CSCI 240 -- Topic 3: Analysis of Algorithms Lecture Notes

This lecture introduces the common functions that are used for analyzing algorithms and some justification techniques for analyzing algorithms.

We are interested in the design of "good" data structures and algorithms. A **data structure** is a systematic way of organizing and accessing data, and an **algorithm** is a step-by-step procedure for performing some task in a finite amount of time. To classify some data structures and algorithms as "good", we must have precise ways of analyzing them.

Analyzing the efficiency of a program involves characterizing the **running time** and **space usage** of algorithms and data structure operations. Particularly, the running time is a natural measure of goodness, since time is precious.

Running Time

Most algorithms transform input objects into output objects. The running time of an algorithm or a data structure method typically grows with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by a lot of factors, such as the hardware environment and the software environment. In spite of the possible variations that come from different environmental factors, we would like to focus on **the relationship between the running time of an algorithm and the size of its input**. In the end, we would like to characterize an algorithm's running time as a function $f(n)$ of the input size n.

But what is the proper way of measuring it?

- Experimental analysis
- Theoretical analysis

Common Functions Used in Analysis

The Constant Function $f(n) = C$

For any argument n, the constant function $f(n)$ assigns the value C. It doesn't matter what the input size n is, $f(n)$ will always be equal to the constant value C. The most fundamental constant function is $f(n) = 1$, and this is the typical constant function that is used in the book.

The constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers. Executing one instruction a fixed number of times also needs constant time only.

Constant algorithm does not depend on the input size.

Examples: arithmetic calculation, comparison, variable declaration, assignment statement, invoking a method or function.

The Logarithm Function $f(n) = \log n$

It is one of the interesting and surprising aspects of the analysis of data structures and algorithms. The general form of a logarithm function is $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows:

- $x = \log_b n$, if and only if $b^x = n$

The value b is known as the **base** of the logarithm. Computing the logarithm function for any integer n is not always easy, but we can easily compute the smallest integer greater than or equal to $\log_b n$, for this number is equal to the number of times we can divide n by b until we get a number less than or equal to 1. For example, $\log_3 27$ is 3, since 27/3/3/3 = 1. Likewise, $\log_2 12 = 4$, since 12/2/2/2/2 = 0.75 <= 1.

The base-two approximating arises in the algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half. In fact, the most common base for the logarithm in computer science is 2. We typically leave it off when it is 2.

Logarithm function gets slightly slower as n grows. Whenever n doubles, the running time increases by a constant.

Examples: binary search.

The Linear Function $f(n) = n$

Another simple yet important function. Given an input value n, the linear function f assigns the value n itself.

This function arises in an algorithm analysis any time we do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n will require n comparisons. The linear function also represents the best running time we hope to achieve for any algorithm that processes a collection of n inputs.

Whenever n doubles, so does the running time.

Example: print out the elements of an array of size n.

The N-Log-N Function $f(n) = n\log n$

This function grows a little faster than the linear function and a lot slower than the quadratic function (n^2). If we can improve the running time of solving some problem from quadratic to N-Log-N, we will have an algorithm that runs much faster in general. It scales to a huge problem, since whenever n doubles, the running time more than doubles.

Example: merge sort, which will be discussed later.

The Quadratic Function $f(n) = n^2$

It appears a lot in the algorithm analysis, since there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. In such cases, the algorithm performs $n*n = n^2$ operations.

The quadratic function can also be used in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is $1 + 2 + 3 + ... + (n-1) + n$.

For any integer $n >= 1$, we have $1 + 2 + 3 + ... + (n-1) + n = n*(n+1) / 2$.

Quadratic algorithms are practical for relatively small problems. Whenever n doubles, the running time increases fourfold.

Example: some manipulations of the n by n array.

The Cubic Function and Other Polynomials

The cubic function $f(n) = n^3$. This function appears less frequently in the context of the algorithm analysis than the constant, linear, and quadratic functions. It's practical for use only on small problems. Whenever *n* doubles, the running time increases eightfold.

Example: *n* by *n* matrix multiplication.

The functions we have learned so far can be viewed as all being part of a larger class of functions, the **polynomials**. A polynomial function is a function of the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0.$$

where a_0, a_1, ... , a_n are constants, called the coefficients of the polynomial, and a_n != 0. Integer *n*, which indicates the highest power in the polynomial, is called the **degree** of the polynomial.

The Exponential Function $f(n) = b^n$

In this function, *b* is a positive constant, called the **base**, and the argument *n* is the **exponent**. In the algorithm analysis, the most common base for the exponential function is *b* = 2. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the *n*th iteration is 2^n.

Exponential algorithm is usually not appropriate for practical use.

Example: Towers of the Hanoi.

The Factorial Function $f(n) = n!$

Factorial function is even worse than the exponential function. Whenever *n* increases by 1, the running time increases by a factor of *n*.

For example, permutations of *n* elements.

Comparing Growth Rates

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or *n*-log-*n* time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs.

Let's draw the growth rates for the above functions and take a look at the following table.

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Focusing on the Worst Case

Think about the example of a linear search on an array. What are the best-, average-, and worst-case performance?

An algorithm may run faster on some inputs than it does on others of the same size. Thus we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. However, an **average case** analysis is typically challenging.

An average case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. In real-time computing, the **worst case** analysis is often of particular concern since it is important to know how much time might be needed in the worst case to guarantee that the algorithm would always finish on time. The term **best case** performance is used to describe the way an algorithm behaves under optimal conditions.

A worst case analysis is much easier than an average case analysis, as it requires only the ability to identify the worst case input. This approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input.

Primitive Operations

Primitive operations are basic computations performed by an algorithm. Examples are evaluating an expression, assigning a value to a variable, indexing into an array, calling a method, returning from a method, etc. They are easily identifiable in pseudocode and largely independent from the programming language.

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm as a function of the input size. Think about the worst case, best case, and average case.

```
Algorithm arrayMax(A, n):
  Input: An array A of n integers     Number of operations:
  Output: the max element


1 max = A[0]                          2
2 for i = 1 to n-1 do                 2n
3 if (A[i] > max) then max = A[i]     6(n-1) (including increment counter)
4 return max                          1
                                      Total: 8n-3
```

The algorithm arrayMax executes about 8n - 3 primitive operations in the worst case. Define:

- a = Time taken by the fastest primitive operation
- b = Time taken by the slowest primitive operation
- Let T(n) be the worst case time of arrayMax. Then a(8n - 3) <= T(n) <= b(8n - 3)
- The running time T(n) is bounded by two linear functions
- Changing the hardware/software environment will not affect the growth rate of T(n)

Asymptotic Notation

In the algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "**big-picture**" approach. It is often enough to know that the running time of an algorithm such as a linear search on an array **grows proportionally to *n***, with its true running time being n times a constant factor that depends on the specific computer. We analyze algorithms using a mathematical notation for functions that disregards constant factors. That is, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the **main factor** that determines the growth rate in terms of n. This approach allows us to focus on the **big-picture** aspects of an algorithm's running time.

```
Algorithm findElement(A, n, x):
  Input: An array A of n integers
  Output: True if the element x  is inside A, false otherwise
  for (i = 0; i < n; i++) do
     if (A[i] == x) then return true
  return false
```

The above algorithm can be expanded to the following set of instructions:

1. i = 0
2. if i < n, goto line 6
3. if A[i] = = x, goto line 7
4. i++
5. goto line 2
6. return false
7. return true

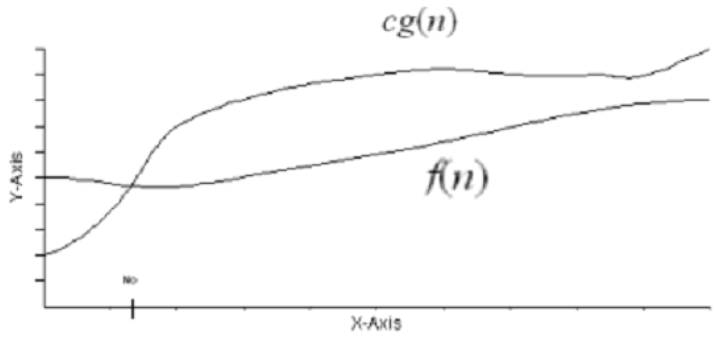
Assume that line i take Ci time, where Ci is a constant. The worst case total time of running this block of code can be calculated as: $C1 + (n+1)C2 + nC3 + nC4 + nC5 + 1 = (C2 + C3 + C4 + C5)n + (C1 + C2 + 1) = An + B$, where A = $C2 + C3 + C4 + C5$ and $B = C1 + C2 + 1$ are both constants.

The asymptotic analysis of an algorithm determines the running time in a big-Oh notation. To perform the asymptotic analysis:

- We first find the worst case number of primitive operations executed as a function of the input size.
- We then express this function with the big-Oh notation.
- Since constant factors and lower order terms are eventually dropped anyhow, we can disgard them when counting primitive operations.

The Big-Oh Notation

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that ***f*(*n*) is O(*g*(*n*))** if there is a real constant $c > 0$ and an integer constant $n_0 >= 1$ such that ***f*(*n*) <= *cg*(*n*)**, for $n >= n_0$. This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "***f*(*n*) is big-Oh of *g*(*n*)**." Alternatively, we can also say "$f(n)$ is **order of** $g(n)$."



Example: The function $f(n) = 8n - 2$ is $O(n)$.

Justification: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 >= 1$ such that $8n - 2 <= cn$ for every integer $n >= n_0$. It is easy to see that a possible choice is c = 8 and $n_0 = 1$.

More examples: $2n + 10$ is $O(n)$, $7n - 2$ is $O(n)$, $3n^3 + 20n^2 + 5$ is $O(n^3)$, $3\log n + 5$ is $O(\log n)$

The big-Oh notation allows us to say that **a function *f*(*n*) is less than or equal to another function *g*(*n*)** up to a constant factor and in the asymptotic sense as n grows towards infinity. If $f(n)$ is of the form of $An + B$, where A and B are constants. It's called a linear function, and it is $O(n)$.

The big-Oh notation gives an **upper bound** on the growth rate of a function. The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

Characterizing Running Times using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the size of the problem. For example, if we are interested in finding a specific element in an array of integers, we should let n denote the number of elements of the array. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of a sequential search algorithm for any computer.

Proposition: The sequential search algorithm, for searching a specific element in an array of n integers, runs in $O(n)$ time.

Justification: The number of primitive operations executed by the algorithm findElement in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of the algorithm findElement on an input of size n is at most a constant times n, that is, we may conclude that the running time of the algorithm findElement is $O(n)$.

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth the most.

Example: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Justification: $5n^4 + 3n^3 + 2n^2 + 4n + 1 <= (5 + 3 + 2 + 4 + 1)n^4 = cn^4$, for $c = 15$ and $n_0 = 1$.

Proposition: If $f(n)$ is a polynomial of degree d, that is, $f(n) = a_0 + a_1n + a_2n^2 + ... + a_dn^d$, and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n >= 1$, we have $1 <= n <= n^2 <= ... <= n^d$; hence $f(n) = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d <= (a_0 + a_1 + ... + a_d)\, n^d$; therefore, we can show that $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 + ... + a_d$ and $n_0 = 1$.

The highest degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial.

General rules: Characterizing Functions in Simplest Terms

In general we should use the big-Oh notation to characterize a function as closely as possible. For example, while it is true that $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$.

It is also considered a poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is unfashionable to say that the function $2n^3$ is $O(4n^3 + 8n\log n)$, although it is completely correct. We should strive to describe the function in the big-Oh in **simplest terms**.

Rules of using big-Oh:

- **If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$**. We can drop the lower order terms and constant factors.
- **Use the smallest/closest possible class of functions**, for example, "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- **Use the simplest expression of the class, for example**, "$3n + 5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"

Some words of caution

It is somewhat misleading when the constant factors are very large. It is true that $10^{100}n$ is $O(n)$. When it is compared with another running time $10n\log n$, we should prefer $O(n\log n)$ time, even though the linear-time algorithm is asymptotically faster.

Generally speaking, any algorithm running in $O(n\log n)$ time (with a reasonable constant factor) should be considered efficient. Even $O(n^2)$ may be fast when n is small. But $O(2^n)$ should almost never be considered efficient.

If we must draw a line between efficient and inefficient algorithms, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. Again, be reasonable here. $O(n^{100})$ is not efficient at all.

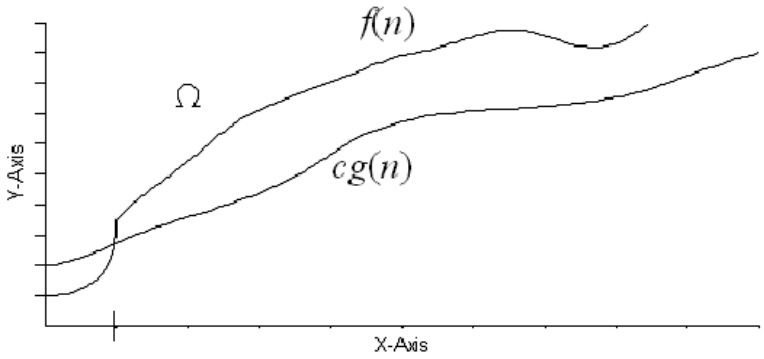
Exercises

- Show that n is $O(n\log n)$
- Show that if $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.
- If $f(n)$ is in $O(g(n))$, and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)*f_2(n)$ is in $O(g_1(n)*g_2(n))$.

Big-Omega

The big-Oh notation provides an asymptotic way of saying that a function is less than or equal to another function. This big-Omega notation provides an asymptotic way of saying that a function grows at a rate that is **greater than or equal to** that of another.

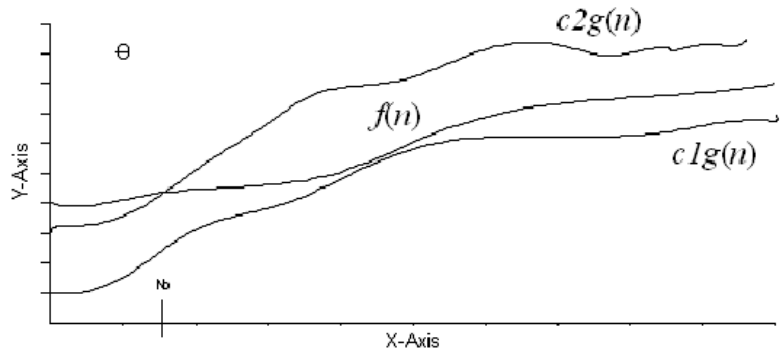
Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is **$\Omega(g(n))$** (pronounced "$f(n)$ is big-Omega of $g(n)$") if there is a real constant $c > 0$ and an integer constant $n_0 >= 1$ such that **$f(n) >= cg(n)$**, for $n >= n_0$.



Example, show that n^2 is $\Omega(n\log n)$.

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We way that ***f(n)* is Θ(*g*(*n*))** (pronounced "*f*(*n*) is big-Theta of *g*(*n*)") if *f*(*n*) is O(*g*(*n*)) and f(n) is Ω(*g*(*n*)), that is, there are real constants $c_1 > 0$ and $c_2 > 0$, and an integer constant $n_0 >= 1$ such that $c_1 g(n) <= f(n) <= c_2 g(n)$, for $n >= n_0$.



Example: $3n\log n + 4n + 5\log n$ is $\Theta(n\log n)$.

Justification: $3n\log n <= 3n\log n + 4n + 5\log n <= (3 + 4 + 5)\, n\log n$ for $n >= 2$.

To summarize, the asymptotic notations of big-Oh, big-Omega, and big-Theta provide a convenient language for us to analyze data structures and algorithms. They let us concentrate on the "big-picture" rather than low-level details.

Example, show that $5n^2$ is $O(n^2)$, $\Omega(n^2)$ and $\Theta(n^2)$.

Examples of Asymptotic Algorithm Analysis

Consider a problem of computing the prefix averages of a sequence of numbers. Namely, given an array *X* storing *n* numbers, we want to compute an array *A* such that *A*[*i*] is the average of elements *X*[0], ..., *X*[*i*], for *i* = 0, 1, ..., *n*-1.

```
Algorithm prefixAvg1(X):
  input: An n-element array X
  output: An n-element array A of  numbers such that A[i] is the average of elements X[0],
..., X[i]

1 Declare and initialize array A
2 for i = 0 to n-1 do
```

```
3   a = 0
4   for j = 0 to i do
5     a += X[j]
6   A[i] = a / (i+1)
7 return array A
```

```
Algorithm prefixAvg2(X):
  input: An n-element array X
 output: An n-element array A of numbers  such that A[i] is the average of elements X[0],
..., X[i]

1 Declare and initialize array A
2 a = 0
3 for i = 0 to n-1 do
4   a += X[i]
5   A[i] = a / (i+1)
6 return array A
```

Consider another program that raises a number *x* to an arbitrary nonnegative integer, *n*. We have solved this problem using linear recursion before.

```
Algorithm rpower(int x, int n):
1 if n == 0 return 1
2 else return x*rpower(x, n-1)

Algorithm brpower(int x, int n):
1 if n == 0 return 1
2 if n is odd then
3   y = brpower(x, (n-1)/2)
4   return x*y*y
5 if n is even then
6   y = brpower(x, n/2)
7   return y*y
```

Consider a program that returns true if the elements in an array are unique.

```
Algorithm Unique(A)
 for i = 0 to n-1 do
  for j = i+1 to n-1 do
   if A[i] equals to A[j] then
        return false
 return true
```

Merge-sort.

```
Algorithm MergeSort(A, 0, n-1)
 MergeSort(A, 0, n/2)
  MergeSort(A, n/2+1, n-1)
 MergeTogether(2 arrays above)
```

Analyze the big-Oh of the above MergeSort.

We can actually solve the recurrence relation given above. We'll sketch how to do that here. We'll write *n* instead of *O(n)* in the first line below because it makes the algebra much simpler.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4T(n/4) + 2n \\
&= \ldots \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

We know that $T(1) = 1$ and this is a way to end the derivation above. In particular we want $T(1)$ to appear on the right hand side of the = sign. This means we want:

$n/2^k = 1$ OR $n = 2^k$ OR $\log_2 n = k$

Continuing with the previous derivation we get the following since $k = \log_2 n$:

$= 2^k T(n/2^k) + kn$
$= 2^{\log 2n} T(1) + (\log_2 n)n$
$= n + n$ [remember that $T(1) = 1$]
$= O(n \log n)$

So we've solved the recurrence relation and its solution is what we "knew" it would be. To make this a formal proof you would need to use induction to show that O(*n* log *n*) is the solution to the given recurrence relation, but the "plug and chug" method shown above shows how to derive the solution --- the subsequent verification that this is the solution is something that can be left to a more advanced algorithms class.

Recurrence Relations to Remember

Recurrence	Algorithm	Big-Oh Solution
$T(n) = T(n/2) + O(1)$	Binary Search	O(log n)
$T(n) = T(n-1) + O(1)$	Sequential Search	O(n)
$T(n) = 2\ T(n/2) + O(1)$	tree traversal	O(n)
$T(n) = T(n-1) + O(n)$	Selection Sort (other n^2 sorts)	$O(n^2)$
$T(n) = 2\ T(n/2) + O(n)$	Mergesort (average case Quicksort)	O(n log n)

Last updated: Sept. 2022