

Orders of Growth of Processes

- Today's topics
 - Resources used by a program to solve a problem of size n
 - Time
 - Space
 - Define order of growth
 - Visualizing resources utilization using our model of evaluation
 - Relating types of programs to orders of growth

2/10/05

6.001 SICP

1/36

Resources matter

- How many seconds in a year?
 - $60 * 60 * 24 * 365 \approx \pi * 10^7$
 - Lifetime of Universe (since "Big Bang") $\approx 10^{10}$ years
 - Computer "clock" $\approx 3 * 10^9$ operations/sec
 - Operations since big bang
 - $\pi * 10^7 * 10^{10} * 3 * 10^9 \approx 10^{27}$
 - Number of atoms in the universe $\approx 10^{79}$
 - Imagine that every atom is a contemporary computer!
 - Total number of operations $\approx 10^{106}$
-
- Shannon/Knuth: Number of possible chess games $\approx 10^{120}$
 - Lloyd QM universe lifetime: $\approx 10^{122}$ ops on 10^{90} bits

2/10/05

6.001 SICP

2/36

Orders of growth of processes

- Suppose n is a parameter that measures the size of a problem
- Let $R(n)$ be the amount of resources needed to compute a procedure of size n .
- We say $R(n)$ has order of growth $\Theta(f(n))$ if there are constants k_1 and k_2 such that $k_1 f(n) \leq R(n) \leq k_2 f(n)$ for large n ($n > k$)
- Two common resources are **space**, measured by the number of deferred operations, and **time**, measured by the number of primitive steps.

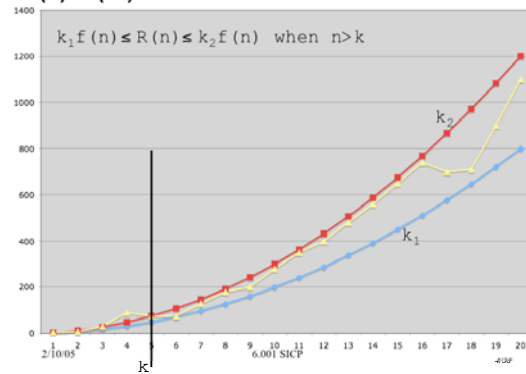
We need to specify what is primitive – typically we will use simple arithmetic operations and simple data structure operations (to be defined next time)

2/10/05

6.001 SICP

3/36

$R(n) = \Theta(n^2)$



Partial trace for (fact 4)

```
(define fact (lambda (n)
  (if (= n 1) 1
      (* n (fact (- n 1))))))

(fact 4)
(* 4 (fact 3))
(* 4 (if (= 3 1) 1 (* 3 (fact (- 3 1)))))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

2/10/05

6.001 SICP

5/36

Partial trace for (ifact 4)

```
(define ifact-helper (lambda (product count n)
  (if (> count n) product
      (ifact-helper (* product count)
                    (+ count 1) n))))

(define ifact (lambda (n) (ifact-helper 1 1 n)))

(ifact 4)
(ifact-helper 1 1 4)
(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))
(ifact-helper 1 2 4)
(if (> 2 4) 1 (ifact-helper (* 1 2) (+ 2 1) 4))
(ifact-helper 2 3 4)
(if (> 3 4) 2 (ifact-helper (* 2 3) (+ 3 1) 4))
(ifact-helper 6 4 4)
(if (> 4 4) 6 (ifact-helper (* 6 4) (+ 4 1) 4))
(ifact-helper 24 5 4)
(if (> 5 4) 24 (ifact-helper (* 24 5) (+ 5 1) 4))
24
```

2/10/05

6.001 SICP

6/36

Examples of orders of growth

```
(define fact (lambda (n)
  (if (= n 1) 1
      (* n (fact (- n 1))))))
```

• FACT

- Space $\Theta(n)$ – linear – (n-1 deferred ops)
- Time $\Theta(n)$ – linear – (2(n-1) primitive ops)

```
(define ifact-helper
  (lambda (product count n)
    (if (> count n) product
        (ifact-helper
         (* product count)
         (+ count 1) n))))
(define ifact
  (lambda (n) (ifact-helper 1 1 n)))
```

• IFACT

- Space $\Theta(1)$ – constant
- Time $\Theta(n)$ – linear – (2n primitive ops)

2/10/05

6.001 SICP

7/36

Computing Fibonacci

- Consider the following function
- $F(n) = 0$ if $n = 0$
- $F(n) = 1$ if $n = 1$
- $F(n) = F(n-1) + F(n-2)$ otherwise

2/10/05

6.001 SICP

8/36

Fibonacci

```
(define fib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2)))))))
```

New expression:

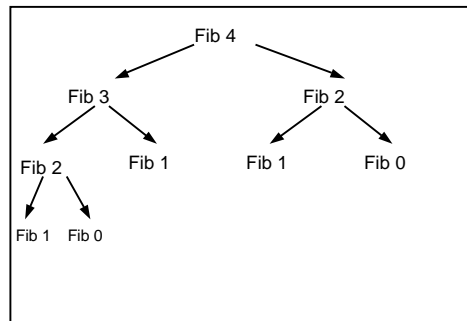
```
(cond (<predicate1> <consequent> <consequent> ...)
      (<predicate2> <consequent> <consequent> ...)
      ...
      (else <consequent> <consequent>))
```

2/10/05

6.001 SICP

9/36

A tree recursion



2/10/05

6.001 SICP

10/36

Orders of growth for Fibonacci

```
(define fib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2)))))))
```

- Let t_n be the number of steps that we need to take to solve the case for size n . Then
- $t_n = t_{n-1} + t_{n-2} = 2 t_{n-2} = 4 t_{n-4} = 8 t_{n-6} = 2^{n/2}$
- So in time we have $\Theta(2^n)$ -- exponential
- In space, we have one deferred operation for each increment of the argument -- $\Theta(n)$ -- linear

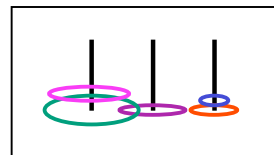
2/10/05

6.001 SICP

11/36

Towers of Hanoi

- Three posts, and a set of different size disks
- any stack must be sorted in decreasing order from bottom to top
- the goal is to move the disks one at a time, while preserving these conditions, until the entire stack has moved from one post to another



2/10/05

6.001 SICP

12/36

Using different processes for the same goal

- Identify smallest size subproblem
 - $a^0 = 1$

```
(define my-expt
  (lambda (a b)
    (if (= b 0)
        1
        (* a (my-expt a (- b 1))))))
```

Using different processes for the same goal

```
(define my-expt
  (lambda (a b)
    (if (= b 0)
        1
        (* a (my-expt a (- b 1))))))
```

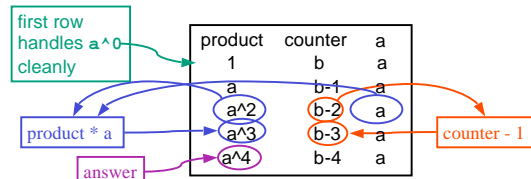
- Orders of growth
 - Time: **linear**
 - Space: **linear**

Using different processes for the same goal

- Are there other ways to decompose this problem?
- Use the idea of state variables, and table evolution

Iterative algorithm to compute a^b as a table

- In this table:
 - One column for each piece of information used
 - One row for each step



- The last row is the one where counter = 0
- The answer is in the product column of the last row

Iterative algorithm to compute a^b

```
(define exp-i (lambda (a b) (exp-i-help 1 b a)))

(define exp-i-help
  (lambda (prod count a)
    (if (= count 0)
        prod
        (exp-i-help (* prod a) (- count 1) a))))
```

Iterative algorithm to compute a^b

```
(define exp-i (lambda (a b) (exp-i-help 1 b a)))

(define exp-i-help
  (lambda (prod count a)
    (if (= count 0)
        prod
        (exp-i-help (* prod a) (- count 1) a))))
```

- Orders of growth
 - Space: **constant**
 - Time: **linear**

Another kind of process

- Let's compute a^b just using multiplication and addition
- If b is even, then $a^b = (a^2)^{b/2}$
- If b is odd, then $a^b = a * a^{b-1}$
- Note that here, we reduce the problem in half in one step

```
(define fast-exp-1
  (lambda (a b)
    (cond ((= b 1) a)
          ((even? b) (fast-exp-1 (* a a) (/ b 2)))
          (else (* a (fast-exp-1 a (- b 1)))))))
```

2/10/05

6.001 SICP

25/36

Orders of growth

```
(define fast-exp-1
  (lambda (a b)
    (cond ((= b 1) a)
          ((even? b) (fast-exp-1 (* a a) (/ b 2)))
          (else (* a (fast-exp-1 a (- b 1)))))))
```

- If n even, then 1 step reduces to $n/2$ sized problem
- If n odd, 2 steps reduces to $n/2$ sized problem
- Thus in $2k$ steps reduces to $n/2^k$ sized problem
- We are done when the problem size is just 1, which implies order of growth in time of $\Theta(\log n)$ -- logarithmic
- Space is similarly $\Theta(\log n)$ -- logarithmic

2/10/05

6.001 SICP

26/36

Another example of different processes

- Suppose we want to compute the elements of Pascal's triangle

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
```

2/10/05

6.001 SICP

27/36

Fun with Pascal's Triangle

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
13
```

2/10/05

6.001 SICP

28/36

Pascal's triangle

- We need some notation
 - Let's order the rows, starting with $n=0$ for the first row
 - The n th row then has $n+1$ elements, also ordered from 0
 - Let's use $P(j,n)$ to denote the j th element of the n th row.
 - We want to find ways to compute $P(j,n)$ for any n , and any j , such that $0 \leq j \leq n$

2/10/05

6.001 SICP

29/36

Pascal's triangle the traditional way

- Traditionally, one thinks of Pascal's triangle being formed by the following informal method:
 - The first element of a row is 1
 - The last element of a row is 1
 - To get the second element of a row, add the first and second element of the previous row
 - To get the k 'th element of a row, add the $(k-1)$ 'st and k 'th element of the previous row

2/10/05

6.001 SICP

30/36

Pascal's triangle the traditional way

- Here is a procedure that just captures that idea:

```
(define pascal
  (lambda (j n)
    (cond ((= j 0) 1)
          ((= j n) 1)
          (else (+ (pascal (- j 1) (- n 1))
                   (pascal j (- n 1)))))))
```

2/10/05

6.001 SICP

31/36

Pascal's triangle the traditional way

```
(define pascal
  (lambda (j n)
    (cond ((= j 0) 1)
          ((= j n) 1)
          (else (+ (pascal (- j 1) (- n 1))
                   (pascal j (- n 1)))))))
```

- What kind of process does this generate?
- Looks a lot like fibonacci
 - There are two recursive calls to the procedure in the general case
 - In fact, this has a time complexity that is **exponential** and a space complexity that is **linear**

2/10/05

6.001 SICP

32/36

Solving the same problem a different way

- Can we do better?
- Yes, but we need to do some thinking.
 - Pascal's triangle actually captures the idea of how many different ways there are of choosing objects from a set, where the order of choice doesn't matter.
 - P(0, n) is the number of ways of choosing collections of no objects, which is trivially 1.
 - P(n, n) is the number of ways of choosing collections of n objects, which is obviously 1, since there is only one set of n things.
 - P(j, n) is the number of ways of picking sets of j objects from a set of n objects.

2/10/05

6.001 SICP

33/36

Solving the same problem a different way

- So what is the number of ways of picking sets of j objects from a set of n objects?
 - Pick the first one – there are n possible choices
 - Then pick the second one – there are (n-1) choices left.
 - Keep going until you have picked j objects

$$n(n-1)\dots(n-j+1) = \frac{n!}{(n-j)!}$$

- But the order in which we pick the objects doesn't matter, and there are j! different orders, so we have

$$\frac{n!}{(n-j)!j!} = \frac{n(n-1)\dots(n-j+1)}{j(j-1)\dots 1}$$

2/10/05

6.001 SICP

34/36

Solving the same problem a different way

- So here is an easy way to implement this idea:

```
(define pascal
  (lambda (j n)
    (/ (fact n)
       (* (fact (- n j)) (fact j)))))
```

- What is complexity of this approach?
 - Three different evaluations of fact
 - Each is linear in time and in space
 - So combination takes 3n steps, which is also **linear** in time; and has at most n deferred operations, which is also **linear** in space

2/10/05

6.001 SICP

35/36

Solving the same problem a different way

- What about computing with a different version of fact?

```
(define pascal
  (lambda (j n)
    (/ (ifact n)
       (* (ifact (- n j)) (ifact j)))))
```

- What is complexity of this approach?
 - Three different evaluations of fact
 - Each is linear in time and constant in space
 - So combination takes 3n steps, which is also **linear** in time; and has no deferred operations, which is also **constant** in space

2/10/05

6.001 SICP

36/36

Solving the same problem the direct way

$$\frac{n!}{(n-j)!j!} = \frac{n(n-1)\dots(n-j+1)}{j(j-1)\dots 1}$$

- Now, why not just do the computation directly?

```
(define pascal
  (lambda (j n)
    (/ (help n 1 (+ n (- j) 1))
       (help j 1 1))))
(define help
  (lambda (k prod end)
    (if (= k end)
        (* k prod)
        (help (- k 1) (* prod k) end))))
```

2/10/05

6.001 SICP

37/38

Solving the same problem the direct way

- So what is complexity here?
 - Help is an iterative procedure, and has **constant** space and linear time
 - This version of Pascal only uses two versions of help (as opposed the previous version that used three versions of ifact).
 - In practice, this means this version uses fewer multiplies than the previous one, but it is still **linear** in time, and hence has the same order of growth.

2/10/05

6.001 SICP

38/38

So why do these orders of growth matter?

- Main concern is general order of growth
 - Exponential is very expensive as the problem size grows.
 - Some clever thinking can sometimes convert an inefficient approach into a more efficient one.
- In practice, actual performance may improve by considering different variations, even though the overall order of growth stays the same.

2/10/05

6.001 SICP

39/38