6.001: Lecture 4
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., $n = 10000$)

Fibonacci Numbers

The Fibonacci numbers are described by the following equations:

$$\begin{aligned} fib(1) &= 1 \\ fib(2) &= 1 \\ fib(n) &= fib(n-2) + fib(n-1) \text{ for } n \geq 3 \end{aligned}$$

Expanding this sequence, we get

$$\begin{aligned} fib(1) &= 1 \\ fib(2) &= 1 \\ fib(3) &= 2 \\ fib(4) &= 3 \\ fib(5) &= 5 \\ fib(6) &= 8 \\ fib(7) &= 13 \\ &\ldots \end{aligned}$$

A Contrast to `(fact n)`: Computing Fibonacci

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

- We can run this for various values of n:

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

- Takes **much** longer to run as n gets larger

A Contrast: Computing Fibonacci

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

- Later we'll see that when calculating `(fib n)`, we need more than $2^{\frac{n}{2}}$ addition operations
- For example, to calculate `(fib 100)`, we need to use `+` at least $2^{50} = 1125899906842624$ times
- For example, to calculate `(fib 2000)`, we need to use `+` at least $2^{1000} =$

1071508607186267320948425049060001810561
4048117055336074437503883703510511249361
2249319837881569585812759467291755314682
5187145285692314043598457757469857480393
4567774824230985421074605062371141877954
1821530464749835819412673987675591655439
4607706291457119647768654216766042983165
2624386837205668069376
times

A Contrast: Computing Fibonacci

- A rough estimate: the universe is approximately 10^{10}years $=$ 3×10^{17}seconds old
- Fastest computer around can do $\approx 250 \times 10^{12}$ arithmetic operations a second, or $\approx 10^{30}$ operations in the lifetime of the universe
- $2^{100} \approx 10^{30}$
- 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's 1000 6.001 lectures, or 40 semesters, or 20 years of 6.001...

An Overview of This Lecture

- Measuring time requirements of a function
- Asymptotic notation
- Calculating the time complexity for different functions
- Measuring space requirements of a function

Measuring the Time Complexity of a Function

- Suppose n is a parameter that measures the size of a problem
- 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'll define $t(n)$ to be the **number of primitive arithmetic 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:

$$\begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

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

Expanding the Recurrence

$$\begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

$$\begin{aligned} t(0) &= 0 \\ t(1) &= 1 + t(0) = 1 \\ t(2) &= 1 + t(1) = 2 \\ t(3) &= 1 + t(2) = 3 \\ &\ldots \end{aligned}$$

In general:

$$\boxed{t(n) = n}$$

Expanding the Recurrence

$$\begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

- How would we prove that $t(n) = n$ for all n?
- **Proof by induction** (see the last lecture):
  - Base case: $t(n) = n$ is true for $n = 0$
  - Inductive case: if $t(n) = n$ then it follows that $t(n+1) = n + 1$

A Second Example: Computing Fibonacci

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

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

$$\begin{aligned} t(1) &= 0 \\ t(2) &= 0 \\ t(n) &= 1 + t(n-1) + t(n-2) \text{ for } n \geq 3 \end{aligned}$$

- In other words: solving `(fib n)` for any $n \geq 3$ **requires one more addition than solving** `(fib (- n 1))` *and* solving `(fib (- n 2))`

Looking at the Recurrence

$$\begin{aligned} t(1) &= 0 \\ t(2) &= 0 \\ t(n) &= 1 + t(n-1) + t(n-2) \text{ for } n \geq 3 \end{aligned}$$

- We can see that $t(n) \geq t(n-1)$ for all n
- So, for $n \geq 3$ we have

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

- **Every time n increases by 2, we more than double the number of additions that are required**

- If we iterate the argument, we get

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

- A little more math shows that

$$t(n) \geq 2^{\frac{n}{2}} = (\sqrt{2})^n$$

Different Rates of Growth

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	1.3×10^{30}
1,000	10.0	1,000	10^6	10^9	1.1×10^{300}
10,000	13.3	10,000	10^9	10^{12}	—
100,000	16.68	100,000	10^{12}	10^{15}	—

Aysmptotic Notation

- Formal definition:

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

Examples

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

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

  for all $n \geq k$ if we pick $k = k_1 = k_2 = 1$

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

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

  for all $n \geq k$ if we pick $k = 1$, and $k_1 = k_2 = 8$

Examples

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

  $$k_1 \times n^2 \leq t(n) \leq k_2 \times n^2$$

  for all $n \geq k$ if we pick $k = 1$, and $k_1 = k_2 = 3$

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

  $$k_1 \times n^2 \leq t(n) \leq k_2 \times n^2$$

  for all $n \geq k$ if we pick $k = 5$, $k_1 = 3$, and $k_2 = 8$

Motivation

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

  $$t(n) = 5n^3 + 6n^2 + 8n + 7 \quad \text{or} \quad t(n) = 4n^3 + 18n^2 + 14$$

- In both of these cases, $t(n)$ has order of growth $\Theta(n^3)$
- Advantages of asymptotic notation:
  - In many cases, it's much easy to show that $t(n)$ has a particular order of growth (e.g., $\Theta(n^3)$), 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're 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)
- $\Theta(\alpha^n)$ for any $\alpha > 1$ (exponential growth)

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:

  $$\begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

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

A General Result

- For any recurrence of the form

$$\begin{aligned} t(0) &= c_1 \\ t(n) &= c_2 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

  where c_1 is a constant that is ≥ 0,
  and c_2 is a constant that is > 0,
  we have **linear growth** (i.e., $\Theta(n)$)

- Why? If we expand this out we get

$$t(n) = c_1 + n \times c_2$$

  which has order of growth $\Theta(n)$

Another Example of Linear Growth

```
(define (exp a n)
   (if (= n 0)
       1
       (* a (exp a (- n 1)))))
```

- `(exp a n)` calculates a raised to the power n (e.g., `(exp 2 3)` has the value 8)
- Define the size of the problem to be n (the second parameter) define $t(n)$ to be the number of arithmetic operations required ($=$, $*$ or $+$)
- By looking at `exp`, we can see that $t(n)$ has the form:

$$\begin{aligned} t(0) &= 1 \\ t(n) &= 2 + t(n-1) \text{ for } n \geq 1 \end{aligned}$$

A More Efficient version of `(exp a n)`

```
(define (exp2 a n)
   (if (= n 0)
       1
       (if (even? n)
           (exp2 (* a a) (/ n 2))
           (* a (exp2 a (- n 1))))))
```

- This makes use of the trick

$$a^b = (a \times a)^{\frac{b}{2}}$$

The Order of Growth of `(exp2 a n)`

```
(define (exp2 a n)
   (if (= n 0)
       1
       (if (even? n)
           (exp2 (* a a) (/ n 2))
           (* a (exp2 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 $2k$ 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 `(exp2 a n)`

```
(define (exp2 a n)
   (if (= n 0)
       1
       (if (even? n)
           (exp2 (* a a) (/ n 2))
           (* a (exp2 a (- n 1))))))
```

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

$$\begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n/2) \text{ if } n \text{ is even} \\ t(n) &= 1 + t(n-1) \text{ if } n \text{ is odd} \end{aligned}$$

- It follows that $t(n) = 2 + t((n-1)/2)$ if n is odd

Another General Result

- For any recurrence of the form

$$\begin{aligned} t(0) &= c_1 \\ t(n) &= c_2 + t(n/2) \text{ for } n \geq 1 \end{aligned}$$

  where c_1 is a constant that is ≥ 0,
  and c_2 is a constant that is > 0,
  we have **logarithmic growth** (i.e., $\Theta(\log n)$)

- 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., $n = 0$)

Different Rates of Growth

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	1.3×10^{30}
1,000	10.0	1,000	10^6	10^9	1.1×10^{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 n)
   (if (= n 1)
       1
       (if (= n 2)
           1
           (+ (fib (- n 1)) (fib (- n 2))))))
```

- By looking at `fib`, we can see that:

$$\begin{aligned} t(1) &= 0 \\ t(2) &= 0 \\ t(n) &= 1 + t(n-1) + t(n-2) \text{ for } n \geq 3 \end{aligned}$$

  and for $n \geq 3$ we have

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

A General Result

- If we can show

$$\begin{aligned} t(0) &= c_1 \\ t(n) &\geq c_2 + \alpha \times t(n-\beta) \text{ for } n \geq 1 \end{aligned}$$

  where $c_1 \geq 0$, $c_2 > 0$
  α is a constant that is > 1
  β is an integer that is ≥ 1
  we get exponential growth

- Intuition? Every time we **add** β to the problem size n, the amount of computation required is **multiplied** by a factor of α that is greater than 1

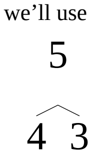
Why is Our Version of `fib` so Inefficient?

```
(define (fib n)
   (if (= n 1)
       1
       (if (= n 2)
           1
           (+ (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?!

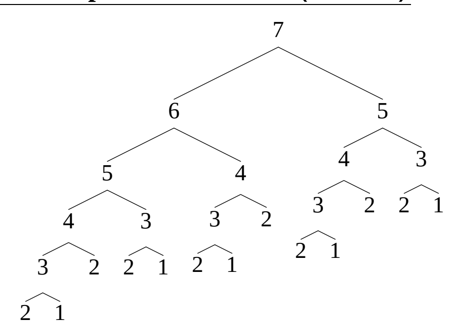
Why is Our Version of `fib` so Inefficient?

- A **Computation tree:** we'll use



  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 (fib2 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 ($t(n)$ is number of additions):

$$
\begin{aligned}
t(0) &= 0 \\
t(n) &= 2 + t(n-1) \text{ for } n \geq 1
\end{aligned}
$$

- Order of growth of $t(n)$ is $\Theta(n)$

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

```
(fib2 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've considered the order of growth of $t(n)$ for various functions. $t(n)$ 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 $s(n)$?

Tracing Factorial

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

- A trace of `fact`, showing 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 $\Theta(n)$
  i.e., linear growth in space

A Contrast: Iterative Factorial

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

(define (ifact-helper product counter n)
  (if (> counter n)
      product
      (ifact-helper (* product counter)
                    (+ counter 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 $\Theta(1)$. Its time complexity $t(n)$ is $\Theta(n)$
- In contrast, `(fact n)` has $t(n) = \Theta(n)$ and $s(n) = \Theta(n)$, i.e., linear growth in both space and time
- In general, *iterative processes* often have a lower order of growth for $s(n)$ than *recursive processes*

Summary

- We've describe 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 (e.g., $\Theta(n)$, $\Theta(n^2)$)
- There is a **huge** difference between exponential order of growth and non-exponential growth (e.g., if your procedure $t(n) = \Theta(2^n)$, you will not be able to run it for large values of n)
- We've given examples of functions 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 function depend on the number of pending operations. Iterative processes tend to have fewer pending operations.