Complexity and Sorting

Carlos Cotta

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga

http://www.lcc.uma.es/∼ccottap

Comput Eng, Softw Eng, Comput Sci & Math – 2023-2024

UNIVERSIDAD DE MÁLAGA

Complexity and Sorting

An Eternal Gentle Loop

Abu Ja'far Muḥammad ibn Mūsā al Ḵwārizimī

Algorithm

An algorithm is a systematic procedure, non-ambiguously described step by step and amenable for automated execution, that allows solving a computational problem in a finite amount of time.

Computational Problem

A computational problem is a class of tasks that can be solved by means of computers.

Problems, Instances and Solutions

Let P be a computational problem.

Associated to a problem P, we can define I_P, the set of instances of the problem (potential incarnations of the latter).

Let $x \in I_P$ be a certain instance of a problem P.

Associated to x, we have:

- $sol_P(x)$, the set of potential solutions for x.
- $\Psi_P(x, y) : I_P \times sol_P(x) \longrightarrow \mathbb{B}$, a feasibility function, i.e., $y \in sol_P(x)$ is a feasible solution for x if, and only if, $\Psi_P(x, y) = \mathrm{TRUE}$.
- $val_P(x) = \{y \in sol_P(x) \mid \Psi_P(x, y)\}$ is the set of feasible solutions for x.

Example

Example

Let $\mathcal{T} = \{t_1, \cdots, t_m\}$ and $\mathcal{A} = \{a_1, \cdots, a_n\}$ be a set of teachers and courses respectively. Let $C : \mathcal{T} \longrightarrow 2^{\mathcal{A}}$ be a function indicating the capabilities of each teacher (i.e., the courses (s)he can teach).

Find a feasible teaching assignment for all teachers in $\mathcal{T}$.

Each $x \in I_P$ captures specific values for $\mathcal{T}$, $\mathcal{A}$, and C.

$sol_P(x)$ is a set of functions $F : \mathcal{A} \longrightarrow \mathcal{T}$ (i.e., assignment of courses to teachers).

$\Psi_P(x, F)$ is $\mathtt{TRUE}$ (F is a feasible assignment) if, and only if, for all $a_i \in \mathcal{A}$, $a_i \in C(F(a_i))$.

Satisfaction Problems

In the previous example, the objective was finding a feasible assignment. A problem of this kind is termed satisfaction problem.

Satisfaction Problem

Given an instance $x \in I_P$ of problem P, find a solution $y \in val_P(x)$ (i.e., a feasible solution, that is, a solution for which $\Psi_P(x, y)$ is $\mathtt{TRUE}$).

There are other types of problems...

Decision Problem

Decision Problem

Given an instance $x \in I_P$ of problem P, determine whether $val_P(x) = \emptyset$ or not.

Following the previous example, the objective in this case would be determining whether there exists a feasible teaching assignment or not.

The answer to a decision problem is always $\mathtt{YES}$ or $\mathtt{NO}$.

This kind of problem plays an important role in the Theory of Computational Complexity.

Counting Problems

Counting Problem

Given an instance $x \in I_P$ of problem P, determine the cardinality of $val_P(x)$.

In the previous example, the objective under this formulation of the problem would be computing how many feasible assignments exist.

It is easy to see that a counting problem is always at least as hard to solve as its decision version.

Enumeration Problems

Enumeration Problem

Given an instance $x \in I_P$ of problem P, find every solution $y \in val_P(x)$.

In the context of the previous example, our goal would be to actually find all feasible assignments, rather than merely knowing mow many of them exist.

Again, it is easy to see that an enumeration problem generalizes a counting problem.

Optimization Problems

Optimization Problem

Let $x \in I_P$ be an instance of problem P, and let $\prec_P$ be a partial order relation such that given $y, y' \in val_P(x)$, if $y \prec_P y'$ then y is preferred to y'. Find an optimal solution, that is, a solution y^* for which no $y \in val_P(x)$ exists such that $y \prec_P y^*$.

The most typical situation is that in which a function $f : val_P(x) \longrightarrow \mathbb{R}$ is available, and the solution y^* maximizing or minimizing f is sought.

In that case, $y \prec_P y'$ if, and only if, $f(y) < f(y')$ (minimization assumed).

Note that a satisfaction problem can be formulated as an optimization problem.

Optimization Problem

In the previous example, each teacher could have expressed his/her preferences for each course. This information might be captured by a function

$$\pi : \mathcal{T} \times \mathcal{A} \longrightarrow \mathbb{R}^+$$

The higher the value of $\pi(t, a)$, the stronger the preference of teacher t for course a.

The objective might then be finding an assignment $F \in val_P(x)$ maximizing

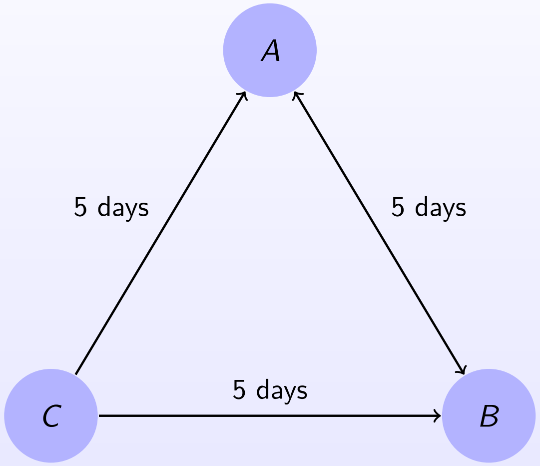
$$f = \sum_{a \in \mathcal{A}} \pi(F(a), a)$$

Classes of Algorithms

We will mainly focus on satisfaction and optimization problems. Algorithms dealing with this kind of problems can be classified according to different criteria:

- Reproducibility
- Completeness
- Complexity
- ...

A Fairy Tale



We know that there is a treasure comprising X gold coins hidden either in A or in B.

The time required to traverse each of the roads $\overline{AB}$, $\overline{AC}$, and $\overline{BC}$ is 5 days.

We also have a map showing the location of the treasure, but it is encrypted, and we will need 4 days to decrypt it.

A dragon takes Y coins from the treasure every day.

A Fairy Tale

First Solution

Decrypt the map. Our final gain is $X - (4+5)Y = X - 9Y$ coins.

Second Solution

Make a random decision. If we guess correctly the location of the treasure the gain is $X - 5Y$. Otherwise, it is $X - 10Y$. The mathematical expectation is $X - 7.5Y$.

The Bottom Line

Sometimes it is better to make a random decision rather than spending too many resources in determining the best course of action.

Classes of Algorithms

Reproducibility

Deterministic Algorithms

A deterministic algorithm always provides the same output for a given input.

Stochastic Algorithms

A stochastic algorithm can behave in a different way each time it is run on a certain input.

If multiple runs of a stochastic algorithm are performed, the right sequence of decisions can be eventually made.

Classes of Algorithms

Completeness

Complete Algorithms

A complete algorithm ensures the successful resolution of the problem. In the particular case of optimization problems, these can be in turn:

- Exact: provide an optimal solution.
- Approximate: provide a solution whose quality is within a known distance of that of the optimal solution.

Heuristic Algorithms

A heuristic algorithm can be often successful, but does not ensure anything about the solutions provided.

Classes of Algorithms

Completeness

Finding the optimal solution or a solution of bounded quality can be very costly in many problems.

A heuristic can provide a high-quality solution at a low computational cost (although we have no assurance this will be always the case).

Resources Consumed by an Algorithm

Generate List of Primes

func GeneratePrimeList ($\downarrow N$: $\mathbb{N}$): List$\langle\mathbb{N}\rangle$
variables
  i: $\mathbb{N}$
  L: List$\langle\mathbb{N}\rangle$
begin
  $L \leftarrow \langle\rangle$
  for $i \leftarrow 2$ **to** N **do**
    if $\neg$divisible(i,L) **then** L.add(i); **endif**
  endfor
  return L
end

Resources Consumed by an Algorithm

Generate List of Primes

func divisible ($\downarrow n$: $\mathbb{N}$, $\downarrow L$: List$\langle\mathbb{N}\rangle$):$\mathbb{B}$
// Traverse list L looking for a factor of n.
// Return TRUE if, and only if, it is found.
variables
  p: $\mathbb{N}$
begin
  for each $p \in$ L **do**
    if mod(n, p)$=0$ **then**
      return TRUE
    endif
  endfor
  return FALSE
end

Resources Consumed by an Algorithm

The previous algorithm consumes space to store the prime numbers it finds and time to determine whether each number is prime or not.

Can we bound this consumption?

- space: a node in the list for each prime number.
- time: $N-1$ primality tests, each of them implying a traversal of list L.

It is known (by the prime number theorem) that the density of prime numbers is $1/\ln n$, and hence space consumption is approx. $N/\ln N$ nodes.

If the list has $N/\ln N$ nodes at most, the total number of divisions performed is bounded from above by $N^2/\ln N$.

Resources Consumed by an Algorithm

What have we just done?

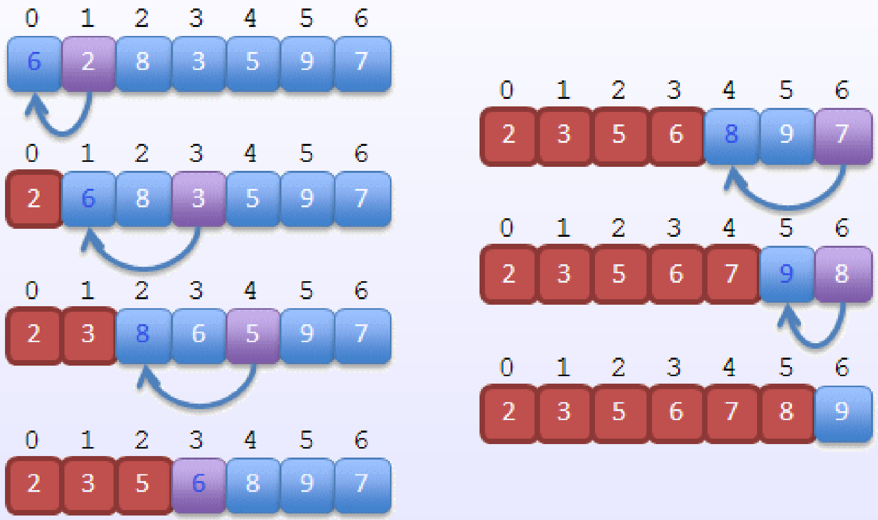
- We have estimated resource consumption in terms of the input parameters.
- We cannot know the exact amount of space used, but we know it is proportional to $N/\ln N$.
- We have assumed the division operation is the most time-consuming part of the algorithm. This is what we call the basic operation.
- We cannot know the exact CPU time the algorithm will take, but we know it will roughly proportional to $N^2/\ln N$ at most.

Selection Sort

Selection Sort

```
proc SelectionSort (↓↑A: ARRAY [1..N] OF ℕ)
variables i, j, min: ℕ
          temp: ℕ  // element type
begin
   for i ← 1 to N − 1 do
      min ← i
      for j ← i + 1 to N do
         if A[j]<A[min] then min ← j endif
      endfor
      temp ← A[i]
      A[i] ← A[min]
      A[min] ← temp
   endfor
end
```


Selection Sort at Work



Complexity of Selection Sort

All loops in the algorithm are non-conditional. Hence, the initial state of the array does not influence the computational cost (measured either the number of element comparisons or assignments). Only the array size matters.

The number of comparisons $T_c(n)$ and assignments $T_a(n)$ is respectively:

$$T_c(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

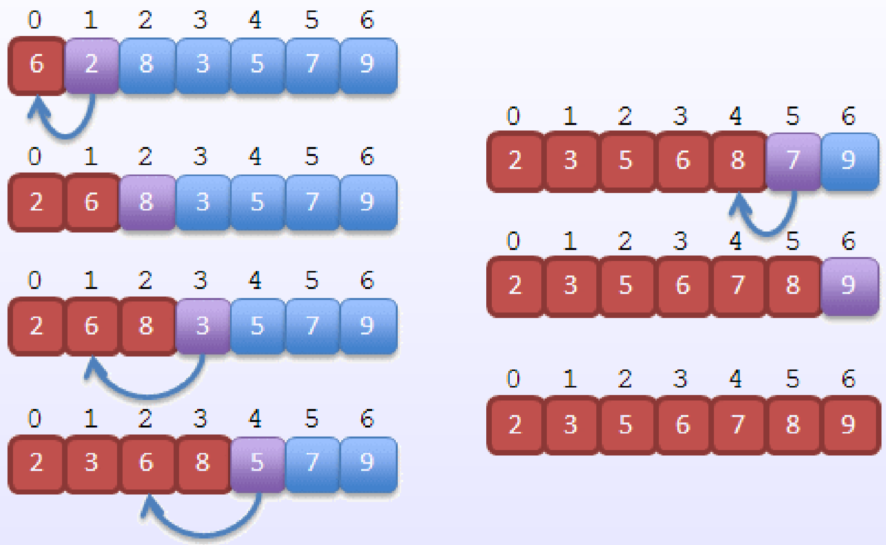
$$T_a(n) = \sum_{i=1}^{n-1} 3 = 3(n-1)$$

Size matters (but so do other things)

Insertion Sort

proc InsertionSort ($\downarrow\uparrow A$: ARRAY $[1..N]$ OF $\mathbb{N}$)
variables i, j, k, r: $\mathbb{N}$
begin
  for $i \leftarrow 2$ **to** N **do**
    $r \leftarrow A[i]$
    $j \leftarrow i{-}1$
    while $(j \geqslant 1) \wedge (A[j] > r)$ **do**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j{-}1$
    endwhile
    $A[j+1] \leftarrow r$
  endfor
end

Insertion Sort at Work



Best Case, Worst Case, Average Case

If the array we want to sort is already sorted, the algorithm will never enter in the inner loop: only $N-1$ comparisons and $2(N-1)$ element assignments are done.

If the array we want to sort is inversely sorted at the beginning, the inner loop will always iterate until reaching the first position of the array: $N(N-1)/2$ comparisons (and as many assignments) are done in that loop.

The first scenario depicts the best case, whereas the second one corresponds to the worst case.

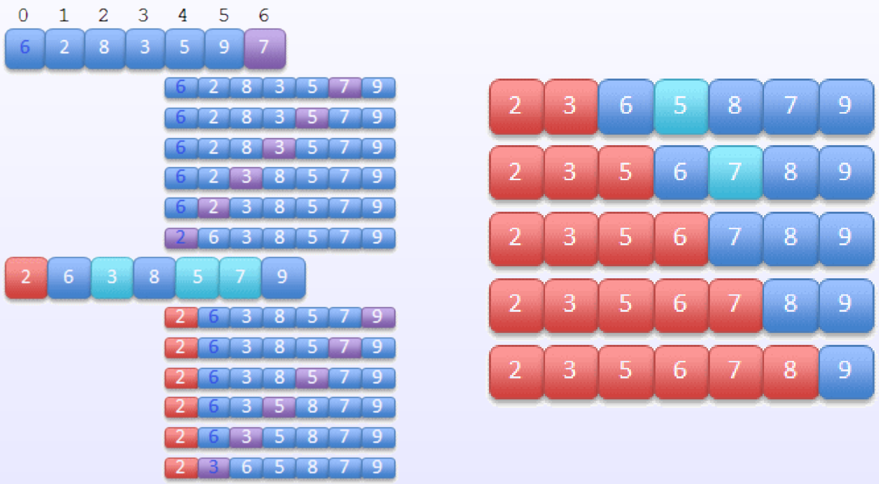
The complexity in the average case would be a weighted sum of the computational cost for each possible problem input (the weights would indicate the probability of each particular input).

BubbleSort

Bubble Sort

proc BubbleSort ($\downarrow\uparrow A$: ARRAY $[1..N]$ OF $\mathbb{N}$)
variables i, j, min: $\mathbb{N}$
      $temp$: $\mathbb{N}$ // element type
begin
  for $i \leftarrow 1$ **to** $N-1$ **do**
    for $j \leftarrow N$ **to** $i+1$ **step** -1 **do**
      if $A[j] < A[j-1]$ **then**
        $temp \leftarrow A[j]$
        $A[j] \leftarrow A[j-1]$
        $A[j-1] \leftarrow temp$
      endif
    endfor
  endfor
end

BubbleSort at Work



Complexity of BubbleSort

The number of comparisons does not depend on the initial state of the array:

$$T_c(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

However, the number of assignments does depend on the initial state. If the array is already ordered, no assignment is ever done. In the worst case (inverse ordering at the beginning), 3 assignments are done in each iteration of the inner loop:

$$T_a(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 3 = \frac{3n(n-1)}{2}$$

Objective

We are interested in estimating the computational cost $t(n)$ of an algorithm as a function of the input size.

More precisely, we are specifically interested in the order of growth of the computational cost when the input size grows arbitrarily large.

The growth order tell us the precise way in which the computational cost increases when the input size grows, e.g., logarithmically, linearly, quadratically, exponentially, etc.

Orders of Growth

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3.3	$3.3 \cdot 10^1$	10^2	10^3	$1.0 \cdot 10^3$	$3.6 \cdot 10^6$
10^2	6.6	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	$1.0 \cdot 10^4$	10^6	10^9	$1.1 \cdot 10^{301}$	$4.0 \cdot 10^{2567}$
10^4	13	$1.3 \cdot 10^5$	10^8	10^{12}	$2.0 \cdot 10^{3010}$	$2.8 \cdot 10^{35659}$
10^5	17	$1.7 \cdot 10^6$	10^{10}	10^{15}	$1.0 \cdot 10^{30103}$	$2.8 \cdot 10^{456573}$
10^6	20	$2.0 \cdot 10^7$	10^{12}	10^{18}	$9.9 \cdot 10^{301029}$	$8.3 \cdot 10^{5565708}$

Notation

The asymptotic notation allows comparing the orders of growth of different algorithms:

- $O(\cdot)$: Big Oh
- $\Omega(\cdot)$: Big Omega
- $\Theta(\cdot)$: Theta

Each of these notations allows bounding in a different way the order of growth of an algorithm.

Let $g, h, t : \mathbb{N} \longrightarrow \mathbb{R}^+$ be arbitrary non-negative functions of natural numbers henceforth.

Big Oh – Asymptotic Upper Bound

Definition

A function $t(n)$ is in $O(g(n))$, i.e., $t(n) \in O(g(n))$ if $g(n)$ bounds $t(n)$ from above for arbitrarily large n:

$$\exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : \forall n \geqslant n_0 : t(n) \leqslant cg(n)$$

In other words, $t(n)$ grows no faster than $g(n)$.

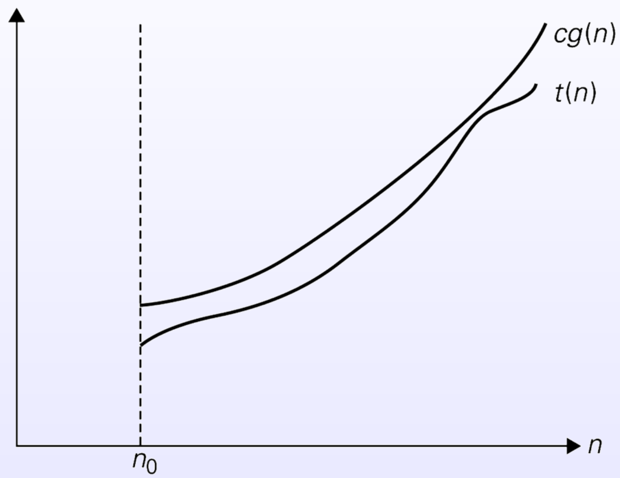
Example

Let $t(n) = 10n + 2$. We can see that $t(n) \in O(n)$:

$$10n + 2 \leqslant cn$$
$$(10 - c)n \leqslant -2$$

This inequality holds for, e.g., $c = 11$ and $n > n_0 = 2$ (and for infinitely many other values of c and n_0).

Big Oh – Asymptotic Upper Bound



A. Levitin, © 2007 Pearson Addison-Wesley

Big Omega – Asymptotic Lower Bound

Definition

A function $t(n)$ is in $\Omega(g(n))$, i.e., $t(n) \in \Omega(g(n))$ if $g(n)$ bounds $t(n)$ from below for arbitrarily large n:

$$\exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : \forall n \geqslant n_0 : t(n) \geqslant cg(n)$$

In other words, $t(n)$ grows at least as fast as $g(n)$.

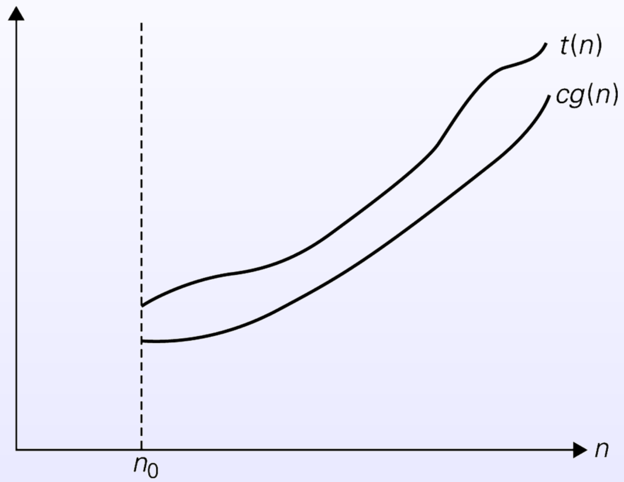
Example

Let $t(n) = 10n^3$. $t(n) \in \Omega(n^2)$ since:

$$10n^3 \geqslant cn^2$$
$$10n \geqslant c$$

For any $c > 0$ and $n > n_0 = c/10$, it holds that $10n^3 \geqslant cn^2$.

Big Omega – Asymptotic Lower Bound



A. Levitin, © 2007 Pearson Addison-Wesley

Theta – Asymptotic Tight Bound

Definition

A function $t(n)$ is in $\Theta(g(n))$, i.e., $t(n) \in \Theta(g(n))$ if $g(n)$ bounds $t(n)$ both from above and from below for arbitrarily large n:

$$\exists c_1, c_2 \in \mathbb{R}^+ : \ \exists n_0 \in \mathbb{N} : \ \forall n \geqslant n_0 : \ c_1 g(n) \leqslant t(n) \leqslant c_2 g(n)$$

In other words, $t(n)$ grows as fast as $g(n)$.

Example

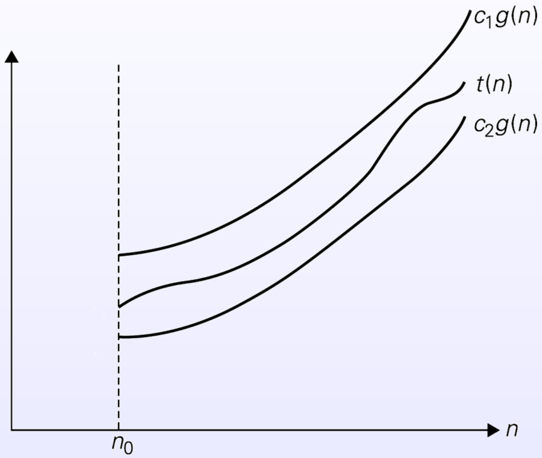
Let $t(n) = n(n-1)/2$. $t(n) \in \Theta(n^2)$ since:

$$n(n-1)/2 = \frac{n^2}{2} - \frac{n}{2} \leqslant \frac{n^2}{2} \text{ for all } n \geqslant 0$$

$$n(n-1)/2 = \frac{n^2}{2} - \frac{n}{2} \geqslant \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4} \text{ for all } n \geqslant 2$$

We can take $c_1 = 1/4$, $c_2 = 1/2$, $n_0 = 2$.

Theta – Asymptotic Tight Bound



A. Levitin, © 2007 Pearson Addison-Wesley

$t(n) \in \Theta(g(n))$ if, and only if, $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$.

Useful Properties of Asymptotic Notation

The asymptotic notation described has the following properties:

1. Transitive (also holds for $\Omega(\cdot)$ and $O(\cdot)$): $t(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n)) \Rightarrow t(n) \in \Theta(h(n))$.
2. Reflexive (also holds for $\Omega(\cdot)$ and $O(\cdot)$): $g(n) \in \Theta(g(n))$
3. Symmetric: $h(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(h(n))$
4. Transpose symmetry: $h(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(h(n))$.

Useful Properties of Asymptotic Notation

Sum of functions

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$$

Idem for $\Omega(\cdot)$ and for $\Theta(\cdot)$.

Proof

$t_1(n) + t_2(n) \leqslant c_1 g_1(n) + c_2 g_2(n)$ for $n \geqslant \max(n_1, n_2)$

$t_1(n) + t_2(n) \leqslant c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)]$, $(c_3 = \max(c_1, c_2))$

$t_1(n) + t_2(n) \leqslant 2c_3 \max(g_1(n), g_2(n))$

$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$ $\square$

Useful Properties of Asymptotic Notation

The previous property is useful when we have algorithms composed of several parts that run sequentially, and whose complexity is known to us.

Example

We want to find duplicates in a non-sorted array of n elements. To this end:

- We firstly sort the array using an algorithm with complexity $O(n \log n)$.
- We compare elements in successive positions, i.e., $O(n)$.

The complexity of the algorithm is dominated by the first part, i.e., $O(n \log n)$.

Useful Properties of Asymptotic Notation

Multiplication of functions

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) \cdot t_2(n) \in O(g_1(n) \cdot g_2(n))$$

Idem for $\Omega(\cdot)$ and $\Theta(\cdot)$.

Proof

$t_1(n) t_2(n) \leqslant c_1 g_1(n) \cdot c_2 g_2(n)$ for $n \geqslant \max(n_1, n_2)$

$t_1(n) t_2(n) \leqslant c_3 g_1(n) g_2(n)$, $(c_3 = c_1 c_2)$

$t_1(n) t_2(n) \in O(g_1(n) g_2(n))$ $\square$

Useful Properties of Asymptotic Notation

This property is useful when we know a part of an algorithm has complexity $O(g_1(n))$ and it is invoked $O(g_2(n))$ times.

Example

We want to use selection sort to sort an array:

- We successively look for the i-th smallest element of the array and place it in position i, $(1 \leqslant i < n) \Rightarrow O(n)$.
- Looking for the smallest of n elements has complexity $O(n)$.

Therefore the overall complexity is $O(n \cdot n) = O(n^2)$.

Useful Properties of Asymptotic Notation

Comparing orders of growth

Let $t(n)$ and $g(n)$ be two orders of growth. The limit of their ratio can take three values:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & (1) \\ c > 0 & (2) \\ \infty & (3) \end{cases}$$

In cases (1) and (2) $t(n) \in O(g(n))$, in cases (2) and (3) $t(n) \in \Omega(g(n))$, and therefore in case (2) $t(n) \in \Theta(g(n))$

Useful Properties of Asymptotic Notation

Rule of L'Hôpital

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

where $f'(n) = df(n)/dn$.

Example

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/\left(2\sqrt{n}\right)} = 2 \lim_{n\to\infty} \frac{\sqrt{n}}{n} = 2 \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

Therefore $\log n \in O(\sqrt{n})$.

Analyzing Recursive Algorithms

Estimating the complexity of an iterative algorithm is often straightforward thanks to the use of arithmetic or geometric series.

The case of recursive algorithms is different. The estimation itself is in general recursive too, and other mathematical tools have to be used.

Factorial

A recursive formulation of the factorial function is as follows:

$$fact(n) = \begin{cases} 1 & n = 0 \\ n \cdot fact(n-1) & n > 0 \end{cases}$$

If multiplication is the basic operation, the complexity of the algorithm is $t(n) = 1 + t(n-1)$ for $n > 0$, with $t(0) = 0$.

Recurrences

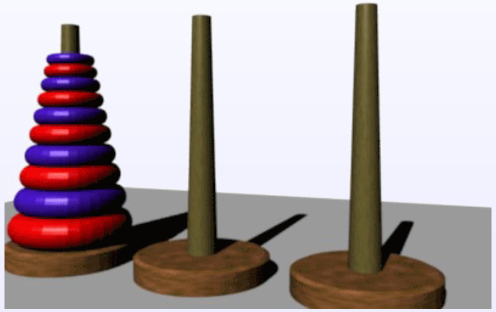
Expressions such as these are termed recurrences.

We have a large mathematical arsenal in order to obtain a closed form for a recurrence. The simplest method is the so-called backwards substitution:

$$t(n) = 1+t(n-1) = 1+1+t(n-2) = 3+t(n-3) = \cdots = i+t(n-i)$$

Since the recurrence has $n = 0$ as base case, we replace i by n in the last term (so that $n - i = 0$) to obtain $t(n) = n + t(0) = n$.

Towers of Hanoi



It is well-known that in order to move $n > 1$ discs from A to C we have to move $n-1$ discs from A to B, 1 disc from A to C and $n-1$ discs from B to C. Hence

$$t(n) = 2t(n-1) + 1.$$

Towers of Hanoi

By backwards substitution:

$$\begin{aligned} t(n) = &\ 2t(n-1) + 1 = 2\,[2t(n-2) + 1] + 1 = \\ = &\ 2^2 t(n-2) + 2 + 1 = 2^2\,[2t(n-3) + 1] + 2 + 1 = \\ = &\ 2^3 t(n-3) + 2^2 + 2 + 1 = \ldots \\ = &\ 2^i t(n-i) + \textstyle\sum_{j=0}^{i-1} 2^j \end{aligned}$$

Replacing i by $n-1$ (so that $n-i=1$) we obtain

$$t(n) = 2^{n-1} t(1) + \sum_{j=0}^{n-2} 2^j = \sum_{j=0}^{n-1} 2^j = 2^n - 1$$

Linear Recurrences

A systematic approach is possible when the recurrence has the form:

$$t(n) = a_{k-1} t(n-1) + a_{k-2} t(n-2) + \cdots + a_0 t(n-k) + f(n)$$

where each a_i is a certain constant. We need in this case k initial values $t(1), \cdots, t(k)$.

If $f(n) = 0$ we have a homogeneous linear recurrence of order k, whose characteristic equation is

$$r^k - a_{k-1} r^{k-1} - a_{k-2} r^{k-2} - \cdots - a_0 = 0$$

Finding the roots of the characteristic equation will solve the recurrence as shown next.

Linear Recurrences

If the roots $\lambda_1, \cdots, \lambda_k$ are all different, the solution to the homogeneous recurrence is

$$t(n) = \alpha_1 \lambda_1^n + \alpha_2 \lambda_2^n + \cdots + \alpha_k \lambda_k^n$$

The values for coefficients α_i depend on the initial conditions. To obtain these we solve the system of equations that results from plugging the initial values in the above expression:

$$\begin{aligned} t(1) = &\ \alpha_1 \lambda_1 + \alpha_2 \lambda_2 + \cdots + \alpha_k \lambda_k \\ t(2) = &\ \alpha_1 \lambda_1^2 + \alpha_2 \lambda_2^2 + \cdots + \alpha_k \lambda_k^2 \\ &\vdots \\ t(k) = &\ \alpha_1 \lambda_1^k + \alpha_2 \lambda_2^k + \cdots + \alpha_k \lambda_k^k \end{aligned}$$

Fibonacci Numbers

Fibonacci numbers

Let $Fib(n) = Fib(n-1) + Fib(n-2)$, with $Fib(1) = 1$, $Fib(2) = 1$. The characteristic equation is

$$r^2 - r - 1 = 0$$

whose roots are $r = \frac{1 \pm \sqrt{5}}{2}$. Hence we obtain

$$Fib(n) = \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Plugging the initial values in the equation we obtain $\alpha_1 = 1/\sqrt{5}$ y $\alpha_2 = -1/\sqrt{5}$.

Linear Recurrences

If any root λ_i is multiple, the corresponding term in the solution is $p_\mu(n)\lambda_i^n$ (rather than λ^n), where μ is the multiplicity of the root, and $p_\mu(n)$ is a $(\mu - 1)$-degree polynomial.

Example

Let $t(n) = 4t(n-1) - 4t(n-2)$, with $t(1) = 1$ and $t(2) = 4$. The characteristic equation is

$$r^2 - 4r + 4 = 0$$

whose double root is $r = 2$. Hence

$$t(n) = (\alpha_1 + \alpha_2 n)\, 2^n$$

Plugging the initial values we get $\alpha_1 = 0$ y $\alpha_2 = 1/2$.

Non-Homogeneous Linear Recurrences

If $f(n)$ is non-zero, the recurrence is non-homogeneous. In this case:

1. We firstly compute the solution to the homogeneous part.
2. Then we compute a particular solution for the whole recurrence $f(n)$, and we sum it to the homogeneous solution.

Example: Towers of Hanoi

Let $t(n) = 2t(n-1) + 1$, with $t(1) = 1$. The characteristic equation of the homogeneous part is $r - 2 = 0$, so the homogeneous solution is $t^{(h)}(n) = \alpha 2^n$.

Since $f(n)$ is a 0-degree polynomial (a constant), we assume a particular solution of the same form $t^{(p)}(n) = c$. Thus:

$$c = 2c + 1 \Rightarrow -c = 1 \Rightarrow c = -1$$

Adding both solutions we get $t(n) = \alpha 2^n - 1$. Plugging the initial condition, $\alpha = 1$.

Non-Homogeneous Linear Recurrences

In general, if $f(n)$ is a polynomial $\alpha_0 + \alpha_1 n + \alpha_2 n^2 + \cdots$ or an exponential function β^n (or a combination of both) we can try with a particular solution with the same structure.

If the base β of the exponential function is also a root of the homogeneous part, we have to increase the polynomial degree as before.

Proposed example

Find a closed form for the following recurrence

$$t(n) = 3t(n-1) + 2^n - n$$

Non-Linear Recurrences

If the recurrence is non-linear, we can try to look for changes of variables or transformations that turn it into a linear recurrence.

For a particular case there is a very useful theoretical results: the Master Theorem.

Master Theorem

Master Theorem

Let $t(n)$ be an asymptotically non-decreasing function for which

$$t(n) = \begin{cases} at(n/b) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

with $a \geqslant 1$, and $b \geqslant 2$. Let $d = \log_b a$. Then,

$$t(n) \in \begin{cases} \Theta(n^d) & \exists\epsilon > 0: \ f(n) \in O(n^{d-\epsilon}) \\ \Theta(n^d \log^{k+1} n) & f(n) \in \Theta(n^d \log^k n) \\ \Theta(f(n)) & \exists\epsilon > 0: \ f(n) \in \Omega(n^{d+\epsilon}) \text{ and} \\ & \exists e < 1: \ \forall n \geqslant n_0: \ af(n/b) \leqslant ef(n) \end{cases}$$

Master Theorem

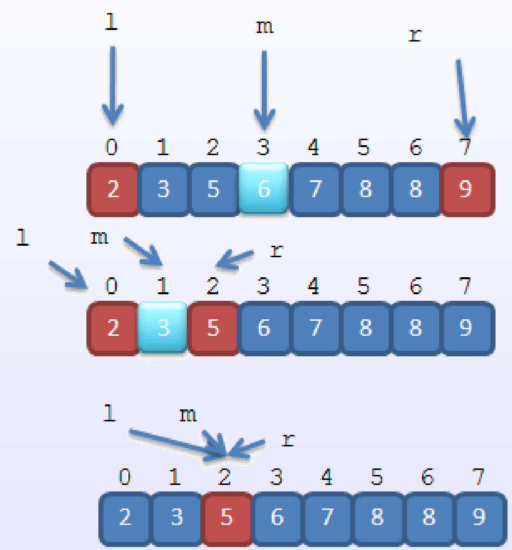
Binary Search

Binary Search

func BinarySearch ($\downarrow e$: $\mathbb{N}$, $\downarrow A$: ARRAY $[1..N]$ OF $\mathbb{N}$): $\mathbb{Z}$
variables l, r, m: $\mathbb{N}$
begin
  $l \leftarrow 1$; $r \leftarrow$ N;
  while $(l \leqslant r) \wedge (A[m] \neq e)$ **do**
    $m \leftarrow (l+r)/2$
    if $A[m] < e$ **then** $l \leftarrow m+1$
    else $r \leftarrow m-1$
  endwhile
  if $A[m] = e$ **then return** m
  else return -1 **endif**
end

Master Theorem

Binary Search at Work



Master Theorem

Binary Search

Binary search

Let $t(n) = t(n/2) + 1$ be the number of comparisons performed by the binary search algorithm in the worst case ($t(1) = 1$). With regard to the Master Theorem we have here that $a = 1$, $b = 2$, and $f(n) = 1$.

Since $f(n) \in \Theta(1) = \Theta(n^0)$, and $d = \log_2 1 = 0$, we are in the second case of the theorem with $k = 0$. Therefore, $t(n) \in \Theta(n^d \log^{k+1} n) = \Theta(\log n)$.

Complementary Bibliography

Articles regarding the empirical analysis of computational complexity:

S. Chakraborty, P.P. Choudhury
"A statistical analysis of an algorithm's complexity",
Applied Mathematics Letters 13(5):121-126, 2000

C. Cotta, P. Moscato
"A mixed evolutionary-statistical analysis of an algorithm's complexity",
Applied Mathematics Letters 16(1):41–47, 2003.

Image Credits

- Picture of al Ḵwārizimī: Soviet Union stamp, public domain.
- Asymptotic notation figures: A. Levitin, © 2007 Pearson Addison-Wesley
- Picture of Towers of Hanoi: GeniXPro at English Wikibooks.