Today's topics

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

Computing factorial

```scheme
(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 .2 seconds of "real time")

Fibonacci numbers

The Fibonacci numbers are described by the following equations:

- `fib(1) = 1`
- `fib(2) = 1`
- `fib(n) = fib(n-2) + fib(n-1)` for `n >= 3`

Expanding this sequence, we get

- `fib(1) = 1`
- `fib(2) = 1`
- `fib(3) = 2`
- `fib(4) = 3`
- `fib(5) = 5`
- `fib(6) = 8`
- `fib(7) = 13`
- ...

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

```scheme
(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)`

- These take much longer to run as n gets larger

A contrast: Computing Fibonacci

```scheme
(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 $2^n$ addition operations
- For example, to calculate `(fib 100)`, we need to use + at least $2^{100}$ times (or $1.25899006842624 	imes 10^{30}$)
- For example, to calculate `(fib 2000)`, we need to use + at least $2^{2000}$ times (or $5.54 	imes 10^{607}$ times)
A contrast: Computing Fibonacci

- A rough estimate: the universe is approximately $10^{10}$ years = $3 \times 10^{17}$ seconds old
- Fastest computer around can do about $250 \times 10^{12}$ arithmetic operations a second, or about $10^{30}$ operations in the lifetime of the universe
- $2^{100}$ is roughly $10^{30}$
- 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 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 will 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

\begin{verbatim}
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
\end{verbatim}

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

Expanding the recurrence

\[
t(0) = 0 \\
t(n) = 1 + t(n-1) \text{ for } n \geq 1
\]

- 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

\[
\text{define (fib n)} \\
\text{ (if (= n 1) 1) \text{ if (= n 2) 1))}}
\]

• Define \(t(n)\) to be the number of additions required by \(\text{fib n}\)

• By looking at \(\text{fib}\), we can see that:

\[
t(1) = 0 \\
t(2) = 0 \\
t(3) = 1 + t(n-1) + t(n-2) \text{ for } n \geq 3
\]

• In other words: solving \(\text{fib n}\) for any \(n \geq 3\) requires one more addition than solving \(\text{fib (n-1)}\) and solving \(\text{fib (n-2)}\)

Looking at the Recurrence

\[
t(1) = 0 \\
t(2) = 0 \\
t(3) = 1 + t(n-1) + t(n-2) \text{ for } n \geq 3
\]

• We can see that \(t(n) \geq t(n-1)\) for all \(n\)

• So, for \(n \geq 3\), we have

\[
t(n) = 1 + t(n-1) + t(n-2) \geq 2t(n-2)
\]

• Every time \(n\) increases by 2, we more than double the number of additions that are required

• A little more math shows that

\[
t(n) \geq \frac{2^n}{4}\]

Different Rates of Growth

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

<table>
<thead>
<tr>
<th>(n)</th>
<th>(\log n) (logarithmic)</th>
<th>(n) (linear)</th>
<th>(n^2) (quadratic)</th>
<th>(n^3) (cubic)</th>
<th>(2^n) (exponential)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>1024</td>
</tr>
<tr>
<td>100</td>
<td>6.6</td>
<td>100</td>
<td>10,000</td>
<td>10^6</td>
<td>1.3 x 10^30</td>
</tr>
<tr>
<td>1,000</td>
<td>10.0</td>
<td>1,000</td>
<td>10^6</td>
<td>10^9</td>
<td>1.1 x 10^300</td>
</tr>
<tr>
<td>10,000</td>
<td>13.3</td>
<td>10,000</td>
<td>10^9</td>
<td>10^12</td>
<td>---</td>
</tr>
<tr>
<td>100,000</td>
<td>16.68</td>
<td>100,000</td>
<td>10^12</td>
<td>10^15</td>
<td>---</td>
</tr>
</tbody>
</table>

Motivation

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

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

• 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

```scheme
(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 $n > 0$

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

A General Result

• For any recurrence of the form

$$
t(0) = c_1 \\
t(n) = c_1 + t(n-1) \text{ for } n \geq 1
$$

Where $c_1$ is a constant that is $\geq 0$
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
    $$t(n) = c_1 + n \times c_2$$
  - 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

• Identify smallest size subproblem

```scheme
(a^n = a*a*...*a = a*a^(n-1))
```

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

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
  - $a^n = a\times a\times...\times a = a\times a^{(n-1)}$

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

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

```scheme
(define my-expt
  (lambda (a n)
    (if (= n 0)
      1
      (* a (my-expt a (- n 1))))))
```
Connecting orders of growth to algorithm design

\[
\text{(define my-expt}
\begin{align*}
& \text{(lambda (a n))} \\
& \text{(if (= n 0)} \
& \quad \text{1} \\
& \quad (* a (my-expt a (- n 1)))))
\end{align*}
\]

- Define the size of the problem to be \(n\) (the second parameter) and define \(t(n)\) to be the number of arithmetic operations required (\(*\) or \(+\))
- By looking at the code, we can see that \(t(n)\) has the form:

\[
t(0) = 1
\]

\[
t(n) = 2 + t(n-1) \text{ for } n \geq 1
\]

- 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:

\[
\text{(define (new-exp a n))}
\begin{align*}
& \text{(cond ((= n 0)} \
& \quad \text{1} \\
& \quad ((\text{even? n)} \text{ (new-exp (* a a) (/ n 2)))} \\
& \quad \text{(else (* a (new-exp a (- n 1))))})
\end{align*}
\]

The Order of Growth of \((\text{new-exp a n})\)

\[
\text{(define (new-exp a n)}
\begin{align*}
& \text{(cond ((= n 0)} \
& \quad \text{1} \\
& \quad ((\text{even? n)} \text{ (new-exp (* a a) (/ n 2)))} \\
& \quad \text{(else (* a (new-exp a (- n 1))))})
\end{align*}
\]

• 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

\[
\Theta(\log n)
\]

Another General Result

• For any recurrence of the form:

\[
t(0) = c_0
\]

\[
t(n) = c_n + t(n/2) \text{ for } n \geq 1
\]

Where \(c_n\) is a constant that is \(\geq 0\)
And \(c_0\) is a constant that is \(> 0\)

We have \(\text{logarithmic growth, i.e., }\)

\[
\Theta(\log n)
\]

Different Rates of Growth

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
n & t(n) = \log n & t(n) = n & t(n) = n^2 & t(n) = n^3 & t(n) = 2^n \\
\hline
1 & 0 & 1 & 1 & 1 & 2 \\
10 & 3.3 & 10 & 100 & 1000 & 1024 \\
100 & 6.6 & 100 & 10,000 & 10^6 & 1.3 \times 10^{10} \\
1,000 & 10.0 & 1,000 & 10^6 & 10^8 & 10^9 \\
10,000 & 13.3 & 10,000 & 10^9 & 10^{10} & 1.1 \times 10^{12} \\
100,000 & 16.68 & 100,000 & 10^{10} & 10^{11} & 10^{15} \\
\hline
\end{array}
\]
Back to Fibonacci

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

• By looking at this code, we can see that
  \[ t(0) = 0 \]
  \[ t(1) = 0 \]
  \[ t(n) = t(n-1) + t(n-2) \text{ for } n \geq 2 \]
• And for \( n \geq 3 \) we have
  \[ t(n) \geq 2t(n-2) \]

Yet Another General Result

• If we can show:
  \[ t(0) = c_1 \]
  \[ t(n) \geq c_1 + c \alpha \times t(n - \beta) \text{ for } n \geq 1 \]

Where \( c_1 \geq 0, c > 0, \alpha > 1, \) \( \beta \) is an integer that is \( \geq 1 \)

We have exponential growth

• Intuition: Every time we add \( \beta \) to the problem size \( n \), the amount of computation required is multiplied by a factor of \( \alpha \) that is greater than 1.

Why is our version of \( \text{fib} \) so inefficient?

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

• When computing \( \text{fib} 6 \), the recursion computes \( \text{fib} 5 \) and \( \text{fib} 4 \)

• The computation of \( \text{fib} 5 \) then involves computing \( \text{fib} 4 \) and \( \text{fib} 3 \). At this point \( \text{fib} 4 \) has been computed twice. Isn’t this wasteful?

Why is our version of \( \text{fib} \) so inefficient?

• A Computation tree:

- We’ll use the notation

  \[
  \begin{array}{c}
  5 \\
  4 \\
  3 \\
  2 \\
  1 \\
  \end{array}
  \]

  to signify that computing \( \text{fib} 5 \) involves recursive calls to \( \text{fib} 4 \) and \( \text{fib} 3 \)

The computation tree for \( \text{fib} 7 \)

• There’s a lot of repeated computation here: e.g., \( \text{fib} 3 \) 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 (measured in number of additions):
  \[ t(0) = 0 \]
  \[ t(n) = 2 + t(n - 1) \text{ for } n \geq 1 \]
• Order of growth is \( \Theta(n) \)
• If you trace the function, you will 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 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 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

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))))

• 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

• A trace of (ifact 4):

(ifact 4)
(ifact-helper 1 1 4)
(ifact-helper 1 2 4)
(ifact-helper 2 3 4)
(ifact-helper 4 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
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

A tree recursion

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?

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 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 than their corresponding recursive processes.