Introduction to Computers and Programming

Prof. I. K. Lundqvist

Recitation 3
April 23 2004

Big-O

- Given function f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 so that
  $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: 2n + 10 is O(n)
  - $2n + 10 \leq cn$
  - $10 \leq n(c - 2)$
  - $n \geq 10/(c-2)$
  - Pick $c = 3$ and $n_0 = 10$

Big-O

- $4n - 2$ is $O(n)$
  - Need a $c > 0$ and $n_0 \geq 1$ so that
    $\mathbf{4n-2 \leq cn}$ for $\mathbf{n \geq n_0}$
    true for $\mathbf{c = 4}$ and $\mathbf{n_0 = 1}$

- $5n^3 + 10n^2 + 4n + 2$ is $O(n^3)$
  - Need a $c > 0$ and $n_0 \geq 1$ so that
    $\mathbf{5n^3+10n^2+4n+2 \leq cn^3}$ for $\mathbf{n \geq n_0}$
    true for $\mathbf{c = 21}$ and $\mathbf{n_0 = 1}$

- $2 \log_2 n + 3$ is $O(\log_2 n)$
  - Need a $c > 0$ and $n_0 \geq 1$ so that
    $\mathbf{2\log_2 n + 3 \leq c \log_2 n}$ for $\mathbf{n \geq n_0}$
    true for $\mathbf{c = 5}$ and $\mathbf{n_0 = 2}$

Big-O

- Given function f(n) and g(n), we say that **f(n) is O(g(n))** if there are positive constants c and n_0 so that
  $f(n) \leq cg(n)$ for $n \geq n_0$

	f(n) is O(g(n))	g(n) is O(f(n))
g(n) grows more	Yes	No
f(n) grows more	No	Yes
g(n) and f(n) has same growth	Yes	Yes

Ex 1

```
type Int_Array is array (Integer range <>) of Integer;

procedure Measure (A : Int_Array ) is
   Sum   : Integer := 0;
begin

   for I in A'range loop

      for J in A'range loop
         Sum := Sum + A(J);
      end loop;                          Inner loop

   end loop;                                              Outer loop

end Measure;
```


Statement	Runs in X time	Executes # of times
Variable Sum is initialized	Constant1	1
Array of size n is created	Constant2	1
Variable I is created and initialized	Constant3	1
I is tested against A'range (n)	Constant4	n+1
Variable J is created and initialized	Constant5	n
J is tested against A'range (n)	Constant6	n(n+1)
Sum is incremented by A(J)	Constant7	n^2
J is incremented by 1	Constant8	n^2
I is incremented by 1	Constant9	n

BigO.adb

Ex 2

```
type Int_Array is array (Integer range <>) of Integer;

procedure Measure (A : Int_Array ) is
   Sum   : Integer := 0;
begin

   for I in A'range loop

      for J in 1 .. I loop -- only change to Ex 1
         Sum := Sum + A(J);
      end loop;

   end loop;

end Measure;
```

BigO2.adb

CQ – Ex 2

Variable J is created and initialized	Constant5	
J is tested against I	Constant6	
Sum is incremented by A(J)	Constant7	
J is incremented by 1	Constant8	

1. N, N*(N+1), N*N, N
2. N, N*(I+1), N*N, N*N
3. N, N*(I+1), N*I, N*I
4. I still don't get it

Ex 3

```
type Int_Array is array (Integer range <>) of Integer;

procedure Measure (A : Int_Array ) is
   Sum   : Integer := 0;
begin

   for I in A'range loop

      for J in 1 .. 4 loop     -- only change to Ex 2
         Sum := Sum + A(I);    -- only change to Ex 2
      end loop;

   end loop;

end Measure;
```

BigO3.adb

CQ – Ex 3

Variable J is created and initialized	Constant5	
J is tested against I	Constant6	
Sum is incremented by A(J)	Constant7	
J is incremented by 1	Constant8	

1. N, N*(I+1), N*I, N*I
2. N, N*5, N*4, N*4
3. N, N*5, 4, 4
4. I still don't get it

Ex 4

```
function Factorial (N : in Natural ) return Positive is
begin
   if N = 0 then
      return 1;
   else
      return N * Factorial (N-1);
   end if;
end Factorial;
```


CQ – Ex 4

How long time does executing the Factorial algorithm take?

1. O(n)
2. $O(n^2)$
3. log n
4. 42

Divide and Conquer

- It is an algorithmic design paradigm that contains the following steps
  - **Divide**: Break the problem into smaller sub-problems
  - **Recur**: Solve each of the sub-problems recursively
  - **Conquer**: Combine the solutions of each of the sub-problems to form the solution of the problem

Represent the solution using a recurrence equation

Merge Sort

- **Divide**: Split the array into into two subarrays $A(p .. mid)$ and $A(mid+1 .. r)$, where *mid* is $(p + r)/2$
- **Conquer** by recursively sorting the two subarrays $A(p .. mid)$ and $A(mid+1 .. r)$
- **Combine** by merging the two sorted subarrays $A(p .. mid)$ and $A(mid+1 .. r)$ to produce a single sorted subarray $A(p .. r)$

Merge

- **Input**: Array *A* and indices *p, mid, r* such that
  - $p \leq mid < r$
  - subarray *A(p .. mid)* is sorted and subarray *A(mid+1 .. r)* is sorted

- **Output**: single sorted array *A(p .. r)*

- **T(n) = O(n),** where $n=r-p+1$ = # of elements being merged

Merge Sort Analysis

- **The base case**: when $n = 1$, $T(n)=O(1)$
- When $n \geq 2$, time for merge sort steps:
  - **Divide**: Compute mid as the average of p, r
    $\Rightarrow$*cost* = O(1)
  - **Conquer**: Solve 2 subproblems, each of size *n/ 2*
    $\Rightarrow$cost = $2T(n/2)$
  - **Combine**: merge to an *n element subarray*
    $\Rightarrow$ cost = $O(n)$

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(n) + O(1) & n > 1 \end{cases}$$

Solving Recurrences: Iteration

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- $T(n) =$
  $aT(n/b) + cn$
  $a(aT(n/b/b) + cn/b) + cn$
  $a^2T(n/b^2) + cna/b + cn$
  $a^2T(n/b^2) + cn(a/b + 1)$
  $a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1)$
  $a^3T(n/b^3) + cn(a^2/b^2) + cn(a/b + 1)$
  $a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1)$
  …
  $a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So we have
  - $T(n) = a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
- For $k = \log_b n$
  - $n = b^k$
  - $T(n) = a^kT(1) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= a^kc + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= ca^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= cna^k/b^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a = b$?
  - $T(n) = cn(k + 1)$
    $= cn(\log_b n + 1)$
    $= O(n \log n)$

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(n) + O(1) & n > 1 \end{cases}$$

The Master Method

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n/b*
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f(n)*
  - The **master method** provides a simple "**cookbook**" solution

Simplified Master Method

- **T(n) = aT(n/b) + cn^k**, where a,c > 0 and b > 1

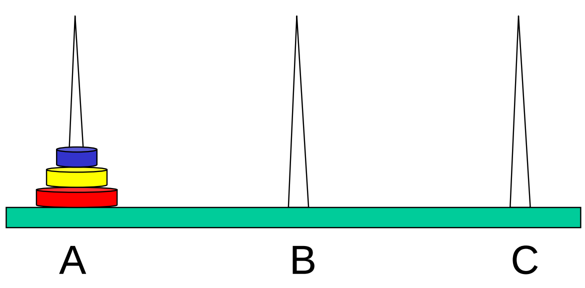
$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & a > b^k \\ O\left(n^k \log_b n\right) & a = b^k \\ O\left(n^k\right) & a < b^k \end{cases}$$

The Towers of Hanoi

- **Goal**: Move stack of rings to another peg
  - May only move 1 ring at a time
  - May never have larger ring on top of smaller ring

The Towers of Hanoi

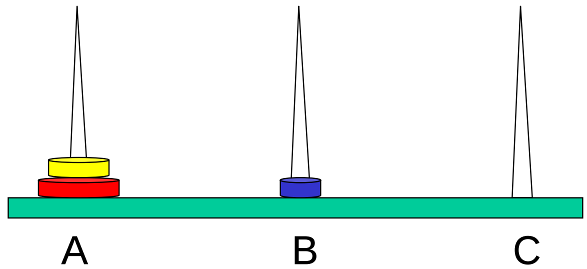
For simplicity, suppose there were just 3 disks



Since we can only move one disk at a time, we move the top disk from A to B.

The Towers of Hanoi

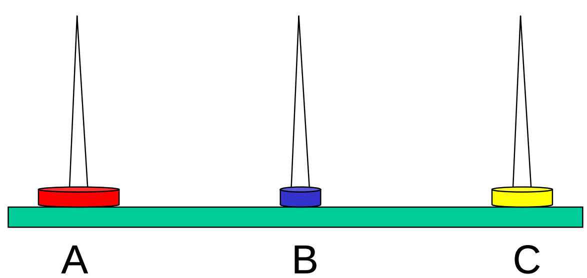
For simplicity, suppose there were just 3 disks



We then move the top disk from A to C.

The Towers of Hanoi

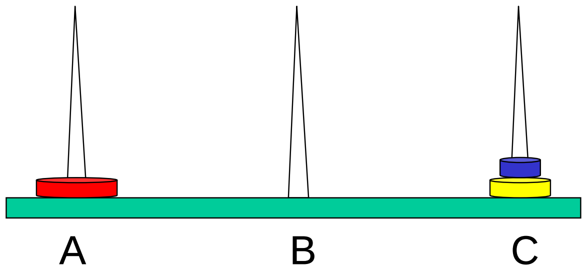
For simplicity, suppose there were just 3 disks



We then move the top disk from B to C.

The Towers of Hanoi

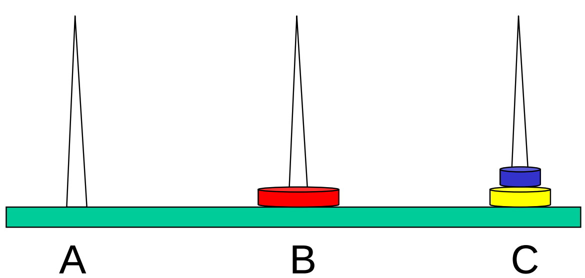
For simplicity, suppose there were just 3 disks



We then move the top disk from A to B.

The Towers of Hanoi

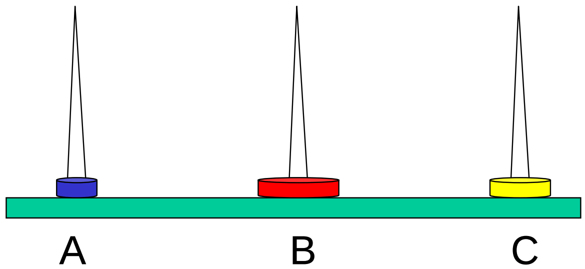
For simplicity, suppose there were just 3 disks



We then move the top disk from C to A.

The Towers of Hanoi

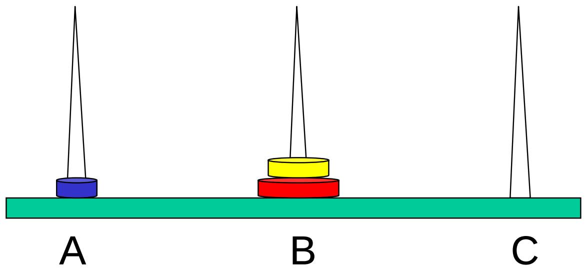
For simplicity, suppose there were just 3 disks



We then move the top disk from C to B.

The Towers of Hanoi

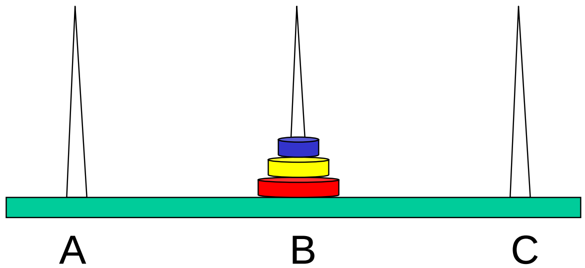
For simplicity, suppose there were just 3 disks



We then move the top disk from A to B.

The Towers of Hanoi

For simplicity, suppose there were just 3 disks



and we're done!

The problem gets more difficult as the number of disks increases...

The Towers of Hanoi

- 1 ring → 1 operation
- 2 rings → 3 operations
- 3 rings → 7 operations
- 4 rings → 15 operations

Cost: $2^N-1 = O(2^N)$

- 64 rings → 2^{64} operations

Towers of Hanoi

- ```
  hanoi(from,to,other,number)
      -- move the top number disks
      -- from needle from to needle to
  if number=1 then
     move the top disk from needle from
     to needle to
  else
      hanoi(from,other,to, number-1)
      hanoi(from,to,other, 1)
      hanoi(other,to, from, number-1)
  end
  ```


Some math that is good to know

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b xa = a\log_b x$
- $\log_b a = \log_x a/\log_x b$
- $a^{(b+c)} = a^b a^c$
- $a^{bc} = (a^b)^c$
- $a^b/a^c = a^{(b-c)}$
- $b = a^{\log_a b}$
- $b^c = a^{c\log_a b}$
  ```