a polynomial of degree k, then $f(n) \in O(n^k)$.
- $\log(n^k) \in O(\log n)$ for any constant k.
- If $f(n)$ is a constant independent of n, then $f(n) \in O(1)$.

To simplify matters, we also write that $f(n) = O(g(n)$ instead of $f(n) \in O(g(n))$. The $O()$ notation can be also be viewed as specifying rules to write $f(n) < g(n)$.

We now introduce another defnition.

Definition 2.3.2 (Big-Omega) *Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$, we say that $f(n) \in \Omega(g(n))$ if there exists positive constants c, n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.*

The Ω notation conveys that $f()$ grows at a faster rate than $g(n)$ for large values of n. For example, if $f(n) = n^2$ and $g(n) = 1000n + 500$, then we can say that $f(n) \geq g(n)$ for $n \geq 2000$. So, $f(n) \in \Omega(g(n))$. As is the case with the $O()$ notation, here too one is interested in establishing that for large values of n, the growth rate of $f(n)$ exceedes that of the growth rate of $g(n)$.

Some examples are given below.

TODO

To simplify matters, we also write that $f(n) = \Omega(g(n)$ instead of $f(n) \in \Omega(g(n))$.

We introduce the following definition to capture near equal growth rate of functions.

Definition 2.3.3 (Big-Theta) *Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$, we say that $f(n) \in \Omega(g(n))$ if and only if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$.*

if $f(n) \in \Theta(g(n)$, by definition we can say that f and g grow at the same rate. To simplify matters, we also write that $f(n) = \Theta(g(n)$ instead of $f(n) \in \Theta(g(n))$.

Recall the law of trichotomy with repsect to real numbers which states that for any two real numbers a and b, one of the following hold: $a < b$, $a = b$, or $a > b$. It is however not the case that with respect to functions, the definitions for O, Θ, Ω induce a law of trichotomy of relative growth rates of functions.

2.4 The Best, Average, and the Worst Case Analysis

In using the model so far, we have not considered the possibility that the runtime of the algorithm may actually depend on the nature of the input apart from the size of the input. Even simple algorithms such as binary search can exhibit different runtimes depending on the key that is being searched. For instance, we may find that the key is present in the (sorted) array after one comparison, or we may declare the same after 10 comparisons. To handle this situation, we introduce a concept called the best, average, and the worst case analysis.

The best case runtime of an algorithm refers to the runtime of the algorithm in the best possible scenario with respect to the input. Naturally, this depends on the algorithm. Consider the example of searching a key in an array that is not sorted. The following algorithm, called linear search, can be used.

```
Algorithm LinearSearch($(A, x)$)
begin
    int i;
    for i=1 to $n$ do
        if $A[i] == x$ then return found;
    end-for
return not-found
end
```

The best case scenario for this algorithm is when $x = A[1]$. In this case, the algorithm returns after one iteration of the loop. The case of binary search is similar. If the key is found after one comparison, then the algorithm can return after the first iteration.

It may be tempting to conclude that the best case runtime of *any* algorithm is $O(1)$. However it is note the case as can be seen by the example of insertion sort or merge sort. In the case of merge sort, no matter what the input is, even if it is completely sorted, the algorithm still takes $O(n \log n)$ time to run.

The term worst case analysis refers to the maximum time taken by an algorithm for any input. In analyzing the worst case time, one considers that the input will force the algorithm to maximize its runtime. The exact way this can happen shall depend on the algorithm. In the case of algorithm LinearSearch given above, if the key being searched is either $A[n]$ or is not present in $A[]$, then the algorithm makes n comparisons. This can be taken as the worst case runtime of Algorithm LinearSearch.

However, the worst case analysis may be applicable only for a small number of inputs. In this situation, it may turn out that the worst case time is a huge overestimate. The recourse is to consider the average runtime of the algorithm. The average runtime of an algorithm may be defined as the average of the time taken by the algorithm over all inputs. In this definition, it is assumed that all inputs are equally likely. Notice that this is a big assumption in itself and is hard to justify. Even with this assumption, it is extremely complicated to estimtate the average runtime of an algorithm in many cases.

As an example, let us consider the average runtime of the Algorithm LinearSearch given earlier. Notice that if the key that is being searched in not present in the array, then in any case, the algorithm makes n comparisons. So, let us assume that the key is present in the array at some index. We will assume that in this case, the key is equally likely to be any of the elements on the array. Under this assumption, the number of comparisons when $x = A[i]$ is exactly i. So, the average runtime is $\frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = n + 1/2$. Thus, as is reasonable, we say that on average the algorithm LinearSearch requires $n + 1/2$ comparisons.

Such examples are possible only for a few simple cases: binary search, linear search, and possibly a few sorting algorithms. Moreover, given the nature of the strong assumptions required to perform such an average case analysis, this is often not done in practice. Instead, one uses the worst case runtime as a guideline in making the choice of an algorithm for a given problem. In case of algorithms with comparable growth rates, one has to then consider implementations on the algorithms and go for empirical observations.

2.5 A Review of Recurrence Relations

Considering the examples from Section 2.2 and the idea of asymptotic analysis from Section 2.3, we note that solving recurrence relations can also be done in an asymptotic manner. This means that we are only interested in the growth rate of the solution to a recurrence relation than the exact solution. To this end, we shall see three general methods to solve recurrences namely: the substitution method, the recursion tree method, and the master theorem method. We shall look at each one in the following.

First a few technicalities: We ignore floors, ceilings, and boundary values.

2.5.1 The Substitution Method

The idea behind this method is that we first try to guess a solution to the recurrence relation and then verify whether our guess is correct. The verification is often done using mathematical induction. Thus, we substitute the guessed value in to the recurrence and hence the name. Let us take an example.

$$T(n) = 2T(n/2) + n$$

Guess: $T(n) \leq cn \log n$ for an appropriate choice of c. Verification is easy.

base case: $n = 0$ is a problem. Our asymptotic notation lets us pick n_0. So pick $n_0 > 1$.

Then the question one has to answer to how to guess a solution. It our guess good enough? Normally this comes through practice but there are some guidelines. Some subtelites.

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

A guess that does not work is $T(n) \leq c \cdot n$. However, the guess $T(n) \leq cn - b$ for a constant b works! The idea here is that using MI we can show something strong if the hypothesis is strong enough. Hence, reducing a lower order term helps in strengthening the hypothesis.

Change of Variables

This is a algebraic manipulation technique to simplify certain recurrences. Analogy: Change of variable in summation and integrations.

$$T(n) = 2T(\sqrt{n}) + \log n$$

It is difficult to guess a solution for the above recurrence. However, put $n = 2^m$. Then, $T(2^m) = 2T(2^{m-1}) + m$. Let $S(m) = T(2^m)$. Then we get $S(m) = 2S(m/2) + m$ for which we know the solution to be $S(m) = O(m \log m)$. Substuting back, we get $T(n) = O(\log n \cdot \log \log n)$ as $m = \log n$.

2.5.2 The Recursion Tree Method

Sometimes difficult to make a guess. Hence, expand the relation and get to the solution.

Imagine a tree where each internal node represents the cost of a subproblem. The root represents the cost of the entire problem. Leaf nodes represnent boundary values. Summing up the costs of all the problems at a level gives us the per-level cost. Adding up all the per-level costs gives us the total cost of the entire problem.

One can use the recursion tree to arrive at a good guess and then verify the guess as in the substitution method. Example:

$$T(n) = 2T(n/2) + O(n)$$

Draw the tree.
Another non-straightforward example:

$$T(n) = T(n/5) + T(4n/5) + cn$$

Both solutions are $O(n \log n)$.

2.5.3 Master Thoerem

A general rule to solve a class of recurrence relations of the form

$$T(n) = aT(n/b) + f(n)$$

where a, b are constants.

Theorem 2.5.1 (Masters Theroem) *Let $a \geq 1, b > 1$ be constants and $f(n)$ be a function. Let $T(n)$ be defined as*

$$T(n) = aT(n/b) + f(n)$$

Then $T(n)$ can be bounded as follows.

1. *If $f(n) = O(n^{\log_a b - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = O(n^{\log_a b})$.*
2. *If $f(n) = \Theta(n^{\log_a b})$ then $T(n) = \Theta(n^{\log_a b} \log n)$.*
3. *If $f(n) = \Omega(n^{\log_a b + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ all all suffciently large n then $T(n) = \Theta(f(n))$.*

Some intuitive explaination...
examples...

2.6 Lab, Tutorial, and Homework

The tutorial shall discuss a few examples of asymptotic analysis and the masters theorem.

The lab shall be used to implement a few algorithms with orders $O(n)$, $O(\log n)$, $O(n^2)$, $O(n \log n)$, and $O(n^3)$ and try to time these algorithms and plot them on a common plot.

Homework shall be a written homework with examples of solving recurrence relation, using masters theorem, and asymptotic analysis.