BIRZEIT UNIVERSITY

Algorithmic Complexity

Introduction

- Algorithmic complexity is concerned about how fast or slow particular algorithm performs.
- We define complexity as a numerical function *T(n)* - time versus the input size *n*.
- We want to define time taken by an algorithm without depending on the implementation details. But you agree that *T(n)* does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as:
    - processor speed; instruction set, disk speed, brand of compiler and etc.

The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time *T(n)* as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where *c* is time taken by addition of two bits. On different computers, addition of two bits might take different time, say c_1 and c_2, thus the addition of two n-bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively. This shows that different machines result in different slopes, but time *T(n)* grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

Asymptotic Notations

- The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function T(n) using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

$T(n) = O(n^2)$
says that an algorithm has a quadratic time complexity.

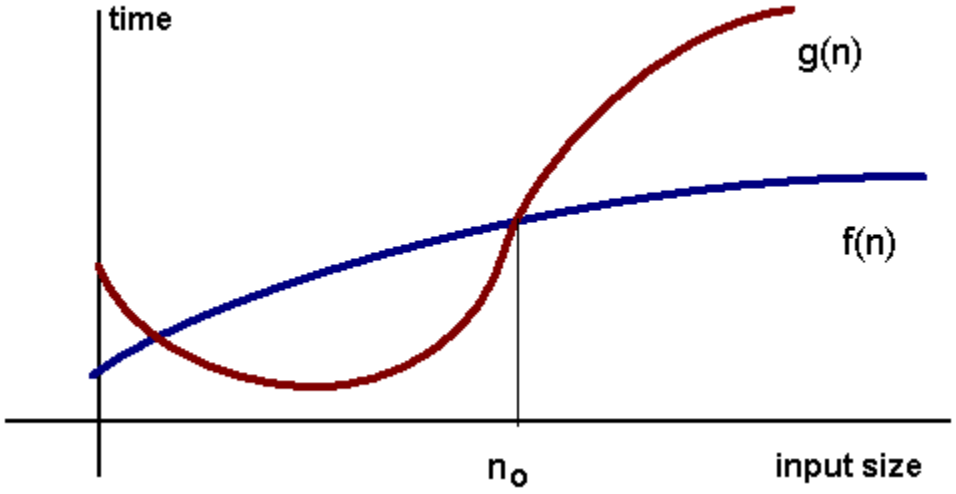
Definition of "big Oh"

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = O(g(n)) when there exist constants c > 0 and $n_0 > 0$ such that
$f(n) \leq c * g(n)$, for all $n \geq n_0$

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an **upper bound** for f(n), for all sufficiently large $n \to \infty$

By: Dr. Majdi Mafarja

Here is a graphic representation of f(n) = O(g(n)) relation:



Examples:

- $1 = O(n)$
- $n = O(n^2)$
- $\log(n) = O(n)$
- $2n + 1 = O(n)$

The "big-O" notation is not symmetric: $n = O(n^2)$ but $n^2 \neq O(n)$.

Exercise. Let us prove $n^2 + 2n + 1 = O(n^2)$. We must find such c and n_0 that $n^2 + 2n + 1 \leq c*n^2$. Let $n_0=1$, then for $n \geq 1$

$1 + 2n + n^2 \leq n + 2n + n^2 \leq n^2 + 2n^2 + n^2 = 4n^2$
Therefore, c = 4.

Constant Time: O(1)

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

Logarithmic Time: O(log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Example:

- binary search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess iss too high or too low. Twenty questions game imploies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as **binary search**:

> locate the element a in a sorted (in ascending order) array by first comparing a with the middle element and then (if they are not equal) dividing the array into two subarrays; if a is less than the middle element you repeat the whole procedure in the left subarray, otherwise - in the right subarray. The procedure repeats until a is found or subarray is a zero dimension.

Note, $\log(n) < n$, when $n \to \infty$. Algorithms that run in O(log n) does not use the whole input.

Quadratic Time: $O(n^2)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort, selection sort, insertion sort

Definition of "big Omega"

We need the notation for the **lower bound**. A capital omega Ω notation is used in this case. We say that $f(n) = \Omega(g(n))$ when there exist constant c that $f(n) \geq c*g(n)$ for for all sufficiently large n. Examples

- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $2n + 1 = O(n)$

Definition of "big Theta"

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that $f(n) = \Theta(g(n))$ if and only $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Examples

- $2\,n = \Theta(n)$
- $n^2 + 2\,n + 1 = \Theta(n^2)$

Analysis of Algorithms

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:

- the *worst-case runtime complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size a.
- the *best-case runtime complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size a.
- the *average case runtime complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size a.

Example. Let us consider an algorithm of sequential searching in an array.of size n.
Its *worst-case runtime complexity* is O(n)

Its *best-case runtime complexity* is O(1)

Its *average case runtime complexity* is O(n/2)=O(n)

Time Complexity meaningful for large value of n

The time complexity of an algorithm is given by the function which counts the total number of operations for n elements of data. What is really of concern here is not what the function is exactly but a description of how the function grows. Concretely, consider the two functions:

$f(n) = n + 20$

$g(n) = n - 10$

Which function grows faster?

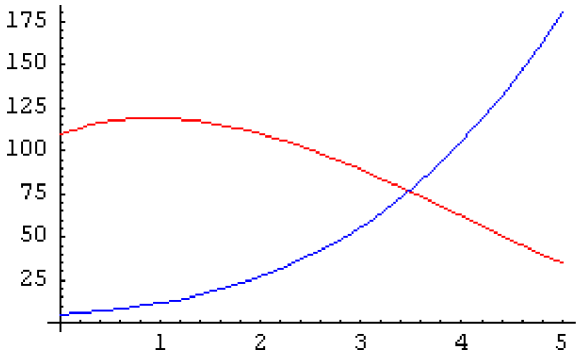
Try with small values of n say up to 100. It would show that f(n) grows faster. Now try values between 1000 and 10,000. What does it show? There is hardly any noticeable difference between values of f(n)

and g(n). Both grow linearly with n. Thus both are of order n.

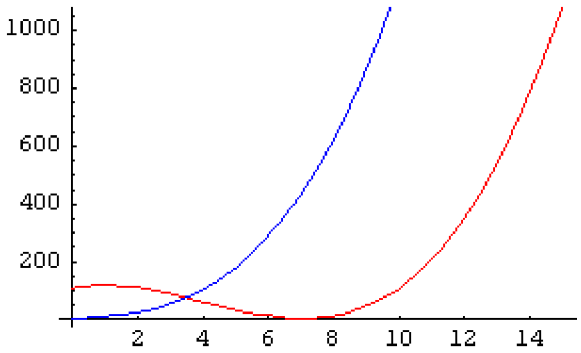
Let us now consider another pair of functions $f(n) = n^3 - 12n^2 + 20n + 110$

$g(n) = n^3 + n^2 + 5n + 5$

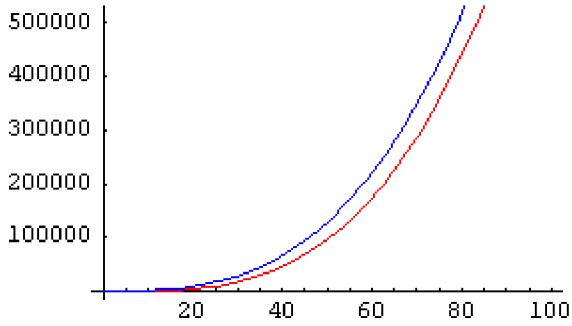
They look quite different, but how do they behave for different values of n? Let's look at a few plots of the function (f(n) is in red color , and g(n) is in blue color):



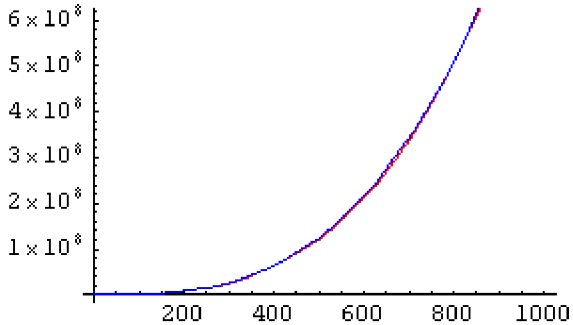
Plot of f and g, in range 0 to 5



Plot of f and g, in range 0 to 15



Plot of f and g, in range 0 to 100



Plot of f and g, in range 0 to 1000

In the first graph, drawn for very limited values of n, the curves appear somewhat different. In the second graph, they sort of start moving somewhat similar, in the third graph for appreciable values of n, there is only a very small difference between f(n) and g(n), and in the last graph drawn for very large values of n, the plots are virtually identical. In fact, they approach n^3, the dominant term. As n gets larger, the other terms become minuscule in comparison to n^3.

As you can see, improving an algorithm's non-dominant terms doesn't help much. What really matters is the dominant term. This is why we adopt the big-O notation for this. We say that:

$$f(n) = n^3 - 12n^2 + 20n + 110 = O(n^3)$$

Or in other words, f(n) is of order n^3 .

GENERAL RULES:

1. Loops: number of iterations/ recursions with respect to the input problem instance size N

2. Nested loops: multiplied

   for I = 1 through N do
   for j = 1 through N do
   -*whatever*-

   Analysis: O(N*N)

3. Consecutive statements: add steps, which leads to the max

   for I = 1 through N do
   -*whatever*-
   for I = 1 through N do
   for j = 1 through N do
   -*moreover*-

   Analysis: $O(N) + O(N^2)$ --> $O(N^2)$

4. Conditional statements: *max* of the alternative paths (worst-case)

   If (condition)
   S1
   Else
   S2

Example of two algorithms for the same problem:

Problem (***MAXIMUM SUBSEQUENCE SUM***): Given an array of numbers find a subsequence whose sum is maximum out of all such subsequences.

Example: 3, 4, –7, 1, 9, -2, 3, -1 (*e.g. stock-market data*)
Answer: 11 (*for subsequence 1+ 9 -2+ 3 = 11*)
[Note: for all positive integers the answer is sum of the whole array.]

Algorithm 1*:*

```
for (i = 0 through N-1) do
    for (j = i through N-1) do   // choice of subsequence i through j
    {       thisSum = 0;
            for (k = i through j) do  // addition loop
                thisSum = thisSum + a[k];        // O(N^3)

            if (thisSum > maxSum) then
                maxSum = thisSum;                // O(N^2)
    }
return maxSum;
```

End Algorithm 1.

Analysis of Algorithm 1: $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1 = \ldots = O(N^3)$

Algorithm 2*:*

```
for (i = 0 through N-1) do
    thisSum = 0;
    for (j = i through N-1) do
    {       thisSum = thisSum + a[j];  // reuse the partial sum from
                                       // the previous iteration
            if (thisSum > maxSum) then
                maxSum = thisSum;
    }
return maxSum;
```

End Algorithm 2.

Analysis of Algorithm 2: $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} 1 = \ldots = O(N^2)$

Algorithm 3:

```
maxSum = 0;  thisSum = 0;

for (j = 0 through N-1) do
        {       thisSum = thisSum + a[j];  // reuse the partial sum from
                                           // the previous iteration
                if (thisSum > maxSum) then
                        maxSum = thisSum;
                else if (thisSum < 0) then
                        thisSum =0;          // ignore computation so far
        }
return maxSum;
```

End Algorithm 3.

Analysis of Algorithm 3: $\sum_{i=0}^{N-1} O(1) = \ldots = O(N)$

Analysis of Recursive Algorithms

Let T(n) ≡ time complexity of (recursive) algorithm
Time complexity can be defined recursively.

Example 1: Factorial Function

Code:
```
int factorial (int n) {
      if (n = = 0) return 1;
      else return n * factorial (n – 1);
}
```

Recurrence Equation:

$$T(n) = T(n-1) + 1; \quad n > 0 \quad \{\text{all } n > 0\}$$
$$T(0) = 1 \quad ; \quad n = 0 \quad \{\text{base case}\}$$

Thus,

$T(n-1) = T(n-2) + 1$
$T(n-2) = T(n-3) + 1$
$T(n-3) = T(n-4) + 1$
. .
. .
. .

There are a variety of techniques for solving the recurrence equation (to solve for T(n) without T(n-1) on right side of the equation**)** to get a **closed form**. We'll start with the simplest technique: **Repeated Substitution**

Solving the recurrence equation of the factorial form to get the closed form,

$T(n) = T(n-1) + 1$

$= [T(n-2) + 1] + 1$

$= [[T(n-3) + 1] + 1] + 1$

$= \ldots\ldots$

$= T(n-i) + i$ *{i^{th} level of substitution}*

if 'i = n',

$T(n) = T(0) + n$

or, $\boxed{T(n) = n + 1}$

Example 2: Towers of Hanoi

Code:
```
procedure Hanoi (n: integer; from, to, aux: char)
    if  n > 0
        Hanoi (n – 1, from, aux, to)
        write ('from', from, 'to', to)
        Hanoi (n – 1, aux, to, from)
```

Here,

$T(0) = 0$

$T(n) = T(n – 1) + 1 + T(n – 1); \quad n > 0$

$= 2T(n – 1) + 1$

Solving the recurrence equation to get the closed form using repeated substitution,

$T(n) = 2T(n – 1) + 1$

$T(n – 1) = 2T(n – 2) + 1$

$T(n – 2) = 2T(n – 3) + 1$

$T(n – 3) = 2T(n – 4) + 1$

$T(n) = 2T(n – 1) + 1$

$= 2[2T(n – 2) + 1] + 1$

$= 2[2[2T(n – 3) + 1] + 1] + 1$

$= 2[2^2T(n – 3) + 2^1 + 2^0] + 2^0$

$= 2^3T(n – 3) + 2^2 + 2^1 + 2^0$

Repeating 'i' times,

$T(n) = 2^iT(n – i) + 2^{i-1} + 2^{i-2} + \ldots + 2^0$

If 'i = n',

$T(n) = 2^nT(0) + 2^{n-1} + 2^{n-2} + \ldots + 2^0$

$= 2^{n-1} + 2^{n-2} + \ldots + 2^0$

$= \sum_{i=0}^{n-1} 2^i$

This is a geometric series whose sum is given by: $S_n = \frac{a - ar^n}{1 - r}$

Here, a = 1, r = 2, n = n. Therefore, $S_n = \frac{1 - 2^n}{1 - 2} = 2^n - 1$.

i.e. $T(n) = 2^n - 1$

Example 3: Binary Search

Code:

```
function BinSearch (var R: Ratype; a,b: indextype; x: keytype): boolean
        var mid: indextype
            if a > b
                BinSearch:= false
            else
                mid:= (a + b) div 2
                if x = R[mid]
                    BinSearch:= true
                else
                    if x < R[mid]
                        BinSearch:=BinSearch(R,a,mid-1,x)
                    else
                        BinSearch:=BinSearch(R,mid+1,b,x)
```

Characteristic (OR Basic) operation

Here,

$T(0) = 0$

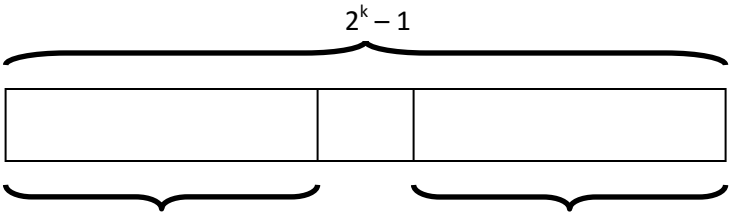
$T(n) = 1$ if x = R[mid]

$1 + T(\lfloor (n+1)/2 \rfloor - 1)$ if x < R[mid]

$1 + T(n - \lfloor (n+1)/2 \rfloor - 1)$ if x > R[mid]

} Complexity of BinSearch(R, 1,n, x)

We can eliminate the floor function by limiting 'n' to $2^k - 1$; $k \in Z^+$



Now,

$T(0) = 0$

$T(n) = 1$ if x = R[mid]

$1 + T(2^{k-1} - 1)$ if x ≠ R[mid]

Best case occurs if x = R[mid] is true the very first time.

Thus,

B(n) = 1

Worst case occurs if x = R[mid] is always false.

Now,

W(0) = 0

$W(2^k - 1) = 1 + W(2^{k-1} - 1)$

Solving the recurrence equation to get the closed form using repeated substitution,

$W(2^k - 1) = 1 + W(2^{k-1} - 1)$
$= 1 + [1 + W(2^{k-2} - 1)]$
$= 1 + [1 + [1 + W(2^{k-3} - 1)]]$

Repeating 'i' times,

$W(2^k - 1) = i + W(2^{k-i} - 1)$

if 'i = k',

$W(2^k - 1) = k + 0 = k$

We want W(n), so

$2^k - 1 = n$
or, $\log 2^k = \log (n + 1)$
or, $k = \log (n + 1)$ {k in terms of n}

Thus,

W(n) = log (n + 1)