



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Chapter 1:

From problems to algorithms and programs

Computational problems

Definition (Computational problem)

A *computational problem* is a triple

$$\mathcal{P} = (I, S, R),$$

where I is a set of admissible instances, S is a set of possible solutions, and $R \subseteq I \times S$ is the relation of acceptable instance–solution pairs. For $x \in I$, an output $y \in S$ solves the instance exactly when $(x, y) \in R$.

A relation permits several acceptable answers. For an index of a maximum in $[4, 9, 9, 2]$, either occurrence of 9 may be valid unless the contract requires the first or last.

Algorithms and programs

Definition (Algorithm)

An *algorithm* for $\mathcal{P} = (I, S, R)$ is a finite, unambiguous procedure that, for every $x \in I$, terminates and produces some $y \in S$ such that $(x, y) \in R$.

Definition (Program)

A *program* is a concrete expression of one or more algorithms in a programming language, together with choices of representations, interfaces, and machine-level operations. Several programs may implement the same algorithm, and the same problem may admit fundamentally different algorithms.

Specifying the maximum value

Let D be a totally ordered domain. Inputs are nonempty finite sequences over D .

$$I = \bigcup_{n \geq 1} D^n, \quad S = D.$$

Acceptable instance–solution pairs

$$(A, m) \in R_{\max} \iff (\exists i \in \{0, \dots, |A| - 1\}) A[i] = m \\ \wedge (\forall j \in \{0, \dots, |A| - 1\}) A[j] \leq m.$$

The first condition requires an input value. The inequality alone would permit an artificial value larger than every record.

Maintaining the maximum of a prefix

```
Maximum(A)
  require length(A) >= 1 # precondition
  best <- A[0]
  for i <- 1 to length(A)-1
    if A[i] > best
      best <- A[i]
  return best
```

On $A = [4, 1, 7, 7, 2]$, the successive values of best are

4, 4, 7, 7, 7.

The trace describes one execution.

Why the maximum scan is correct

Prefix invariant

Immediately before the iteration with index i , best is the maximum of $A[0], \dots, A[i - 1]$.

1. **Initialization.** The processed prefix contains only $A[0]$.
2. **Preservation.** Comparing $A[i]$ with best preserves the invariant for the extended prefix.
3. **Termination.** The processed prefix is the whole input, so the returned value satisfies $(A, m) \in R_{\max}$.

For $n = |A|$, the scan performs exactly $n - 1$ comparisons and uses constant auxiliary storage.

Different algorithms for the same problem

A parallel reduction could compare several pairs simultaneously.

Its implementation and cost would differ from the sequential scan. Its admissible inputs and correctness obligations would be the same.

Fibonacci numbers make this comparison more evident. Different methods return the same mathematical value while performing very different amounts of work.

This comparison motivates the cost notation introduced in Chapter 2.

The Fibonacci sequence

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 3.$$

The first terms are

$$1, 1, 2, 3, 5, 8, 13, 21, \dots$$

Computational task

Given an integer $n \geq 1$, compute F_n .

The task stays fixed while the method and representation change.

Binet's formula

Both φ^n and $\widehat{\varphi}^n$ satisfy the recurrence, as does every linear combination

$$A\varphi^n + B\widehat{\varphi}^n.$$

Choosing A and B to satisfy $F_1 = F_2 = 1$ yields

$$F_n = \frac{\varphi^n - \widehat{\varphi}^n}{\sqrt{5}}. \tag{1}$$

Proposition (Correctness of Binet's formula)

For every integer $n \geq 1$, this formula equals the recursively defined Fibonacci number F_n .

Exponential sequences satisfying the recurrence

Look for a sequence $X_n = a^n$ satisfying $X_n = X_{n-1} + X_{n-2}$.

$$X_{n-1} = a^{n-1}, \quad X_{n-2} = a^{n-2}.$$

Substitution gives

$$a^n = a^{n-1} + a^{n-2}.$$

For $a \neq 0$, divide by a^{n-2} :

$$a^2 = a + 1 \quad \Longleftrightarrow \quad a^2 - a - 1 = 0.$$

Its two roots are

$$\varphi = \frac{1 + \sqrt{5}}{2}, \quad \hat{\varphi} = \frac{1 - \sqrt{5}}{2}.$$

Exact identities and finite representations

Binet's formula is exact over the real numbers.

A direct digital implementation uses finite-precision approximations of $\sqrt{5}$, φ , and $\hat{\varphi}$.

Rounding can produce an incorrect integer for large n .

Representation matters

A correct mathematical identity does not automatically give a correct implementation under every representation.

Trying the floating-point formula

```
from math import sqrt

def binet(n):
    """Compute F_n using floating point arithmetic."""
    phi = (1 + sqrt(5)) / 2
    phi_hat = (1 - sqrt(5)) / 2
    return round((phi**n - phi_hat**n) / sqrt(5))

for n in [10, 50, 71, 80]:
    approximate = binet(n)
    print(f"Binet={approximate}")
    print(sqrt(5))
```

Direct recursive Fibonacci

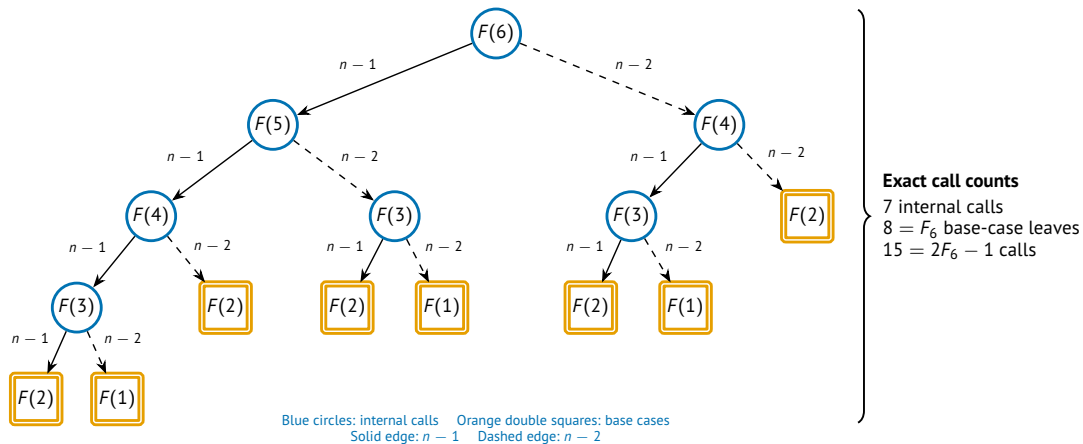
For every integer $n \geq 1$:

```
Fibonacci-recursive(n)
  if n <= 2
    return 1
  return Fibonacci-recursive(n-1) + Fibonacci-recursive(n-2)
```

The base case implements $F_1 = F_2 = 1$. The recursive case implements $F_n = F_{n-1} + F_{n-2}$. These cases give correctness directly from the recurrence.

The two recursive calls are evaluated independently, even when they need the same Fibonacci number.

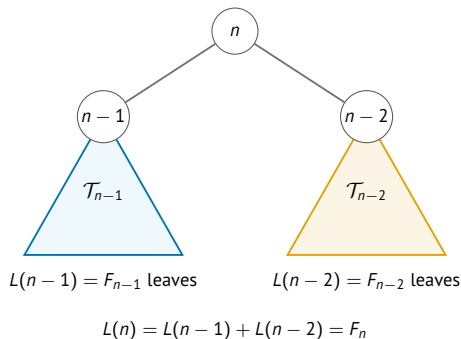
Repeated subproblems in the call tree



Computing F_6 evaluates F_4 inside the F_5 call and again as the second child of the root. Duplication continues at lower levels.

Correctness does not imply efficiency.

Decomposing the recursion tree



Let L_n , also written $L(n)$, count leaves.

$$L_1 = L_2 = 1.$$

For $n > 2$, the root joins the two subtrees and is not a leaf.

$$L_n = L_{n-1} + L_{n-2}.$$

Each leaf belongs to exactly one subtree.

Retaining every Fibonacci value

For an integer $n \geq 1$:

```
Fibonacci-array(n)
  if n <= 2
    return 1
  allocate F[1 .. n]
  F[1] <- 1
  F[2] <- 1
  for i <- 3 to n
    F[i] <- F[i-1] + F[i-2]
  return F[n]
```

For $n \geq 3$, the loop performs $n - 2$ additions. The array uses a number of memory cells proportional to n .

Retaining only the two preceding values

At iteration i , only F_{i-1} and F_{i-2} are needed.

```
Fibonacci-iterative(n)
  if n <= 2
    return 1
  previous <- 1
  current <- 1
  for i <- 3 to n
    next <- previous + current
    previous <- current
    current <- next
  return current
```

The arithmetic work is essentially the same as in the array version. The stored state contains only what the next step needs.

Time, storage, and a trace for iteration

Under the usual **unit-cost model**:

Method	Time	Auxiliary space
Array version	Linear	Linear
Two-value version	Linear	Constant

For $n = 6$, (previous, current) evolves as

$(1, 1), (1, 2), (2, 3), (3, 5), (5, 8).$

A trace is evidence for one input, not a proof for every n .

Experiment, part 1 of 3: exact iteration

```
from math import sqrt

def fibonacci(n):
    """Compute F_n with exact iterative Fibonacci algorithm."""

    if not isinstance(n, int) or n < 1:
        raise ValueError("n must be a positive integer")

    previous = 0
    current = 1

    for _ in range(n):
        previous, current = current, previous + current
    return previous
```

Experiment, part 2 of 3: Binet evaluation

```
def binet(n):  
    """Compute F_n using floating point arithmetic."""  
    if not isinstance(n, int) or n < 1:  
        raise ValueError("n must be a positive integer")  
  
    phi = (1 + sqrt(5)) / 2  
    phi_hat = (1 - sqrt(5)) / 2  
    return round((phi**n - phi_hat**n) / sqrt(5))
```

Experiment, part 3 of 3: comparing outputs

```
for n in [10, 50, 71, 80]:  
    exact = fibonacci(n)  
    approximate = binet(n)  
  
    print(  
        f"n={n:3d}  exact={exact}  "  
        f"Binet={approximate}  "  
        f"difference={approximate - exact}"  
    )
```

Run the combined script and observe how the difference changes as n increases.

Combining consecutive Fibonacci updates

Iteration computes preceding values in order, with a number of updates proportional to n .

A fixed matrix can transform the state containing F_k, F_{k-1} into the state containing F_{k+1}, F_k .

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}.$$

Repeated transformations correspond to matrix powers.

Set $F_0 = 0$, consistently with $F_2 = F_1 + F_0$.

The Fibonacci matrix identity

Lemma (Fibonacci matrix identity)

Let $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. For every $n \geq 2$,

$$A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}.$$

The upper-left entry of A^{n-1} yields F_n .

The convention $F_0 = 0$ supplies the lower-right entry when $n = 2$.

Matrix identity proof: base and hypothesis

Base case $n = 2$.

$$A^1 = A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}.$$

This uses $F_2 = F_1 = 1$ and $F_0 = 0$.

Induction hypothesis. Fix $n > 2$ and assume the identity for $n - 1$:

$$A^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}.$$

Matrix identity proof: the inductive step

Substitute the induction hypothesis in $A^{n-1} = A^{n-2}A$:

$$\begin{aligned} A^{n-1} &= \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix}. \end{aligned}$$

The two sums in the first column satisfy

$$F_{n-1} + F_{n-2} = F_n, \quad F_{n-2} + F_{n-3} = F_{n-1}.$$

Hence $A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$ for every $n \geq 2$. □

Reducing the exponent by repeated squaring

Repeated multiplication computes A^n using $n - 1$ matrix multiplications, still a linear number of stages.

For $n = 2m$, compute A^m once and square it:

$$A^n = A^{2m} = A^m A^m = (A^m)^2.$$

For $n = 2m + 1$,

$$A^n = A^{2m+1} = A(A^m)^2.$$

Thus

$$A^n = \begin{cases} (A^{n/2})^2, & n \text{ even,} \\ A (A^{\lfloor n/2 \rfloor})^2, & n \text{ odd.} \end{cases}$$

Both cases compute only one smaller power, with approximately half the exponent.

Reconstructing the thirteenth power

The recursive computation of A^{13} considers the exponents

$$13, \quad 6, \quad 3, \quad 1.$$

Once A is known, reconstruct the powers in reverse order:

$$A^3 = A(A)^2,$$

$$A^6 = (A^3)^2,$$

$$A^{13} = A(A^6)^2.$$

The smaller power is reused when squaring.

Matrix powering by repeated squaring

I denotes the identity matrix with the same dimensions as A . For an integer $k \geq 0$:

```
Matrix-power(A, k)
    if k == 0
        return I
    if k == 1
        return A

    half = Matrix-power(A, floor(k / 2))
    result = half * half

    if k is odd
        result = result * A

    return result
```

Only one recursive call is made. An odd exponent requires one extra multiplication.

Extracting a Fibonacci number from a power

For an integer $n \geq 0$:

```
Fibonacci-matrix(n)
    if n == 0
        return 0

    Q = [[1, 1],
         [1, 0]]

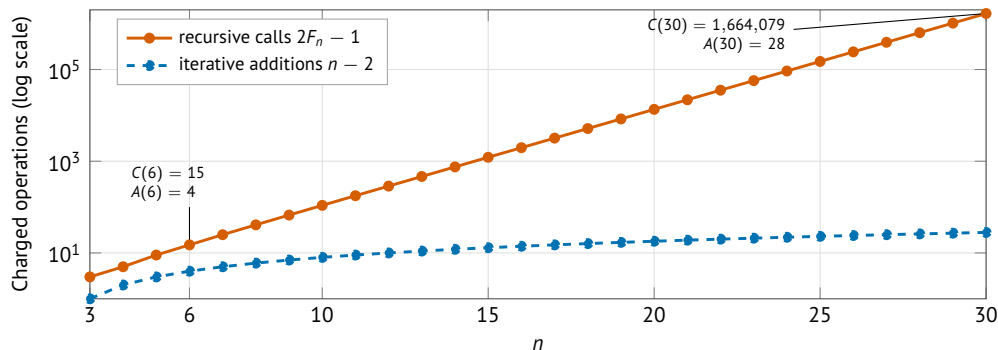
    P = Matrix-power(Q, n - 1)
    return P[0, 0]
```

For $n = 0$, return the convention $F_0 = 0$. For $n \geq 2$, use the matrix identity and the zero-based entry $(0, 0)$.

For $n = 1$, $Q^0 = I$ has upper-left entry $1 = F_1$.

The general powering routine is reusable beyond Fibonacci numbers.

Comparing charged-operation growth



Calls and additions are different charged operations. The curves compare growth rates, not equal execution times. Matrix powering is omitted because matrix multiplications are a different counter; their unit-cost count grows logarithmically.

Chapter 1 exercises 1–4

1. Formalize “return an index of a minimum element” as a relation. Give one version that permits any minimizing index and another that requires the last minimizing index. Exhibit an input for which the sets of acceptable outputs differ.
2. Draw the complete recursion tree for F_7 . Count leaves, internal nodes, and total calls, and verify the formulas $L_n = F_n$ and $C_n = 2F_n - 1$.
3. Trace both iterative algorithms for $n = 8$. Count additions and compare their stored state after every iteration.

Chapter 1 exercises 5–8

5. Use $x^2 = x + 1$ to verify that both φ^n and $\hat{\varphi}^n$ satisfy the Fibonacci recurrence. Then check that Equation (1) has the correct two initial values.
6. Implement the three principal algorithms in Python and instrument them without using execution time as the counter. Plot calls or additions against n and explain how the observed curves motivate asymptotic notation.
7. Reconstruct the induction proof of the matrix identity without consulting the text, including the role of $F_0 = 0$. Then give pseudocode for repeated squaring and prove that it uses only logarithmically many recursive levels under the unit-cost model.
8. **The first failure of Binet's formula.** Use the Python implementation of Binet's formula and compare its output with the exact iterative computation. Determine the smallest integer $n \geq 1$ for which the difference between the two results is not zero. Report the exact value, the value returned by Binet's formula, and their difference.

Sources for further study

- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd ed., Chapters 1–2.
- Dasgupta, Papadimitriou, and Vazirani, *Algorithms*, Prologue, Sections 0.1–0.2.
- Horowitz, Sahni, and Rajasekaran, *Computer Algorithms*, Sections 1.1–1.2.

Next chapter in the afternotes: Correctness, cost models, and asymptotic growth.

Thank you!



UNIVERSITY
OF TRIESTE