Cost of Algorithms

Inputs: parameterized by an integer n, called the size

$T(I)$ = runtimes on input I → runtime of a particular instance

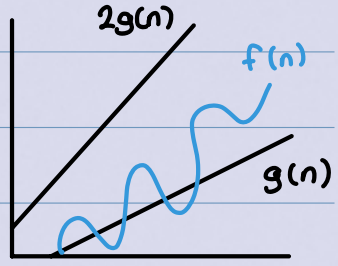
$T_{worst}(n) = \max_{I \text{ of size } n}(T(I))$ → worst-case runtime (default)

$T_{best}(n) = \min_{I \text{ of size } n}(T(I))$ → best-case runtime, not used much

Asymptotic Notation

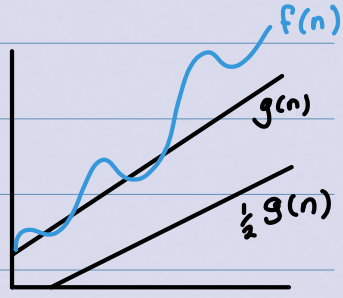
- consider 2 functions $f(n)$ and $g(n)$ with values in $\mathbb{R}_{>0}$

big-O: we say that $f(n) \in O(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \leq C\,g(n)$



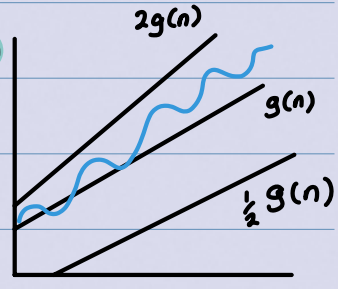
big-Ω: we say that $f(n) \in \Omega(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \geq C\,g(n)$

⇒ equivalent to $g(n) \in O(f(n))$



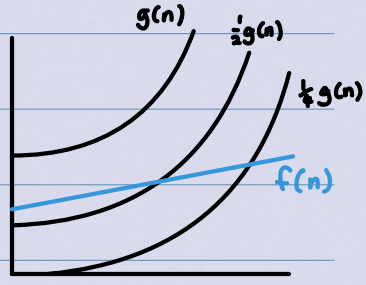
Θ: we say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

⇒ in particular true if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$ for some $0 < C < \infty$



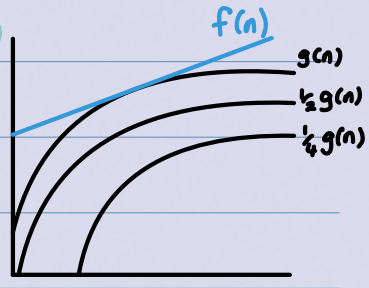
little-o: we say that $f(n) \in o(g(n))$ if for all $C > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \leq C\,g(n)$

⇒ equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$



little-ω: we say that $f(n) \in \omega(g(n))$ if for all $c > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \geq cg(n)$

⇒ equivalent to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$



Examples

a) $n^k + c_{k-1}n^{k-1} + \ldots + c_0 \in \Theta(n^k)$

b) $2^{n-1} \notin \Theta(2^n)$

c) $(n-1)! \in \Theta(n!)$

Definitions for Several Parameters

consider two functions $f(n,m)$, $g(n,m)$ with values in $\mathbb{R}_{>0}$

$f(n,m)$ is in $O(g(n,m))$ if there exist c, n_0, m_0 such that $f(n,m) \leq Cg(n,m)$ for $n \geq n_0$ or $m \geq m_0$

Case Study: Maximum Subarray

Given an array $A[0,\ldots,n]$, find a contiguos subarray $A[i,\ldots,j]$ that maximises the sum $A[i] + \ldots + A[j]$.

Example: given $A = [10, -5, 4, 3, -5, 6, -1, -1]$, the subarray $A[0,\ldots,5] = [10, -5, 4, 3, -5, 6]$ has sum $10-5+4+3-5+6=13$ is the max.

Brute Force algorithm:

runtime is $\Theta(n^3)$

but, we can improve on this!
idea: we recompute the same sum many times in the j loop

```
BruteForce (A)
1. opt = 0
2. for (i=0→n) {
3.     for (j=i→n) {
4.         sum = 0
5.         for (k=i→j) {
6.             sum += A[k]
```

```
BetterBruteForce(A)
1. opt = 0
2. for (i=0→n) {
3.   sum = 0
4.   for (j=1→n) {
5.     sum += A[j]
6.     if (sum > opt) {
7.       opt = sum
8.     }
9.   }
10. }
11. return opt
```

```
7.     }
8.     if (sum > opt) {
9.       opt = sum
10.    }
11.  }
12. }
13. return opt
```

⇒ runtime $O(n^2)$

but, we can still do better using a **divide-and-conquer approach**:

idea: solve the problem twice in size $n/2$ (assuming n is a power of 2). Then, the **optimal subarray** (if not empty):

1. is completely **in the left half** $A[0, \dots, n/2]$
2. or is completely **in the right half** $A[n/2+1, \dots, n]$
3. or **contains both** $A[n/2]$ **and** $A[n/2+1]$

⇒ the three cases are mutually exclusive

to find the optimal subarray in **case 3**, we have:

$$A[i] + \dots + A[n/2] + A[n/2+1] + \dots + A[j]$$

more abstractly, we have $F(i,j) = f(i) + g(j)$, for $i \in [0, \dots, n/2]$ and $j \in [n/2+1, \dots, n]$.

To maximise $F(i,j)$, we **maximise $f(i)$ and $g(j)$ independently!**

```
MaximiseLowerHalf(A)
1. opt = A[n/2]
```

```
MaximiseUpperHalf(A)
1. opt = A[n/2+1]
```

```
2. sum = A[n/2]
3. for (i = n/2 - 1 → 0) {
4.     sum += A[i]
5.     if (sum > opt) {
6.         opt = sum
7.     }
8. }
9. return opt
```

```
2. sum = A[n/2 + 1]
3. for (i = n/2 + 1 → n) {
4.     sum += A[i]
5.     if (sum > opt) {
6.         opt = sum
7.     }
8. }
9. return opt
```

↳ runtimes are $\Theta(n)$! ↵

So, final divide and conquer algorithm:

```
DivideAndConquerMaximumSubarray(A[0, ..., n])
1. if (n == 1) return max(A[0], 0)
2. opt_low = DivideAndConquerMaximumSubarray(A[0, ..., n/2])
3. opt_high = DivideAndConquerMaximumSubarray(A[n/2 + 1, ..., n])
4. opt_mid = MaximiseLowerHalf(A) + MaximiseUpperHalf(A)
5. return max(opt_low, opt_mid, opt_high)
```

⇒ runtime: $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$

Solving Recurrences

Consider a recursive algorithm called Algo.

Assumption: for any input of size $n \geq n_0$, Algo does:

- a recursive calls, in size either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ ($a > 0$, $b > 1$, constant)
- between $c'n^y$ and cn^y extra operations ($c, c' \neq 0$, y constant)

Claim: Solving the sloppy recurrence $T(n) = aT(n/b) + cn^y$ for powers of b gives a valid Θ-bound for best and worst-

case runtimes.

- Remark: if we only know that we do at most cn^3 extra operations, we only get a big-O bound.

Best and Worst Case Recurrence Relations

let $T_{worst}(n)$, $T_{best}(n)$ be the worst / best cases in size n.

worst-case recurrence: $T_{worst}(1) = d$, and

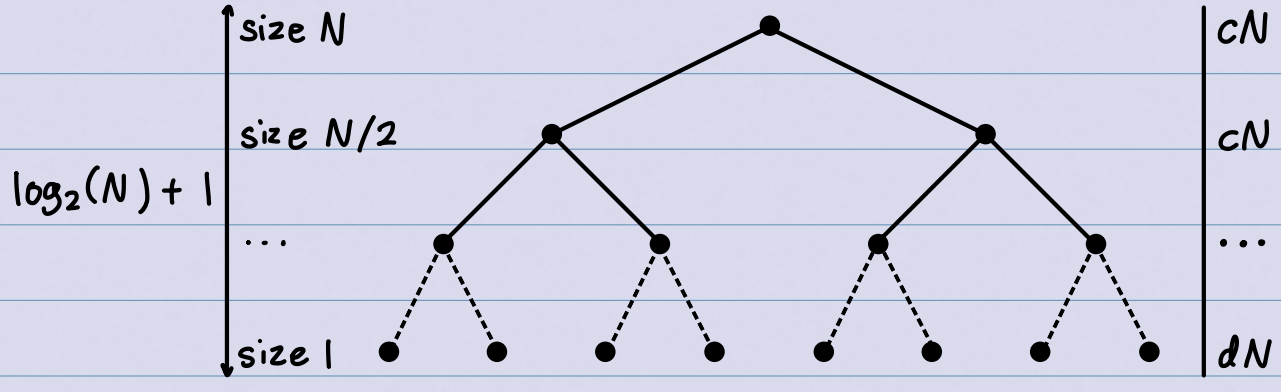
$$T_{worst}(n) \leq T_{worst}(\lceil n/2 \rceil) + T_{worst}(\lfloor n/2 \rfloor) + cn \quad \text{if } n>1$$

best-case recurrence: $T_{best}(1) = d'$, and

$$T_{best}(n) \geq T_{best}(\lceil n/2 \rceil) + T_{best}(\lfloor n/2 \rfloor) + cn \quad \text{if } n>1$$

we define N as the next power of 2 of n, so $N < 2n$.

the mergesort recursion tree



total: $t(N) = cN\log_2(N) + dN \leq KN\log_2(N)$ for N a power of 2

consequence: $T_{worst}(n) \leq KN\log_2(N) \leq K(2n)\log_2(2n) \leq K' n \log_2 n$,

so, $T_{worst}(n) \in O(n \log n)$

remark: same approach proves $T_{best}(n) \in \Omega(n \log n)$, and so,

$T_{best}(n)$, $T_{worst}(n) \in \Theta(n \log n)$

The Master Theorem

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = aT(n/b) + cn^y \quad (n \geq b), \qquad T(1) = d.$$

Let $x = \log_b a$ (so $a = b^x$).

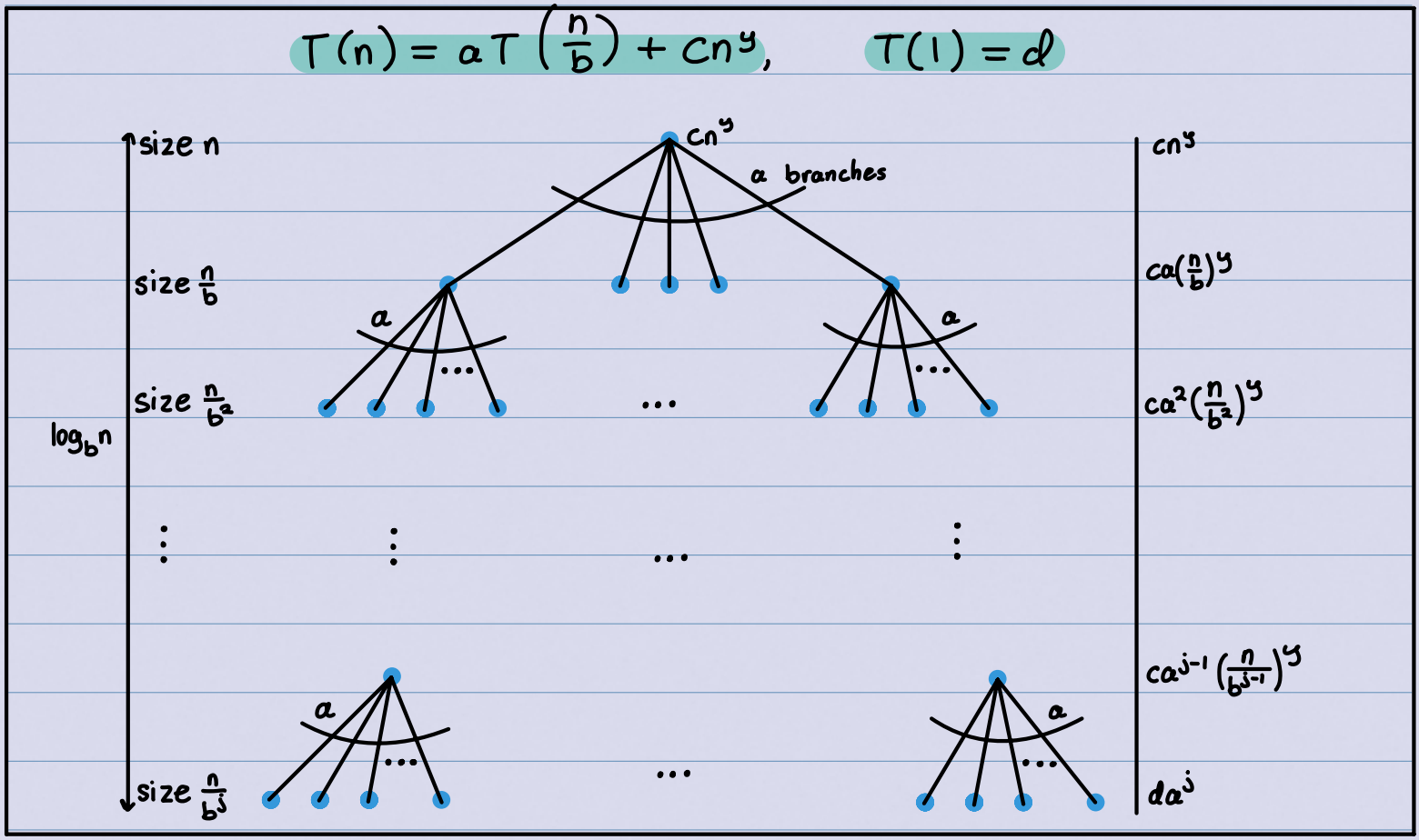
Then, for n a power of b,

$$T(n) \in \begin{cases} \Theta(n^y) & \text{if } y > x \quad \text{root heavy} \\ \Theta(n^y \log n) & \text{if } y = x \quad \text{balanced} \\ \Theta(n^x) & \text{if } y < x \quad \text{leaf heavy} \end{cases}$$

Recursion Tree

Suppose that $n = b^j$, $a \geq 1$, $b \geq 2$ are integers and

$$T(n) = aT(n/b) + cn^y, \qquad T(1) = d$$



Breakdown of the Cost

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = aT(n/b) + cn^y, \quad T(1) = d.$$

Let $n = b^j$.

Size of subproblem	# nodes	cost/node	total cost
$n = b^j$	1	cn^y	cn^y
$n/b = b^{j-1}$	a	$c(n/b)^y$	$ca(n/b)^y$
$n/b^2 = b^{j-2}$	a^2	$c(n/b^2)^y$	$ca^2(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n/b^{j-1} = b$	a^{j-1}	$c(n/b^{j-1})^y$	$ca^{j-1}(n/b^{j-1})^y$
$n/b^j = 1$	a^j	d	da^j
generally: $n/b^i = b^{j-i}$	a^i	$c(n/b^i)^y$	$ca^i(n/b^i)^y$

Computing $T(n)$:

①: $x = \log_b a$, so $a = b^x$. we know $n = b^j$. $\therefore$, $a^j = (b^x)^j = (b^j)^x = n^x$.

total: $T(n) = da^j + cn^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i \overset{\text{using ①}}{=} dn^x + cn^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i$

observation: geometric sum with ratio $r = \frac{a}{b^y} = \frac{b^x}{b^y} = b^{x-y}$!

- if $r < 1$, then $b^{x-y} < 1$, so $x < y$.

  we know that $\sum_{i=0}^{j-1} r^i \in \Theta(1)$ if $r < 1$.

  $\therefore$, $T(n) = dn^x + cn^y \times \Theta(1) \in \Theta(n^y)$. $\therefore$, $x < y \Rightarrow T(n) \in \Theta(n^y)$

- if $r=0$, then $b^{x-y}=1$, so $x=y$.
  we know that $\sum_{i=0}^{j-1} r^i \in \mathcal{O}(j) = \mathcal{O}(\log n)$ if $r=1$
  $\therefore$, $T(n) = dn^x + cn^y \times \mathcal{O}(\log n) \in \mathcal{O}(n^y \log n)$ $\therefore$, $x=y \Rightarrow T(n) \in \mathcal{O}(n^y \log n)$

- if $r>1$, then $b^{x-y} > 1$, so $x>y$
  we know that $\sum_{i=0}^{j-1} r^i \in \mathcal{O}(r^j)$ if $r>1$
  $\therefore$, $T(n) = dn^x + cn^y \times \mathcal{O}(r^j)$. $r^j = a^j/b^{yj} = n^x/n^y$.
  $T(n) = dn^x + cn^y \times \mathcal{O}(n^x/n^y) \in \mathcal{O}(n^x)$. $\therefore$, $x>y \Rightarrow T(n) \in \mathcal{O}(n^x)$.

Examples: find $\mathcal{O}$-bounds for the following:

1) $T(n) = 4T(n/2) + n$ (multiplying polynomials)
   $a=4$, $b=2$, $y=1$, so $x = \log_b a = \log_2 4 = 2$.

   $\therefore$, $x=2 > y=1$, so $T(n) \in \mathcal{O}(n^2)$

2) $T(n) = 2T(n/2) + n^2$ (kd-trees)
   $a=2$, $b=2$, $y=2$, so $x = \log_b a = \log_2 2 = 1$

   $\therefore$, $x=1 < y=2$, so $T(n) \in \mathcal{O}(n^2)$

3) $T(n) = 2T(n/4) + 1$ (kd-trees)
   $a=2$, $b=4$, $y=0$, so $x = \log_b a = \log_4 2 = 1/2$

   $\therefore$, $x=1/2 < y=0$, so $T(n) \in \mathcal{O}(\sqrt{n})$

4) $T(n) = T(n/2) + 1$ (binary search)
   $a=1$, $b=2$, $y=0$, so $x = \log_b a = \log_2 1 = 0$

$\therefore, x = 0 = y$, so $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

5) $T(n) = T(n/2) + n$ (amortized analysis of dynamic arrays)

$a = 1, b = 2, y = 1,$ so $x = \log_b a = \log_2 1 = 0$

$\therefore, x = 0 < y = 1,$ so $T(n) \in \Theta(n)$

Alternative: guess and prove: $T(n) = 2T(n/2) + n, \quad T(1) = 0.$

guess $T(n) \leq n$. Then $T(n/2) \leq n/2$.

$T(n) = 2T(n/2) + n \leq 2(n/2) + n = 2n \not\leq n.$

guess $T(n) \leq kn$ for some k. Then $T(n/2) \leq kn/2$.

$T(n) = 2T(n/2) + n \leq 2(kn/2) + n = kn + n \not\leq kn$

guess $T(n) \leq kn\log_2 n$ for some k. Then $T(n/2) \leq kn\log_2(n/2)$

$T(n) = 2T(n/2) + n \leq 2\left(\frac{kn\log_2(n/2)}{2}\right) + n = kn\log_2 n - kn + n \leq n$ for $k \geq 1$.

$\therefore,$ $T(n) \leq kn\log n$ for $k \geq 1$, so $T(n) \in O(n\log n)$.

(proving $T(n) = \ldots$ is harder)

Divide and Conquer Framework

to solve a problem in size n:

- Divide
  - break the input into smaller problems
  - ideally few such problems, all of size n/b for some b
- Conquer

- solve these problems recursively

- Recombine
  - deduce the solution of the main problem from the subproblems

Polynomial and Matrix Multiplication

Multiplying Polynomials

Goal: given $F = F_0 + \dots + F_{n-1}x^{n-1}$ and $G = G_0 + \dots + G_{n-1}x^{n-1}$

compute $H = FG = F_0G_0 + (F_0G_1 + F_1G_0)x + \dots + F_{n-1}G_{n-1}x^{2n-2} = \sum_{i=0}^{2n-2} H_i x^i$

Remark: unit cost model, assume all f_i and g_i fit in one word. Then, input and output size $O(n)$, easy algorithm in $O(n^2)$:

Naive Polynomial Multiplication (F, G)

```
1. for (i = 0 → 2n-2) {
2.     H_i = 0
3. }
4. for (i = 0 → n-1) {
5.     for (j = 0 → n-1) {
6.         H_{i+j} += F_i G_j
7.     }
8. }
```

In degree 1: $F = F_0 + F_1x$, $G = G_0 + G_1x$,

$H = F_0G_0 + (F_0G_1 + F_1G_0)x + F_1G_1x^2$

In higher degree: $F = F_0'(x) + F_1'(x)x^{n/2}$, $G = G_0'(x) + G_1'(x)x^{n/2}$,

$H = F_0'(x)G_0'(x) + (F_0'(x)G_1'(x) + F_1'(x)G_0'(x))x^{n/2} + F_1'(x)G_1'(x)x^n$

Analysis:

- 4 recursive calls in size $n/2$
- $\Theta(n)$ additions to compute $F_0'(x)G_1'(x) + F_1'(x)G_0'(x)$
- multiplications by $x^{n/2}$ and x^n are free
- $\Theta(n)$ additions to obtain H

(Sloppy) recurrence: $T(n) = 4T(n/2) + cn$,

↳ $a=4, b=2, y=1$, so $T(n) \in \Theta(n^2)$

∴, no better than the naive algorithm!

Karatsuba's Algorithm

Key Idea: a formula for degree 1 polynomials that does only 3 (instead of 4) multiplications: with $F = F_0 + F_1 x$, $G = G_0 + G_1 x$, $H = F_0G_0 + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x + F_1G_1x^2$

In higher degree: with $F = F_0' + F_1' x^{n/2}$, $G = G_0' + G_1' x^{n/2}$ (where F_0', F_1', G_0', G_1' are polynomials of degree $\leq n/2 - 1$):
$H = F_0'G_0' + ((F_0' + F_1')(G_0' + G_1') - F_0'G_0' - F_1'G_1')x^{n/2} + F_1'G_1'x^n$.

Analysis:

- $\Theta(n)$ additions to compute $F_0' + F_1'$ and $G_0' + G_1'$
- 3 recursive calls in size $n/2$
- multiplications by $x^{n/2}$ and x^n are free
- $\Theta(n)$ additions and subtractions to combine the results

Recurrence: $T(n) = 3T(n/2) + cn$

↳ $a=3, b=2, y=1$, so $T(n) \in \Theta(n^{\log_2 3})$

Toom-Cook:

- A family of algorithms based on similar expressions as

Karatsuba

- for $k \geq 2$, $2k-1$ recursive calls in size n/k
- so $T(n) \in O(n^{\log_k(2k-1)})$
- gets as close to exponent 1 as we want (but very slowly)

Fast Fourier Transform (FFT):

- if we use complex coefficients, FFT can be used to multiply polynomials
- FFT follows the same recurrence as merge sort, $T(n) = 2T(n/2) + cn$
- so we can multiply polynomials in $O(n \log n)$ operations over $\mathbb{C}$

Multiplying Matrices

→ reminder: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot5+2\cdot6 & 1\cdot7+2\cdot8 \\ 3\cdot5+4\cdot6 & 3\cdot7+4\cdot8 \end{bmatrix} = \begin{bmatrix} 17 & 23 \\ 39 & 53 \end{bmatrix}$

$[i \times j] \cdot [k \times l]$ → can only multiply if $j=k$. resulting matrix is $[i \times l]$

Goal: given $A = [a_{ij}]_{1 \leq i,j \leq n}$ and $B = [b_{jk}]_{1 \leq j,k \leq n}$, compute $C = AB$

↳ remark: input and output $O(n^2)$ elements, easy algorithm in $O(n^3)$:

Naive Matrix Multiplication (A, B)

```
1. for (i=0→n) {
2.   for (j=0→n) {
3.     for (k=0→n) {
4.       C_{i,k} += A_{ij} · B_{jk}
5.     }
6.   }
7. }
```

this is $O(n^3)$, we can do better! Let's try divide-and-conquer:

divide by splitting the matrices into 4 sections:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} \qquad B = \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix}$$

with all $A_{i,k}$ and $B_{j,k}$ of size $n/2 \times n/2$. Then,

$$C = \begin{bmatrix} A_{0,0}B_{0,0} + A_{0,1}B_{1,0} & A_{0,0}B_{0,1} + A_{0,1}B_{1,1} \\ A_{1,0}B_{0,0} + A_{1,1}B_{1,0} & A_{1,0}B_{0,1} + A_{1,1}B_{1,1} \end{bmatrix}$$

however, this is still 8 recursive calls in size $\frac{n}{2}$ + $\mathcal{O}(n^2)$ additions which is in $\mathcal{O}(n^3)$! no improvement :(

Strassen's Algorithm

compute:

$$Q_1 = (A_{0,0} - A_{0,1})B_{1,1}$$
$$Q_2 = (A_{1,0} - A_{1,1})B_{0,0}$$
$$Q_3 = A_{1,1}(B_{0,0} + B_{1,0})$$
$$Q_4 = A_{0,0}(B_{0,1} + B_{1,1})$$
$$Q_5 = (A_{0,0} + A_{1,1})(B_{1,1} - B_{0,0})$$
$$Q_6 = (A_{0,0} + A_{1,0})(B_{0,0} + B_{0,1})$$
$$Q_7 = (A_{0,1} + A_{1,1})(B_{1,0} + B_{1,1})$$

and

$$C_{0,0} = Q_1 - Q_3 - Q_5 + Q_7$$
$$C_{0,1} = Q_4 - Q_1$$
$$C_{1,0} = Q_2 + Q_3$$
$$C_{1,1} = Q_5 + Q_6 - Q_2 - Q_4$$

now, we have 7 instead of 8 recursive calls! $= \mathcal{O}(n^{2.8}) < \mathcal{O}(n^3)$!

size is $n/2$ with $\mathcal{O}(n)$ additions, so $T(n) \Rightarrow \mathcal{O}(n^{\log_2 7})$

direct generalisation: an algorithm that does k multiplications for matrices of size l gives $T(n) \in \mathcal{O}(n^{\log_l k})$

Counting Inversions

Inversion: (i, j) is an inversion if $i<j$ and $A[i] > A[j]$

Goal: given an unsorted array $A[1, \ldots, n]$, find the number of inversions in it.

Example: with $A = [1, 5, 2, 6, 3, 8, 7, 4]$.

we get: $(2,3), (2,5), (2,8), (4,5), (4,8), (6,7), (6,8), (7,8)$

Remarks:

1. we show the indices where inversions occur
2. easy algorithm (2 nested loops) in $\mathcal{O}(n^2)$
3. to do better than n^2, we cannot list all inversions

towards a divide & conquer algorithm

idea:

- C_ℓ = number of inversions in $A[0, \ldots, n/2]$ (first half)
- C_r = number of inversions in $A[n/2+1, \ldots, n]$ (second half)
- C_t = number of transverse inversions with $i \leq n/2$ and $j > n/2$ (both halves)
- return $C_\ell + C_r + C_t$

↳ example with $A = [1, 5, 2, 6, 3, 8, 7, 4]$ (indices 0 1 2 3 4 5 6 7)

$C_\ell = 1$ $\{(1,2)\}$

$C_r = 3$ $\{(5,6), (5,7), (6,7)\}$

} calculated recursively

$C_t = 4$ $\{(1,4), (1,7), (3,4), (3,7)\}$ → calculate w/ merge sort

new goal: counting transverse inversions: how many pairs (i, j) with $i \leq n/2$, $j > n/2$, and $A[i] > A[j]$?

↳ eg, with $A = [1, 5, 2, 6, 3, 8, 7, 4]$, $C_t = \#i_s > 3 + \#i_s > 8 + \#i_s > 7 + \#i_s > 4$

or $C_t = \#j_s < 1 + \#j_s < 5 + \#j_s < 2 + \#j_s < 6$.

observation: this number does not change if both sides are sorted!

⇒ assume left and right sides are sorted, so $A = [1, 2, 5, 6, 3, 4, 7, 8]$

observation 2: since both sides sorted, merge will sort whole array:

```
Merge(A[1,...,n]) (both halves of A sorted)
1. copy A into new array S
2. i=1; j=n/2+1;
3. for (k=1→n) {
4.   if (i=n/2+1) A[k]=S[j++]
5.   else if (j=n+1) A[k]=S[i++]
6.   else if (S[i]≤S[j]) A[k]=S[i++]
7.   else A[k]=S[j++]
8. }
```

↳ goal: find c_t during merge

how? ⇒ when we insert S[i] back in A, count how many S[j]'s have already been inserted

easy! ☺ the difference is $j - j_{init} = j - (n/2 + 1)$!

```
Enhanced Merge(A[1,...,n]) (both halves of A sorted)
1. copy A into new array S
2. i=1; j=n/2+1; c=0;
3. for (k=1→n) {
4.   if (i=n/2+1) A[k]=S[j++]
5.   else if (j=n+1) A[k]=S[i++]; c+=n/2
6.   else if (S[i]≤S[j]) A[k]=S[i++]; c+=j-(n/2+1)
7.   else A[k]=S[j++]
8. }
9. return c;
```

Example: with $A = [1, 2, 5, 6, 3, 4, 7, 8]$

- when we insert 1 back into A, $j = 5$ so $c = c + 0$
- when we insert 2 back into A, $j = 5$ so $c = c + 0$
- when we insert 5 back into A, $j = 7$ so $c = c + 2$
- when we insert 6 back into A, $j = 7$ so $c = c + 2$

$\therefore$, $c_e = 4$! ☺

⇒ enhanced Merge is still $O(n)$, so the total remains $O(n \log n)$ to count the number of inversions!