Algorithms

Myrto Arapinis
School of Informatics
University of Edinburgh

October 15, 2014

Algorithms

Definition

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem

- **Input** - an algorithm has input values from a specified set
- **Output** - from the input values, the algorithm produces the output values from a specified set. The output values are the solution
- **Correctness** - an algorithm should produce the correct output values for each set of input values
- **Finiteness** - an algorithm should produce the output after a finite number of steps for any input
- **Effectiveness** - it must be possible to perform each step of the algorithm correctly and in a finite amount of time
- **Generality** - the algorithm should work for all problems of the desired form

Description of algorithms in pseudocode

- Intermediate step between English prose and formal coding in a programming language
- Focus on the fundamental operation of the program, instead of peculiarities of a given programming language
- Analyze the time required to solve a problem using an algorithm, independent of the actual programming language

Example - maximum

Describe an algorithm for finding the maximum value in a finite sequence of integers

Input: finite sequence of integers: $\bar{a} = a_1, \cdots a_n$
Output: integer a_i $(1 \leq i \leq n)$ *st* for all $j \in \{1, \cdots, n\}$, $a_j \leq a_i$

```
procedure max(a_1, ..., a_n)
max:= a_1
for i:=2 to n
    if max< a_i
    then max:=a_i
return max
```

Example - linear search

Describe an algorithm for locating an item in a sequence of integers

Input: integer: x, finite sequence of integers: $\bar{a} = a_1, \cdots a_n$
Output: integer i $(0 \leq i \leq n)$ *st* $a_i = x$ if $x \in \bar{a}$, $i = 0$ otherwise

```
procedure linear_search(x, a1, ..., an)
i:= 1
while i≤n and x≠ ai
    i:=i+1
if i≤n
then location:=i
else location:=0
return location
```

Example - binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input: integer: x, finite sequence of integers: $\bar{a} = a_1, \cdots a_n$
Output: integer i $(0 \leq i \leq n)$ $st\ a_i = x$ if $x \in \bar{a}$, $i = 0$ otherwise

- The algorithm begins by comparing the target with the middle element
  - if the middle element is strictly lower than the target, then the search proceeds with the upper half of the list
  - otherwise, the search proceeds with the lower half of the list (including the middle)
- Repeat this process until we have a list of size 1
  - if target is equal to the single element in the list, then the position is returned
  - otherwise, 0 is returned to indicate that the element was not found

Example - binary search

```
procedure binary_search(x, a_1, ..., a_n)
i:= 1
j:= m
while i<j
    m:=⌊(i + j)/2⌋
    if x> a_m
    then i:=m+1
    else j:=m
if x= a_i
then location:=i
else location:=0
return location
```

The growth of function

Given functions $f : \mathbb{N} \to \mathbb{R}$ or $f : \mathbb{R} \to \mathbb{R}$. Analyzing how fast a function grows

- Comparing two functions
- Comparing the efficiently of different algorithms that solve the same problem
- Applications in number theory (Chapter 4) and combinatorics (Chapters 6 and 8)

Big-O Notation

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. We say that f is $O(g)$ if there is a constant k and a positive constant C such that

$$\forall x > k.\ |f(x)| \leq C|g(x)|$$

- We say "f is big-O of g" or "g asymptotically dominates f"
- C and k are called witnesses to the relationship between f and g. Only one pair of witnesses is needed. (One pair implies many pairs: one can always make k or C larger)
- Common abuses of notation: "$f(x)$ is big-O of $g(x)$" or "$f(x) = O(g(x))$". This is not strictly true, since big-O refers to functions and not their values, and the equality doesn't hold
- $O(g)$ is the class of all functions f that satisfy the condition above. So it would be formally correct to write $f \in O(g)$

Examples

- $f(x) = a_n x^n + a_{n1} x^{n1} + \; + a_1 x + a_0$ is $O(x^n)$
- $1 + 2 + \; + n$ is $O(n^2)$
- $\log(n)$ is $O(n)$
- $n! = 1 \times 2 \times \cdots \times n$ is $O(n^n)$
- $\log(n!)$ is $O(n \log(n))$

Useful big-O estimates

- if $d > c > 1$,then n^c is $O(n^d)$, but n^d is not $O(n^c)$
- if $b > 1$ and c and d are positive, then $(\log_b(n))^c$ is $O(n^d)$, but n^d is not $O((\log_b(n))^c)$
- if $b > 1$ and d is positive, then n^d is $O(b^n)$, but b^n is not $O(n^d)$
- if $c > b > 1$,then b^n is $O(c^n)$, but c^n is not $O(b^n)$
- if f_1 is $O(g_1)$ and f_2 is $O(g_2)$ then $(f_1 + f_2)$ is $O(\max(|g_1|, |g_2|))$
- if f_1 is $O(g_1)$ and f_2 is $O(g_2)$ then $(f_1 \times f_2)$ is $O(g_1 \times g_2)$

Big-Omega notation

Definition

Let $f, g : \mathbb{R} \to \mathbb{R}$. We say that f is $\Omega(g)$ if there if there is a constant k and a positive constant C such that

$$\forall x > k.\ |f(x)| \geq C|g(x)|$$

- We say "f is big-Omega of g". The constants "C" and "k" are called witnesses to the relationship between f and g
- Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound
- Big-Omega tells us that a function grows at least as fast as another
- Similar abuse of notation as for big-O
- f is $\Omega(g)$ if and only if g is $O(f)$ (Prove this by using the definitions of O and Ω)

Big-Theta notation

Definition

Let $f, g : \mathbb{R} \to \mathbb{R}$. We say that f is $\Theta(g)$ iff f is $O(g)$ and f is $\Omega(g)$

- We say "f is big-Theta of g" and also "f is of order g" and also "f and g are of the same order"
- f is $\Theta(g)$ iff there exists constants C_1, $C_2$and k such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ if $x > k$. This follows from the definitions of big-O and big-Omega

Example

Show that the sum $1 + 2 + \cdots + n$ of the first n positive integers is $\Theta(n^2)$ Solution: Let $f(n) = 1 + 2 + \cdots + n$. We have previously

shown that $f(n)$ is $O(n^2)$
To show that $f(n)$ is $\Omega(n^2)$, we need a positive constant C such that $f(n) > Cn^2$ for sufficiently large n
Summing only the terms greater than $n/2$ we obtain the inequality:

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\ &= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (n/2)(n/2) \\ &= n^2/4 \end{aligned}$$

Taking $C = 1/4$, $f(n) > Cn^2$ for all positive integers n. Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$

Complexity of algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size?
  How much time does this algorithm use to solve a problem?
  How much computer memory does this algorithm use to solve a problem?
- We measure time complexity in terms of the number of operations an algorithm uses and use big-O and big-Theta notation to estimate the time complexity
- Compare the efficiency of different algorithms for the same problem
- We focus on the worst-case time complexity of an algorithm. Derive an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size. (As opposed to the average-case complexity)
- Here: Ignore implementation details and hardware properties
  $\longrightarrow$ See courses on algorithms and complexity.

Worst-Case complexity of linear search

```
procedure linear_search(x, a1, ..., an)
i:= 1
while i≤n and x≠ ai
    i:=i+1
if i≤n
then location:=i
else location:=0
return location
```

Count the number of comparisons':

- at each step two comparisons are made; $i \leq n$ and $x \neq ai$
- to end the loop, one comparison $i \leq n$ is made
- after the loop, one more $i \leq n$ comparison is made

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$

Average-Case complexity of linear search

For many problems, determining the average-case complexity is very difficult. (And often not very useful, since the real distribution of input cases does not match the assumptions.) However, for linear search the average-case is easy.
Assume the element is in the list and that the possible positions are equally likely. By the argument on the previous slide, if $x = a_i$, the number of comparisons is $2i + 1$. Hence, the average-case complexity of linear search is

$$\frac{1}{n}\sum_{i=1}^{n} 2i \ + \ 1 = n + 2$$

Which is $\Theta(n)$

Worst-Case complexity of binary search

```
procedure binary_search(x, a_1, ..., a_n)
i:= 1
j:= m
while i<j
    m:=⌊(i + j)/2⌋
    if x> a_m then i:=m+1 else j:=m
if x= a_i then location:=i else location:=0
return location
```

Assume (for simplicity) $n = 2^k$ elements. Note that $k = log_2 n$. Two comparisons are made at each stage; $i < j$, and $x > a_m$. At the first iteration the size of the list is 2^k and after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$. At the last step, a comparison tells us that the size of the list is the size is $2^0 = 1$ and the element is compared with the single remaining element. Hence, at most $2k + 2 = 2log_2 n + 2$ comparisons are made. $\Theta(logn)$