Lecture 8

What we must do is that take the new problem and map it into an already known set of algorithms. See what does the new problem most look like.

* additive constants do not really matter. 90000 is almost same as 90002
* multiplicative constants do not matter. 300 years or 900 years. does it matter?!

We will be measuring the rate of growth as the size of the problem grows

And we will do this by the asymptotic notation.

especially by the “big oh notation” – this represents upper limit to growth of a function as the input gets large.

this is read as:f(x) is in big O of n^2

and represented as:f(x) ε Ο(n^2)

what this means is that: the function f(x) is such that it has an upper limit on itself and it grows no faster than a quadratic in n.

what are x and n? x is the input to the problem. and n measures the size of x.

* elements in a list
* characters in string
* size of the integer as we go along

we have to come up with how we characterize the growth of this problem in terms of this exponential growth.

for calculating is a\*\*b: steps by exp1(a,b) is 3b+2. the two and the three dnt matter. Thus the big oh notation for this are: Ο(b) – this is linear.

**exp2(a,b) – recursive exponentiation**

the number of steps is:

t(n) = is the time required to solve the problem of size n

t(b) = 3 + t(b-1) –recurrence relation

t(b) = 3 + 3 + t(b-2)

t(b) = 3k + t(b-k)

and we will be done when b-k = 1 or b-1 = k

t(b) = 3(b-1) +2 = 3b-1

* it is linear Ο(b)

**exp3(a,b) – yet another way to do this**

t(b) = what?

b is even : t(b) = 6 + t(b/2)

b is odd: t(b) = 6 + t(b-1)

= 12 + t((b-1)/2)

in any case we have:

t(b) = 12 + t(b/2)

-- even though this is b be even or odd we will be down to b/2 so its approximately okay!

t(b) = 12 + 12 + t(b/4)

t(b) = 12 + 12 + 12 + t(b/8)

t(b) = 12k + t(b/(2^k))

we are done when b/(2^k) = 1

or k = log b

Ο(log b)

thus the order is logarithmic and this matters. because it is not linear.

**g(n,m) – simple addition method for exponentiation**

we do n\*m steps. Ο(n\*m) approximately Ο(n squared).

**Towers of Hanoi**

move a tower of size n-1 from the fromStack to the SpareStack.

Then move the one disk that is left in the fromStack to the ToStack.

and now move the tower of size n-1 from the spareStack to the toStack.

t(n) = 1 + 2\*t(n-1) + t(1)

= 1 + 2\*(1+2\*t(n-2)+t(1))

= 1+2+4\*t(n-2) + 2\*t(1)

t(1) = 2(check and then print)

= 3(1+2+4+8…2^(k-1)) + 2^k(t(n-k))

this is Ο(2^n). Exponential growth.

**n = 1000 at nanosecond speed -- comparison of different class of algorithms**

|  |  |
| --- | --- |
| Logarithmic | 10 nanoseconds |
| Linear | 1 microsec |
| Quadratic | 1 millisec |
| Exponential | 10^284 years!!! |

**Identifying the classes**

Linear – one go, cut the problem by a constant size

Quadratic – we have 2 nested loops

Logarithmic – we cut the problem in half after one go

Exponential – after one go the problem is reduced to two or more problems of a smaller size

**Search a list that is sorted**

The worst case is that the element is not there in the list. Then the order is the length of the list: Ο(len(list))

**Binary Search:**

Ο(logarithmic)

accessing the nth element of a list is not done in constant time. It is not a primitive step.

if the list is made up of only integers then it would take a constant time to reach the element. But in general lists there would be a problem. the time is not constant.

in linked lists the time is not constant.

in python there is a list of pointers that points off to the value of that element. This makes sure that the time is constant.

In conclusion what we must remember is that “**WE MUST BE CAREFUL ABOUT CHOOSING WHAT A PRIMITIVE STEP IS”**

Summary:

1. must be able to recognize the basic classes of algorithms
   * log
   * linear
   * quadratic
   * exponential
2. we must know how to map any algorithm into those class of algorithms