Discrete Mathematics, Chapter 3: Algorithms

Richard Mayr

University of Edinburgh, UK

Outline

Algorithms  (Abu Ja ’far Mohammed Ibin Musa Al-Khowarizmi, 780-850)

Definition

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

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

Properties of Algorithms

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.

Example: Linear Search

Prose: Locate an item in a list by examining the sequence of list elements one at a time, starting at the beginning.

More formal prose: Find item x in the list $[a_1, a_2, \ldots, a_n]$.

- First compare x with a_1. If they are equal, return the position 1.
- If not, try a_2. If $x = a_2$, return the position 2.
- Keep going, and if no match is found when the entire list is scanned, return 0.

Pseudocode:

Algorithm 1: Linear Search

Input: x : *integer*, $[a_1, \ldots, a_n]$: list of distinct integers
Output: Index i s.t. $x = a_i$ or 0 if x is not in the list.

```
i := 1;
while i ≤ n and x ≠ a_i do
    i := i + 1;
if i ≤ n then result := i else result := 0;
return result;
```

Binary Search

Prose description:

- Assume the input is a list of items in increasing order, and the target element to be found.
- 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.

Binary Search

Pseudocode:

Algorithm 2: Binary Search

Input: x : *integer*, $[a_1, \ldots, a_n]$: strictly increasing list of integers
Output: Index i s.t. $x = a_i$ or 0 if x is not in the list.

$i := 1$; // i is the left endpoint of the interval
$j := n$; // j is the right endpoint of the interval
while $i < j$ **do**
  $m := \lfloor (i + j)/2 \rfloor$;
  if $x > a_m$ **then** $i := m + 1$ **else** $j := m$;
if $x = a_i$ **then** *result* $:= i$ **else** *result* $:= 0$;
return *result*;

Example: Binary Search

Find target **19** in the list: 1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. The list has 16 elements, so the midpoint is 8. The value in the 8th position is 10. As $19 > 10$, search is restricted to positions 9-16.
   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22
2. The midpoint of the list (positions 9 through 16) is now the 12th position with a value of 16. Since $19 > 16$, further search is restricted to the 13th position and above.
   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22
3. The midpoint of the current list is now the 14th position with a value of 19. Since $19 \not> 19$, further search is restricted to the portion from the 13th through the 14th positions.
   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22
4. The midpoint of the current list is now the 13th position with a value of 18. Since $19 > 18$, search is restricted to position 14.
   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22
5. Now the list has a single element and the loop ends.
   Since $19 = 19$, the location 14 is returned.

Greedy Algorithms

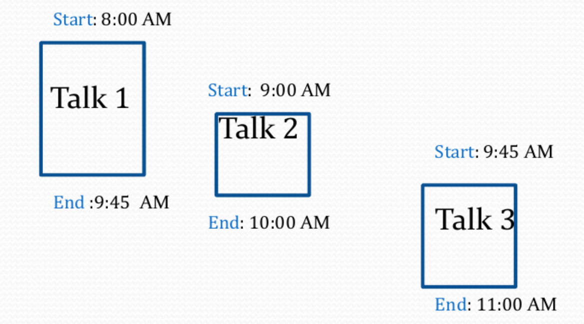
- Optimization problems minimize or maximize some parameter over all possible inputs.
- Examples of optimization problems:
  - Finding a route between two cities with the smallest total mileage.
  - Determining how to encode messages using the fewest possible bits.
  - Finding the fiber links between network nodes using the least amount of fiber.
- Optimization problems can often be solved using a greedy algorithm, which makes the “best” (by a local criterion) choice at each step. This does not necessarily produce an optimal solution to the overall problem, but in many instances, it does.
- After specifying what the “best choice” at each step is, we try to prove that this approach always produces an optimal solution, or find a counterexample to show that it does not.

Example: Greedy Scheduling

We have a group of proposed talks with start and end times. Construct a greedy algorithm to schedule as many as possible in a lecture hall, under the following assumptions:

- When a talk starts, it continues till the end. (Indivisible).
- No two talks can occur at the same time. (Mutually exclusive.)
- A talk can begin at the same time that another ends.
- Once we have selected some of the talks, we cannot add a talk which is incompatible with those already selected because it overlaps at least one of these previously selected talks.
- How should we make the "best choice" at each step of the algorithm? That is, which talk do we pick?
  - The talk that starts earliest among those compatible with already chosen talks?
  - The talk that is shortest among those already compatible?
  - The talk that ends earliest among those compatible with already chosen talks?

Greedy Scheduling



- Picking the shortest talk doesn't work.
- But picking the one that ends soonest does work. The algorithm is specified on the next page.

A Greedy Scheduling Algorithm

At each step, choose the talks with the earliest ending time among the talks compatible with those selected.

Algorithm 3: Greedy Scheduling by End Time

Input: $s_1, s_2, \ldots, s_n$ start times and $e_1, e_2, \ldots, e_n$ end times
Output: An optimal set $S \subseteq \{1, \ldots, n\}$ of talks to be scheduled.
Sort talks by end time and reorder so that $e_1 \leq e_2 \leq \cdots \leq e_n$
$S := \emptyset$;
for $j := 1$ ***to*** n **do**
  if *Talk j is compatible with S* **then**
    $S := S \cup \{j\}$
return S;

Note: Scheduling problems appear in many applications. Many of them (unlike this simple one) are NP-complete and do not allow efficient greedy algorithms.

The Growth of Functions

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

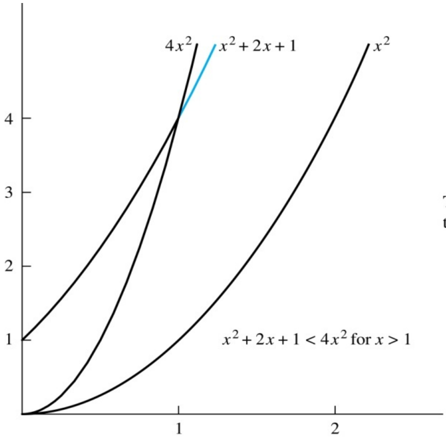
Let $f, g : \mathbb{R} \to \mathbb{R}$. We say that f is $O(g)$ if there are constants C and k such that

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

- This is read as "f is big-O of g" or "g asymptotically dominates f".
- The constants 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, since one can always make k or C larger.)
- Common abuses of notation: Often one finds this written as "$f(x)$ 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 does not hold.
- Strictly speaking $O(g)$ is the class of all functions f that satisfy the condition above. So it would be formally correct to write $f \in O(g)$.

Illustration of Big-O Notation

$f(x) = x^2 + 2x + 1$, $g(x) = x^2$.
f is $O(g)$ witness $k = 1$ and $C = 4$.
Abusing notation, this is often written as $f(x) = x^2 + 2x + 1$ is $O(x^2)$.



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

Properties of Big-O Notation

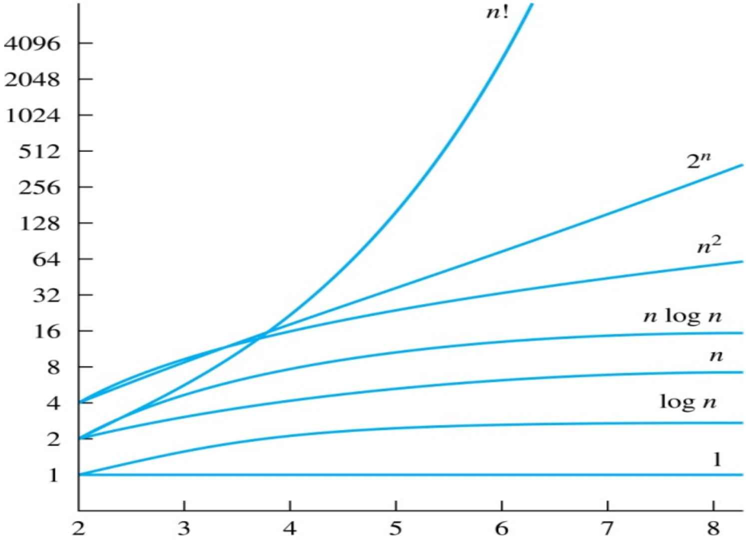
- If f is $O(g)$ **and** g is $O(f)$ then one says that f and g are of the same order.
- If f is $O(g)$ and $h(x) \geq g(x)$ for all positive real numbers x then f is $O(h)$.
- The O-notation describes upper bounds on how fast functions grow. E.g., $f(x) = x^2 + 3x$ is $O(x^2)$ but also $O(x^3)$, etc.
- Often one looks for a **simple** function g that is as small as possible such that still f is $O(g)$.
  (The word 'simple' is important, since trivially f is $O(f)$.)

Example

Bounds on functions. Prove that

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

Growth of Common Functions



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 $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.
- If f_1 is $O(g_1)$ and f_2 is $O(g_2)$ then $(f_1 \circ f_2)$ is $O(g_1 \circ g_2)$.

Note: These estimates are very important for analyzing algorithms. Suppose that $g(n) = 5n^2 + 7n - 3$ and f is a very complex function that you cannot determine exactly, but you know that f is $O(n^3)$.
Then you can still derive that $n \cdot f(n)$ is $O(n^4)$ and $g(f(n))$ is $O(n^6)$.

Big-Omega Notation

Definition

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

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

- This is read as "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 $\Theta(g)$ if f is $O(g)$ and f is $\Omega(g)$.

- We say that "f is big-Theta of g" and also that "f is of order g" and also that "f and g are of the same order".
- f is $\Theta(g)$ if and only if there exists constants C_1, C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ 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

Algorithm 4: Linear Search

Input: x : *integer*, $[a_1, \ldots, a_n]$: list of distinct integers
Output: Index i s.t. $x = a_i$ or 0 if x is not in the list.

$i := 1$;
while $i \leq n$ *and* $x \neq a_i$ **do**
    $i := i + 1$;
if $i \leq n$ **then** *result* $:= i$ **else** *result* $:= 0$;
return *result*;

Count the number of comparisons.

- At each step two comparisons are made; $i \leq n$ and $x \neq a_i$.
- 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

Algorithm 5: Binary Search

Input: x : *integer*, $[a_1, \ldots, a_n]$: strictly increasing list of integers
Output: Index i s.t. $x = a_i$ or 0 if x is not in the list.
$i := 1$; // i is the left endpoint of the interval
$j := n$; // j is the right endpoint of the interval
while $i < j$ **do**
  $m := \lfloor (i + j)/2 \rfloor$;
  if $x > a_m$ **then** $i := m + 1$ **else** $j := m$;
if $x = a_i$ **then** *result* $:= i$ **else** *result* $:= 0$;
return *result*;

Assume (for simplicity) $n = 2^k$ elements. Note that $k = \log n$.
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 $2k + 2 = 2 \log n + 2$ comparisons are made. $\Theta(\log n)$.

Terminology for the Complexity of Algorithms

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Further topics

See courses on algorithms and complexity for

- Space vs. time complexity
- Intractable problems
- Complexity classes: E.g., P, NP, PSPACE, EXPTIME, EXPSPACE, etc.
- Undecidable problems and the limits of algorithms.