Foundations of Computer Science: Lecture 2

Recursion and Complexity

Recursion and Complexity
Recursion and Complexity
Recursion and Complexity
Recursion and Complexity
Recursion and Complexity
Recursion and Complexity
Recursion and Complexity
Recursion and Complexity

9th October 2023
Anil Madhavapeddy
The Practical Classes

https://www.cl.cam.ac.uk/teaching/2324/OCaml/

- Executed online in the hub.cl.cam.ac.uk server

- There are 5 ticks, each of which have a deadline for submission 10 days after they are issued (except last tick, which goes into Lent term).

  Tick 1: released 2023-10-06  due 2023-10-16
  Tick 2: released 2023-10-13  due 2023-10-23
  Tick 3: released 2023-10-20  due 2023-10-30
  Tick 4: released 2023-10-27  due 2023-11-06
  Tick 5: released 2023-11-03  due 2024-01-19
Expression Evaluation

\[ E_0 \rightarrow E_1 \rightarrow \ldots \rightarrow E_n \rightarrow v \]
Expression Evaluation

\[ E_0 \rightarrow E_1 \rightarrow \ldots \rightarrow E_n \rightarrow v \]

Focus on expressions; ignore side-effects for now.

This discipline of separating expression from effects is often known as functional programming

We will return to side effects later in the course to make useful programs!
Expression Evaluation

\[ E_0 \rightarrow E_1 \rightarrow \ldots \rightarrow E_n \rightarrow v \]

```ocaml
# let rec power x n =
  if n = 1 then x
  else if even n then
    power (x *. x) (n / 2)
  else
    x *. power (x *. x) (n / 2)
```
Expression Evaluation

\[ E_0 \rightarrow E_1 \rightarrow \ldots \rightarrow E_n \rightarrow v \]

```ocaml
# let rec power x n =
    if n = 1 then x
    else if even n then
        power (x *. x) (n / 2)
    else
        x *. power (x *. x) (n / 2)

power(2, 12) =>
power(4, 6) =>
power(16, 3) =>
16 \times power(256, 1) =>
16 \times 256 =>
4096
```
Summing first $n$ integers

```ocaml
# let rec nsum n =  
  if n = 0 then  
    0  
  else  
    n + nsum (n - 1)
```

$nsum\,3 \Rightarrow 3 + (nsum\,2)$

$\Rightarrow 3 + (2 + (nsum\,1))$

$\Rightarrow 3 + (2 + (1 + nsum\,0))$

$\Rightarrow 3 + (2 + (1 + 0))$
Summing first $n$ integers

# let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)

$nsum\ 3 \Rightarrow 3 + (nsum\ 2)$
$\Rightarrow 3 + (2 + (nsum\ 1))$  
$\Rightarrow 3 + (2 + (1 + nsum\ 0))$  
$\Rightarrow 3 + (2 + (1 + 0))$

Nothing can progress until the final expression is calculated!
Summing first $n$ integers

```plaintext
# let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```

Intermediate results are stored in the program stack which is usually of limited size.

$nsum\ 3 \Rightarrow 3 + (nsum\ 2)$
$\Rightarrow 3 + (2 + (nsum\ 1))$
$\Rightarrow 3 + (2 + (1 + nsum\ 0))$
$\Rightarrow 3 + (2 + (1 + 0))$

Nothing can progress until the final expression is calculated!
Summing first $n$ integers

# let rec nsum n =
if n = 0 then
  0
else
  n + nsum (n - 1)

Intermediate results are stored in the program stack which is usually of limited size.

$nsum\ 3 \Rightarrow 3 + (nsum\ 2)$
  $\Rightarrow 3 + (2 + (nsum\ 1))$
  $\Rightarrow 3 + (2 + (1 + nsum\ 0))$
  $\Rightarrow 3 + (2 + (1 + 0))$

Nothing can progress until the final expression is calculated!

Two types of storage:
heap is a global area where the memory storing values bound to names are tracked
stack is a list where function call arguments are pushed and return values popped.
Iteratively summing

```ocaml
# let rec summing n total =
  if n = 0 then
    total
  else
    summing (n - 1) (n + total)
```

```ocaml
# let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```
Iteratively summing

# let rec summing n total =
let rec nsum n =
  if n = 0 then
    total
  else
    summing (n - 1) (n + total)

  n + nsum (n - 1)

summing 3 0 ⇒ summing 2 3
⇒ summing 1 5
⇒ summing 0 6
⇒ 6

nsum 3 ⇒ 3 + (nsum 2)
⇒ 3 + (2 + (nsum 1))
⇒ 3 + (2 + (1 + nsum 0))
⇒ 3 + (2 + (1 + 0))
Iteratively summing

```ocaml
# let rec summing n total =
  if n = 0 then
    total
  else
    summing (n - 1) (n + total)

summing 3 0 ⇒ summing 2 3
⇒ summing 1 5
⇒ summing 0 6
⇒ 6
```

Extra argument `total` acts as the *accumulator* to keep track explicitly instead of using the stack.

Algorithms like this are known as *iterative* or *tail recursive*. 
Recursion vs iteration

• Why two terms *iterative* and *tail recursive*?
  • “Iterative” normally refers to a loop: e.g. coded using `while`.
  • “Tail-recursion” involves the recursive function call being the last thing that expression does.

• Tail-recursion is efficient only if the compiler detects it.
  • Mainly it saves space, though iterative code can run faster.

• Do not make programs iterative unless you determine the gain is significant.
How can we analyse our programs for efficiency?
Silly summing first $n$ integers

```ocaml
# let rec sillySum n =    if n = 0 then    0    else    n + (sillySum (n-1) + sillySum (n-1)) / 2
```

*Recursively* calls itself twice for every invocation
Silly summing first $n$ integers

```
# let rec sillySum n =
  if n = 0 then
    0
  else
    n + (sillySum (n-1) + sillySum (n-1)) / 2
```

Recursively calls itself twice for every invocation

Should **assign** the result to a local variable to prevent evaluating it twice

```
# let x = 2.0 in
let y = Float.pow x 20.0 in
y *. (x /. y)
```
Asymptotic complexity refers to how program costs grow with increasing inputs.

Usually space or time, with the latter usually being larger than the former.

Question: if we double our processing power, how much does our computation capability increase?
## Time Complexity

<table>
<thead>
<tr>
<th>complexity</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
<td>$\times 60$</td>
</tr>
<tr>
<td>$n \lg n$</td>
<td>140</td>
<td>4,893</td>
<td>200,000</td>
<td>$\times 41$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>31</td>
<td>244</td>
<td>1,897</td>
<td>$\times 8$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10</td>
<td>39</td>
<td>153</td>
<td>$\times 4$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>9</td>
<td>15</td>
<td>21</td>
<td>$+6$</td>
</tr>
</tbody>
</table>

Complexity = milliseconds of runtime given an input of size $n$
Comparing Algorithms with $O(n)$

Formally, define $f(n) = O(g(n))$ provided that $f(n) \leq c \cdot g(n)$
Comparing Algorithms with $O(n)$

Formally, define $f(n) = O(g(n))$ provided that $f(n) \leq c \cdot g(n)$

Intuitively, consider the most significant term and ignore constant or smaller factors

E.g. simplify $3n^2 + 34n + 433 \rightarrow n^2$
Facts about O notation

\[ O(2g(n)) \text{ is the same as } O(g(n)) \]

\[ O(\log_{10} n) \text{ is the same as } O(\ln n) \]

\[ O(n^2 + 50n + 36) \text{ is the same as } O(n^2) \]

\[ O(n^2) \text{ is contained in } O(n^3) \]

\[ O(2^n) \text{ is contained in } O(3^n) \]

\[ O(\log n) \text{ is contained in } O(\sqrt{n}) \]
Common complexity classes

\[ O(1) \quad \text{constant} \]
\[ O(\log n) \quad \text{logarithmic} \]
\[ O(n) \quad \text{linear} \]
\[ O(n \log n) \quad \text{quasi-linear} \]
\[ O(n^2) \quad \text{quadratic} \]
\[ O(n^3) \quad \text{cubic} \]
\[ O(a^n) \quad \text{exponential (for fixed } a) \]
Sample costs in O-notation

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>npower, nsum</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>summing</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(n(n + 1)/2)</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>power</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>sillySum</td>
<td>(O(2^n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>
Simple recurrence relations

\( T(n) \): a cost we want to bound using \( O \) notation

Typical base case: \( T(1) = 1 \)

Some recurrences:

\[
\begin{align*}
T(n + 1) &= T(n) + 1 & \quad O(n) \\
T(n + 1) &= T(n) + n & \quad O(n^2) \\
T(n) &= T(n/2) + 1 & \quad O(\log n) \\
T(n) &= 2T(n/2) + n & \quad O(n \log n)
\end{align*}
\]
Mapping this to OCaml

```ocaml
# let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```

Given \((n+1)\), does a constant amount of work

Then calls itself with \(n\)
Mapping this to OCaml

```ocaml
# let rec nsum n =
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```

Given (n+1), does a constant amount of work

Then calls itself with n

Therefore, recurrence relations are:

\[ T(0) = 1 \]
\[ T(n + 1) = T(n) + 1 \]
Mapping this to OCaml

```ocaml
# let rec nsum n =  
  if n = 0 then
    0
  else
    n + nsum (n - 1)
```

Given \((n+1)\), does a constant amount of work

Then calls itself with \(n\)

Therefore, recurrence relations are:

\[
T(0) = 1 \\
T(n + 1) = T(n) + 1
\]

\(O(n)\)
Mapping this to OCaml

```ocaml
# let rec nsumsum n = 
  if n = 0 then 
    0 
  else 
    nsum n + nsumsum (n - 1)
```

Calls itself recursively once

Calls nsum which takes $O(n)$
Mapping this to OCaml

```ocaml
# let rec nsumsum n =
  if n = 0 then
    0
  else
    nsum n + nsumsum (n - 1)
```

Calls itself recursively once

Calls nsum which takes $O(n)$

Therefore, recurrence relations are:

$$T(0) = 1$$

$$T(n + 1) = T(n) + n$$
Mapping this to OCaml

# let rec nsumsum n =
    if n = 0 then
        0
    else
        nsum n + nsumsum (n - 1)

Calls itself recursively once

Calls nsum which takes O(n)

Therefore, recurrence relations are:

\[
T(0) = 1
\]

\[
T(n + 1) = T(n) + n
\]

\[O(n^2)\]
Mapping this to OCaml

```ocaml
# let rec power x n =
  if n = 1 then x
  else if even n then
    power (x *. x) (n / 2)
  else
    x *. power (x *. x) (n / 2)
```

- Calls itself recursively once
- Always divides iteration count by 2
Mapping this to OCaml

```ocaml
# let rec power x n =  
  if n = 1 then x  
  else if even n then  
    power (x *. x) (n / 2)  
  else  
    x *. power (x *. x) (n / 2)
```

Calls itself recursively once

Always divides iteration count by 2

Therefore, recurrence relations are:

\[
T(0) = 1 \\
T(n) = T(n/2) + 1
\]
Mapping this to OCaml

```ocaml
# let rec power x n =  
  if n = 1 then x  
  else if even n then  
    power (x *. x) (n / 2)  
  else  
    x *. power (x *. x) (n / 2)
```

Calls itself recursively once

Always divides iteration count by 2

Therefore, recurrence relations are:

\[
T(0) = 1 \\
T(n) = T(n/2) + 1 \\
O(\log n)
\]