

UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Common Algorithmic Techniques

The Art of Computer Programming

Pierre Senellart



2 November 2026

Outline

- Overview
- Making change
- Recursion
- Memoization
- Dynamic programming
- Greedy algorithms
- Divide and conquer
- Master theorem
- Coda

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Overview



Common algorithmic techniques

- Lecture 3: how to **measure** the cost of an algorithm. This lecture: a handful of **techniques for designing** one, and the tools to predict their cost
- Four techniques cover a large part of what you will write:

Brute force	try all candidate solutions, keep the good ones
Divide and conquer	cut the problem into **independent** smaller problems, solve each recursively, combine the results
Dynamic programming	when the smaller problems **repeat**, compute each of them once and store the result
Greedy	build the solution by always making the choice that looks best **now**, and never reconsider it

- The difficulty is rarely writing the code: it is **recognizing which technique the problem allows**

Which technique applies?

Technique	What the problem must satisfy
Brute force	nothing
Divide and conquer	optimal substructure
Dynamic programming	optimal substructure, overlapping subproblems
Greedy	optimal substructure, greedy choice property

Optimal substructure an optimal solution of the problem is built from optimal solutions of **smaller instances of the same problem**

Overlapping subproblems the same smaller instances come back **many times**

Greedy choice property the locally best choice is part of **some** globally optimal solution

We meet the three properties on one example, then treat each technique in turn

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Making change



Making change

- Coins of n fixed values $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n$, in unlimited supply; in the euro zone, in cents:

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	2	5	10	20	50	100	200

- **Question**: in how many ways can a sum $S \in \mathbb{N}$ be made?
- For $S = 7$ with coins of 1, 2, and 5, there are **6 ways**:

1 + 1 + 1 + 1 + 1 + 1 + 1	1 + 1 + 1 + 2 + 2	1 + 2 + 2 + 2
1 + 1 + 1 + 1 + 1 + 2	1 + 1 + 5	2 + 5

- The order does not count: 2 + 5 and 5 + 2 are the same way

Brute force

- A combination uses at most $\lfloor S/p_i \rfloor$ coins of value p_i: enumerate all the combinations in that range and count those of total S
- For $S = 7$ with coins of 1, 2, 5: $8 \times 4 \times 2 = 64$ combinations to test, 6 of which work
- In general

$$\Theta\left(\left(\left\lfloor\frac{S}{p_1}\right\rfloor + 1\right) \times \cdots \times \left(\left\lfloor\frac{S}{p_n}\right\rfloor + 1\right)\right) = \Theta\left(\frac{S^n}{\prod_i p_i}\right) = O(S^n) \text{ combinations}$$

- Brute force is **always available** and easy to get right: a good way to test a better algorithm on small inputs
- But $O(S^n)$ is unusable: for $S = 200$ and the 8 euro coins, about 6×10^9 combinations

A recurrence

- For $S \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $N(S, k)$ be the number of ways to make S **using only the first k kinds of coins**
- Base cases:

$$\begin{cases} N(0, k) = 1 & \text{for all } k \\ N(S, 0) = 0 & \text{for all } S > 0 \\ N(S, k) = 0 & \text{for all } S < 0 \text{ and all } k \end{cases}$$

- Recursive case: either a way **does not use** the coin p_k at all, or it **uses it at least once**, and the rest makes $S - p_k$ with the same coins:

$$\boldsymbol{N(S, k) = N(S, k-1) + N(S - p_k, k)} \qquad \text{for } S > 0,\ k > 0$$

- The answer to the problem is $N(S, n)$; since every case is built from **smaller** cases, the formula can be evaluated: it is an **algorithm**

The recurrence at work, $S = 7$, coins 1, 2, 5

S \ k	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	2
4	0	1	3	3
5	0	1	3	4
6	0	1	4	5
7	0	1	4	**6**

$$\begin{cases} N(0,k) = 1 & \\ N(S,0) = 0 & \text{for } S > 0 \\ N(S,k) = 0 & \text{for } S < 0 \\ N(S,k) = N(S,k-1) + N(S-p_k,k) & \text{for } S,k > 0 \end{cases}$$

- Column $k = 0$: no coin, nothing can be made (except 0)
- Column $k = 1$: only coins of 1, exactly one way
- $N(7,3) = N(7,2) + N(2,3) = 4 + 2 = \mathbf{6}$

A recursive function

Python

```python
def nb_ways(s, p, k=None):
    """Number of ways to make s with the first k kinds of coins."""
    if k is None:
        k = len(p)
    if s == 0:                    # base cases
        return 1
    if s < 0 or k == 0:
        return 0
    return nb_ways(s, p, k - 1) + nb_ways(s - p[k - 1], p, k)

print(nb_ways(7, (1, 2, 5)))      # 6
```

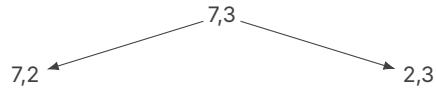
- The formula **is** the program: three base cases, one recursive case, nothing else
- Correct, short, easy to trust – and, as we will see, hopelessly slow

The tree of recursive calls, $S = 7$, coins 1, 2, 5

7,3

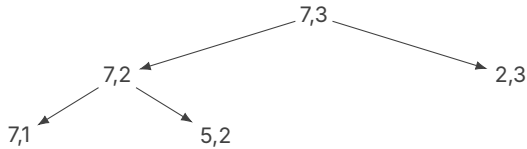
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



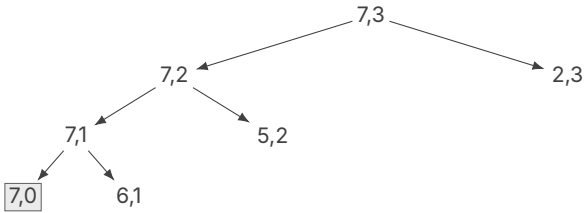
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



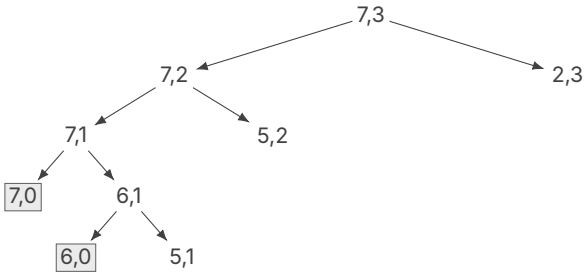
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



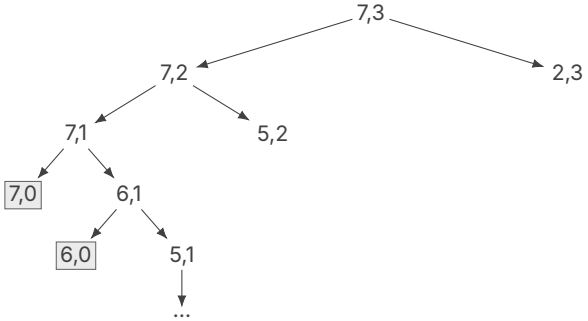
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



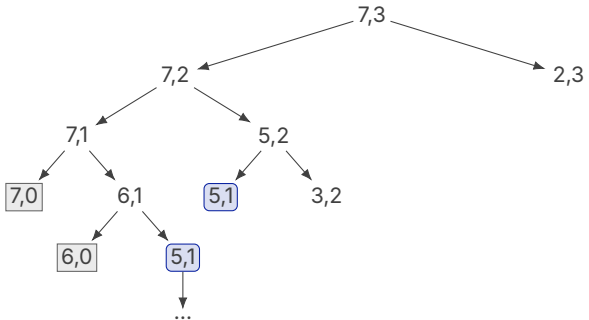
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



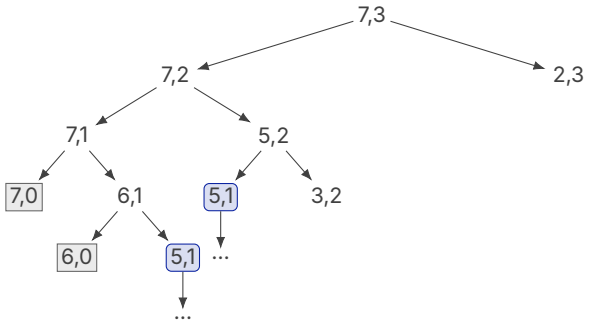
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases

The tree of recursive calls, $S = 7$, coins 1, 2, 5



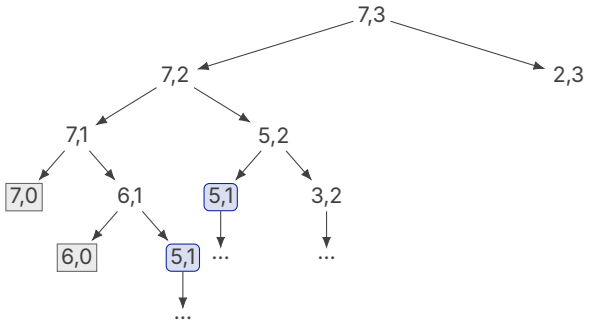
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases
- The pair $(5, 1)$ is computed **twice**, and so are many others: the subproblems **overlap**

The tree of recursive calls, $S = 7$, coins 1, 2, 5



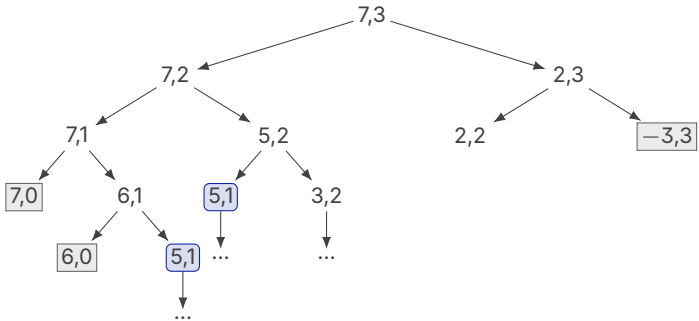
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases
- The pair $(5, 1)$ is computed **twice**, and so are many others: the subproblems **overlap**

The tree of recursive calls, $S = 7$, coins 1, 2, 5



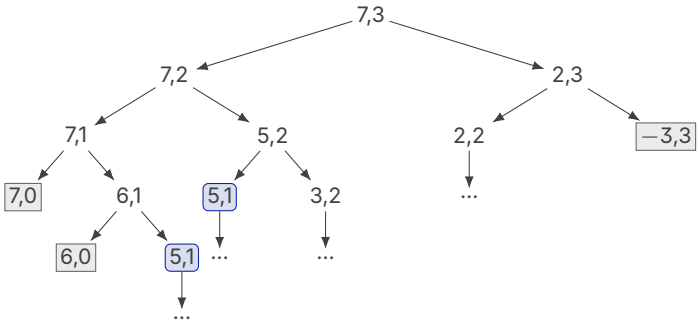
- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases
- The pair $(5, 1)$ is computed **twice**, and so are many others: the subproblems **overlap**

The tree of recursive calls, $S = 7$, coins 1, 2, 5



- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases
- The pair $(5, 1)$ is computed **twice**, and so are many others: the subproblems **overlap**

The tree of recursive calls, $S = 7$, coins 1, 2, 5



- Each node (S, k) calls $(S, k-1)$ and $(S - p_k, k)$; the leaves are the base cases
- The pair $(5, 1)$ is computed **twice**, and so are many others: the subproblems **overlap**

How expensive is the recursive version?

- The tree has at least one leaf per way of making S, so it grows **at least as fast as the answer**; in the worst case, exponentially
- Measured on the 8 euro coins (CPython 3.12):

S	Ways	Recursive
50	451	0.001 s
100	4 563	0.015 s
200	73 682	0.42 s
300	471 363	3.6 s

- Unusable beyond a few hundred. And yet almost all of that work is **recomputation**: the same pairs (S, k) come back again and again

Remembering the results: memoization

Python

```
def nb_ways(s, p):
    memo = {}                              # (s, k) -> number of ways
    def n(s, k):
        if (s, k) in memo:                 # already computed: return it
            return memo[(s, k)]
        if s == 0:
            result = 1
        elif s < 0 or k == 0:
            result = 0
        else:
            result = n(s, k - 1) + n(s - p[k - 1], k)
        memo[(s, k)] = result              # remember it
        return result
    return n(s, len(p))
```

- The recursion is unchanged; a **memo** short-circuits the calls already made
- Each of the $(S + 1) \times (n + 1)$ pairs (S, k) is computed **at most once**: $\boldsymbol{O(S \times n)}$

Filling the table: dynamic programming

Python

```
def nb_ways(s, p):
    n = len(p)
    t = [[0] * (n + 1) for _ in range(s + 1)]    # the table, all zeros
    for k in range(n + 1):
        t[0][k] = 1                              # N(0, k) = 1
    for i in range(1, s + 1):                    # every sum, in order
        for k in range(1, n + 1):                # the first k kinds of coins
            t[i][k] = t[i][k - 1]
            if p[k - 1] <= i:
                t[i][k] += t[i - p[k - 1]][k]
    return t[s][n]
```

- Same recurrence, **no recursion**: the table is filled in an order that guarantees every value read is already computed
- $\Theta(S \times n)$ time and memory, and no call stack at all
- Note `t[i][k] = t[i][k-1]` then `+= t[i - p[k-1]][k]`: the two terms of the recurrence, the second only when $p_k \leqslant i$

Measured: the same recurrence, three ways

S	Ways	Recursive	Memoized	Dynamic
50	451	0.0011 s	0.00006 s	0.00005 s
100	4 563	0.0151 s	0.00015 s	0.00009 s
200	73 682	0.4155 s	0.00021 s	0.00019 s
300	471 363	3.6093 s	0.00025 s	0.00031 s
1000	321 335 886	–	**RecursionError**	0.0011 s

- The 8 euro coins, CPython 3.12, same machine
- Memoization and dynamic programming do the **same arithmetic** in the same order of magnitude of time; the recursive version does the same arithmetic **thousands of times**
- At $S = 1000$ the memoized version exceeds Python's recursion limit: same complexity, but it uses the **call stack** where the table-filling version uses none

Variant: the fewest coins

- A different question on the same data: what is the **smallest number of coins** that makes S?
- A new recurrence, with a min instead of a sum:

$$\begin{cases} M(0,k) = 0 & \text{for all } k \\ M(S,0) = +\infty & \text{for all } S > 0 \\ M(S,k) = +\infty & \text{for all } S < 0 \\ \mathbf{M(S,k) = \min\big(M(S,k-1),\ M(S-p_k,k)+1\big)} & \text{for } S,k > 0 \end{cases}$$

  where $+\infty$ means “impossible” (with coins of 3 and 5, the sum 1 cannot be made)
- Same three solutions: recursive, memoized, or a table
- **Same pattern**: once the recurrence is written, the three implementations follow mechanically

Reconstructing a solution

- The table gives the **value** of the optimum, not the coins. **Trick**: when taking a min (or a max), **remember which choice** achieved it, then walk back

Python

```
m = [[0] * (n + 1)] + [[infinity] * (n + 1) for _ in range(s)]
used = [[False] * (n + 1) for _ in range(s + 1)]  # coin k used for i?
for i in range(1, s + 1):
    for k in range(1, n + 1):
        c = p[k - 1]
        m[i][k] = m[i][k - 1]                     # M(i, k-1)
        if c <= i and m[i - c][k] + 1 < m[i][k]:
            m[i][k] = m[i - c][k] + 1             # M(i-c, k) + 1
            used[i][k] = True                     # remember the argmin
coins, i, k = [], s, n
while i > 0:                                      # 137 -> [100, 20, 10, 5, 2]
    if used[i][k]: coins.append(p[k - 1]); i -= p[k - 1]
    else: k -= 1
```

Same trick with recursion and memoization: store the choice next to the value

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Recursion



When is a problem recursive?

A problem P can be solved by recursion when:

- it can be described by **a few integer parameters**, $P(n_1, \ldots, n_k)$ – here $k = 2$, with the sum and the number of kinds of coins
- its solution can be obtained from the solutions of $\ell \geqslant 1$ instances $P(n^{(1)}), \ldots, P(n^{(\ell)})$ whose parameters are **no larger**, at least one **strictly smaller**: the **recursive case** ($\ell = 2$ for making change)
- a few instances are solved **directly**: the **base cases**

The two numbers that decide the cost: k, how many parameters, and ℓ, how many recursive calls per instance

Two one-parameter examples

Factorial $P(n) = n!$, with $k = 1$ and $\ell = 1$:

$$P(n) = n \times P(n-1) \text{ for } n \geqslant 1, \qquad P(0) = 1$$

One call per instance, no repetition: recursion is enough

Fibonacci $P(n) = F_n$, with $k = 1$ and $\ell = 2$:

$$P(n) = P(n-1) + P(n-2) \text{ for } n \geqslant 2, \qquad P(0) = 0,\ P(1) = 1$$

Two calls per instance, and $P(n-2)$ is computed twice: the subproblems **overlap**

A two-parameter example: edit distance

The **Levenshtein distance** $d(s, t)$ is the smallest number of insertions, deletions, and substitutions of single characters that turns s into t (used by spell checkers and in DNA comparison).

- $P(n_1, n_2)$: the distance between the prefix of length n_1 of s and the prefix of length n_2 of t; $k = 2$, $\ell = 3$
- Base cases: $P(n_1, 0) = n_1$ and $P(0, n_2) = n_2$ (only deletions, or only insertions)
- Recursive case, for $n_1, n_2 \geqslant 1$: delete, insert, or substitute the last character

$$P(n_1, n_2) = \min \begin{cases} P(n_1 - 1, n_2) + 1 \\ P(n_1, n_2 - 1) + 1 \\ P(n_1 - 1, n_2 - 1) + [s_{n_1} \neq t_{n_2}] \end{cases}$$

where $[s_{n_1} \neq t_{n_2}]$ is 1 if the two characters differ, 0 if they are equal

Edit distance: the values of P

chien \ chat		c	h	a	t
	0	1	2	3	4
c	1	0	1	2	3
h	2	1	0	1	2
i	3	2	1	1	2
e	4	3	2	2	2
n	5	4	3	3	**3**

Python

```
def d(i, j):                    # s, t: global strings
    if i == 0: return j
    if j == 0: return i
    return min(d(i-1, j) + 1,
               d(i, j-1) + 1,
               d(i-1, j-1)
                 + (s[i-1] != t[j-1]))
```

Each cell is the minimum of its upper, left, and upper-left neighbors (plus 0 or 1), and $P(n_1, n_2)$ is the bottom right one: `chien` $\longrightarrow$ `chat` costs **3** edits.

Cost of a plain recursive function

- With $\ell = 1$ recursive call per instance, the calls form a **chain**: $O(n_1 + \cdots + n_k)$ calls, usually acceptable
- With $\ell > 1$, they form a **tree of branching factor ℓ** and depth up to $n_1 + \cdots + n_k$:

$$O\left(\ell^{n_1+\cdots+n_k}\right) \text{ calls: unusable, except on tiny inputs}$$

- Measured on the edit distance of two strings of length n ($\ell = 3$):

n	Recursive
8	0.033 s
10	0.96 s
12	29.8 s
500	centuries

- Two more characters, thirty times the work: this is what an exponential cost looks like

Recursion and the call stack

- Lecture 2: every call that has not returned yet keeps its parameters and local variables on the **call stack**, so a recursion uses stack memory proportional to its **depth**, $O(n_1 + \cdots + n_k)$ – not to its total number of calls
- The stack is **small** (lecture 4: a few megabytes), and the depth is what exhausts it. Whatever the language, a recursion a few thousand calls deep is at risk: a stack overflow in C or C++, an error in Python
- When $\ell = 1$ and the recursive call is the **last thing** the function does, the recursion is a loop in disguise: **rewrite it as a loop** (some C and C++ compilers do it for you, Python never does)
- When $\ell > 1$, the depth is the same, at most $n_1 + \cdots + n_k$, with the same limitation; but the **number of calls** explodes, and that is what the next section addresses

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Memoization



Memoization

- **Memoization**: keep a **memo** of the results already computed, and look it up before computing anything
- The function becomes:
  1. if the memo holds an entry for these parameters, return it
  2. otherwise compute the result as usual (recursively)
  3. store it in the memo, and return it
- The recursion is untouched: the **shape** of the program still follows the recurrence, which is what makes memoization so easy to apply to a working recursive function
- **Cost**: $O(n_1 \times \cdots \times n_k \times \ell)$ operations, $O(n_1 \times \cdots \times n_k)$ memory on the heap, $O(n_1 + \cdots + n_k)$ on the stack

Memoization in practice

Python

```
memo = {}                    # n -> F_n, filled as the calls are made
def fib(n):
    if n in memo:            # already known: no call at all
        return memo[n]
    result = n if n < 2 else fib(n - 1) + fib(n - 2)
    memo[n] = result
    return result

print(fib(200))              # very fast
```

- Four lines added to the recursive function, otherwise unchanged; the memo is an **associative array** (lecture 6) keyed by the parameters, which must be **hashable**
- When the parameters are small contiguous integers, an **array** with a marker for “not computed yet” is faster and more compact

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Dynamic programming



Dynamic programming

- Same recurrence, opposite direction: instead of starting from the answer and recursing down, **start from the base cases** and fill a table **up** to the answer
- Build a table of $(n_1 + 1) \times \cdots \times (n_k + 1)$ cells, write the base cases into it, then loop over the cells in an order such that the values needed are **already there**
- **Cost**: $\Theta(n_1 \times \cdots \times n_k \times \ell)$ operations, $\Theta(n_1 \times \cdots \times n_k)$ memory on the heap, and ***O*(1) on the stack**: no recursion at all
- **The one thing to get right**: the order in which the cells are filled. For making change, increasing S then increasing k; for the edit distance, row by row
- The name is historical (Bellman, 1950s): "programming" here means planning, not writing code

Memoization or dynamic programming?

- The two are **the same algorithm**, and have the same complexity; they differ in how the work is scheduled
- **In favor of memoization**: the code follows the recurrence, so it is easy to write and to trust; only the cells that are **actually needed** are computed, which can be a small part of the table
- **In favor of dynamic programming**: no call stack, so no depth limit; no associative array, so less memory and a smaller constant factor
- **In practice**: write the recursive version first, add a memo, and turn it into a table only if the depth or the constant factor becomes a problem

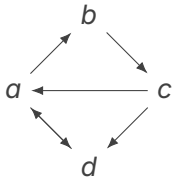
When does dynamic programming apply?

- Two conditions, both necessary:

  Optimal substructure the optimal solution is built from **optimal** solutions of subproblems

  Overlapping subproblems the same subproblems are needed **many times**
- Without **overlapping subproblems**, there is nothing to remember: plain recursion is enough, and we are in the world of **divide and conquer** (next section)
- Without **optimal substructure**, the recurrence is simply false, and no amount of memoization will fix it – the next slide shows a problem where it fails

Optimal substructure, and where it fails



Shortest path the shortest path from x to y through z is the shortest from x to z followed by the shortest from z to y: **optimal substructure holds**, and the same shortest paths are needed again and again, so dynamic programming works

Longest path with cycles forbidden, the longest path from x to y through z is **not** made of the longest paths from x to z and from z to y: the two halves may share a node, and the result is not a path

Here, the longest path from b to d is $b \to c \to a \to d$; it does not contain the longest path from b to a, which is $b \to c \to d \to a$. No recurrence.

Example: the 0–1 knapsack

- n objects, object i of **weight** w_i and **value** v_i; a bag carrying at most W in weight
- Choose $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} w_i \leqslant W$ and $\sum_{i \in I} v_i$ **maximal**; each object is taken or left (hence 0–1)

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
value per unit of weight	6.0	5.5	5.0

$W = 50$

- **Greedy attempt**: take the objects by decreasing value per unit of weight, while they fit. This takes o_1 then o_2: weight 30, value **170**
- But o_2 and o_3 weigh exactly 50 and are worth **260**: the best local choice is not part of the best solution

Knapsack by dynamic programming

- Both conditions hold; let $V(c, k)$ be the best value with capacity c and the first k objects:

$$V(c,k) = \begin{cases} 0 & \text{if } k = 0 \\ \max\big(V(c,k-1),\ V(c-w_k,k-1)+v_k\big) & \text{if } w_k \leqslant c \\ V(c,k-1) & \text{otherwise} \end{cases}$$

  (leave object k out, or take it and lose w_k of capacity)
- $\Theta(W \times n)$ time and memory; on the example, $V(50,3) = 260$, and walking the table back gives $\{o_2, o_3\}$

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Greedy algorithms



Greedy algorithms

- Build the solution **one choice at a time**, always taking what looks best at that moment, and **never reconsidering**
- No table, no recursion: the cost is that of making the choices, typically $\Theta(n)$, plus that of sorting the candidates first when the rule needs it
- Two conditions:

  Optimal substructure as for dynamic programming
  Greedy choice property the locally best choice belongs to **some** optimal solution
- The second condition is the delicate one, and it needs a **proof**: a greedy algorithm that happens to work on the examples one tries is the classic way of shipping a wrong program

Greedy: making change

- **Greedy rule**: hand over the largest coin that fits, and repeat. For 1 euro 37 with euro coins: 100 + 20 + 10 + 5 + 2, five coins – which is optimal
- The rule is optimal for the euro coins, but **not in general**: US coins of 1875–1878, with a twenty-cent piece,

$$p = \{1, 5, 10, 20, 25\},\ S = 40: \quad \begin{array}{l} \text{greedy } 25 + 10 + 5 \text{ (3 coins)} \\ \text{optimal } 20 + 20 \text{ (2 coins)} \end{array}$$

- The greedy choice property **fails**: the coin of 25 is in no optimal solution. Nothing in the algorithm can detect this – only a proof, or a comparison with the exact answer, can
- Dynamic programming, on the other hand, is **always** correct here: it tries both choices

Greedy: the fractional knapsack

- Same data as the 0–1 knapsack, but objects can now be **cut**: any fraction of an object can be taken, for the same fraction of its value
- **Greedy rule**: sort by decreasing value per unit of weight, then take as much as possible of each object in turn
- On the example: all of o_1 (10), all of o_2 (20), then two thirds of o_3 (20 of 30): value 60 + 110 + 100 = **270** > 260
- Here the greedy choice property **holds**: filling the bag with the most valuable material available is part of an optimal solution. Cost: $\Theta(n \log n)$, dominated by the sort (we will see why in the lecture on sorting)
- **Same data, one word changed in the problem, and the right technique changes**

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Divide and conquer



Divide and conquer

- From the Latin maxim *divide et impera*, “divide and rule”: the Roman policy of splitting enemies to defeat them one at a time. Three steps, applied recursively:
  1. **divide** the instance into smaller instances of the same problem
  2. **solve** each of them, recursively
  3. **combine** their solutions into a solution of the whole
- One condition: **optimal substructure**. Overlapping subproblems are **not** wanted here – the subproblems are independent, and each is solved once
- Typical gain: cutting the instance in **halves** makes the depth $\log_2 n$ instead of n
- The question of this section: given the shape of the recursion, **what is the cost**?

Example: binary search

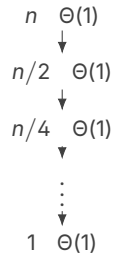
Input a **sorted** array T of n elements, a key x

Output a position of x in T, or "absent"

Lecture 3, seen again as divide and conquer:

- **divide**: compare x with the middle element $T[m]$, $m = \lfloor (lo + hi)/2 \rfloor$
- **solve**: recurse on the left half if $T[m] > x$, on the right half otherwise
- **combine**: nothing to do

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1), \qquad T(1) = \Theta(1)$$



One subproblem per level, $\Theta(1)$ of work each, and $\log_2 n$ levels, since the size is halved until it reaches 1: $\boldsymbol{T(n) = \Theta(\log n)}$

Example: multiplying matrices

For $A, B \in \mathbb{R}^{n \times n}$, compute $C = A \times B$, where $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

```
1: for i ← 1 to n do
2:     for j ← 1 to n do
3:         c_ij ← 0
4:         for k ← 1 to n do
5:             c_ij ← c_ij + a_ik b_kj
6:         end for
7:     end for
8: end for
```

- Three nested loops: $\Theta(n^3)$ multiplications
- Nothing can do better than $\Omega(n^2)$: the n^2 entries of C must at least be written
- Can divide and conquer close the gap?

Matrices by blocks

Assume n is a power of 2 and cut each matrix into four blocks of size $n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Block multiplication behaves like the scalar one:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Eight products of matrices of size $n/2$, plus four additions costing $\Theta(n^2)$:

$$T(n) = 8\,T\left(\frac{n}{2}\right) + \Theta(n^2), \qquad T(1) = \Theta(1)$$

Strassen's algorithm

Seven products instead of eight [Strassen, 1969]:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}), \quad M_2 = (A_{21} + A_{22})B_{11},$$
$$M_3 = A_{11}(B_{12} - B_{22}), \quad M_4 = A_{22}(B_{21} - B_{11}),$$
$$M_5 = (A_{11} + A_{12})B_{22}, \quad M_6 = (A_{21} - A_{11})(B_{11} + B_{12}),$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7, \quad C_{12} = M_3 + M_5,$$
$$C_{21} = M_2 + M_4, \quad C_{22} = M_1 - M_2 + M_3 + M_6$$

More additions, but additions are cheap:

$$T(n) = 7\,T\left(\frac{n}{2}\right) + \Theta(n^2), \qquad T(1) = \Theta(1)$$

Strassen: checking C_{11}

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\qquad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\qquad + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Strassen: checking C_{11}

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) \\
&\qquad - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + A_{22}B_{22} \\
&\qquad + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} \\
&\qquad + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}} \\
&= A_{11}B_{11} + A_{12}B_{21}
\end{aligned}
$$

which is what block multiplication asks for. The other three blocks check out the same way

One shape, many algorithms

- The three examples gave recurrences of the same shape:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n), \qquad T(1) = \Theta(1)$$

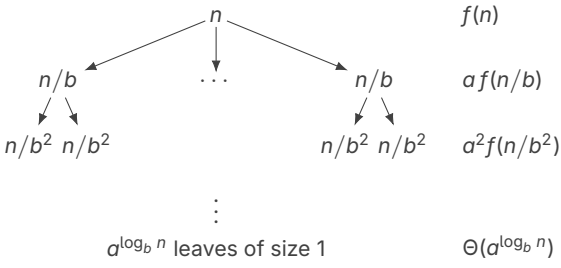
  with $a \geqslant 1$ the number of subproblems, $b > 1$ the factor by which the size shrinks, and f the cost of dividing and combining
- Binary search: $a = 1$, $b = 2$, $f(n) = \Theta(1)$. Blocks: $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$. Strassen: $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$
- One theorem settles all of them at once

UNIVERSITÉ PARIS SCIENCES & LETTRES

Master theorem



Unfolding the recursion



With $n = b^t$, summing the levels:

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) + \Theta\left(a^{\log_b n}\right)$$

- level j holds a^j subproblems of size n/b^j
- there are $\log_b n$ levels
- we write $\boldsymbol{c = \log_b a}$

Whether the root, the leaves, or all the levels dominate is exactly what the theorem answers

A logarithm identity

Proposition

For all $x, y > 0$ and $b > 1$: $x^{\log_b y} = y^{\log_b x}$

Proof.

$$x^{\log_b y} = e^{\ln x \frac{\ln y}{\ln b}} = e^{\ln y \frac{\ln x}{\ln b}} = y^{\log_b x}$$

□

A logarithm identity

Proposition

For all $x, y > 0$ and $b > 1$: $x^{\log_b y} = y^{\log_b x}$

Proof.

$$x^{\log_b y} = e^{\ln x \frac{\ln y}{\ln b}} = e^{\ln y \frac{\ln x}{\ln b}} = y^{\log_b x}$$

□

What it is for: the number of leaves of the tree, $a^{\log_b n}$, is a **power of *n***:

$$a^{\log_b n} = n^{\log_b a} = \boldsymbol{n^c}$$

So the leaves contribute $\Theta(n^c)$, and the comparison between $f(n)$ and n^c decides everything.

The master theorem

Theorem

Let $T(n) = a\,T(n/b) + f(n)$ with $a \geqslant 1$, $b > 1$, $T(1) = \Theta(1)$, and let $c = \log_b a$.

1. *If $f(n) = O(n^{c'})$ for some $c' < c$, then $\boldsymbol{T(n) = \Theta(n^c)}$* *(the leaves dominate)*
2. *If $f(n) = \Theta(n^c)$, then $\boldsymbol{T(n) = \Theta(n^c \log n)}$* *(all levels cost the same)*
3. *If $f(n) = \Omega(n^{c'})$ for some $c' > c$, and $a\,f(n/b) \leqslant \alpha\,f(n)$ for some $\alpha < 1$ and all large enough n, then $\boldsymbol{T(n) = \Theta(f(n))}$* *(the root dominates)*

Three cases, one question: is the work at the **top** of the tree, at the **bottom**, or spread evenly?

Using the master theorem

Binary search $a = 1$, $b = 2$, $f(n) = \Theta(1)$, $c = \log_2 1 = 0$. Case 2: $\boldsymbol{T(n) = \Theta(\log n)}$

Blocked matrices $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, $c = \log_2 8 = 3$. Case 1: $\boldsymbol{T(n) = \Theta(n^3)}$ – no better than the three loops

Strassen $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, $c = \log_2 7 \approx 2.807$. Case 1: $\boldsymbol{T(n) = \Theta(n^{\log_2 7})}$

One multiplication fewer at each level, and an exponent of 2.807 instead of 3: for $n = 4096$, a factor of about 5

Notes on the master theorem

- It does **not cover every case**: for instance $f(n) = \Theta(n^c \log n)$ falls between cases 2 and 3 of this statement (Cormen et al. [2022] extend case 2 to cover it), and case 3 needs its extra condition (which is satisfied by every f we meet in practice)
- Generalizations exist; the **Akra–Bazzi method** handles subproblems of **unequal** sizes, as in $T(n) = T(n/3) + T(2n/3) + \Theta(n)$
- The proof sketches that follow assume n is a power of b and ignore the rounding of n/b to an integer; Cormen et al. [2022], chapter 4, proves the theorem in full
- **In practice**: identify a, b, f, compute $c = \log_b a$, compare $f(n)$ with n^c, and read off the answer

Proof sketch, case 1: $f(n) = O(n^{c'}), c' < c$

Write $n = b^t$ and split the sum of the levels:

$$T(n) = \underbrace{\sum_{j=0}^{t-1} a^j f\left(\frac{n}{b^j}\right)}_{g(n)} + \Theta(n^c)$$

Since $a = b^c$:

$$g(n) = \sum_{j=0}^{t-1} (b^c)^j \cdot O\left(\left(\frac{n}{b^j}\right)^{c'}\right)$$

Proof sketch, case 1: $f(n) = O(n^{c'}), c' < c$

Write $n = b^t$ and split the sum of the levels:

$$T(n) = \underbrace{\sum_{j=0}^{t-1} a^j f\left(\frac{n}{b^j}\right)}_{g(n)} + \Theta(n^c)$$

Since $a = b^c$:

$$g(n) = \sum_{j=0}^{t-1} (b^c)^j \cdot O\left(\left(\frac{n}{b^j}\right)^{c'}\right) = O\left(n^{c'} \sum_{j=0}^{t-1} b^{(c-c')j}\right)$$

Proof sketch, case 1: $f(n) = O(n^{c'}), c' < c$

Write $n = b^t$ and split the sum of the levels:

$$T(n) = \underbrace{\sum_{j=0}^{t-1} a^j f\left(\frac{n}{b^j}\right)}_{g(n)} + \Theta(n^c)$$

Since $a = b^c$:

$$g(n) = \sum_{j=0}^{t-1} (b^c)^j \cdot O\left(\left(\frac{n}{b^j}\right)^{c'}\right) = O\left(n^{c'} \sum_{j=0}^{t-1} b^{(c-c')j}\right) = O\left(n^{c'} \times \frac{b^{(c-c')t} - 1}{b^{c-c'} - 1}\right)$$

Proof sketch, case 1: $f(n) = O(n^{c'}), c' < c$

Write $n = b^t$ and split the sum of the levels:

$$T(n) = \underbrace{\sum_{j=0}^{t-1} a^j f\left(\frac{n}{b^j}\right)}_{g(n)} + \Theta(n^c)$$

Since $a = b^c$:

$$g(n) = \sum_{j=0}^{t-1} (b^c)^j \cdot O\left(\left(\frac{n}{b^j}\right)^{c'}\right) = O\left(n^{c'} \sum_{j=0}^{t-1} b^{(c-c')j}\right) = O\left(n^{c'} \times \frac{b^{(c-c')t} - 1}{b^{c-c'} - 1}\right)$$

$$= O\left(n^{c'} \times \left(n^{c-c'} - 1\right)\right) = \boldsymbol{O(n^c)}$$

The leaves alone already cost $\Theta(n^c)$, so $\boldsymbol{T(n) = \Theta(n^c)}$.

Proof sketch, case 2: $f(n) = \Theta(n^c)$

Each level costs the same:

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^c\right) = \Theta\left(n^c \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^j\right) = \Theta(n^c \log n)$$

since $a = b^c$ makes the ratio 1 and the sum has $\log_b n$ terms.

Proof sketch, case 2: $f(n) = \Theta(n^c)$

Each level costs the same:

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^c\right) = \Theta\left(n^c \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^j\right) = \Theta(n^c \log n)$$

since $a = b^c$ makes the ratio 1 and the sum has $\log_b n$ terms.

The leaves contribute only $\Theta(n^c)$, so $T(n) = \Theta(n^c \log n)$.

Proof sketch, case 3: $f(n) = \Omega(n^{c'})$, $c' > c$

The extra condition, $a f(n/b) \leqslant \alpha f(n)$ with $\alpha < 1$, applied to n/b^{j-1} instead of n, then to n/b^{j-2}, and so on down to n:

$$a^j f\left(\frac{n}{b^j}\right) \leqslant \alpha\, a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \leqslant \alpha^2 a^{j-2} f\left(\frac{n}{b^{j-2}}\right) \leqslant \ldots \leqslant \alpha^j f(n)$$

Proof sketch, case 3: $f(n) = \Omega(n^{c'})$, $c' > c$

The extra condition, $a f(n/b) \leqslant \alpha f(n)$ with $\alpha < 1$, applied to n/b^{j-1} instead of n, then to n/b^{j-2}, and so on down to n:

$$a^j f\left(\frac{n}{b^j}\right) \leqslant \alpha\, a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \leqslant \alpha^2 a^{j-2} f\left(\frac{n}{b^{j-2}}\right) \leqslant \ldots \leqslant \alpha^j f(n)$$

The levels **decrease geometrically** from the root:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \leqslant \sum_{j=0}^{\infty} \alpha^j f(n) = \frac{f(n)}{1-\alpha} = O\big(f(n)\big)$$

Proof sketch, case 3: $f(n) = \Omega(n^{c'})$, $c' > c$

The extra condition, $a f(n/b) \leqslant \alpha f(n)$ with $\alpha < 1$, applied to n/b^{j-1} instead of n, then to n/b^{j-2}, and so on down to n:

$$a^j f\left(\frac{n}{b^j}\right) \leqslant \alpha\, a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \leqslant \alpha^2 a^{j-2} f\left(\frac{n}{b^{j-2}}\right) \leqslant \ldots \leqslant \alpha^j f(n)$$

The levels **decrease geometrically** from the root:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \leqslant \sum_{j=0}^{\infty} \alpha^j f(n) = \frac{f(n)}{1-\alpha} = O\big(f(n)\big)$$

Conversely $g(n) \geqslant f(n)$, the term $j = 0$ of the sum; and the leaves cost $\Theta(n^c) = O(f(n))$ since $c' > c$. Hence $\boldsymbol{T(n) = \Theta(f(n))}$.

Historical note: the exponent of matrix multiplication

What is the smallest ω such that multiplying $n \times n$ matrices costs $O(n^\omega)$?

Year	Algorithm	Exponent
1812	the three loops (Binet)	$\omega \leqslant 3$
1969	Strassen [Strassen, 1969]	$\omega < 2.808$
1990	Coppersmith–Winograd	$\omega < 2.376$
2025	Alman, Duan, Vassilevska Williams, Xu, Xu, Zhou [Alman et al., 2025]	$\omega < 2.371339$
2026	Dupont et al., with AlphaEvolve, an AI system [Dupont et al., 2026]	$\omega < 2.371177$
2026	an OpenAI model; human proof by Shpilka [OpenAI, 2026, Shpilka, 2026]	$\omega \leqslant 9/4 = 2.25$

- The lower bound is still only $\Omega(n^2)$: the truth is unknown, and many believe $\omega = 2$
- Beyond Strassen, the constant factors are so large that no practical size benefits. Libraries use the three loops, tuned for the memory hierarchy, and sometimes Strassen on very large matrices

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Coda



Which technique when?

Question to ask	If yes	What decides the cost
Can I enumerate everything?	brute force, to test the rest	number of candidates
Are the subproblems independent?	divide and conquer	master theorem
Do they repeat?	memoization, then a table if needed	size of the table
Is the local choice provably safe?	greedy	cost of the choices

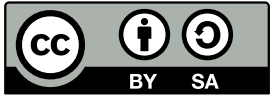
- **The method**: write the **recurrence** first, on paper. The program follows from it, and so does its complexity
- Then choose **how** to evaluate it: recursion for clarity, memoization for speed with a few lines more, a table when the depth or the memory matters
- Greedy algorithms are the exception: cheapest to run, hardest to justify. Compare them with an exact solution on small inputs

Wednesday's tutorial and references

- Wednesday's tutorial: making change and the edit distance in the three styles (recursive, memoized, table), **timing** them to see the difference; reconstructing the coins and the alignment; a greedy algorithm and a counter-example to it; using the master theorem on the recurrences of a few short programs
- **This Friday**: the mid-semester exam in the afternoon, covering lectures 1 to 6 (not this one), followed by a short lecture on exceptional situations
- References: Cormen et al. [2022], chapters 4 (divide and conquer and the master theorem), 14 (dynamic programming), and 15 (greedy algorithms, with the knapsack)

Licensing

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



Bibliography I

Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2005–2039, 2025. doi: 10.1137/1.9781611978322.63.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5. URL https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

Emilien Dupont, Marvin Eisenberger, Borislav Kozlovskii, Abbas Mehrabian, Francisco J. R. Ruiz, Abigail See, Renfei Zhou, Josh Alman, Virginia Vassilevska Williams, and Matej Balog. Improving the matrix multiplication exponent with modern optimization and AlphaEvolve. arXiv:2608.16884, 2026.

Bibliography II

OpenAI. An upper bound of 9/4 for the matrix multiplication exponent. OpenAI Math Release preprint, 2 October 2026, 2026. https://github.com/openai/math.

Amir Shpilka. Exposition of OpenAI's matrix multiplication in exponent 9/4 (and an explicit algorithm). Technical Report TR26-281, Electronic Colloquium on Computational Complexity, 2026. URL https://eccc.weizmann.ac.il/report/2026/281/.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4): 354–356, 1969. doi: 10.1007/BF02165411.