Scientific Programming

Algorithm analysis

Alberto Montresor

Università di Trento

2018/11/17

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Maximal sum problem

**Input:** a list $A$ containing $n$ numbers

**Output:** a slice (sublist) $A[i : j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Is the problem clear?
Maximal sum problem

Input: a list $A$ containing $n$ numbers

Output: a slice (sublist) $A[i : j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Is the problem clear?

| 1 | 3 | 4 | -8 | 2 | 3 | -1 | 3 | 4 | -3 | 10 | -3 | 2 |
Maximal sum problem

**Input**: a list $A$ containing $n$ numbers

**Output**: a slice (sublist) $A[i : j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Is the problem clear?

```
1  3  4  -8  2  3  -1  3  4  -3  10  -3  2
```
Maximal sum problem

**Input**: a list $A$ containing $n$ numbers

**Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Is the problem clear?
Can you solve it?
Can you solve it in an efficient way?
Loop over every pair \((i, j)\) such that \(i \leq j\):

- Extract the slice \(A[i : j]\) (i.e., elements between \(i\) and \(j\), included)
- Compute the sum of the slice using \texttt{sum}()
- Update \texttt{maxSoFar} if the sum is larger than the maximum found so far

```python
def maxsum1(A):
    n = len(A)
    maxSoFar = 0  # Maximum found so far
    for i in range(0, n):
        for j in range(i, n):
            tot = sum(A[i:j+1])
            maxSoFar = max(maxSoFar, tot)
    return maxSoFar
```
Version 1 (BIS) – $O(n^3)$

We could use list comprehension to generate all the possible pairs, storing their sums in a list, and taking the maximum from that. Behold! For a list of size $n = 10^5$, this would require around 40 GB of RAM...

```python
def maxsum1(A):
    n = len(A)
    return max(
        sum(A[i:j+1])
        for i in range(0,n) for j in range(i,n) )
```

We could even store the slices in a list, and then apply `sum()` with another list comprehension. Behold! For a list of size $n = 10^5$, this would require 1.3 PB of RAM....

```python
def maxsum1(A):
    n = len(A)
    slices = [ A[i:j+1] for i in range(0,n) for j in range(i,n) ]
    sums = [ sum(sl) for sl in slices ]
    return max(sums)
```
Version 2 – $O(n^2)$

If the sum $s$ of the slice $A[i : j]$ has already been computed, the sum of $A[i : j + 1]$ can be computed as $s + A[j]$. In other words, we can use an accumulator to avoid computing over and over the same partial sums.

```python
def maxsum2(A):
    n = len(A)
    maxSoFar = 0  # Maximum found so far
    for i in range(0,n):
        tot = 0  # Accumulator
        for j in range(i,n):
            tot = tot + A[j]
            maxSoFar = max(maxSoFar, tot)
    return maxSoFar
```
Version 2 (BIS) – $O(n^2)$

The `accumulate()` function of the `itertools` module takes a list $I$ as input and returns a list $O$ as output such that $O[k] = \sum_{i=0}^{k} I[i]$.

It can substitute the accumulator in the previous code; it is normally faster because the underlying implementation is C-based.

```python
from itertools import accumulate

def maxsum2(A):
    n = len(A)
    maxSoFar = 0  # Maximum found so far
    for i in range(0, n):
        tot = max(accumulate(A[i:]))
        maxSoFar = max(maxSoFar, tot)
    return maxSoFar
```

Alberto Montresor (UniTN)
SP - Algorithm analysis
2018/11/17
Version 3 – $O(n \log n)$

Divide-et-impera

- Split the list in two parts (right, left) of (almost) equal size
- $\text{maxL}$ is the sum of the maximal sublist on the left part
- $\text{maxR}$ is the sum of the maximal sublist on the right part
- Is it correct?

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$\text{maxL}$ $\text{maxR}$
Version 3 – $O(n \log n)$

**Divide-et-impera**

- Split the list in two parts (right, left) of (almost) equal size
- $\text{maxL}$ is the sum of the maximal sublist on the left part
- $\text{maxR}$ is the sum of the maximal sublist on the right part
- $\text{maxLL} + \text{maxRR}$ is the value of the maximal sublist that spans the two parts
```python
def maxsum_rec(A, i, j):
    if (i==j):
        return max(0, A[i])
    m = (i+j)//2
    maxLL = 0  # Maximal subvector on the left ending in m
    sum = 0
    for k in range(m, i-1,-1):
        sum = sum + A[k]
        maxLL = max(maxLL, sum);
    maxRR = 0  # Maximal subvector on the right starting in m+1
    sum = 0
    for k in range(m+1,j+1):
        sum = sum + A[k]
        maxRR = max(maxRR, sum);
    maxL = maxsum_rec(A, i, m)  # Maximal subvector on the left
    maxR = maxsum_rec(A, m+1, j)  # Maximal subvector on the right
    return max(maxL, maxR, maxLL + maxRR)

def maxsum3(A):
    return maxsum_rec(A,0,len(A)-1)
```
Version 3 – $O(n \log n)$

The `accumulate()` function can be used also here. Please note the strange slicing needed to compute `maxLL`; this because the slice `A[m:i-1:-1]` is empty when `i=0`, because `-1` refers to the last element.

```python
from itertools import accumulate

def maxsum3_rec(A, i, j):
    if i==j:
        return max(0, A[i])
    m = (i+j)//2
    maxL = maxsum3_rec(A, i, m)
    maxR = maxsum3_rec(A, m+1, j)
    maxLL = max(accumulate(A[m:-len(A)+i-1:-1]))
    maxRR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxLL+maxRR)
```

Alberto Montresor (UniTN)
Version 4 – $O(n)$

We apply here a technique called **dynamic programming**. We compute $\text{maxHere}$ as the value of the maximum sublist that ends in position $i$.

$$
\text{maxHere}[i] = \begin{cases} 
0 & i < 0 \\
\max(\text{maxHere}[i - 1] + A[i], 0) & i \geq 0
\end{cases}
$$

The final result is given by the maximum slice that ends in any position.

```python
def maxsum4(A):
    maxSoFar = 0 # Maximum found so far
    maxHere = 0 # Maximum slice ending at the current pos
    for number in A:
        maxHere = max(maxHere + number, 0)
        maxSoFar = max(maxHere, maxSoFar)
    return maxSoFar
```
**Version 4 – $O(n)$**

\[
A = [ 1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2 ]
\]

\[
\begin{array}{cccccccccccccc}
\text{maxHere} & 0 & 1 & 4 & 8 & 0 & 2 & 5 & 4 & 7 & 11 & 8 & 18 & 15 & 17 \\
\text{maxSoFar} & 0 & 1 & 4 & 8 & 8 & 8 & 8 & 8 & 8 & 11 & 11 & 18 & 18 & 18 \\
\end{array}
\]

The column associated to each list entry contains the value of \(\text{maxHere}\) and \(\text{maxSoFar}\) after the execution of the loop on that list entry.
Same technique as before, but returning the slice indexes rather than the sum of the slice.

```python
def maxsum4_slice(A):
    maxSoFar = 0  # Maximum found so far
    maxHere = 0   # Maximum slice ending at the current pos
    start = end = 0  # Start, end of the maximal slice found so far
    last = 0       # Beginning of the maximal slice ending here

    for i in range(len(A)):
        maxHere = maxHere + A[i]
        if maxHere <= 0:
            maxHere = 0
            last = i+1
        if maxHere > maxSoFar:
            maxSoFar = maxHere
            start, end = last, i

    return (start, end)
```
Version 4 – $O(n)$

\[
A = [ 1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2 ]
\]

\[
\text{maxHere} = 0 \quad 1 \quad 4 \quad 8 \quad 0 \quad 2 \quad 5 \quad 4 \quad 7 \quad 11 \quad 8 \quad 18 \quad 15 \quad 17
\]

\[
\text{maxSoFar} = 0 \quad 1 \quad 4 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 11 \quad 11 \quad 18 \quad 18 \quad 18 \quad 18
\]

\[
\text{last} = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4
\]

\[
\text{start} = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4
\]

\[
\text{end} = 0 \quad 0 \quad 1 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 8 \quad 8 \quad 10 \quad 10 \quad 10 \quad 10
\]

The column associated to each list entry contains the value of the variables after the execution of the loop on that list entry.
Execution times

Time (ms)

Size

0 2000 4000 6000 8000 10000

$O(n^3)$

$O(n^2)$

$O(n^2)$ (accumulate)

$O(n \log n)$

$O(n \log n)$ (accumulate)

$O(n)$
Historic background

From an educational point of view, the best problem ever!

- The problem has been formulated in 1977 by Ulf Granander (Brown University), as a simplified version of a more general 2D problem (maximum likelihood in image processing).
- A linear algorithm for the 1D version has been proposed by Jay Kadane (Carnegie Mellon University) in 1984.
- The best algorithm for the 2D version is $O(n^3)$, with input size $n^2$.
- The story is narrated by Jon Bentley in his columns in Communication of the ACM.

Historic background

Genome Sequence Analysis

"One line of investigation in genome sequence analysis is to locate the biologically meaningful segments, like conserved regions or GC-rich regions. A common approach is to assign a real number (also called score) to each residue, and then look for the maximum-sum segment."

# Table of contents

1. Teaser
2. Introduction
   - Efficiency
   - Correctness
3. Complexity analysis
   - Definitions
   - Analysis examples
4. Asymptotic notation
   - Notation
   - Properties of the asymptotic notation
   - Recurrences
5. Problem vs algorithm complexity
   - Binary sum
   - Binary product
6. Sorting algorithms
   - Selection Sort
   - Insertion Sort
   - MergeSort
   - Quicksort
Introduction

Computational problem

The formal relationship between the input and the desired output

Algorithm

- The description of the sequence of actions that an executor must execute to solve the problem
- Among their tasks, algorithms represent and organize the input, the output, and all the intermediate data required for the computation
Algorithms in history

- Ahmes’ Papyrus (1850 BC, peasant algorithm for multiplication)
- Numerical algorithms have been studied by Babylonians and Indian mathematicians
- Algorithms used even today have been studied by Greek mathematicians more than 2000 years ago
  - Euclid’s Algorithm for the greatest common divisor
  - Geometrical algorithms (angle bisection and trisection, tangent drawing, etc)
Name origin

Abu Abdullah Muhammad bin Musa al-Khwarizmi

- He was a Persian mathematician, astronomer, astrologer, geographer
- He introduced the Indian numbers in the western world
- From his name: algorithm

Algoritmi de numero indorum

- Latin translation of an Arabic text now lost
- He introduce the Indian (Arab) numbers in the west
- From the Arab number sifr = 0: zephirum → zevero → zero, but also cifra
Introduction

Name origin

Abu Abdullah Muhammad bin Musa al-Khwarizmi

- He was a Persian mathematician, astronomer, astrologer, geographer
- He introduced the indian numbers in the western world
- From his name: algorithm

Al-Kitab al-muhtasar fi hisab al-gabr wa-l-muqabala

- His most famous work (820 AC)
- Translated in Latin with the title: Liber algebræ et almucabala
Computational problems: examples

Minimum

The minimum of a set $S$ is the element of $S$ which is smaller or equal that any other element of $S$.

$$ \min(S) = a \iff \exists a \in S : \forall b \in S : a \leq b $$

Lookup

Let $S = s_0, s_1, \ldots, s_{n-1}$ be a sequence of distinct, sorted numbers, i.e. $s_0 < s_1 < \ldots < s_{n-1}$. To perform a lookup of the position of value $v$ in $S$ corresponds to returning the index $i$ such that $0 \leq i < n$, if $v$ is contained at position $i$, $-1$ otherwise.

$$ \text{lookup}(S, v) = \begin{cases} 
 i & \exists i \in \{0, \ldots, n - 1\} : S_i = v \\
 -1 & \text{otherwise}
\end{cases} $$
Computational problems: naive solutions

**Minimum**

To find the minimum of a set, compare each element with every other element; the element that is smaller than any other is the minimum.

**Lookup**

To find a value $v$ in the sequence $S$, compare $v$ with any other element of $S$, in order, and return the corresponding index if a correspondence is found; returns $-1$ if none of the elements is equal to $v$. 
Computational problems: naive solutions

**Minimum**

```python
def min(S):
    isMin = True
    for y in S:
        if x > y:
            isMin = False
    if (isMin):
        return x
```

**Lookup**

```python
def lookup(S, v):
    for i in range(len(S)):
        if (S[i] == v):
            return i
    return -1
```
How to evaluate an algorithm

Does it solve the problem in a **correct** way?
- Mathematical proof vs informal description
- Some problems can only be solved in an approximate way
- Some problems cannot be solved at all

Does it solve the problem in an **efficient** way?
- How to measure efficiency
- Some solutions are **optimal**: you cannot find better solutions
- For some problems, there are no efficient solutions
Other properties

Simplicity, modularity, maintainability, expandability, robustness, \ldots

- Fundamental properties in a course on software engineering
- Secondary importance in a course of algorithm and data structures

Comment

Some of these properties have an additional cost from the point of view of performance:

- Modular code $\rightarrow$ call overhead
- Java bytecode $\rightarrow$ interpretation overhead

*The design of efficient algorithms is a prerequisite to pay these additional costs*
Passages from the Life of a Philosopher, Charles Babbage, 1864

As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise — By what course of calculation can these results be arrived at by the machine in the shortest time?

Model of the analytical machine, London Museum of Science, picture Bruno Barral

Charles Babbage, 1860
Algorithm evaluation – Efficiency

Algorithm complexity

Analysis of the resources employed by an algorithm to solve a problem, depending on the size and the type of input.

Resources

- **Time**: time needed to execute the algorithm
  - Should we measure it with a cronometer?
  - Should I measure it by counting the number of elementary operations?
- **Space**: amount of used memory
- **Bandwidth**: amount of bit transmitted (distributed algorithms)
Algorithm evaluation – Minimum

Let’s count the number of comparisons for the minimum solution

```python
def min(S):
    isMin = True
    for y in S:
        if x > y:
            isMin = False
    if (isMin):
        return x
```

"Naïf" algorithm:

$n^2$
Algorithm evaluation – Minimum

Let’s count the number of comparisons for the minimum solution

```python
def min(S):
    isMin = True
    for x in S:
        for y in S:
            if x > y:
                isMin = False
    if (isMin):
        return x
```

• “Naïf” algorithm: $n^2$

**Question**

Why counting the comparisons?

**Question**

Can do better than this?
Algorithm evaluation – A better solution

Let’s count the number of comparisons for the minimum solution

```python
def min(S):
    # Partial minimum
    minSoFar = S[0]
    i = 1
    while i < len(S):
        if S[i] < minSoFar:
            # Update partial minimum
            minSoFar = S[i]
        i = i + 1
    return minSoFar
```

“Naïf” algorithm: $n^2$
Algorithm evaluation – A better solution

Let’s count the number of comparisons for the minimum solution

```python
def min(S):
    # Partial minimum
    minSoFar = S[0]
    i = 1
    while i < len(S):
        if S[i] < minSoFar:
            # Update partial minimum
            minSoFar = S[i]
            i = i + 1
    return minSoFar
```

- “Naïf” algorithm: $n^2$
- Efficient algorithm: $n - 1$
Algorithm evaluation – Lookup

Let’s count the number of comparisons for the lookup solution

```python
def lookup(S, v):
    for i in range(len(S)):
        if (S[i] == v):
            return i
    return -1
```
Algorithm evaluation – Lookup

Let’s count the number of comparisons for the lookup solution

```python
def lookup(S, v):
    for i in range(len(S)):
        if (S[i] == v):
            return i
    return -1
```

“Naïf” algorithm: $n$
Algorithm evaluation – A better solution

A more efficient solution

Let's consider the median \( m \) element of the list

- If \( S[m] = v \), the looked-up element has been found
- If \( v < S[m] \), look in the "left part"
- If \( S[m] < v \), look in the "right part"

```
1 5 12 15 20 23 32
```

21?
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median $m$ element of the list

- If $S[m] = v$, the looked-up element has been found
- If $v < S[m]$, look in the "left part"
- If $S[m] < v$, look in the "right part"

$m$

\[
\begin{array}{ccccccc}
1 & 5 & 12 & 15 & 20 & 23 & 32 \\
\end{array}
\]

21?
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median \( m \) element of the list

- If \( S[m] = v \), the looked-up element has been found
- If \( v < S[m] \), look in the "left part"
- If \( S[m] < v \), look in the "right part"
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median \( m \) element of the list

- If \( S[m] = v \), the looked-up element has been found
- If \( v < S[m] \), look in the "left part"
- If \( S[m] < v \), look in the "right part"
Algorithm evaluation – A better solution

A more efficient solution

Let's consider the median $m$ element of the list

- If $S[m] = v$, the looked-up element has been found
- If $v < S[m]$, look in the "left part"
- If $S[m] < v$, look in the "right part"

$$m$$

1 5 12 15 20 23 32

21?
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median \( m \) element of the list

- If \( S[m] = v \), the looked-up element has been found
- If \( v < S[m] \), look in the "left part"
- If \( S[m] < v \), look in the "right part"

\[
\begin{array}{ccccccc}
1 & 5 & 12 & 15 & 20 & 23 & 32 \\
\end{array}
\]

\( m \)

21?
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median $m$ element of the list

- If $S[m] = v$, the looked-up element has been found
- If $v < S[m]$, look in the "left part"
- If $S[m] < v$, look in the "right part"

```
1  5  12  15  20  23  32
```

21?
Algorithm evaluation – A better solution

A more efficient solution

Let’s consider the median $m$ element of the list

- If $S[m] = v$, the looked-up element has been found
- If $v < S[m]$, look in the "left part"
- If $S[m] < v$, look in the "right part"

$m$

1 5 12 15 20 23 32

21?
Binary search – Recursive version

Let’s count the number of comparisons for the `lookup_rec()` solution.

```python
def lookup_rec(S, start, finish, v):
    if (finish < start):
        return -1
    else:
        mid = (start+finish) // 2
        if S[mid] == v:
            return mid
        elif v < S[mid]:
            return lookup_rec(S, start, mid-1, v)
        else:
            return lookup_rec(S, mid+1, finish, v)
```

“Naïf” algorithm: \( n \)
Binary search – Recursive version

Let’s count the number of comparisons for the lookup_rec() solution.

```python
def lookup_rec(S, start, finish, v):
    if (finish < start):
        return -1
    else:
        mid = (start+finish) // 2
        if S[mid] == v:
            return mid
        elif v < S[mid]:
            return lookup_rec(S, start, mid-1, v)
        else:
            return lookup_rec(S, mid+1, finish, v)
```

- “Naïf” algorithm: $n$
- Binary search: $\lceil \log n \rceil$
Binary search – Recursive version

Let’s count the number of comparisons for the `lookup_rec()` solution.

```python
def lookup_rec(S, start, finish, v):
    if (finish < start):
        return -1
    else:
        mid = (start+finish) // 2
        if S[mid] == v:
            return mid
        elif v < S[mid]:
            return lookup_rec(S, start, mid-1, v)
        else:
            return lookup_rec(S, mid+1, finish, v)
```

- “Naïf” algorithm: $n$
- Binary search: $\lceil \log n \rceil$
Algorithm evaluation – Correctness

**Invariant**
A condition that is always true in a specific point in an algorithm

**Loop invariant**
- A condition that is always true at the beginning of a loop iteration
- what is exactly the beginning of a loop iteration?

**Class invariant**
- A condition always true when the execution of a class method is completed
Algorithm evaluation – Correctness

The concept of loop invariant help us in proving that the algorithm is correct:

- **Initialization** (base case): The condition is true before the first iteration
- **Conservation** (inductive step): If the condition is true before the iteration of the loop, then it remains true at the end (before the next iteration)
- **Conclusion**: When the loop is finished, the invariant must represent the "correctness" of the algorithm
Algorithm evaluation – Correctness

At the beginning of each iteration of the while loop, variable minSoFar contains the partial minimum of the elements in S[0:i].

def min(S):
    minSoFar = S[0] # Partial minimum
    i = 1
    while i < len(S):
        if S[i] < minSoFar:
            minSoFar = S[i] # Update partial minimum
            i = i + 1
    return minSoFar

Exercise

Prove that the invariant of min() is respected.
Algorithm evaluation – Correctness

Exercise

- Prove the correctness of `lookup()`
- Think about a loop invariant that applies here

```python
def lookup(S, v):
    start = 0
    finish = len(S)-1
    mid = (start + finish) // 2
    while start <= finish and S[mid] != v:
        if v < S[mid]:
            finish = mid-1
        else:
            start = mid+1
        mid = (start+finish) // 2
    return mid if S[mid] == v else -1
```
Algorithm evaluation – Correctness

Exercise

- Prove the correctness of `lookup_rec()`
- By induction on the size of the current slice

```python
def lookup_rec(S, start, finish, v):
    if (finish<start):
        return -1
    else:
        mid = (start+finish) // 2
        if S[mid] == v:
            return mid
        elif v < S[mid]:
            return lookup_rec(S, start, mid-1, v)
        else:
            return lookup_rec(S, mid+1, finish, v)
```
Binary search – Recursive version

By induction on the size $n$ of the current slice

- **Base case** ($n = 0$): if $n = 0$, this means that $\text{finish} < \text{start}$. The algorithm returns $-1$, which is correct given that if $n = 0$, $v$ is not present.

- **Inductive hypothesis**: given a size $n$, let assume that the algorithm is correct for all sizes $n' < n$

- **Inductive step**: given a size $n > 0$, let $\text{mid}$ be the median element.
  
  - If $S[\text{mid}] == v$, then the algorithm returns $\text{mid}$, because $\text{mid}$ is the actual position of $v$
  
  - If $v < S[\text{mid}]$, then if $v$ is present, it must be located in $S[\text{start}:\text{mid}]$. By inductive hypothesis, $\text{lookup}\_\text{rec}(S, \text{start}, \text{mid}-1, v)$ will return the correct position of $v$ if present, or $-1$ if not present.
  
  - Symmetrically for $v > S[\text{mid}]$. 
Table of contents

1 Teaser
2 Introduction
   • Efficiency
   • Correctness
3 Complexity analysis
   • Definitions
   • Analysis examples
4 Asymptotic notation
   • Notation
   • Properties of the asymptotic notation
   • Recurrences
5 Problem vs algorithm complexity
   • Binary sum
   • Binary product
6 Sorting algorithms
   • Selection Sort
   • Insertion Sort
   • MergeSort
   • Quicksort
Introduction

Goal: *estimate the complexity in time of algorithms*

- Definitions
- Computing models
- Evaluation examples
- Notation

Why?

- To estimate the time needed to process a given input
- To estimate the largest input computable in a reasonable time
- To compare the efficiency of different algorithms
- To optimize the most important part
Definition: Complexity: "Input size" $\rightarrow$ "Time"

- How to measure the size of the input?
- How to measure time?
Definition of input size

**Logarithmic cost model**
- The input size is equal to the number of bits representing it
- Example: binary number multiplication of numbers of $n$ bits

**Uniform cost model**
- The input size is equal to the number of elements composing it
- Example: minimum search in a list of $n$ elements

**In several cases...**
- We can assume that the elements are represented by a constant number of bits
- The two measures are the same, apart from a constant multiplication factor
Definition of time

**Time \equiv \text{wall-clock time}**

The actual time used to complete an algorithm

It depends on too many parameters:

- how good is the programmer
- programming language
- code generated by the compiler/interpreter
- CPU, memory, hard-disk, etc.
- operating system, other processes currently running, etc.

We need to adopt a more abstract representation
Random Access Model (RAM) - Time definition

Time $\equiv$ number of basic instructions

An instruction is considered basic if it can be executed in constant time by the processor

Basic

- $a = a*2$ ? Yes
- $\text{math.cos}(d)$ ? Yes
- $\text{min}(A)$ ? No
Computation time of \textit{min}()

- Each statement requires a constant time to be executed
- This constant may be different for each statement
- Each statement is executed a given number of times, function of \( n \)

\begin{verbatim}
def min(S):
    minSoFar = S[0]
    i = 1
    while i < len(S):
        if S[i] < minSoFar:
            minSoFar = S[i]
        i = i + 1
    return minSoFar
\end{verbatim}

\begin{tabular}{|c|c|c|}
\hline
Cost & N. of times \\
\hline
\hline
\end{tabular}

\[ T(n) = c_1 + c_2 + c_3 n + c_4 (n - 1) + c_5 (n - 1) + \ldots + c_6 (n - 1) + c_7 \]

\[ = (c_3 + c_4 + c_5 + c_6) n + (c_1 + c_2 - c_4 - c_5 - c_6 + c_7) = an + b \]
## Computation time of `binarySearch()`

The list is divided in two parts:

- **SX part:** \([n - 1]/2\)
- **DX part:** \(n/2\)

```python
def lookupr(S, s, f, v):
    if (f < s):
        return -1
    m = (s+f) // 2
    if S[m] == v:
        return m
    elif v < S[m]:
        return lookupr(S, s, m-1, v)
    else:
        return lookupr(S, m+1, f, v)
```

<table>
<thead>
<tr>
<th>Cost</th>
<th>#s &gt; f</th>
<th>#s ≥ f</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(c_2)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(c_3)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(c_4)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(c_5)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(c_6)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(c_7 + T([n - 1]/2]))</td>
<td>0</td>
<td>0/1</td>
</tr>
<tr>
<td>(c_7 + T([n/2])))</td>
<td>0</td>
<td>1/0</td>
</tr>
</tbody>
</table>
**Computation time of `binarySearch()`**

- **Assumptions (worst cases):**
  - For simplicity, let us assume that $n$ is a power of 2: $n = 2^k$
  - The searched element is not present
  - At each call, we select the right part whose size is $n/2$

- **Two cases:**
  
  \[
  s > f \quad (n = 0) \quad T(n) = c_1 + c_2 = c
  \]

  \[
  s \leq f \quad (n > 0) \quad T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 \\
  = T(n/2) + d
  \]

- **Recurrence relation:**

  \[
  T(n) = \begin{cases} 
  c & n = 0 \\
  T(n/2) + d & n \geq 1 
  \end{cases}
  \]
Computation time of `binarySearch()`

Solution of the recurrence relation:

\[ T(n) = T(n/2) + d \]

\[ = (T(n/4) + d) + d = T(n/4) + 2d \]

\[ = (T(n/8) + d) + 2d = T(n/8) + 3d \]

\[ \vdots \]

\[ = T(1) + kd \]

\[ = T(0) + (k + 1)d \]

\[ = kd + (c + d) \]

\[ = d \log n + e. \]
Scientific Programming

Algorithm analysis
Asymptotic notation

Alberto Montresor
Università di Trento

2018/11/17

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Table of contents

1 Teaser
2 Introduction
   - Efficiency
   - Correctness
3 Complexity analysis
   - Definitions
   - Analysis examples
4 Asymptotic notation
   - Notation
   - Properties of the asymptotic notation
   - Recurrences
5 Problem vs algorithm complexity
   - Binary sum
   - Binary product
6 Sorting algorithms
   - Selection Sort
   - Insertion Sort
   - MergeSort
   - Quicksort
Complexity classes

So far, we analyzed two algorithms and we have obtained two complexity functions:

- **Lookup**: \( T(n) = d \log n + e \)
- **Minimum**: \( T(n) = an + b \)
Complexity classes

So far, we analyzed two algorithms and we have obtained two complexity functions:

- **Lookup**: $T(n) = d \log n + e$ logarithmic
- **Minimum**: $T(n) = an + b$ linear
Complexity classes

So far, we analyzed two algorithms and we have obtained two complexity functions:

- Lookup: $T(n) = d \log n + e$ \hspace{1cm} \text{logarithmic}
- Minimum: $T(n) = an + b$ \hspace{1cm} \text{linear}

A third function comes from the \textit{naïf} algorithm to compute the minimum:

- Minimum: $T(n) = fn^2 + gn + h$ \hspace{1cm} \text{quadratic}
Complexity classes

So far, we analyzed two algorithms and we have obtained two complexity functions:

- **Lookup**: \( T(n) = d \log n + e \)  \[\text{logarithmic} \quad O(\log n)\]
- **Minimum**: \( T(n) = an + b \)  \[\text{linear} \quad O(n)\]

A third function comes from the *naïf* algorithm to compute the minimum:

- **Minimum**: \( T(n) = fn^2 + gn + h \)  \[\text{quadratic} \quad O(n^2)\]
### Complexity classes

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>( n = 10^1 )</th>
<th>( n = 10^2 )</th>
<th>( n = 10^3 )</th>
<th>( n = 10^4 )</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>3</td>
<td>10</td>
<td>31</td>
<td>100</td>
<td>sublinear</td>
</tr>
<tr>
<td>( n )</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>linear</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>30</td>
<td>664</td>
<td>9965</td>
<td>132877</td>
<td>log-linear</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>( 10^2 )</td>
<td>( 10^4 )</td>
<td>( 10^6 )</td>
<td>( 10^8 )</td>
<td>quadratic</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>( 10^3 )</td>
<td>( 10^6 )</td>
<td>( 10^9 )</td>
<td>( 10^{12} )</td>
<td>cubic</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>1024</td>
<td>( 10^{30} )</td>
<td>( 10^{300} )</td>
<td>( 10^{3000} )</td>
<td>exponential</td>
</tr>
</tbody>
</table>
**Definition – O notation**

Let $g(n)$ be a cost function; $O(g(n))$ is the set of all functions $f(n)$ such that:

\[ \exists c > 0, \exists m \geq 0 : f(n) \leq cg(n), \forall n \geq m \]

- How we read it: $f(n)$ is “big-Oh” of $g(n)$
- How we write it: $f(n) = O(g(n))$
- $g(n)$ is asymptotic upper bound for $f(n)$
- $f(n)$ grows at most as $g(n)$
Asymptotic notation

**O, Ω, Θ notations**

**Definition – Ω notation**

Let $g(n)$ be a cost function; $\Omega(g(n))$ is the set of all functions $f(n)$ such that:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n), \forall n \geq m$$

- How we read it: $f(n)$ is “big-omega” of $g(n)$
- How we write it: $f(n) = \Omega(g(n))$
- $g(n)$ is an asymptotic lower bound for $f(n)$
- $f(n)$ grows at least as $g(n)$
**O, Ω, Θ notations**

**Definition – Notation Θ**

Let $g(n)$ be a cost function; $\Theta(g(n))$ is the set of all functions $f(n)$ such that:

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq m$$

- How we read it: $f(n)$ is “theta” of $g(n)$
- How we write it: $f(n) = \Theta(g(n))$
- $f(n)$ grows as $g(n)$
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
Graphically

The graph illustrates the functions $c_2g(n)$, $f(n)$, $c_1g(n)$, and $g(n)$, showing how they grow as $n$ increases. The functions are represented by different line styles and colors, with $c_2g(n)$ shown with a red dashed line, $f(n)$ with a black solid line, $c_1g(n)$ with a blue dotted line, and $g(n)$ with a dashed line.

The graph indicates that $c_2g(n)$ grows faster than $f(n)$ and $c_1g(n)$, while $g(n)$ grows relatively slowly compared to the other functions. This visual representation helps in understanding the asymptotic behavior of these functions as $n$ becomes large.
True or false?

\[ f(n) = 10n^3 + 2n^2 + 7 \overset{?}{=} O(n^3) \]

We need to prove that \( \exists c > 0, \exists m \geq 0 : f(n) \leq cn^3, \forall n \geq m \)

\[ f(n) = 10n^3 + 2n^2 + 7 \]
\[ \leq 10n^3 + 2n^3 + 7 \quad \forall n \geq 1 \]
\[ \leq 10n^3 + 2n^3 + 7n^3 \quad \forall n \geq 1 \]
\[ = 19n^3 \]
\[ \overset{?}{\leq} cn^3 \]

which is true for each \( c \geq 19 \) and for each \( n \geq 1 \), thus \( m = 1 \).
Graphically
True or false?

\[ f(n) = 3n^2 + 7n \overset{?}{=} \Theta(n^2) \]

**Lower bound:** \( \exists c_1 > 0, \exists m_1 \geq 0 : f(n) \geq c_1 n^2, \forall n \geq m_1 \)

\[
\begin{align*}
f(n) &= 3n^2 + 7n \\
&\geq 3n^2 \\
&\overset{?}{\geq} c_1 n^2
\end{align*}
\]

Per \( n \geq 0 \) which is true for each \( c_1 \leq 3 \) and for each \( n \geq 0 \), thus \( m_1 = 0 \)
True or false?

\[ f(n) = 3n^2 + 7n \overset{?}{=} \Theta(n^2) \]

**Upper bound:** \( \exists c_2 > 0, \exists m_2 \geq 0 : f(n) \leq c_2 n^2, \forall n \geq m_2 \)

\[
\begin{align*}
f(n) &= 3n^2 + 7n \\
&\leq 3n^2 + 7n^2 \\
&= 10n^2 \\
? &\leq c_2 n^2
\end{align*}
\]

che è vera per ogni \( c_2 \geq 10 \) e per ogni \( n \geq 1 \), quindi \( m_2 = 1 \)
True or false?

\[ f(n) = 3n^2 + 7n = \Theta(n^2) \]

**\( \Theta \) notation:**

\[ \exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 n^2 \leq f(n) \leq c_2 n^2, \forall n \geq m \]

With this parameters

\[ c_1 = 3 \]
\[ c_2 = 10 \]
\[ m = \max\{m_1, m_2\} = \max\{0, 1\} = 1 \]
Graphically

\[ 10n^2 \]
\[ 3n^2 + 7n \]
\[ 3n^2 \]
True or false?

\[ n^2 \overset{?}{=} O(n) \]

We want to prove that \( \exists c > 0, \exists m > 0 : n^2 \leq cn, \forall n \geq m \)

- We get this: \( n^2 \leq cn \Leftrightarrow c \geq n \)
- This means that \( c \) should grow with \( n \), i.e. we cannot choose a constant \( c \) valid for all \( n \geq m \)
Graphically
True or false?

\[ n^2 \overset{?}{=} O(n^3) \]

We need to prove that \( \exists c > 0, \exists m > 0 : n^2 \leq cn^3, \forall n \geq m \)

- We get this: \( n^2 \leq cn^3 \iff c \geq \frac{1}{n} \)
- Given that \( 1/n \) is monotonically decreasing for \( n > 0 \), we can choose any value of \( m \) (e.g., \( m = 1 \)), and select a constant \( c \geq 1/m \), such as \( c = 1 \).
Properties

**Polynomial expressions**

\[ f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n + a_0, \quad a_k > 0 \Rightarrow f(n) = \Theta(n^k) \]

**Constant elimination**

\[ f(n) = O(g(n)) \iff af(n) = O(g(n)), \forall a > 0 \]
\[ f(n) = \Omega(g(n)) \iff af(n) = \Omega(g(n)), \forall a > 0 \]
Properties

Sums

\[ f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))) \]
\[ f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) = \Omega(\min(g_1(n), g_2(n))) \]

Relation with algorithm analysis

- If an algorithm is composed by two parts, one which is \( \Theta(n^2) \) and one which \( \Theta(n) \), the resulting complexity is \( \Theta(n^2 + n) = \Theta(n^2) \)
Properties

Products

\[ f_1(n) = O(g_1(n)), \quad f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)) \]
\[ f_1(n) = \Omega(g_1(n)), \quad f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n)) \]

Relation with algorithm analysis

- If algorithm \( A \) calls algorithm \( B \) \( n \) times, and the complexity of algorithm \( B \) is \( \Theta(n \log n) \), the resulting complexity is \( \Theta(n^2 \log n) \).
Properties

Transitivity

\[ f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n)) \]

Proof:

\[ f(n) = O(g(n)) \land g(n) = O(h(n)) \Rightarrow \]
\[ f(n) \leq c_1 g(n) \land g(n) \leq c_2 h(n) \Rightarrow \]
\[ f(n) \leq c_1 c_2 h(n) \Rightarrow \]
\[ f(n) = O(h(n)) \]
Function classification

Is it possible to create a total order between the main function classes.

For each $0 < r < s, 0 < h < k, 1 < a < b$:

$$O(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n) \subset$$
$$O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$
def maxsum1(A):
    maxSoFar = 0 # Maximum found so far
    for i in range(0, len(A)):
        for j in range(i, len(A)):
            tot = sum(A[i:j+1])
            maxSoFar = max(maxSoFar, tot)
    return maxSoFar

The complexity of this algorithm can be approximated as follows (we are counting the number of sums that are executed).

\[
T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)
\]

We want to prove that \( T(n) = \theta(n^3) \), i.e.

\[
\exists c_1, c_2 > 0, \exists m \geq 0 : c_1 n^3 \leq T(n) \leq c_2 n^3, \forall n \geq m
\]
Time complexity of Version 1 - $O(n^3)$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$$

$$\leq \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} n$$

$$\leq \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n^2$$

$$= n^3 \leq c_2 n^3$$

This inequality is true for $n \geq m = 0$ and $c_2 \geq 1$. 
Time complexity of Version 1 - $\Omega(n^3)$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$$

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

$$= \sum_{i=0}^{n/2} \sum_{j=i}^{i+n/2-1} n/2$$

$$= \sum_{i=0}^{n/2} n^2/4 \geq n^3/8 \geq c_1 n^3$$

This inequality is true for $n \geq m = 0$ and $c_1 \leq 8$. 
def maxsum2(A):
    maxSoFar = 0  # Maximum found so far
    for i in range(0, len(A)):
        tot = 0  # Accumulator
        for j in range(i, len(A)):
            tot = tot + A[j]
            maxSoFar = max(maxSoFar, tot)
    return maxSoFar

The complexity of this algorithm can be approximated as follows (we are counting the number of sums that are executed).

\[
T(n) = \sum_{i=0}^{n-1} n - i
\]
Time complexity of Version 2 - $\theta(n^2)$

We want to prove that $T(n) = \theta(n^2)$.

$$T(n) = \sum_{i=0}^{n-1} n - i$$

$$= \sum_{i=1}^{n} i$$

$$= \frac{n(n + 1)}{2} = \Theta(n^2)$$

This does not require further proofs.
**Time complexity of Version 4**

```python
def find_max_slice(A):
    maxSoFar = 0  # Maximum found so far
    maxHere = 0   # Maximum slice ending at the current pos

    for number in A:
        maxHere = max(maxHere + number, 0)
        maxSoFar = max(maxHere, maxSoFar)

    return maxSoFar
```

It’s easy to see that the time complexity of Version 4 is \(\theta(n)\).
Recurrence equations

Whenever the complexity of a recursive algorithm is computed, this is expressed through recurrence equation, i.e. a mathematical formula defined in a... recursive way!

Example

\[ T(n) = \begin{cases} 
2T(n/2) + n & n > 1 \\
\Theta(1) & n \leq 1 
\end{cases} \]
Recurrences

Closed formulas

Our goal is to obtain, whenever possible, a closed formula that represents the complexity class of our function.

Example

\[
T(n) = \begin{cases} 
2T(n/2) + n & n > 1 \\
\Theta(1) & n \leq 1 
\end{cases}
\]
Recurrences

Closed formulas

Our goal is to obtain, whenever possible, a closed formula that represents the complexity class of our function.

Example

\[ T(n) = \Theta(n \log n) \]
Master theorem

Theorem

Let $a$ and $b$ two integer constants such that $a \geq 1$ e $b \geq 2$, and let $c$, $\beta$ be two real constants such that $c > 0$ e $\beta \geq 0$. Let $T(n)$ be defined by the following recurrence:

$$T(n) = \begin{cases} 
\Theta(n^{\alpha}) & \alpha > \beta \\
\Theta(n^{\alpha \log n}) & \alpha = \beta \\
\Theta(n^\beta) & \alpha < \beta
\end{cases}$$

Given $\alpha = \log a / \log b = \log_b a$, then:

$$T(n) = \begin{cases} 
\Theta(n^{\alpha}) & \alpha > \beta \\
\Theta(n^{\alpha \log n}) & \alpha = \beta \\
\Theta(n^\beta) & \alpha < \beta
\end{cases}$$
Linear recurrences with constant order

**Theorem**

Let $a_1, a_2, \ldots, a_h$ be non-negative integer constants; let $c$ and $\beta$ real constant such that $c > 0$ and $\beta \geq 0$; let $T(n)$ be a recurrence defined as follows:

$$T(n) = \begin{cases} 
\sum_{1 \leq i \leq h} a_i T(n - i) + cn^\beta & n > m \\
\Theta(1) & n \leq m \leq h
\end{cases}$$

Given $a = \sum_{1 \leq i \leq h} a_i$, then:

1. $T(n) = \Theta(n^{\beta+1})$, if $a = 1$,
2. $T(n) = \Theta(a^n n^{\beta})$, if $a \geq 2$. 
## Some examples

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>a</th>
<th>b</th>
<th>( \log_b a )</th>
<th>Case</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = 4T(n/2) + n )</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>(1)</td>
<td>( T(n) = \Theta(n^2) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + n )</td>
<td>3</td>
<td>2</td>
<td>( \log_2 3 )</td>
<td>(1)</td>
<td>( T(n) = \Theta(n^{\log_2 3}) )</td>
</tr>
<tr>
<td>( T(n) = 2T(n/2) + n )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>(2)</td>
<td>( T(n) = \Theta(n \log n) )</td>
</tr>
<tr>
<td>( T(n) = T(n/2) + 1 )</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>(2)</td>
<td>( T(n) = \Theta(\log n) )</td>
</tr>
<tr>
<td>( T(n) = 9T(n/3) + n^3 )</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>(3)</td>
<td>( T(n) = \Theta(n^3) )</td>
</tr>
<tr>
<td>( T(n) = T(n-1) + n )</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>(3)</td>
<td>( T(n) = \Theta(n^2) )</td>
</tr>
<tr>
<td>( T(n) = T(n-1) + T(n-2) + 1 )</td>
<td>2</td>
<td>-</td>
<td>0</td>
<td>(3)</td>
<td>( T(n) = \Theta(2^n) )</td>
</tr>
</tbody>
</table>
Time complexity of Version 3

```python
from itertools import accumulate

def maxsum3_rec(A, i, j):
    if i == j:
        return max(0, A[i])
    m = (i + j) // 2
    maxL = maxsum3_rec(A, i, m)
    maxR = maxsum3_rec(A, m + 1, j)
    maxLL = max(accumulate(A[m:-len(A)+i-1:-1]))
    maxRR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxLL + maxRR)
```

For this, we need to define a recurrence relation:

\[
T(n) = 2T(n/2) + cn
\]

Using the theorem, we can see that \(\alpha = \log_2 2 = 1\) and \(\beta = 1\), thus \(T(n) = \Theta(n \log n)\).
**Time complexity of Version 3**

```python
from itertools import accumulate

def maxsum3_rec(A, i, j):
    if i == j:
        return max(0, A[i])
    m = (i + j) // 2
    maxL = maxsum3_rec(A, i, m)
    maxR = maxsum3_rec(A, m + 1, j)
    maxLL = max(accumulate(A[m:-len(A)+i-1:-1]))
    maxRR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxLL + maxRR)
```

For this, we need to define a recurrence relation:

\[
T(n) = 2T(n/2) + cn
\]

Using the theorem, we can see that \( \alpha = \log_2 2 = 1 \) and \( \beta = 1 \), thus

\[
T(n) = \Theta(n \log n).
\]
Scientific Programming

Algorithm analysis
Algorithm complexity vs Problem complexity

Alberto Montresor

Università di Trento

2018/11/17

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Table of contents

1 Teaser
2 Introduction
   ● Efficiency
   ● Correctness
3 Complexity analysis
   ● Definitions
   ● Analysis examples
4 Asymptotic notation
   ● Notation
   ● Properties of the asymptotic notation
   ● Recurrences
5 Problem vs algorithm complexity
   ● Binary sum
   ● Binary product
6 Sorting algorithms
   ● Selection Sort
   ● Insertion Sort
   ● MergeSort
   ● Quicksort
Introduction

**Goal:** reason about complexity of problems and algorithms

- In some cases, it is possible to improve what is considered "normal"
- In other cases, it is impossible to improve the existing solutions
- What is the relation between a problem and the algorithms that solve it?

**Back to basics!**

- Sums
- Products
Computing the sum of binary numbers

**Basic sum algorithm – sum()**

- each of the \( n \) bits have to be considered
- total cost is equal to \( cn \) \((c \equiv \) the cost required to sum three bits and generate the carry-over\)

**Question**

Is there a better method?
Lower bound to the complexity of a problem

**Notation Ω(f(n)) – Lower bound**

The computational complexity of a problem is equal to $\Omega(f(n))$ if all possible algorithms solving the problem have a complexity which is equal to $c \cdot f(n)$ or larger, where $c$ is an appropriate constant.

**Lower bound for the sum problem**

The problem of summing two binary numbers with $n$ bits is $\Omega(n)$. 
Computing the product of binary numbers

"Elementary school" algorithm – \texttt{prod()}

- bit-by-bit product
- total cost $cn^2$
**Arithmetic algorithms**

**Comparison of computational complexity**

- **Sum**: $T_{\text{sum}}(n) = O(n)$
- **Products**: $T_{\text{prod}}(n) = O(n^2)$

We could conclude that:

- The product problem is inherently more costly than the sum problem
- This confirms our intuition
Arithmetic algorithms

Comparing problems

To prove that the product problem is more costly than the sum problem, we must prove that there is no solution in linear time for the product.

We compared the algorithms, not the problems!

- We only know the "elementary school" algorithm for the sum is more efficient than the "elementary school" algorithm for the product.
- In 1960, during a conference, Kolmogorov claimed that the product has $\Omega(n^2)$ lower bound
- A week later, he was proved wrong!
Product of binary numbers - Divide-et-impera

**Binary number product**

\[
X = a \cdot 2^{n/2} + b \\
Y = c \cdot 2^{n/2} + d \\
XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd
\]

**Decimal number product:** \(9977 \times 2348\)

\[
X = a \cdot 10^{n/2} + b \\
Y = c \cdot 10^{n/2} + d \\
XY = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd \\
XY = 99 \times 23 \cdot 10^4 + (99 \times 48 + 77 \times 23) \cdot 10^2 + 77 \times 48
\]
Binary number multiplication through Divide-et-impera

```
pdi(X, Y, int n)

if n == 1 then
  return X[0] · Y[0]
else
  break X in two parts a; b and Y in two parts c; d
  return pdi(a, c, n/2) · 2^n +
    (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^{n/2} +
    pdi(b, d, n/2)
```

\[ T(n) = \begin{cases} 
  c_1 & n = 1 \\
  4T(n/2) + c_2 \cdot n & n > 1 
\end{cases} \]

Note: Multiplying by \(2^t\) \(\equiv\) shift of \(t\) position, in linear time
Recurrence analysis

Recurrence

\[ T(n) = \begin{cases} 
  c_1 & n = 1 \\
  4T(n/2) + c_2 \cdot n & n > 1 
\end{cases} \]

Applying the master theorem

\[ \alpha = \log_2 4 = 2 \]
\[ \beta = 1 \]
\[ T(n) = \Theta(n^2) \]
Computing the product of binary numbers

**Comparing the computational complexity**

- Product: $T_{prod}(n) = O(n^2)$
- Product: $T_{pdi}(n) = O(n^2)$

All this, for nothing?

**Question: Is it possible to improve this idea?**

Note that this recursive version calls itself 4 times
Multiplying complex numbers

(a + bi)(c + di) = [ac − bd] + [ad + bc]i

Input: $a, b, c, d$

Output: $a \times c − b \times d, a \times d + b \times c$

Questions

Let’s consider a computing model where scalar multiplications cost 1, scalar sums/subtractions cost 0.01

- How much does it cost to multiply two complex numbers?
- Can you do better than this?
Multiplying complex numbers

- The naïve algorithm associated to the definition cost 4.02

- Gauss solution (1805):
  \[
  \text{Input: } a, b, c, d, \quad \text{Output: } A1 = ac - bd, A2 = ad + bc
  \]
  \[
  m1 = a \times c \\
  m2 = b \times d \\
  A1 = m1 - m2 = ac - bd \\
  m3 = (a + b) \times (c + d) = ac + ad + bc + bd \\
  A2 = m3 - m1 - m2 = ad + bc
  \]

  In this case, the cost is 3.05.
Multiplying complex numbers

Open questions

- Can you do better than this?
- Or, is it possible to prove that you cannot be more efficient?

Remarks

- In this model, performing 3 multiplications instead of 4 saves around 25% of the total cost.
- Can you see how to apply this line of reasoning to the binary multiplication problem?
Karatsuba Algorithm (1962) (Inspired by Gauss)

\[ A_1 = a \times c \]
\[ A_3 = b \times d \]
\[ m = (a + b) \times (c + d) = ac + ad + bc + bd \]
\[ A_2 = m - A_1 - A_3 = ad + bc \]

```java
boolean [] KARATSUBA(boolean[] X, boolean[] Y, int n)
if n == 1 then
    return X[0] \cdot Y[0]
else
    break X in a; b e Y in c; d
    A1 = KARATSUBA(a, c, n/2)
    A3 = KARATSUBA(b, d, n/2)
    m = KARATSUBA(a + b, c + d, n/2)
    A2 = m - A1 - A3
    return A1 \cdot 2^n + A2 \cdot 2^{n/2} + A3
```
Recurrence analysis

Recurrence

\[
T(n) = \begin{cases} 
    c_1 & n = 1 \\
    3T(n/2) + c_2 \cdot n & n > 1 
\end{cases}
\]

Applying the master theorem

\[
\alpha = \log_2 3 \approx 1.58 \\
\beta = 1 \\
T(n) = \Theta(n^{1.58})
\]
Computing the product of binary numbers

Comparing the computational complexity

- Product: $T_{prod}(n) = O(n^2)$
  - Example: $T_{prod}(10^6) = 10^{12}$
- Product: $T_{kara}(n) = O(n^{1.58})$
  - Example: $T_{kara}(10^6) = 3 \cdot 10^9$

Conclusions

- The "naive" algorithm is not always the best...
- ...there is often space for improvement...
- ...unless you can prove the opposite!
Further extensions

- **Toom-Cook (1963)**
  - Also called Toom3, its complexity is $O(n^{\log 5/\log 3}) \approx O(n^{1.465})$
  - Karatsuba $\equiv$ Toom2
  - "Elementary school" product $\equiv$ Toom1

- **Schönhage–Strassen (1971)**
  - Complexity: $O(n \cdot \log n \cdot \log \log n)$
  - Based on Fast Fourier Transforms

- **Martin Fürrer (2007)**
  - Complexity: $O(n \cdot \log n \cdot 2^{O(\log^* n)})$
  - Lower bound: $\Omega(n \log n)$ (conjecture)

### Iterated logarithm

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log^* n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-\infty, 1]$</td>
<td>0</td>
</tr>
<tr>
<td>$(1, 2]$</td>
<td>1</td>
</tr>
<tr>
<td>$(2, 4]$</td>
<td>2</td>
</tr>
<tr>
<td>$(4, 16]$</td>
<td>3</td>
</tr>
<tr>
<td>$(16, 2^{16}]$</td>
<td>4</td>
</tr>
<tr>
<td>$(2^{16}, 2^{65536}]$</td>
<td>5</td>
</tr>
</tbody>
</table>
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? $k = \lceil \log n \rceil$
- How many multiplications, in terms of $k$?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? $k = \lceil \log n \rceil$
- How many multiplications, in terms of $k$? $n = 2^k$
- How many bits are necessary, to represent the output?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? $k = \lceil \log n \rceil$
- How many multiplications, in terms of $k$? $n = 2^k$
- How many bits are necessary, to represent the output? $\lceil \log n! \rceil = \Theta(n \log n)$
- In terms of $k$?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? $k = \lceil \log n \rceil$
- How many multiplications, in terms of $k$? $n = 2^k$
- How many bits are necessary, to represent the output?
  $\lceil \log n! \rceil = \Theta(n \log n)$
- In terms of $k$? $\Theta(2^k \cdot k)$
- How much does it cost to multiply two numbers of $2^k \cdot k$ bits?
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res

- What is the size of the input? \( k = \lceil \log n \rceil \)
- How many multiplications, in terms of \( k \)? \( n = 2^k \)
- How many bits are necessary, to represent the output? \( \lceil \log n! \rceil = \Theta(n \log n) \)
- In terms of \( k \)? \( \Theta(2^k \cdot k) \)
- How much does it cost to multiply two numbers of \( 2^k \cdot k \) bits? \( \Theta(2^{2k} \cdot k^2) \)
- Complexity in terms of \( k \)?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? \( k = \lceil \log n \rceil \)
- How many multiplications, in terms of \( k \)? \( n = 2^k \)
- How many bits are necessary, to represent the output?
  \[ \lceil \log n! \rceil = \Theta(n \log n) \]
- In terms of \( k \)? \( \Theta(2^k \cdot k) \)
- How much does it cost to multiply two numbers of \( 2^k \cdot k \) bits?
  \( \Theta(2^{2k} \cdot k^2) \)
- Complexity in terms of \( k \)? \( \Theta(2^{3k} \cdot k^2) \)
- Complexity in terms of \( n \)?
Logarithmic cost model

```python
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res
```

- What is the size of the input? $k = \lceil \log n \rceil$
- How many multiplications, in terms of $k$? $n = 2^k$
- How many bits are necessary, to represent the output?
  \[
  \lceil \log n! \rceil = \Theta(n \log n)
  \]
- In terms of $k$? $\Theta(2^k \cdot k)$
- How much does it cost to multiply two numbers of $2^k \cdot k$ bits?
  $\Theta(2^{2k} \cdot k^2)$
- Complexity in terms of $k$? $\Theta(2^{3k} \cdot k^2)$
- Complexity in terms of $n$? $\Theta(n^3 (\log n)^2)$
# Table of contents

1. Teaser
2. Introduction
   - Efficiency
   - Correctness
3. Complexity analysis
   - Definitions
   - Analysis examples
4. Asymptotic notation
   - Notation
   - Properties of the asymptotic notation
   - Recurrences
5. Problem vs algorithm complexity
   - Binary sum
   - Binary product
6. Sorting algorithms
   - Selection Sort
   - Insertion Sort
   - MergeSort
   - Quicksort
Introduction

Goal: evaluate the algorithms based on the type of input

- In some cases, algorithms behave differently depending on the characteristics of the input
- Knowing these characteristics in advance enable to choose the best algorithm for that particular scenario
- The sorting problem is a good school where to show such concepts

Sorting algorithms

- Selection Sort
- Insertion Sort
- MergeSort
- Quick Sort
Different kind of analysis

**Worst case analysis**

- The most important
- The execution time in the worst case is an upper bound to the execution time for all inputs
- For some algorithms, the worst case is very common: Example: searching data not present in a database

**Average case analysis**

- The most difficult to compute; requires notions of probability and a good definition of what is the "average case"

**Best case analysis**

- If we know particular conditions on the input, it could be worthwhile to compute;
Sorting

**Sorting problem**

- **Input**: A sequence $A = a_1, a_2, \ldots, a_n$ containing $n$ values
- **Output**: A sequence $B = b_1, b_2, \ldots, b_n$ that is a permutation of $A$ such that $b_1 \leq b_2 \leq \ldots \leq b_n$
Sorting problem

- **Input**: A sequence $A = a_1, a_2, \ldots, a_n$ containing $n$ values
- **Output**: A sequence $B = b_1, b_2, \ldots, b_n$ that is a permutation of $A$ such that $b_1 \leq b_2 \leq \ldots \leq b_n$

"Naive" approach:

- Search for the minimum, put it in the correct position, reduce the problem to the $n - 1$ elements that are left.
Selection Sort

This function returns the index of the smallest element included in $A[i:]$

```python
def argmin(A, i):
    minpos = i
    for j in range(i+1, len(A)):
            minpos = j  # New partial minimum position
    return minpos
```

This function repeatedly search the minimum in $A[i:]$, and swaps it with the element in $A[i]$

```python
def selectionSort(A):
    for i in range(len(A)-1):
        minpos = argmin(A, i)
```

Alberto Montresor (UniTN)
## Selection Sort – Example

### Algorithm:

1. **Initialization:**
   - Set a variable `j` to 0.

2. **Iteration:**
   - For each `i` from 0 to 6:
     - Find the minimum element in the subarray from `j` to `i`.
     - Swap it with the element at position `j`.
     - Increment `j`.

### Example:

Given array: 7, 4, 2, 1, 8, 3, 5

1. **Iteration 0:**
   - `j = 0`:
     - Find the minimum element in the subarray 7, 4, 2, 1, 8, 3, 5.
     - Swap with 7:
       - Array becomes: 1, 4, 2, 1, 8, 3, 5

2. **Iteration 1:**
   - `j = 1`:
     - Find the minimum element in the subarray 1, 4, 2, 7, 8, 3, 5.
     - Swap with 7:
       - Array becomes: 1, 4, 2, 7, 8, 3, 5

3. **Iteration 2:**
   - `j = 2`:
     - Find the minimum element in the subarray 1, 2, 4, 7, 8, 3, 5.
     - Swap with 4:
       - Array becomes: 1, 2, 4, 7, 8, 3, 5

4. **Iteration 3:**
   - `j = 3`:
     - Find the minimum element in the subarray 1, 2, 3, 7, 8, 4, 5.
     - Swap with 3:
       - Array becomes: 1, 2, 3, 7, 8, 4, 5

5. **Iteration 4:**
   - `j = 4`:
     - Find the minimum element in the subarray 1, 2, 3, 4, 8, 7, 5.
     - Swap with 4:
       - Array becomes: 1, 2, 3, 4, 8, 7, 5

6. **Iteration 5:**
   - `j = 5`:
     - Find the minimum element in the subarray 1, 2, 3, 4, 5, 7, 8.
     - Swap with 5:
       - Array becomes: 1, 2, 3, 4, 5, 7, 8

7. **Iteration 6:**
   - `j = 6`:
     - Find the minimum element in the subarray 1, 2, 3, 4, 5, 7, 8.
     - No further swaps needed.

Final sorted array: 1, 2, 3, 4, 5, 7, 8
Selection Sort

- Number of comparisons of $\text{argmin}(A, i)$?
Selection Sort

- Number of comparisons of $\arg\min (A, i)$?

$$\text{len}(A) - 1 - i = n - 1 - i$$

- Number of comparison of $\text{selectionSort}(A)$?
Selection Sort

- Number of comparisons of $\text{argmin}(A, i)$?

$$\text{len}(A) - 1 - i = n - 1 - i$$

- Number of comparison of $\text{selectionSort}(A)$?

$$\sum_{i=0}^{n-2} (n - 1 - i)$$

$$= (n - 1) + (n - 2) + \ldots + 2 + 1$$

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

- The complexity (in the worst, average, optimal) case is:
Selection Sort

- Number of comparisons of argmin(A, i)?

\[ \text{len}(A) - 1 - i = n - 1 - i \]

- Number of comparison of selectionSort(A)?

\[
\sum_{i=0}^{n-2} (n - 1 - i) \\
= (n - 1) + (n - 2) + \ldots + 2 + 1 \\
= \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2} = n^2 - n/2
\]

- The complexity (in the worst, average, optimal) case is: $\Theta(n^2)$
Insertion Sort

- Efficient algorithm to sort small sets of elements
- It can be compared to the sorting procedure that a human performs on a hand of cards (e.g. scala quaranta)

def insertionSort(A):
    for i in range(1, len(A)):
        temp = A[i]
        j = i
        while j>0 and A[j-1]>temp:
            j = j-1
        A[j] = temp
## Insertion Sort

Sorting algorithms

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### Example Iterations

- **i = 1, j = 1**
  - Current row: 7
  - Current column: 4
  - temp = 4

- **i = 1, j = 0**
  - Current row: 4
  - Current column: 7
  - temp = 4

- **i = 2, j = 2**
  - Current row: 4
  - Current column: 7
  - temp = 2

- **i = 1, j = 1**
  - Current row: 4
  - Current column: 4
  - temp = 2

- **i = 2, j = 0**
  - Current row: 2
  - Current column: 4
  - temp = 2
## Insertion Sort

### Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>i</strong> = 3, <strong>j</strong> = 3</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>i</strong> = 3, <strong>j</strong> = 2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>i</strong> = 3, <strong>j</strong> = 1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>i</strong> = 3, <strong>j</strong> = 0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>i</strong> = 4, <strong>j</strong> = 4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>i</strong> = 5, <strong>j</strong> = 5</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
### Insertion Sort

The Insertion Sort algorithm sorts an array by iteratively inserting elements into their correct positions. Here is a graphical representation showing the sorting process for an array of numbers:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 5, j = 4$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>$i = 5, j = 3$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>$i = 5, j = 2$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>$i = 6, j = 6$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$i = 6, j = 5$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$i = 6, j = 4$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

In each step, the value at $i$ is inserted into the correct position within the sorted portion of the array from $0$ to $i-1$. The value at $j$ is moved one position to the right until the correct position is found. This process continues until the entire array is sorted.
Computational complexity

In this algorithm

- The cost does not depend only on the size of the input
- but also on how the values are sorted

Questions

- What is the cost in the case the sequence is already ordered?
- What is the cost in the case the sequence is ordered in the opposite direction?
- What is the cost on average (informally)?
MergeSort

**Divide-et-impera**

*MergeSort* is based on the *divide-et-impera* technique discussed before

- **Divide**: Break (virtually) the sequence of $n$ elements in two sub-sequences
- **Impera**: Call *MergeSort* recursively on both sub-sequences
- **Combine**: Join (*merge*) the two sorted sub-sequences

**Idea**

We exploit the fact that the two sub-sequences are already sorted to sort the total list in a faster way
Merge

merge(A, first, last, mid)

Input:
- A is a list of \( n \) integers
- first, last, mid are such that \( 1 \leq \text{first} \leq \text{mid} < \text{last} \leq n \)
- The subsequences \( A[\text{first}:\text{mid}+1] \) and \( A[\text{mid}+1:\text{last}+1] \) are already ordered

Output:
- The two subsequences are merged in a single sorted sequence \( A[\text{first}:\text{last}+1] \), using another list B.
How merge() works

- **A**: 1 3 7 8 2 4 6 9
- **B**: 
- **i, j, k**

1. **A**: 3 7 8 2 4 6 9
2. **B**: 1
3. **i, j, k**
4. **A**: 7 8 4 6 9
5. **B**: 1 2
6. **i, j, k**
7. **A**: 7 8 6 9
8. **B**: 1 2 3
9. **k**
10. **A**: 8 9
11. **B**: 1 2 3 4
12. **k**
How `merge()` works

Beginning at the top of arrays `A` and `B`, `merge()` compares elements at positions `i` and `j` until the end of one of the arrays is reached. The smaller element is placed at position `k` and the process repeats with the next elements.

1. **Initial State**
   - `A`: 7 8 9
   - `B`: 1 2 3 4 6
   - `k`: 1

2. **First Comparison**
   - `i`: 7
   - `j`: 1
   - `k`: 1
   - Place 1 at `k`: 7 8 9 1

3. **Second Comparison**
   - `i`: 8
   - `j`: 2
   - `k`: 2
   - Place 2 at `k`: 7 8 2 9

4. **Third Comparison**
   - `i`: 8
   - `j`: 3
   - `k`: 3
   - Place 3 at `k`: 7 8 3 9

5. **Fourth Comparison**
   - `i`: 8
   - `j`: 4
   - `k`: 4
   - Place 4 at `k`: 7 8 3 4

6. **Fifth Comparison**
   - `i`: 9
   - `j`: 6
   - `k`: 5
   - Place 6 at `k`: 7 8 3 4 6

7. **Sixth Comparison**
   - `i`: 9
   - `j`: 7
   - `k`: 6
   - Place 7 at `k`: 7 8 3 4 6 7

8. **Seventh Comparison**
   - `i`: 9
   - `j`: 8
   - `k`: 7
   - Place 8 at `k`: 7 8 3 4 6 7 8

9. **Eighth Comparison**
   - `i`: 9
   - `j`: 9
   - `k`: 8
   - Place 9 at `k`: 7 8 3 4 6 7 8 9

The process continues until all elements are placed in the correct order.
merge()\

```python
def merge(A, first, last, mid):
    i = first
    j = mid+1
    B = []
    while i <= mid and j<=last:
        if A[i] <= A[j]:
            B.append(A[i])
            i = i+1
        else:
            B.append(A[j])
            j = j+1
    while i <= mid:
        B.append(A[i])
        i = i+1
    for k in range(len(B)):
        A[first+k] = B[k]
```

- Variables i and j are used to scan the values of the two sublists.
- The first `while` loop compares the elements when both `A[i:mid+1]` and `A[j:last+1]` are not empty, add the smaller to `B`, and increase either i or j.
- The second `while` loop moves the remaining elements in `A[i:mid+1]` to `B`.
- The elements in `B` are smaller than those in `A[j:last+1]`.
- They are moved back into `A[start:start+len(B)]`.
merge() – Variant with slicing

```python
def merge(A, first, last, mid):
    i = first
    j = mid+1
    B = []
    while i <= mid and j<=last:
        if A[i] <= A[j]:
            B.append(A[i])
            i = i+1
        else:
            B.append(A[j])
            j = j+1
    B.extend(A[i:mid+1])
    A[first:first+len(B)] = B
```

- Moving the remaining elements in `A[i:mid+1]` to `B` can be done through `extend()`.
- Substituting the elements in `A[first:first+len(B)]` with those in `B` can be done through slicing.
Computational cost

Question

What is the computational cost of Merge()?
Computational cost

**Question**

What is the computational cost of Merge()? ⇒ $O(n)$
MergeSort

- Base case: sequences of length \( \leq 1 \) are already ordered
- Compute the median position \( \text{mid} \)
- Call recursively itself on \( A[\text{first}:\text{mid}+1] \) e \( A[\text{mid}+1:\text{last}] \)
- Call \text{merge()} \) to merge together the results

```python
def mergeSortRec(A, first, last):
    if first < last:
        mid = (first+last)//2
        mergeSortRec(A, first, mid)
        mergeSortRec(A, mid+1, last)
        merge(A, first, last, mid)

def mergeSort(A):
    mergeSortRec(A, 0, len(A)-1)
```
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

33, 21

33, 21

33

21

7

48

28, 13, 65, 17

28, 13

28

13

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

33, 21

33

21
mergeSort(): Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

33, 21

7, 48

28, 13

65, 17
mergeSort(): Execution
**mergeSort(): Execution**

- **33, 21, 7, 48, 28, 13, 65, 17**
- **33, 21, 7, 48**
- **33, 21**
- **33, 21**
- **33**
- **21**
- **7, 48**
- **28, 13**
- **65, 17**
- **28, 13, 65, 17**
- **65, 17**
- **17**
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

33, 21

33, 21

33

21

7, 48

28, 13

65, 17

28, 13

65

17

13

65

17
**mergeSort(): Execution**

```
33, 21, 7, 48, 28, 13, 65, 17
```

```
33, 21, 7, 48
```

```
33, 21
```

```
33
```

```
21
```

```
7, 48
```

```
28, 13
```

```
65, 17
```

```
7
```

```
48
```

```
28
```

```
13
```

```
65
```

```
17
```
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

33, 21

21, 33

33, 21

21, 33

33, 21

21, 33

33

21

7, 48

28, 13, 65, 17

28, 13

65, 17

28

13

65

17

7

48

28

13

65

17
**mergeSort(): Execution**

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

21, 33

7, 48

28, 13

65, 17

7

48

28

13

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

21, 33  7, 48

28, 13, 65, 17

28, 13

65, 17

7, 48

28, 13

65, 17
mergeSort(): Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

21, 33

7, 48

28, 13

65, 17

28

13

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

21, 33
7, 48

28, 13, 65, 17

28, 13
65, 17

7, 48
21, 33
33, 21, 7, 48
33, 21, 7, 48, 28, 13, 65, 17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

28, 13, 65, 17

65, 17

28

13

65

17
mergeSort(): Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

21, 33
7, 48

28, 13, 65, 17

28, 13
65, 17

28
13
65
17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

21, 33

7, 48

28, 13, 65, 17

28, 13

65, 17

28

13

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

21, 33, 7, 48

28, 13, 65, 17

28, 13

65, 17
**mergeSort(): Execution**

```
33, 21, 7, 48, 28, 13, 65, 17
```

```
33, 21, 7, 48
28, 13, 65, 17

7, 21
21, 33
7, 48

28, 13
65, 17
```

```
28
13
65
17
```
mergeSort(): Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

7, 21, 33, 48

21, 33

7, 48
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

28, 13

65, 17

65, 17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

65, 17

28

13

65

17
mergeSort(): Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48
28, 13, 65, 17

28, 13

28
13
65
17

Alberto Montresor (UniTN)
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

28, 13

65, 17

65, 17

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

28

13

65, 17

65

17
**mergeSort(): Execution**

```
33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

13

28, 13

65, 17

65

17
```
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

28, 13

13, 28

28, 13

65, 17

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

65, 17

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

65, 17

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48
28, 13, 65, 17

13, 28
65, 17

65
17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

65, 17

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48
28, 13, 65, 17
13, 28
65, 17
65
17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

65, 17

17

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

65, 17

17, 65

65

17
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48
28, 13, 65, 17

13, 28
17, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

17, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 28

17, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 17

13, 28

17, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 17, 28

13, 28

17, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

28, 13, 65, 17

13, 17, 28, 65

13, 28

17, 65
mergeSort() Execution
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28

7, 21, 33, 48

13, 17, 28, 65
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28, 33

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

```
33, 21, 7, 48, 28, 13, 65, 17
```

```
7, 13, 17, 21, 28, 33, 48
```

```
7, 21, 33, 48
```

```
13, 17, 28, 65
```
mergeSort(): Execution

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28, 33, 48, 65

7, 21, 33, 48
13, 17, 28, 65
mergeSort(): Execution

7, 13, 17, 21, 28, 33, 48, 65
mergeSort(): Execution

7, 13, 17, 21, 28, 33, 48, 65

7, 21, 33, 48

21, 33
33 21

7, 48
7
48

13, 17, 28, 65

13, 28
28
13

17, 65
65
17
Analysis of `mergeSort()`

Simplifying assumptions:

- \( n = 2^k \), i.e. the number of subdivisions is equal to \( k = \log n \);
- All the subsequences have size that are exact powers of 2

**Computational cost:**

\[
T(n) = \begin{cases} 
  c & n = 1 \\
  2T(n/2) + dn & n > 1 
\end{cases}
\]
Computational cost of MergeSort

**Question**

What is the computational cost of `mergeSort()`?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>......</th>
<th>log n - 1</th>
<th>log n</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n/2</td>
<td>n/4</td>
<td>n/4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>n/2</td>
<td>n/4</td>
<td>n/4</td>
<td>n/4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>n/4</td>
<td>n/4</td>
<td>n/4</td>
<td>n/4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ T(n) = O(n \log n) \]
Computational cost of MergeSort

Question

What is the computational cost of `mergeSort()`? \(\Rightarrow O(n \log n)\)
## Computational cost of MergeSort

### Question

What is the computational cost of `mergeSort()`? \( \Rightarrow O(n \log n) \)

<table>
<thead>
<tr>
<th>( \log n ) - 1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>1/2</th>
<th>1/4</th>
<th>1/4</th>
<th>1/4</th>
<th>1/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( n/2 )</td>
<td>( n/4 )</td>
<td>( n/4 )</td>
<td>( n/4 )</td>
<td>( n/4 )</td>
</tr>
</tbody>
</table>
Quicksort

Another sorting algorithm

- Based on divide-et-impera
- Average case: $O(n \log n)$, worst case $O(n^2)$

Average case vs worst case

- The multiplicative factor of QuickSort is better than MergeSort
- It is possible to use “heuristic” techniques to avoid the worst case
- Thus, it is often preferred to other algorithms

Quicksort

Input

- Sequence $A$ containing $n$ values
- Indexes $\text{first}$, $\text{last}$ such that $0 \leq \text{first} \leq \text{last} < n$

Divide

- Select a value $p \in A[\text{first} : \text{last} + 1]$ called pivot (perno)
- Move all the elements in slice $A[\text{first} : \text{last} + 1]$ in a way that
  $\forall i \in A[\text{first} : j] : A[i] \leq p$
  $\forall i \in A[j + 1 : \text{last}] : A[i] \geq p$
- Index $j$ is computed in a way that satisfies such condition
- The pivot is moved in position $A[j]$
QuickSort

**Impera**

Sort the slices \(A[first:j]\) and \(A[j + 1:last+1]\) by recursively calling QuickSort

**Combina**

Do nothing:
- the left subslice \(A[first:j]\)
- \(A[j]\),
- the right subslice \(A[j+1:last+1]\)

are already ordered
def pivot(a, first, last):
    p = A[first]
    j = first
    for i in range(first+1, last+1):
        if A[i] < p:
            j = j+1
    A[j] = p
    return j
Quicksort – Procedura principale

```python
def quickSortRec(A, first, last):
    if first < last:
        j = pivot(A, first, last)
        quickSortRec(A, first, j-1)
        quickSortRec(A, j+1, last)

def quickSort(A):
    quickSortRec(A, 0, len(A)-1)
```

Alberto Montresor (UniTN)
SP - Algorithm analysis
perno()
**perno()**

```
\begin{array}{cccccccccccc}
\hline
\end{array}
```

\[ A[i] \geq p \]

```
\begin{array}{cccccccccccc}
\hline
\end{array}
```

\[ A[i] \geq p \]

```
\begin{array}{cccccccccccc}
\hline
\end{array}
```

\[ A[i] \geq p \]

```
\begin{array}{cccccccccccc}
\hline
\end{array}
```

\[ A[i] < p: \quad j \leftarrow j+1, \quad A[i] \leftarrow A[j] \]

```
\begin{array}{cccccccccccc}
\hline
\end{array}
```

\[ A[start] \leftarrow A[j]; \quad A[j] \leftarrow p \]
Recursive calls

20 14 28 34 15 27 12 30 21 25 13

13 14 15 12 20 27 29 30 21 25 28

12 13 15 14

12 14 15

14 21 25

21 25

14 21 28 30

30
Recursive calls

20 14 28 34 15 27 12 30 21 25 13

13 14 15 12

27 29 30 21 25 28

12 15 14

25 21

29 30 28

14

21

28 30

12 13 14 15 20 21 25 27 28 29 30
Quicksort: Computational complexity

Cost of pivot()

- $\Theta(n)$

Cost of Quicksort: depends on the partitioning

- Worst partitioning
  - Given a list of size $n$, the list is subdivided in two sublist of size 0 and $n-1$
  - $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
  - Question: When do you get the worst case?

- Best partitioning
  - Given a list of size $n$, the list is subdivided in two sublist of size $n/2$
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
Quicksort: Computational complexity

Average case

- The cost depends on the order of elements, not on their values
- We need to consider all possible permutations
- Complex probabilistic analysis

Average case: an intuition

- Some partitionings will be very good
- Some partitionings will be very bad
- Good partitioning greatly reduce the size of subproblems
- We quickly reach the base case anyway
Summary - Sorting algorithms

- **SelectionSort** - $\Theta(n^2)$
- **InsertionSort** - $\Omega(n)$, $O(n^2)$
- **ShellSort** - $\Omega(n)$, $O(n^{3/2})$
- **MergeSort** - $\Theta(n \log n)$
- **HeapSort** - $\Theta(n \log n)$
- **QuickSort** - $\Omega(n \log n)$, $O(n^2)$

Summary - Sorting algorithms

- All these algorithms are based on comparisons
  - Decisions about sorting are based on comparisons between two values ($<$, $=$, $>$)
- **Best algorithms**: $O(n \log n)$
  - InsertionSort and ShellSort are faster only in special cases
Sorting problem – Lower bound

It is possible to show that any sorting algorithms based on comparisons is $\Omega(n \log n)$. 
Counting Sort

Assumption

- The numbers to be sorted are included in a range $[0 \ldots k - 1]$

How it works

- Build a list $B$ with $k$ entries, where $B[i]$ contains the number of times that $i$ is contained in $A$.
- Put back together the elements in $A$, using the counters in $B$

Possible improvements:

- The interval could not be limited to $[0 \ldots k - 1]$; any known interval $[i, j]$ can work. In such case, you must subtract $i$ from each number.
Counting Sort

def countingSort(A, k):
    B = [0]*k
    for a in A:
        B[a] = B[a] + 1
    j = 0
    for i in range(k):
        while B[i] > 0:
            A[j] = i
            B[i] = B[i] - 1
            j = j + 1
Counting Sort

Complexity of Counting Sort

- $O(n + k)$
- If $k$ is $O(n)$, then the complexity of Counting Sort is $O(n)$

Counting Sort and lower bounds for sorting

- Counting Sort is not based on comparisons
- If $k \in \Omega(n^3)$, this algorithm is worse than any other algorithm seen so far
Pigeonhole Sort

Pigeonhole

- What happens when numbers are not integer, but tuples associated with a key to be sorted?
- We cannot use counters..
- But we can use lists instead!