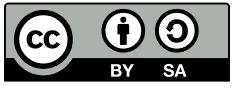
Common Algorithmic Techniques

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS



10 November 2025

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:
  - Brute-force: Try all possible solutions and select the best one

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:
  - Brute-force: Try all possible solutions and select the best one
  - Divide and conquer: Split the problem into smaller subproblems, solve each recursively, and combine results

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:
  - Brute-force: Try all possible solutions and select the best one
  - Divide and conquer: Split the problem into smaller subproblems, solve each recursively, and combine results
  - Dynamic programming: Solve problems with overlapping subproblems by storing intermediate results

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:
  - Brute-force: Try all possible solutions and select the best one
  - Divide and conquer: Split the problem into smaller subproblems, solve each recursively, and combine results
  - Dynamic programming: Solve problems with overlapping subproblems by storing intermediate results
  - Greedy algorithms: Make a sequence of locally optimal choices, hoping to find a global optimum

Common Algorithmic Techniques

- When designing an algorithm to solve a problem, it is helpful to know high-level algorithmic techniques
- These techniques provide systematic ways to approach problem-solving, analyze complexity, and ensure correctness
- Common techniques include:
  - Brute-force: Try all possible solutions and select the best one
  - Divide and conquer: Split the problem into smaller subproblems, solve each recursively, and combine results
  - Dynamic programming: Solve problems with overlapping subproblems by storing intermediate results
  - Greedy algorithms: Make a sequence of locally optimal choices, hoping to find a global optimum
- Understanding which technique fits a given problem is crucial for designing efficient and correct algorithms.

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\left\{ \begin{array}{l} 1+1+1+1+1+1+1 \\ \\ \\ \\ \\ \end{array} \right.$$

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ \\ \\ \\ \end{cases}$$

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \end{cases}$$

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \end{cases}$$

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \\ 1+2+2+2 \end{cases}$$

Making Change

- We fix $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n \in \mathbb{N}^*$ as the value of n types of coins
- For example, in the Euro zone, value of coins in Euro 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

- How many ways are there to make change for a sum $S \in \mathbb{N}$?
- For example, if $S = 7$, there are 6 different ways:

$$\begin{cases} 1+1+1+1+1+1+1 \\ 1+1+1+1+1+2 \\ 1+1+1+2+2 \\ 1+1+5 \\ 1+2+2+2 \\ 2+5 \end{cases}$$

Brute Force Solution

- Try all possible combinations and check which ones sum to S

Brute Force Solution

- Try all possible combinations and check which ones sum to S
- For example, for $S = 7$, we can try combinations with $\leqslant 7$ coins of 1, $\leqslant 3$ coins of 2, and $\leqslant 1$ coin of 5

Brute Force Solution

- Try all possible combinations and check which ones sum to S
- For example, for $S = 7$, we can try combinations with $\leqslant 7$ coins of 1, $\leqslant 3$ coins of 2, and $\leqslant 1$ coin of 5
- 8 possibilities for coins of 1, 4 for coins of 2, 2 for coins of 5, so $8 \times 4 \times 2 = 64$ possibilities to test

Brute Force Solution

- Try all possible combinations and check which ones sum to S
- For example, for $S = 7$, we can try combinations with $\leqslant 7$ coins of 1, $\leqslant 3$ coins of 2, and $\leqslant 1$ coin of 5
- 8 possibilities for coins of 1, 4 for coins of 2, 2 for coins of 5, so $8 \times 4 \times 2 = 64$ possibilities to test
- In the general case, the algorithm runs in $\Theta\left(\left\lceil \frac{S}{p_1} \right\rceil \times \cdots \times \left\lceil \frac{S}{p_n} \right\rceil\right) = O(S^n)$.

Brute Force Solution

- Try all possible combinations and check which ones sum to S
- For example, for $S = 7$, we can try combinations with $\leqslant 7$ coins of 1, $\leqslant 3$ coins of 2, and $\leqslant 1$ coin of 5
- 8 possibilities for coins of 1, 4 for coins of 2, 2 for coins of 5, so $8 \times 4 \times 2 = 64$ possibilities to test
- In the general case, the algorithm runs in $\Theta\left(\left\lceil \frac{S}{p_1} \right\rceil \times \cdots \times \left\lceil \frac{S}{p_n} \right\rceil\right) = O(S^n)$.
- A brute-force approach is always possible, but often very inefficient

Brute Force Solution

- Try all possible combinations and check which ones sum to S
- For example, for $S = 7$, we can try combinations with $\leqslant 7$ coins of 1, $\leqslant 3$ coins of 2, and $\leqslant 1$ coin of 5
- 8 possibilities for coins of 1, 4 for coins of 2, 2 for coins of 5, so $8 \times 4 \times 2 = 64$ possibilities to test
- In the general case, the algorithm runs in $\Theta\left(\left\lceil \frac{S}{p_1} \right\rceil \times \cdots \times \left\lceil \frac{S}{p_n} \right\rceil\right) = O(S^n)$.
- A brute-force approach is always possible, but often very inefficient
- Can we do better?

Recurrence Formula

- For $S \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $N(S, k)$ be the number of combinations to make change for the sum S using the first k coins.

Recurrence Formula

- For $S \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $N(S, k)$ be the number of combinations to make change for the sum S using the first k coins.
- We have very simple (base) cases:

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

Recurrence Formula

- For $S \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $N(S, k)$ be the number of combinations to make change for the sum S using the first k coins.
- We have very simple (base) cases:

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

- We can solve a complex case using simpler ones:

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

(Either we don't use coin p_k, or we use it at least once.)

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

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

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

	k			
S	**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			
5	0			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

	k			
S	**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			
6	0			
7	0			

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

	k			
S	**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			
7	0			

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

	k			
S	**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			

p_1	p_2	p_3
1	2	5

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

Application to the Case $S = 7$

	k			
S	**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

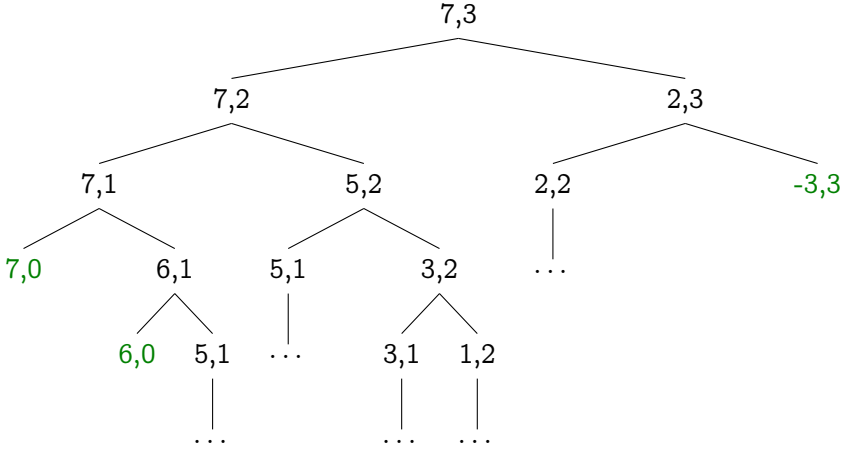
p_1	p_2	p_3
1	2	5

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

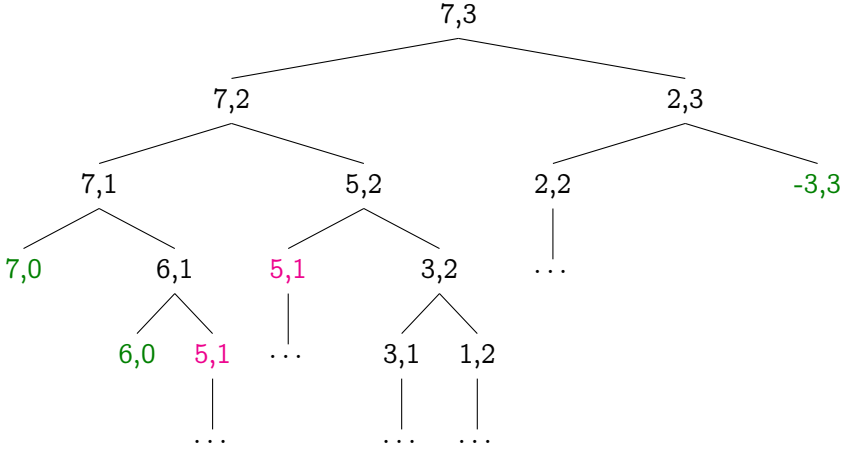
Recursive Function Solution

function $\mathrm{nb\_coins}(S, k)$
if $S = 0$ **then**
    return 1;
else
    if $S < 0$ ***or*** $k = 0$ **then**
        return 0;
    else
        return $\mathrm{nb\_coins}(S, k-1) + \mathrm{nb\_coins}(S - p_k, k)$;

Recursive Calls for $S = 7$, $p = \{1, 2, 5\}$



Recursive Calls for $S = 7$, $p = \{1, 2, 5\}$



Recursive Function: Complexity

- To simplify the analysis, consider the (worst but possible) case where $p = \{1, \ldots, 1\}$.

Recursive Function: Complexity

- To simplify the analysis, consider the (worst but possible) case where $p = \{1, \ldots, 1\}$.
- Let $N = S + n$, then:

$$T(N) = \begin{cases} O(1) & \text{if } S \leqslant 0 \text{ or } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{otherwise} \end{cases}$$

Recursive Function: Complexity

- To simplify the analysis, consider the (worst but possible) case where $p = \{1, \ldots, 1\}$.
- Let $N = S + n$, then:

$$T(N) = \begin{cases} O(1) & \text{if } S \leqslant 0 \text{ or } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{otherwise} \end{cases}$$

- Note that $T(N) \approx 2 \times T(N-1)$, hence $T(N) = \Theta\left(2^N\right) = \Theta\left(2^{S+n}\right)$

Recursive Function: Complexity

- To simplify the analysis, consider the (worst but possible) case where $p = \{1, \ldots, 1\}$.
- Let $N = S + n$, then:

$$T(N) = \begin{cases} O(1) & \text{if } S \leqslant 0 \text{ or } n = 0 \\ T(N-1) + T(N-1) + O(1) & \text{otherwise} \end{cases}$$

- Note that $T(N) \approx 2 \times T(N-1)$, hence $T(N) = \Theta\left(2^N\right) = \Theta\left(2^{S+n}\right)$
- This is worse than brute force which was $O(S^n)$!

Solution with Memoization

function nb_coins(S, k)
if $M(S,k)$ *is defined* **then**
  return $M(S,k)$;
if $S = 0$ **then**
  $M(S,k) \leftarrow 1$;
else
  if $S < 0$ ***or*** $k = 0$ **then**
    $M(S,k) \leftarrow 0$;
  else
    $M(S,k) \leftarrow \text{nb\_coins}(S, k-1) + \text{nb\_coins}(S - p_k, k)$;
return $M(S,k)$;

Memoization: Complexity

- Each call is $O(1)$, except at most once per entry in the table M, of size $(S + p_n) \times (n + 1)$
- Therefore, the total complexity is $O(S \times n)$: much more efficient!

Solution by Dynamic Programming

function nb_coins(S, n)
$T \leftarrow$ create_table$(S+1, n+1)$;
for $i \leftarrow 1$ **to** S **do**
  $T(i, 0) \leftarrow 0$;
for $i \leftarrow 0$ **to** n **do**
  $T(0, i) \leftarrow 1$;
for $i \leftarrow 1$ **to** S **do**
  for $j \leftarrow 1$ **to** n **do**
    if $p_j \leqslant i$ **then**
      $T(i, j) \leftarrow T(i, j-1) + T(i - p_j, j)$;
    else
      $T(i, j) \leftarrow T(i, j-1)$;
return $T(S, n)$;

Solution by Dynamic Programming

function nb_coins(S, n)
$T \leftarrow$ create_table$(S+1, n+1)$;
for $i \leftarrow 1$ **to** S **do**
  $T(i, 0) \leftarrow 0$;
for $i \leftarrow 0$ **to** n **do**
  $T(0, i) \leftarrow 1$;
for $i \leftarrow 1$ **to** S **do**
  for $j \leftarrow 1$ **to** n **do**
    if $p_j \leqslant i$ **then**
      $T(i, j) \leftarrow T(i, j-1) + T(i - p_j, j)$;
    else
      $T(i, j) \leftarrow T(i, j-1)$;
return $T(S, n)$;

Complexity:
$\Theta(S \times n)$

Variant: Minimum Number of Coins

- What if we want the minimum number of coins, instead of the number of combinations?

Variant: Minimum Number of Coins

- What if we want the minimum number of coins, instead of the number of combinations?
- We establish a new recurrence formula:

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

Variant: Minimum Number of Coins

- What if we want the minimum number of coins, instead of the number of combinations?
- We establish a new recurrence formula:

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

- We can use this recurrence to write recursive, memoized, or dynamic programming solutions...

Outline

When to Use Dynamic Programming

- When can and should we use dynamic programming (or memoization)?

When to Use Dynamic Programming

- When can and should we use dynamic programming (or memoization)?
- We need two conditions:

When to Use Dynamic Programming

- When can and should we use dynamic programming (or memoization)?
- We need two conditions:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

When to Use Dynamic Programming

- When can and should we use dynamic programming (or memoization)?
- We need two conditions:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

  Overlapping subproblems. The same subproblems occur many times.

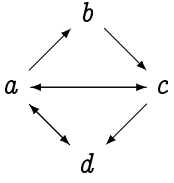
When to Use Dynamic Programming

- When can and should we use dynamic programming (or memoization)?
- We need two conditions:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

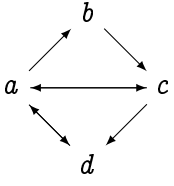
  Overlapping subproblems. The same subproblems occur many times.
- If we have only optimal substructures but not overlapping subproblems, then simple recursion is sufficient (see section on *Divide and Conquer*)

Example: Paths in Graphs



- Shortest Path:
  - Path from one node to another with the smallest number of edges
  - Optimal substructures: The shortest path from x to y passing through z is composed of the shortest path from x to z combined with the shortest path from z to y
  - Overlapping subproblems: The shortest path from x to y is useful for computing many other shortest paths
  - Dynamic programming is appropriate!

Example: Paths in Graphs



- Longest Path (without cycles):
  - Path from one node to another with the largest number of edges, without visiting the same node twice
  - Non-optimal substructures: The longest path from x to y through z is not necessarily composed of the longest path from x to z combined with the longest path from z to y.
  - Example: from b to d passing through a
  - Dynamic programming does not work here!

Example: 0–1 Knapsack (1/3)

- We are given n objects $o_1, \ldots, o_n$, each object o_i having a weight w_i and a value v_i
- We have a backpack of maximum capacity (by weight) W
- How can we maximize the total value of objects in the backpack while keeping the total weight $\leqslant W$?
- We seek $S \subseteq \{1, \ldots, n\}$ such that:

$$\sum_{i \in S} w_i \leqslant W \text{ and } \sum_{i \in S} v_i \text{ is maximal}$$

Example: 0–1 Knapsack (2/3)

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6	5.5	5

$W = 50$

- Greedy method: take the objects in decreasing order of value/weight ratio, as long as capacity allows.
- Does not work! It leads to taking o_1 and o_2, with a total value of 170 vs. 260 for o_2 and o_3.
- The best local choice (o_1) is not the best global choice.

Example: 0–1 Knapsack (3/3)

- Dynamic programming works (optimal substructures, overlapping subproblems)

Example: 0–1 Knapsack (3/3)

- Dynamic programming works (optimal substructures, overlapping subproblems)
- Let $V(W, k)$ be the maximum value attainable with capacity W and the first k objects

Example: 0–1 Knapsack (3/3)

- Dynamic programming works (optimal substructures, overlapping subproblems)
- Let $V(W, k)$ be the maximum value attainable with capacity W and the first k objects
- Then:

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

Example: 0–1 Knapsack (3/3)

- Dynamic programming works (optimal substructures, overlapping subproblems)
- Let $V(W, k)$ be the maximum value attainable with capacity W and the first k objects
- Then:

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

- Using dynamic programming, complexity is $\Theta(W \times n)$.

Outline

When Can We Use Recursion?

A problem P can be solved recursively when:

- It can be parameterized by one or more integers $P(n_1 \dots n_k)$
- The solution to $P(n_1 \dots n_k)$ can be obtained from the solutions of $P(n_1^{(1)} \dots n_k^{(1)}) \dots P(n_1^{(\ell)} \dots n_k^{(\ell)})$ for some $\ell \geqslant 1$, with for all $1 \leqslant i \leqslant \ell$, $n_j^{(i)} \leqslant n_j$ for $1 \leqslant j \leqslant k$, and at least one inequality strict ($\forall i, \exists j, n_j^{(i)} < n_j$): recursive case
- $P(0 \dots 0)$ (or $P(n_1 \dots n_k)$ for small n_i, depending on the case) is easy to compute: base case

Example 1: Factorial

- $P(n) = n!$
- $k = 1$, $\ell = 1$
- $P(n) = n \times P(n-1)$ for $n \geqslant 1$
- $P(0) = 1$

Example 2: Fibonacci

- $P(n)$ is the nth Fibonacci number
- $k = 1$, $\ell = 2$
- $P(n) = P(n-1) + P(n-2)$ for $n \geqslant 1$
- $P(0) = 1$, $P(1) = 1$

Example 3: Levenshtein Edit Distance

$d(s, s')$ is the edit distance (minimum number of character additions, deletions, or substitutions) to transform string s into string s'.

- $P(n_1, n_2)$ is the edit distance between the prefix of length n_1 of s and the prefix of length n_2 of s'
- $k = 2$, $\ell = 3$
- $P(n_1, n_2) = \min\left(P(n_1 - 1, n_2) + 1, P(n_1, n_2 - 1) + 1, P(n_1 - 1, n_2 - 1) + \mathbb{I}_{s_{n_1} \neq s'_{n_2}}\right)$ for $n_1 \geqslant 1$, $n_2 \geqslant 1$
- $P(n_1, 0) = n_1$ for $n_1 \geqslant 0$; $P(0, n_2) = n_2$ for $n_2 \geqslant 0$

$\mathbb{I}_b$ is 1 if b is true, 0 otherwise.

Pseudo-code

- Recursive function

function $P(n_1, \ldots, n_k)$
if *base case* **then**
    ...;
    return ...;
else
    // recursive case
    ...;
    // ℓ calls to P
    return ...;

- Recursion is tail recursion if $\ell = 1$ and the call to P is the last instruction of the recursive case (**return** $P(n'_1, \ldots, n'_k)$)
- Remark: in the example above, the **else** may be omitted

Complexity

- Upper bound (sometimes tighter, e.g., if the recursive case is on $\frac{n}{2}$ instead of $n-1$, see section on *Divide and Conquer*):
  - If $\ell = 1$: $O(n_1 + \cdots + n_k)$: often acceptable
  - If $\ell > 1$: $O(\ell^{n_1+\cdots+n_k})$: often unreasonable
- Beware of memory used on the stack, $O(\sum_{i=1}^{k} n_i)$ (often, only a few hundred kilobytes or a few megabytes are available on the stack) and language limits (in Python, `sys.getrecursionlimit()` gives the recursion depth limit, typically 1000; in C/C++, this depends on the operating system)
- When recursion is tail-recursive, the compiler can rewrite it as an iteration, removing stack management issues, in most programming languages (including C/C++), but not in Python! (decision by Guido van Rossum)

Outline

Idea (1/3)

Cache: a global object (or static class property) that remembers the results of function P between calls.

Idea (2/3)

function $P(n_1, \ldots, n_k)$
if $M(n_1, \ldots, n_k)$ *is defined* **then**
    return $M(n_1, \ldots, n_k)$;
if *base case* **then**
    $\ldots$;
    $r \leftarrow \ldots$;
else
    // recursive case
    $\ldots$;
    // ℓ calls to P
    $r \leftarrow \ldots$;
$M(n_1, \ldots, n_k) \leftarrow r$;
return r;

Idea (3/3)

- Tail recursion is no longer possible!
- Data structures:

  Python. `array.array`, Python `list`, or dictionary, depending on what needs to be stored

  C. C array

  C++. `std::array`, `std::vector`, or `std::unordered_map`, depending on what needs to be stored
- Use an `array.array`, Python `list`, `std::array`, or `std::vector` when the parameters are integers with contiguous values, otherwise use a hash table (Python dictionary or `std::unordered_map`)

Complexity

- $O(n_1 \times \cdots \times n_k \times \ell)$ operations
- $O(n_1 \times \cdots \times n_k)$ heap memory (precise space depends on the data structure)
- $O(n_1 + \cdots + n_k)$ stack memory
- No redundant computations!

Outline

Idea

- We construct a multi-dimensional table $n_1 \times \cdots \times n_k$, filled iteratively, cell by cell: iterative computation

function $P(n_1, \ldots, n_k)$
$M \leftarrow \text{table}(n_1, \ldots, n_k)$;
`// fill` M `with base cases`
for $i_1 \leftarrow 1$ **to** n_1 **do**
  $\ldots$;
  for $i_k \leftarrow 1$ **to** n_k **do**
    $\ldots$;
    $M(i_1, \ldots, i_k) \leftarrow \ldots$;
    `// use already computed values`
return $M(n_1 \ldots n_k)$;

Complexity

- $\Theta(n_1 \times \cdots \times n_k \times \ell)$ operations
- $\Theta(n_1 \times \cdots \times n_k)$ heap memory
- $O(1)$ stack memory
- Sometimes too many operations!

Memoization vs Dynamic Programming

- The two techniques are essentially equivalent
- Advantage of memoization: only the necessary computations are performed (we don't fill the entire table)
- Disadvantage of memoization: recursive calls have a small overhead (compared to imperative style) and, more importantly, use the call stack (which is limited in size)
- We can simulate dynamic programming behavior using memoization
- For more complex cases, we can simulate recursive calls with a manually maintained stack on the heap

Reconstructing the Solution

- It is often useful to reconstruct a solution, not just to find an optimal value: best set of coins for change, shortest path, best subset of items for the knapsack
- How? Simply remember, when taking a min (or max), which choice led to that min (or max) – record the arg min along with the min!
- In recursive programming: return the optimal substructure along with the optimal value
- With memoization: store in memory the optimal substructure along with the optimal value
- In dynamic programming: associate with each table cell the optimal substructure along with the optimal value

Outline

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items
- Recall the greedy method:

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items
- Recall the greedy method:
  - Sort the items by their $\frac{v_i}{w_i}$ in decreasing order

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items
- Recall the greedy method:
  - Sort the items by their $\frac{v_i}{w_i}$ in decreasing order
  - Take the items in this order, taking as much as possible of each item

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items
- Recall the greedy method:
  - Sort the items by their $\frac{v_i}{w_i}$ in decreasing order
  - Take the items in this order, taking as much as possible of each item
- In the fractional case, this is optimal!

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

Fractional Knapsack

- Variant of the knapsack problem: we can take fractions of items
- Recall the greedy method:
  - Sort the items by their $\frac{v_i}{w_i}$ in decreasing order
  - Take the items in this order, taking as much as possible of each item
- In the fractional case, this is optimal!
- Gives a complexity of $\Theta(n \log n)$, see upcoming course on *Sorting Algorithms*

	o_1	o_2	o_3
weight	10	20	30
value	60	110	150
ratio	6.0	5.5	5.0

$W = 50$

When to Apply Greedy Algorithms

- When can one use, and is it interesting to use, a greedy algorithm?

When to Apply Greedy Algorithms

- When can one use, and is it interesting to use, a greedy algorithm?
- Two conditions are required:

When to Apply Greedy Algorithms

- When can one use, and is it interesting to use, a greedy algorithm?
- Two conditions are required:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

When to Apply Greedy Algorithms

- When can one use, and is it interesting to use, a greedy algorithm?
- Two conditions are required:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

  Greedy choice property. The locally optimal choice is a globally optimal choice.

Outline

Divide and Conquer (Divide et Impera)

- Three steps:

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems
  - Solve the subproblems independently (recursively)

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems
  - Solve the subproblems independently (recursively)
  - Combine the subproblem solutions

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems
  - Solve the subproblems independently (recursively)
  - Combine the subproblem solutions
- One condition required:

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems
  - Solve the subproblems independently (recursively)
  - Combine the subproblem solutions
- One condition required:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.

Divide and Conquer (Divide et Impera)

- Three steps:
  - Divide into subproblems
  - Solve the subproblems independently (recursively)
  - Combine the subproblem solutions
- One condition required:

  Optimal substructures. The optimal solutions of subproblems are substructures of the optimal solution of the original problem.
- Not necessarily *overlapping subproblems* (differs from dynamic programming)

Example: Binary Search

Input: Sorted array T of n elements; key x.

Task: Decide if $x \in T$ (and optionally return i s.t. $T[i] = x$)

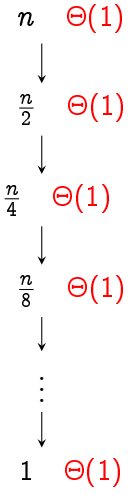
Recursive idea:

- Let $m = \left\lfloor \frac{low + high}{2} \right\rfloor$
- If $T[m] = x$, return m
- If $T[m] > x$, recurse on $[low, m-1]$; else on $[m+1, high]$

Cost: (assuming to simplify that n is a power of 2)

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

Binary Search: Recursion Tree



- Depth: c levels, with $2^c = n \iff c = \log_2 n$.
- Complexity: $T(n) = \Theta(\log n)$

Example: Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}$ (assume n is a power of 2). Compute $C = A \times B$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example: Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}$ (assume n is a power of 2). Compute $C = A \times B$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $c_{ij} \leftarrow 0$;
    for $k \leftarrow 1$ **to** n **do**
      $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$;

Example: Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}$ (assume n is a power of 2). Compute $C = A \times B$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $c_{ij} \leftarrow 0$;
    for $k \leftarrow 1$ **to** n **do**
      $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$;

- Direct algorithm: $\Theta(n^3)$ operations.

Example: Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}$ (assume n is a power of 2). Compute $C = A \times B$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $c_{ij} \leftarrow 0$;
    for $k \leftarrow 1$ **to** n **do**
      $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$;

- Direct algorithm: $\Theta(n^3)$ operations.
- Lower bound: $\Omega(n^2)$ (we must at least compute the n^2 outputs)

Matrix Multiplication: Blocked Divide & Conquer

Partition into quadrants:

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

Then compute:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

This yields the recurrence

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

Strassen's Algorithm

Seven multiplications instead of eight. Define

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

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

Recurrence:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

Strassen's Algorithm — Verifying C_{11}

Strassen's algorithm defines:

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

General Recurrence Formula

- In all three examples we saw, the recurrence formula for the cost was of the form:

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

  with a a positive integer, b a rational $\geqslant 1$ and f a function for which we had a Θ expression
- Can we find a closed form of such recurrences?

Outline

Analyzing Recurrences by Recursion Trees

- Binary search: $T(n) = T(n/2) + \Theta(1)$ produces a tree of depth $\log_2 n$; each level costs $\Theta(1)$. $\Rightarrow T(n) = \Theta(\log n)$

Analyzing Recurrences by Recursion Trees

- Binary search: $T(n) = T(n/2) + \Theta(1)$ produces a tree of depth $\log_2 n$; each level costs $\Theta(1)$. $\Rightarrow T(n) = \Theta(\log n)$
- Blocked MM: $T(n) = 8T(n/2) + \Theta(n^2)$. Level j has 8^j subproblems of size $n/2^j$, each of cost $\Theta((n/2^j)^2)$. There are $\log_2 n$ levels before reaching size 1, plus leaf work $\Theta(1)$ per leaf. Total cost?

Analyzing Recurrences by Recursion Trees

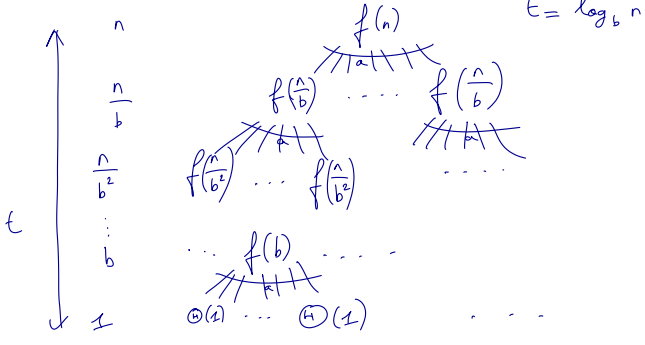
- Binary search: $T(n) = T(n/2) + \Theta(1)$ produces a tree of depth $\log_2 n$; each level costs $\Theta(1)$. $\Rightarrow T(n) = \Theta(\log n)$
- Blocked MM: $T(n) = 8T(n/2) + \Theta(n^2)$. Level j has 8^j subproblems of size $n/2^j$, each of cost $\Theta((n/2^j)^2)$. There are $\log_2 n$ levels before reaching size 1, plus leaf work $\Theta(1)$ per leaf. Total cost?
- Strassen: $T(n) = 7T(n/2) + \Theta(n^2)$. Level j has 7^j subproblems of size $n/2^j$, each of cost $\Theta((n/2^j)^2)$. There are $\log_2 n$ levels before reaching size 1, plus leaf work $\Theta(1)$ per leaf. Total cost?

General Form of Divide-and-Conquer Recurrences

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

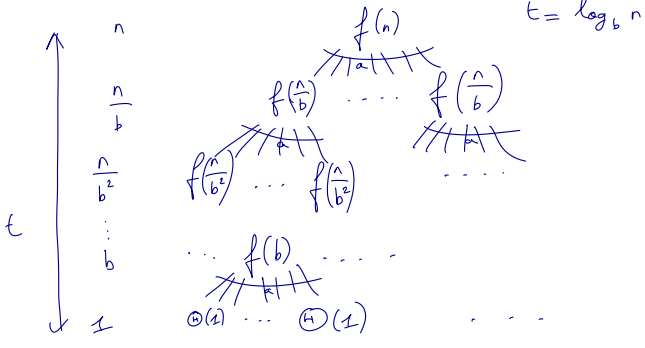
General Form of Divide-and-Conquer Recurrences

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



General Form of Divide-and-Conquer Recurrences

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



Assume $n = b^t$; expanding the recursion tree gives:

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

We pose: $c = \log_b a$.

Logarithm Manipulation

Proposition

For all $x > 0, y > 0, b > 1$, *we have:*

$$x^{\log_b y} = y^{\log_b x}$$

Logarithm Manipulation

Proposition

For all $x > 0, y > 0, b > 1$, *we have:*

$$x^{\log_b y} = y^{\log_b x}$$

Proof.

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

□

Logarithm Manipulation

Proposition

For all $x > 0, y > 0, b > 1$, we have:

$$x^{\log_b y} = y^{\log_b x}$$

Proof.

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

□

Application

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

Master Theorem for Divide-and-Conquer Problems

Theorem

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

Master Theorem for Divide-and-Conquer Problems

Theorem

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

1. *If* $f(n) = O(n^{c'})$ *with* $c' < c$, *then*

$$T(n) = \Theta(n^c)$$

Master Theorem for Divide-and-Conquer Problems

Theorem

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

1. *If* $f(n) = O(n^{c'})$ *with* $c' < c$, *then*

$$T(n) = \Theta(n^c)$$

2. *If* $f(n) = \Theta(n^c)$, *then*

$$T(n) = \Theta(n^c \log n)$$

Master Theorem for Divide-and-Conquer Problems

Theorem

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

1. *If* $f(n) = O(n^{c'})$ *with* $c' < c$, *then*

$$T(n) = \Theta(n^c)$$

2. *If* $f(n) = \Theta(n^c)$, *then*

$$T(n) = \Theta(n^c \log n)$$

3. *If* $f(n) = \Omega(n^{c'})$ *with* $c' > c$ *and there exists* $\alpha < 1$ *s.t. for all sufficiently large* n, $a\,f(n/b) \leqslant \alpha\,f(n)$, *then*

$$T(n) = \Theta(f(n))$$

Notes on the Master Theorem

- Very useful!
- We give a high-level proof of the theorem assuming that n is a power of b, and that there is no rounding issue, see Cormen et al. [2009] for a rigorous proof in all cases
- Does not cover all cases (e.g., if $f(n) = \Theta(n^c \log n)$)
- There exist generalizations to more complex cases
- In particular, the Akra–Bazzi method deals with the case when we cut in two parts of inequal sizes

Applying the Master Theorem (examples)

Binary search: $a = 1$, $b = 2$, $f(n) = \Theta(1)$, $c = \log_2 1 = 0$.

Case 2 $\Rightarrow T(n) = \Theta(\log n)$

Applying the Master Theorem (examples)

Binary search: $a = 1$, $b = 2$, $f(n) = \Theta(1)$, $c = \log_2 1 = 0$.

Case 2 $\Rightarrow T(n) = \Theta(\log n)$

Blocked MM: $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, $c = \log_2 8 = 3$.

Case 1 $\Rightarrow T(n) = \Theta(n^3)$ – not better than direct algorithm!

Applying the Master Theorem (examples)

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

Blocked MM: $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, $c = \log_2 8 = 3$.
Case 1 $\Rightarrow T(n) = \Theta(n^3)$ – not better than direct algorithm!

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

Historical Note: Matrix Multiplication Exponents

What is known about the exponent ω such that $n \times n$ matrix multiplication is $O(n^\omega)$?

Year	Algorithm	Exponent
1812	Direct	$\omega \leqslant 3$
1969	Strassen	$\omega < 2.808$
1990	Coppersmith–Winograd	$\omega < 2.376$
2025	Alman, Duan, Williams, Xu, Xu, Zhou (SODA 2025)	$\omega < 2.371339$

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

$$= O\Big(n^{c'} \times \big(b^{(c-c')t} - 1\big)\Big)$$

$$= O\Big(n^{c'} \times \big(n^{c-c'} - 1\big)\Big) = O(n^c)$$

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

Write $n = b^t$ and expand:

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

Since $a = b^c$, we obtain:

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

and the leaves contribute $\Theta(n^c)$. Hence $T(n) = \Theta(n^c)$.

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

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

But $a = b^c$, so the ratio is 1 and the sum has $\log_b n$ terms:

$$g(n) = \Theta(n^c \log n)$$

Leaf work is $\Theta(n^c)$ and does not change the order. Thus $T(n) = \Theta(n^c \log n)$.

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

Assume $a\, f(n/b) \leqslant \alpha f(n)$ for some $\alpha < 1$ and large n. Then, along any path of length j,

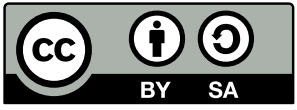
$$a^j f\left(\frac{n}{b^j}\right) \leqslant \alpha^j f(n)$$

Hence

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

Also $g(n) \geqslant f(n)$ from the root level, so $g(n) = \Omega(f(n))$. Leaf work is $\Theta(n^c) = O(f(n))$ since $c' > c$. Therefore $T(n) = \Theta(f(n))$.

Licensing



Bibliography I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.