COM[illegible]0

Foundations of Computer Science

Lecture 11: Algorithmic Analysis

Outline

Motivation

Standard Approach

Examples

Simplifying with Worst-case and Big-O

Recursive Examples

Outline

Motivation

Standard Approach

Examples

Simplifying with Worst-case and Big-O

Recursive Examples

Algorithmic analysis: motivation

Want to compare alg particularly ones that can solve *arbitrarily large* insta

We would like to be able to talk about the resources (running time, memory, energy consumption) required by a program/algorithm as a function $f(n)$ of some parameter n (e.g. the size) of its input.

Example

How long does a given sorting algorithm take to run on a list of n elements?

Issues

Problems

- The exact resou[illegible]red for an algorithm are difficult to pin down. Heav[illegible]ent on:
    - Environment the program is run in (hardware, software, choice of language, external factors, etc)
    - Choice of inputs used
- Cost functions can be complex, e.g.

$$2n\log(n) + (n-100)\log(n)^2 + \frac{1}{2^n}\log(\log(n))$$

Need to identify the "important" aspects of the function.

Order of growth

Example

Consider two time-co[illegible]ns:

- $f_1(n) = \frac{1}{10}n^2$ mi[illegible] and
- $f_2(n) = 10n \log n$ milliseconds

Input size	$f_1(n)$	$f_2(n)$
100	0.01s	2s
1000	1s	30s
10000	1m40s	6m40s
100000	2h47m	1h23m
1000000	11d14h	16h40h
10000000	3y3m	8d2h

Outline

Algorithmic analysis

Asymptotic analysis is about how costs **scale** as the input increases.

Standard (default) a[illegible]

- Consider **asymptotic growth** of cost functions
- Consider **worst-case** (highest cost) inputs
- Consider **running time** cost: number of **elementary operations**

NB

Other common analyses include:

- *Average-case analysis*
- *Space (memory) cost*

Elementary operations

Informally: A single computational "step"; something that takes a constant number of computation cycles.

Examples:

- Arithmetic oper[illegible]
- Comparison of two values
- Assignment of a value to a variable
- Accessing an element of an array
- Calling a function
- Returning a value
- Printing a single character

NB

Count operations up to a constant factor, $O(1)$, rather than an exact number.

Outline

Motivation

Standard Approach

Examples

Simplifying with Worst-case and Big-O

Recursive Examples

Examples

Example

Squaring a number (First version):


```
square(n) :
    return n * n        O(1)
```


Running time: $O(1)$

Running time vs Execution time

Previous example shows one difference between running time and execution time.

In general, running time *approximates* execution time:

- Simplifying assumptions about elementary operations
- Hidden constants in big-O
- Big-O only looks at limiting performance as n gets large.

Examples

- Implementations of square(n) will take longer as n gets bigger
- A program that "solves chess" will run in $O(1)$ time.

Examples

Example

Squaring a number (…rsion):

$\texttt{square}(n)$:
  $r := 0$ — $O(1)$
  for $i = 1$ to n : — $O(1)$ — n times — $O(n)$
    $r := r + n$ — $O(1)$
  return r — $O(1)$

Running time: $O(1) + O(n) + O(1) = O(n)$

Examples

Example

Cubing a number (us[illegible]d squaring program):

cube(n) :			
$r := 0$			$O(1)$
for $i = 1$ to n :	$O(1)$	n times	$O(n^2)$
$r := r + \texttt{square}(n)$	$O(1) + O(n)$		
return r			$O(1)$

Running time: $O(1) + O(n^2) + O(1) = O(n^2)$

Outline

- Motivation
- Standard Approach
- Examples
- Simplifying with Worst case and Big-O
- Recursive Examples

Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

Example

Sum of squares (Using second squaring program):


```
sumOfSquares(n) :
  r := 0                                   O(1)
  for i = 1 to n :           O(1)  |
     r := r + square(i)      O(n)  | n times     O(n^2)
  return r                                 O(1)
```


Running time: $O(1) + O(n^2) + O(1) = O(n^2)$

Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

Example

Finding an element (x) in an array (L) of length n:


```
find(x, L) :
    for i = 0 to n − 1 :          O(1)  |
        if L[i] == x :            O(1)  |  O(n) times     O(n)
            return i              O(1)  |
    return −1                                             O(1)
```


Running time: $O(n) + O(1) = O(n)$

Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

NB

Simplifications might lead to sub-optimal bounds, may have to do a better analysis to get best bounds:

- *Finer-grained upper bound analysis*
- *Analyse specific cases to find a matching lower bound (big-Ω)*

NB

Big-Ω is a **lower bound** *analysis of the worst-case; NOT a "best-case" analysis.*

Worst-case and big-O

Analyse specific cases to find a matching lower bound (big-Ω)

Example

Let L_n be an n-element array of 0's.
Finding an element (x) in an array (L) of length n:

find(x, L) :			
for $i = 0$ to $n-1$:	$\Omega(1)$		
if $L[i] == x$:	$\Omega(1)$	$\Omega(n)$ times	$\Omega(n)$
return i	$\Omega(1)$		
return -1			$\Omega(1)$

Running time of find$(1, L_n)$: $\Omega(n)$

Therefore, running time of find(x, L): $\Theta(n)$

Outline

Recursive examples

Example

Factorial:

$$
\begin{array}{ll}
\texttt{fact}(n): & \\
\quad \text{if } n == 0: & O(1) \\
\quad\quad \text{return } 1 & O(1) \\
\quad \text{else}: & \\
\quad\quad \text{return } n * \texttt{fact}(n-1) & O(1) + T(n-1)
\end{array}
$$

Running time for $\texttt{fact}(n)$: $T(n)$, where:

$$
\begin{aligned}
T(0) &\in O(1) + O(1) = O(1) \\
T(n) &= T(n-1) + O(1) \\
&\in O(n)
\end{aligned}
$$

Running time: $T(n) \in O(n)$

Recursive examples

Example

Summing elements of a linked list (length n):

```
sum(L) :
  if  L.isEmpty() :                              O(1)
    return 0                                     O(1)
  else :
    return L.data + sum(L.next)                  O(1) + T(n − 1)
```


Running time for `sum(L)` : $T(n)$, where:

$$
\begin{aligned}
T(0) &\in O(1) + O(1) = O(1) \\
T(n) &= T(n-1) + O(1) \\
&\in O(n)
\end{aligned}
$$

Recursive examples

Example

Insertion sort (`L` has n elements):

`sort(L)`[illegible]	
if `L.i`[illegible] :	$O(1)$
return `L`	$O(1)$
else :	
`L2` := `sort(L.next)`	$T(n-1)$
insert `L.data` into `L2`	$O(n)$
return `L2`	$O(1)$

Running time for `sort(L)`: $T(n)$, where:

$$
\begin{aligned}
T(0) &\in O(1) + O(1) = O(1) \\
T(n) &= T(n-1) + O(n) + O(1) \\
&\in O(n^2)
\end{aligned}
$$

Recursive examples

Example

Euclidean algorithm for $\gcd(m, n)$ ($N = m + n$):

$$
\begin{array}{lll}
\gcd(m,n): & & \\
\quad \text{if } m > n: & & O(1) \\
\quad\quad \text{return } \gcd(m-n, n) & & \leq T(N-1) \\
\quad \text{else if } n > m: & O(1) & \\
\quad\quad \text{return } \gcd(m, n-m) & & \leq T(N-1) \\
\quad \text{else}: \quad \text{return } m & & O(1)
\end{array}
$$

Running time for $\gcd(m, n)$: $T(N)$, where:

$$
\begin{aligned}
T(1) &\in O(1) \\
T(N) &\leq T(N-1) + O(1) \\
&\in O(N)
\end{aligned}
$$

Recursive examples

Example

Euclidean algorithm for $\gcd(m, n)$ $(N = m + n)$:

Running time: $O(N)$

NB

N is not the input **size***. Input size is* $\log(m) + \log(n)$

Recursive examples

Example

Faster Euclidean algorithm for $\gcd(m, n)$ $(N = m + n)$:

$$
\begin{array}{lr}
\gcd(m, n) : & \\
\quad \text{if } m > n > 0 : & O(1) \\
\quad\quad \text{return } \gcd(m \,\%\, n, n) & \leq T(N/1.5) \\
\quad \text{else if } n > m > 0 : & O(1) \\
\quad\quad \text{return } \gcd(m, n \,\%\, m) & \leq T(N/1.5) \\
\quad \text{else} : \quad \text{return } \max(m, n) & O(1)
\end{array}
$$

Running time for $\gcd(m, n)$: $T(N)$, where:

$$
\begin{aligned}
T(1) &\in O(1) \\
T(N) &\leq T(N/1.5) + O(1) \\
&\in O(\log N)
\end{aligned}
$$

Recursive examples

Example

Faster Euclidean alg[illegible] $\gcd(m, n)$ ($N = m + n$):

What about lower bounds?

- Can show algorithm takes k steps to compute $\gcd(F_k, F_{k-1})$ where F_k is the k-th Fibonacci number
- Can show $1.5^k \leq F_k \leq 2^k$, so $k \in \Theta(\log F_k)$
- Therefore $\gcd(F_k, F_{k-1}) \in \Omega(\log(F_k + F_{k-1}))$

Exercise

Exercise

RW: 4.3.22 The following algorithm raises a number a to a power n.

```
exp(a, n) :
    p = 1
    i = n
    while i > 0 :
        p = p * a
        i = i - 1
    return p
```


Determine the running time of this algorithm.

Exercise

Exercise

RW: 4.3.21 The following algorithm gives a fast method for raising a number a to a pow

```
fast-exp(a, n) :
    p = 1
    q = a
    i = n
    while i > 0 :
        if i is odd :
            p = p * q
        q = q * q
        i = ⌊i/2⌋
    return p
```


Determine the running time of this algorithm.