Today’s topics

- Orders of growth of processes
- Relating types of procedures to different orders of growth

Computing factorial

```
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))
```

- We can run this for various values of *n:*

```
   (fact 10)
   (fact 100)
   (fact 1000)
   (fact 10000)
```

- Takes longer to run as *n* gets larger, **but** still manageable for large *n* (e.g. *n* = 10000 – takes about 13 seconds of “real time” in DrScheme; while *n* = 1000 – takes about 0.2 seconds of “real time”)

Fibonacci numbers

The Fibonacci numbers are described by the following equations:

$$\mathit{fib}(0) = 0$$

$$\mathit{fib}(1) = 1$$

$$\mathit{fib}(n) = \mathit{fib}(n-2) + \mathit{fib}(n-1) \text{ for } n \geq 2$$

Expanding this sequence, we get

$$\mathit{fib}(0) = 0$$

$$\mathit{fib}(1) = 1$$

$$\mathit{fib}(2) = 1$$

$$\mathit{fib}(3) = 2$$

$$\mathit{fib}(4) = 3$$

$$\mathit{fib}(5) = 5$$

$$\mathit{fib}(6) = 8$$

$$\mathit{fib}(7) = 13$$

...

A contrast to `(fact n)`: computing Fibonacci

```
(define (fib n)
   (if (= n 0)
       0
       (if (= n 1)
           1
           (+ (fib (- n 1)) (fib (- n 2))))))
```

- We can run this for various values of n:

```
      (fib 10)
      (fib 20)
      (fib 100)
      (fib 1000)
```

- These take **much longer** to run as *n* gets larger

A contrast: computing Fibonacci

```
(define (fib n)
   (if (= n 0)
       0
       (if (= n 1)
           1
           (+ (fib (- n 1)) (fib (- n 2))))))
```

- Later we'll see that when calculating `(fib n)`, we need more than $2^{n/2}$ addition operations

`(fib 100)` uses + at least 2^{50} times = 1,125,899,906,842,624

`(fib 2000)` uses + at least 2^{1000} times

=10,715,086,071,862,673,209,484,250,490,600,018,105,614,048,117,055,336,074,437,503,883,703,510,511,249,361,224,931,983,788,156,958,581,275,946,729,175,531,468,251,871,452,856,923,140,435,984,577,574,698,574,803,934,567,774,824,230,985,421,074,605,062,371,141,877,954,182,153,046,474,983,581,941,267,398,767,559,165,543,946,077,062,914,571,196,477,686,542,167,660,429,831,652,624,386,837,205,668,069,376

Computing Fibonacci: putting it in context

- A rough estimate: the universe is approximately 10^{10} years = $3x10^{17}$ seconds old
- Fastest computer around (not your laptop) can do about $280x10^{12}$ arithmetic operations a second, or about 10^{32} operations in the lifetime of the universe
- 2^{100} is roughly 10^{30}
- So with a bit of luck, we could run **`(fib 200)`** in the lifetime of the universe …
- A more precise calculation gives around 1000 hours to solve **`(fib 100)`**
- That is 1000 6.001 lectures, or 40 semesters, or 20 years of 6.001 or …

An overview of this lecture

- Measuring time requirements (**complexity**) of a function
- Simplifying the time complexity with **asymptotic notation**
- Calculating the time complexity for different functions
- Measuring space complexity of a function

Measuring the time complexity of a function

- Suppose *n* is a parameter that measures the size of a problem
    - For **`fact`** and **`fib`**, *n* is just the procedure's parameter
- Let *t(n)* be the amount of time necessary to solve a problem of size *n*
- What do we mean by "the amount of time"? How do we measure "time"?
    - Typically, we will define *t(n)* to be the **number of primitive operations** (e.g. the number of additions) required to solve a problem of size *n*

An example: factorial

```
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))
```

- Define $t(n)$ to be the number of multiplications required by **`(fact n)`**
- By looking at **`fact`**, we can see that:

$$t(0) = 0$$

$$t(n) = 1 + t(n-1) \text{ for } n \geq 1$$

- In other words: solving **`(fact n)`** for any $n \geq 1$ **requires one more multiplication than solving `(fact (- n 1))`**

Expanding the recurrence

$$t(0) = 0$$

$$t(n) = 1 + t(n-1) \text{ for } n>=1$$

$$t(0) = 0$$

$$t(1) = 1 + t(0) = 1$$

$$t(2) = 1 + t(1) = 2$$

$$t(3) = 1 + t(2) = 3$$

…

In general:

$$t(n) = n$$

Expanding the recurrence

$$t(0) = 0$$

$$t(n) = 1 + t(n-1) \text{ for } n>=1$$

- How would we prove that $t(n) = n$ for all n?
- **Proof by induction** (remember from last lecture?):
  - **Base case:** $t(n) = n$ is true for $n = 0$
  - **Inductive step:** if $t(n) = n$ then it follows that

    $$t(n+1) = n+1$$
  - Hence by induction this is true for all n

A second example: Computing Fibonacci

```
(define (fib n)
   (if (= n 0)
       0
       (if (= n 1)
           1
           (+ (fib (- n 1)) (fib (- n 2))))))
```

- Define $t(n)$ to be the number of primitive operations (`=`,`+`,`-`) required by **`(fib n)`**
- By looking at **`fib`**, we can see that:

$$t(0) = 1$$

$$t(1) = 2$$

$$t(n) = 5 + t(n-1) + t(n-2) \text{ for } n \geq 2$$

- In other words: solving **`(fib n)`** for any $n \geq 2$ **requires 5 more primitive ops than solving `(fib (- n 1))` and solving `(fib (- n 2))`**

Looking at the Recurrence

$$t(0) = 1$$
$$t(1) = 2$$
$$t(n) = 5 + t(n-1) + t(n-2) \text{ for } n \geq 2$$

- We can see that $t(n) \geq t(n-1)$ for all $n \geq 2$
- So, for $n \geq 2$, we have

$$\begin{aligned} t(n) &= 5 + t(n-1) + t(n-2) \\ &\geq 2\, t(n-2) \end{aligned}$$

- **Every time *n* increases by 2, we more than double the number of primitive ops that are required**
- If we iterate the argument, we get

$$t(n) \geq 2\, t(n-2) \geq 4\, t(n-4) \geq 8\, t(n-6) \geq 16\, t(n-8) \ldots$$

- A little more math shows that

$$\boldsymbol{t(n) \geq 2^{n/2}}$$

Different Rates of Growth

- So what does it **really mean** for things to grow at different rates?

n	***t(n) = log n* (logarithmic)**	***t(n) = n* (linear)**	***t(n) = n^2* (quadratic)**	***t(n) = n^3* (cubic)**	***t(n) = 2^n* (exponential)**
1	**0**	**1**	**1**	**1**	**2**
10	**3.3**	**10**	**100**	**1000**	**1024**
100	**6.6**	**100**	**10,000**	**10^6**	**~10^30**
1,000	**10.0**	**1,000**	**10^6**	**10^9**	**~10^300**
10,000	**13.3**	**10,000**	**10^9**	**10^12**	**~10^3,000**
100,000	**16.68**	**100,000**	**10^12**	**10^15**	**~10^30,000**

Asymptotic Notation

- Formal definition:

We say ***t(n)*** has **order of growth *$\Theta(f(n))$*** if there are constants ***N***, ***k_1*** and ***k_2*** such that for all ***$n \geq N$***, we have

$$k_1 f(n) \leq t(n) \leq k_2 f(n)$$

- This is what we call a **tight asymptotic bound**.
- Examples

$t(n)=n$ has order of growth $\Theta(n)$

because $1n \leq t(n) \leq 1n$ for all $n \geq 1$ (pick $N=1$, $k_1=1$, $k_2=1$)

$t(n)=8n$ has order of growth $\Theta(n)$

because $8n \leq t(n) \leq 8n$ for all $n \geq 1$ (pick $N=1$, $k_1=8$, $k_2=8$)

Asymptotic Notation

- Formal definition:

We say ***t(n)*** has **order of growth $\Theta(f(n))$** if there are constants N, k_1 and k_2 such that for all $n \geq N$, we have

$$k_1 f(n) \leq t(n) \leq k_2 f(n)$$

- More examples

$t(n)=3n^2$ has order of growth $\Theta(n^2)$

because $3n^2 \leq t(n) \leq 3n^2$ for all $n \geq 1$ (pick N=1, k_1=3, k_2=3)

$t(n)=3n^2+5n+3$ has order of growth $\Theta(n^2)$

because $3n^2 \leq t(n) \leq 4n^2$ for all $n \geq 6$ (pick N=6, k_1=3, k_2=4)

or because $3n^2 \leq t(n) \leq 11n^2$ for all $n \geq 1$ (pick N=1, k_1=3, k_2=11)

Theta, Big-O, Little-o

- $\Theta(f(n))$ is called a tight asymptotic bound because it squeezes $t(n)$ from above and below:
  - **$\Theta(f(n))$** means **$k_1 f(n) \le t(n) \le k_2 f(n)$** “theta”
- We can also talk about the upper bound or lower bound separately
  - **$O(f(n)$** means **$t(n) \le k_2 f(n)$** “big-O”
  - **$\Omega(f(n))$** means **$k_1 f(n) \le t(n)$** “omega”
- Sometimes we will abuse notation and use an upper bound as our approximation
  - We should really use “big-O” notation in that case, saying that $t(n)$ has order of growth **$O(f(n))$**, but we are sometimes sloppy and call this **$\Theta(f(n))$** growth.

Motivation

- In many cases, calculating the precise expression for $t(n)$ is laborious, e.g.:

$$t(n) = 5n^3 + 6n^2 + 8n + 7 \qquad t(n) = 4n^3 + 18n^2 + 14$$

- In both of these cases, $t(n)$ has order of growth $\Theta(n^3)$
- Advantages of asymptotic notation
    - In many cases, it's much easier to show that $t(n)$ has a particular order of growth, e.g., cubic, rather than calculating a precise expression for $t(n)$
    - Usually, the order of growth is **what we really care about:** the most important thing about the above functions is that they are both **cubic** (i.e., have order of growth $\Theta(n^3)$)

Some common orders of growth

$\Theta(1)$	Constant
$\Theta(\log n)$	Logarithmic growth
$\Theta(n)$	Linear growth
$\Theta(n^2)$	Quadratic growth
$\Theta(n^3)$	Cubic growth
$\Theta(2^n)$	Exponential growth
$\Theta(\alpha^n)$	Exponential growth for any $\alpha > 1$

An example: factorial

```
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))
```

- Define $t(n)$ to be the number of multiplications required by `(fact n)`
- By looking at `fact`, we can see that:

  $t(0) = 0$

  $t(n) = 1 + t(n-1)$ for $n >= 1$

- Solving this recurrence gives $t(n) = n$, so order of growth is $\Theta(n)$

A general result: linear growth

For any recurrence of the form

$$t(0) = c_1$$

$$t(n) = c_2 + t(n-1) \text{ for } n \geq 1$$

where c_1 is a constant ≥ 0

and c_2 is a constant > 0

Then we have **linear growth,** i.e.,

$$\Theta(n)$$

Why?

- If we expand this out, we get

$$t(n) = c_1 + nc_2$$

- And this has order of growth $\Theta(n)$

Connecting orders of growth to algorithm design

- We want to compute a^b, just using multiplication and addition
- Remember our stages:
  - Wishful thinking
  - Decomposition
  - Smallest sized subproblem

Connecting orders of growth to algorithm design

- Wishful thinking
  - Assume that the procedure **`my-expt`** exists, but only solves smaller versions of the same problem
- Decompose problem into solving smaller version and using result

$$a^n = a \cdot a \cdots a = a \cdot a^{n-1}$$

```
(define my-expt
   (lambda (a n)
      (* a (my-expt a (- n 1)))))
```

Connecting orders of growth to algorithm design

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

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

The order of growth of `my-expt`

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

- Define the size of the problem to be *n* (the second parameter)
- Define *t(n)* to be the number of primitive operations required (`=`,`*`,`-`)
- By looking at the code, we can see that *t(n)* has the form:

$$t(0) = 1$$

$$t(n) = 3 + t(n-1) \text{ for } n \geq 1$$

- Hence this is also **linear**

Using different processes for the same goal

- Are there other ways to decompose this problem?
- We can take advantage of the following trick:

$$a^n = (a \cdot a)^{\frac{n}{2}}$$

```
(define (new-expt a n)
  (cond ((= n 0)  1)
        ((even? n) (new-expt (* a a) (/ n 2)))
        (else (* a (new-expt a (- n 1))))))
```

New special form:

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

The order of growth of `new-expt`

```
(define (new-expt a n)
  (cond ((= n 0)  1)
        ((even? n) (new-expt (* a a) (/ n 2)))
        (else (* a (new-expt a (- n 1))))))
```

- If *n* is even, then 1 step reduces to *n/2* sized problem
- If *n* is odd, then 2 steps reduces to *n/2* sized problem
- Thus in at most *2k* steps, reduces to *n/2^k* sized problem
- We are done when problem size is just 1, which implies order of growth in time of

$$\Theta(\log n)$$

The order of growth of `new-expt`

```
(define (new-expt a n)
  (cond ((= n 0)  1)
        ((even? n) (new-expt (* a a) (/ n 2)))
        (else (* a (new-expt a (- n 1))))))
```

- *t(n)* has the following form:

$$t(0) = 1$$

$$t(n) = 4 + t(n/2) \text{ if } n \text{ is even}$$

$$t(n) = 4 + t(n-1) \text{ if } n \text{ is odd}$$

- It follows that

$$t(n) = 8 + t((n-1)/2) \text{ if } n \text{ is odd}$$

A general result: logarithmic growth

For any recurrence of the form

$$t(0) = c_1$$

$$t(n) = c_2 + t(n/2) \text{ for } n \geq 1$$

where c_1 is a constant ≥ 0

and c_2 is a constant > 0

Then we have **logarithmic growth,** i.e.,

$$\boldsymbol{\Theta(\log n)}$$

- Intuition: at each step we **halve the size of the problem**
- We can only halve *n* around *log n* times before we reach the base case (e.g. $n=1$ or $n=0$)

Different Rates of Growth

- Note why this makes a difference

n	*t(n) = log n* (logarithmic)	*t(n) = n* (linear)	*t(n) = n^2* (quadratic)	*t(n) = n^3* (cubic)	*t(n) = 2^n* (exponential)
1	0	1	1	1	2
10	3.3	10	100	1000	1024
100	6.6	100	10,000	10^6	1.3 x 10^30
1,000	10.0	1,000	10^6	10^9	1.1 x 10^300
10,000	13.3	10,000	10^9	10^12	---
100,000	16.68	100,000	10^12	10^15	---

Back to Fibonacci

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

- If t(n) is defined as the number of primitive operations (`=`,`+`,`-`), then:

$$t(0) = 1$$

$$t(1) = 2$$

$$t(n) = 5 + t(n-1) + t(n-2) \text{ for } n \geq 2$$

- And for *n* ≥ *2* we have

$$t(n) \geq 2t(n-2)$$

Another general result: exponential growth

- If we can show:

$$t(0) = c_1$$

$$t(n) \ge c_2 + \alpha t(n - \beta) \text{ for } n \ge 1$$

with constants $c_1 \ge 0$, $c_2 > 0$,
and constant $\alpha > 1$
and constant $\beta \ge 1$

Then we have **exponential growth,** i.e.,

$$\boldsymbol{\Omega(\alpha^{n/\beta})}$$

- Intuition: Every time we **add** β to the problem size *n,* the amount of computation required is **multiplied** by a factor of α.

Why is our version of `fib` so inefficient?

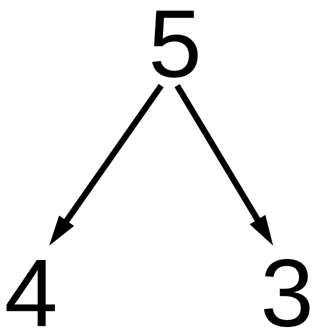
```
(define fib
    (lambda (n)
       (cond ((= n 0) 0)
             ((= n 1) 1)
             (else (+ (fib (- n 1))
                      (fib (- n 2)))))))
```

- When computing **`(fib 6)`**, the recursion computes **`(fib 5)`** and **`(fib 4)`**
- The computation of **`(fib 5)`** then involves computing **`(fib 4)`** and **`(fib 3)`**. At this point **`(fib 4)`** has been computed **twice.** Isn't this wasteful?

Why is our version of `fib` so inefficient?

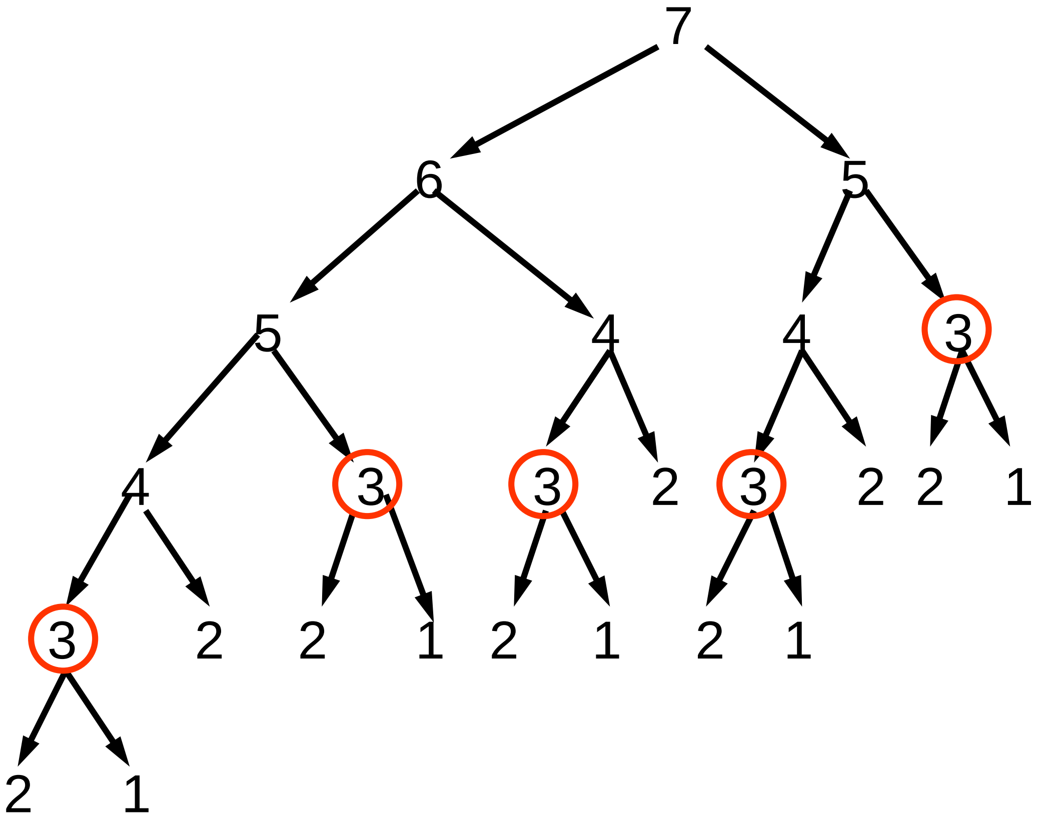
- Let’s draw the **computation tree:** the subproblems that each (fib n) needs to call

- We’ll use the notation



…to signify that computing **`(fib 5)`** involves recursive calls to **`(fib 4)`** and **`(fib 3)`**

The computation tree for `(fib 7)`



- There’s a lot of repeated computation here: e.g., `(fib 3)` is recomputed 5 times

An efficient implementation of Fibonacci

```
(define (ifib n) (fib-iter 0 1 0 n))

(define (fib-iter i a b n)
  (if (= i n)
      b
      (fib-iter (+ i 1) (+ a b) a n)))
```

- Recurrence (measured in number of primitive operations):

$$t(0) = 1$$

$$t(n) = 3 + t(n-1) \text{ for } n \geq 1$$

- Order of growth is

$$\Theta(n)$$

`ifib` is now linear

- If you trace the function, you will see that we avoid repeated computations. We’ve gone from exponential growth to linear growth!

```
(ifib 5)
(fib-iter 0 1 0 5)
(fib-iter 1 1 1 5)
(fib-iter 2 2 1 5)
(fib-iter 3 3 2 5)
(fib-iter 4 5 3 5)
(fib-iter 5 8 5 5)
5
```

How much space (memory) does a procedure require?

- So far, we have considered the order of growth of *t(n)* for various procedures. *T(n)* is the **time** for the procedure to run, when given an input of size *n.*
- Now, let's define *s(n)* to be the **space** or **memory** requirements of a procedure when the problem size is *n.* What is the order of growth of *s(n)*?
- Note that for now we will measure space requirements in terms of the **maximum number of pending operations**.

Tracing factorial

```
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))
```

- A trace of `fact` shows that it leads to a recursive process, with **pending operations.**

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 (* 2 1)))
…
24
```

Tracing factorial

- In general, running **`(fact n)`** leads to *n* pending operations

- Each pending operation takes a constant amount of memory

- In this case, *s(n)* has order of growth that is linear in space:

$$\Theta(n)$$

A contrast: iterative factorial

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

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

A contrast: iterative factorial

- A trace of `(ifact 4):`

```
(ifact 4)
(ifact-helper 1 1 4)
(ifact-helper 1 2 4)
(ifact-helper 2 3 4)
(ifact-helper 6 4 4)
(ifact-helper 24 5 4)
24
```

- `(ifact n)` has no pending operations, so *s(n)* has an order of growth that is constant $\Theta(1)$

- Its time complexity *t(n)* is $\Theta(n)$
- In contrast, `(fact n)` has linear growth in both space and time $\Theta(n)$
- In general, *iterative processes* often have a lower order of growth for *s(n)* than *recursive processes*

Summary

- We’ve described how to calculate *t(n)*, the time complexity of a procedure as a function of the size of its input
- We’ve introduced asymptotic notation for orders of growth
- There is a **huge** difference between exponential order of growth and non-exponential growth, e.g., if your procedure has

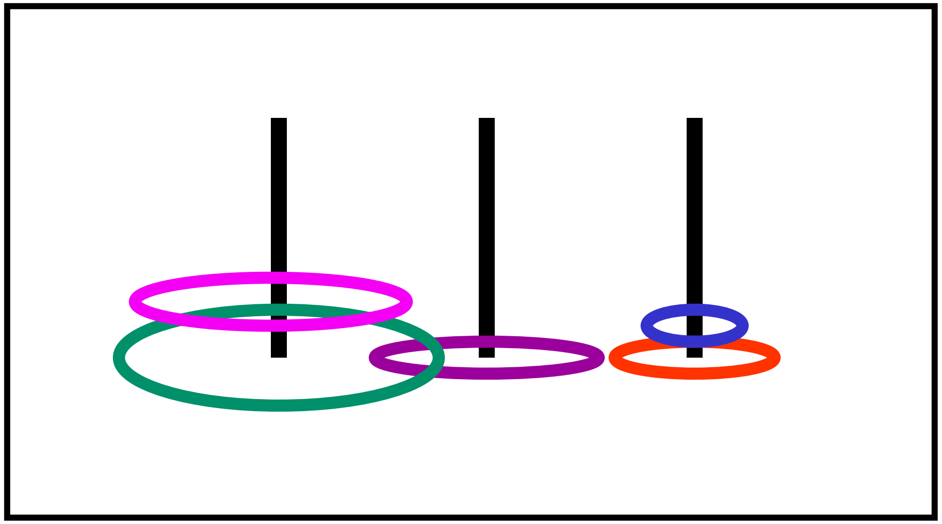
$$t(n) = \Theta(2^n)$$

You will not be able to run it for large values of *n.*

- We’ve given examples of procedures with linear, logarithmic, and exponential growth for *t(n).* Main point: you should be able to work out the order of growth of *t(n)* for simple procedures in Scheme
- The space requirements *s(n)* for a procedure depend on the number of pending operations. Iterative processes tend to have fewer pending operations than their corresponding recursive processes.

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

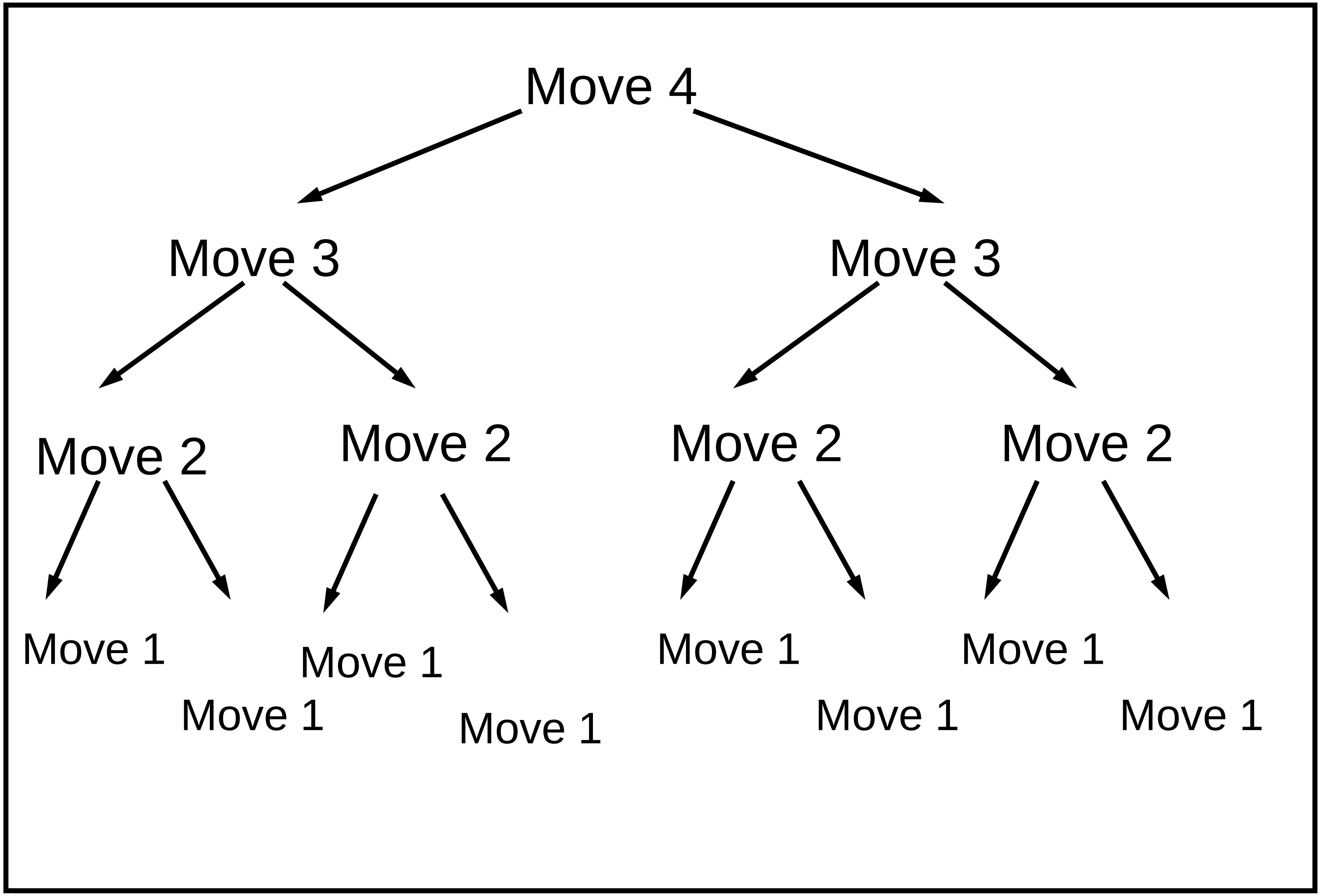


Towers of Hanoi

```
(define move-tower
    (lambda (size from to extra)
       (cond ((= size 0) true)
             (else (move-tower (- size 1) from extra to)
                   (print-move from to)
                   (move-tower (- size 1) extra to from)))))

(define print-move
     (lambda (from to)
       (display ``Move top disk from ``)
       (display from)
       (display `` to ``)
       (display to)
       (newline)))
```

A tree recursion



Move 4

Move 3
Move 3

Move 2
Move 2
Move 2
Move 2

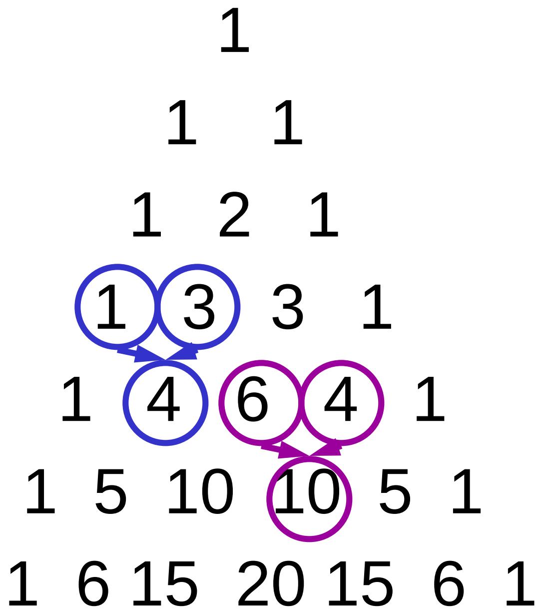
Move 1
Move 1
Move 1
Move 1
Move 1
Move 1
Move 1
Move 1

Orders of growth for towers of Hanoi

- What is the order of growth in time for towers of Hanoi?
- What is the order of growth in space for towers of Hanoi?

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

Pascal’s triangle

- We need some notation
  - Let’s order the rows, starting with n=0 for the first row
  - The nth row then has n+1 elements
  - Let’s use P(j,n) to denote the jth element of the nth row.
  - We want to find ways to compute P(j,n) for any n, and any j, such that 0 <= j <= n

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, and the (k-1)’st and k’th element of the previous row

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)))))))
```

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

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.

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

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

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

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

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

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

Solving the same problem the direct way

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

- 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))))
```

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 that the previous one, but it is still linear in time, and hence has the same order of growth.

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.