

Cost of Algorithms

Inputs: parameterized by an integer n , called the size

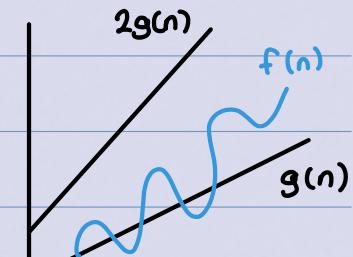
$T(I)$ = runtimes on input $I \Rightarrow$ runtime of a particular instance

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

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

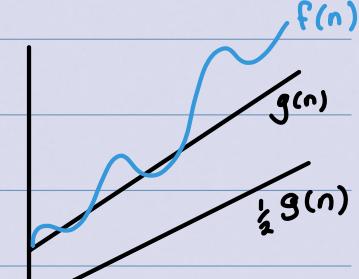
Asymptotic Notation - consider 2 functions $f(n)$ and $g(n)$ with values in $\mathbb{R}_{>0}$

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



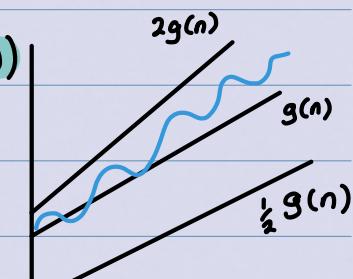
big- Ω : we say that $f(n) \in \Omega(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \geq C g(n)$

\Rightarrow equivalent to $g(n) \in O(f(n))$



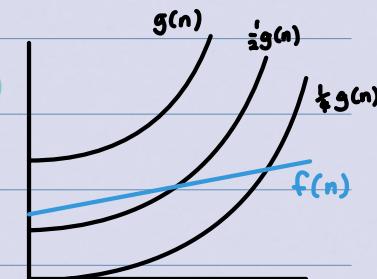
Θ : we say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

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



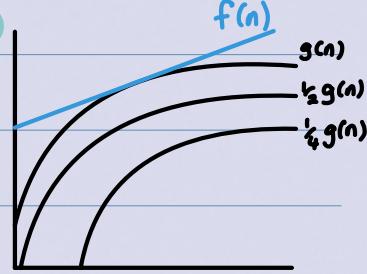
little-O: we say that $f(n) \in o(g(n))$ if for all $C > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \leq C g(n)$

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



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

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



Examples

a) $n^k + C_{k-1} n^{k-1} + \dots + C_0 \in \Theta(n^k)$

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

$f(n, m) \leq C g(n, m)$ for $n \geq n_0$ or $m \geq m_0$

Case study: maximum subarray

Given an array $A[0, \dots, n]$, find a contiguous subarray $A[i, \dots, j]$

that maximises the sum $A[i] + \dots + A[j]$.

Example: given $A = [10, -5, 4, 3, -5, 6, -1, -1]$, the subarray $A[0, \dots, 5] = [10, -5, 4, 3, -5, 6]$ 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 ($i = 0 \rightarrow n$) {
3. for ($j = i \rightarrow n$) {
4. $Sum = 0$
5. for ($k = i \rightarrow j$) {
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 $\Theta(n^2)$

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!

Maximise Lower Half (A)

```
1. opt = A[n/2]
```

Maximise Upper Half (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:

Divide And Conquer Maximum Subarray ($A[0, \dots, n]$)

1. if ($n=1$) return $\max(A[0], 0)$
2. opt_low = DivideAndConquerMaximumSubarray ($A[0, \dots, n/2]$)
3. opt_high = DivideAndConquerMaximumSubarray ($A[n/2+1, \dots, n]$)
4. opt_mid = MaximiseLowerHalf(A) + MaximiseUpperHalf(A)
5. return $\max(\text{opt_low}, \text{opt_mid}, \text{opt_high})$

⇒ 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:

- α recursive calls, in size either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ ($\frac{\alpha}{b} > 1$, constant)
- between $c'n^y$ and Cn^y extra operations ($c, C \neq 0, y$ constant)

Claim: Solving the sloppy recurrence $T(n) = \alpha T(n/b) + cn^y$ 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 $T_{\text{worst}}(n)$, $T_{\text{best}}(n)$ be the worst/best cases in size n .

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

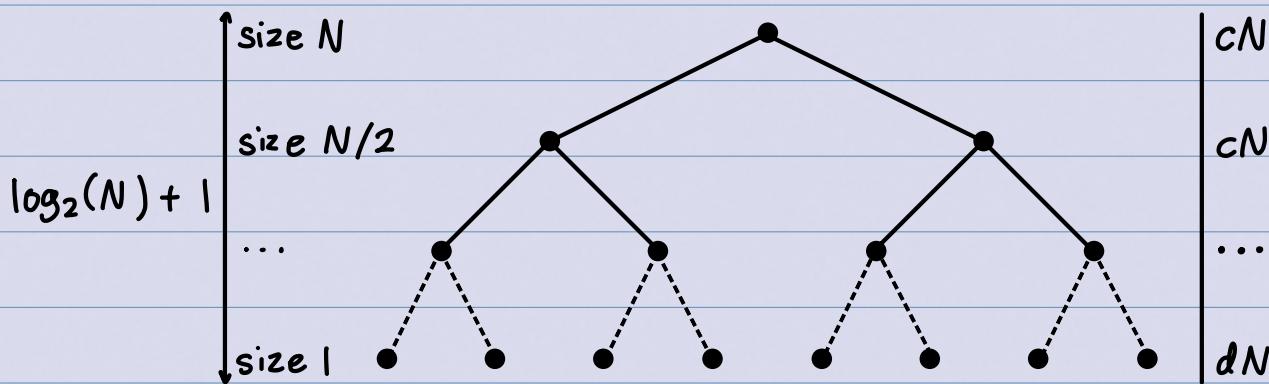
$$T_{\text{worst}}(n) \leq T_{\text{worst}}(\lceil n/2 \rceil) + T_{\text{worst}}(\lfloor n/2 \rfloor) + cn \quad \text{if } n > 1$$

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

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

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: $T_{\text{worst}}(n) \leq KN \log_2(N) \leq K(2n) \log_2(2n) \leq K'n \log_2 n$,
so, $T_{\text{worst}}(n) \in O(n \log n)$

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

The Master Theorem

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

$$T(n) = \alpha T(n/b) + cn^3 \quad (n \geq b), \quad T(1) = d.$$

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

Then, for n

a power of b,

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

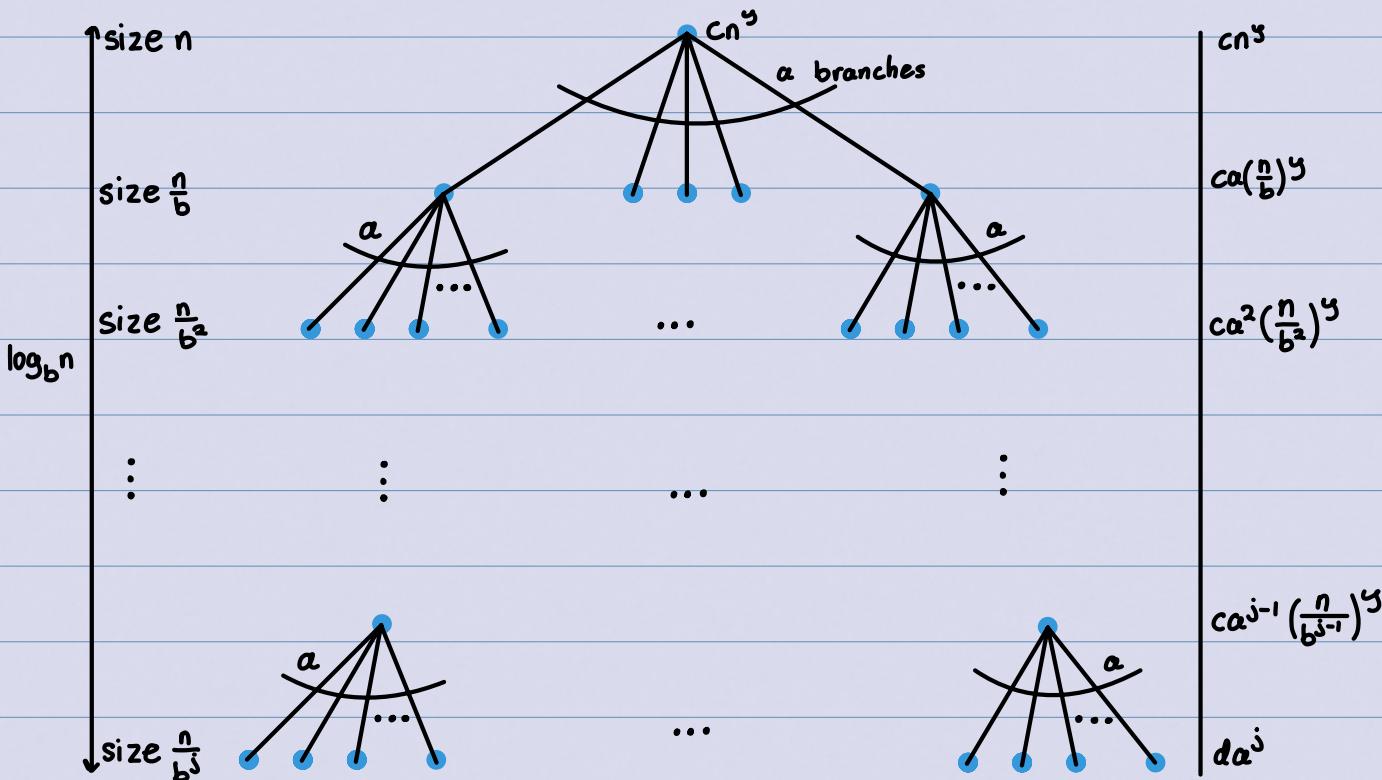
root
 heavy
 balanced
 leaf
 heavy

Recursion Tree

Suppose that $n = b^{\alpha}$, $\alpha \geq 1$, $b \geq 2$ are integers and

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

$$T(n) = \alpha T\left(\frac{n}{b}\right) + cn^y, \quad T(1) = d$$



Breakdown of the Cost

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

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

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	$c(\frac{n}{b})^y$	$ca(\frac{n}{b})^y$
$n/b^2 = b^{j-2}$	a^2	$c(\frac{n}{b^2})^y$	$ca^2(\frac{n}{b^2})^y$
\vdots	\vdots	\vdots	\vdots
$n/b^{j-1} = b$	a^{j-1}	$c(\frac{n}{b^{j-1}})^y$	$ca^{j-1}(\frac{n}{b^{j-1}})^y$
$n/b^j = 1$	a^j	d	da^j
generally: $n/b^i = b^{j-i}$	a^i	$c(\frac{n}{b^i})^y$	$ca^i(\frac{n}{b^i})^y$

Computing $T(n)$:

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

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

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 $r < 1$.

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

if $r=0$, then $b^{\sum_{j=1}^{x-y}} = 1$, so $x=y$.

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

$\therefore T(n) = dn^x + cn^y \times \Theta(\log n) \in \Theta(n^y \log n)$ $\therefore x=y \Rightarrow T(n) \in \Theta(n^y \log n)$

if $r > 1$, then $b^{\sum_{j=1}^{x-y}} > 1$, so $x > y$

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

$\therefore T(n) = dn^x + cn^y \times \Theta(r^j)$. $r^j = \frac{a^j}{b^j} = \frac{n^x}{n^y}$.

$T(n) = dn^x + cn^y \times \Theta(\frac{n^x}{n^y}) \in \Theta(n^x)$. $\therefore x > y \Rightarrow T(n) \in \Theta(n^x)$.

Examples: find Θ -bounds for the following:

1) $T(n) = 4T(\frac{n}{2}) + n$ (multiplying polynomials)

$$a=4, b=2, y=1, \text{ so } x=\log_b a = \log_2 4 = 2.$$

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

2) $T(n) = 2T(\frac{n}{2}) + n^2$ (kd-trees)

$$a=2, b=2, y=2, \text{ so } x=\log_b a = \log_2 2 = 1$$

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

3) $T(n) = 2T(\frac{n}{4}) + 1$ (kd-trees)

$$a=2, b=4, y=0, \text{ so } x=\log_b a = \log_4 2 = \frac{1}{2}$$

$\therefore x=\frac{1}{2} < y=0, \text{ so } T(n) \in \Theta(\sqrt{n})$

4) $T(n) = T(\frac{n}{2}) + 1$ (binary search)

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

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

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: $T(n) = 2T(n/2) + n$, $T(1) = 0$.

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 $T(n/2) \leq kn \log_2(n/2)$

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

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

(proving $T(n) = \dots$ 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

Polyomial 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 $\Theta(n)$, easy algorithm in $\Theta(n^2)$:

Naive Polynomial Multiplication (F, G)

1. for ($i = 0 \rightarrow 2n-2$) {
2. $H_i = 0$
3. }
4. for ($i = 0 \rightarrow n-1$) {
5. for ($j = 0 \rightarrow n-1$) {
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: $T(n) = 4T(n/2) + cn,$
 $\hookrightarrow a=4, b=2, y=1, \text{ so } T(n) \in \Theta(n^2)$
 \therefore , 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 $F = F_0 + F_1 x$, $G = G_0 + G_1 x$, $H = F_0 G_0 + ((F_0 + F_1)(G_0 + G_1) - F_0 G_0 - F_1 G_1)x + F_1 G_1 x^2$

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

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

Toom-Cook:

- A family of algorithms based on similar expressions as

Karatsuba

- for $R \geq 2$, $2R-1$ recursive calls in size n/k
- so $T(n) \in \Theta(n^{\log_k(2R-1)})$
- 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 $\Theta(n\log n)$ operations over \mathbb{C}

Multiplying Matrices

$$\Rightarrow \text{reminder: } \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 6 & 1 \cdot 7 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 6 & 3 \cdot 7 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 17 & 23 \\ 39 & 53 \end{bmatrix}$$

$[i \times j] \cdot [k \times l] \rightarrow$ 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 $\Theta(n^2)$ elements, easy algorithm in $\Theta(n^3)$:

Naive Matrix Multiplication (A, B)

```
1. for (i=0 → n) {  
2.   for (j=0 → n) {  
3.     for (k=0 → n) {  
4.       Ci,k += Aij · Bjk  
5.     }  
6.   }  
7. }
```

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

divide by splitting the matrices into 4 sections:

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

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 $\frac{n}{2} + \Theta(n^2)$ additions which is in $\Theta(n^3)$! no improvement :)

Strassen's Algorithm

Compute:

$$Q_1 = (A_{0,0} - A_{0,1})B_{1,1}$$

$$Q_2 = (A_{1,0} - A_{1,1})B_{0,0}$$

$$Q_3 = A_{1,1}(B_{0,0} + B_{1,0})$$

$$Q_4 = A_{0,0}(B_{0,1} + B_{1,1}) \quad \text{and}$$

$$Q_5 = (A_{0,0} + A_{1,1})(B_{1,1} - B_{0,0})$$

$$Q_6 = (A_{0,0} + A_{1,0})(B_{0,0} + B_{0,1})$$

$$Q_7 = (A_{0,1} + A_{1,1})(B_{1,0} + B_{1,1})$$

$$C_{0,0} = Q_1 - Q_3 - Q_5 + Q_7$$

$$C_{0,1} = Q_4 - Q_1$$

$$C_{1,0} = Q_2 + Q_3$$

$$C_{1,1} = Q_5 + Q_6 - Q_2 - Q_4$$

now, we have 7 instead of 8 recursive calls!

$$= \Theta(n^{2.8}) < \Theta(n^3)!$$

size is $n/2$ with $\Theta(n)$ additions, so $T(n) \geq \Theta(n^{\log_2 7})$

direct generalisation: an algorithm that does k multiplications for matrices of size ℓ gives $T(n) \in \Theta(n^{\log_2 k})$

Counting Inversions

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

Goal: given an unsorted array $A[1, \dots, n]$, 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 $\Theta(n^2)$
3. to do better than n^2 , we cannot list all inversions

towards a divide & conquer algorithm

idea:

- C_L = number of inversions in $A[0, \dots, \frac{n}{2}]$ (first half)
- C_R = number of inversions in $A[\frac{n}{2}+1, \dots, n]$ (second half)
- C_T = number of transverse inversions with $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$ (both halves)
- return $C_L + C_R + C_T$

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

$$C_L = 1 \quad \{(1, 2)\}$$

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

$$C_T = 4 \quad \{(1, 4), (1, 7), (3, 4), (3, 7)\} \rightarrow \text{calculate w/ merge sort}$$

} calculated recursively

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

↳ eg, with $A = [1, 5, 2, 6, 3, 8, 7, 4]$, $C_T = \#_{i < 4} > 3 + \#_{i < 4} > 8 + \#_{i < 4} > 7 + \#_{i < 4} > 6$

or $C_T = \#_{j > 4} < 1 + \#_{j > 4} < 5 + \#_{j > 4} < 2 + \#_{j > 4} < 6$.

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($A[1, \dots, n]$) (both halves of A sorted)

1. copy A into new array S
2. $i=1; j=n/2+1;$
3. for ($k=1 \rightarrow n$) {
4. if ($i=n/2+1$) $A[k] = S[j++]$
5. else if ($j=n+1$) $A[k] = S[i++]$
6. else if ($S[i] \leq S[j]$) $A[k] = S[i++]$
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 $j - j_{\text{init}} = j - (n/2 + 1)$!

Enhanced Merge($A[1, \dots, n]$) (both halves of A sorted)

1. copy A into new array S
2. $i=1; j=n/2+1; C=0;$
3. for ($k=1 \rightarrow n$) {
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 ($S[i] \leq S[j]$) $A[k] = S[i++]; C += j - (n/2 + 1)$
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 $C=C+2$
- $\therefore C_t = 4!$

⇒ 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

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad T(n) = d \quad (n \leq 1), \quad \inf_n f(n) > 0.$$

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

Then $T(n) \in \begin{cases} \mathcal{O}(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is increasing for some } \epsilon > 0 \\ \mathcal{O}(n^x \log n) & \text{if } f(n) \in \mathcal{O}(n^x) \\ \mathcal{O}(n^x) & \text{if } f(n) \in \mathcal{O}(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \end{cases}$

Example: $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

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

Example: $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

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

• $\forall \epsilon > 0$, $f(n)/n^{x+\epsilon} = \log n/n^\epsilon$, never increasing
∴ we can't say it

Linear Time Median

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

1. $p = \text{choose-pivot}(A)$
2. $i, j = \text{partition}(A, p)$
3. if ($i \leq k < j$) { return p }
4. else if ($k < i$) { return quick-select($A[0, \dots, i-1]$, k) }
5. else if ($j < i$) { return quick-select($A[j+1, \dots, n-1]$, $k-j-1$) }

partition(A, p): reorders A so that [$< p, A[i]=p, \dots, A[j]=p, > p$] 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 $G_1, G_2, \dots, G_{n/5}$.
find the medians $m_1, m_2, \dots, m_{n/5}$ of each group $G_1, \dots, G_{n/5}$
and make array $A' = [m_1, m_2, \dots, m_{n/5}]$.
recursively call quick-select to find the median element p of A' .

Visualisation: $A = [11, \underbrace{9, 12, 3, 8}_G, \underbrace{1, 7, 2, 13, 4}_G, \dots]$, so $A' = [9, 4, \dots]$
now, quick-select(A' , $k = \frac{n}{5} \cdot \frac{1}{2} = \frac{n}{10}$) $\Rightarrow \frac{n}{10}$ is median of A'

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

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

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

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 $\frac{3n}{10}$ elements are $\geq p$
- so at most $\frac{7n}{10}$ elements are $< p$
- so i is at most $\frac{7n}{10}$. same for $n-i-1$.

↳ **visualisation:**

G_1	G_2	
3	1	...
8	2	..
medians \Rightarrow	9	4
	11	7
	12	13

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

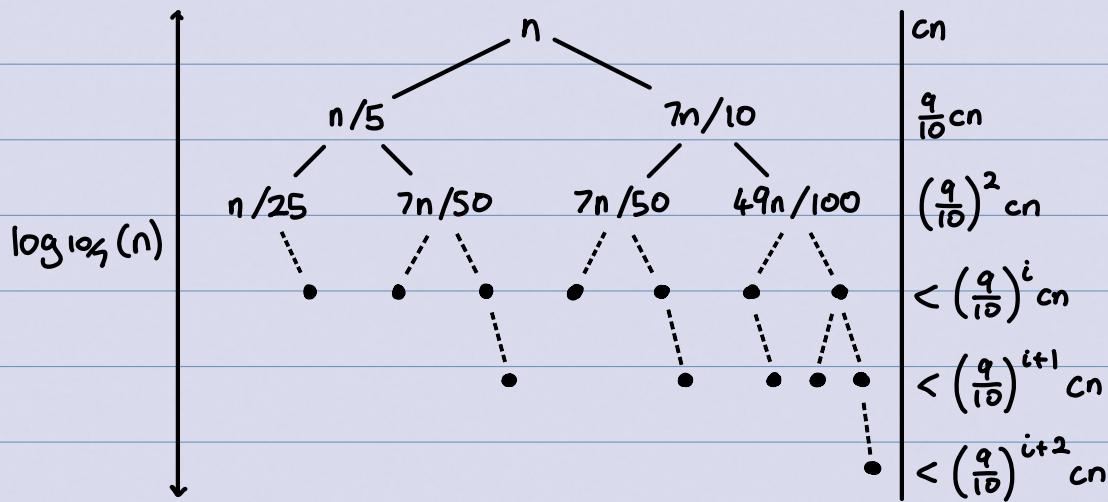
→ $\frac{n}{10}$ elements here that are $\leq p$

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

Sloppy recurrence: $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10}) + cn$

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

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



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

Why not median of groups of 3?

- we do $n/3$ groups of 3 and find their medians $m_1, \dots, m_{n/3}$ $\Theta(n)$
- p is the median of $[m_1, \dots, m_{n/3}]$ $T(n/3)$
- half of the m_i 's are $\geq p$ $n/6$
- in each group, 2 elements are $\geq m_i$
- so overall at least $n/3$ elements are $\geq p$
- so at most $2n/3$ elements are $< p$
- so $i \leq 2n/3$ and $n-i-1 \leq 2n/3$
- 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

$$\text{minimizes the distance } d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

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

Assumption: all x_i 's are distinct

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

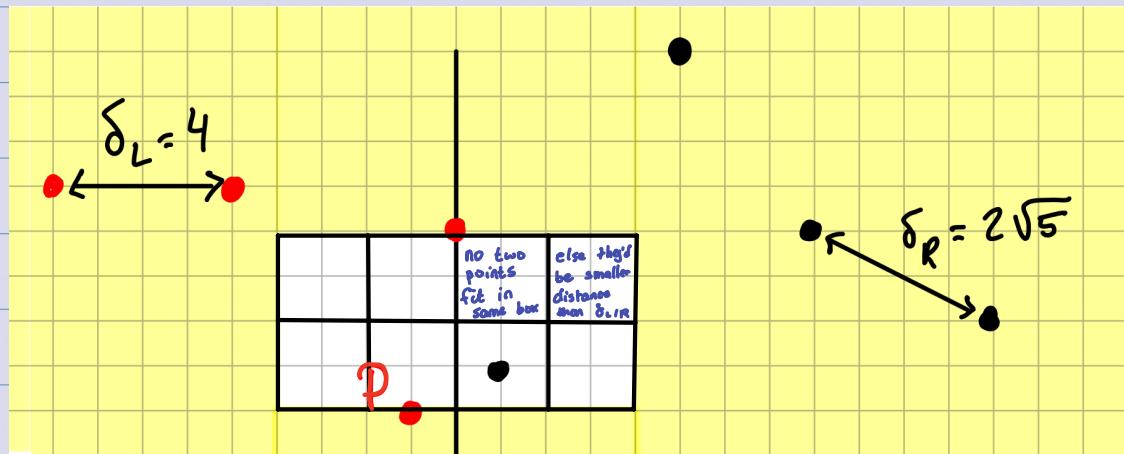
↳ $L = \text{all } n/2 \text{ points with } x \leq x_{\text{median}}$ (no other point on the median line)

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

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 $\text{dist}(P, R) \leq \delta$ and $\text{dist}(Q, L) \leq \delta$ 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 $\Theta(n \log n)$, 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 $\Theta(1)$
- for the next recursive calls, split both A_x and A_y $\Theta(n)$
- remove the points at distance $\geq \delta$ from the x -median line from A_y $\Theta(n)$
- 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 $\Theta(n)$

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

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 / maximizes 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 EGreedy progressively, 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, $T = 2 + (2+8) + (2+8+1) + (2+8+1+10) + (2+8+1+10+5) = 70$
- 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 $L = [e_1, \dots, e_n]$ be a permutation of $[1, \dots, n]$
- Suppose that we don't have $t(e_1) \leq t(e_2) \leq \dots \leq t(e_n)$
- 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 $nt(e_1) + (n-1)t(e_2) + \dots + t(e_n)$
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 $I_1 = [s_1, f_1], \dots, I_n = [s_n, f_n]$ (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:

$I_1: 2\text{pm to } 8\text{pm}, I_2: 3\text{pm to } 4\text{pm}, \text{ and } I_3: 5\text{pm to } 6\text{pm}$

→ Optimum is $T = [I_2, I_3]$

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

no conflicts

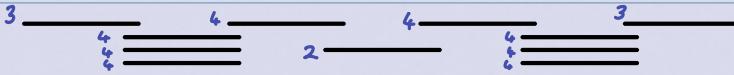
→ doesn't work (previous example is a counterexample)

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 $\Theta(n \log n)$ implementation:

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

1. $T = []$

2. sort I by non-decreasing finish time

3. for ($k=1 \rightarrow n$) {

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

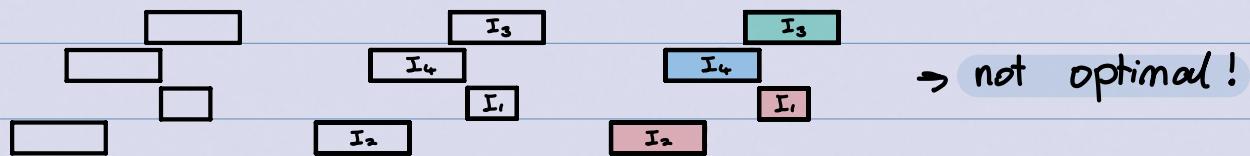
Attempt 1: sort intervals by non-decreasing finish times

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



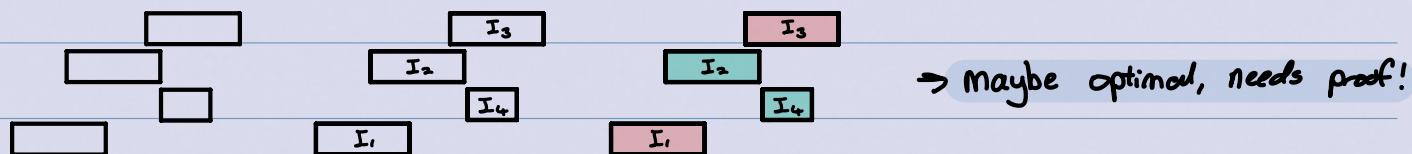
Attempt 2: sort intervals from shortest to longest

↳ 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



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 a new color if needed.



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 $A = [[s_i, "start", i]]_{i=1,\dots,n}$ cat $[[f_i, "finish", i]]_{i=1,\dots,n}$ by time
(to break ties, finish comes before start)
2. $C[1, \dots, n] = \text{array (of color indices)}$
3. $H = \text{min-heap of available color indices, initially empty}$
4. $k = 0$
5. for all entries of A (in increasing order) {
6. if (interval i starts) { set $C[i] = \min \text{ element in } H$ (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

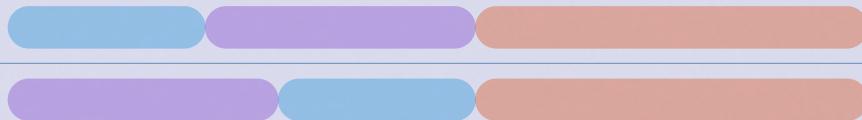
Output:

- 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 $l(i) = f(i) - d(i)$, otherwise 0
- maximal lateness: $\max(l(i))$.

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

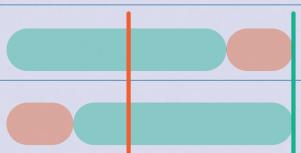
↳ doesn't work! Counterexample: the above 3 jobs example

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

↳ doesn't work!

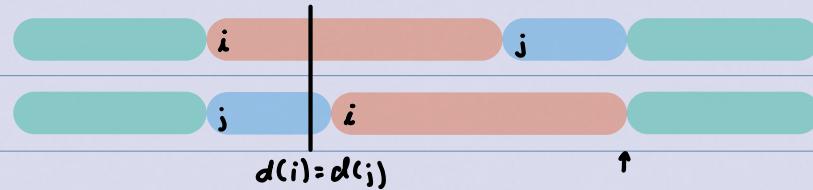
Counterexample: take $t(1)=8$, $d(1)=10$, so $s(1)=2$.

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



Attempt 3: do jobs in non-decreasing deadline order

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



Proof:

- Let $L = [e_1, \dots, e_n]$ be any permutation of $\{1, \dots, n\}$
- Suppose that L is not in decreasing order of deadlines
- Build L' with $\text{MaxLateness}(L') \leq \text{MaxLateness}(L)$ by removing an inversion.
- \hookrightarrow no inversion $\Leftrightarrow L$ is non-decreasing deadline order

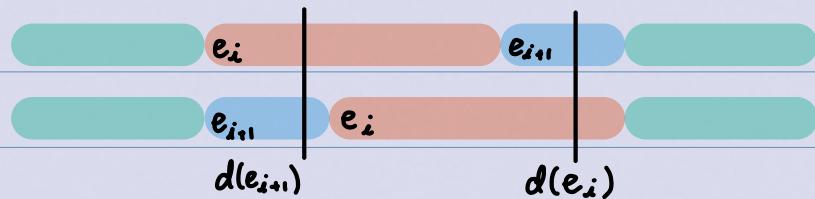
\Rightarrow there exists i such that $d(e_i) > d(e_{i+1})$

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 $\text{MaxLateness}(L_{\text{final}}) \leq \text{MaxLateness}(L)$

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

Fractional Knapsack

The Problem:

Inputs:

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

Output

- fractions $E = e_1, \dots, e_n$ such that:

1. $0 \leq e_j \leq 1$ for all j

2. $e_1 w_1 + \dots + e_n w_n \leq W$

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 $e_1 = 1, e_2 = \frac{2}{3}, e_3 = 1$, total value 220.

Remark: the knapsack should be full.

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

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:

Greedy Knapsack(v, w, W)

1. $E = [0, \dots, 0]$

2. sort items by non-increasing order of v_i/w_i

```

3. for ( $k=1 \rightarrow n$ ) {
4.   if ( $w_k < w$ ) {
5.      $E[k] = 1$ 
6.      $w = w - w_k$ 
7.   } else {
8.      $E[k] = w/w_k$ 
9.   }
10.  }
11. }

```

Remark: Output is $E = [1, \dots, 1, e_k, 0 \dots 0]$ Runtime: $\Theta(n \log n)$

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 $\text{val}(S) \leq \text{val}(S')$
- 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 $\text{weight}(S') = W$. So, $\beta w_i = \gamma w_j$.
- Then, $\text{val}(S') = \text{val}(S) + \beta v_i - \gamma v_j$. Since $v_i/w_i > v_j/w_j$, it follows that $\text{val}(S) \leq \text{val}(S')$

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

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 $T = [0, 1, \dots, \dots]$ be a global array

Fib(n)

1. if ($T(n) == \cdot$) $T(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
2. return $T(n)$

Iterative Version:

Fib(n)

1. let $T = [0, 1, \dots, \dots]$
2. for ($i = 2 \rightarrow n$) { $T[i] = T[i-1] + T[i-2]$; }
3. return $T[n]$;

Iterative Version (better, not always feasible):

Fib(n)

1. $(u, v) = (0, 1)$
2. for ($i = 2 \rightarrow n$) { $(u, v) = (v, u+v)$; }

3. return v;

→ all these improved versions use $\Theta(n)$ 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 $1, \dots, n$
- 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:

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

↑
start time, finish time

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

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

Goal:

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

The Indices P_j

Assume I_1, \dots, 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, \dots, 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 $[f_1, \dots, f_n]$
- 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, \dots, I_j

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

Number of Subproblems: n

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

The Algorithm: bottom-up DP

Weighted Interval Scheduling ($n, [s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$)

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

↳ Runtime: $\Theta(n \log n)$ (sorting p_i 's) and $\Theta(n)$ (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 c_n in or not, take max value

↳ A pitfall: subproblems $O(j)$?

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

Want: $O[W, n]$

w	0	1	$i-1$	i
0				
1				
2				
3				
4				
5				
6				
7				
8				

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 :

O/1 Knapsack ($V_1, \dots, V_n, W_1, \dots, W_n, W$)

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 ($i=1 \rightarrow n$) {
3. for ($w=1 \rightarrow W$) {
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 $\Theta(nW)$.

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 $\Theta(n)$ 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 $l[i]$ of a longest increasing subsequence of $A[1, \dots, i]$
- On the example: $l[6] = 4$
- So what? not enough to deduce $l[7]$.

Attempt 2:

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

Attempt 3:

- $\text{OPT}(i)$ = 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 $A[j] < A[i]$

$$\hookrightarrow \text{OPT}(i) = \max_{\substack{j < i \\ A[j] < A[i]}} \{ 1 + \text{opt}(j) \}, \quad l(n) = \max_{i \leq n} \{ \text{OPT}(i) \}$$

Iterative Algorithm

Longest Increasing Subsequence ($A[1, \dots, n]$)

```
1.  $L[1] = 1$ 
2. for ( $i = 2 \rightarrow n$ ) {
3.    $L[i] = 1$ 
4.   for ( $j = 1 \rightarrow i - 1$ ) {
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 $\Theta(n^2)$.

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

Bonus: a faster algorithm

As before, $l[i] = \text{length of longest increasing subsequence of } A[1, \dots, i]$

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

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

Example: $A = [2, 8, 10, 11, 1, 3, 5]$, $l[6] = 4$, 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



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

- $j=1$, best increasing sequence $[!]$ can add 5
- $j=2$, best increasing sequence $[1, \underline{3}]$ can add 5
- $j=3$, best increasing sequence $[2, 8, \underline{10}]$ can't add 5
- $j=4$, best increasing sequence $[2, 8, 10, \underline{11}]$ can't add 5

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

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

- $j=1$, best increasing sequence $[!]$ can add 15
 - $j=2$, best increasing sequence $[1, \underline{3}]$ can add 15
 - $j=3$, best increasing sequence $[2, 8, 10]$ can add 15
 - $j=4$, best increasing sequence $[2, 8, 10, \underline{11}]$ can add 15
- ↓

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

- $j=1$, best increasing sequence $[!]$ can add 15
- $j=2$, best increasing sequence $[1, \underline{3}]$ can add 15
- $j=3$, best increasing sequence $[2, 8, \underline{10}]$ can add 15
- $j=4$, best increasing sequence $[2, 8, 10, \underline{11}]$ can add 15

$1 < 3 < 5 < 11 < 15$, so $\lambda[7] = 5$, 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 ($A[1, \dots, n]$)

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

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: $A = \text{'blurny'}$, $B = \text{'burgry'}$, longest common subsequence is 'burry'

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

$\text{OPT}(i, j) =$ length of the longest common subsequence of $A[1, \dots, i]$ and $B[1, \dots, j]$.

↳ # subproblems: $O(mn)$. $i=1 \rightarrow n$ $j=1 \rightarrow m$

	b	l	u	r	r	y
	0	1	2	...		n
b	0					
u	1					
r	2					
g	:					
e	:					
r						
y	m					

length of LCS ('blu', 'burg')

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

A Bivariate Recurrence

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

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

- $M[i, j]$ 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 $A[i] = B[j]$, $| + M[i-1, j-1]$.

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

↳ 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 $A[1, \dots, n]$ and $B[1, \dots, m]$ of characters

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

Example: $A = \text{snowy}$, $B = \text{sunny}$

$s n o w y$	$s - n o w y$	$- s n o w y -$
$s u n n y$	$s u n n y -$	$s u n - - n y$
↳ 3C	↳ 1A, 2C, 1D	↳ 2A, 2C, 2D

The Recurrence

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

- $D[0, j] = j$ ∀ j (add j characters to empty A)
- $D[i, 0] = i$ ∀ 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 $A[1, \dots, i-1]$ with $B[1, \dots, j]$)
 - $D[i, j-1] + 1$ (add $B[j]$ and match $A[1, \dots, i]$ with $B[1, \dots, j-1]$)

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

Maximum Independent Set in a Tree

The Problem:

The Input: a tree T (connected undirected graph with no cycles) with n vertices

The Output:

- a maximum cardinality independent set of vertices in T
- a subset S of vertices is independent if there's no edge in T connecting 2 elements of S

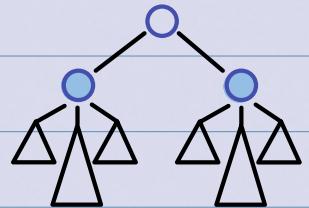
Remarks:

- in a general graph, INDEPENDENTSET is NP-complete
- a priori not a rooted tree, but we can suppose we chose a root r
- Vertices = $\{1, \dots, n\}$, each vertex i stores a linked list $C[i]$ children.

Is the root in S or not?

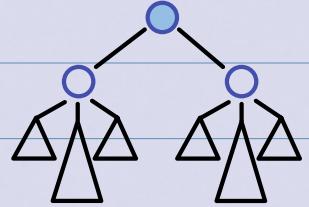
No:

- S is the union of independent sets that are contained in the children of the root
- conversely, any such union gives an independent set in T



Yes:

- none of its children can be in S
- so S is the union of independent sets that are contained in the grandchildren of the root
- conversely, any such union gives (with the root) an independent set in T .



Setting Up the Recurrence

$$\hookrightarrow O(T) = \max \left\{ 1 + \sum_{C \text{ grandchild of } r} O(C), \sum_{C' \text{ child of } r} O(C') \right\}.$$

Algorithm:

- level-order traversal, get an array $V[0, \dots, h]$, $V[i] =$ linked list of vertices at level i
- for v in $V[h]$, set $O[v] = 1$
- for $i = h-1, \dots, 0$, for v in $V[i]$, use the recurrence to get $O[v]$
↳ (loop over children and grandchildren to get the sums)

Runtime: proportional to $\sum_{v \text{ vertex in } T} 1 + \sum_{v \text{ vertex in } T} \# \text{children}(v) + \sum_{v \text{ vertex in } T} \# \text{grandchildren}(v)$

- Second sum is number of vertices of level at least 1
- Third sum is number of vertices of level at least 2
- So, $\Theta(n)$ altogether.

Optimal Binary Search Trees

The Problem

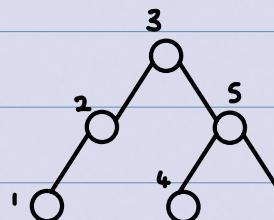
The Input:

- integers (or something else that can be ordered) $1, \dots, n$.
- probabilities of access p_1, \dots, p_n (so $p_1 + \dots + p_n = 1$, but we don't need this)

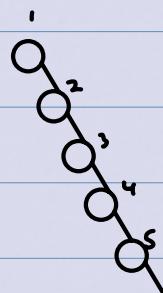
The Output:

- an optimal BST with keys $1, \dots, n$
- optimal: minimizes $\sum_{i=1}^n p_i \cdot \text{depth}(i)$ = expected number of tests for a search
↳ depth starts at 1.

Example: $p_1 = p_2 = p_3 = p_4 = p_5 = \frac{1}{5}$.



$$1 \cdot \frac{1}{5} + 2 \cdot 2 \cdot \frac{1}{5} + 2 \cdot 3 \cdot \frac{1}{5} = \frac{11}{5}$$



$$\frac{1}{5} (1+2+3+4+5) = 3$$

Remark:

- there are $\frac{1}{n+1} \binom{2n}{n}$ BSTs w/ n keys
- this is $\Theta(4^n / n^{1.5})$.

Setting Up the Recurrence

Definition for i, j in $\{1, \dots, n\}$, we define $M[i, j]$ by:

- $M[i, j] = \text{the minimal cost for items } \{1, \dots, j\}, \text{ if } i \leq j.$
- $M[i, j] = 0 \text{ if } j < i.$

Want: $M[1, n]$.

$$\begin{aligned} \text{Recurrence: } M[i, j] &= \min_{i \leq k \leq j} \left\{ M[i, k-1] + \sum_{l=i}^{k-1} p_l + M[k+1, j] + \sum_{l=k+1}^j p_l \right\} \\ &= \min_{i \leq k \leq j} \left\{ M[i, k-1] + M[k+1, j] \right\} + \sum_{l=i}^j p_l \end{aligned}$$

check: gives $M[i, i] = p_i$

Algorithm:

↳ Remark: to get $\sum_{l=i}^j p_l$:

- Compute $S[\lambda] = p_1 + \dots + p_\lambda$ for $\lambda = 1, \dots, n$
- Then $p_i + \dots + p_j = S[j] - S[i-1]$, with $S[0] = 0$.

Optimal BST ($p_1, \dots, p_n, S_0, \dots, S_n$)

1. for ($i = 1 \rightarrow n+1$) {

2. $M[i, i-1] = 0$

3. }

4. for ($d = 0 \rightarrow n-1$) { // $d = j-i$

5. for ($i = 1, \dots, n-d$) {

6. $j = d+i$

7. $M[i, j] = \min_{i \leq k \leq j} (M[i, k-1] + M[k+1, j]) + S[j] - S[i-1]$

8. }

9. }

$\Theta(n^3)$

Bonus: a Faster Algorithm

for all i, j , let $R_{i,j}$ be the largest index that gives a min
on line 7

↓

Claim (difficult): for all i, j with $j > i$, we have $R_{i,j-1} \leq R_{i,j} \leq k_{i+1,j}$
(root shifts left (right) if you add elements on the left (right)).

Better Optimal BST ($p_1, \dots, p_n, s_0, \dots, s_n$)

```
1. for ( $i=1 \rightarrow n+1$ ) {  
2.    $M[i, i-1] = 0$   
3. }  
4. for ( $d=0 \rightarrow n-1$ ) {           //  $d=j-i$   
5.   for ( $i=1 \rightarrow n-d$ ) {  
6.      $j = d+i$ ;  
7.     if ( $d=0$ ) { range =  $\{i\}$  } else { range =  $\{k_{i,j-1}, \dots, k_{i+1,j}\}$  }  
8.      $M[i, j], R_{i,j} = \min_{k \in \text{range}} \{M[i, k-1] + M[k+1, j]\} + S[j] - S[i-1]$   
9.   }  
10. }
```

Runtime, Revised:

Work is proportional to $\sum_{d=0}^{n-1} \sum_{i=1}^{n-d} (R_{i+1,j} - R_{i,j-1} + 1)$

$$\begin{aligned} &= \sum_{d=0}^{n-1} \sum_{i=1}^{n-d} (R_{i+1,i+d} - R_{i,i-1+d} + 1) \leq \sum_{d=0}^{n-1} \sum_{i=1}^{n-d} (R_{i+1,i+d} - R_{i,i-1+d}) + \sum_{d=0}^{n-1} \sum_{i=1}^{n-d} 1 \\ &\leq \sum_{d=0}^{n-1} (R_{n-d+1}, R_{0,d-1}) + \sum_{d=0}^{n-1} \sum_{i=1}^{n-d} 1 \leq 2n^2 \end{aligned}$$

↳ conclusion: $\Theta(n^2)$!

Text Segmentation

The Problem

The Input: a string, represented as an array $A[1, \dots, n]$.

The Output:

- true if we can segment A into words from a given dictionary

- false otherwise

↳ we assume that we can test if $A[i, \dots, j]$ is a word in $O(1)$.

Example: $A = \text{caramelow} \rightarrow$ true w/ car a me low.

Remark: there are 2^{n-1} ways to segment A

Subproblems: can we split $A[1, \dots, i]$ into words?

Definition: for $i=1, \dots, n$, let $s[i]$ be

- true if we can segment $A[1, \dots, i]$ into words

- false otherwise

↳ we set $s[0]$ to true.

Recurrence: $s[i] = \text{or}_{j=0}^{i-1} (s[j] \text{ and } \text{isWord}(A[j+1, \dots, i]))$.

Algorithm: (note: can be done recursively, but we'll do iteratively)

IsSplitable($A[1, \dots, n]$)

1. $s[0] = \text{true}$

2. for ($i=1 \rightarrow n$) {

3. $s[i] = \text{false}$

4. for ($j=0 \rightarrow i-1$) {

5. $s[i] = s[i]$ or $(s[j] \text{ and } \text{isWord}(A[j+1, \dots, i]))$
6. }
7. }
8. return $s[n]$

$\mathcal{O}(n^2)$

Graphs

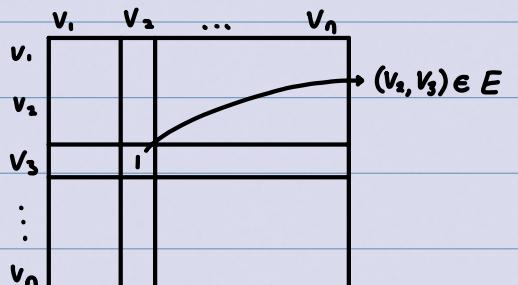
An (undirected) graph G is a pair (V, E) :

- V is a finite set, whose elements are called vertices
(we often take $V = \{1, \dots, n\}$)
- E is a finite set, whose elements are sets of two (distinct) vertices, and are called edges.

convention: n is the number of vertices, m is the number of edges

Adjacency List: an array $A[1, \dots, n]$ s.t. $A[v]$ is the linked list of all edges connected to v . $2m$ list cells, total size $\mathcal{O}(n+m)$, but testing if an edge exists is not $\mathcal{O}(1)$. eg: $V_1: [v_1] \rightarrow [v_2] \rightarrow [v_3] \rightarrow \emptyset$, $V_2: \emptyset$, ...

Adjacency Matrix: $(0, 1)$ matrix M of size $n \times n$, with $M[v, w] = 1$ iff $\{v, w\}$ is an edge. Size $\mathcal{O}(n^2)$ but testing if an edge exists is $\mathcal{O}(1)$.



Walk: a sequence v_0, \dots, v_k of vertices with $\{v_i, v_{i+1}\}$ in E for $i = 0, \dots, k-1$. length = # steps = k ($k=0$ is fine)

Path: a walk with distinct vertices

Connected Graph: $G = (V, E)$ such that for all v, w in V , there

is a path/walk from $v \rightarrow w$.

Cycle: a walk v_0, \dots, v_k, v_0 with $k \geq 2$ and v_i 's distinct
 $\hookrightarrow k=1$ would be v_0, v_1, v_0 , this is not a cycle)

Tree: a connected graph with no cycle.

\hookrightarrow equivalently, a connected graph with $m = n - 1$

equivalently, a graph with $m = n - 1$ and no cycle

Rooted Tree: a tree with a special vertex called the root.

Breadth-First Exploration of a Graph

\hookrightarrow this explores one connected component

BFS(G, s)

G : a graph with n vertices, given by adjacency lists

s : a vertex from G

1. let Q be an empty queue
2. let discovered be an array of size n , with all entries false
3. enqueue(s, Q)
4. $\text{discovered}[s] = \text{true}$
5. while (Q not empty) {
 6. $v = \text{dequeue}(Q)$
 7. for (all w in $A[v]$) {
 8. if ($\text{discovered}[w]$ is false) {
 9. enqueue(s, Q)
 10. $\text{discovered}[w] = \text{true}$
 11. }
 12. }

Runtime:

- each vertex is enqueued at most once
 - so each vertex is dequeued at most once
 - so each adjacency list is read at most once!
- $O(n)$ for
Steps 5-6

For all v , write $d_v = \text{number of neighbors of } v = \text{length of } A[v] = \text{degree of } v$.

Then, the total cost at step 7 is $O\left(\sum_v 1 + d_v\right) = O(m+n)$

c.f. the adjacency array A has $2m$ cells.

Correctness

Claim: If vertices v , if $\text{discovered}[v]$ is true at the end, then there is a walk $s \rightsquigarrow v$ in G .

Proof: let $S = v_0, \dots, v_k$ be the vertices for which discovered is set to true, in this order.

We prove: for all i , there is a walk $s \rightsquigarrow v_i$ by induction.

- OK for $i=0$
- Suppose true for v_0, \dots, v_{i-1} .

When $\text{discovered}[v_i]$ is true, we are examining the neighbors of a certain v_j

v_j came out of the queue, so it was previously discovered and enqueued, so $j < i$.

by assumption, there is a walk $s \rightsquigarrow v_j$.

because $\{s, v_j\}$ is in E , there is a walk $s \rightsquigarrow v_i$.

Claim 2: If vertices v , if there is a walk $s \rightsquigarrow v$ in G ,

$\text{discovered}[v]$ is true at the end.

Proof: let $V_0 = S, \dots, V_k$ be a walk $S \rightsquigarrow v$. We prove $\text{discovered}[v]$ is true for all i by induction.

- $\text{discovered}[V_0]$ is true
- if $\text{discovered}[V_i]$ is true, we will examine all neighbors v of V_i so at the end of step 7, all $\text{discovered}[v]$ will be true, for v neighbor of V_i .
in particular, $\text{discovered}[V_{i+1}]$ will be true

Summary: for all vertices v , there is a walk $S \rightsquigarrow v$ in G if and only if $\text{discovered}[v]$ is true at the end.

Keeping track of parents and levels

BFS(G, s)

1. let Q be an empty queue
2. let parent be an array of size n , initially all NIL
3. let level be an array of size n , initially all ∞
4. enqueue(s, Q)
5. $\text{parent}[s] = s$
6. $\text{level}[s] = 0$
7. while (Q is not empty) {
8. $v = \text{dequeue}(Q)$
9. for (all w neighbors of v) {
10. if ($\text{parent}[w]$ is NIL) {
11. enqueue(w, Q)
12. $\text{parent}[w] = v$
13. } $\text{level}[w] = \text{level}[v] + 1$

14. }

15. }

16. }

↳ (old discovered[v] true iff new parent[v] ≠ NIL iff new level[v] ≠ ∞

BFS Tree

the BFS Tree is the subgraph made of:

- all w edges that $\text{parent}[w] \neq \text{NIL}$
- all edges $\{w, \text{parent}[w]\}$, for w as above (except $w=s$)

Claim: the BFS Tree is a tree

Proof: T connected, n_s vertices, $n_s - 1$ edges where n_s is the number of vertices reachable from s. \therefore a tree.

Remark: we make the BFS Tree a rooted tree by choosing s as the root

Shortest Paths from the BFS Tree

Claim: if there is a walk $s \rightsquigarrow v$ in G, then $\text{level}[s] = \text{dist}(s, v)$

Observation:

- $\text{dist}(s, v) = \text{length of the shortest path from } s \text{ to } v$
- so $\text{dist}(s, v) \leq \text{level}[v]$.

↓

- the levels in the queue are always of the form $[l, \dots, l]$ or $[l, \dots, l, l+1, \dots, l+1]$

↳ Proof: check what happens when we enqueue/deque:

$$[\underbrace{l, \dots, l}_{i \geq 0}, \underbrace{l+1, \dots, l+1}_{j \geq 0}] \rightarrow [\underbrace{l, \dots, l}_{i-1}, \underbrace{l+1, \dots, l+1}_{j' \geq j}]$$

Consequence: if we dequeue v before w , $\text{level}[v] \leq \text{level}[w]$.

- If there is an edge (v, w) then $\text{level}[w] \leq \text{level}[v] + 1$

Proof: let $z = \text{parent}[w]$, so $\text{level}[w] = \text{level}[z] + 1$

- either $z = v$

- or z came out of the queue before v

but then $\text{level}[z] \leq \text{level}[w]$ (from consequence at top of this page)

- If there is a walk $s \rightsquigarrow v$ in G , then $\text{level}[v] \leq \text{dist}(s, v)$

Proof:

- take any walk $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = v$

- $\text{level}[v_0] = 0$, $\text{level}[v_1] \leq 0 + 1 = 1$, $\text{level}[v_2] \leq 1 + 1 = 2$, \dots ,

- $\text{level}[v_k] \leq k$

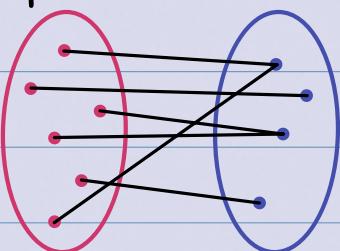
- take a shortest path and get $\text{level}[v] \leq \text{dist}(s, v)$

Bipartite Graphs

a graph $G = (V, E)$ is bipartite if there is a partition

$V = V_1 \cup V_2$ such that all edges have exactly

one edge in V_1 and the other in V_2 .



All (BFS) Trees are bipartite!

Suppose G is connected, run BFS from any s , and set

$V_1 = \text{vertices with odd level}$

$V_2 = \text{vertices with even level}$

Then G is bipartite if and only if all edges have one end in V_1 and one end in V_2 (testable in $O(n+m) = O(m)$). If so, bipartition is $V_1 \cup V_2$.

Subgraph of $G = (V, E)$: a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, with all edges E' joining vertices in V'

Connected component of $G = (V, E)$:

- a connected subgraph of G
 - to which you cannot add any edge or connect any vertex
- ie: a maximal connected subgraph of G .

Note: we can compute all of the components in BFS by:
adding an outer loop that runs BFS on successive vertices, and
each vertex has a new attribute 'component'
↳ each pass $O(n_i + m_i)$, if $G_i(V_i, E_i)$ is the i^{th} component.
so total $O(n+m)$.