Structure and Interpretation of Computer Programs

COMP200
• Rules for evaluation
• Orders of growth of processes
• Relating types of procedures to different orders of growth
SCHEME
Rules for Evaluation

• Elementary expressions are left alone.
SCHEME
Rules for Evaluation

• Elementary expressions are left alone.
• A pairing by `define`: re-write the name as the value
SCHEME
Rules for Evaluation

• Elementary expressions are left alone.
• A pairing by define: re-write the name as the value
• Special form if
  • If predicate is true, re-write it as the consequence
  • Otherwise, re-write it as the alternative
SCHEME

Rules for Evaluation

• Elementary expressions are left alone.

• A pairing by define: re-write the name as the value

• Special form if
  • If predicate is true, re-write it as the consequence
  • Otherwise, re-write it as the alternative

• Combination:
  • operator <=> procedure, operands <=> arguments
SCHEME
Rules for Evaluation

- Elementary expressions are left alone.
- A pairing by define: re-write the name as the value
- Special form if
  - If predicate is true, re-write it as the consequence
  - Otherwise, re-write it as the alternative
- Combination:
  - operator <=> procedure, operands <=> arguments
  - a primitive procedure vs. a compound procedure
SCHEME

Orders of Growth of Processes
• Let
  
  • \(n\) be a parameter to measure the size of a problem.
SCHEME
Orders of Growth of Processes

• Let

  • \( n \) be a parameter to measure the size of a problem.
  
  • \( R(n) \) be the amount of resources needed to compute a procedure of size \( n \).
Let

- $n$ be a parameter to measure the size of a problem.
- $R(n)$ be the amount of resources needed to compute a procedure of size $n$.
- $R(n)$ has order of growth $Q(f(n))$ 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 $n$. 
Two common resources are

- **space**: measured by the number of deferred operations
- **time**: measured by the number of primitive steps
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))

(fact 3)
  (if (= 3 1) 1 (* 3 (fact (- 3 1))))
  (if #false 1 (* 3 (fact (- 3 1))))
  (* 3 (fact (- 3 1)))
  (* 3 (fact 2))
  (* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
  (* 3 (if #false 1 (* 2 (fact (- 2 1)))))
  (* 3 (* 2 (fact (- 2 1))))
  (* 3 (* 2 (fact 1)))
  (* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1)))))
  (* 3 (* 2 (if #true 1 (* 1 (fact (- 1 1)))))
  (* 3 (* 2 1))
  (* 3 2)
  6
(define ifact
  (lambda (n)(ifact-helper 1 1 n)))

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 1)n)))))
(define ifact
  (lambda (n)(ifact-helper 1 1 n)))
(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
        product
        (ifact-helper(* product counter)(+ counter 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
ORDERS OF GROWTH

Examples

• fact
  • space: $\Theta(n)$ - linear
  • time: $\Theta(n)$ - linear
ORDERS OF GROWTH

Examples

• fact
  • space: $\Theta(n)$ - linear
  • time: $\Theta(n)$ - linear

• ifact
  • space: $\Theta(1)$ - constant
  • time: $\Theta(n)$ - linear
SUMMARY

What have we learned?

• Why do these orders of growth matter?
$F(n) = 0$ if $n = 0$

$F(n) = 1$ if $n = 1$

$F(n) = F(n - 1) + F(n - 2)$ otherwise
SPECIAL FORMS

cond

(cond (<predicate1> <consequent> <consequent>) ...)
((<predicate2> <consequent> <consequent>) ...)
...
(else <consequent> <consequent>)
\[ F(n) = 0 \text{ if } n = 0 \]
\[ F(n) = 1 \text{ if } n = 1 \]
\[ F(n) = F(n - 1) + F(n - 2) \text{ otherwise} \]

\[
\begin{align*}
\text{(cond } &<\text{predicate1}> \ <\text{consequent}> \ <\text{consequent}> \text{)} \ldots \text{) } \\
\text{(cond } &<\text{predicate2}> \ <\text{consequent}> \ <\text{consequent}> \text{)} \ldots \text{) } \\
\ldots \text{) } \\
\text{(else } &<\text{consequent}> \ <\text{consequent}> \text{)}
\end{align*}
\]
$F(n) = 0$ if \( n = 0 \)

$F(n) = 1$ if \( n = 1 \)

$F(n) = F(n-1) + F(n-2)$ otherwise

(\texttt{define fib} \\
(\texttt{lambda} \ (n))

(\texttt{cond} \ (\texttt{<predicate1>} \ \texttt{<consequent>} \ \texttt{<consequent>} \ ...) \\
(\texttt{<predicate2>} \ \texttt{<consequent>} \ \texttt{<consequent>} \ ...) \\
... \\
(\texttt{else} \ \texttt{<consequent>} \ \texttt{<consequent>}))
$F(n) = 0$ if $n = 0$

$F(n) = 1$ if $n = 1$

$F(n) = F(n-1) + F(n-2)$ otherwise

\[\text{(define fib (lambda (n)}\]

\[
\begin{aligned}
&\text{(cond (}\langle\text{predicate1}\rangle\text{ }\langle\text{consequent}\rangle\text{ }\langle\text{consequent}\rangle)\text{ ...)}\\
&\text{(}\langle\text{predicate2}\rangle\text{ }\langle\text{consequent}\rangle\text{ }\langle\text{consequent}\rangle)\text{ ...)}\\
&\text{... }\text{...)}\\
&\text{(else }\langle\text{consequent}\rangle\text{ }\langle\text{consequent}\rangle)\end{aligned}
\]
ANOTHER EXAMPLE

Fibonacci

\[ F(n) = \begin{cases} 
0 & \text{if } n = 0 \\ 
1 & \text{if } n = 1 \\ 
F(n-1) + F(n-2) & \text{otherwise} 
\end{cases} \]

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

(cond (<predicate1> <consequent> <consequent> ...) ...)
  (<predicate2> <consequent> <consequent> ...) ...)
  ...
  (else <consequent> <consequent>))
ANOTHER EXAMPLE

Fibonacci

\[ F(n) = 0 \text{ if } n = 0 \]
\[ F(n) = 1 \text{ if } n = 1 \]
\[ F(n) = F(n - 1) + F(n - 2) \text{ otherwise} \]

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

Fibonacci

```
(fib 4)
  /   
(fib 3)     (fib 2)
  /     /   
(fib 2)   (fib 1) (fib 1)
     /       /     
(fib 1) (fib 0) (fib 0)
```
Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:
ORDERS OF GROWTH

Fibonacci

Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:

$$t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}$$
Let \( t(n) \) be the number of steps that we need to take to solve the case for size \( n \).

Then:

\[
    t(n) = t_{n-1} + t_{n-2} = 2 \cdot t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}
\]
ORDERS OF GROWTH

Fibonacci

Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:

$$t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}$$
ORDERS OF GROWTH

Fibonacci

Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:

$$t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}$$
ORDERS OF GROWTH

Fibonacci

Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:

$$t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}$$
ORDERS OF GROWTH

Fibonacci

Let $t(n)$ be the number of steps that we need to take to solve the case for size $n$.

Then:

$$t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}$$

- **time: $\Theta(2^n)$ - exponential**
ORDERS OF GROWTH

Fibonacci

Let \( t(n) \) be the number of steps that we need to take to solve the case for size \( n \).

Then:

\[
t(n) = t_{n-1} + t_{n-2} = 2 \quad t_{n-2} = 4 \quad t_{n-4} = 8 \quad t_{n-6} = 2^{n/2}
\]

- \textbf{time: } \( \Theta(2^n) \) - exponential
- \textbf{space: } \( \Theta(n) \) - linear
ANOTHER WAY

Using Different Processes for the Same Goal

Let’s compute $a^b$

by only using multiplication and addition.
ANOTHER WAY
Using Different Processes for the Same Goal

Remember our stages:

1. Wishful thinking
2. Decomposition
3. The smallest sized sub-problem
ANOTHER WAY
Using Different Processes for the Same Goal

1. Wishful thinking
   Assume we have the procedure `my-expt`
   but only solves smaller versions of the same problem.
ANOTHER WAY
Using Different Processes for the Same Goal

1. Wishful thinking
   Assume we have the procedure `my-expt` but only solves smaller versions of the same problem.

2. Decompose problem into solving smaller version and using result:
ANOTHER WAY
Using Different Processes for the Same Goal

1. Wishful thinking
   Assume we have the procedure `my-expt` but only solves smaller versions of the same problem.

2. Decompose problem into
   solving smaller version and using result:

   \[ a^b = a \times a \times \cdots \times a = a \times a^{b-1} \]
Let's write the procedure.

```
(define my-expt
```

$ a^b = a \cdot a \cdot \ldots \cdot a = a \cdot a^{b-1} $
Another Way

Using Different Processes for the Same Goal

Let’s write the procedure.

```
(define my-expt
  (lambda (a b)
    \[ a^b = a \times a \times \cdots \times a = a \times a^{b-1} \]
```
ANOTHER WAY

Using Different Processes for the Same Goal

Let’s write the procedure.

```
(define my-expt
  (lambda (a b)
    (* a (my-expt a (- b 1)))))
```
$a^b = a \times a \times \cdots \times a = a \times a^{b-1}$

**ANOTHER WAY**

Using Different Processes for the Same Goal

What is wrong?

```scheme
(define my-expt
  (lambda (a b)
    (* a (my-expt a (- b 1))))))
```
ANOTHER WAY
Using Different Processes for the Same Goal

3. Identify the smallest size problem:

\[ a^b = a \cdot a \cdot \cdots \cdot a = a \cdot a^{b-1} \]
3. **Identify** the smallest size problem:

\[ a^0 = 1 \]
ANOTHER WAY
Using Different Processes for the Same Goal

3. **Identify** the smallest size problem:

\[
a^0 = 1 \quad \text{(define my-expt)}
\]
\[
\quad \text{(lambda (a b))}
\]
Another Way
Using Different Processes for the Same Goal

3. Identify the smallest size problem:

\[ a^b = a \times a \times \cdots \times a = a \times a^{b-1} \]

\[ a^0 = 1 \]

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

Using Different Processes for the Same Goal

• Orders of growth?
  
• time: 
  
• space:
  
(define my-expt
  (lambda (a b)
    (if (= b 0)
      1
      (* a (my-expt a (- b 1))))))
ANOTHER WAY
Using Different Processes for the Same Goal

• Orders of growth?
  
  • \textbf{time}: \Theta(n) - \text{linear}
  
  • \textbf{space}: \Theta(n) - \text{linear}

\begin{verbatim}
(define my-expt
 (lambda (a b)
   (if (= b 0)
     1
     (* a (my-expt a (- b 1))))))
\end{verbatim}
Are there other ways to decompose this problem?
Are there other ways to decompose this problem?

Use the idea of state variables and table evolution.
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

• In the table:
  • One column for each piece of information used.
  • One row for each step.
\[ a^b = a \times a \times \cdots \times a \]

**ITERATIVE ALGORITHM**

*Using Different Processes for the Same Goal*

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
$a^b = a \cdot a \cdot \cdots \cdot a$

**ITERATIVE ALGORITHM**

*Using Different Processes for the Same Goal*

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$</td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b - 1</td>
<td>a</td>
</tr>
</tbody>
</table>

Table showing iterative algorithm for processes.
## Iterative Algorithm

**Using Different Processes for the Same Goal**

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b − 1</td>
<td></td>
</tr>
<tr>
<td>$a^2$</td>
<td>b − 2</td>
<td>a</td>
</tr>
</tbody>
</table>
# Iterative Algorithm

Using Different Processes for the Same Goal

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>$b - 1$</td>
<td></td>
</tr>
<tr>
<td>$a^2$</td>
<td>$b - 2$</td>
<td></td>
</tr>
<tr>
<td>$a^3$</td>
<td>$b - 3$</td>
<td></td>
</tr>
<tr>
<td>$a^4$</td>
<td>$b - 4$</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^0$</td>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>$1$</td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>$b-1$</td>
<td></td>
</tr>
<tr>
<td>$a^2$</td>
<td>$b-2$</td>
<td></td>
</tr>
<tr>
<td>$a^3$</td>
<td>$b-3$</td>
<td></td>
</tr>
<tr>
<td>$a^4$</td>
<td>$b-4$</td>
<td></td>
</tr>
</tbody>
</table>

handles $a^0$ cleanly
**ITERATIVE ALGORITHM**

*Using Different Processes for the Same Goal*

- handles $a^0$ cleanly
- product $\times a$

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>$b-1$</td>
<td></td>
</tr>
<tr>
<td>$a^2$</td>
<td>$b-2$</td>
<td></td>
</tr>
<tr>
<td>$a^3$</td>
<td>$b-3$</td>
<td></td>
</tr>
<tr>
<td>$a^4$</td>
<td>$b-4$</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b – 1</td>
<td></td>
</tr>
<tr>
<td>$a^2$</td>
<td>b – 2</td>
<td></td>
</tr>
<tr>
<td>$a^3$</td>
<td>b – 3</td>
<td></td>
</tr>
<tr>
<td>$a^4$</td>
<td>b – 4</td>
<td></td>
</tr>
</tbody>
</table>

handles $a^0$ cleanly

product $\times a$

counter – 1
ITERATIVE ALGORITHM

Using Different Processes for the Same Goal

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$a$</td>
<td>$b - 1$</td>
<td>a</td>
</tr>
<tr>
<td>$a^2$</td>
<td>$b - 2$</td>
<td>a</td>
</tr>
<tr>
<td>$a^3$</td>
<td>$b - 3$</td>
<td>a</td>
</tr>
<tr>
<td>$a^4$</td>
<td>$b - 4$</td>
<td>a</td>
</tr>
</tbody>
</table>

handles $a^0$ cleanly

product $\times a$

answer

counter $- 1$
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

(define iexp (lambda (a b) (iexp-helper 1 b a)))
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

(define iexp (lambda (a b) (iexp-helper 1 b a)))

(define iexp-helper
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

\[
\text{(define iexp (lambda (a b) (iexp-helper 1 b a)))}
\]

\[
\text{(define iexp-helper}
\quad \text{(lambda (product count a))}
\]
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

(let iexp
  (lambda (a b) (iexp-helper 1 b a)))

(let iexp-helper
  (lambda (product count a)
    (if (= count 0)
      product
      (iexp-helper (* product a)
                    (- count 1)
                    a))))
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

(define iexp (lambda (a b) (iexp-helper 1 b a)))

(define iexp-helper
  (lambda (product count a)
    (if (= count 0)
      product
      (iexp-helper (* product a)
                    (- count 1)
                    a))))

• Orders of growth
  • time:
  • space:
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

\[
\text{(define iexp (lambda (a b) (iexp-helper 1 b a)))}
\]

\[
\text{(define iexp-helper}
\quad \text{(lambda (product count a))}
\quad \text{(if (= count 0)}
\quad \quad \text{product}
\quad \quad \text{(iexp-helper (* product a)}
\quad \quad \quad \text{(if (= count 1)}
\quad \quad \quad \quad \text{a)))})
\]

- Orders of growth
  - time: \(\Theta(n)\) - linear
  - space:
ITERATIVE ALGORITHM
Using Different Processes for the Same Goal

\[
\text{(define iexp (lambda (a b) (iexp-helper 1 b a)))}
\]

\[
\text{(define iexp-helper}
\text{  (lambda (product count a)}
\text{    (if (= count 0)
\text{      product
\text{      (iexp-helper (* product a)
\text{        (- count 1)
\text{        a))))))}
\]

- Orders of growth
  - \text{time: } \Theta(n) - \text{ linear}
  - \text{space: } \Theta(1) - \text{ constant}
Let’s compute $a^b$

by only using multiplication and addition.
Let’s compute $a^b$

by only using multiplication and addition.

if $b$ is even $a^b = (a^2)^{b/2}$
ANOTHER WAY
Using Different Processes for the Same Goal

Let’s compute $a^b$ by only using multiplication and addition.

if $b$ is even $a^b = (a^2)^{b/2}$

if $b$ is odd $a^b = a * a^{(b-1)}$
ANOTHER WAY
Using Different Processes for the Same Goal

Let’s compute $a^b$

by only using multiplication and addition.

if \( b \) is even

\[
a^b = (a^2)^{(b/2)}
\]

if \( b \) is odd

\[
a^b = a \times a^{(b-1)}
\]

We reduce the problem in half in one step!
ANOTHER WAY

Using Different Processes for the Same Goal

\[
\begin{align*}
\text{if } b \text{ is even } & \quad a^b = (a^2)^{b/2} \\
\text{if } b \text{ is odd } & \quad a^b = a \times a^{(b-1)}
\end{align*}
\]

\[
\text{(define fast-exp-1)}
\]
\[
\text{(lambda (a b)}
\]
\[
\text{(cond ((= b 1) a)}
\]
ANOTHER WAY

Using Different Processes for the Same Goal

if \(b\) is even \(a^b = (a^2)^{\frac{b}{2}}\)

if \(b\) is odd \(a^b = a \cdot a^{(b-1)}\)

(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)))))))
ANOTHER WAY
Orders of Growth

• If $n$ is even, then 1 step reduces to $n/2$ sized problem.
• If $n$ is odd, 2 steps reduces to $n/2^k$ sized problem.
• If $n$ is even, then 1 step reduces to $n/2$ sized problem.

• If $n$ is odd, 2 steps reduces to $n/2^k$ sized problem.

• We are done when the problem size is 1 which implies order of growth
ANOTHER WAY
Orders of Growth

• If $n$ is even, then 1 step reduces to $n/2$ sized problem.
• If $n$ is odd, 2 steps reduces to $n/2^k$ sized problem.
• We are done when the problem size is 1 which implies order of growth
  • **time:** \( \Theta(\log n) \) - logarithmic
  • **space:** \( \Theta(\log n) \) - logarithmic
SUMMARY
What have we learned?

- Why do these orders of growth matter?
- Main concern: general order of growth.
  - **Exponential** is very expensive as the problem size grows.
  - Clever thinking can turn an **inefficient** approach to a more **efficient** one.
- Actual performance vs. order of growth.