Algorithm complexity analysis

Algoritmos e Estruturas de Dados,
L.EIC, 2021/2022

AP Rocha, P Ribeiro,
R Rossetti, F Ramos, L Grácio, A Costa, S Martins

Algorithm

- **Algorithm**

  set of precise instructions for solving a problem

algorithm $\neq$ program

<u>Algorithm analysis</u>:

- *Correctness*: prove the algorithm is correct
- *Efficiency*: determine the resources requested by the algorithm (time, space)
  - Compare the resource requested by different algorithms that solve the same problem: a more efficient algorithm requires less resources
  - Predict the increasing of required resources as the input size increases

Complexity

- Algorithm space complexity:

  memory space needed to execute

  S(n) – memory space required depending on input size (*n*)

- Algorithm temporal complexity:

  time it takes to execute

  T(n) - execution time depending on input size (*n*)

Complexity ↑ versus Eficiência ↓

Sometimes, complexity is calculated for the “*best case*” (not too useful), the *“worst case"* (more useful) and the *“average case"* (equally useful)

Complexity

- In general, we are not so much interested in the time and space complexity for small inputs

- What is important is the **growth** of the complexity functions
  - The growth of time and space complexity with increasing input size *n* is a suitable measure for the comparison of algorithms.

- Evaluate growth rate
  - As a function of various terms, growth is determined by the fastest growing term (*dominant term*)
  - Constant coefficients influence the initial progress

*Dominant term

Supose you use n^3 to estimate $n^3+350n^2 + n$

- for n = 10 000
  - real value = 1 0003 5000 010 000
  - estimated value = 1 000 000 000 000
  - error = 0.35% (not significant)

- for high values of n
  - the dominant term is indicative of the behavior of the algorithm

- for small values of n
  - The dominant term is not necessarily indicative of the behavior, but usually programs run so quickly that it doesn´t matter

The growth of functions

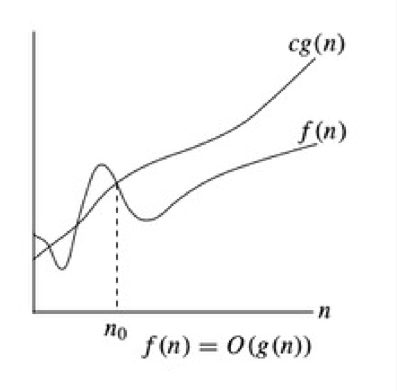
The growth of functions is usually described using the big-O notation

- **Definition**

  $f(n) = O(g(n))$

  if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$, for all $n > n_0$



- The idea behind the *big-O* notation is to establish an upper boundary for the growth of a function $f(n)$ for large n
  - This boundary is specified by a function $g(n)$ that is usually much simpler than $f(n)$
  - We accept the constant c in the requirement $f(n) \leq cg(n)$ whenever $n > n_0$, because c does not grow with n
  - We are only interested in large n, so it is OK if $f(n) > cg(n)$ for $x \leq n_0$

The growth of functions

Example: $f(n) = n^2 + 2n + 1$

- for $n > 1$:

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$$
$$\Rightarrow n^2 + 2n + 1 \leq 4n^2$$

- therefore, for $c=4$ and $n_0=1$: $f(n) \leq cn^2$, whenever $n>n_0$

$$\Rightarrow f(n) = O(n^2)$$

Question: if $f(n)$ is $O(n^2)$, is it also $O(n^3)$?

- yes; n^3 grows faster than n^2, so n^3 grows also faster than $f(n)$
- therefore, we always have to find the smallest simple function $g(n)$ for which $f(n)$ is $O(g(n))$

The growth of functions

more examples:

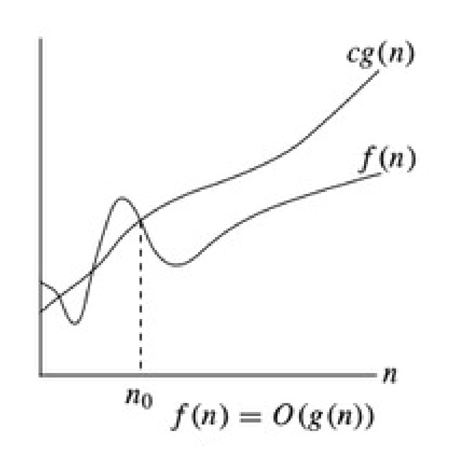
- $c_k\, n^k + c_{k-1}\, n^{k-1} + \ldots + c_0 = O(n^k)$ (c_i - constants)
- $\log_2 n = O(\log n)$ (changing the base is to multiply by a constant)
- $4 = O(1)$ (use 1 for constant order)

Big-O notation

Notation for functions growth

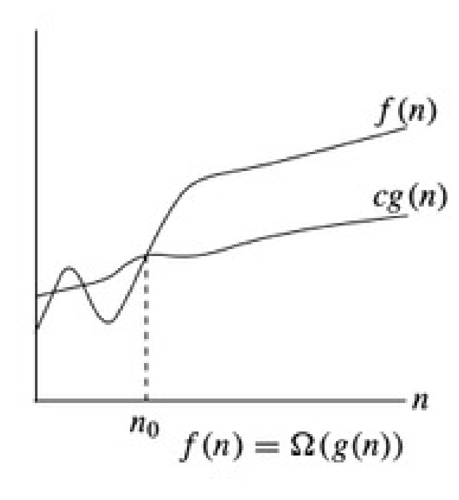
- $f(n) = O(g(n))$

  if there are positive constants c and n_0 such that $f(n) \leq cg(n)$, for $n \geq n_0$



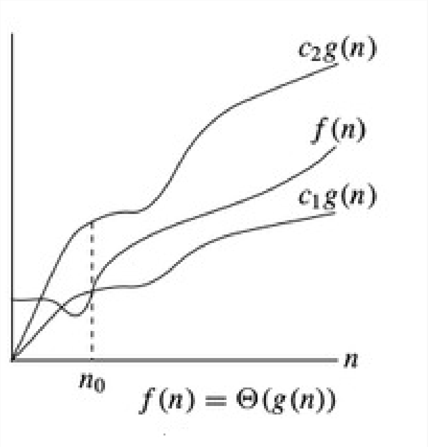
- $f(n) = \Omega(g(n))$

  if there are positive constants c and n_0 such that $f(n) \geq cg(n)$, for $n \geq n_0$



Big-O notation

Notation for functions growth



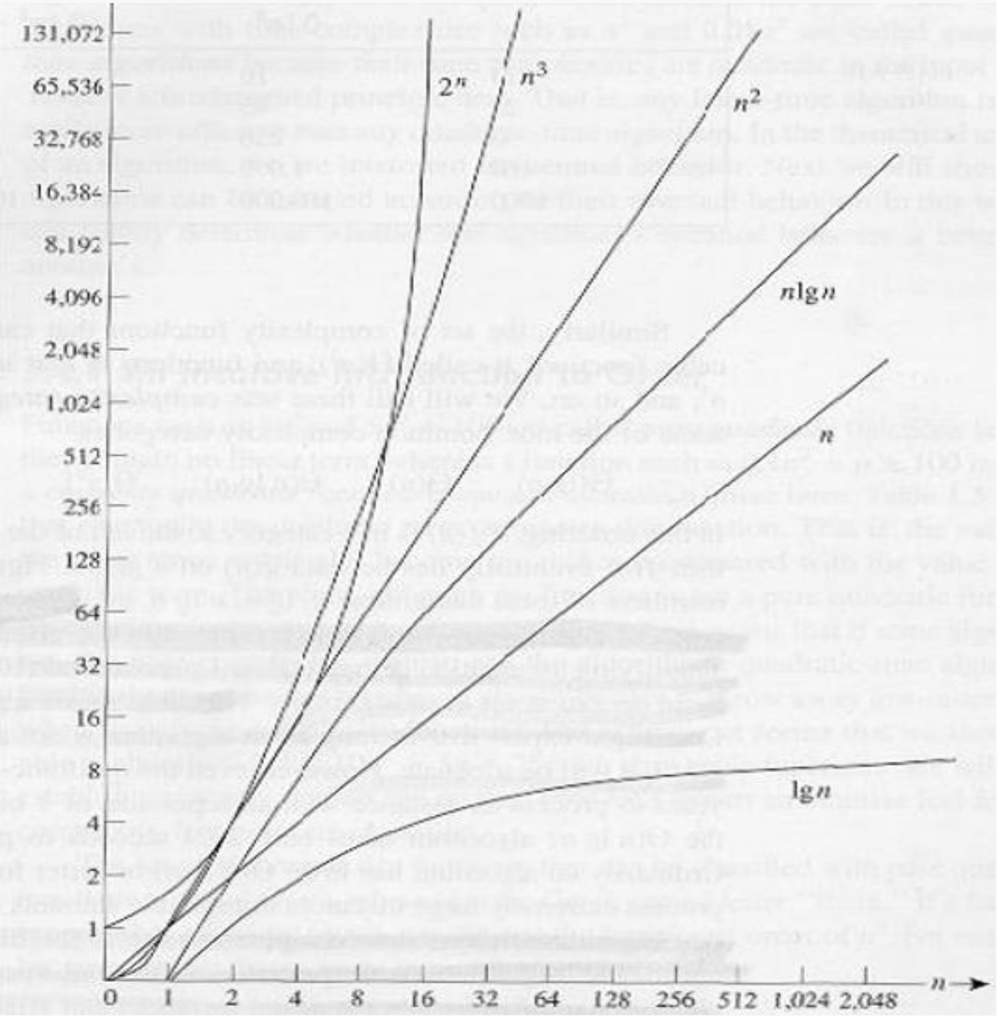
– $f(n) = \Theta(g(n))$

if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

– $f(n) = o(g(n))$

if there are positive constants c and n_0 such that $f(n) < cg(n)$, for $n \geq n_0$

Most common orders of growth



Case study: maximum subsequence

- Problem
  - Given a set of integer values (positive and/or negative) $a_1, a_2, \ldots, a_n$, determine the highest subsequence sum
  - The largest sum subsequence is zero if all values are negative

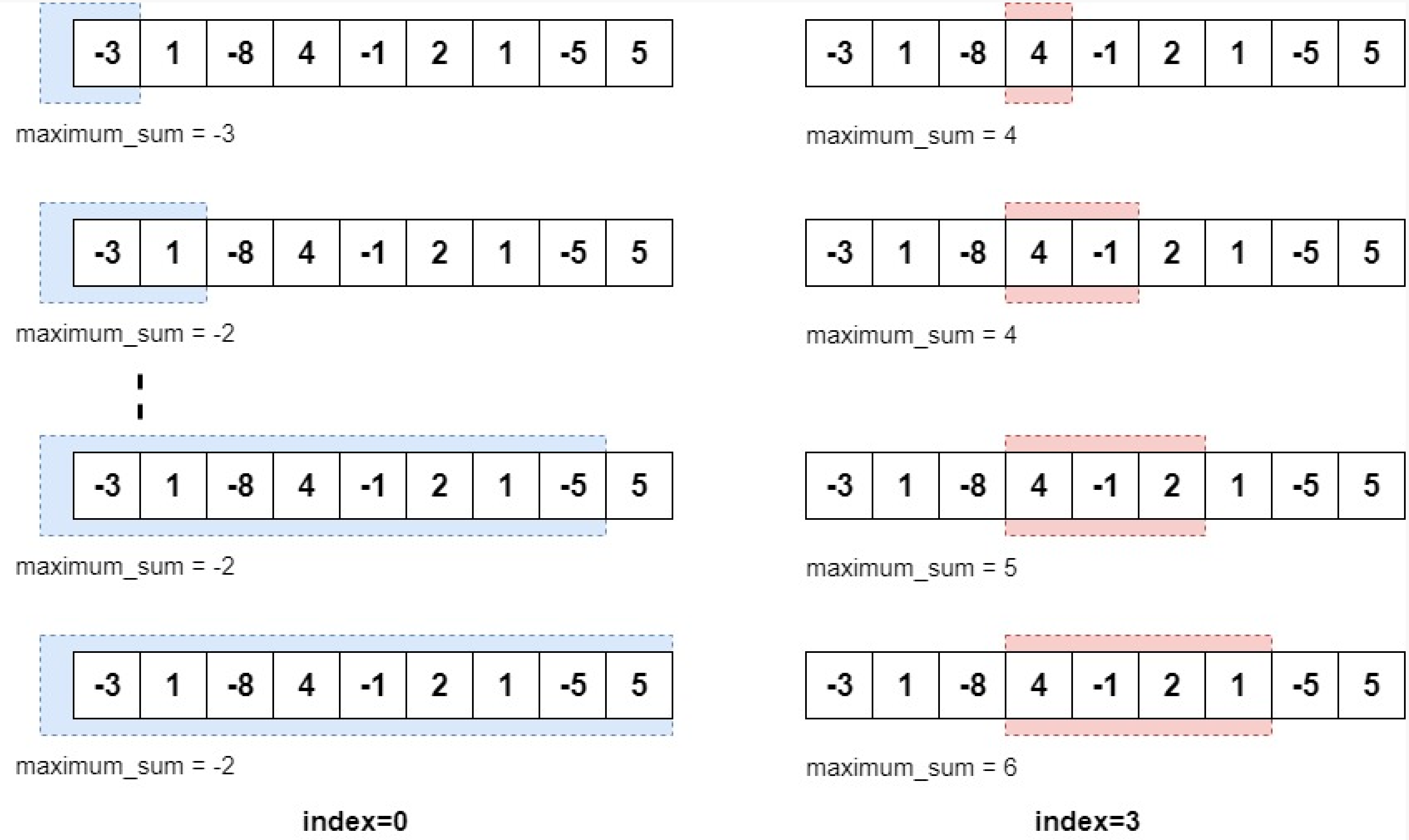
Examples

-2, <u>11, -4, 13</u>, -4, 2

1, -3, <u>4, -2, -1, 6</u>

-3, 1, -8, <u>4, -1, 2, 1</u>, -5, 5

Maximum subsequence – algorithm 1



Maximum subsequence – algorithm 1

```
template <class Comparable>
Comparable maxSubSum1(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
       for (int j = i; j < a.size(); j++)
       {
          Comparable thisSum = 0;
          for (int k = i; k <= j; k++)
             thisSum += a[k];
          if (thisSum > maxSum)
             maxSum = thisSum;
       }
    return maxSum;
}
```

Maximum subsequence – algorithm 1

- Temporal complexity analysis
  - cycle of n iterations within another cycle of n iterations within another cycle of n iterations → $O(n^3)$
  - value estimated by excess, some cycles have less than n iterations

- How to improve
  - remove a cycle
  - inner cycle is not necessary
  - *thisSum* for next *j* can be easily calculated from the old value of *thisSum*

Maximum subsequence – algorithm 2

```
template <class Comparable>
Comparable maxSubSum2(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
    {
        Comparable thisSum = 0;
        for (int j = i; j < a.size(); j++)
        {
            thisSum += a[j];
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    }
    return maxSum;
}
```

Maximum subsequence – algorithm 2

- Temporal complexity analysis
  - cycle of *n* iterations within another cycle of *n* iterations → $O(n^2)$
  - value estimated by excess, some cycles have less than *n* iterations

- Is it possible to improve?
  - linear algorithm is better: execution time is proportional to input size (hard to do better)
  - if a_{ij} is a subsequence with negative cost, a_{iq} with $q>j$ is not the maximum subsequence

Maximum subsequence – algorithm 3

```
template <class Comparable>
Comparable maxSubSum3(const vector<Comparable> &a)
{
    Comparable thisSum = 0; Comparable maxSum = 0;
    for (int j=0 ; j < a.size() ; j++)
     {
       thisSum += a[j];
       if (thisSum > maxSum)
           maxSum = thisSum;
       else if (thisSum < 0)
           thisSum = 0;
     return maxSum;
}
```