L3 Algorithms

Course Organization

28 September 2023

Teachers

Pierre Aboulker

Maître de conférences ENS, Talgo team

Paul Jeanmaire

ATER ENS, Parkas team

Tatiana Starikovskaya

Maître de conférences ENS, Talgo team

Course website

Moodle: https://moodle.psl.eu/course/view.php?id=19050

- Message your teachers, arrange a meeting
- Forum and important announcements
- Preliminary programme and course materials
- Homework uploads

Organization

- **Lectures — room E. Noether (Thursday, 8:30-10:30)**
  - 28 Sept., and 5, 12, 19, 26 Oct., and 9 Nov. — T. Starikovskaya
  - 16, 23 Nov., and 7, 21 Dec., and 11, 18 Jan. — P. Aboulker
- **Exercise sessions (room E. Noether):**
  - either 10:45-12:15 Thursday — P. Aboulker (in English, high speed)
  - or 10:45-12:15 Friday — P. Jeanmaire (in French, moderate speed)
- **4 Homeworks**
  - 30% of the final grade
  - theory questions + coding
  - Tentative deadlines: 18 October, 15 November, 20 December, 17 January
- **Final exam**
  - 70% of the final grade
  - theory questions, might include questions both from lectures and exercise sessions (see Moodle for previous exams)
  - January 25, 2024, 9:00-12:00, room E. Noether

Lectures: Textbooks

- **"Introduction to Algorithms"** by Cormen, Leiserson, Rivest, and Stein: available in the library in English and in French
- **"Algorithms on strings, trees, and sequences"** by Gusfield: available in the library in English
- **"Approximation Algorithms"** by Vazirani: available in the library in French
- **"Parametrized Algorithms"** by Cygan, Fomin, Kowalik, Lokshtanov, Marx, Pilipczuk, Pilipczuk, Saurabh.

Exercise sessions

- Theory only, we discuss the main ideas, but no “clean” solutions provided
- For each session we have a big set of questions, we choose the questions to discuss based on the level of group and the time available
- Post your solutions to the forum: no strict format, as many questions as you’d like, both the questions we discussed and not, we’ll provide you feedback
- Essential for passing the exam successfully!

Homework guidelines

Available on Moodle:
https://moodle.psl.eu/mod/page/view.php?id=553050&inpopup=1

- Language: English or French. No penalties for language errors, but your work must be understandable. Do not use both languages at the same time.
- We'll give you French-English translations of the notions we discuss in lectures. You can also use Wikipedia to find a translation of a term (example).
- Your homework should be typed either in the editor provided by Moodle (you can insert LaTeX formulas using buttons or using \(, \) as delimiters) or using any other editor of your choice and joined as a pdf.
- No hand-written scans will be accepted!
- Your homework must be clearly written with complete sentences and well-organised logic, and should definitely not be your first draft.

Homework guidelines

Available on Moodle: https://bit.ly/3hlfmh6

If you are asked to show a (theoretical) algorithm:

- Explain the algorithm (you must give a plain-text description, and you can also give pseudocode).
- Prove that it returns a correct answer.
- Analyse its space and time complexities, unless said otherwise.

Homework guidelines

Available on Moodle: https://bit.ly/3hlfmh6

Programming exercises:

- Language: Python
- Installation guides: https://bit.ly/3k2vaqJ. Python is also pre-installed on machines in INFO3 & INFO4.
- Python tutorials: https://bit.ly/2VFDSC0. This tutorial https://bit.ly/2YTj2An covers most of the basics you might need for the course.
- To get full points, your code must pass the tests we provide. We reserve the right to add further tests. Think of possible corner cases!
- Your solution must follow questions' guidelines.

Homework guidelines

Available on Moodle: https://bit.ly/3hlfmh6

- You can submit your work as many times as you wish. Only the last submission will be evaluated.
- The deadlines are strict. Late homework is not accepted. (Please contact the teachers for special circumstances.)
- We will provide homework solutions.
- If you have any questions about grading, you can ask them via a personal message.

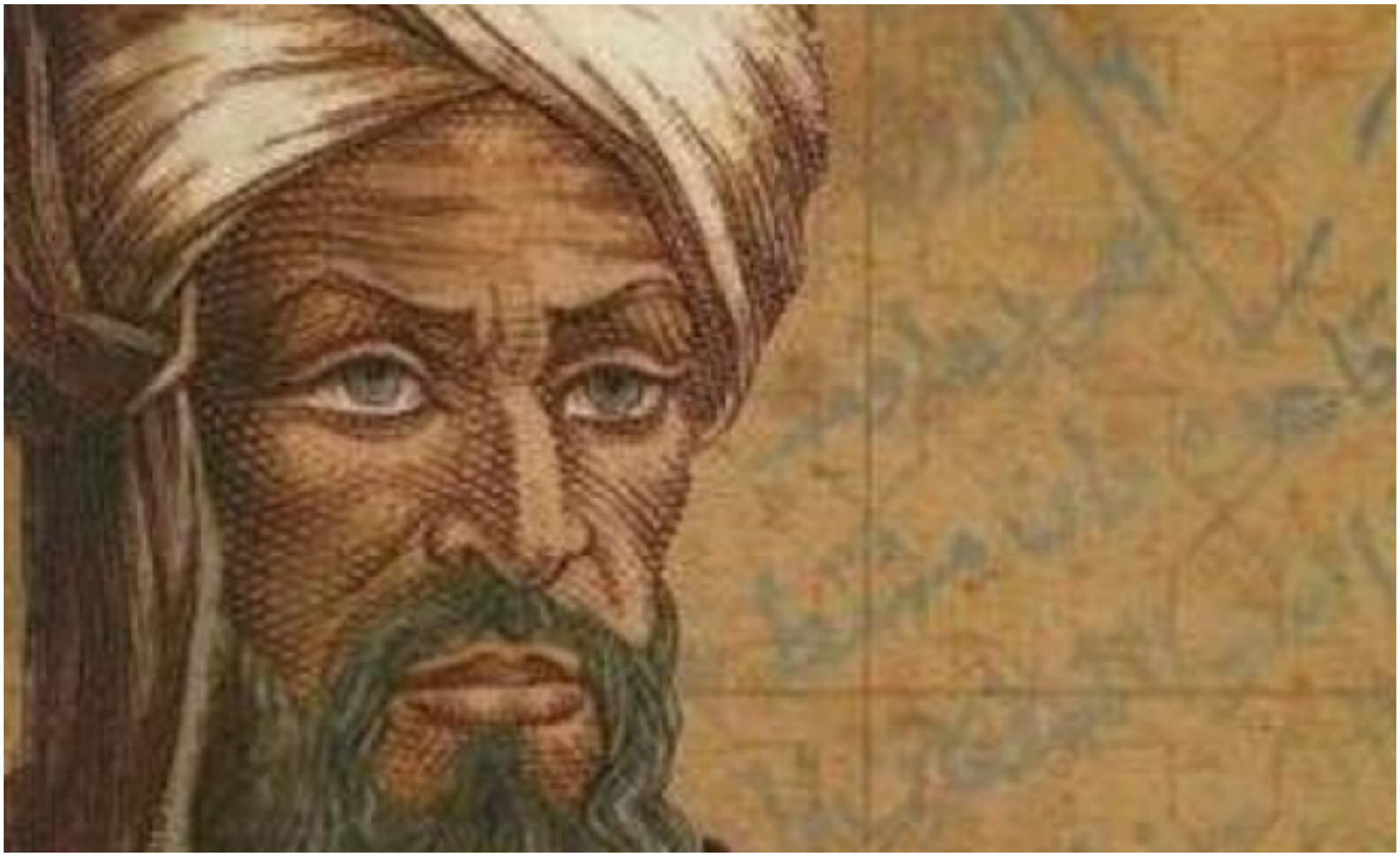
Lecture 1

Introduction

Today's plan

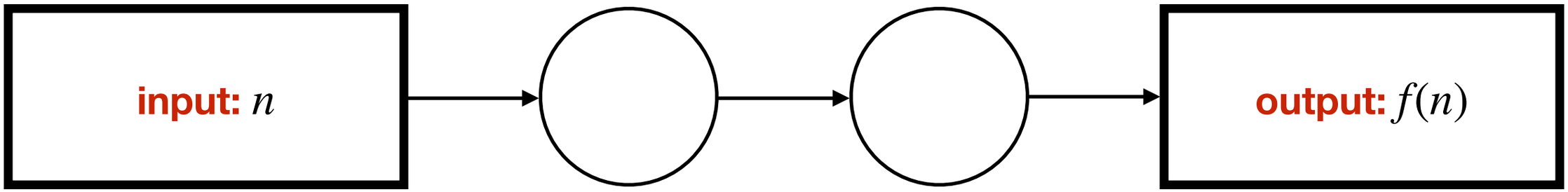
1. Algorithms
2. word RAM model of computation
3. Data structures (array, list, queue, stack)
4. Basic approaches to algorithm design: greedy algorithms, dynamic programming

Algorithms



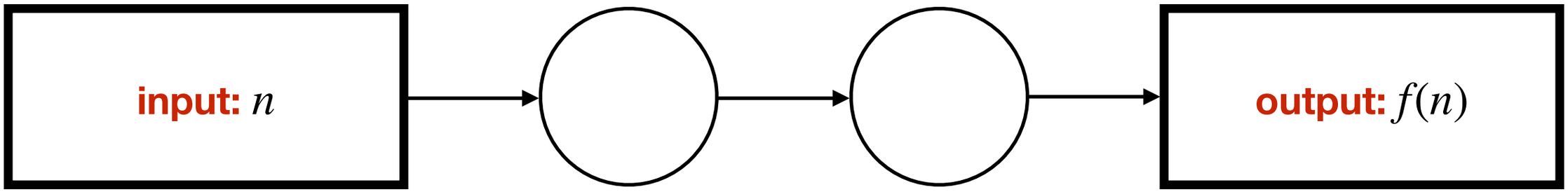
Muhammad ibn Musa al-Khwarizmi, c. 780-850

Algorithms



An algorithm is a **sequence of elementary operations** that transforms the input into the output

Algorithms



Primality test

$$f(n) = \begin{cases} 1, \text{ if } n \text{ is prime} \\ 0, \text{ otherwise} \end{cases}$$

Applications of prime numbers

The **RSA encryption system** is one of the most widely used methods of securely transmitting information

It relies on large prime numbers, that are extremely hard to find

Open Problem: new Mersenne prime

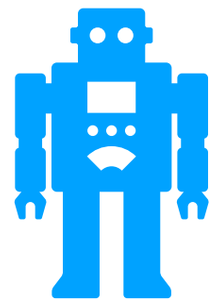
Find a new prime of form $2^n - 1$. More information: **mersenne.org**

Algorithms

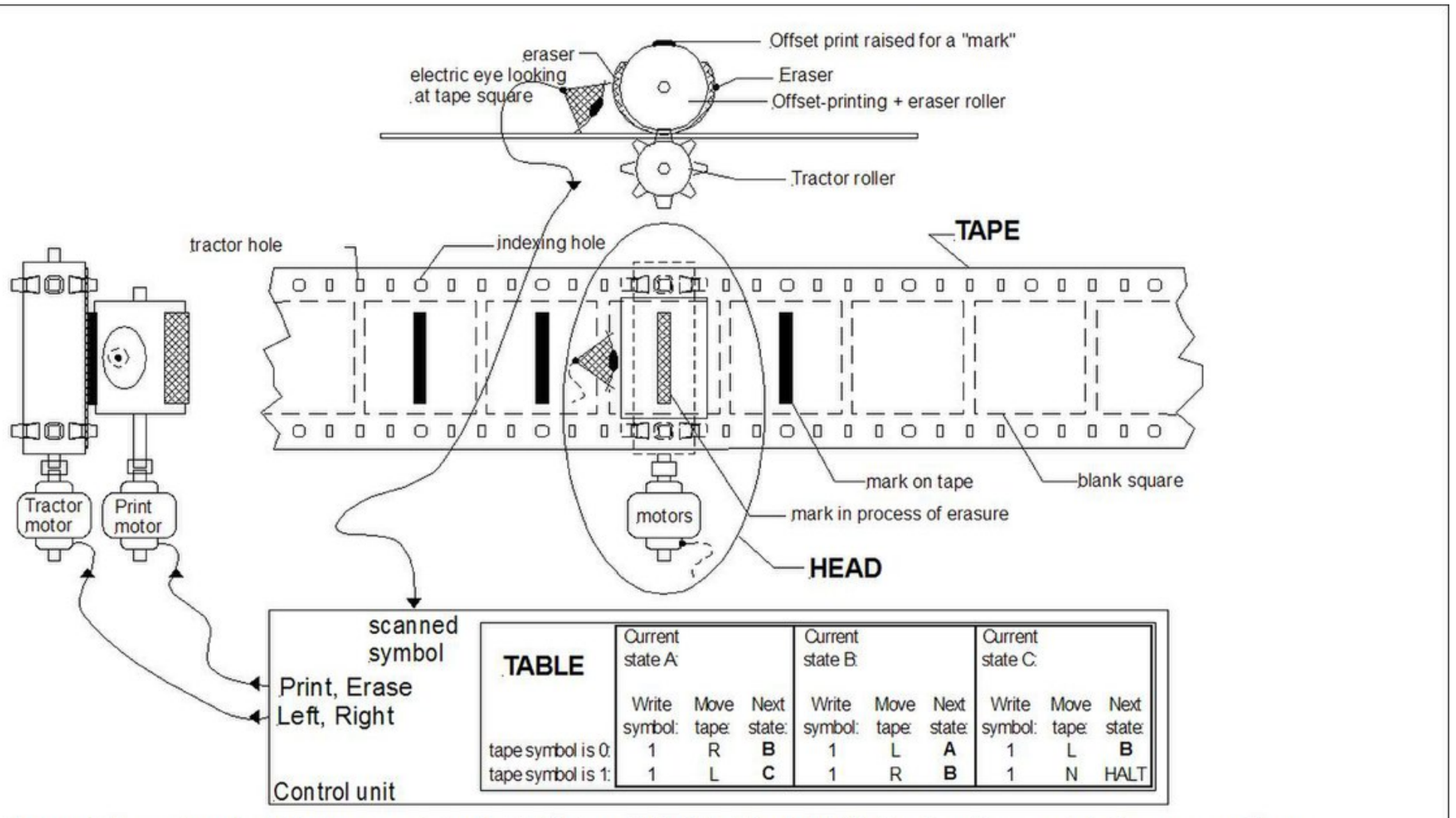
- **Ideal algorithm:** simple to implement, fast, uses little memory.

- To design such algorithms, we need a simple, realistic **model of computation**. The model should not depend on programming languages.

- We will use the word RAM model: it mimics a real computer, but is not over-complicated

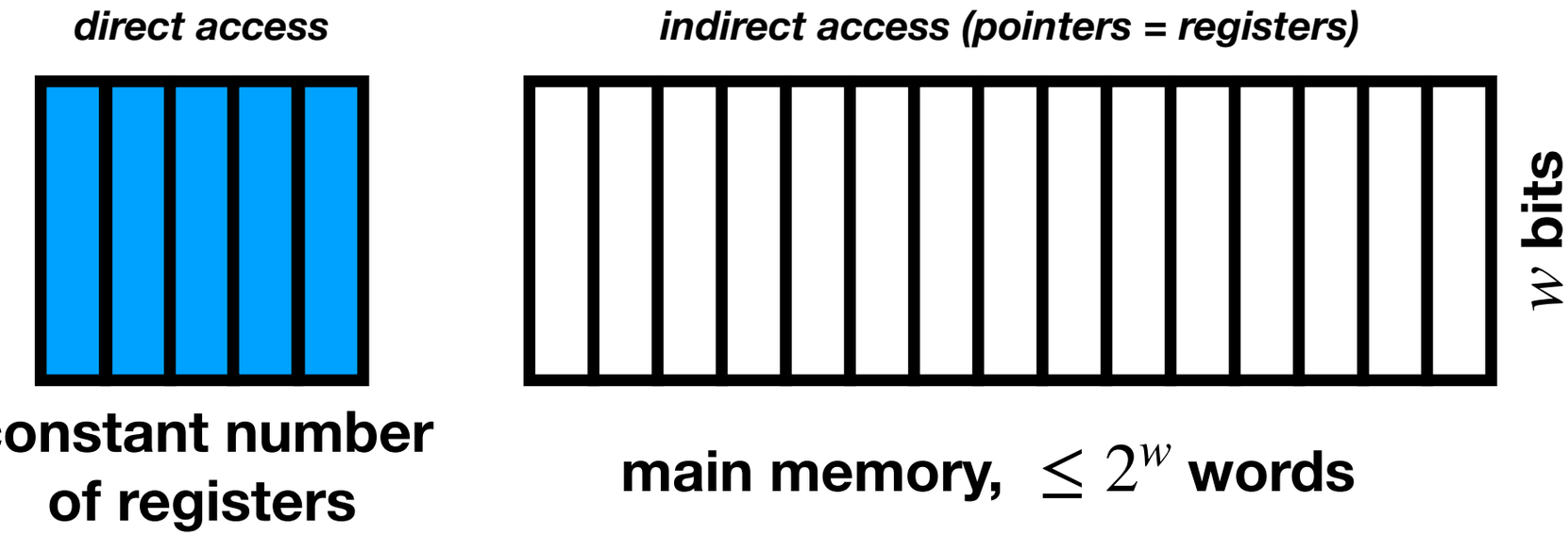


Turing machine



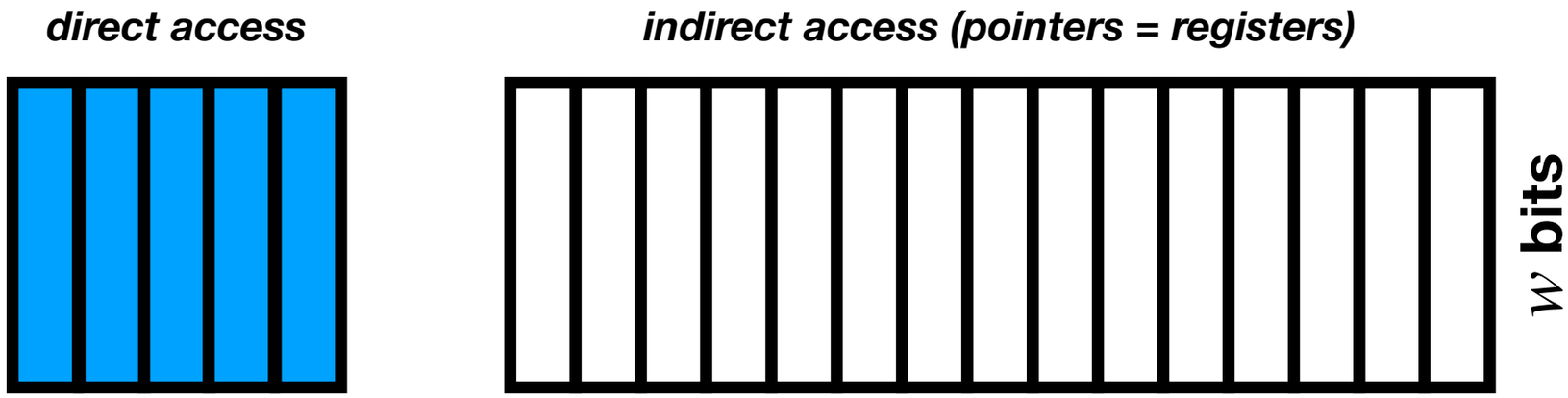
A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

word RAM model



Elementary operations: basic arithmetic and bitwise operations on registers, conditionals (if/then), goto, copying words between registers and main memory, malloc (add an extra memory word), halt

word RAM model

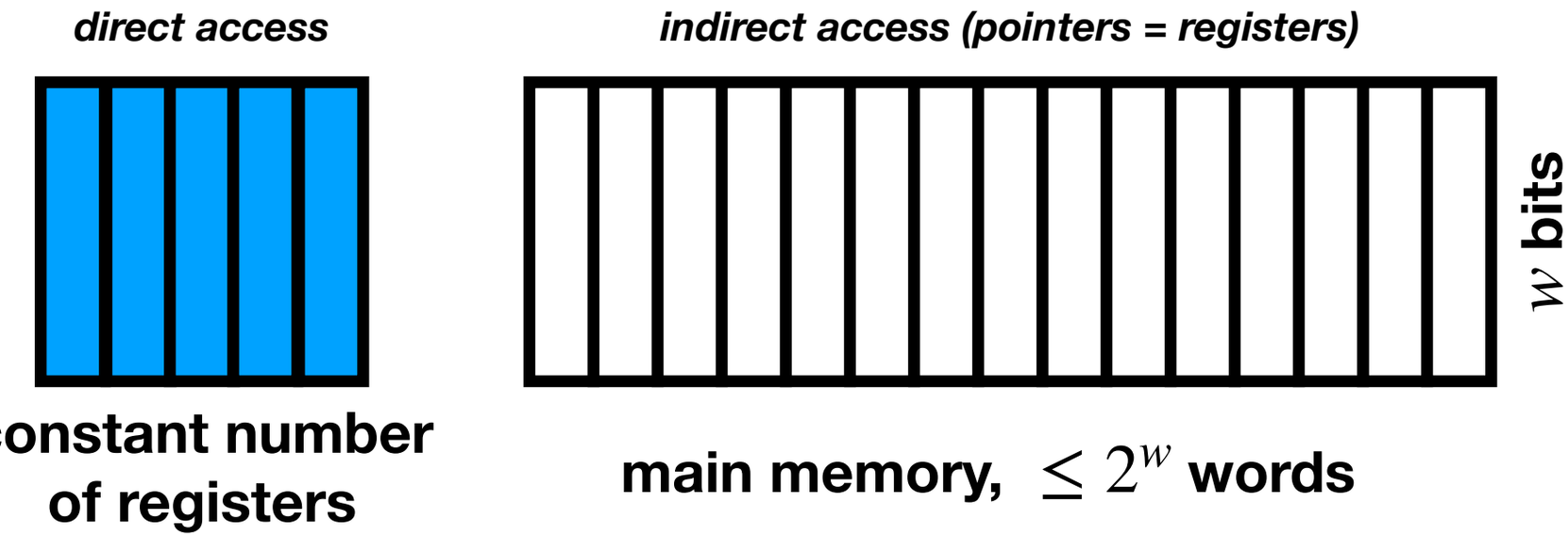


At initialisation, the memory stores the input of size n using n words.

To index it, we need registers of length $w \geq \log n$.

Theoretically, it means that the model changes when the size of the input grows. In practice, it means that we cannot treat problems of size n, where $n \geq 2^w$ for $w = 32$ or $w = 64$ (not a limitation).

word RAM model



To describe algorithms, we will use pseudocode which has semantics similar to Python / C.

Can be rewritten using the elementary operations only.

For details, see the extra material.

Complexity of an algorithm

- $Space(n)$ = the maximum number of memory words used for an input of size n

- $Time(n)$ = the maximum number of elementary operations used for an input of size n

Primality test

```
copy n to a register
for i ← 2 to n − 1 do
    mult ← i
    while mult < n do
        mult ← mult + i
    if mult = n then
        return 0
return 1
```

Space: $n + c$ (input represented in unary)

Time: $\leq c_2 \cdot \sum_{i=2}^{n-2} \lfloor \frac{n}{i} \rfloor \leq c_2 \cdot n \ln n$

It is hard to compute the constants exactly.

We will study the **asymptotic growth** of the complexities.

$O(), \Omega(), \Theta()$ notation

Let $f(n), g(n) \in \mathbb{N} \to \mathbb{R}^+$

- We say that $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ : \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$$

- We say that $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ : \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$$

- We say that $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) if

$$\exists n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_+ : \forall n \geq n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Primality test

```
copy n to a register
for i ← 2 to n − 1 do
    mult ← i
    while mult < n do
        mult ← mult + i
    if mult = n then
        return 0
return 1
```

$$f(n) = \begin{cases} 1, \text{ if } n \text{ is prime} \\ 0, \text{ otherwise} \end{cases}$$

Space: $O(n)$ (input in unary)

Time: $\leq c_2 \cdot \sum_{i=2}^{n-2} \frac{n}{i} \leq c_2 \cdot n \ln n = O(n \log n)$

Asymptotic complexity vs real efficiency

- Asymptotic complexity **matters**. A $\Theta(n)$-time algorithm is slower than a $\Theta(\log n)$-time algorithm if n is large enough.

- An algorithm with time complexity $\Theta(n^2)$ can be faster than an algorithm with time complexity $\Theta(n)$ for all **reasonable values** of n if the hidden constant in $\Theta(n)$ is too large

- Sometimes, the time (or the space) complexity is high, but the **inputs** on which this complexity is reached **are very rare** in practice

- The **programming language** used has a real impact on performance

Data structures

Rivers and Tides: Andy Goldsworthy's project by T. Riedelsheimer

Data structures

- Imagine that we have a collection of objects, i.e. a database of DNA sequences
- We must be able to quickly extract all sequences that have a certain property (i.e. contain a certain gene)
- Scanning the whole database each time is too expensive
- Solution: preprocess the database and store certain information about it in an organised form
- This form is called a data structure

Data structures

We care about:

- the space the data structure occupies
- construction time
- query time
- update time

Storing a sequence of elements

The basic task is to store a **sequence** $e_0, e_2, \ldots, e_{n-1}$ of elements (e.g., integers, floating-point numbers, complex objects, etc.)

Ideally, we would like to maintain the following operations:

- **Random access**: given i, access e_i
- **Access** the first element (e_0), the last element (e_{n-1})
- **Insertion** at the beginning (before e_0), at a random position (between e_i and e_{i+1}), at the end (after e_{n-1})
- **Deletion** of the first element (e_0), of a random element (e_i), of the last element (e_{n-1})

Elementary data structures

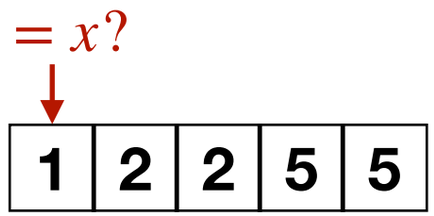
- array (*tableau*)
- linked list (*liste chaînée*)
- queue (*file*)
- stack (*pile*)

Array

- Contiguous memory area of a fixed size, pre-allocated
- Random access in $O(1)$ time
- Insertion, deletion impossible
- Corresponds to **classic arrays** of programming languages:
  - bracketed arrays in C or Java
  - std::array in C++ 2011
  - numpy.array in Python

Linear and binary search in a sorted array

Linear search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Linear search

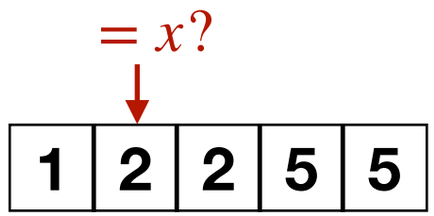
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $e_i = x$ **then**
    return 1
return 0

Time = $O(n)$

Space = $O(n)$

Linear and binary search in a sorted array

Linear search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Linear search

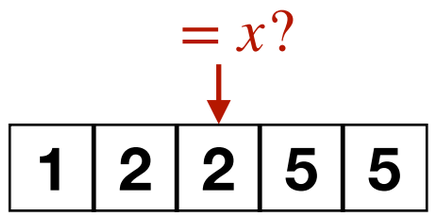
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $e_i = x$ **then**
    return 1
return 0

Time = $O(n)$

Space = $O(n)$

Linear and binary search in a sorted array

Linear search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Linear search

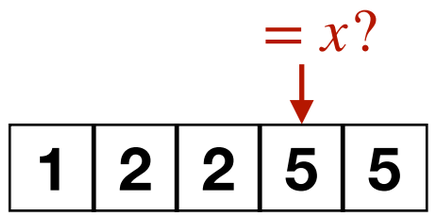
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $e_i = x$ **then**
    return 1
return 0

Time = $O(n)$

Space = $O(n)$

Linear and binary search in a sorted array

Linear search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Linear search

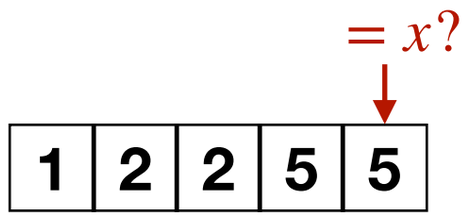
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $e_i = x$ **then**
    return 1
return 0

Time = $O(n)$

Space = $O(n)$

Linear and binary search in a sorted array

Linear search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Linear search

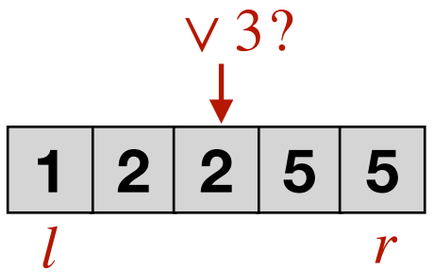
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $e_i = x$ **then**
    return 1
return 0

Time = $O(n)$

Space = $O(n)$

Linear and binary search in a sorted array

Binary search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Binary search

$l \leftarrow 0, r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if $e_m = x$ **then**
    return 1
  else if $e_m < x$
    $l = m + 1$
  else if $e_m > x$
    $r = m - 1$
return 0

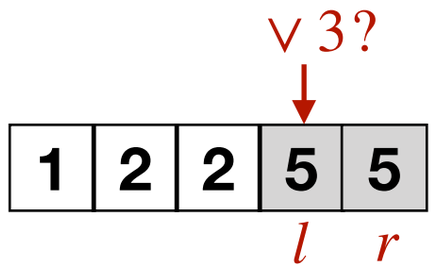
Time = $O(\log n)$

Space = $O(n)$

At each step, the search area shrinks by a factor of at least two.

Linear and binary search in a sorted array

Binary search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Binary search

$l \leftarrow 0, r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if $e_m = x$ **then**
    return 1
  else if $e_m < x$
    $l = m + 1$
  else if $e_m > x$
    $r = m - 1$
return 0

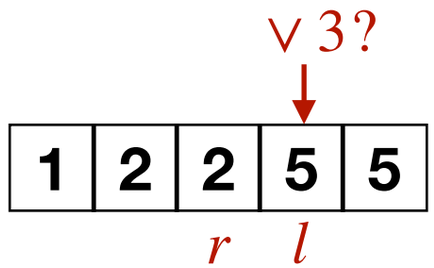
Time = $O(\log n)$

Space = $O(1)$

At each step, the search area shrinks by at least two times.

Linear and binary search in a sorted array

Binary search



Given an integer x, return 1 if $e_i = x$, and 0 otherwise

Binary search

$l \leftarrow 0, r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if $e_m = x$ **then**
    return 1
  else if $e_m < x$
    $l = m + 1$
  else if $e_m > x$
    $r = m - 1$
return 0

Time = $O(\log n)$

Space = $O(n)$

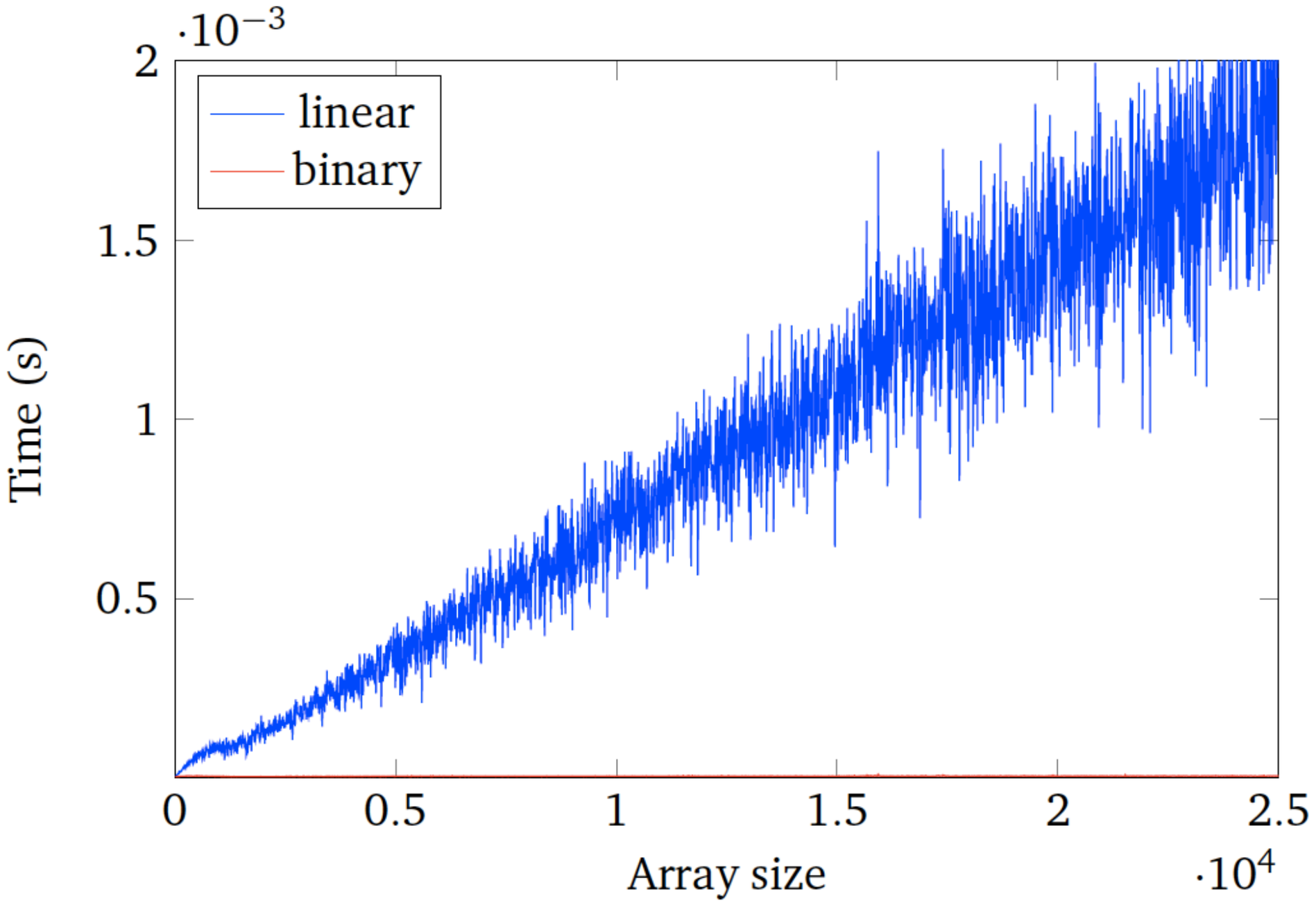
At each step, the search area shrinks by at least two times.

Linear vs binary search in a sorted array

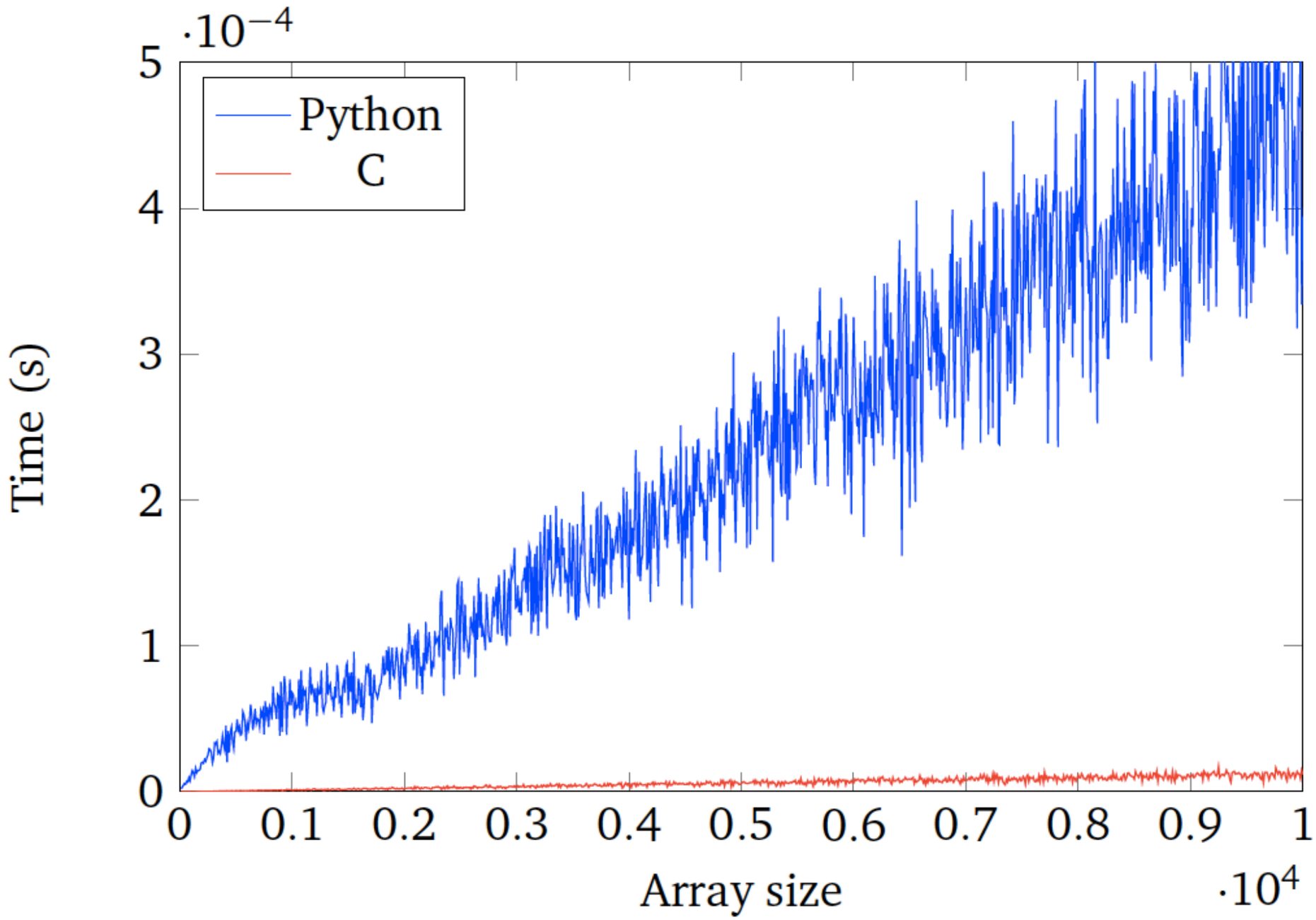
Given an array of integers $a_1 \leq a_2 \leq \ldots \leq a_n$ and an integer x, return 1 if $x = a_i$ for some i, and 0 otherwise.

- Linear search algorithm: $\Theta(n)$ time
- Binary search algorithm: $\Theta(\log n)$ time

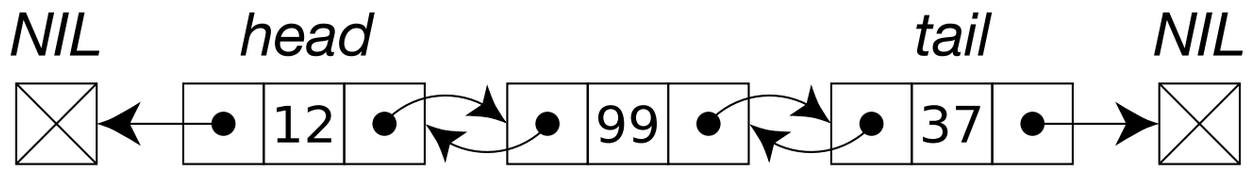
Linear vs binary search in a sorted sequence



Linear search in a sorted sequence: Python vs C

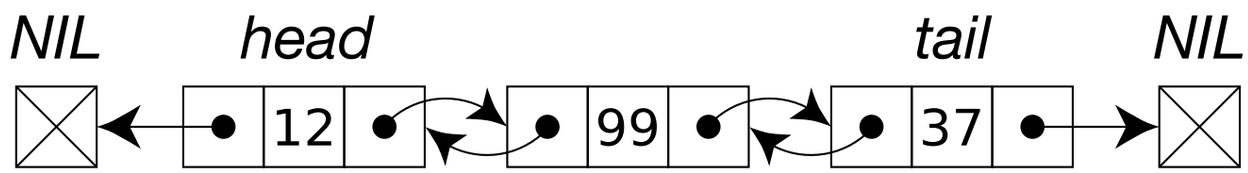


Doubly linked list



- The head and the tail can be accessed in $O(1)$ time
- Random access in $O(n)$ time
- Insertion/deletion at head / tail in $O(1)$ time
- **In programming languages:** std::list in C++

Doubly linked list



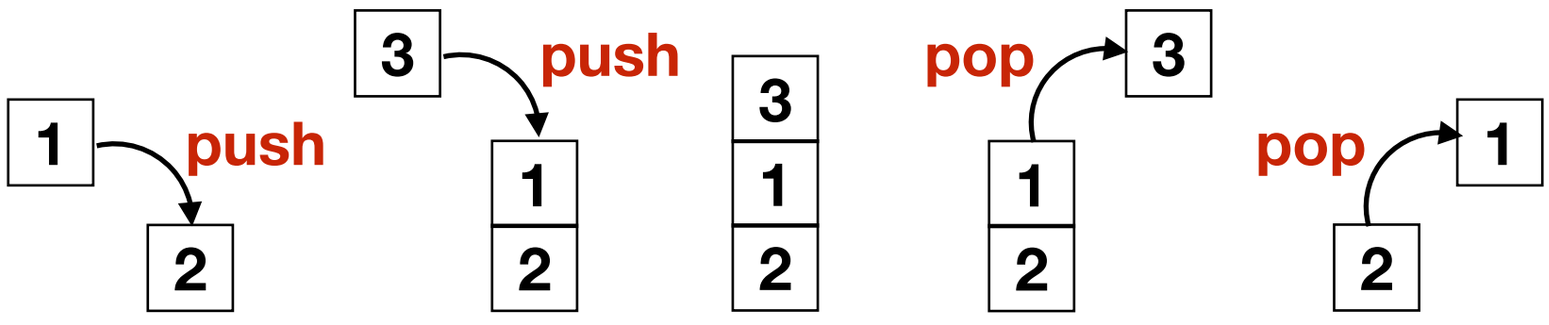
Insert(L, x)

$x\,.\,next \leftarrow L\,.\,head$
if $L\,.\,head \neq NIL$ **then**
  $L\,.\,head\,.\,prev = x$
$L\,.\,head \leftarrow x$
$x\,.\,prev = NIL$

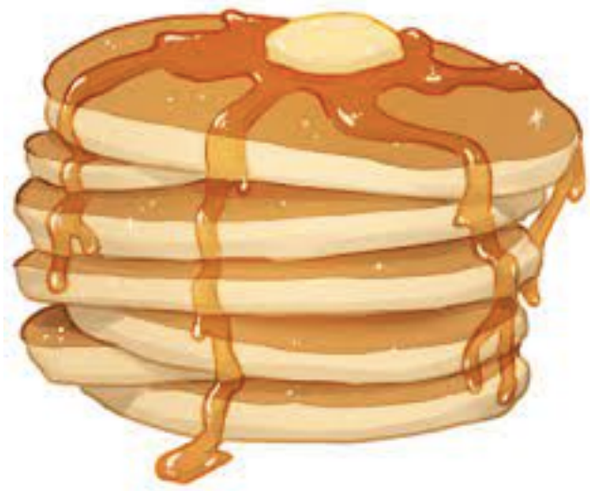
Delete(L, x)

if $x\,.\,prev \neq NIL$ **then**
  $x\,.\,prev\,.\,next \leftarrow x\,.\,next$
else $L\,.\,head \leftarrow x\,.\,next$
if $x\,.\,next \neq NIL$
  $x\,.\,next\,.\,prev \leftarrow x\,.\,prev$

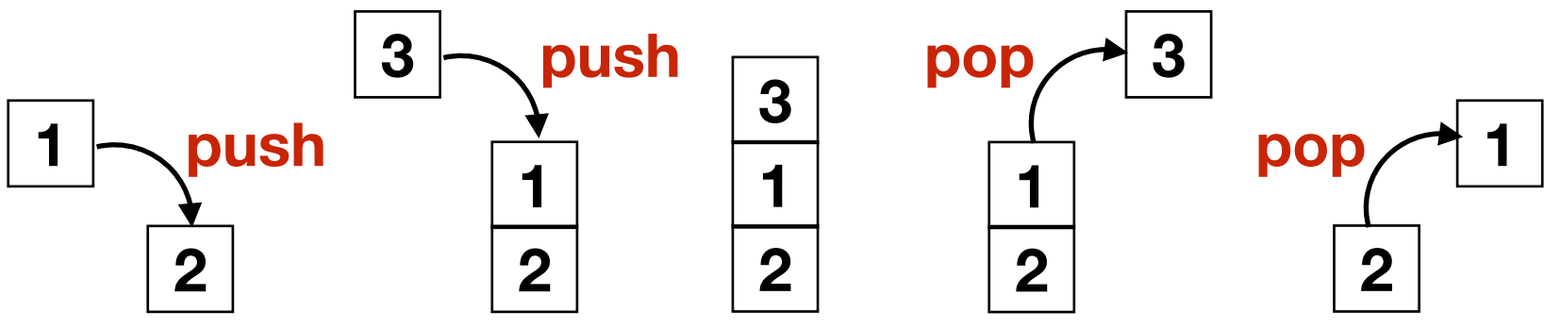
Stack (or LIFO for last-in-first-out)



- Abstract data structure that can be implemented in different ways, e.g. with a linked list
- Access to the top element (**peek**) in $O(1)$ time
- Insertion to the top (**push**) in $O(1)$ time
- Deletion of the top element (**pop**) in $O(1)$ time
- **In programming languages:**
  - std::stack in C++
  - not explicit in Python, but standard lists can be used



Stack (or LIFO for last-in-first-out)

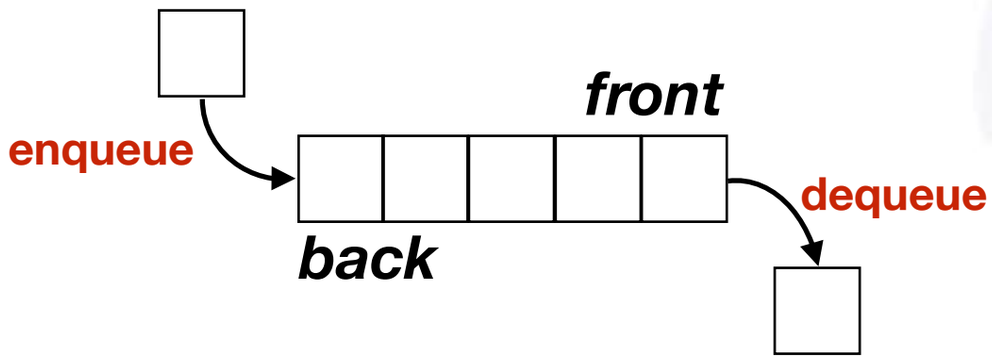


Open problem: Optimum Stack Generation

Given a finite alphabet Σ and a string $X \in \Sigma^n$. Find a shortest sequence of stack operations push, pop, emit (=print the top element) that prints out X. You must start and finish with an empty stack.

The current best algorithm by Bringmann et al. solves the problem in $\tilde{O}(n^{2.8603})$ time. Can it be done faster?

Queue (or FIFO for first-in-first-out)



- Abstract data structure, can be implemented in different ways (e.g. with a doubly linked list)
- Access to the elements in the back / front in $O(1)$ time
- Deletion from the front (**dequeue**) and insertion to the back (**enqueue**) in $O(1)$ time
- **In programming languages:**
  - std::queue in C++
  - not explicit in Python, collections.deque can be used

Lecture 1

Basic approaches to algorithm design

Basic approaches

- Dynamic Programming
- Greedy
- Divide and Conquer — next lecture

Common idea:

•To solve a large, complicated problem, break it into many smaller subproblems.
• Solve the subproblems.
• Recover a solution of the large problems from the solutions of the small subproblems.

1 - Dynamic Programming

Dynamic programming scheme:
Break the problem into many closely related sub-problems, memorize the result of the sub-problems to avoid repeated computation.

Myriads of applications:

- Computational biology: RNA folding, string similarity measures, read mapping, ...
- Operations research: Bellman-Ford algorithm for shortest path routing in networks, ...
- Combinatorial Optimization
- Graph algorithms: FPT algorithm parametrized by treewidth.

Warm-up example: Fibonacci numbers

Fibonacci numbers are defined by the following recurrence relation:

$$F_0 = 0, \qquad F_1 = 1, \qquad \text{and for } n \geq 2 : \ F_n = F_{n-1} + F_{n-2}$$

First Fibonacci numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$$

Problem (Fibonacci Number)

Input : An integer n
Output : The value of F_n.

First idea: recursive algorithm

First idea: straightforward recursive algorithm from the definition of Fibonacci numbers:

Algorithm 1 Fibo: recursive algorithm

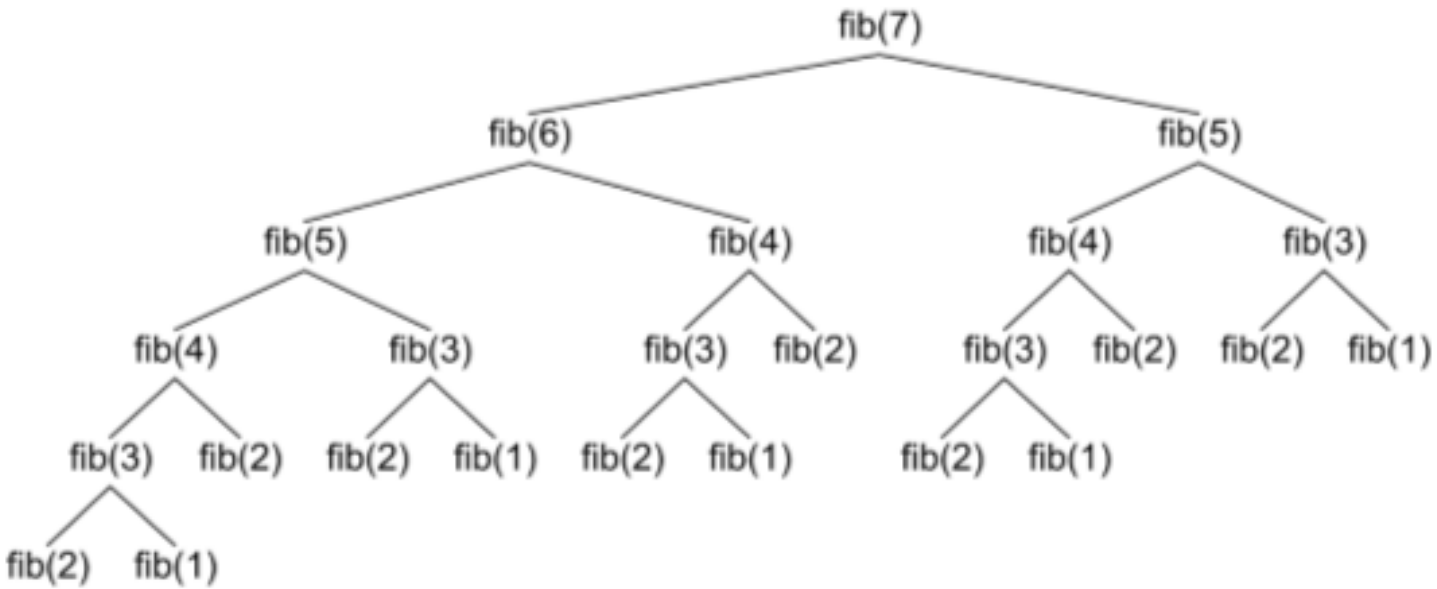
```
1: Fibo(n):
2: if n ≤ 1 then
3:     return n
4: return Fibo(n-1) + Fibo(n-2)
```

Solve the recurrence: $T(n) = T(n-1) + T(n-2)$
Complexity: EXPONENTIAL :(

Question: Why is it so bad??

Why is it so bad??

Answer: Because we compute many time the same values



Memoization!

Instead of computing many times the same value, we compute it once and store it.

Algorithm 2 Fibo: Dynamic Programming

```
1: Fibo(n):
2: Tab ← zeros(n)                    ▷ Array to stock the values of F_i
3: Tab[0] ← 0
4: Tab[1] ← 1
5: for i ← 2 to n do
6:     Tab[i] = Tab[i-1] + Tab[i-2]
7: return Tab[n]
```

- Time complexity: $O(n)$
- Space complexity: $O(n)$

Basic Steps in Designing Dynamic Programming Algorithms

- Relate the problem recursively to smaller sub-problems. (Transition function, f(n) = f(n-1)+f(n-2))
- Organize all sub-problems as a dynamic programming table. (A table for all values of n)
- Fill in values in the table in an appropriate order ($n = 0, 1, 2, 3 \ldots$)

How similar are two strings?

In many applications one want to know how similar two strings are:

- Spell checker the user types something like *struing*, what existing word is close to it?
- Bioinformatics: to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.
- Also for Machine Translation, Information Extraction, Speech Recognition...

Edit distance problem

- The edit operations are:
  - **R**eplacement: replace a letter by another one
  - **D**elete: delete a letter
  - **I**nsert: insert a letter

(introduced by Vladimir Levenstein for error correction)

Problem (Edit Distance)

Input : Two strings S_1 and S_2
Output : Edit distance between S_1 and S_2, i.e. the smallest number of edit operations required to transform S_1 into S_2

Example: $S_1 = \textit{vintners}$ (un vigneron) and $S_2 = \textit{winters}$

Relate the original problem to smaller problems

Problem (Minimum Edit Distance)

Input : Two strings S_1 and S_2 with lengths n and m
Output : Minimum number of basic operation to transform S_1 into S_2

Denote by $D[i,j]$ the distance between $S_1[1,i]$ and $S_2[1,j]$.

Goal: Compute $D[n,m]$
Strategy:
- Compute all values $D[i,j]$ for $i = 0,\ldots,n$ and $j = 0,\ldots,m$
- Each value takes constant time to compute thanks to memoization.

Tasks: Compute $D(i,j)$ knowing $D(u,v)$ for all (u,v) "smaller than" (i,j), which means two tasks:

- find (and prove) a recurrence relation.
- find an efficient way to compute it: tabular computation

The recurrence relation

Denote by $D[i,j]$ the distance between $S_1[1,i]$ and $S_2[1,j]$.

- Base condition: $D(i,0) = i$ and $D(0,j) = j$.

- Recurrence relation: for $i,j > 0$:

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(deletion of } S_1[i]\text{)} \\ D(i,j-1)+1 & \text{(deletion of } S_2[j]\text{)} \\ D(i-1,j-1)+t(i,j) & \text{where } t(i,j) = \begin{cases} 1 & \text{if } S_1(i) \neq S_2(j) \\ 0 & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

Proof by case inspection.

Dynamic programming table

- Create a $n \times m$ array, where the cell (i, j) will contain $D(i, j)$.
- Initialise the first column and the first row and compute the cells row by row.

$D(i,j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4	4	4	*			
n	5	5							
e	6	6							
r	7	7							

Time and space: $O(|S_1| \cdot |S_2|)$

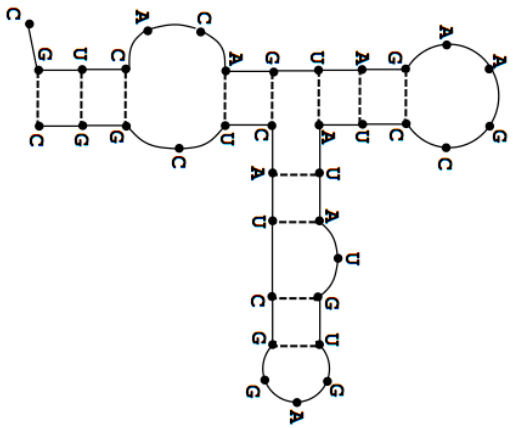
In 2017, Backurs and Indyk showed that conditioned on a widely believed hypothesis, this is the best possible time one can achieve! (see paper here)

RNA folding

Ribonucleic acids (RNA) are molecules essential in various biological processes. RNAs are often represented as sequences of nucleotides: adenine (A), cytosine (C), guanine (G), uracil (U).

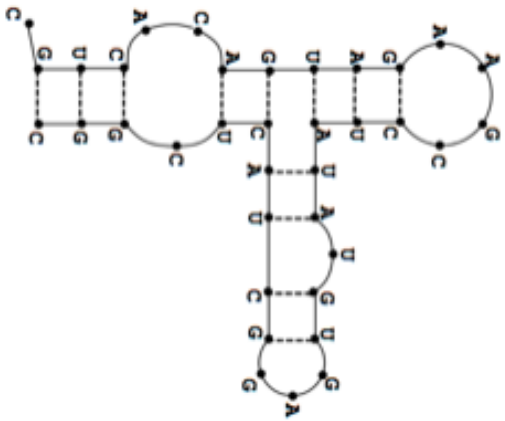
CGUCACAGUAGAAGCCUAUAUGUGAGGCUACUCGGC

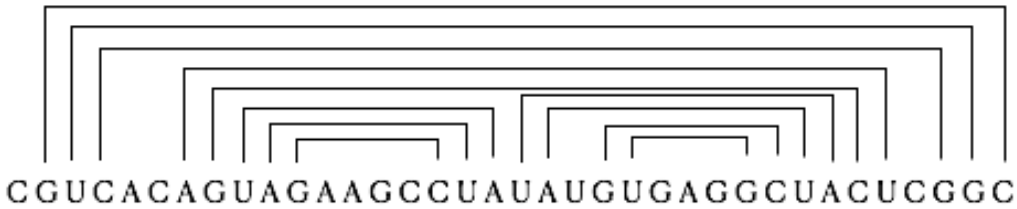
- When we represent an RNA in this form, we forget that it is a 3D molecule, and the 3D shape defines how RNA interacts with other molecules.
- Computing (and storing) the 3D shape of an RNA is not an easy task. As a compromise, one can consider RNA secondary structure:



© rosalind.info

RNA folding



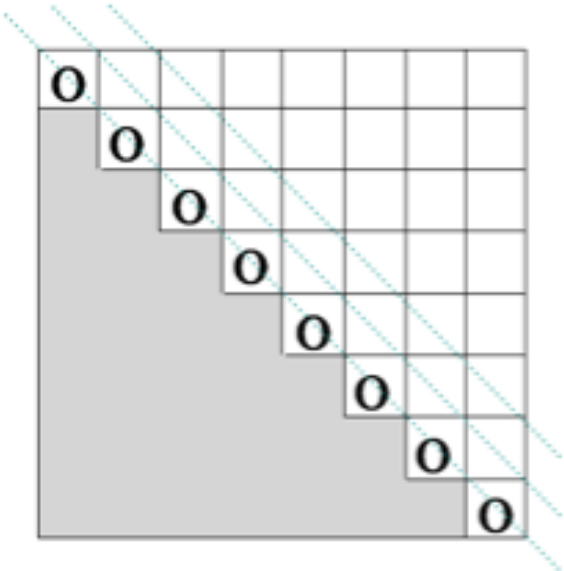


- A can form a (hydrogen) bond with U, and C with G
- Bonds cannot cross
- **Task:** given a sequence of nucleotides, find an optimal secondary structure that maximizes the number of bonds

RNA folding: DP algorithm

In 1980, Nussinov and Jakobson showed that an optimal secondary structure can be found in $O(n^3)$ time, where n is the length of the input RNA sequence, S.

We will only show how to compute the maximal possible number of bonds.



$DP[i,j]$:= max. # of bonds for $S[i,j]$;
our goal is to compute $DP[1,n]$.

$$DP[i,i] = 0$$

$$DP[i,j] = \max \begin{cases} DP[i+1,j]; \\ DP[i+1,k-1] + DP[k+1,j] + 1 \\ \text{for all } k \text{ such that } S[i] \text{ and } S[k] \text{ form a bond.} \end{cases}$$

We compute the table diagonal by diagonal. There are $O(n^2)$ cells, each taking $O(n)$ time to compute. **In total:** $O(n^3)$ time, $O(n^2)$ space.

RNA folding: State of the art

- In FOCS 2016, Bringmann et al. showed that this problem can be solved in $\tilde{O}(n^{2.8606})$ time.
- There is no $O(n^{\omega-\varepsilon})$-time algorithm for RNA folding, unless a widely believed conjecture from the algorithmic graph theory is false, as was shown by Abboud et al. in FOCS 2015.
- Here ω is the matrix multiplication constant, i.e. it is the constant such that an optimal algorithm multiplying two $n \times n$ matrices takes $O(n^{\omega})$ time. It is known that $2 \leq \omega < 2.373$.

Big open questions: What is the true complexity of RNA folding? What is the true complexity of matrix multiplication?

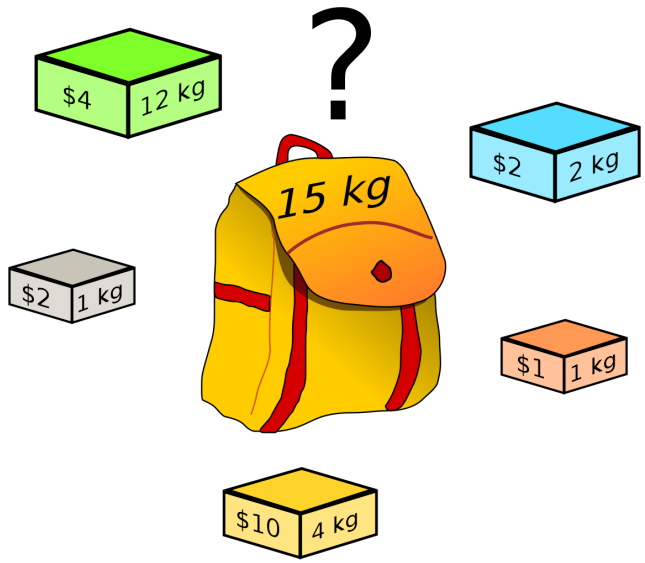
Knapsack

INPUT:

- A knapsack that can hold items of total weight at most W.
- n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$.

Goal: Select some items to put into the knapsack such that:

1. Total weight is at most W.
2. Total value is as large as possible.



Example

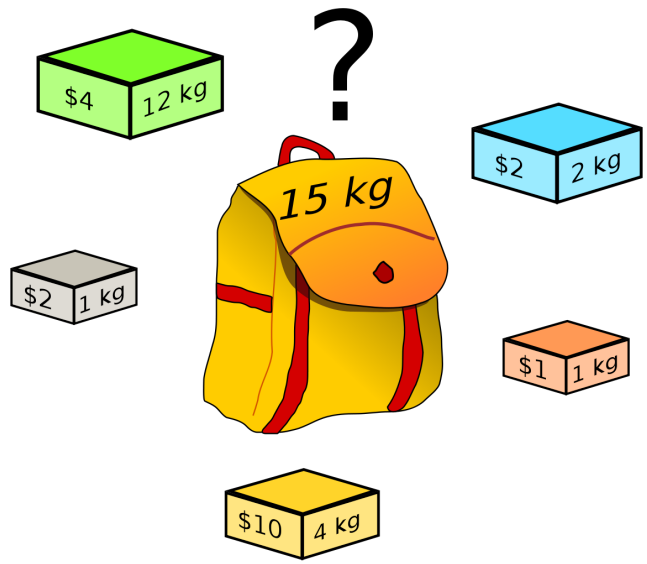
Capacity of the knapsack: $W = 10$

Three items:

- $w_1 = 4$ $\quad v_1 = 2$
- $w_2 = 5$ $\quad v_2 = 3$
- $w_3 = 7$ $\quad v_3 = 4$

Solution: Put items 1 and 2 in the knapsack:

- Total weight is $9 \leq 10$
- Total value is $3 + 2 = 5$



Formal description

INPUT:
Two vectors $w = (w_1, \ldots, w_n)$ (weight vector), $v = (v_1, \ldots, v_n)$ (value vector), and an integer $W > 0$ (capacity)

GOAL: Find $x = (x_1, \ldots, x_n) \in \{0, 1\}^n$ (choose some of the n items)

- subject to $\sum_{i=1}^{n} w_i \cdot x_i \leq W$
- maximizes $\sum_{i=1}^{n} v_i \cdot x_i$

It is an **Optimization Problem**

BRUTE FORCE: 2^n

Can we do better?

DP algorithm for Knapsack

1. Think of the problem as making a sequence of decisions:
   - For each item, decide whether we put it into the knapsack or not

2. Focus on the last item, enumerate the options
   - For the last item, we either put it in, or leave it out.

3. Try to relate each option to a smaller subproblem
   - **Subproblems**: Fill the remaining capacity using the remaining items.
   - leave it out: number of items is now smaller.
   - put it in: both capacity and number of items are smaller.

Relate the original problem to a smaller subproblem

You start with item n, and have two choices:

- Put item n in the knapsack and then solve a new knapsack problem with parameters:
  - $(v_1, \ldots, v_{n-1})$
  - $(w_1, \ldots, w_{n-1})$
  - $W := W - w_{n-1}$
- Don't put item 1 in the knapsack and then solve a new knapsack problem with parameters:
  - $(v_1, \ldots, v_{n-1})$
  - $(w_1, \ldots, w_{n-1})$
  - W

Formalization

Definition of the subproblems:
Let $KP[i, w]$ be the optimal solution when the set of available items is the first i items and the maximum total weight is at most w.

Base condition: $KP[0, w] = 0$ and $KP[i, 0] = 0$.

Recurrence relation for $i, w > 0$:

$$KP[i, w] = \max \begin{cases} KP[i-1, w] & \text{item } i \text{ is left out} \\ KP[i-1, w-w_i] + v_i & \text{item } i \text{ is put in (need } w \geq w_i) \end{cases}$$

Pseudocode

Algorithm 3 Knapsack: Dynamic Programming

1: **for** $i \leftarrow 0$ to n **do**
2: KP[i,0]=0
3: **for** $w \leftarrow 0$ to W **do**
4: KP[0,w]=0
5: **for** $i \leftarrow 1$ to n **do**
6: **for** $w \leftarrow 1$ to W **do**
7: **if** $w < w_i$ **then**
8: $KP[i, w] \leftarrow KP[i-1, w]$
9: **else**
10: $$KP[i, w] = \max \left\{ \begin{array}{l} KP[i-1, w] \\ KP[i-1, w-w_i] + v_i \end{array} \right\}$$
11: **return** $KP[n, W]$

- KP is an array of size $n \times W$
- We fill the first line and the first column (base condition)
- Then we fill the table column by column.

Time and complexity analysis

The computation of each cell of the array KP takes constant time, so

- Time: $O(nW)$
- Space: $O(nW)$ (size of the array)

Is this a polynomial-time algorithm? Its complexity depends on W, which is not accounted for in the size of the problem. We say that the algorithm is pseudo-polynomial.

Recent result by Bateni et al. suggests that there is no algorithm with running time $O\left((nW)^{0.99}\right)$ (again, conditional on a widely believed hypothesis).

On the other hand, they showed that if the values or the weights are bounded by a constant, the problem can be solved in $\tilde{O}(n + W)$ time.

5 easy steps to dynamic programming

1. Define subproblems
2. Guess (part of solution)
3. Relate subproblem solutions
4. Build DP table
5. Solve original problem = a subproblem or by combining subproblems

5 easy steps to dynamic programming

1. Define subproblems
2. Guess (part of solution)
3. Relate subproblem solutions
4. Build DP table
5. Solve original problem = a subproblem or by combining subproblems

Yeah. . . but why "dynamic programming"?

- The term was coined in by Richard Bellman in the 1950s.
- The Secretary of Defense at that time was hostile to mathematical research, and Bellman sought an impressive name to avoid confrontation.
  - *"it's impossible to use dynamic in a pejorative sense"*
  - *"something not even a Congressman could object to"* (Bellman, R. E., Eye of the Hurricane, An Autobiography).

2 - Greedy algorithms

Greedy Technique

• As with dynamic programming, in order to be solved with the greedy technique, the problem must have the optimal substructure property.

• The problems that can be solved with the greedy method are a subset of those that can be solved with dynamic programming.

• The idea of greedy technique is the following: at every step you have a choice. Instead of evaluating all choices recursively and picking the best one, pick what looks like locally the best choice, and go with that.

• So basically a greedy algorithm picks the locally optimal choice hoping to get the globally optimal solution.

• Coming up with greedy heuristics is easy, but proving that a heuristic gives the optimal solution is tricky (usually).

The Fractional Knapsack Problem

The difference is that now the items are infinitely divisible: can put $\frac{1}{2}$ (or any fraction) of an item into the knapsack.

Formal description

INPUT:
Two vectors $w = (w_1, \ldots, w_n)$ (weight vector), $v = (v_1, \ldots, v_n)$ (value vector), and an integer $W > 0$ (capacity)

GOAL: Find $x = (p_1, \ldots, p_n) \in [0, 1]^n$ (choose some fractions of the n items)

- subject to $\sum_{i=1}^{n} w_i \cdot p_i \leq W$
- maximizes $\sum_{i=1}^{n} v_i \cdot p_i$

Greedy criterion

We want to make a sequence of decisions on what to put in the knapsack.

If you are indeed choosing what to put into your knapsack, what would be your strategy?

Greedy criterion

We want to make a sequence of decisions on what to put in the knapsack.

If you are indeed choosing what to put into your knapsack, what would be your strategy?

Algorithm: Iteratively pick the item with the greatest value-per-weight ratio (maximum $\frac{v_i}{w_i}$)

If, at the end, the knapsack cannot fit the entire last item with greatest value-per-weight ratio among the remaining items, we will take a fraction of it to fill the knapsack.

Time Analysis

- The algorithm starts by sorting the value-per-weight ratios, and in the second lecture we will see that n numbers can be sorted in $O(n \log n)$ time.
- The main phase of the algorithm takes another $O(n)$ time.
- Hence the total running time is $O(n \log n)$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.
- Let i be the smallest index such that $p_i \neq q_i$. There is $p_i > q_i$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.
- Let i be the smallest index such that $p_i \neq q_i$. There is $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.
- Let i be the smallest index such that $p_i \neq q_i$. There is $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$.
- Set $q' = (q'_1, q'_2, \ldots, q'_n) = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j - \epsilon\frac{w_i}{w_j}, \ldots, q_n)$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.
- Let i be the smallest index such that $p_i \neq q_i$. There is $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$.
- Set $q' = (q'_1, q'_2, \ldots, q'_n) = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j - \epsilon\frac{w_i}{w_j}, \ldots, q_n)$.
- q' is a feasible solution: $\sum_{i=1}^{n} q'_i \cdot w_i = \sum_{i=1}^{n} q_i \cdot w_i \leq W$.

Proof of correctness

Assume w.l.o.g. that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$. Let $ALG = (p_1, \ldots, p_n)$ be the solution output by the algorithm and $OPT = q = (q_1, q_2, \ldots, q_n)$ be an optimal solution.

- Assume for contradiction that $ALG \neq OPT$.
- Let i be the smallest index such that $p_i \neq q_i$. There is $p_i > q_i$.
- So there exists $j > i$ such that $p_j < q_j$.
- Set $q' = (q'_1, q'_2, \ldots, q'_n) = (q_1, \ldots, q_{i-1}, q_i + \epsilon, q_{i+1}, \ldots, q_j - \epsilon\frac{w_i}{w_j}, \ldots, q_n)$.
- q' is a feasible solution: $\sum_{i=1}^{n} q'_i \cdot w_i = \sum_{i=1}^{n} q_i \cdot w_i \leq W$.
- However, $\sum_{i=1}^{n} q'_i \cdot v_i > \Sigma_{i=1}^{n} q_i \cdot v_i$, which contradicts the optimality of OPT.