Divide & Conquer (I)

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide-and-conquer algorithms

- Strategy:
  - Divide the problem into smaller subproblems of the same type of problem
  - Solve the subproblems recursively
  - Combine the answers to solve the original problem

- The work is done in three places:
  - In partitioning the problem into subproblems
  - In solving the basic cases at the tail of the recursion
  - In merging the answers of the subproblems to obtain the solution of the original problem

Conventional product of polynomials

Example:

$$P(x) = 2x^3 + x^2 - 4$$

$$Q(x) = x^2 - 2x + 3$$

$$(P \cdot Q)(x) = 2x^5 + (-4+1)x^4 + (6-2)x^3 + 8x - 12$$

$$(P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12$$

Conventional product of polynomials

```python
Polynomial = list[float]

def mul(p: Polynomial, q: Polynomial) -> Polynomial:
    """Returns p×q (product of polynomials)"""

    # degree(p) = len(p)-1, degree(q) = len(q)-1
    # degree(r) = degree(p)+degree(q)
    r: Polynomial = [0]*(len(p) + len(q) - 1)
    for i, pi in enumerate(p):
        for j, qj in enumerate(q):
            r[i+j] += pi*qj
    return r
```

Complexity analysis:

- Multiplication of polynomials of degree n: $\mathrm{O}(n^2)$
- Addition of polynomials of degree n: $\mathrm{O}(n)$

Product of polynomials: Divide&Conquer

Assume that we have two polynomials with n coefficients (degree $n-1$)

	$n-1$ $\quad\quad$ $n/2$	0
$\boldsymbol{P}$:	$\boldsymbol{P_L}$	$\boldsymbol{P_R}$
$\boldsymbol{Q}$:	$\boldsymbol{Q_L}$	$\boldsymbol{Q_R}$

$$P(x)\cdot Q(x) = P_L(x)\cdot Q_L(x)\cdot x^n + (P_R(x)\cdot Q_L(x) + P_L(x)\cdot Q_R(x))\cdot x^{n/2} + P_R(x)\cdot Q_R(x)$$

$$T(n) = 4\cdot T(n/2) + \mathrm{O}(n) = \mathrm{O}(n^2)$$ ← Shown later

Product of complex numbers

- The product of two complex numbers requires four multiplications:

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

- Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: ac, bd and $(a + b)(c + d)$

$$bc + ad = (a + b)(c + d) - ac - bd$$

- A similar observation applies for polynomial multiplication.

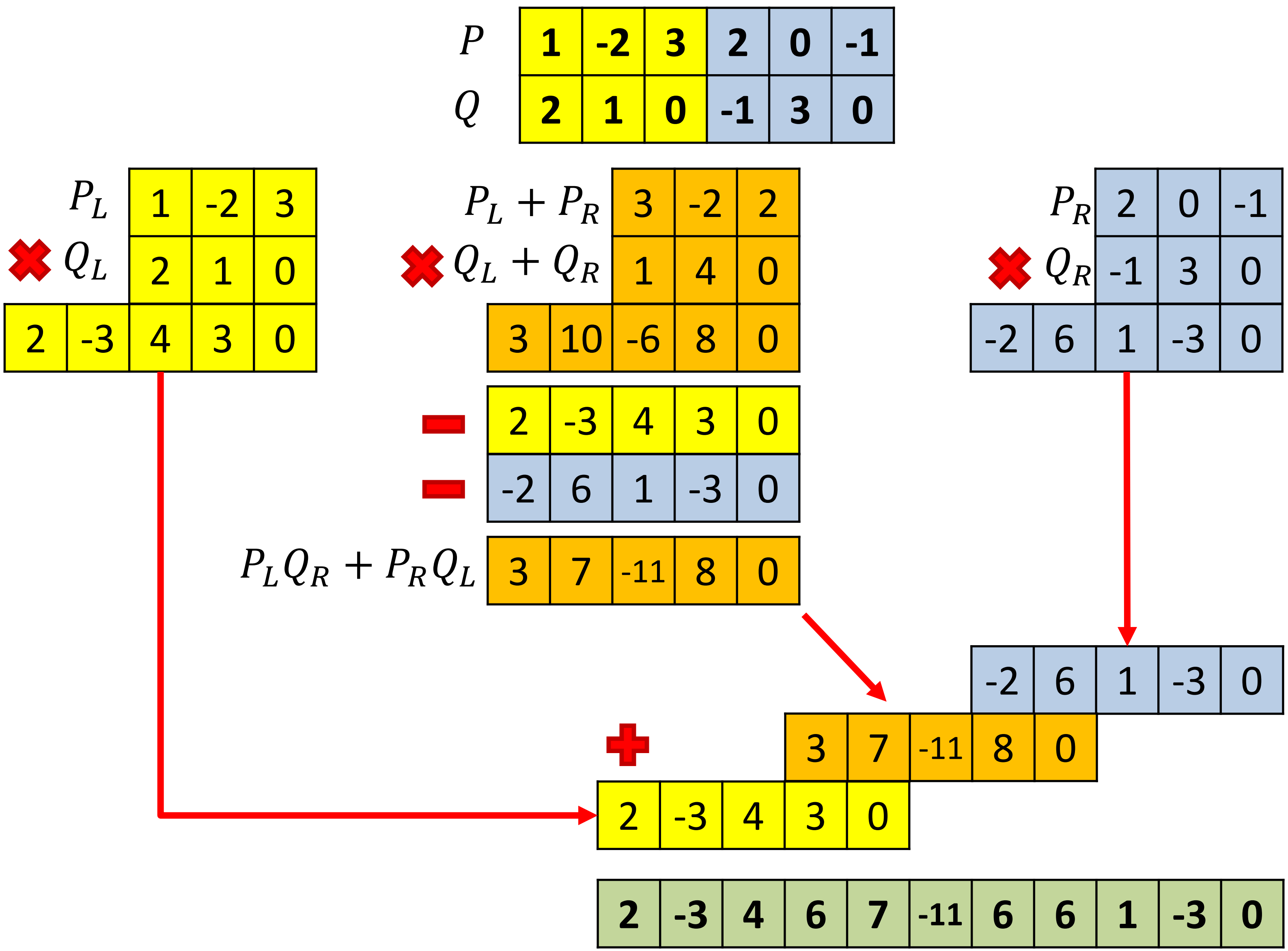
Product of polynomials with Gauss's trick

$$\begin{aligned} R_1 &= P_L Q_L \\ R_2 &= P_R Q_R \\ R_3 &= (P_L + P_R)(Q_L + Q_R) \end{aligned}$$

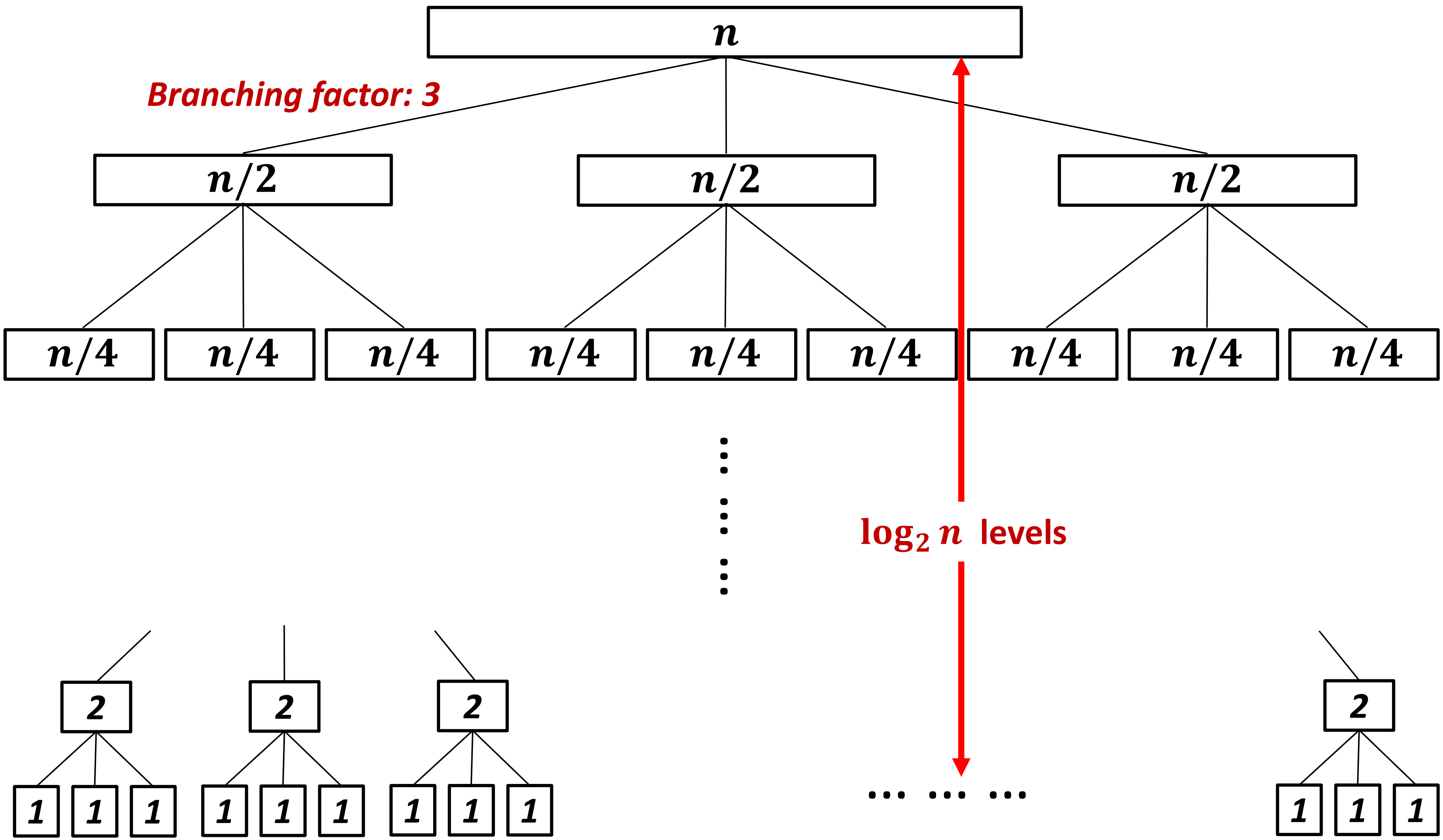
$$PQ = \underbrace{P_L Q_L}_{R_1} x^n + \underbrace{(P_R Q_L + P_L Q_R)}_{R_3 - R_1 - R_2} x^{n/2} + \underbrace{P_R Q_R}_{R_2}$$

$$T(n) = 3T(n/2) + \mathrm{O}(n)$$

Polynomial multiplication: recursive step



Pattern of recursive calls



Useful reminders

- Sum of geometric series with ratio r:

$$S = k + kr + kr^2 + kr^3 + \cdots + kr^{n-1} = k\left(\frac{1-r^n}{1-r}\right)$$

For a decreasing series ($r < 1$): $S \leq \frac{k}{1-r}$

- Logarithms:

$$\log_b n = \log_b a \cdot \log_a n$$

$$a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}$$

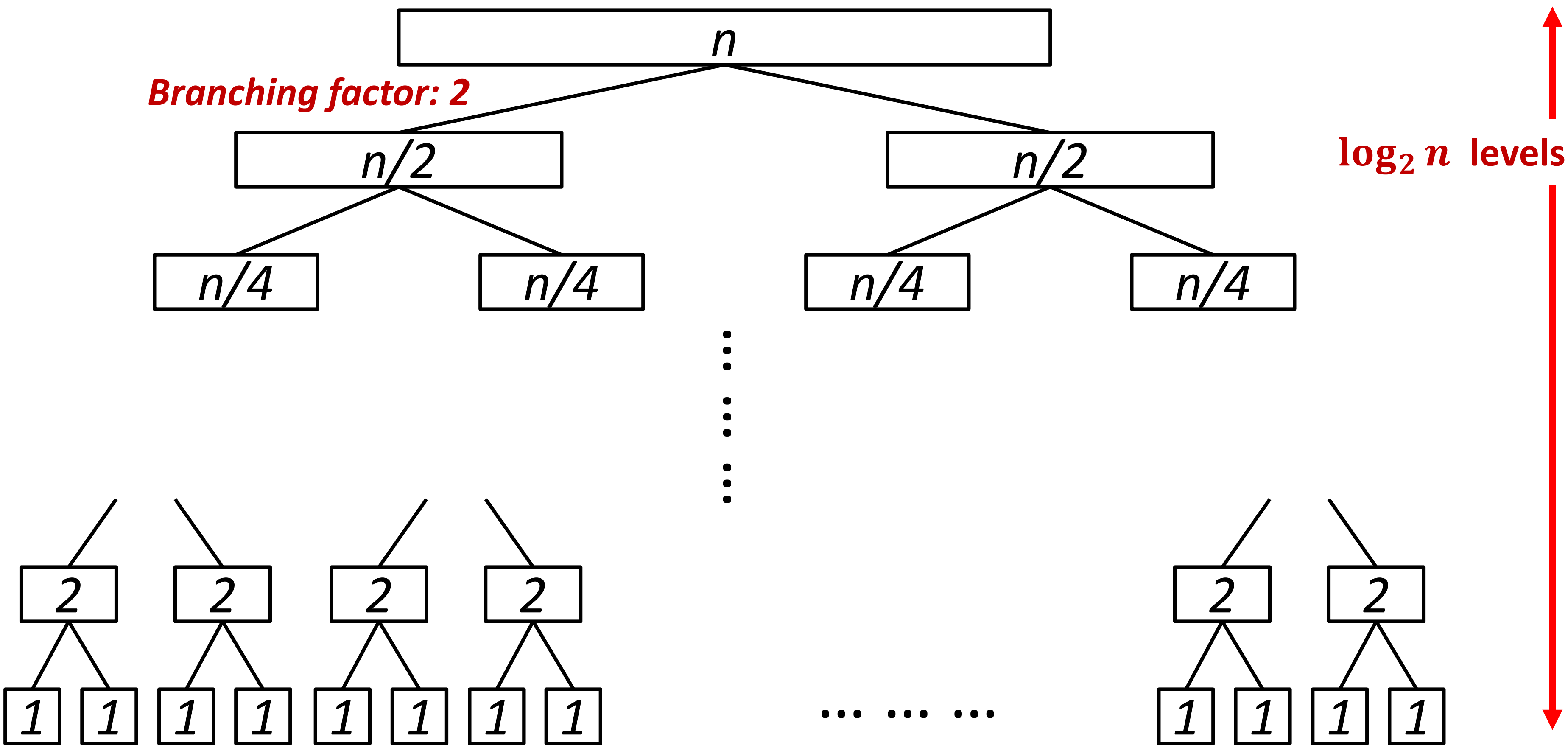
Complexity analysis

- The time spent at level k is

$$3^k \cdot \mathrm{O}\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \cdot \mathrm{O}(n)$$

- For $k = 0$, runtime is $\mathrm{O}(n)$.
- For $k = \log_2 n$, runtime is $\mathrm{O}\left(3^{\log_2 n}\right)$, which is equal to $\mathrm{O}\left(n^{\log_2 3}\right)$.
- The runtime per level increases geometrically by a factor of 3/2 per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series.
- Therefore, the complexity is $\mathrm{O}(n^{1.59})$.

A popular recursion tree



Example: efficient sorting algorithms.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \mathrm{O}(n)$$

Algorithms may differ on the amount of work done at each level: $\mathrm{O}(n^c)$

Examples

Algorithm	**Branch**	**c**	**Runtime equation**
Power (x^y)	1	0	$T(y) = T(y/2) + \text{O}(1)$
Binary search	1	0	$T(n) = T(n/2) + \text{O}(1)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + \text{O}(n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + \text{O}(n)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + \text{O}(n)$

Master theorem

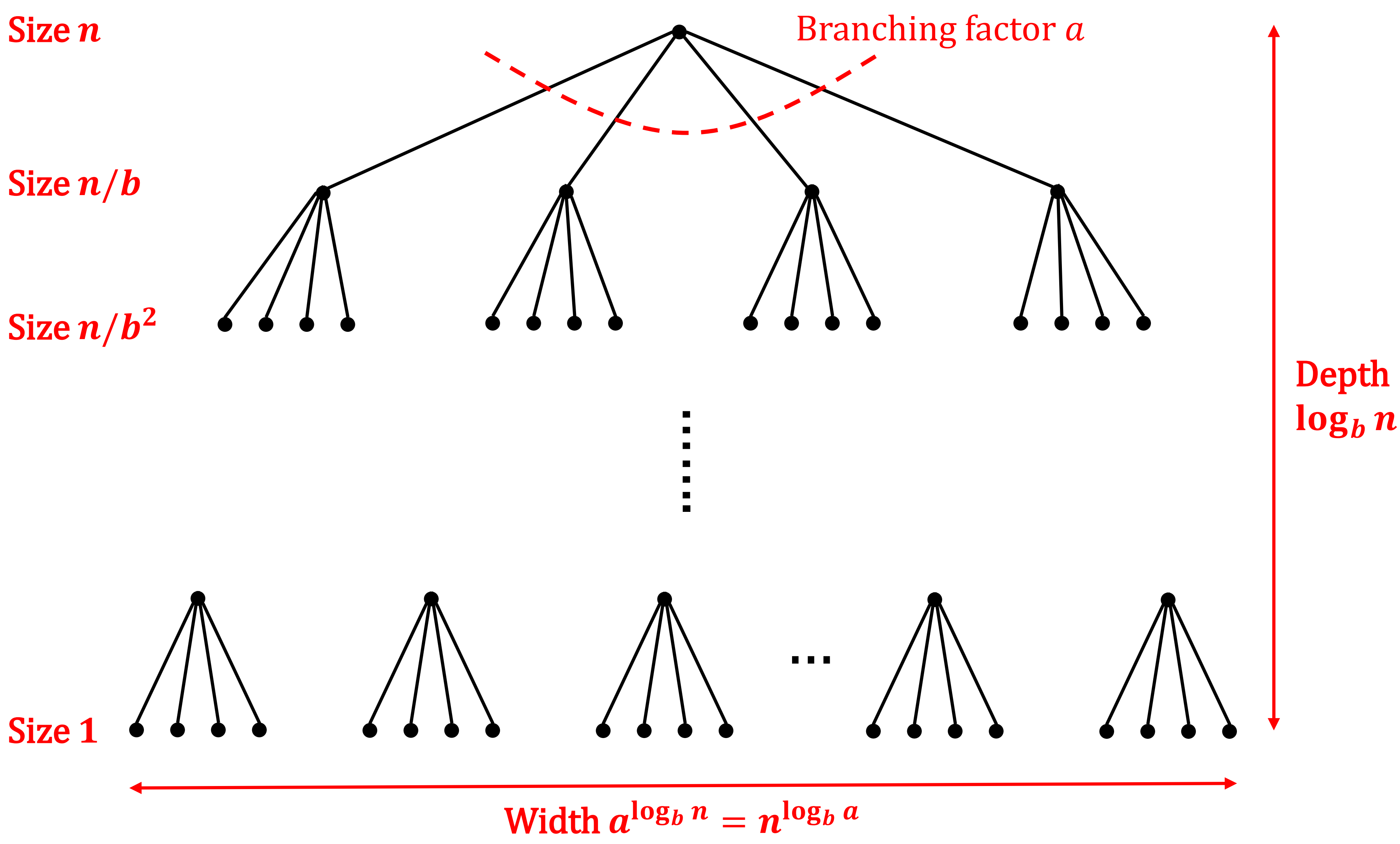
- Typical pattern for Divide&Conquer algorithms:
  - Split the problem into a subproblems of size n/b
  - Solve each subproblem recursively
  - Combine the answers in $\mathrm{O}(n^c)$ time

- Running time: $T(n) = a \cdot T(n/b) + \mathrm{O}(n^c)$

- Master theorem:

$$T(n) = \begin{cases} \mathrm{O}(n^c) & \text{if } \; a < b^c \\ \mathrm{O}(n^c \log n) & \text{if } \; a = b^c \\ \mathrm{O}\left(n^{\log_b a}\right) & \text{if } \; a > b^c \end{cases}$$

Master theorem: recursion tree



Master theorem: proof

- For simplicity, assume n is a power of b.
- The base case is reached after $\log_b n$ levels.
- The kth level of the tree has a^k subproblems of size n/b^k.
- The total work done at level k is:

$$a^k \times \mathrm{O}\left(\frac{n}{b^k}\right)^c = \mathrm{O}(n^c) \times \left(\frac{a}{b^c}\right)^k$$

- As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^c. We need to find the sum of such a series.

$$T(n) = \mathrm{O}(n^c) \cdot \underbrace{\left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \cdots + \frac{a^{\log_b n}}{b^{(\log_b n)c}}\right)}_{\log_b n \text{ terms}}$$

Master theorem: proof

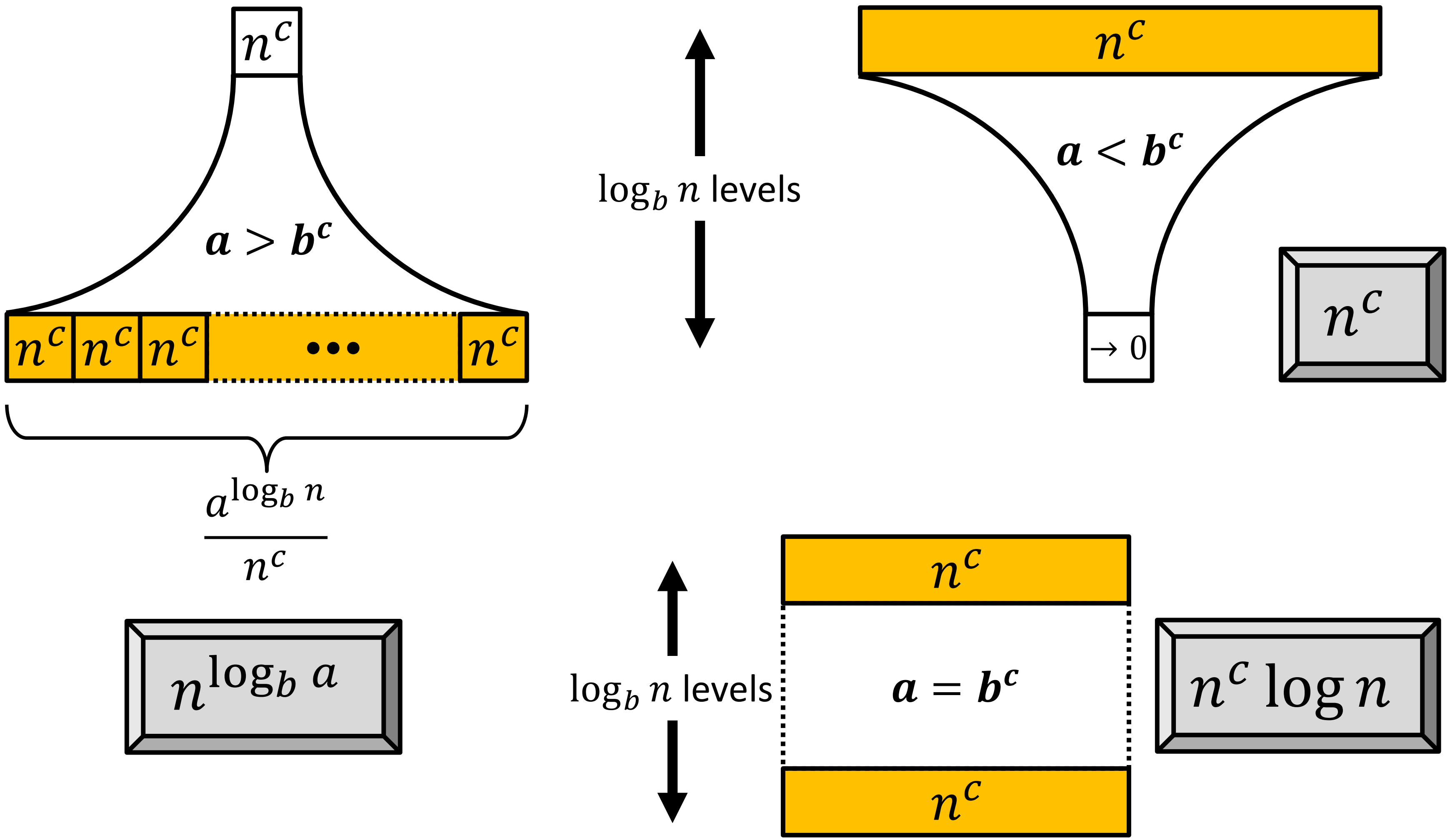
- Case $a/b^c < 1$. Decreasing series. The sum is dominated by the first term ($k = 0$): $\mathrm{O}(n^c)$.

- Case $a/b^c > 1$. Increasing series. The sum is dominated by the last term ($k = \log_b n$):

$$n^c \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \left(\frac{a^{\log_b n}}{(b^{\log_b n})^c}\right) = a^{\log_b n} = n^{\log_b a}$$

- Case $a/b^c = 1$. We have $\mathrm{O}(\log n)$ terms all equal to $\mathrm{O}(n^c)$.

Master theorem: visual proof



Master theorem: examples

Running time: $T(n) = a \cdot T(n/b) + O(n^c)$

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O\left(n^{\log_b a}\right) & \text{if } a > b^c \end{cases}$$

Algorithm	**a**	**c**	**Runtime equation**	**Complexity**
Power (x^y)	1	0	$T(y) = T(y/2) + O(1)$	$O(\log y)$
Binary search	1	0	$T(n) = T(n/2) + O(1)$	$O(\log n)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + O(n)$	$O(n \log n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + O(n)$	$O(n^2)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + O(n)$	$O(n^{\log_2 3})$

$b = 2$ for all the examples

Product multiplication

- Fundamental question:

  Can polynomials be multiplied efficiently when the degree is large?

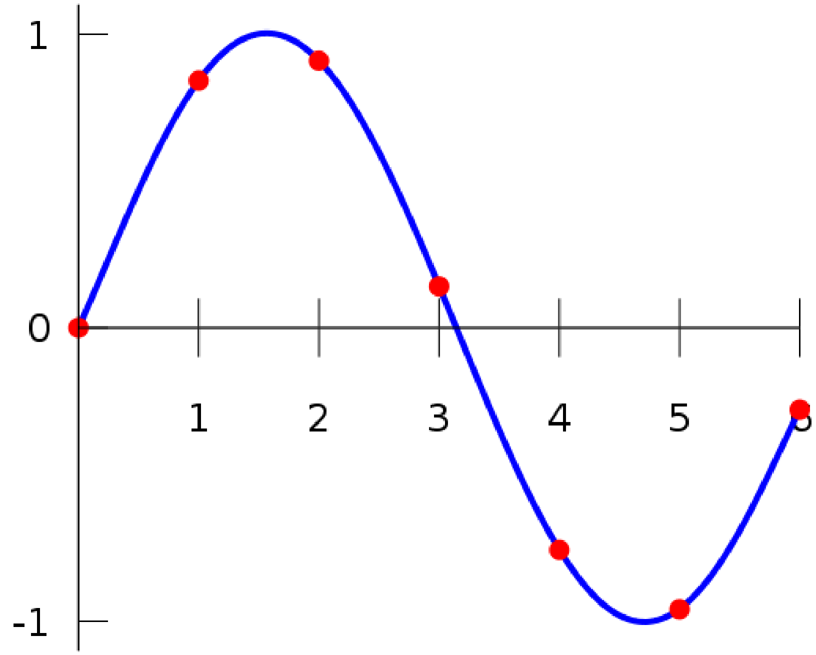
- Answer: yes (FFT: Fast Fourier Transform)

- FFT is an essential algorithm for efficient signal analysis. The algorithm will not be explained in this course.

Polynomials: point-value representation

- **Fundamental Theorem (Gauss)**: A degree-n polynomial with complex coefficients has exactly n complex roots.

- **Corollary**: A degree-n polynomial $A(x)$ is uniquely identified by its evaluation at $n + 1$ distinct values of x.



Polynomial representation

$$P(x) = x^3 - 2x^2 - 3x + 1$$

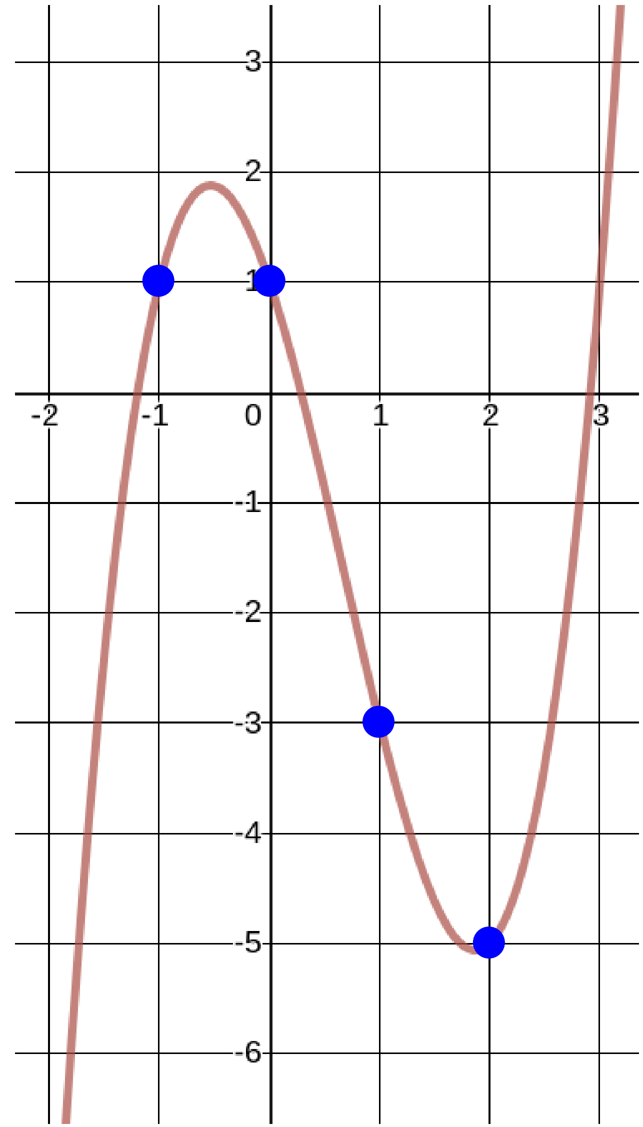
$$P(x) = (1, -2, -3, 1)$$

Coefficient representation

Evaluation ↓ ↑ **Interpolation**

Point-value representation

$$P(x) = \{(-1,1), (0,1), (1,-3), (2,-5)\}$$



Conversion between both representations

representation	addition	multiplication	evaluation
coefficient	$O(n)$	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n)$	$O(n^2)$

$a_0, a_1, \cdots, a_{n-1}$ — evaluation → $(x_0, y_0), \cdots, (x_{n-1}, y_{n-1})$

$a_0, a_1, \cdots, a_{n-1}$ ← interpolation — $(x_0, y_0), \cdots, (x_{n-1}, y_{n-1})$

Coefficient representation

Point-value representation

Could we have an ***efficient*** algorithm to move from coefficient to point-value representation and vice versa?

Fast Fourier Transform (FFT): $O(n \log n)$

Fourier series

- Periodic function $f(t)$ of period 1:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(2\pi n t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n t)$$

- Fourier coefficients:

$$a_n = 2\int_0^T f(t)\cos(2\pi n t)\,dt, \qquad b_n = 2\int_0^T f(t)\sin(2\pi n t)\,dt$$

- Fourier series is fundamental for signal analysis (to move from time domain to frequency domain, and vice versa)

Why Fourier Transform?