Cost of Algorithms

Inputs: parameterized by an integer n, called the size

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

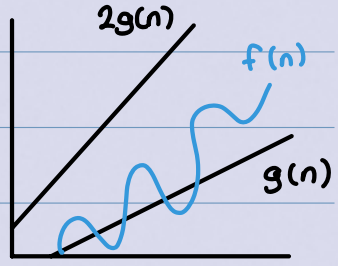
$T_{worst}(n) = \max_{I \text{ of size } n}(T(I))$ → worst-case runtime (default)

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

Asymptotic Notation

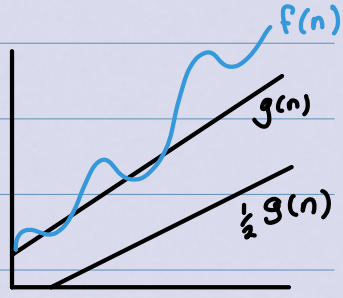
- consider 2 functions $f(n)$ and $g(n)$ with values in $\mathbb{R}_{>0}$

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



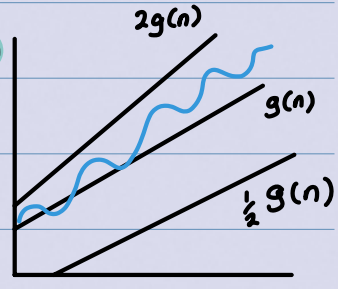
big-Ω: we say that $f(n) \in \Omega(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \geq C\,g(n)$

⇒ equivalent to $g(n) \in O(f(n))$



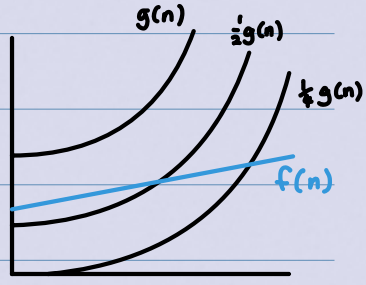
Θ: we say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

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



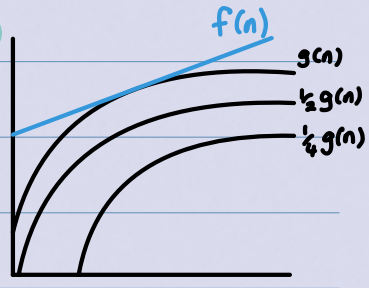
little-o: we say that $f(n) \in o(g(n))$ if for all $C > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \leq C\,g(n)$

⇒ 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

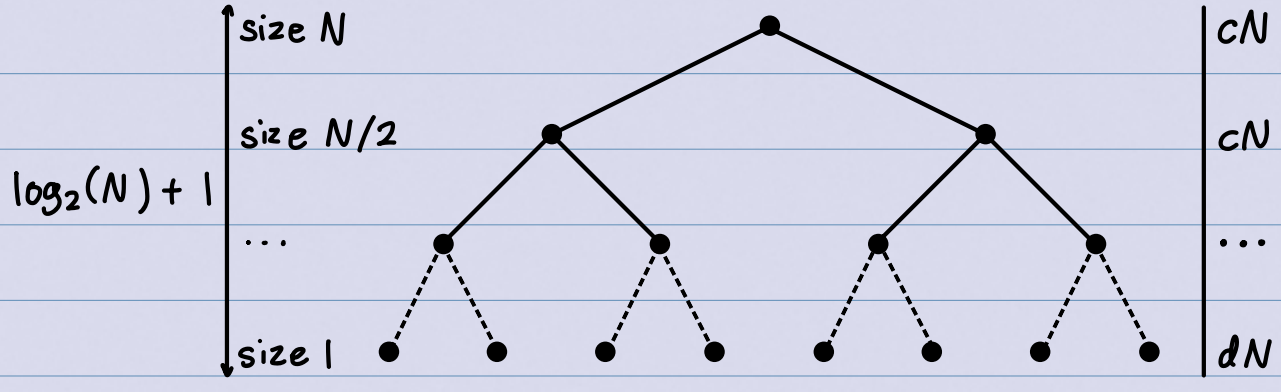
$$T_{worst}(n) \leq T_{worst}(\lceil n/2 \rceil) + T_{worst}(\lfloor n/2 \rfloor) + cn \quad \text{if } n>1$$

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 $a = b^x$).

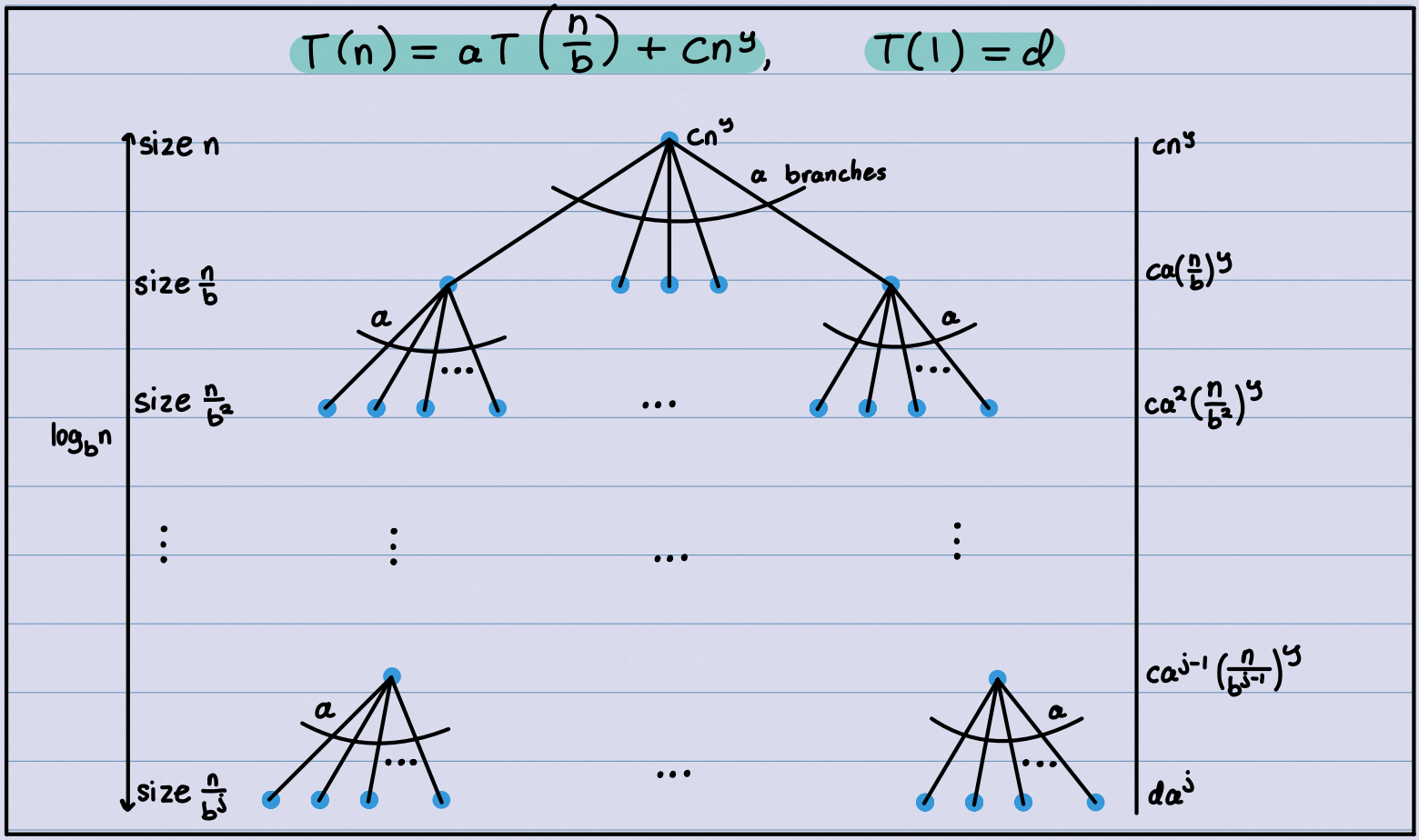
Then, for n a power of b,

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

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 $c = c + 2$

$\therefore$, $c_e = 4$! ü

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

Generalised Master Theorem

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence $T(n) = aT(\frac{n}{b}) + f(n)$, $T(n) = d$ $(n \leq 1)$, $\inf_n f(n) > 0$.

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

Then $T(n) \in$
- $\mathcal{O}(f(n))$ if $f(n)/n^{x+\varepsilon}$ is increasing for some $\varepsilon > 0$
- $\mathcal{O}(n^x \log n)$ if $f(n) \in \mathcal{O}(n^x)$
- $\mathcal{O}(n^x)$ if $f(n) \in O(n^{x-\varepsilon})$ for some $\varepsilon > 0$

Example: $T(n) = 3T(n/4) + n \log n$

- $x = \log_4 3 < 1$, so $n^x \in o(f(n))$: can only be in the first case, still have to check
- set $\varepsilon = 1 - x > 0$, then $f(n)/n^{x+\varepsilon} = \log(n)$, increasing

$\therefore$, $T(n) \in \mathcal{O}(n \log n)$

Example: $T(n) = 2T(n/2) + n \log n$

- $x = 1$ so $n^x \in o(f(n))$: can only be in first case, still have to check

- $\forall \varepsilon > 0$, $f(n)/n^{x+\varepsilon} = \log n / n^{\varepsilon}$, never increasing

$\therefore$ so we can't say $\in$

Linear Time Median

- **Median**: given $A[0, \ldots, n-1]$, find the entry that would be at index $\lfloor \frac{n}{2} \rfloor$ if A was sorted
- **Selection**: given $A[0, \ldots, n-1]$ and k in $\{0, \ldots, n-1\}$, find the entry that would be at index k if A was sorted

→ reminder on quick-select:

```
quick-select (A, k)
A: array of size n,  k: integer s.t. 0≤k<n
1. p = choose-pivot(A)
2. i, j = partition(A, p)
3. if (i≤k<j) { return p }
4. else if (k<i) { return quick-select(A[0,..,i-1], k)
5. else if (j<i) { return quick-select(A[j+1,..,n-1], k-j-1)
```

partition(A, p): reorders A so that $[<p, A[i]=p, \ldots, A[j]=p, >p]$ in linear time!

Goal: find a pivot such that both i and $n-j-1$ aren't too large

Idea: split A into **groups of 5** elements $G_1, G_2, \ldots, G_{n/5}$.
find the **medians** $m_1, m_2, \ldots, m_{n/5}$ **of each group** $G_1, \ldots, G_{n/5}$
and make array $A' = [m_1, m_2, \ldots, m_{n/5}]$.
recursively call quick-select to find the **median element** p **of** A'.

Visualisation: $A = [\underbrace{11, 9, 12, 3, 8}_{G_1}, \underbrace{1, 7, 2, 13, 4}_{G_2}, \ldots]$, so $A' = [9, 4, \ldots]$ (m_1, m_2, $|A'| = n/5$)

now, quick-select(A', $k' = n/5 \cdot \frac{1}{2} = n/10$) → $n/10$ is median of A'

quick-select-using-median-of-medians (A, k)

1. divide A into $n/5$ groups $G_1, \ldots, G_{n/5}$ of size 5
2. find the medians $m_1, \ldots, m_{n/5}$ of each group. call this array A' $\Theta(n)$
3. recursively run select to find the median element p of A' $T(n/5)$
4. i = j = partition(A, p) $\Theta(n)$
5. if (i == k) { return p }
6. else if (k < i) { return select(A[0, 1, ..., i-1], k)
7. else if (k > i) { return select(A[i+1, ..., n-1], k-i-1)

Claim: with this choice of p, the indices i and n-i-1 are at most $7n/10$

Proof:

- half of the m_i's are $\geq p$
- for each m_i, there are 3 elements in G_i that are $\geq m_i$
- so at least $3n/10$ elements are $\geq p$
- so at most $7n/10$ elements are $< p$
- so i is at most $7n/10$. same for n-i-1.

↳ **visualisation:**

medians →

G_1	G_2		
3	1	...	
8	2	...	
9	4	... p	...
11	7	...	
12	13	...	

→ each $n/10$ elements provides 3 elements that are also $\leq p$, so $3n/10$ elements $\leq p$.

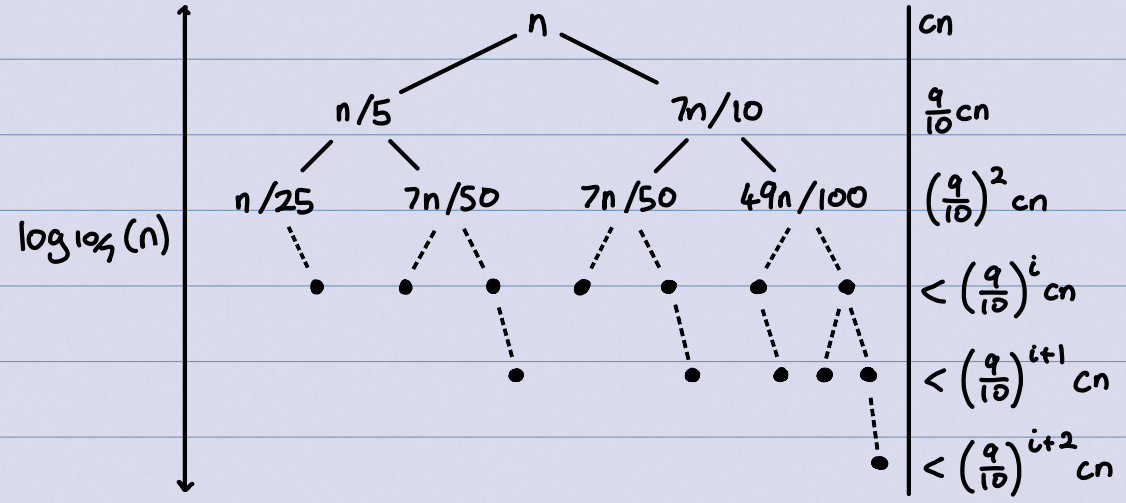
→ $n/10$ elements here that are $\leq p$

→ same logic applied for $\geq p$ as well

Sloppy recurrence: $T(n) = T(n/5) + T(7n/10) + cn$

Claim: this gives worst-case $T(n) \in \Theta(n)$

The recursion tree for $T(n)$ (set $T(n)=0$ for $n \le 1$)



Key point: $T(n) < cn \times$ (geometric sum of ratio $9/10 < 1$), so $T(n) \in O(n)$

Why not median of groups of 3?

- we do $n/3$ groups of 3 and find their medians $m_1, \ldots, m_{n/3}$ $O(n)$
- p is the median of $[m_1, \ldots, m_{n/3}]$ $T(n/3)$
- half of the m_i's are $\ge p$ $n/6$
- in each group, 2 elements are $\le m_i$
- so overall at least $n/3$ elements are $\le p$
- so at most $2n/3$ elements are $< p$
- so $i \le 2n/3$ and $n-i-1 \le 2n/3$
- recurrence is $T(n) \in T(n/3) + T(2n/3) + cn$, which gives $T(n) \in \Theta(n \log n)$.

Closest Pairs

Goal: given n points (x_i, y_i), find a pair (i, j) that minimizes the distance $d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

(equivelent to minimizing $d_{i,j}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$

Assumption: all x_i's are distinct

Idea: separate the points into two halves L, R at the median x-value.

↳ L = all $n/2$ points with $x \leq x_{median}$ (no other point on the median line)

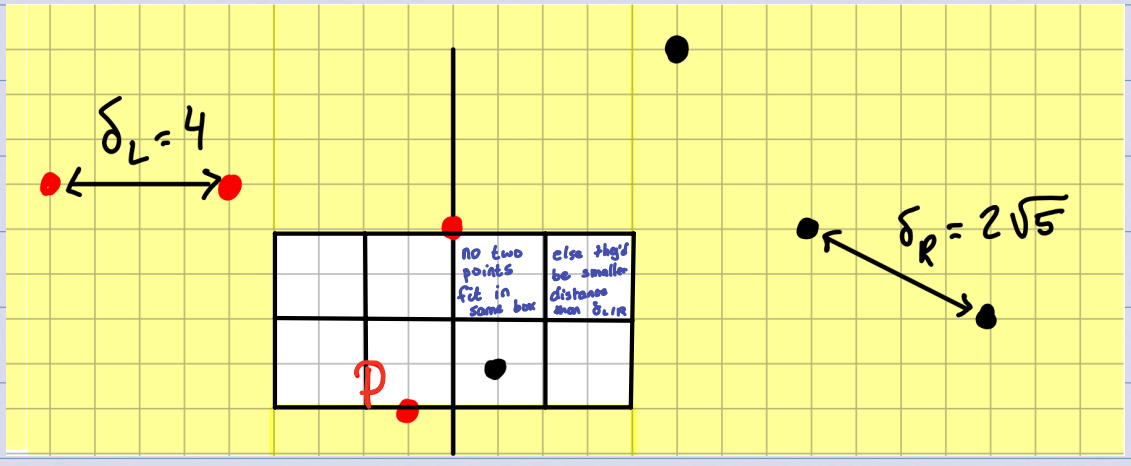
R = all $n/2$ points with $x > x_{median}$

find the closest pair in both L and R recursively

the closest pair is either between points in L (done), between points in R (done), or transverse (one in L, one in R).

Set $\delta = \min(\delta_L, \delta_R)$, since we only need to consider transverse pairs (P, Q) with dist(P, R) $\leq \delta$ and dist(Q, L) $\leq \delta$

for any $P = (x_p, y_p)$, enough to look at points with $y_p \leq y \leq y_p + \delta$



so it's enough to check distances $d(P, Q)$ for Q in the rectangle.

Claim: there are 8 points from our initial set (including P) in the rectangle

Proof: cover the rectangle with 8 squares of side length $\delta/2$

- they overlap along lines, but that's fine
- a square on the left contains at most one point from L
- a square on the right contains at most one point from R

Consequence: at most 8 points in the range $y_p \leq y \leq y_p + \delta$

initialisation: sort the points twice, with respect to x & y

↳ one-time cost $O(n \log n)$, before recursive calls

main algorithm: given two arrays representing the same points (A_x sorted in x, A_y sorted y)

- find the x-median using A_x $O(1)$
- for the next recursive calls, split both A_x and A_y $O(n)$
- remove the points at distance $\geq \delta$ from the x-median line from A_y $O(n)$
- inspect all remaining points P in A_y (= in increasing y-order) for each P, compute the distance to the points Q in A_y with $y_p \leq y_Q \leq y_p + \delta$ and keep the min
  ↳ at most 8 points need to be checked per P $O(n)$

→ runtime: $T(n) = 2T(n/2) + cn$ so $T(n) \in O(n \log n)$

Greedy Algorithms

Context: we're trying to solve a combinatorial optimisation problem:

- have a large, but finite, set S
- want to find an element E in S that minimizes / maximises a cost function

Greedy Strategy:

- build E step by step
- don't think ahead, just try to improve as much as you can every step
- simple algorithms, but often hard to prove correctness

A Recurrent Proof Pattern

- Let E_{Greedy} be the greedy solution
- Let E be any other solution

- Transform E into E_{Greedy} progessively, making sure that the cost never increases.

Example: Huffman

Review from CS240: the Huffman Tree

- we are given "frequencies" $f_1, \dots, f_n$ for characters $c_1, \dots, c_n$
- we want a code (character $c_i \mapsto$ word w_i in $\{0, 1\}$)
- want prefix-free: build a binary tree

Greedy Strategy: we build the tree bottom-up.

- create n single-letter trees
- define the frequency of a tree as the sum of the frequencies of the letters in it
- build the final tree by putting together smaller trees: join the two trees with the least frequencies

→ Claim: this minimizes $\sum_i f_i \times \{\text{length of } w_i\}$

Minimizing Completion Time

The Problem

- Input: n jobs, with processing times $[t(1), \dots, t(n)]$
- Output:
  - an ordering of the jobs that minimizes the sum T of the completion times
  - completion time: how long it took (since the beginning) to complete a job.

Example: n = 5, processing times [2, 8, 1, 10, 5]

- in this order, $T = 2 + (2+8) + (2+8+1) + (2+8+1+10) + (2+8+1+10+2) = 70$
- in the order [1,2,8,5,10], $T = 1 + (1+2) + (1+2+8) + (1+2+8+5) + (1+2+8+5+10) = 57$
- in the order [1,2,5,8,10], $T = 1 + (1+2) + (1+2+5) + (1+2+5+8) + (1+2+5+8+10) = 54$

Greedy Algorithm: order the jobs in non-decreasing processing times

To prove correctness:

- Let $L = [e_1, \ldots, e_n]$ be a permutation of $[1, \ldots, n]$
- Suppose that we don't have $t(e_1) \leq t(e_2) \leq \ldots \leq t(e_n)$
- Then we can find a better permutation L' by removing an inversion

1. by assumption there exists i such that $t(e_i) > t(e_{i+1})$
2. sum of the completion times of L is $nt(e_1) + (n-1)t(e_2) + \ldots + t(e_n)$
3. the contribution of e_i and e_{i+1} is $(n-i+1)t(e_i) + (n-i)t(e_{i+1})$
4. now, swap e_i and e_{i+1} to get a permutation L'
   their contribution becomes $(n-i+1)t(e_{i+1}) + (n-i)t(e_i)$
5. nothing else changes so $T(L') - T(L) = t(e_{i+1}) - t(e_i) < 0$

Interval Scheduling

The Problem

- Input: n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ (start time, finish time)
  - ↳ also write $s_j = \text{Start}(I_j)$, $f_j = \text{finish}(I_j)$.
- Output:
  - a choice T of intervals that do not overlap and that has maximum cardinality
  - $\text{finish}(I_j) = \text{Start}(I_k)$ not an overlap.

Example: a car rental company has the following requests for a day:
I_1: 2pm to 8pm, I_2: 3pm to 4pm, and I_3 = 5pm to 6pm

→ Optimum is $T = [I_2, I_3]$

Attempt 1: pick the interval with the earliest starting time that creates

no conflicts

→ doesn't work (previous example is a counterexample)

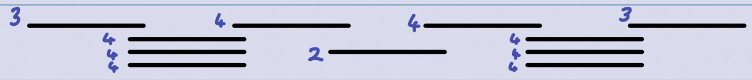
Attempt 2: pick the shortest interval that creates no conflict

→ doesn't work → for example:

Attempt 3: pick the interval with the fewest overlaps that creates no conflicts

→ doesn't work → for example:

conflicts labelled:



so this strategy would output:

but a better schedule is simply:

Attempt 4: pick the interval with the earliest finish time that creates no conflict

↳ A $O(n \log n)$ implementation:

Greedy Scheduler ($I = [I_1, \ldots, I_n]$)

1. $T = []$
2. sort I by non-decreasing finish time
3. for ($k = 1 \to n$) {
4. if I_k doesn't overlap the last entry in T, append I_k to T
5. }
6. return T

To prove correctness:

- Let $T = [x_1, \dots, x_p]$ be the intervals chosen by algorithm
- Let $S = [y_1, \dots, y_q]$ be an optimal solution (sorted by increasing finish time) that has the maximum number of common jobs with T at the beginning: ie, $y_i = x_i$ for $i \in [k]$ for some max k.
- want to prove $p \geq q$

Proof by contradiction: assume $S = [x_1, x_2, \dots, x_k, y_{k+1}, \dots, y_q]$, with $y_{k+1} \neq x_{k+1}$ (for some max $k \geq 0$, $k < q$)

- since $\text{finish}(x_{k+1}) \leq \text{finish}(y_{k+1})$, we can replace y_{k+1} with x_{k+1} in S' that still has q jobs (so it's optimal) but one extra common job with T.
- Contradiction, as we assumed that k is the max number of common jobs with T

Interval Coloring

The Problem

- Input: n intervals $I_1 = [s_1, f_1], \dots, I_n = [s_n, f_n]$ (same as Interval Scheduling)
- Output:
  - assignment of colors to each interval
  - overlapping intervals get different colors
  - minimize the number of colors used overall

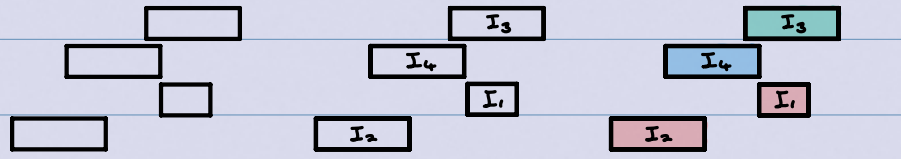
Attempt 1: sort intervals by non-decreasing finish times

↳ for $j = 1, \dots, n$, use for I_j the smallest existing color (smallest index) that creates no conflicts, or ce new color if needed.

→ not optimal !

Attempt 2: sort intervals from **shortest to longest**

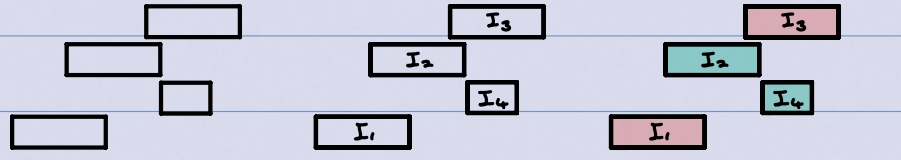
↳ for $j=1, \dots, n$, use for I_j the smallest existing color (smallest index) that creates no conflict, or a new color if needed



→ not optimal!

Attempt 3: sort intervals by **non-decreasing start times**

↳ for $j=1, \dots, n$, use for I_j the smallest existing color (smallest index) that creates no conflicts, or ce new color if needed.



→ maybe optimal, needs proof!

Correctness of Attempt 3:

Claim: suppose the **greedy output uses k colors**. Then, **no other schedule can use fewer than k colors!**

Proof:

- Suppose we color I_t with color k
- So I_t overlaps with k-1 intervals, say $I_{a_1}, \dots, I_{a_{k-1}}$ seen previously
- because we sorted by start time, for all $j=1, \dots, k$, $S_{a_j} \leq S_t \leq f_{a_j}$
- so at time S_t, **we can't do with less than k colors**

Remark: could also do a proof closer in spirit to the previous ones:

- $T = [c_1, \dots, c_n]$ are the colors chosen by the greedy algorithm
- $S = [d_1, \dots, d_n]$ are the colors chosen by any other feasible choice
- prove that S uses more (or same) number of colors by transforming it into T progressively.

↳ A **$\Theta(n \log n)$** implementation

Greedy Interval Colorer

1. sort the array $A = [[s_i, \text{"start"}, i]]_{i,\dots,n}$ cat $[[f_i, \text{"finish"}, i]]_{i,\dots,n}$ by time (to break ties, finish comes before start)
2. $C[1,\dots,n]$ = array (of color indices)
3. H = min-heap of available color indices, initially empty
4. $k = 0$
5. for all entries of A (in increasing order) {
6. if (interval i starts) { set $C[i]$ = min element in H (if not empty) or $k{+}{+}$ (if empty) }
7. if (interval i ends) { insert $C[i]$ in H }
8. }
9. return C

↳ remark: picking any available color would work too.

Minimizing Lateness

The Problem:

Input:

- Jobs $J_1, \dots, J_n$ with processing times $t(1), \dots, t(n)$ and deadlines $d(1), \dots, d(n)$
- Can only do one thing at a time

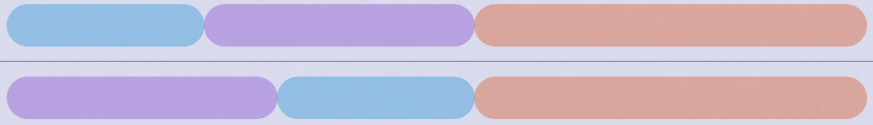
Output:

- a scheduling of jobs which minimizes maximal lateness
  - Job J_i starts at time $s(i)$ and finishes at $f(i) = s(i) + t(i)$
  - if $f(i) \geq d(i)$, lateness $\ell(i) = f(i) - d(i)$, otherwise 0
- maximal lateness: $\max(\ell(i))$.

Example: 3 jobs

- J_1: needs $t(1) = 4$ hours with deadline $d(1) = 2$ hours

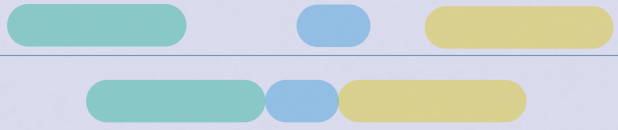
- J_2: needs $t(2) = 6$ hours with deadline $d(2) = 1$ hour
- J_3: needs $t(3) = 10$ hours with deadline $d(3) = 24$ hours



1, then 2, then 3: lateness $[2, 9, 0]$

2, then 1, then 3: lateness $[8, 5, 0]$ (optimal)

Observation: if a scheduling has idle time, we can improve it by removing the breaks:



↳ so, the optimal has no idle time, and is given by some permutation of $\{1, \dots, n\}$.

Attempt 1: do short jobs first

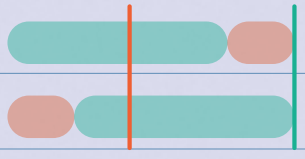
↳ doesn't work! counterexample: the above 3 jobs example

Attempt 2: do jobs with little slack first (slack = $d(i) - t(i)$)

↳ doesn't work!

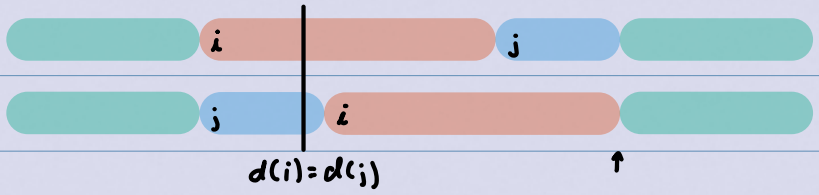
Counterexample: take $t(1) = 8$, $d(1) = 10$, so $s(1) = 2$.

$t(2) = 2$, $d(2) = 5$, so $s(2) = 3$.



Attempt 3: do jobs in non-decreasing deadline order

Non-Uniqueness (observation): if $d(i) = d(j)$, the orderings $[\dots, i, j, \dots]$ and $[\dots, j, i, \dots]$ have the same max lateness (since the second job is the latest).



Proof:

- Let $L = [e_1, \ldots, e_n]$ be any permutation of $\{1, \ldots, n\}$
- Suppose that L is not in decreasing order of deadlines
- Build L' with $\text{MaxLateness}(L') \leq \text{MaxLateness}(L)$ by removing an inversion.

↳ no inversion $\Leftrightarrow$ L is non-decreasing deadline order

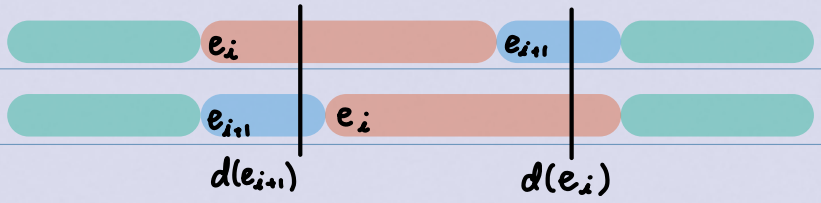
→ there exists i such that $d(e_i) > d(e_{i+1})$

swap e_i and e_{i+1} to get a permutation L'

new lateness of $e_{i+1} \leq$ old lateness of e_{i+1} (since we now do e_{i+1} earlier)

new lateness of $e_i \leq$ old lateness of e_{i+1}

no other change: $\text{MaxLateness}(L') \leq \text{MaxLateness}(L)$ and we removed an inversion!



After at most $n(n-1)/2$ steps, we have L_{final} with no inversions and such that $\text{MaxLateness}(L_{final}) \leq \text{MaxLateness}(L)$

↳ But we know that $\text{MaxLateness}(L_{final}) = \text{MaxLateness}(L_{greedy})$, so done!

Fractional Knapsack

The Problem:

Inputs:

- Items $I_1, \ldots, I_n$ with weights $w_1, \ldots, w_n$ and positive values $v_1, \ldots, v_n$
- a capacity W

Output

- fractions $F = e_1, \ldots, e_n$ such that:

1. $0 \le e_j \le 1$ for all j
2. $e_1 w_1 + \dots + e_n w_n \le W$
3. $e_1 v_1 + \dots + e_n v_n$ maximal

Example: $w_1 = 10$, $v_1 = 60$, $w_2 = 30$, $v_2 = 90$, $w_3 = 20$, $v_3 = 100$, $W = 50$

↳ optimal is $e_1 = 1$, $e_2 = 2/3$, $e_3 = 1$, total value 220.

Remark: the knapsack should be full.

↳ if $\sum_i w_i < W$, just take all $e_i = 1$. So, assume $\sum_i w_i \ge W$

Observation: suppose we have an assignment with $\sum_i e_i w_i < W$

↳ then some e_i must be less than one

So, we can increase the value by increasing this e_i

Consequence: it's enough to consider assignments with $\sum_i e_i w_i = W$.

Attempt 1: pack items in decreasing value v_i

↳ doesn't work! Counterexample is previous example (we get $[0, 1, 1]$ with total 190)

Attempt 2: pack items in increasing weight w_i

↳ doesn't work!

Counterexample: $w_1 = 10$, $v_1 = 100$, $w_2 = 5$, $v_2 = 1$, $W = 10$.

Attempt 3: pack items in non-increasing "value per kilo" v_i / w_i.

↳ first example $[6, 3, 5]$, second example $[10, 1, 5]$

Pseudocode:

```
GreedyKnapsack(v, w, W)
1. E = [0, ..., 0]
2. sort items by non-increasing order of v_i / w_i
```

```
3. for (k=1→n) {
4.   if (w_k < W) {
5.     E[k]=1
6.     W= W-w_k
7.   } else {
8.     E[k] = W/w_k
9.     return
10.  }
11. }
```

Remark: output is $E = [1, \dots, 1, e_k, 0 \dots 0]$ Runtime: $\mathcal{O}(n \log n)$

Proof:

- Let $E = [e_1, \dots, e_n]$ be the greedy output and $S = [s_1, \dots, s_n]$ be any assignment with $\sum_i s_i w_i = W$.
- If $S \neq E$, build S' with one more common entry with E than S, and $val(S) \leq val(S')$

- let i be the first index with $e_i \neq s_i$, so (greedy strategy) $e_i > s_i$
- there is $j > i$ with $s_j > e_j$ (because weight W for both)
- set $s'_i = s_i + \beta$ and $s'_j = s_j - \gamma$, for $\beta = e_i - s_i > 0$, all other $s'_k = s_k$ and such that $weight(S') = W$. So, $\beta w_i = \gamma w_j$.
- Then, $val(S') = val(S) + \beta v_i - \gamma w_j$. Since $v_i/w_i > v_j/w_j$, it follows that $val(S) \leq val(S')$

↳ after at most n steps, we have $S''' \dots = E$ and $val(s) \leq val(E)$.

Dynamic Programming Warmup Example: Fibonacci Numbers

A slow, recursive algorithm: ($F_0 = 0$, $F_1 = 1$, $F_n = F_{n-2} + F_{n-1}$ $\forall\, n \geq 2$).

```
Fib(n)
1. if (n==0) return 0;
2. if (n==1) return 1;
3. return Fib(n-1) + Fib(n-2);
```

Formally speaking, in the unit cost model, the input size is always one

so we'll just measure the runtime as a function of n

→ Assuming we count additions at unit cost, runtime is $T(0)=T(1)=1$, and $T(n) = T(n-1) + T(n-2) + 1$.

This gives $T(n) = F(n+1)-1$, so $T(n) \in \Theta(\varphi^n)$, $\varphi = \frac{1+\sqrt{5}}{2}$

Observations: to compute F_n, we need the values of $F_0, \dots, F_{n-1}$, and recompute them many, many times

Improved Recursive Algorithm:

```
let T=[0, 1, •, •, ...] be a global array
Fib(n)
1. if (T(n) == •) T(n) = Fib(n-1) + Fib(n-2)
2. return T(n)
```

Iterative version:

```
Fib(n)
1. let T=[0, 1, •, •, ...]
2. for (i=2→n) { T[i] = T[i-1] + T[i-2]; }
3. return T[n];
```

Iterative version (better, not always feasible):

```
Fib(n)
1. (u,v) = (0, 1)
2. for (i=2→n) { (u,v) = (v, u+v); }
```

```
3. return v;
```

→ all these improved versions use O(n) additions

↳ Main feature: solve problems "bottom-up" & store solutions if needed

Dynamic Programming

Key features:

- Solve problems through recursion
- Use a small (polynomial) number of nested subproblems
- May have to store results for all subproblems
- Can often be turned into one (or more) loops

Dynamic Programming vs. Divide & Conquer

- Dynamic Programming usually deals w/ all input sizes 1, ..., n
- Divide & Conquer may not solve "subproblems"
- Divide & Conquer algorithms not always easy to rewrite iteratively

Dynamic Programming Recipe

- Identify Subproblems (and typically) store their solutions in an array

  Need to know:
  - dimensions of the array
  - what precisely the array stores
  - where the answer will be found
- Establish Recurrence

  how do small subproblems contribute to the solution of a larger one?
- Find the base cases(s)

- Specify the order of computation
- Recovery of the solution
  - Traceback strategy to determine the final solution

Weighted Interval Scheduling

The Problem:

Inputs:

start time, finish time

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$
- each interval has a weight w_i
- $\text{finish}(I_j) = \text{start}(I_k)$ not an overlap

Output:

- A choice T of intervals that do not overlap and maximise $\sum_{i \in T} w_i$.
- greedy algorithm in the case $w_i = 1$, other natural choice is $w_i = \text{length}(I_i)$

Example: a car rental company has the following requests for a given day: $I_1 = [2, 8]$, $w_1 = 6$, $I_2 = [2, 4]$, $w_2 = 2$, $I_3 = [5, 6]$, $w_3 = 1$, $I_4 = [7, 9]$, $w_4 = 2$.

↳ Answer is $T = [I_1]$, $W = 6$.

Sketch of the Algorithm

Basic Idea: either we choose I_n or not.

- then the optimum $OPT(I_1, \ldots, I_n)$ is the max of two values:
- $w_n + OPT(I_{m_1}, \ldots, I_{m_s})$, if we choose I_n, where $I_{m_1}, \ldots, I_{m_s}$ are the intervals that do not overlap with I_n
- $OPT(I_1, \ldots, I_{n-1})$, if we don't choose I_n

→ In general, we don't know what $I_{m_1}, \ldots, I_{m_s}$ look like

Goal:

- Find a way to ensure that $I_{m_1}, \ldots, I_{m_s}$ are of the form $I_1, \ldots, I_s$, for some $s < n$ (and so on for all indices $< n$)
- Then it suffices to optimise over all $I_1, \ldots, I_j$ $j = 1, \ldots, n$

The Indices P_j

Assume $I_1, \ldots, I_n$ sorted by increasing end time: $f_i \leq f_{i+1}$

Claim: for all j, the set of intervals $I_k \subseteq I_j$ that do not overlap I_j is of the form $I_1, \ldots, I_{p_j}$ for some $0 \leq p_j < j$

↳ $p_j = 0$ if no such interval

↳ The algorithm will need the p_j's

- for a given j, find where s_j would be in $[f_1, \ldots, f_n]$
- precisely: p_j is the last index i such that $f_i \leq s_j$
- binary search, so $O(\log n)$ total

Main Procedure

Definition: $M[j]$ is the maximal weight we can get with intervals $I_1, \ldots, I_j$

Subproblem: given input intervals $I_1, \ldots, I_j$, compute $M[j]$.

Number of Subproblems: n

Recurrence: $M[0] = 0$ and for $j \geq 1$, $M[j] = \max\{M[j-1], M[p_i] + w_j\}$

The Algorithm: bottom-up DP

WeightedIntervalScheduling $(n, [s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n])$

1. sort the finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$
2. compute $p(1), p(2), \ldots, p(n)$
3. $M[0] = 0$
4. for $(i = 2 \to n)$ { $M[i] = \max\{M[i-1], M[p_i] + w_i\}$

↳ Runtime: $O(n \log n)$ (sorting p_j's) and $O(n)$ (finding the $M[j]$'s)

Recovering the Optimal Set

so far: we computed the value $M[n]$ of the optimal set

final goal: recover the optimal set itself! Use backtracking:

```
FindOptimalSet(i)
1. if (i = 0) output nothing
2. else if (M[p_i] + w_i > M[i-1]) {
3.     print i;
4.     FindOptimalSet(p(i));
5. } else {
6.     FindOptimalSet(i-1)
7. }
```

↳ runtime time for backtracking: $O(n)$

Recap: general framework for solving DP problems

- First, compute the value of the optimal solution
- For that, define the subproblems. Their number should be small, ie, polynomial in the input size. There should also be a natural ordering of the subproblems
- Show that computing opt of original problem reduces to computing opt of subproblems. Think recursion! Compute opt bottom-up
- Finally, obtain the items in the solution by backtracking.

0/1 Knapsack

The Problem

Input:

- items $1, \dots, n$ with integer weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$
- an integer capacity W

Output

- a choice of items $S \subseteq \{1, \dots, n\}$
- that satisfies the constraint $\sum_{i \in S} w_i \leq W$
- and maximises the value $\sum_{i \in S} v_i$

→ note: this is just fractional knapsack if we can't divide items.

Example: $v_1 = 2,\ v_2 = 3,\ v_3 = 1,\ v_4 = 5$

$w_1 = 3,\ w_2 = 4,\ w_3 = 6,\ w_4 = 5,\ W = 8$

↳ optimum $S = \{1, 4\}$ with weight 8 and value 7.

Setting Up the Recurrence

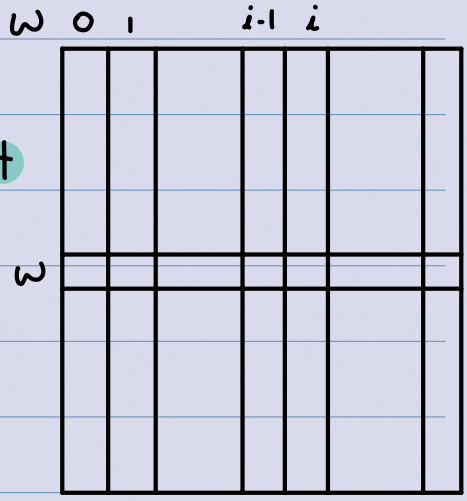
prove by saying I feasible if and only if either a_n in or not, take max value

↳ A pitfall: subproblems $O(j)$?

The Right Subproblem: set $O[w, i]$:= maximum value acheivable using a knapsack of capacity w, items $1, \dots, i$

$po(nw) = c(n+i)(w+i)$

Want: $O[W, n]$



Basic Idea: either we choose item n, or not

- then the optimum $O[W, n]$ is the max of:
- $v_n + O[W - w_n, n-1]$, if we choose n (requires $w_n \leq W$)
- $O[W, n-1]$, if we don't choose n

Initial Conditions: $O[0,i]=0 \ \forall i$, $O[w,0]=0 \ \forall w$.

Algorithm:

0/1Knapsack($V_1, \dots, V_n, w_1, \dots, w_n, W$)

1. initialise an array $O[0,\dots,W,0,\dots,n]$ with all $O[0,j]=0$ and all $O[w,0]=0$
2. for ($i=1 \to n$) {
3. for ($w=1 \to W$) {
4. if ($w_i > w$) { $O[w,i] = O[w, i-1]$; }
5. else { $O[w,i] = \max\{V_i + O[w-w_i, i-1], O[w, i-1]\}$; }
6. }
7. }

↳ Runtime $O(nW)$.

Runtime: This is called a **pseudo-polynomial algorithm**

- in our word RAM model, we've been assuming that all V_i's, w_i's, and W fit in a word
- so input size is $O(n)$ words
- but the runtime also depends on the values of the inputs

→ note: still need to recover the optimum set!!

Longest Increasing Subsequence

The Problem:

The Input: an **array** $[1, \dots, n]$ of integers

The Output: a **longest increasing subsequence of A** (or just its length) (does NOT need to be contiguous)

Example: $A = [7, 1, 3, 10, 11, 5, 19]$ gives $[7, 1, 3, 10, 11, 5, 19]$

↳ Remark: there are 2^n subsequences (including an empty one, which doesn't count).

Tentative Subproblems

Attempt 1:

- Subproblems: the length $\ell[i]$ of a longest increasing subsequence of $A[1, .., i]$
- On the example: $\ell[6] = 4$
- so what? not enough to deduce $\ell[7]$.

Attempt 2:

- Subproblems: the length $\ell[i]$ of a longest increasing subsequence of $A[1, \dots, i]$, together with it's last entry
- On the last example: $\ell[6] = 4$, with last element 11
- OK if we add $A[i+1]$, but what if not?

Attempt 3:

- $OPT(i)$ = length of longest increasing subsequence of $A[1, \dots, i]$.
- Let $L[i]$ be the length of a longest increasing subsequence of $A[1, \dots, i]$ that ends with $A[i]$, for $i = 1, \dots, n$.
- so $L[1] = 1$.

↓

Idea: a longest increasing subsequence S ending at $A[i]$ looks like

$$S = [\dots, A[j], A[i]] = S' \text{ cat } [A[i]].$$

↳ S' is a longest increasing subsequence ending at $A[j]$ (or it's empty)

don't know j, but we can try all $j < i$ for which $A[j] < A[i]$

↳ $$OPT(i) = \max_{\substack{j<i \\ A[j]<A[i]}} \{1 + opt(j)\}, \qquad \ell(n) = \max_{i \le n} \{OPT(i)\}$$

Iterative Algorithm

```
Longest Increasing Subsequence (A[1, ..., n])
1.  L[1] = 1
2.  for (i = 2 → n) {
3.      L[i] = 1
4.      for (j = 1 → i-1) {
5.          if (A[j] < A[i]) {
6.              L[i] = max(L[i], L[j]+1)
7.          }
8.      }
9.  }
10. return the maximum entry in L
```

↳ runtime $O(n^2)$.

→ Remark: the algorithm doesn't return the sequence itself, but it could be modified to do so.

Bonus: a faster algorithm

As before, $\ell[i]$ = length of longest increasing subsequence of $A[1, \ldots, i]$

Idea: we consider the "best" increasing sequences in $A[1, \ldots, i]$

- can have several increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, best increasing sequence of length j: one whose last entry is smallest

Example: $A = [2, 8, 10, 11, 1, 3, 5]$, $\ell[6] = 4$, done $i = 6$

- $j = 1$, best increasing sequence $[1]$ — can add 5
- $j = 2$, best increasing sequence $[1, 3]$ — can add 5
- $j = 3$, best increasing sequence $[2, 8, 10]$ — can't add 5
- $j = 4$, best increasing sequence $[2, 8, 10, 11]$ — can't add 5

↓

A = [2, 8, 10, 11, 1, 3, 5], $\ell[6] = 4$, doing $i = 7$

- $j = 1$, best increasing sequence [1] — can add 5
- $j = 2$, best increasing sequence [1, 3] — can add 5
- $j = 3$, best increasing sequence [2, 8, 10] — can't add 5
- $j = 4$, best increasing sequence [2, 8, 10, 11] — can't add 5

$1 < 3 < 5 < 10 < 11$, so $\ell[7] = 4$ and we update the $j = 3$ sequence to [1, 3, 5].

Example: A = [2, 8, 10, 11, 1, 3, 15], $\ell[6] = 4$, done $i = 6$

- $j = 1$, best increasing sequence [1] — can add 15
- $j = 2$, best increasing sequence [1, 3] — can add 15
- $j = 3$, best increasing sequence [2, 8, 10] — can add 15
- $j = 4$, best increasing sequence [2, 8, 10, 11] — can add 15

↓

A = [2, 8, 10, 11, 1, 3, 15], $\ell[6] = 4$, doing $i = 7$

- $j = 1$, best increasing sequence [1] — can add 15
- $j = 2$, best increasing sequence [1, 3] — can add 15
- $j = 3$, best increasing sequence [2, 8, 10] — can add 15
- $j = 4$, best increasing sequence [2, 8, 10, 11] — can add 15

$1 < 3 < 5 < 11 < 15$, so $\ell[7] = 5$, and we have the $j = 5$ sequence [2, 8, 10, 11, 15]

Iterative Algorithm (not quite DP)

- sufficient to store the last entry in the best increasing sequence
- these last entries are increasing ($1 < 3 < 10 < 11$)
- so we can use binary search to find where the new A[i] fits

Longest Increasing Subsequence (A[1, ..., n])

1. $b = [-\infty, \infty, \infty, \ldots, \infty]$, $\ell = 0$ // indexed starting from 0
2. for ($i = 1 \to n$) {

3. find $k \in \{0, \dots, l\}$ s.t. $b[k] < A[i] \leq b[k+1]$
4. $b[k+1] = A[i]$
5. if $(k = l)$ { l++ }
6. }
7. return l

$O(n \log n)$

Longest Common Subsequence

The Problem

The Input: Arrays $A[1, \dots, n]$ and $B[1, \dots, m]$ of characters / integers

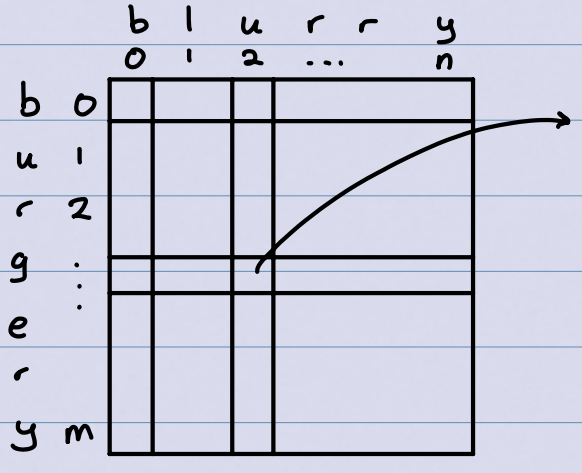
The Output: the maximum length k of a common subsequence to A and B
(subsequences do NOT need to be contiguous)

Example: A = 'blurry', B = 'burgery', longest common subsequence is 'burry'

↳ Remark: there are 2^n subsequences in A, 2^m subsequences in B

$OPT(i, j)$ = length of the longest common subsequence of $A[1, \dots, i]$ and $B[1, \dots, j]$.
$i = 1 \to n$ $j = 1 \to m$

↳ # subproblems: $O(mn)$.



→ length of LCS ('blu', 'burg')

$$\rightarrow OPT(i, j) = \max \begin{cases} OPT(i-1, j) & \text{if } A[i] \neq B[j] \\ OPT(i, j-1) & \\ OPT(i-1, j-1)+1 & \text{if } A[i] = B[j] \end{cases}$$

A Bivariate Recurrence

Definition: let $M[i, j]$ be the length of a longest subsequence between $A[1, \dots, i]$ and $B[1, \dots, j]$

- $M[0, j] = 0 \ \forall j$, $M[i, 0] = 0 \ \forall i$.

- $M[i, j]$ is the max of up to 3 values
  - $M[i, j-1]$ (don't use $B[j]$)
  - $M[i-1, j]$ (don't use $A[i]$)
  - if $A[i] = B[j]$, $1 + M[i-1, j-1]$.

The algorithm computes all $M[i, j]$, using 2 nested loops, so runtime $O(nm)$

↳ Bonus: if $A[i] = B[j]$, no need to consider $M[i, j-1]$ and $M[i-1, j]$

Edit Distance

The Problem

The Input: arrays $A[1, \ldots, n]$ and $B[1, \ldots, m]$ of characters

The Output: minimum number of {add, delete, change} operations that turn A into B

Example: A = snowy, B = sunny

s n o w y	S _ n o w y	_ s n o w y _
s u n n y	s u n n y _	s u n _ _ n y
↳ 3C	↳ 1A, 2C, 1D	↳ 2A, 2C, 2D

The Recurrence

Definition: let $D[i, j]$ be the edit distance between $A[1, \ldots, i]$ and $B[1, \ldots, j]$

- $D[0, j] = j \ \forall j$ (add j characters to empty A)
- $D[i, 0] = i \ \forall i$ (delete i characters from A)
- $D[i, j]$ is the min of 3 values:
  - $D[i-1, j-1]$ (if $A[i] = B[j]$) or $D[i-1, j-1] + 1$ (otherwise)
  - $D[i-1, j] + 1$ (delete $A[i]$ and match $A[1, \ldots, i-1]$ with $B[1, \ldots, j]$
  - $D[i, j-1] + 1$ (add $B[j]$ and match $A[1, \ldots, i]$ with $B[1, \ldots, j-1]$

↳ the algorithm computes all $D[i, j]$ using 2 nested loops, so runtime $O(mn)$.