# Algorithm Complexity

# Recursion

A subroutine which calls itself, with different parameters.

Need to evaluate factorial(n)

factorial(n) = n.(n-1)…2.1

= n∗factorial(n-1)

Suppose routine factorial(p) can find factorial of p for all p < m. Then factorial(m+1) can be solved as follows:

factorial(m+1) =(m+1)∗factorial(m)

Anything missing?

```
Factorial(m)
{
   If m = 1, Factorial(m) = 1;
   Else  Factorial(m) = m*Factorial(m-1)
}
```

Rules:

Have a base case for which the subroutine need not call itself.

For general case, the subroutine does some operations, calls itself, gets result and does some operations with the result

To get result, subroutine should progressively move towards the base case.

# Recursion Versus Iteration

```
prod = 1
For j=1  to m
   prod →prod *j
```

In general, iteration is more efficient than recursion because of maintenance of state information.

# Towers of Hanoi

Source peg, Destination peg, Auxiliary peg

k disks on the source peg, a bigger disk can never be on top of a smaller disk

Need to move all k disks to the destination peg using the auxiliary peg, without ever keeping a bigger disk on the smaller disk.

Why can this be solved by recursion?

We know how to move 1 disk from source to destination.

For two disks, move the top one to the auxiliary, bottom one to the destination, then first to the destination.

For three disks, move top two disks from source to auxiliary, using destination.

Then move the bottom one from the source to the destination.

Finally move the two disks from auxiliary to destination using source.

We know how to solve this for k=1

Suppose we know how to solve this for k-1 disks.

We will first move top k-1 disks from source to auxiliary, using destination.

Will move the bottom one from the source to the destination.

Will move the k-1 disks from auxiliary to destination using source.

```
Tower of Hanoi(k, source, auxiliary, destination)
  {
    If k=1 move disk from source to destination; (base case)
    Else,
      {
        Tower of Hanoi(top k-1, source, destination,
                                              auxiliary);
        Move the kth disk from source to destination;
        Tower of Hanoi(k-1, auxiliary, source, destination);
      }
}
```

# Efficient Algorithms

A city has n view points

Buses ply from one view point to another

A bus driver wishes to follow the shortest path (travel time wise).

Every view point is connected to another by a road.

However, some roads are less congested than others.

Also, roads are one-way, i.e., the road from view point 1 to 2, is different from that from view point 2 to 1.

How to find the shortest path between any two pairs?

Naïve approach:

List all the paths between a given pair of view points

Compute the travel time for each.

Choose the shortest one.

How many paths are there between any two view points?

$$n! \cong (n/e)^n$$

Will be impossible to run your algorithm for n = 30

# What is efficiency of an algorithm?

Run time in the computer

Machine Dependent

Need to multiply two positive integers a and b

Subroutine 1: Multiply a and b

Subroutine 2: V = a, W = b

While W > 1

V $\rightarrow$V + a; W $\rightarrow$W-1

Output V

First subroutine has 1 multiplication.
Second has b additions and subtractions.

For some architectures, 1 multiplication is more expensive than b additions and subtractions.

Ideally, we would like to program all choices and run all of them in the machine we are going to use and find which is efficient!

# Machine Independent Analysis

We assume that every basic operation takes constant time:

Example Basic Operations:

Addition, Subtraction, Multiplication, Memory Access

Non-basic Operations:

Sorting, Searching

Efficiency of an algorithm is the number of basic operations it performs

We do not distinguish between the basic operations.

Subroutine 1 uses 1 basic operation

Subroutine 2 uses b basic operations

Subroutine 1 is more efficient.

This measure is good for all large input sizes

In fact, we will not worry about the exact values, but will look at ``broad classes' of values.

Let there be n inputs.

If an algorithm needs n basic operations and another needs 2n basic operations, we will consider them to be in the same efficiency category.

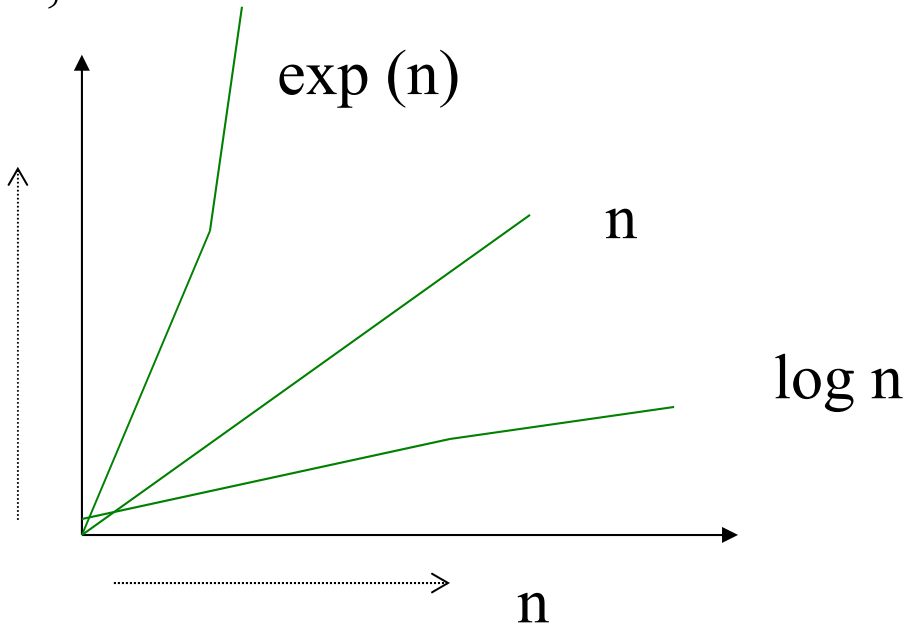
However, we distinguish between exp(n), n, log(n)

# Order of Increase

We worry about the speed of our algorithms for large input sizes.

Note that for large n, log(n)/n , and n/exp(n) are very small.

However, n/2n is a constant for all n.



# Function Orders

A function f(n) is O(g(n)) if ``increase'' of f(n) is not faster than that of g(n).

A function f(n) is O(g(n)) if there exists a number $n_0$ and a nonnegative c such that for all $n \geq n_0$ , $0 \leq f(n) \leq cg(n)$.

If $\lim_{n \to \infty} f(n)/g(n)$ exists and is finite, then f(n) is O(g(n))

# Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + $n^2$

$\lim_{n\to\infty}$ sqrt(n) /n = 0, sqrt(n) is O(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity, n is not O(sqrt(n))

$\lim_{n\to\infty}$ n /2n = 1/2, n is O(2n)

$\lim_{n\to\infty}$ 2n /n = 2, 2n is O(n)

$\lim_{n\to\infty} \ln(n)/n = 0$, ln(n) is O(n)

$\lim_{n\to\infty} n/\ln(n) = \text{infinity}$, n is not O(ln(n))

$\lim_{n\to\infty} \exp(n)/n = \text{infinity}$, exp(n) is not O(n)

$\lim_{n\to\infty} n/\exp(n) = 0$, n is O(exp(n))

$\lim_{n\to\infty} (n+\text{sqrt}(n))/n = 1$, n + sqrt(n) is O(n)

$\lim_{n\to\infty} n/(\text{sqrt}(n)+n) = 1$, n is O(n+sqrt(n))

$\lim_{n\to\infty} (n + n^2)/n = \text{infinity}$, $n + n^2$ is not O(n)

$\lim_{n\to\infty} n/(n + n^2) = 0$, n is $O(n + n^2)$

# Implication of O notation

Suppose we know that our algorithm uses at most O(f(n)) basic steps for any n inputs, and n is sufficiently large, then we know that our algorithm will terminate after executing at most constant times f(n) basic steps.

We know that a basic step takes a constant time in a machine.

Hence, our algorithm will terminate in a constant times f(n) units of time, for all large n.

# Other Complexity Notation

Intuitively, (not exactly) f(n) is O(g(n)) means $f(n) \leq g(n)$

g(n) is an upper bound for f(n).

Now a lower bound notation, $\Omega(n)$

f(n) is $\Omega(g(n))$ if $f(n) \geq cg(n)$ for some positive constant c, and all large n.

$\lim_{n\to\infty} (f(n)/g(n)) > 0$, if $\lim_{n\to\infty} (f(n)/g(n))$ exists

f(n) is $\theta(g(n))$ if f(n) is $O(g(n))$ and $\Omega(g(n))$

$\theta(g(n))$ is ``asymptotic equality''

$\lim_{n\to\infty} (f(n)/g(n))$ is a finite, positive constant, if it exists

f(n) is $o(g(n))$ if f(n) is $O(g(n))$ but not $\Omega(g(n))$

``asymptotic strict inequality''

f(n) is $o(g(n))$ if given any positive constant c, there exists some m such that $f(n) < cg(n)$ for all $n \geq m$

$\lim_{n\to\infty} (f(n)/g(n)) = 0$, if $\lim_{n\to\infty} (f(n)/g(n))$ exists

| | |
|---|---|
| Asymptotically less than or equal to | $O$ |
| Asymptotically greater than or equal to | $\Omega$ |
| Asymptotically equal to | $\theta$ |
| Asymptotically strictly less | $o$ |

# Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + $n^2$

$\lim_{n\to\infty}$ sqrt(n) /n = 0, sqrt(n) is o(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity, n is Ω(sqrt(n))

$\lim_{n\to\infty}$ n /2n = 1/2, n is θ(2n), Ω(2n)

$\lim_{n\to\infty}$ 2n /n = 2, 2n is θ(n), Ω(n)

$\lim_{n\to\infty} \ln(n)/n = 0$, ln(n) is o(n)

$\lim_{n\to\infty} n/\ln(n) = \text{infinity}$, n is $\Omega(\ln(n))$

$\lim_{n\to\infty} \exp(n)/n = \text{infinity}$, exp(n) is $\Omega(n)$

$\lim_{n\to\infty} n/\exp(n) = 0$, n is o(exp(n))

$\lim_{n\to\infty} (n+\text{sqrt}(n))/n = 1$, n + sqrt(n) is $\Omega(n)$,$\theta(n)$

$\lim_{n\to\infty} n/(\text{sqrt}(n)+n) = 1$, n is $\Omega(n+\text{sqrt}(n))$, $\theta(n+\text{sqrt}(n))$,

$\lim_{n\to\infty} (n + n^2)/n = \text{infinity}$, $n + n^2$ is $\Omega(n)$

$\lim_{n\to\infty} n/(n + n^2) = 0$, n is $o(n + n^2)$

# Implication of the Ω Notation

Suppose, an algorithm has complexity Ω(f(n)) . This means that there exists a positive constant c such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least cf(n) steps.

# Complexity of a Problem Vs Algorithm

A problem is O(f(n)) means there is some O(f(n)) algorithm to solve the problem.

A problem is $\Omega$(f(n)) means every algorithm that can solve the problem is $\Omega$(f(n))