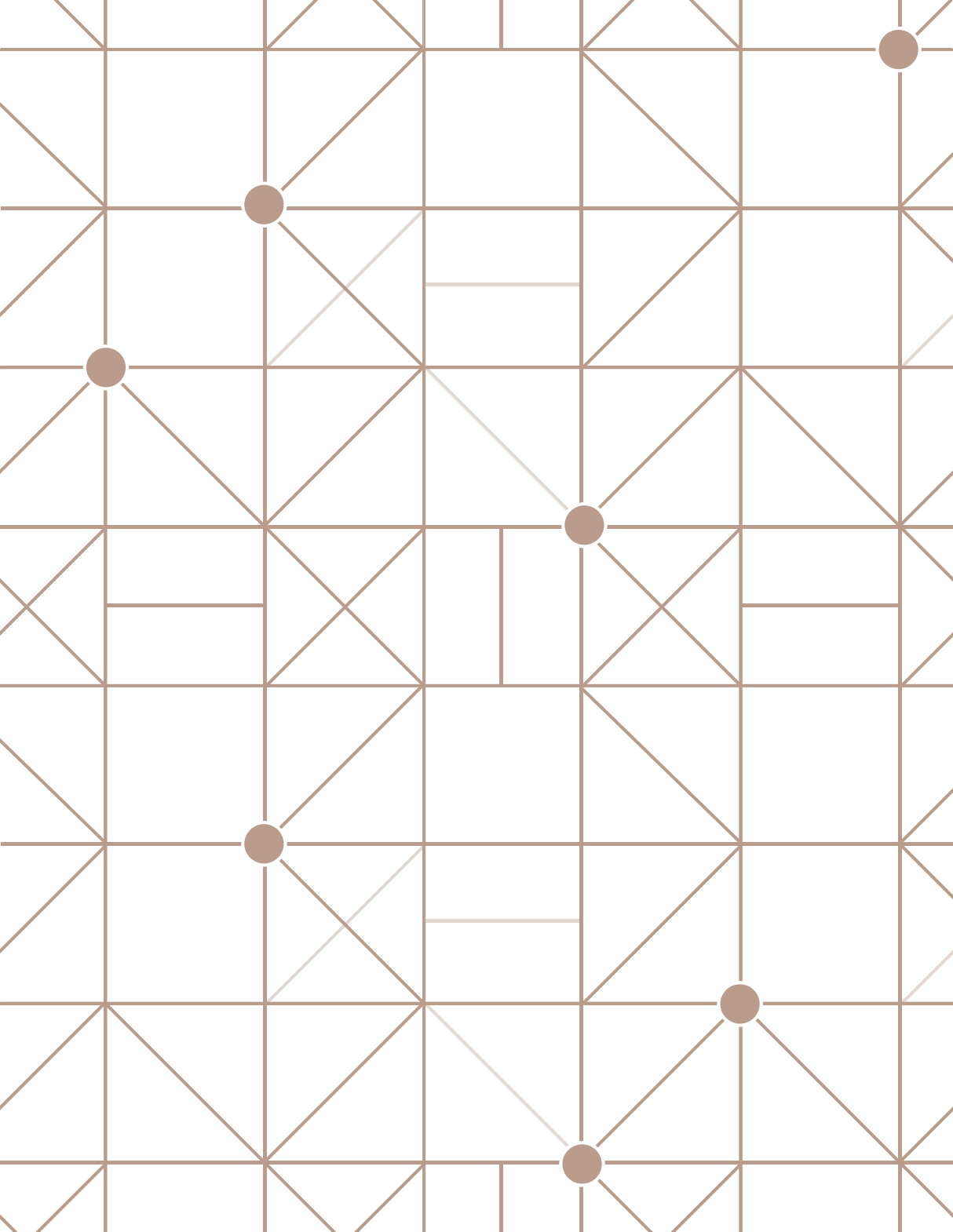


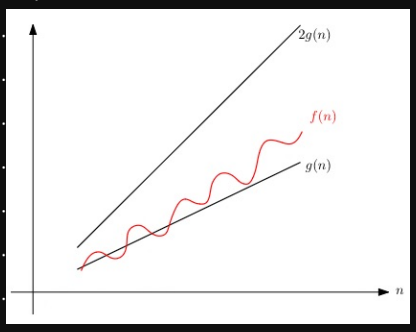
INTRODUCTION

COST OF ALGORITHMS

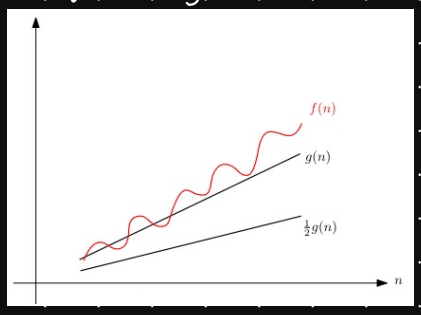
- inputs are parametrized by `int n (size)`
  - ↳ runtime of particular instance: $T(I)$ = runtime on input I
  - ↳ worst-case runtime: $T(n) = \max_{I \text{ of size } n} T(I)$
  - ↳ avg runtime: $T_{avg}(n) = \frac{\sum_{I \text{ of size } n} T(I)}{\#\text{ inputs of size } n}$
- $O(n) = O(2^{\log n})$ ($\log n$ = input size)

ASYMPTOTIC NOTATION

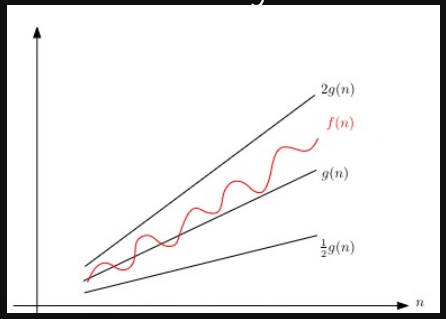
- **big-O**: $f(n) \in O(g(n))$ if there exist $C > 0$ & n_0 st for $n \geq n_0$, $f(n) \leq C g(n)$



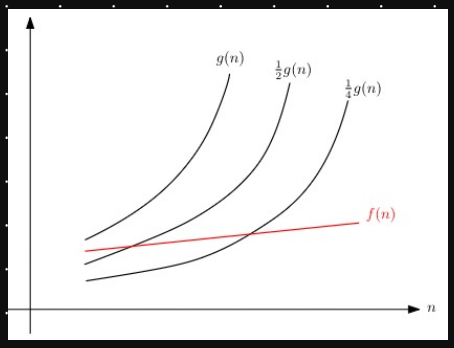
- **big-Ω**: $f(n) \in \Omega(g(n))$ if there exist $C > 0$ & n_0 st for $n \geq n_0$, $f(n) \geq C g(n)$
  - ↳ equiv to $g(n) \in O(f(n))$



- **Θ**: $f(n) \in \Theta(g(n))$ if there exist $C, C' > 0$ & n_0 st for $n \geq n_0$, $C' g(n) \leq f(n) \leq C g(n)$
  - ↳ equiv to $f(n) \in O(g(n))$ & $f(n) \in \Omega(g(n))$
  - ↳ true if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = C$ for some $0 < C < \infty$



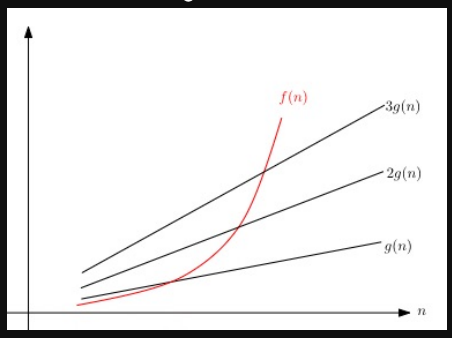
- **little-o**: $f(n) \in o(g(n))$ if for all $C > 0$, there exists n_0 st for $n \geq n_0$, $f(n) \leq C g(n)$
  - ↳ equiv to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$



little-ω · $f(n) \in \omega(g(n))$ if for all $C > 0$, there exists n_0 st for $n \geq n_0$, $f(n) > Cg(n)$

↳ equiv to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

↳ equiv to $g(n) \in o(f(n))$



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

e.g. $n^{o(1)}$ means at most polynomial in n

e.g. nlogn is in $O(n^2)$ + $\Omega(n)$

e.g. 2^{n-1} is in $\Theta(2^n)$? T

↳ $2^n = 2 \; 2^{n-1}$

e.g. $(n-1)! \in \Theta(n!)$? F

↳ $\lim_{n\to\infty} \frac{(n-1)!}{n!} = \lim_{n\to\infty} \frac{1}{n} = 0$

$(n-1)! \in o(n!) \Rightarrow (n-1)! \notin \Omega(n!)$

$f(n,m) \in O(g(n,m))$ if there exist C, n_0, m_0 st $f(n,m) \leq C\,g(n,m)$ for $n \geq n_0$ or $m \geq m_0$

↳ less strict defn is for $n \geq n_0$ + $m \geq m_0$

COMPUTATIONAL MODEL: WORD RAM

mem locations contain integer words of b bits each

assume $b \geq \log(n)$ for input size n

RAM: can access any mem loc at unit cost

basic operations $(+, -, \times, \div)$ on words have unit costs

e.g.

Sum$(A[1..n])$
1. $s \leftarrow 0$
2. **for** $i = 1, \ldots, n$
3. $\quad s \leftarrow s + A[i]$

↳ if all entries of A fit into a word, cost is $O(n)$
- at most 1 extra digit in each loop
- max possible space needed is 2 words

e.g.

Product$(A[1..n])$
1. $s \leftarrow 1$
2. **for** $i = 1, \ldots, n$
3. $\quad s \leftarrow s \times A[i]$

↳ if all entries of A fit into a word, runtime is $O(n^2)$
- in worst case, size of s is i words on i^{th} turn

↳ matrix multiplication w/ word-size inputs are OK
↳ other matrix algos (e.g. Gaussian elim) need more care

big-O is **only upper bound**
↳ e.g. 1 is in $O(n^2)$ + n is in $O(n)$ but 1 is better runtime
↳ give Θs if possible

big-anything **hides constants** by design
↳ e.g.

$O(n^3)$	$\rightsquigarrow 4n^3$	this turns out to be better in most cases
$O(n^{2.8\ldots})$	$\rightsquigarrow 300n^{2.8\ldots}$	
$O(n^{2.37\ldots})$	$\rightsquigarrow 10^{67}n^{2.37\ldots}$	

we use **simplified model**
↳ artificial computational model
↳ focus on ops + forget abt mem requirements, data locality, etc.

CASE STUDY: MAX SUBARRAY

Task

Given an array $A[1..n]$ (ints), find a contiguous subarray $A[i..j]$ that maximizes the sum $A[i] + \cdots + A[j]$.

Example. Given

$$A = [10, -5, 4, 3, -5, 6, -1, -1]$$

the subarray

$$A[1..6] = [10, -5, 4, 3, -5, 6]$$

has sum $10 + \cdots + 6 = 13$. It is the best we can do.

convention is to take $j = i - 1$ so $A[i \ldots j]$ is empty (since $j < i$) + sum = 0

brute force algo:

BruteForce(A)
1. opt $\leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $\quad$ **for** $j \leftarrow i$ **to** n **do**
4. $\quad\quad$ sum $\leftarrow 0$
5. $\quad\quad$ **for** $k \leftarrow i$ **to** j **do**
6. $\quad\quad\quad$ sum $\leftarrow$ sum $+ A[k]$
7. $\quad\quad$ **if** sum $>$ opt
8. $\quad\quad\quad$ opt $\leftarrow$ sum
9. **return** opt

↳ runtime is $\Theta(n^3)$

improved brute force algo:

```
BetterBruteForce(A)
1.    opt ← 0
2.    for i ← 1 to n do
3.        sum ← 0
4.        for j ← i to n do
5.            sum ← sum + A[j]
6.            if sum > opt
7.                opt ← sum
8.    return opt
```

↳ recompute same sum many times in 3rd loop

↳ runtime is $\Theta(n^2)$

divide + conquer: solve problem twice in size $\frac{n}{2}$ (assume n is power of 2), then optimal subarr (mutually exclusive):

↳ is completely in left half $A[1 \ldots \frac{n}{2}]$

↳ is completely in left half $A[\frac{n}{2}+1 \ldots n]$

↳ contains both $A[\frac{n}{2}]$ + $A[\frac{n}{2}+1]$

- to find optimal subarr, $A[i]+\ldots+A[j] = A[i]+\ldots+A[\frac{n}{2}]+A[\frac{n}{2}+1]+\ldots+A[j]$
  - → $F(i,j) = f(i) + g(j)$ for i in $1, \ldots, \frac{n}{2}$ + j in $\frac{n}{2}+1, \ldots, n$
  - → maximize $f(i) + g(j)$ independently to maximize $F(i,j)$

```
MaximizeLowerHalf(A)
1.    opt ← A[n/2]
2.    sum ← A[n/2]
3.    for i = n/2 − 1, . . . , 1 do
4.        sum ← sum + A[i]
5.        if sum > opt
6.            opt ← sum
7.    return opt
```

↳ runtime is $\Theta(n)$

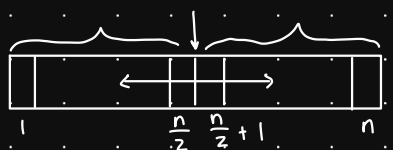
Maximize Upper Half (A) runtime is $\Theta(n)$

```
DivideAndConquer(A[1..n])
1.    if n = 1 return max(A[1], 0)
2.    opt_lo ← DivideAndConquer(A[1..n/2])
3.    opt_hi ← DivideAndConquer(A[n/2 + 1..n])
4.    opt_middle ← MaximizeLowerHalf(A) + MaximizeUpperHalf(A)
5.    return max(opt_lo, opt_hi, opt_middle)
```

↳ runtime is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ so $T(n) \in \Theta(n \log n)$

↳ same as Merge Sort

↳



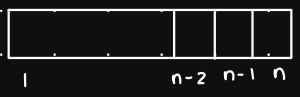
dynamic programming: solve problem in subarrs $A[1 \dots j]$ of sizes $1, \dots, n$

↳ optimal subarr (mutually exclusive):

- subarr of $A[1 \dots n-1]$
- contains $A[n]$

↳ $M(n) = \max(M(n-1), \bar{M}(n))$

- $M(j)$ = max sum for subarrs of $A[1 \dots j]$
- $\bar{M}(j)$ = max sum for subarrs of $A[1 \dots j]$ that include j

↳ 

to compute $\bar{M}(j)$ $\forall j \in [1, n]$, we say optimal subarr that contains $A[n]$ is (mutually exclusive):

↳ $A[i \dots n-1, n]$ $\exists i \leq n-1$

↳ $A[n]$

$\bar{M}(n) = \max(\bar{M}(n-1) + A[n], A[n]) = A[n] + \max(\bar{M}(n-1), 0)$

↳ elim recursive calls + write as loop:

```
1.  M̄ ← A[1]
2.  for i = 2, ..., n do
3.      M̄ ← A[i] + max(M̄, 0)
```

main algo for DP:

```
DynamicProgramming(A)
1.  M̄ ← A[1]
2.  M ← max(M̄, 0)
3.  for i = 2, ..., n do
4.      M̄ ← A[i] + max(M̄, 0)
5.      M ← max(M, M̄)
6.  return M
```

↳ runtime is $\Theta(n)$

SOLVING RECURRENCES

MERGESORT

input is arr A of n ints

1) split A into 2 subarrs
   - A_L is 1st $\lceil \frac{n}{2} \rceil$ elmts
   - A_R is last $\lfloor \frac{n}{2} \rfloor$ elmts
2) recursively run MergeSort on A_L + A_R
3) after A_L + A_R are sorted, use Merge to merge into single sorted arr

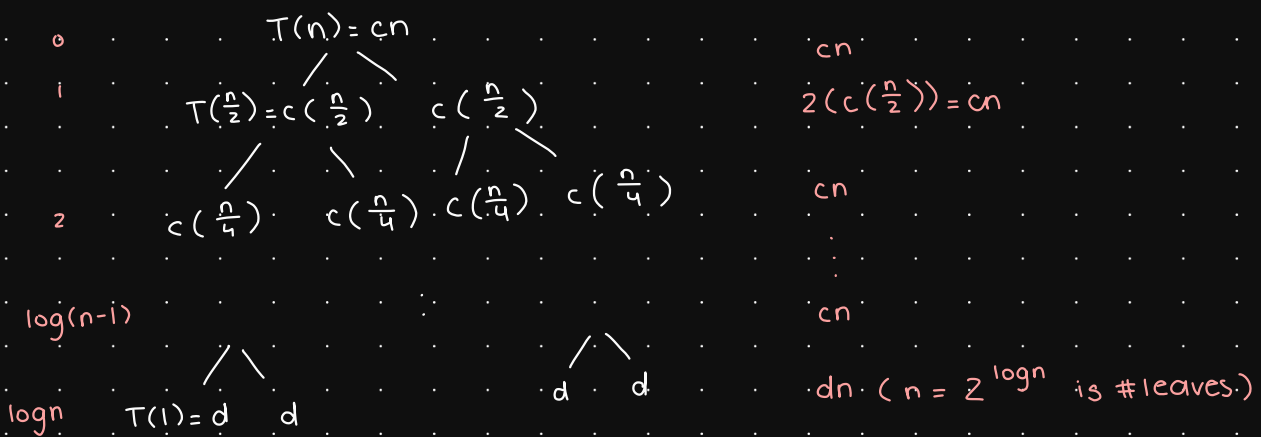
recurrence is $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$

$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$ exact recurrence w/ constants

$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$ sloppy recurrence

↳ exact + sloppy recurrences are identical when n is power of 2

construct recursion tree, assuming $n = 2^j$



↳ sum vals on every lvl to get cn + ... cn + dn = $cn\log n + dn$

↳ $T(n) \in \Theta(n \log n)$

↳ method is soln of exact recurrence when $n = 2^j$

↳ if soln expressed in Θ-notation, we have complexity $\forall n$

↳ not proof b/c induction is necessary

MASTER METHOD

Master Theorem provides formula for soln of many recurrence rltns typically encountered in analysis of divide + conquer algos

simplified version:

Theorem (Master theorem)

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

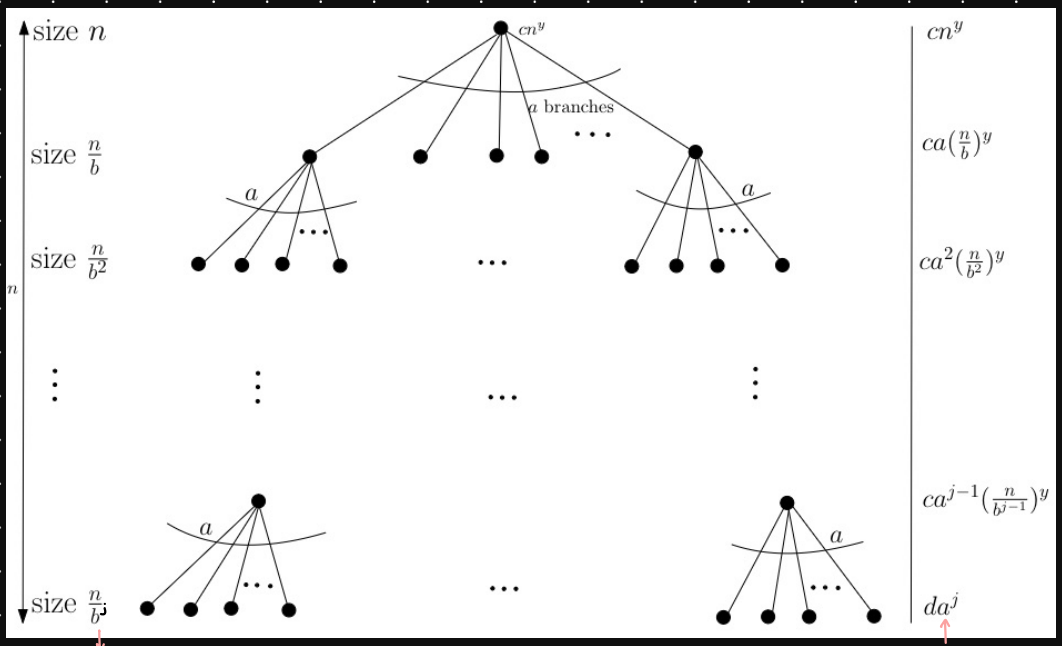
$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^y)$$

in sloppy or exact form. Denote $x = \log_b a$. Then

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

suppose $n = b^j$, $a \geq 1$, $b \geq 2$ are ints., $T(n) = a\,T(\frac{n}{b}) + cn^y$, $T(1) = d$:

(size of 1 node) (sum of costs on that lvl)



since $n = b^j$, $\frac{n}{b^j} = 1$

a^j = #leaves

↳ total is $da^j + cn^y \sum_{i=0}^{j-1} (\frac{a}{b^y})^i$

↳

size of subproblem	# nodes	cost/node	total cost
$n = b^j$	1	$c\,n^y$	$c\,n^y$
$n/b = b^{j-1}$	a	$c\,(n/b)^y$	$c\,a\,(n/b)^y$
$n/b^2 = b^{j-2}$	a^2	$c\,(n/b^2)^y$	$c\,a^2\,(n/b^2)^y$
⋮	⋮	⋮	⋮
$n/b^{j-1} = b$	a^{j-1}	$c\,(n/b^{j-1})^y$	$c\,a^{j-1}\,(n/b^{j-1})^y$
$n/b^j = 1$	a^j	d	$d\,a^j$

↳ writing $T(n)$ in terms of n:

$T(n) = da^j + cn^y \sum_{i=0}^{j-1} (\frac{a}{b^y})^i$

$= dn^x + cn^y \sum_{i=0}^{j-1} r^i$

- case 1: $r = 1$ $(x = y)$

$\sum_{i=0}^{j-1} r^i = \sum_{i=0}^{j-1} b^{x-y} = \sum_{i=0}^{j-1} b^0 = \sum_{i=0}^{j-1} 1 = j$

$T(n) = dn^x + cn^y \log_b n \in \Theta(n^y)$

- case 2: $r > 1$ $(x > y)$

$n = b^j \rightarrow j = \log_b n$

$x = \log_b a \rightarrow a = b^x$

$a^j = (b^x)^j = (b^j)^x = n^x$

$r = \frac{a}{b^y}$

$= \frac{b^x}{b^y}$

$= b^{x-y}$

$\sum_{i=0}^{j-1} r^i = \frac{r^j - 1}{r-1} \in \Theta(r^j)$

$r^j = (b^{x-y})^{\log_b n} = (b^{\log_b n})^{x-y} = n^{x-y}$

$T(n) = dn^x + cn^y (n^{x-y}) \in \Theta(n^x)$

- case 3: $r < 1$ $(y > x)$

$\sum_{i=0}^{j-1} r^i \in \Theta(1)$

$T(n) = dn^x + cn^y \in \Theta(n^y)$

↳

case	r	y, x	complexity of $T(n)$
heavy leaves	$r > 1$	$y < x$	$T(n) \in \Theta(n^x)$
balanced	$r = 1$	$y = x$	$T(n) \in \Theta(n^y \log n)$
heavy top	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

- heavy leaves means cost of recursion tree is dominated by leaf nodes
- heavy top means cost of recursion tree is dominated by root node

substitution method: solve recurrence

↳ e.g. $T(n) = 2T(\frac{n}{2}) + n$

We want to show $T(n) \in O(n \log n)$. Inductive hypothesis is $T(k) \leq ck \log k$ for all $k < n$

Base case: $n = 1$

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

$= 2(1) + 1$

$= 3$

$T(n) \in \Theta(1)$ so $T(n) \in O(n \log n)$. We've proven the base case.

Inductive step:

$T(n) \leq 2T(\frac{n}{2}) + n$

$\leq 2(c \frac{n}{2} \log(\frac{n}{2})) + n$ inductive hypothesis

$\leq cn(\log n - \log 2) + n$

$\leq cn \log n - cn + n$

$\leq cn \log n + (1 - c)n$

$\leq cn \log n$

So, $T(n) \leq cn \log n$, assuming $c \geq 1$. Thus, we've proven $T(n) \in O(n \log n)$.

Master method:

$a = 2$

$b = 2$

$x = \log_2 2 = 1$

$y = 1$

$\therefore T(n) \in \Theta(n \log n)$

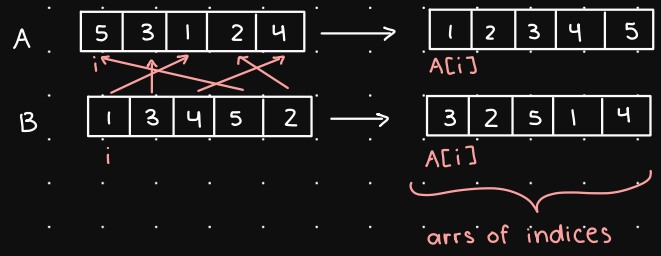
DIVIDE AND CONQUER

general algo paradigm (aka strategy):
- ↳ divide: split problem into several subproblems
- ↳ conquer: solve subproblems recursively by applying same algo
- ↳ combine: use subproblem results to derive final result

use **divide + conquer** when og problem is easily decomposable, combining is not too costly, + subproblems aren't overly unbalanced

e.g.

Collaborative filtering:
- matches users preference (movies, music, ...)
- determine users with *similar* tastes
- recommends new things to users based on preferences of similar users



e.g.

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

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

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

$$(2,3), (2,5), (2,8), (4,5), (4,8), (6,7), (6,8), (7,8)$$

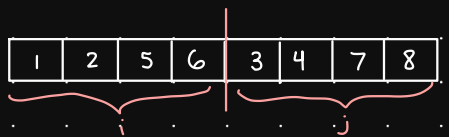
- ↳ we show **indices** where inversions occur
- ↳ easy algo w/ 2 nested loops in $\Theta(n^2)$
- ↳ for n is power of 2:
  - c_ℓ : # inversions in $A[1 \dots \frac{n}{2}]$
  - c_r : # inversions in $A[\frac{n}{2}+1 \dots n]$
  - c_t : # transverse inversions w/ $i \leq \frac{n}{2}$ + $j > \frac{n}{2}$
  - return $c_\ell + c_r + c_t$

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

• $c_\ell = 1$	$(2,3)$
• $c_r = 3$	$(6,7), (6,8), (7,8)$
• $c_t = 4$	$(2,5), (2,8), (4,5), (4,8)$

- ↳ $c_\ell + c_r$ done recursively

To get c_t, note it doesn't change even if both sides are sorted. Assume we sort $A_\ell + A_r$ after recursive calls.



$C_+ = \#i > 3 + \#i > 4 + \#j > 7 + \#i > 8$

↳ option 1: take each $i \le \frac{n}{2}$ + binary search its pos in A_r

- $O(\log n)$ per i, so total is $O(n \log n)$ + another $O(n \log n)$ for sorting $A_\ell + A_r$
- recurrence: $T(n) \le 2T(\frac{n}{2}) + O(n \log n)$

  sketchy proof (add details like # log terms for full credit):

  $T(n) \le 2T(\frac{n}{2}) + n \log n$

  $\le 4T(\frac{n}{4}) + n \log(\frac{n}{2}) + n \log n$

  $\le 8T(\frac{n}{8}) + n \log(\frac{n}{4}) + n \log(\frac{n}{2}) + n \log n$

  ⋮ log n times

  $\le n(\log n + \dots + \log 2)$

  $\le n \log^2 n$

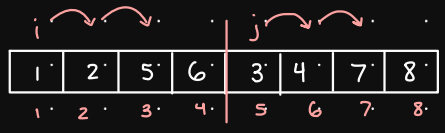
- $T(n) \in O(n \log^2 n)$

↳ option 2: enhance mergesort

Merge$(A[1..n])$ (both halves of A assumed sorted)

```
1.  copy A into a new array S;  c = 0
2.  i = 1; j = n/2 + 1;
3.  for (k ← 1; k ≤ n; k++) do
4.      if (i > n/2) A[k] ← S[j++]                    only A_r left
5.      else if (j > n) A[k] ← S[i++]; c = c + n/2    only A_ℓ left
6.      else if (S[i] < S[j]) A[k] ← S[i++]; c = c + j − (n/2 + 1)
7.      else A[k] ← S[j++]
```

- find C_+ during merge



Inserting 1 ($j = 5$): $c = c + 0$

Inserting 2 ($j = 5$): $c = c + 0$

Inserting 5 ($j = 7$): $c = c + 2$

- enhanced merge is still $\Theta(n)$, so $T(n) \in \Theta(n \log n)$

e.g. multiplying polynomials

Goal: given $F = f_0 + \cdots + f_{n-1}x^{n-1}$ and $G = b_0 + \cdots + b_{n-1}x^{n-1}$, compute

$$H = FG = f_0 g_0 + (f_0 g_1 + f_1 g_0)x + \cdots + f_{n-1} g_{n-1} x^{2n-2}$$

$T(n) \in O(n^2)$

```
1.  for i = 0, ..., n − 1 do
2.      for j = 0, ..., n − 1 do
3.          h_{i+j} = h_{i+j} + f_i g_j
```

↳ write $F = F_0 + F_1 x^{n/2}$, $G = G_0 + G_1 x^{n/2}$ so that
$H = F_0G_0 + (F_0G_1 + F_1G_0)x^{n/2} + F_1G_1 x^n$

- e.g.
  $n = 4$, $\frac{n}{2} = 2$
  $a_0 + a_1 x^1 + a_2 x^2 + a_3 x^3$ ← n-1
  $= (a_0 + a_1 x^1) + (a_2 + a_3 x^1)x^2$

- 4 recursive calls in size $\frac{n}{2}$ (multiplication)
- $\Theta(n)$ additions

Recurrence:
$T(n) = 4T(\frac{n}{2}) + \Theta(n)$
Using Master Thm, $a = 4$, $b = 2$, $x = \log_2 4 = 2$, $y = 1$. Since $x > y$, then $T(n) \in \Theta(n^2)$

↳ Karatsuba's algo: use identity
$(F_0 + F_1 x^{n/2})(G_0 + G_1 x^{n/2}) = F_0G_0 + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x^{n/2} + F_1G_1 x^n$

- 3 recursive calls in size $\frac{n}{2}$ (multiplication)
- $\Theta(n)$ additions + substractions
- multiplications by $x^{n/2}$ + x^n are free

Recurrence:
$T(n) = 3T(\frac{n}{2}) + \Theta(n)$
Using Master Thm., $a = 3$, $b = 2$, $x = \log_2 3$, $y = 1$. Since $x > y$, then $T(n) \in \Theta(n^{\log_2 3 \simeq 1.58})$

↳ Toom-Cook: family of algos based on similar exprs as Karatsuba

- for $k \geq 2$, $2k-1$ recursive calls in size $\frac{n}{k}$
- $T(n) = \Theta(n^{\log_k(2k-1)})$
- gets close to exp !, but very slowly

↳ fast Fourier transform (FFT): used w/ complex coeffs

- same recurrence as merge sort : $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- multiply polynomials in $\Theta(n \log n)$ ops

e.g multiplying matrices

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

Remark: input and output size $\Theta(n^2)$, easy algorithm in $\Theta(n^3)$

```
1.  for i = 1, ..., n do
2.      for j = 1, ..., n do
3.          for k = 1, ..., n do
4.              c_{i,k} = c_{i,k} + a_{i,j} b_{j,k}
```

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

↳ A_{ik} + B_{ij} of size $\frac{n}{2} \times \frac{n}{2}$

$$C = \begin{bmatrix} \overset{1}{A_{11}B_{11}} + \overset{2}{A_{12}B_{21}} & \overset{3}{A_{11}B_{12}} + \overset{4}{A_{12}B_{22}} \\ \underset{5}{A_{21}B_{11}} + \underset{6}{A_{22}B_{21}} & \underset{7}{A_{21}B_{12}} + \underset{8}{A_{22}B_{22}} \end{bmatrix}$$

↳ 8 recursive calls in size $\frac{n}{2}$ + $\Theta(n^2)$ additions

↳ master thm: $a = 8$, $b = 2$, $x = \log_2 8 = 3$, $y = 2$

$x > y$

$T(n) \in \Theta(n^3)$

↳ Strassen's algo:

Compute

$$\begin{aligned} Q_1 &= (A_{1,1} - A_{1,2})B_{2,2} \\ Q_2 &= (A_{2,1} - A_{2,2})B_{1,1} \\ Q_3 &= A_{2,2}(B_{1,1} + B_{2,1}) \\ Q_4 &= A_{1,1}(B_{1,2} + B_{2,2}) \\ Q_5 &= (A_{1,1} + A_{2,2})(B_{2,2} - B_{1,1}) \\ Q_6 &= (A_{1,1} + A_{2,1})(B_{1,1} + B_{1,2}) \\ Q_7 &= (A_{1,2} + A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

and

$$\begin{aligned} C_{1,1} &= Q_1 - Q_3 - Q_5 + Q_7 \\ C_{1,2} &= Q_4 - Q_1 \\ C_{2,1} &= Q_2 + Q_3 \\ C_{2,2} &= -Q_2 - Q_4 + Q_5 + Q_6 \end{aligned}$$

- 7 recursive calls in size $\frac{n}{2}$ + $\Theta(n^2)$ additions
- master thm: $a = 7$, $b = 2$, $x = \log_2 7$, $y = 2$

$x > y$:

$T(n) \in \Theta(n^{\log_2 7 \simeq 2.8})$

algo that does k multiplications for matrices of size ℓ gives $T(n) \in \Theta(n^{\log_\ell k})$

↳ algo that does k multiplications for matrices of size ℓ, m by m, p gives $T(n) \in \Theta(n^{3 \log_{\ell mp} k})$

e.g. closest pairs

Goal: given n points (x_i, y_i) in the plane, find a pair (i, j) that minimizes the distance

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

Equivalent to minimize

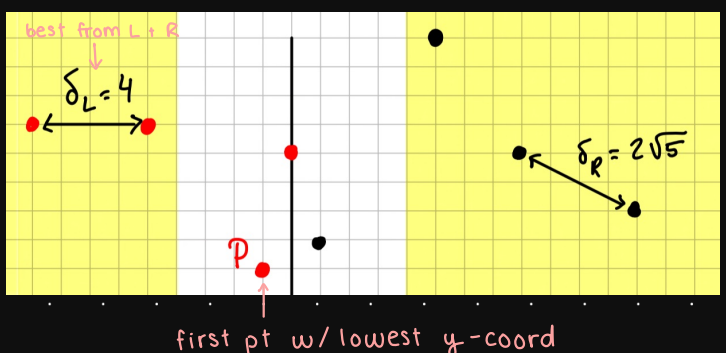
$$d_{i,j}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$$

Assumption: all x_i's are pairwise distinct

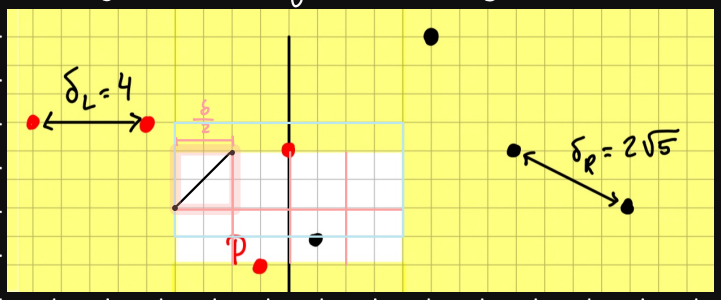
↳ separate pts into 2 halves L + R at median x-val

- L = all $\frac{n}{2}$ pts w/ $x \leq x_{med}$
- R = all $\frac{n}{2}$ pts w/ $x > x_{med}$
- closest pair is either btwn pts in L (recursion), btwn pts in R (recursion), or transverse (1 in L + 1 in R)

↳



- set $\delta = \min(\delta_L, \delta_R)$
- consider transverse pairs (P, Q) w/ dist $P \to$ middle $\leq \delta$ + $Q \to$ middle $\leq \delta$
- for any $P = (x_P, y_P)$, enough to look at pts w/ $y_P \leq y < y_P + \delta$
-



  - → enough to check dist (P, Q) for all Q in rectangle
  - → claim: at most 8 pts from init set (including P) in rectangle

    Proof by contradiction:

    Assume there can be 2 pts within 1 square of sidelength $\frac{\delta}{2}$. The max dist btwn these 2 pts is $\sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \sqrt{2(\frac{\delta^2}{4})} = \frac{\sqrt{2}}{2}\delta < \delta$

    Since each box is contained either all on left/right side, then this would mean there's 2 pts in L/R that have dist $< \delta$. This is a contradiction, which leads to above claim.
  - → only need to move up b/c box around upper pts (blue box) wouldn't include lower pts

↳ runtime analysis:

- init: sort pts twice wrt x (for median) + y (to sort pts from lowest to highest)
  - → one-time cost is $O(n \log n)$
- recursion:
  - → finding $x_{med} \in O(1)$
  - → for next recursive calls, split sorted lists $\in O(n)$
  - → remove pts at dist $\geq \delta$ from x-splitting line $\in O(n)$
  - → inspect all remaining pts in inc y-order + for each, compute dist to next 7pts + keep min $\in O(n)$
- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ so $T(n) \in \Theta(n \log n)$
- total runtime is $O(n \log n) + \Theta(n \log n) =$ $O(n \log n)$

e.g. quickselect (median of medians)

Median: given $A[0..n-1]$, find the entry that would be at index $\lfloor n/2 \rfloor$ if A was sorted

Selection: given $A[0..n-1]$ and k in $\{0, \ldots, n-1\}$, find the entry that would be at index k if A was sorted **Known results:**

sorting A in $O(n \log(n))$, or a simple randomized algorithm in expected time $O(n)$

↳ finding median is a special selection problem

quick-select(A, k)
A: array of size n, k: integer s.t. $0 \leq k < n$

```
1.  p ← choose-pivot(A)
2.  i ← partition(A, p)            i is the correct index of p
3.  if i = k then
4.      return A[i]
5.  else if i > k then
6.      return quick-select(A[0, 1, ..., i − 1], k)
7.  else if i < k then
8.      return quick-select(A[i + 1, i + 2, ..., n − 1], k − i − 1)
```

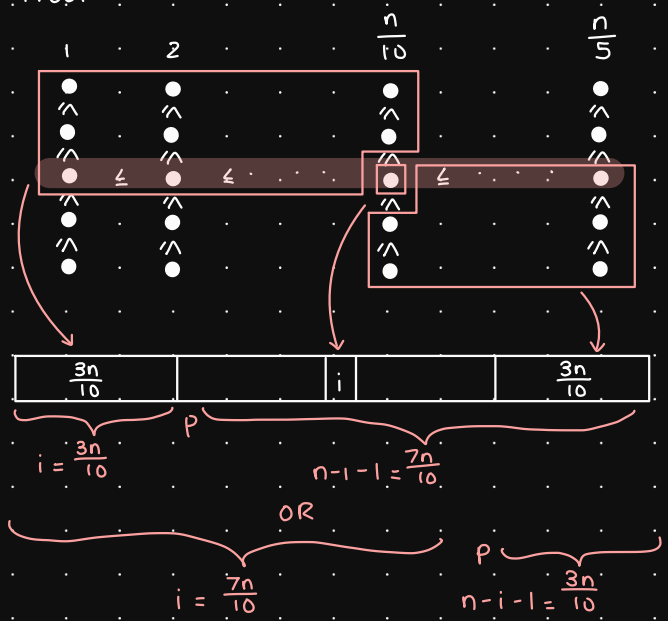
Question: how to find a pivot such that both i and $n - i - 1$ are not too large?

↳ worst-case runtime is $O(n^2)$ where smallest / largest elmt is repeatedly pivot

↳ use median of medians strategy:

- divide A into $\frac{n}{5}$ groups $G_1, \ldots, G_{n/5}$ of size 5
- find medians $m_1, \ldots, m_{n/5}$ of each group $\in O(n)$
- pivot p is median of medians $[m_1, \ldots, m_{n/5}]$
  $\rightarrow T(\frac{n}{5})$
- claim: w/ this choice of p, indices i & $(n-i-1)$ are max $\frac{7n}{10}$

Proof:



(groups not acc sorted, just drawn that way to count groups)

Half of the m_is are $\geq p$ & for each m_i, there's 3 elmts in $G_i \geq m_i$
So, at least $\frac{3n}{10}$ elmts are $\geq p$ & at most $\frac{7n}{10}$ elmts are $< p$. Then, i is at most $\frac{7n}{10}$ & same for $n-i-1$. So, $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$

This gives $T(n) \in O(n)$

$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$

$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10} + 6) + O(n)$

Proof by induction:

$$T(n) \leq \begin{cases} O(1) & n \leq 120 \\ T(\frac{n}{5}) + T(\frac{7n}{10} + 6) + O(n) & n > 120 \end{cases}$$

We show that $T(n) \leq cn$ for a large enough c and all $n > 0$. We know that if c is large enough, then $T(n) \leq cn$ for all $n \leq 120$. Choose a constant a to write $O(n)$ as an. Assume stmt. $T(m) \leq cm$ is correct for all $0 < m < n$.

$$T(n) \leq T(\tfrac{n}{5}) + T(\tfrac{7n}{10} + 6) + O(n)$$

$$\leq c(\tfrac{n}{5}) + c(\tfrac{7n}{10} + 6) + an$$

$$\leq c\tfrac{n}{5} + 7c\tfrac{n}{10} + 6c + an$$

$$\leq 9c\tfrac{n}{10} + 6c + an$$

$$\leq cn + (-c\tfrac{n}{10} + 6c + an) \longrightarrow c(6 - \tfrac{n}{10}) + an \leq 0$$

$$c(6 - \tfrac{n}{10}) \leq -an$$

$$c(\tfrac{n}{10} - 6) \geq an$$

$$c \geq 10a(\tfrac{n}{n-60})$$

We'll have $cn + (-c\frac{n}{10} + 6c + an) \leq cn$ if $c \geq 10a(\frac{n}{n-60})$. If $n > 120$, then $\frac{n}{n-60} \leq 2$ so we choose $c \geq 20a \geq 10a(\frac{n}{n-60})$. With this choice of c, it's true that $T(n) \leq cn$ for all $n > 0$ and we've proven the statement.

BREADTH FIRST SEARCH

GRAPHS

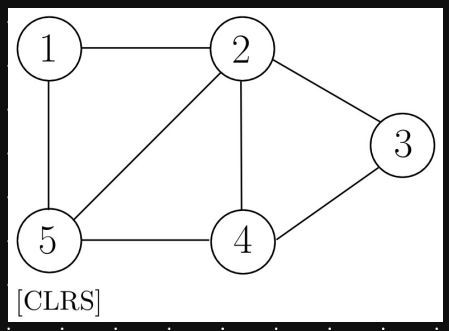
graph G is pair (V, E)
- ↳ V is finite set of vertices
- ↳ E is finite set of edges (i.e unordered pairs of distinct vertices)

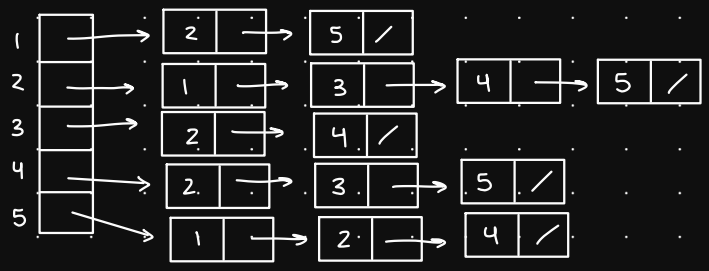
n is # vertices = |V|
m is # edges = |E|
data structs:
- ↳ adjacency list: arr A[1...n] st A[v] is linked list of all edges connected to v
  - 2m list cells (2 cells per edge)
  - total size $\theta(n+m)$
  - testing if edge exists is not O(1) but O(n)
  - 

  - use when graph's not full / sparse (less edges)
- ↳ adjacency matrix: (0,1) matrix M of size n×n w/ M[v,w] = 1 iff ⟨v, w⟩ is edge
  - size $\theta(n^2)$
  - testing if edge exists is O(1)
  - M = 

  - use if graph is almost complete / dense (more edges)

e.g.





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

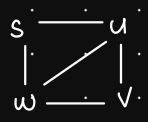
- **path**: a seq $v_1, \ldots, v_k$ of vertices, w/ $\{v_i, v_{i+1}\}$ in E for all i
  - ↳ $k = 1$ is ok
- **connected graph**: $G = (V, E)$ st $\forall v, w$ in V, there's a path $v \rightsquigarrow w$
- **cycle**: path $v_1, \ldots, v_k, v_1$ w/ $k \geq 3$ + v_is pairwise distinct (e.g. no 2 v_is are same)
- **tree**: connected graph w/o any cycle
- **rooted tree**: tree w/ special vertex called root
- **subgraph** of $G = (V, E)$: graph $G' = (V', E')$ where $V' \subseteq V$ + $E' \subseteq E$
  - ↳ all edges E' joining vertices V'
- **connected component** of $G = (V, E)$: connected subgraph of G that's not contained in larger connected subgraph of G
  - ↳ let $G_i = (V_i, E_i)$, $i = 1, \ldots, s$ be connected components of $G = (V, E)$
    - V_is are partition of V w/ $\Sigma_i n_i = n$ where $n_i = |V_i|$
    - E_is are partition of E w/ $\Sigma_i m_i = m$ where $m_i = |E_i|$

BREADTH-FIRST SEARCH

e.g. **breadth-first search (BFS)** of graph

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 visited be an array of size n, with all entries set to false
3.  enqueue(s, Q)
4.  visited[s] ← true
5.  while Q not empty do
6.      v ← dequeue(Q)
7.      for all w neighbours of v do
8.          if visited[w] is false
9.              enqueue(w, Q)
10.             visited[w] ← true
```



Q: □ V: | f | f | f | f |
s u v w

Q: s V: | t | f | f | f |
s u v w

Q: u w V: | t | t | f | t |
s u v w

Q: w v V: | t | t | t | t |
s u v w

Q: [v] V: [t][t][t][t] (s u v w)

Q: [ ] (empty) V: [t][t][t][t] (s u v w)

↳ each vertex enqueued / dequeued at most once $\in O(n)$
- vertices could be disconnected from s so it's nvr enqueued

↳ each adjacency list read at most once

↳ $\forall v$, d_v = # neighbours of v = length of $A[v]$ = deg of v
- total cost at line 7 (loop) is $O(\sum_v d_v) = O(m)$ b/c adjacency arr A has $2m$ cells (handshaking lemma)

↳ total runtime $O(n+m)$

$\forall$ vertices v, if visited[v] is T at end, there's path $s \rightsquigarrow v$ in G

↳ proof:

Let $s = v_0, \ldots, v_k$ be vertices for which visited is set to T, in this order.

We prove $\forall i$, there's a path $s \rightsquigarrow v_i$ by induction

Base case: $i = 0$

$v_0 = s$ + there's path from s to itself. Base case is T.

Inductive step: suppose T for $v_0, \ldots, v_{i-1}$

When visited[v_i] is set to T, we're examining neighbours of a v_j st $j < i$. Using hypothesis, there's a path $s \rightsquigarrow v_j$. Since $\{v_j, v_i\}$ is in E, there's a path $s \rightsquigarrow v_i$.

$\forall$ vertices v, if there's path $s \rightsquigarrow v$ in G, visited[v] is T at end

↳ proof:

Let $v_0 = s, \ldots, v_k = v$ be path $s \rightsquigarrow v$. We prove visited[v_i] is T $\forall i$

Base case: $i = 0$

visited[$v_0 = s$] is T.

Inductive step: suppose T for all $v_0, \ldots, v_i$

If visited[v_i] is T, then we'll examine all neighbours u of v_i. In particular, v_{i+1} is neighbour of v_i b/c $\{v_i, v_{i+1}\}$ is edge in path $s \rightsquigarrow v$.

So, visited[v_{i+1}] is T.

lemma: $\forall$ vertices v, there's a path $s \rightsquigarrow v$ in G iff visited[v] is T at end

↳ applications in $O(n+m)$:
- testing if there's path $s \rightsquigarrow v$
- testing if G's connected

for connected graph, $m \geq n - 1$

BFS tree T is subgraph made of

↳ all w st parent[w] $\neq$ NIL

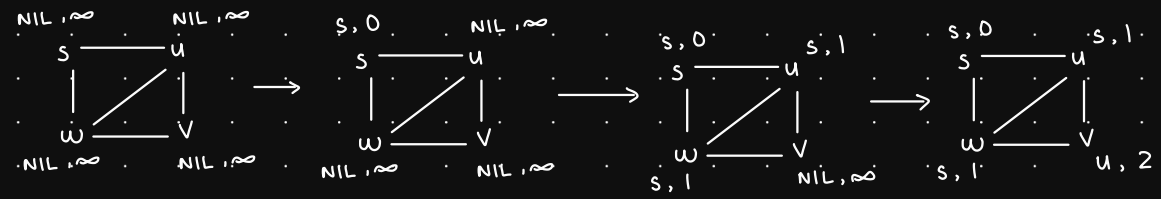
↳ all edges $\{w, \text{parent}[w]\}$ for w as above, except $w = s$

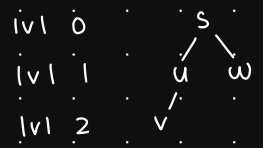
↳ to make rooted tree, choose s as root

```
BFS(G, s)
1.    let Q be an empty queue
2.    let parent be an array of size n, with all entries set to NIL
3.    let level be an array of size n, with all entries set to ∞
4.    enqueue(s, Q)
5.    parent[s] ← s
6.    level[s] ← 0
7.    while Q not empty do
8.        v ← dequeue(Q)
9.        for all w neighbours of v do
10.           if parent[w] is NIL
11.               enqueue(w, Q)
12.               parent[w] ← v
13.               level[w] ← level[v] + 1
```

↳ don't need visited arr b/c we can check if parent[v_i] is NIL or not
- if NIL, v_i hasn't been visited

↳ e.g.





- BFS tree T is tree
  ↳ proof by induction on vertices for which parent[v] is not NIL:
  - Base case:
    When we set parent[s] ← s, only 1 vertex so it's a tree.
  - Inductive step: suppose stmt is T before we set parent[w] ← v
    v was in T before + w was not so we add 1 vertex w + 1 edge {v, w} to T. This doesn't create cycle b/c w would've alr been connected to some other vertex that's not v in T. This is a contradiction so T remains a tree
- sub-claim 1: lvls in queue are non-dec
- sub-claim 2: ∀ vertices u, v, if there's edge {u, v}, then level[v] ≤ level[u] + 1

  ↳ proof:
  - Case 1: if v is dequeued before u, level[v] ≤ level[u] b/c sub-claim 1
  - Case 2: if u is dequeued before v, then:
    - parent[v] = u so level[u] = level[u]
    - parent[v] was dequeued before u (since another vertex had to have

put it in the queue) so since level[parent[v]] ≤ level[u], then level[v] = level[parent[v]] + 1 ≤ level[u] + 1

claim: ∀ v in G, there's a path s⇝v in G iff there's a path s⇝v in T (shortest path)
↳ if so, path in T is shortest path + level[v] = dist(s,v)

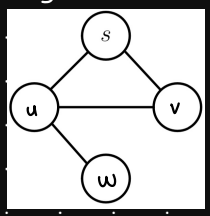
- proof:
  We know dist(s,v) ≤ level[v] by following path on T
  ∀ i, v, if there's path s⇝v of length i, then level[v] ≤ i
  Base case: i = 0
  Path of length 0, which is s⇝s + level[s] = 0.
  Inductive step: suppose T for i-1
  We take a path s⇝v of length i + let u be vertex before v.
  Using hypothesis level[u] ≤ i-1 so level[v] ≤ i using sub-claim 2.
  Thus, level[v] ≤ dist(s,v).

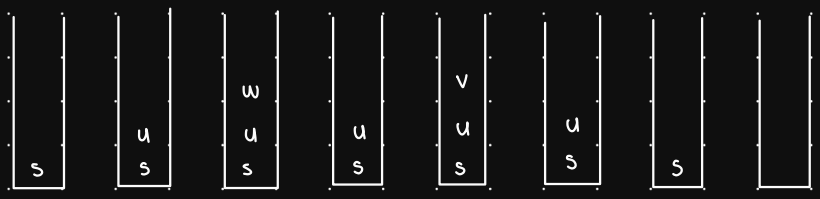
graph G = (V, E) is bipartite if there's partition V = $V_1 \cup V_2$ st all edges have 1 end in V_1 + 1 end in V_2

claim: suppose G is connected, run BFS from any s, w/ sets V_1 = vertices w/odd lvl + V_2 = vertices w/even lvl, then G is bipartite iff all edges have 1 end in V_1 + 1 end in V_2

DEPTH FIRST SEARCH

- idea of going depth-first:
  - ↳ travel as deep as possible
  - ↳ when we can't go further, backtrack
- depth-first search (DFS) based on stacks, either implicitly (recursion) or explicitly (as BFS does w/queues)
- e.g.





- recursive algo

DFS(G)
G: a graph with n vertices, given by adjacency lists
1. let visited be an array of size n, with all entries set to **false**
2. **for all** v in G
3. **if** visited[v] is **false**
4. **explore**(v)

explore(v)
1. visited[v] = **true**
2. **for all** w neighbour of v **do**
3. **if** visited[w] = **false**
4. **explore**(w)

Remark: can add parent array as in BFS

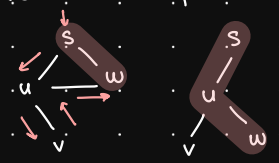
- white path lemma: when exploring vertex v, any w that's connected to v by an unvisited path will be visited during explore(v)
  - ↳ proof:
    - Let $v_0 = v, v_1, \ldots, v_k = w$ be path $v \rightsquigarrow w$, $w/\ v_1, \ldots, v_k$ not visited. We prove all v_is are visited before explore(v) is finished.
    - Base case: $i = 0$
    - True for v b/c it's starting vertex.
    - Inductive step: Suppose T for all $i < k$.
    - When we visit v_i, explore(v) is not finished + v_{i+1} is one of its neighbours.
      - ↳ if visited[v_{i+1}] = T, then we've alr visited it
        - could've been visited by prev vertex

↳ if visited[v_{i+1}] = F, then we'll visit it right now by calling explore(v_{i+1})
- will be done before explore(v) is finished

claim: if w is visited during explore(v), there's a path $v \rightsquigarrow w$

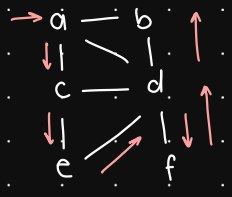
after calling explore at $v_1, \dots, v_k$ in DFS, we visited exactly connected components containing $v_1, \dots, v_k$

↳ can't find shortest path
- counterexample:

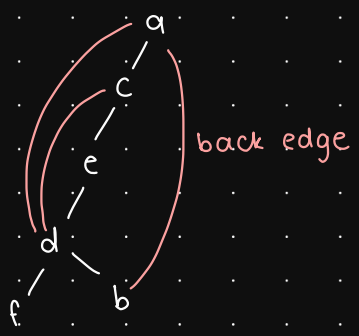


runtime of DFS is $O(n+m)$

e.g.



DFS tree:



DFS(G) gives partition of G into vertex-disjoint rooted trees $T_1, \dots, T_k$ (i.e. **DFS forest**)

suppose DFS forest is $T_1, \dots, T_k$ + let u, v be 2 vertices:

↳ u is **ancestor** of v if they're on same T_i + u is on path: root $\rightsquigarrow$ v

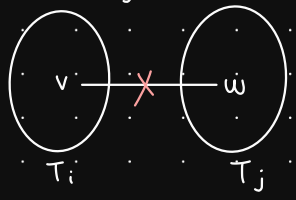
↳ i.e. v is **descendant** of u

claim: all edges in G connect a vertex to one of its descendants / ancestors

↳ i.e. there exists no edge btwn 2 DFS trees in G

↳ proof:

Let {v, w} be edge + suppose we visit v first. When we visit v, (v, w) is unvisited path btwn v + w so w has to be descendant of v according to white path lemma.



back edge is edge in G connecting ancestor to descendant, which is not a tree edge

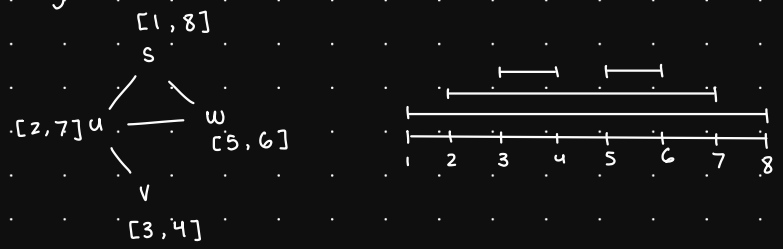
↳ key property: all edges in G are either tree or back edges

e.g. start + finish times

Set a variable t to 1 initially, create two arrays start and finish, and change **explore**:

```
explore(v)
1.    visited[v] = true
2.    start[v] = t
3.    t++        ← inc after using t
4.    for all w neighbour of v do
5.        if visited[w] = false
6.            explore(w)
7.    finish[v] = t
8.    t++        ← inc after using t
```

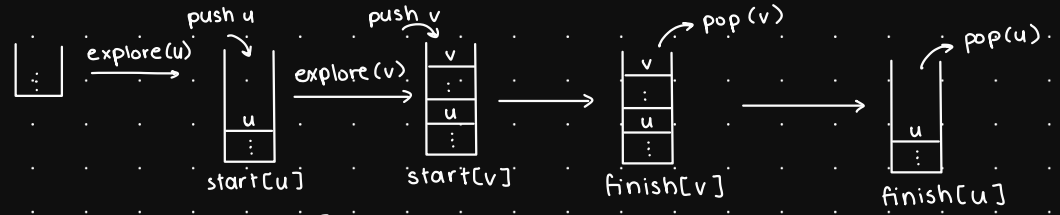
↳ e.g.

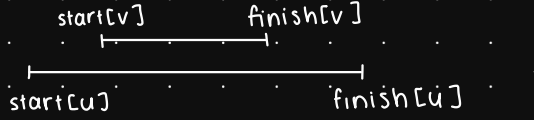


↳ time intervals are either contained in one another or disjoint

- 
- i.e. if start[u] $^{1}<$ start[v], then either finish[u] $^{2}<$ start[v] or finish[v] $^{3}<$ finish[u]
  - ↳ proof: if 1 and not 2, then 3

    start[u] < start[v] + finish[u] > start[v]

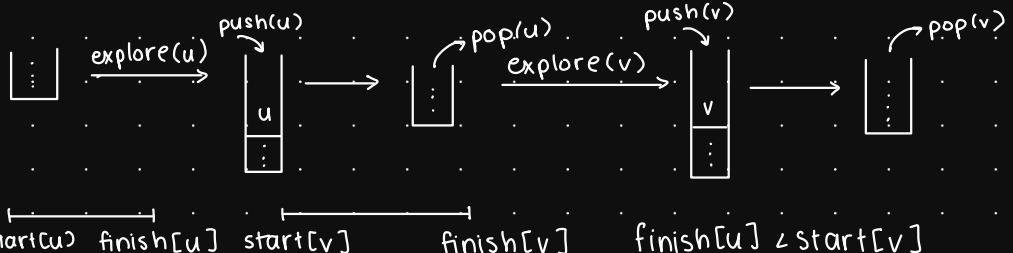




finish[v] < finish[u]

  - ↳ proof: if 1 and not 3, then 2

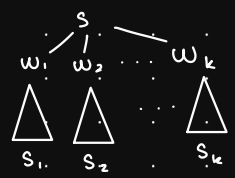
    start[u] < start[v] + finish[v] > finish[u]



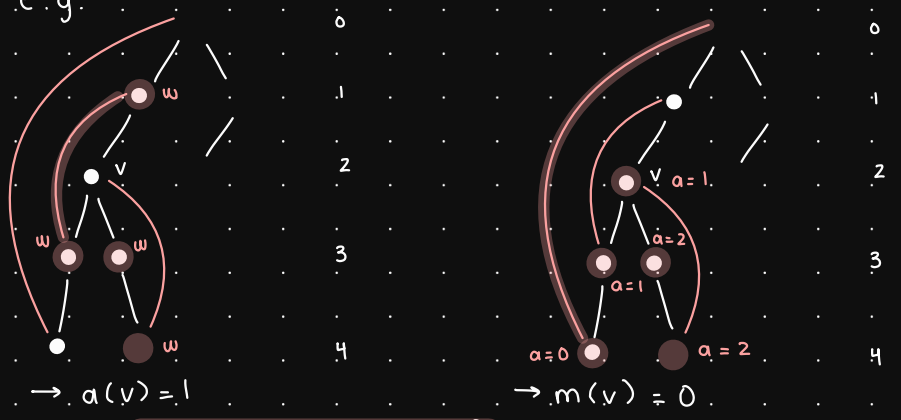
CUT VERTICES

- for G connected, vertex v in G is **cut vertex** if removing v (+ all edges that contain it) makes G disconnected
  - ↳ aka articulation pts
- setup: start from rooted DFS tree T, knowing parent + lvl
- **claim**: root s is vertex iff it has ≥ 1 child
  - ↳ proof:
    - if s has 1 child, removing s leaves T connected so s is not cut vertex

    - suppose s has subtrees $S_1, \ldots, S_k$, $k > 1$
      - → key property: no edge connecting S_i to S_j for $i \neq j$ so removing s creates k connected property

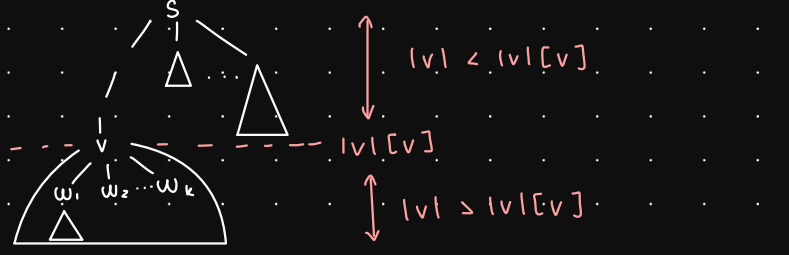


- finding cut vertices which aren't root:
  - ↳ for vertex v + any neighbour w, let:
    - $a(v) = \min\{lvl[w] \text{ for } \{v, w\} \text{ edge}\}$
      - → $a(v)$ is lowest lvl v is connected to
    - $m(v) = \min\{a(w) \text{ for } w \text{ descendant of } v\}$
      - → $m(v)$ is lowest lvl v's descendants is connected to
    - e.g.



→ $a(v) = 1$ → $m(v) = 0$

  - ↳ note v is **descendant of itself**
- **claim**: $\forall v$ except root, v is cut vertex iff it has at least 1 child w w/ $m(w) \geq lvl[v]$



↳ proof:

$\Rightarrow$:

Assume v is cut vertex. Suppose all children w of v have $m(w) < lvl[v]$. This means that any of v's subtrees has back edge that goes to lvl above v. Once v is removed, G is still connected, which is a contradiction since v is supposed to be cut vertex.

$\Leftarrow$:

Assume v has child w w/ $m(w) \geq lvl[v]$. After removing v, subtree rooted at w will be disconnected from G. This happens since all edges of G are either a tree / back edge. So, if the subtree rooted at w was connected after removing v, it should have had an edge going above lvl of v, which means $m(w) < lvl[v]$. That's a contradiction so v is a cut vertex.

if v has children $w_1, \ldots, w_k$, then $m(v) = \min\{a(v), m(w_1), \ldots, m(w_k)\}$

↳ computing $a(v)$ is $O(d_v)$

- d_v = deg of v

↳ knowing all $m(w_1), \ldots, m(w_k)$, then $m(v) \in O(d_v)$

↳ all vals $m(v)$ can be computed in $O(m)$

- $O(n+m) = O(m)$ when G is connected b/c $m = n-1$ at least so #edges m dominates in connected graph

↳ testing cut-vertex condition at v is $O(d_v)$

↳ testing all v is $O(m)$

DIRECTED GRAPHS

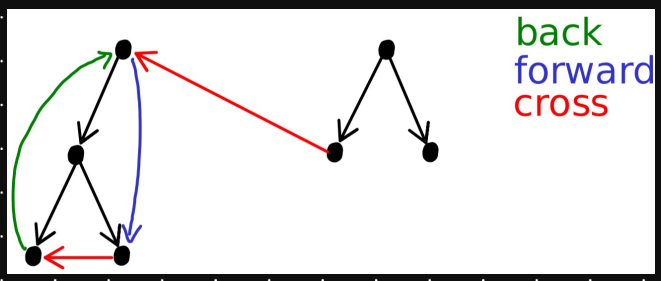
- $G = (V, E)$ is same as undirected defn w/ diff that edges are directed pairs (v, w)
  - ↳ edges aka **arcs**
  - ↳ usually allow **loops** w/ $v = w$
  - ↳ v is source node & w is target
- **path** is seq $v_1, \ldots, v_k$ of vertices w/ (v_i, v_{i+1}) in E for all i
  - ↳ $k = 1$ is ok
- **cycle** is path $v_1, \ldots, v_k, v_1$
  - ↳ $k \geq 1$
- **directed acyclic graph (DAG)** is directed graph w/o cycles
  - ↳ e.g. DAG

  - ↳ e.g. not DAG

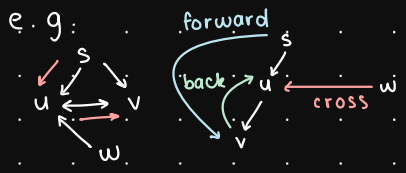
- **in-deg** of v is #edges of form (u, v)
- **out-deg** of v is #edges of form (v, w)
- data structs:
  - ↳ adjacency lists
  - ↳ adjacency matrices
    - not symmetric, unlike undirected graphs

DFS FOR DIRECTED GRAPHS

- algos all work w/o change for BFS & DFS (we focus on **DFS**)
- when doing DFS for directed graphs:
  - ↳ obtain partition of V into vertex-disjoint trees $T_1, \ldots, T_k$
  - ↳ **white path lemma**: when we start exploring vertex v, any w w/ unvisited path $v \rightsquigarrow w$ becomes descendant of v
  - ↳ start & finish time properties
  - ↳ but, edges connecting trees T_i & T_j can exist
- suppose we have DFS forest, then edges of G are:
  - ↳ **tree edges**
  - ↳ **back edges**: from descendant to ancestor
  - ↳ **fwd edges**: from ancestor to descendant that's not tree edge
  - ↳ **cross edges**: all others
- e.g.



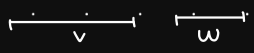
e.g.:




```
explore(v)
1.   visited[v] = true
2.   start[v] = t, t++
3.   for all w neighbour of v do
4.       if visited[w] = false
5.           explore(w)              (v, w) tree edge
6.   finish[v] = t, t++
```

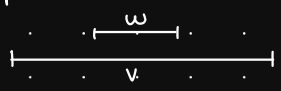
NOTE
v is descendant of itself

↳ for given edge (v, w), 4 cases for start + finish times while we explore v

1) 

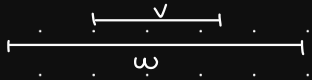
finish[v] < start[w]

Not possible b/c when we explore v, w is unvisited + by white path lemma, w will be visited before exploration of v is over.

2) 

start[v] < start[w] < finish[w] < finish[v]

If we visit w while we're checking neighbours of v + this happens directly from v, then v is parent of w + (v, w) is tree edge. Else, w is visited thru another neighbour of v + this means w is descendant of v. So, (v, w) is fwd edge.

3) 

start[w] < start[v] < finish[v] < finish[w]

w was visited but not finished. This means we're visiting v thru neighbour of w + this implies v becomes descendant of w. Hence, (v, w) is back edge.

4) 

finish[w] < start[v]

w was visited + finished before we started explore(v). Then, v + w aren't descendants of each other so they belong to 2 diff trees. Hence, (v, w) is cross edge.

claim: G has cycle iff there's back edge in DFS forest

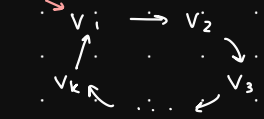
↳ proof:

⇐:

Since there's back edge, we have edge from descendant to ancestor. So, there's cycle $v \rightsquigarrow w \to v$.

$\Rightarrow$:

Assume G has cycle $v_1, v_2, \ldots, v_k, v_1$. WLOG, we can assume we visit v_1 first in cycle.

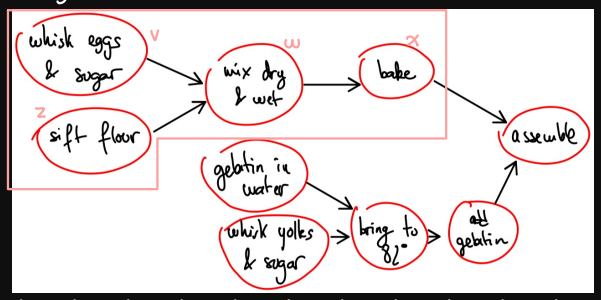


At the time we start v_1, path $v_2, \ldots, v_k$ is unvisited. By white path lemma, we'll visit v_k before we finish explore(v_1). Hence, v_k becomes descendant of v_1 so there's back edge from v_k to v_1.

suppose $G = (V, E)$ is DAG, then **topological order** is ordering $\prec$ of V st for any edge (v, w), we have $v \prec w$

↳ no order if there's cycles

↳ e.g.



- a topological order $v \prec z \prec w \prec x$ or $z \prec v \prec w \prec x$

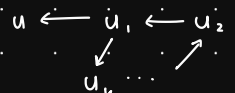
claim: G is DAG $\Leftrightarrow$ there's topological ordering on V

↳ not a formal proof for $\Rightarrow$ (rough sketch):

Claim: G must have at least 1 vertex w/ in-deg 0.

- proof:

Assume contrary stmt so all $v \in V$ have in-deg ≥ 1.



(must reuse existing node to make in-deg of $n_k \geq 1$ b/c we have finite nodes)

We have contradiction.

Take node w/ in-deg = 0 as first vertex in order. Remove from G & remaining part is DAG. Repeat to get entire topological order.

finding topological order from **DFS forest**:

↳ e.g.

[start time, end time]

u [3,4]

↓

v [1,2]

DFS starts here

- order: $u < v$

↳ e.g.

u [1,4] ← DFS starts here

↓

v [2,3]

- order: $u < v$

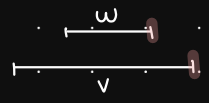
↳ start times are unhelpful

↳ topological order can be found from dec order of finish times

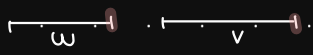
claim: suppose V is ordered using reverse of finishing order, then $v < w \iff$ finish[w] < finish[v] is a topological order

↳ proof: for any edge $(v, w) \in E \Rightarrow (v < w \iff$ finish[w] < finish[v])

If we discover v before w, w will become descendant of v (white path lemma) & we'll finish exploring it before we finish v.



If we discover w before v, there's no path $w \rightsquigarrow v$ b/c G is DAG. Then, we'll finish w before we start v.



↳ topological order in $O(n+m)$

- no sorting needed for finish times b/c we can add them on as we finish exploring each node

directed graph G is strongly connected if $\forall v, w$ in G, there's a path $v \rightsquigarrow w$ & path $w \rightsquigarrow v$

↳ claim: G is strongly connected $\iff$ there exists s that $\forall w$, there's paths $s \rightsquigarrow w$ & $w \rightsquigarrow s$

- proof:

$\Rightarrow$:

Defn of a strongly connected graph.

$\Leftarrow$:

Take vertices v & w. We have paths $v \rightsquigarrow s$ & $s \rightarrow w$, so $v \rightsquigarrow w$. Same applies for $w \rightsquigarrow v$.

testing strong connectivity algo:
- ↳ call explore 2x, starting from some vertex s
- ↳ edges reversed 2nd time (i.e transposed graph)
- ↳ e.g.

NOTE
Consider how to reverse graph in linear time

- ↳ correctness:
  - 1st run tells whether ∀v, there's path s ⇝ v
  - 2nd run tells whether ∀v, there's path s ⇝ v in G_T so there's path v ⇝ s in G
- ↳ test in O(n+m)

strongly connected component of G is subgraph of G that's strongly connected itself, but not contained in larger strongly connected subgraph of G

directed graph G is DAG of disjoint strongly connected components
- ↳ e.g.

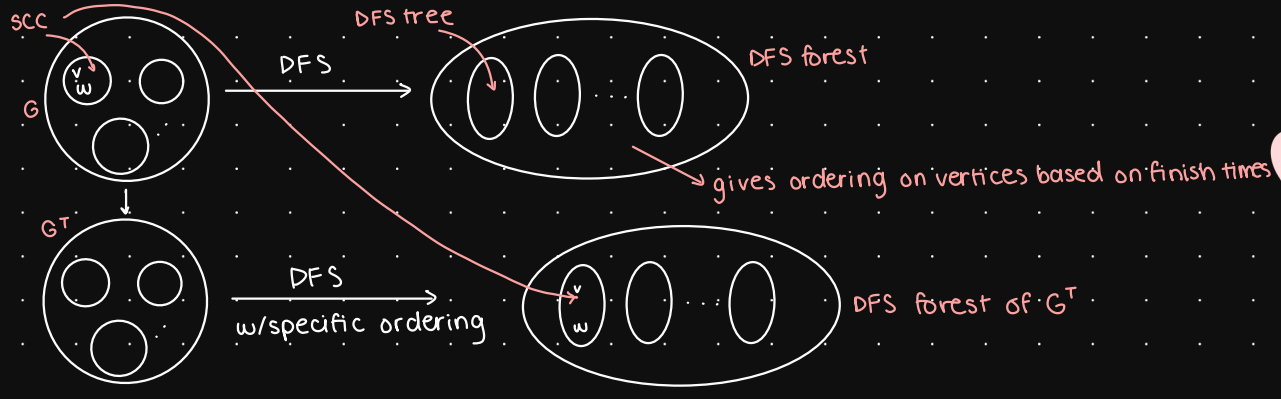
for directed graph G = (V, E), reverse/transpose $G^T = (V, E^T)$ is graph w/ same vertices + reversed edges

Kosaraju's algo for strongly connected components:

SCC(G)
1. run a DFS on G and record finish times
2. run a DFS on G^T, with vertices ordered in **decreasing finish time**
3. return the trees in the DFS forest of G^T

- ↳ runtime is O(n+m)
  - includes cost of reversing G

claim: ∀v, w, v + w are in[1] same SCC of G ⟺ v + w are in same[2] tree in DFS forest of G^T (w/vertices ordered in dec finish time)



↳ proof:

1 ⇒ 2:

Order of vertices don't matter. Let C be SCC of G that contains v + w. Let s be 1st vertex of C visited in DFS of G^T. There's a path $s \rightsquigarrow v$ in G^T + all vertices on this path are in C. All of them are unvisited when we arrive at s so v becomes descendant of s b/c white path lemma. This applies to w as well. Thus, v + s are in same tree of G^T DFS forest. Same for w + s.

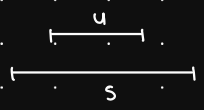
2 ⇒ 1:

Let T be tree in DFS forest of G^T containing v + w, w/ root s. We prove that ∀vertex t in T, s + t are in same SCC of G.

1) $\forall t \in T$, there's a path $s \rightsquigarrow t$ in G^T so there's a path $t \rightsquigarrow s$ in G.
2) $\forall t \in T$, t is descendant of s in DFS forest of G^T, which gives path $s \rightsquigarrow t$ in G.
   ↳ proof: by induction
   Base case: t is s itself
   There's path of $s \rightsquigarrow s$ in both G^T + G.
   Inductive step:
   Suppose this is true for some t in T + prove it's true for its children.
   Let u be child of t in T.

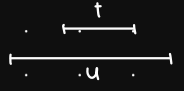
Based on ordering in 2nd round of DFS + fact that we visited s before u, we can conclude that in 1st round of DFS, finish[u] < finish[s].

Case 1:

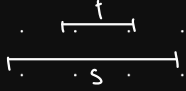


Case 2:

1) u is descendant of s in G so there's a path $s \rightsquigarrow u$.
2) $(t, u) \in T \subseteq G^T$ so $(u, t) \in G$. By white path lemma + fact that t is unvisited at time of exploration of u, we'll visit t before we're done w/ explore(u) (due to induction hypothesis). Hence, we have start[u] < start[t] < finish[t] < finish[u].



which is a contradiction to



Thus, case 2 is impossible.

hamiltonian path in graph is simple path that visits every vertex exactly once

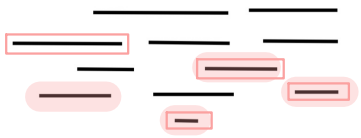
↳ for undirected graph G, there's no linear time algo for deciding whether G contains hamiltonian path (NP-complete)

↳ for DAG, there's linear time algo for deciding whether G contains hamiltonian path
- run DFS algo to get topological ordering $v_1, v_2, \ldots, v_n$ & check if this is a hamiltonian path
- can only find hamiltonian path iff there's a unique topological ordering

GREEDY ALGORITHMS

- computational model is word RAM
  - ↳ assume all weights, capacities, deadlines, etc. fit in a word
- trying to solve **combinatorial optimization** problem:
  - ↳ large but finite domain D
  - ↳ find elmt E in D that mins/maxes cost fcn
- e.g. **Huffman tree**: given freq. $f_1, \ldots, f_n$ for chars $c_1, \ldots, c_n$, build binary tree for whole code
  - ↳ greedy strategy: build tree bottom up
    - create many single letter trees
    - define freq of tree as sum of freqs of letters in it
    - build final tree by putting tgt smaller trees by joining 2 trees w/ least freqs
  - ↳ minimizes $\sum_i f_i \times$ (length of encoding of c_i) (i.e. total weighted path length to c_i)
- **interval scheduling** problem:
  - ↳ input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$
  - ↳ output: maximal subset of disjoint intervals
  - ↳ e.g.

Example:



Example: A car rental company has the following requests for a given day:

I_1: 2pm to 8pm
I_2: 3pm to 4pm
I_3: 5pm to 6pm

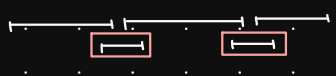
Answer is $S = [I_2, I_3]$.

  - # items of in maximal subset is unique but subsets aren't

GREEDY STRATEGIES

- consider **earliest starting time** (i.e. choose interval w/ $\min_i s_i$)
  - ↳ doesn't work counterexample:

- consider **shortest interval** (i.e. choose interval w/ $\min_i (f_i - s_i)$)
  - ↳ doesn't work counterexample:



- consider **min conflicts** (i.e. choose interval that overlaps w/ min # of other intervals)
  - ↳ doesn't work counterexample:

consider **earliest finishing time** (i.e. choose interval w/ $\min_i f_i$)

↳ interval scheduling algo:

1) $S = \emptyset$
2) Sort intervals st $f_1 \leq f_2 \leq \ldots \leq f_n$ $\in O(n \log n)$
3) for $i \leftarrow 1$ to n $\in O(n)$
   if interval $i: [s_i, f_i]$ has no conflicts w/ intervals in S
   add i to S
4) return S

↳ in 3, only need to consider if i conflicts w/ last interval in S b/c they're sorted in ascending finishing time

- if i doesn't conflict w/ last one, then since last one didn't conflict w/ others, i is safe to add (inductive assumption)

correctness: greedy algo stays ahead

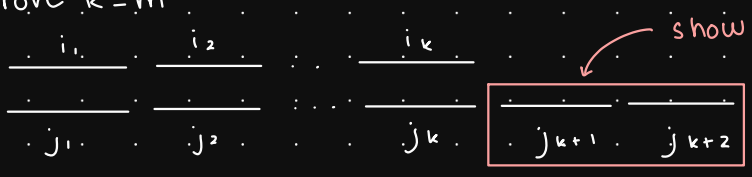
↳ assume O is optimal soln & we'll show $|S| = |O|$

↳ suppose $i_1, i_2, \ldots, i_k$ are intervals in S in order they're added to $|S|$ by greedy algo.

↳ let intervals in O are denoted by $j_1, \ldots, j_m$

- intervals are ordered by start & finish times

↳ prove $k = m$



↳ **lemma**: $\forall r \leq k$, we have $f(i_r) \leq f(j_r)$

- proof by induction:

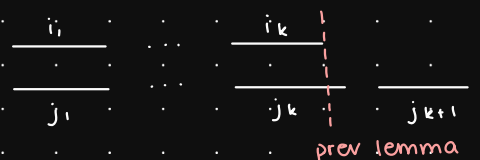
Base case:
For $r = 1$, stmt is true b/c greedy algo always chooses earliest finishing time.

Inductive step:
Suppose $r > 1$ & stmt is true for $r-1$. We'll show stmt is true for r.
By inductive hypothesis, $f(i_{r-1}) \leq f(j_{r-1})$ 1
Compare j_{r-1} & j_r: $f(j_{r-1}) < s(j_r)$ 2
From 1 & 2: $f(i_{r-1}) < s(j_r)$
At the time greedy algo had to choose r^{th} interval, j_r was an option b/c it had no intersection w/ i_{r-1} & others. But, greedy algo chose i_r. Since greedy algo choose interval w/ earliest finish time, then $f(i_r) \leq f(j_r)$ We've proven the stmt

↳ **thm**: greedy algo returns optimal soln

- proof by contradiction:

Show $|S| = |O|$. If S is not optimal, then assume $|S| < |O|$.
There exists at least j_{k+1} in O b/c O must have at least 1 more interval in comparison to S.



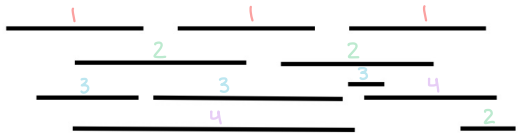
Apply prev lemma w/ $r = k$, then we get $f(i_k) \leq f(j_k)$. Then, $f(i_k) \leq f(j_k) < s(j_{k+1})$. So, j_{k+1} was a possible choice to add to S by greedy algo, but it didn't. This is contradiction to greedy algo.

e.g. interval colouring

Interval Coloring Problem

Input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$
Output: use the minimum number of colors to color the intervals, so that each interval gets one color and two overlapping intervals get two different colors.

Example:



↳ algo.

1) sort intervals by starting time: $s_1 \leq s_2 \leq \ldots \leq s_n$ $\in O(n \log n)$
2) for $i \leftarrow 1$ to n, use min avail colour c_i to colour interval i $\in O(n^2)$
   → i.e. use min # to colour interval i so that it doesn't conflict w/ colours of intervals that are alr coloured

↳ correctness:

Assume greedy algo uses k colours. We'll show there's no other way to solve problem using at most $k-1$ colours (i.e. k is min # colours). Show that there exists time t st it's contained in k intervals.

Assume ℓ is 1st interval to use colour k. Interval ℓ overlaps w/ intervals w/ colours $1, \ldots, k-1$. Call these intervals $[s_{i_1}, f_{i_1}], \ldots, [s_{i_{k-1}}, f_{i_{k-1}}]$. For $1 \leq j \leq k-1$, $s_{ij} \leq s_\ell$ b/c we have inc order of start times. Since all these intervals overlap w/ $[s_\ell, f_\ell]$, then $s_\ell \leq f_{ij}$ for $1 \leq j \leq k-1$. Hence, s_ℓ is time contained in k intervals so there's no $k-1$ colouring.

e.g. minimizing total completion time

The problem

Input: n jobs, each requiring processing time p_i
Output: An ordering of the jobs such that the total completion time is minimized.

Note: The completion time of a job is defined as the time when it is finished.

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

2	8	1	10	5	
2	2	2	2	2	$5 \cdot 2$
	8	8	8	8	$4 \cdot 8$
		1	1	1	$3 \cdot 1$
			10	10	$2 \cdot 10$
				5	$1 \cdot 5$
2	10	11	21	26	70

Optimal:

1	2	5	8	10	
1	1	1	1	1	$5 \cdot 1$
	2	2	2	2	$4 \cdot 2$
		5	5	5	$3 \cdot 5$
			8	8	$2 \cdot 8$
				10	$1 \cdot 10$
1	3	8	16	26	54

↳ algo: order jobs in non-dec processing times

↳ proof of correctness:

Let $L = [e_1, \ldots, e_n]$ be optimal soln w/ diff ordering in comparison to soln given by greedy algo. Then, L is not in non-dec order of processing times. So, there exists i st $t(e_i) > t(e_{i+1})$.

e_1	e_2	. . .	e_{i-1}	e_i	e_{i+1}	. . .	e_n	
$t(e_1)$	$t(e_1)$		$t(e_1)$	$t(e_1)$	$t(e_1)$		$t(e_1)$	$n\, t(e_1)$
	$t(e_2)$		$t(e_2)$	$t(e_2)$	$t(e_2)$			$(n-1)\, t(e_2)$
			$t(e_{i-1})$	$t(e_{i-1})$	$t(e_{i-1})$			
				$t(e_i)$	$t(e_i)$			
					$t(e_{i+1})$			
							$t(e_n)$	$t(e_n)$

Contribution of e_i & e_{i+1} in L is $(n-i+1)\, t(e_i) + (n-i)\, t(e_{i+1})$.

Switch e_i & e_{i+1} to get L'. Contribution of e_i & e_{i+1} in L' is

$(n-i+1)t(e_{i+1}) + (n-i)t(e_i)$. Let T rep total completion time.

$$T(L') - T(L) = (n-i+1)t(e_{i+1}) + (n-i)t(e_i) - ((n-i+1)t(e_i) + (n-i)t(e_{i+1}))$$

$$= t(e_{i+1}) - t(e_i) < 0 \leftarrow \text{b/c } t(e_i) > t(e_{i+1})$$

This is a contradiction b/c that would be L' is better soln than optimal one.

DIJKSTRA'S ALGORITHMS

PRELIMINARIES

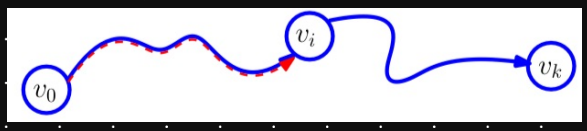
- $G = (V, E)$ is a directed graph w/ weight fcn $w: E \to \mathbb{R}$
  - ↳ weight of path $P = \langle v_0, \ldots, v_k \rangle$ is $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- shortest path exists in any directed weighted graph only if G has no -ve weighted cycles b/c it can infinitely become lower
  - ↳ assuming G has no -ve weight cycles, shortest path weight from u to v is

$$\delta(u,v) = \begin{cases} \min\{w(P) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- e.g.

Single-Source Shortest Path Problem
Input: $G = (V, E)$, $w : E \to \mathbb{R}$ and a source $s \in V$
Output: A shortest path from s to each $v \in V$

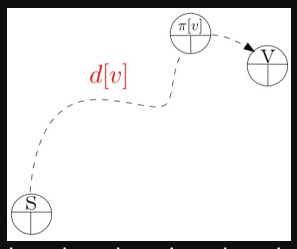
  - ↳ if $\langle v_0, v_1, \ldots, v_k \rangle$ is shortest path from v_0 to v_k, then $\langle v_0, v_1, \ldots, v_i \rangle$ is shortest path from v_0 to v_i, $\forall\, 0 \le i \le k$



    - if there's another shorter path from v_0 to v_i, then we would've found a shorter path from v_0 to v_k, which is a contradiction

DIJKSTRA'S ALGORITHM

- Dijkstra's algo is greedy algo
  - ↳ input: weighted directed graph w/ non-ve edge weights
  - ↳ $\forall$ vertices, maintain quantities:
    - $d[v]$: shortest path estimate from s to v (best path so far)
      - → init entries to ∞
      - → similar to lvl arr in BFS/DFS
    - $\pi[v]$: predecessor in path
      - → init entries to NIL
      - → similar to parent arr in BFS/DFS
  - ↳

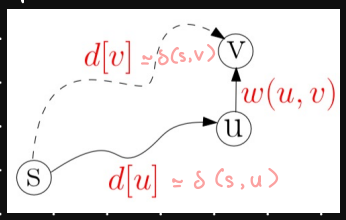


- process of Dijkstra's algo: init $C = \emptyset$; repeat until $C = V$ (i.e. all vertices are covered):
  - ↳ add $u \in V - C$ w/ smallest d-val to C

↳ update d-vals of vertices v w/ $(u,v) \in E$ st $d[v] \leftarrow \min\{d[v], d[u] + w(u,v)\}$

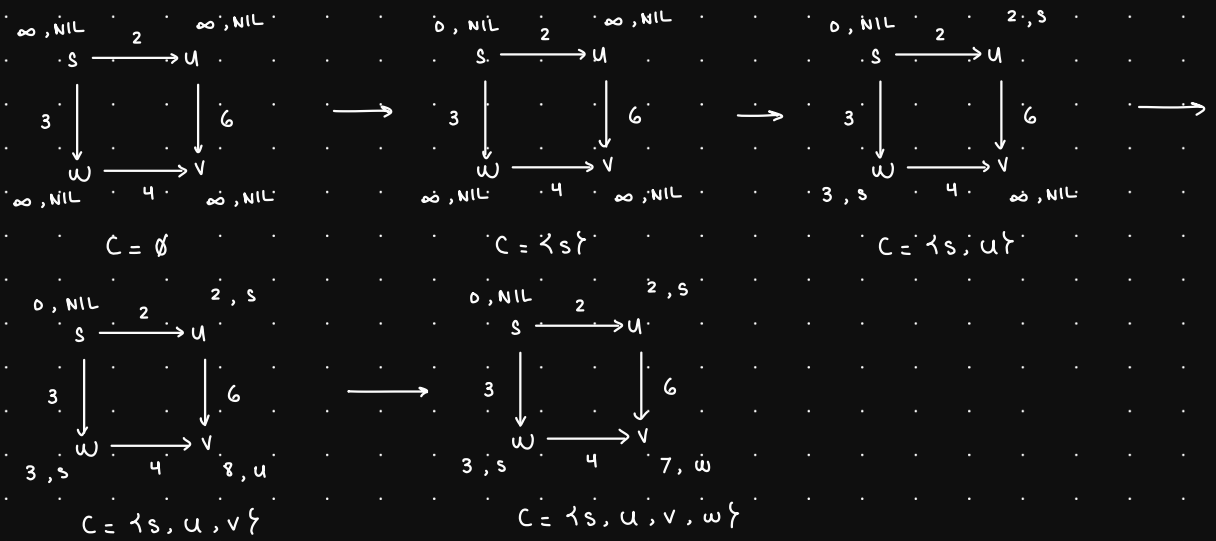
↳ update $\pi[v]$ if $d[v]$ is changed

↳



NOTE
This rltnship must always hold:
$\delta(s,v) \leq \delta(s,u) + w(u,v)$
$d[v] \leq d[u] + w[u,v]$

↳ e.g.



use bin **min-heap** for vertices

↳ insert $\in O(\log n)$

↳ extract-min $\in O(\log n)$

↳ update-key $\in O(\log n)$

- we have $O(1)$ ref handler to every node in heap

pseudocode

```
DIJKSTRA(G, w, s)
1   for each vertex v ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
5   C ← ∅          ← not needed for implementation
6   Q ← V[G]
7   while Q ≠ ∅
8       u ← EXTRACT-MIN(Q)
9       C ← C ∪ {u}
10      for each vertex v ∈ Adj[u]
11          if d[v] > d[u] + w(u, v)
12              d[v] ← d[u] + w(u, v)
13              π[v] ← u
```

↳ last for loop is **relaxation** step, meaning we check all neighbours of u st $d[v]$ will always be the curr shortest path from s to v

↳ e.g.

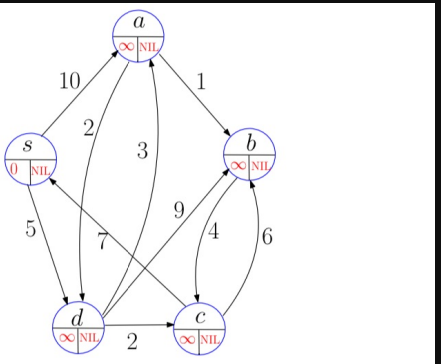


Figure: An example from CLRS

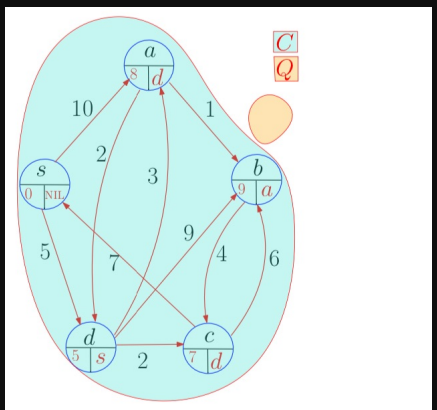


Figure: An example from CLRS

↳ complexity analysis:

line	Array Imp.	Heap Imp.
1 2 3 4 5	$O(\|V\|)$	$O(\|V\|)$
6	$O(\|V\|)$	$O(\|V\|)$
7	$O(\|V\|)$ iter.	$O(\|V\|)$ iter.
8 9	$O(\|V\|^2)$	$O(\|V\| \log \|V\|)$
10	$O(\|E\|)$ iter.	$O(\|E\|)$ iter.
11 12 13	$O(\|E\|)$	$O(\|E\| \log \|V\|)$
Tot.	$O(\|V\|^2)$	$O((\|V\| + \|E\|) \log \|V\|)$

Note: The of number of iterations on lines 7 and 10, are already considered in the cost of lines 8-9

- arr: dense graph
- heap: sparse graph
  - ↳ max #edges in directed graph is $n(n-1) \in O(|V|^2)$ so runtime would be $O(|V|^2 \log(|V|)$ for dense graph

proof of correctness claim: $\forall v \in V$, we have $d[v] = \delta(s,v)$ at the time when v is added to set C

↳ proof by contradiction: assume claim is incorrect. $\exists u \in V$ is 1st vertex for which $d[u] \neq \delta(s,u)$ when it's added to C

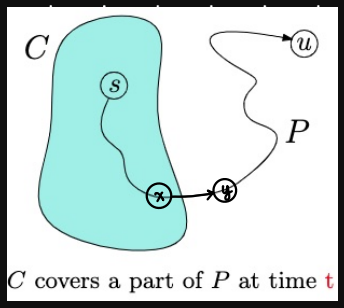
- time t: beginning of iteration in which u is added to C
- use shortest path P from s to u

→ proof:

$s \neq u$ since $d[s] = \delta(s,s) = 0$

$s \in C \Rightarrow C \neq \emptyset$

If there's no path from s to u, then $\delta(s,u) = \infty = d[u]$. There exists a path from s to u. Hence, there exists a shortest path, which we'll name P.



C covers a part of P at time t

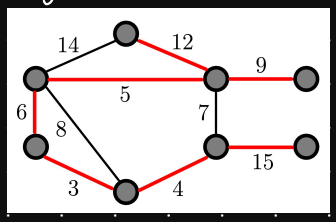
- on P, find y w/ predecessor x, which is 1st vertex in V-C.
- at time t, $d[y] = \delta(s,y)$
  - → proof:

    x is added to C before u so $d[x] = \delta(s,x)$. When x was added to C, its edges were relaxed, including (x,y). y is on shortest path P from s to u so part of P from s to y is shortest path. Hence, $\delta(s,y) = \delta(s,x) + w(x,y)$. For relaxing (x,y), $d[y]$ is compared to $\delta(s,x) + w(x,y)$, which is min val for $d[y]$. Thus, at time of relaxation, $d[y] = \delta(s,x) + w(x,y) = \delta(s,y)$.
- fact 1: $d[y] = \delta(s,y) \underbrace{\leq \delta(s,u) \leq d[u]}_{\text{b/c non-ve weights}}$
- fact 2: at time t, algo chose u so $d[u] \leq d[y]$
- from facts 1 + 2, we conclude $d[y] = \delta(s,y) = \delta(s,u) = d[u]$
  - → we arrive at contradiction

MINIMUM SPANNING TREES

SPANNING TREES

- if $G = (V, E)$ is connected graph, then **spanning tree** in G is tree of form (V, A) w/ A as subset of E
  - ↳ i.e. tree w/ edges from E that cover all vertices
  - ↳ e.g. BFS + DFS trees
- suppose edges have **weights** $w(e_i)$
- goal is to find spanning tree w/ **minimal weight**
  - ↳ e.g.



KRUSKAL'S ALGORITHM

Kruskal's algo is greedy

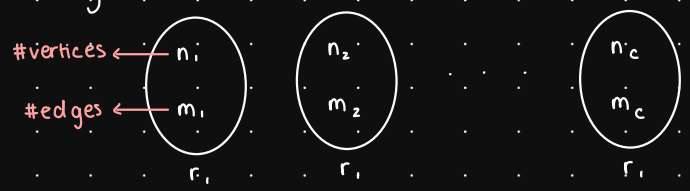
- ↳ pseudocode:

```
GreedyMST(G)
1.  A ← [ ]
2.  sort edges by increasing weight
3.  for k = 1, ..., m do
4.      if e_k does not create a cycle in A then
5.          append e_k to A
```

NOTE
A will have n-1 edges in the end

- **claim**: let G be connected graph + let A be subset of edges in G; if (V, A) has no cycle + $|A| < n-1$, then we can find an edge e not in A st $A \cup \{e\}$ still has no cycle
  - ↳ proof:

In general:



If r_i is connected, then $m_i \geq n_i - 1$.

$$\left(\underbrace{\sum_{i=1}^{c} n_i}_{n}\right) - \underbrace{c}_{\#\text{connected components}} = \sum_{i=1}^{c} (n_i - 1) \leq \sum_{i=1}^{c} m_i \leq m$$

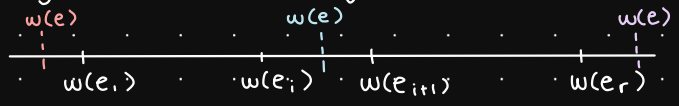
In any graph, # vertices - #connected components $\leq$ # edges.
For (V, A), $n - c \leq |A| < n-1$ so $c > 1$. This means (V, A) has at least 2 connected components which belong to connected graph G.

Let e to be any edge in G that connects 2 diff connected components in A.

claim: if output is $A = [e_1, \ldots, e_r]$, then (V, A) is spanning tree $\iff r = n - 1$.

↳ proof:

A is built st. (V, A) has no cycles (above claim). Suppose (V, A) is not a spanning tree. Then, $\exists$ edge e not in A st $(V, A \cup \{e\})$ still has no cycle.



Case 1) $w(e) < w(e_1)$

- impossible since e_1 is elmt w/ smallest weight

Case 2) $w(e_i) < w(e) < w(e_{i+1})$

- impossible b/c at the moment we inserted e_{i+1}, we decided not to include e
  - → only reason is b/c e created cycle w/ $e_1, \ldots, e_i$

Case 3) $w(e) > w(e_r)$

- impossible b/c we would've included it in A since there's no cycle in $A \cup \{e\}$
  - → how greedy algo works

Thus, we've proven by contradiction no such e exists $\therefore (V, A)$ is spanning tree.

claim: let (V, A) & (V, T) be 2 spanning trees & let e be edge in T but not A, then $\exists$ edge e' in A but not in T st $(V, T + e' - e)$ is still a spanning tree.

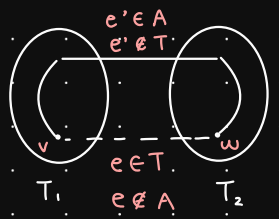
↳ e' is on cycle that e creates in A

↳ proof:

$e \in T$, $e \notin A$

Let $e = \{v, w\}$. $(V, A + e)$ contains a cycle $c = v, w, \ldots, v$. Removing e from T splits $(V, T - e)$ into 2 connected components T_1 & T_2. c starts in T_1 & crosses over to T_2, so it contains another edge e' btwn T_1 & T_2.

$e' \in A$, but $e' \notin T$. $(V, T' = T \cup \{e'\} \setminus \{e\})$ is spanning tree.



to get spanning tree w/ minimal weight

↳ let A be output of algo

↳ let (V, T) be any spanning tree

↳ if $T \neq A$, let e be edge in T but not in A

↳ there's edge e' in A but not in T st $(V, T + e' - e)$ is spanning tree & e' is on cycle that e creates in A

↳ during algo, we considered e but rejected it b/c it created cycle in A

↳ all other elmts in cycle have weight smaller/equal so $w(e') \leq w(e)$
↳ then, $T' = T + e' - e$ has $w(T') \leq w(T)$ + 1 more edge in common w/ A
↳ keep repeating until $T = A$

to **merge** connected sets of vertices:

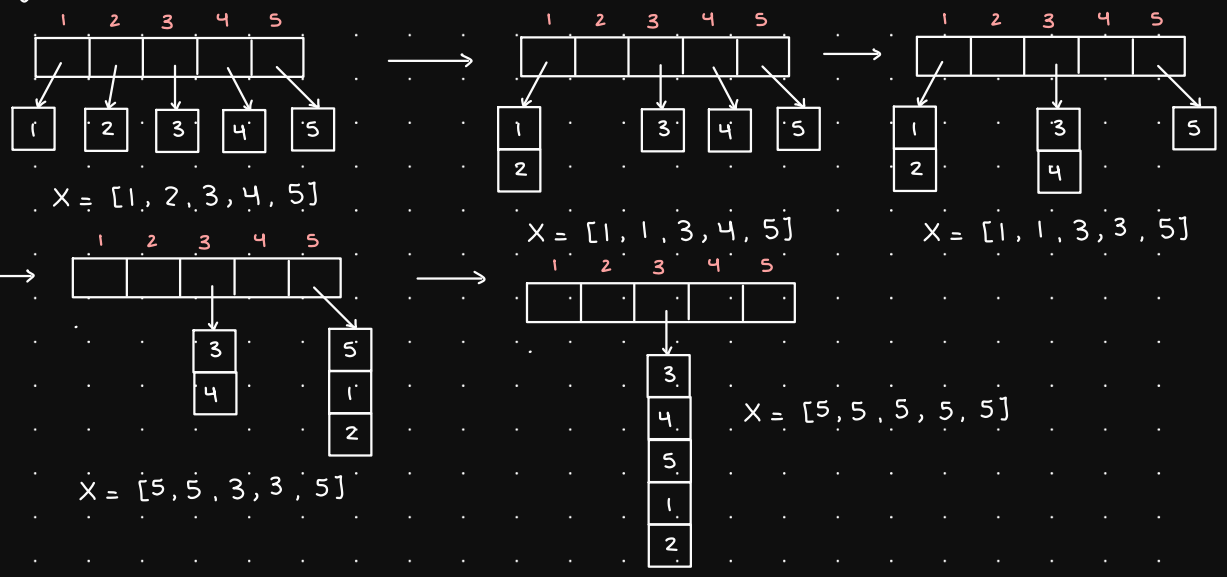
↳

GreedyMST_UnionFind(G)
1. $T \leftarrow [\,]$
2. $U \leftarrow \{\{v_1\}, \ldots, \{v_n\}\}$
3. sort edges by increasing weight
4. **for** $k = 1, \ldots, m$ **do**
5. **if** U.Find$(e_k.1) \neq U$.Find$(e_k.2)$ **then**
6. U.Union$(U$.Find$(e_k.1), U$.Find$(e_k.2))$
7. append e_k to T

- operations on **disjoint sets of vertices**:
  - → find: identify which set contains given vertex
  - → union: replace 2 sets by their union

↳ 1 data struct is to have **U be arr of linked lists**

- to do find, add arr of indices: X[i] = set that contains i
- e.g.



↳ worst case:
- find $\in O(1)$
- union traverses 1 of the linked lists, updates corresponding entries of X, + concatenates 2 linked lists
  - → $\in \Theta(n)$

Kruskal's algo runtime analysis:
↳ sorting edges $\in O(m \log m)$
↳ find $\in O(m)$
↳ union $\in O(n)$
↳ worst case is $O(m \log m + n^2)$

modify union so that each set in U keeps track of its size.

↳ only traverse smaller list
↳ add ptr to tail of lists to concatenate in $O(1)$
↳ worst case for 1 union is still $\Theta(n)$, but better total time
- $\forall$ vertices v, size of set containing v at least doubles when we update $x[v]$
- $x[v]$ updated at most $\log n$ times
- total cost of union per vertex is $O(\log n)$

↳ $O(n \log n)$ for all unions & $O(m \log m)$ total

DYNAMIC PROGRAMMING

e.g. Fibonacci #s

↳ defn:

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

↳ recursive algo:

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

- assuming additions are unit cost, runtime is:

$T(0) = T(1) = 0$

$T(n) = T(n-1) + T(n-2) + 1$

$T(n) = F(n+1) - 1$

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

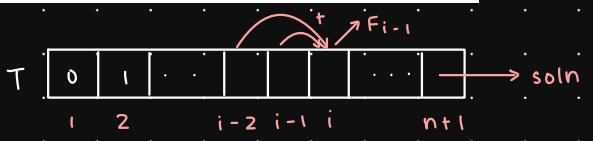
↳ improved recursive algo.

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

- prev. algo recomputes $F_0, \ldots, F_{n-1}$ for F_n many times
- now, we store previously computed vals in a global arr

↳ iterative:

```
Fib(n)
1.  let T = [0, 1, •, •, ...]
2.  for i = 2, ..., n
3.      T[i] = T[i − 1] + T[i − 2]
4.  return T[n]
```



↳ enhanced iterative:

```
Fib(n)
1.  (u, v) ← (0, 1)
2.  for i = 2, ..., n
3.      (u, v) ← (v, u + v)
4.  return v
```

- not always feasible

↳ all 3 improved versions use $O(n)$ additions

recipe for DP algo:

1) identify subproblem

↳ usually natural to retain solns in arr
↳ must know dimensions of arr
↳ specify precise meaning of any val of any cell of arr
↳ specify where ans is in arr

2) establish DP recurrence
↳ how val of cell in arr depend on other cells' vals

3) set vals for base cases

4) specify order of computation

5) recovery of soln
↳ keep track of subproblems that provided best solns
↳ use traceback strat to get full soln

key features of DP:
↳ solve problems thru recursion
↳ use small (polynomial) # nested subproblems
↳ may have to store results for all subproblems
↳ can often be turned into It loops

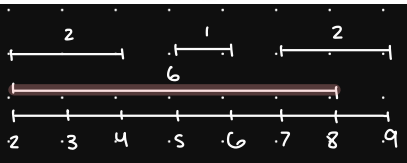
e.g. interval scheduling

Input:
- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- each interval has a weight $\boldsymbol{w_i}$

Output:
- a choice T of intervals that **do not overlap** and **maximizes** $\sum_{i \in T} w_i$
- greedy algorithm in the case $w_i = 1$

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 $T = [I_1]$, $W = 6$



↳ idea:

- In is interval w/ latest finish time

An optimal soln: Os (I1, ..., In) w/ weight Ow (I1, ..., In)

intervals that don't overlap w/ I_n

$$O_w(I_1,\dots,I_n) = \begin{cases} w_n + O_w(I_{m_1},\dots,I_{m_s}) & \text{if we choose } I_n \\ O_w(I_1,\dots,I_{n-1}) & \text{if we don't choose } I_n \end{cases}$$

- don't know what $I_{m_1},\dots,I_{m_s}$ look like
  - → find way to sort them st they're of form $I_1,\dots,I_s$ for some $s<n$
  - → then, optimize over all $I_1,\dots,I_j$ for $j=1,\dots,n$

↳ finding p_j:

Define p_j for interval I_j to be largest index $<j$ st interval I_{p_j} + I_j are disjoint.

Assume $I_1,\dots,I_n$ are sorted by inc finishing time so $f_i \le f_{i+1}$.

- claim: $\forall j$, set of intervals $I_k < I_j$ that don't overlap I_j is of form $I_1,\dots,I_{p_j}$ for some $0 \le p_j < j$
  - → $p_j = 0$ if no such interval
- algo will need p_i's for each interval
  - → $f_0 = \infty$
  - → if $\infty \le s_i < f_1$, $p_i = 0$
    - f_1 = earliest finishing time
  - → if $f_1 \le s_i < f_2$, $p_i = 1$

Finding p_i for interval $i = [s_i, f_i]$:

$-\infty = f_0 \le f_1 \le f_2 \le f_3 \le \dots \le f_n$

if s_i is here, $p_i = 0$ | if s_i is here, $p_i = 1$ | if s_i is here, $p_i = 2$

$f_* \le s_i \le f_{*+1} \Rightarrow p_i = *$

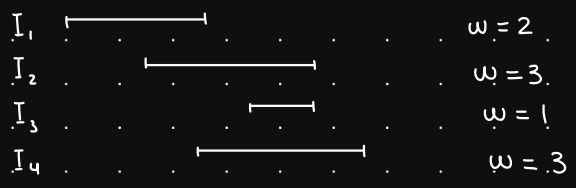
- let A be permutation of $[1,\dots,n]$ st $s_{A[1]} \le s_{A[2]} \le \dots \le s_{A[n]}$
  - → i.e. A is sorting permutation of start times
- pseudocode:

```
FindPj(A, s_1, ..., s_n, f_1, ..., f_n)
1.   f_0 ← −∞
2.   i ← 1
3.   for k = 0, ..., n
4.       while i ≤ n and f_k ≤ s_{A[i]} < f_{k+1}
5.           p_i ← k
6.           i++
```

  - → runtime is $O(n\log n)$ sorting + $O(n)$ loops
    - overall $O(n\log n)$

↳ main proc:

- $M[i]$ is max weight we can get w/ intervals $I_1,\dots,I_i$
- recurrence:
  - → $M[0] = 0$
  - → for $i \ge 1$, $M[i] = \max(M[i-1], M[p_i] + w_i)$
- runtime is $O(n\log n)$ for sorting twice + $O(n)$ for finding $M[i]$'s
  - → overall $O(n\log n)$
- e.g.



$f_0 = 0 \quad f_1 = 2 \quad f_2 = 4 \quad f_3 = 4 \quad f_4 = 5$

$0 = s_1 \leq s_2 \leq s_4 \leq s_3$

$A = [1, 2, 4, 3]$

Finding p_j's:

$i = 1, k = 0$ while: $-\infty = f_0 \leq s_{A[1]} = 0 < f_1 = 2$

$p_1 = 0, i = 2$

$-\infty = f_0 \leq s_{A[2]} = 1 < f_1 = 2$

$p_2 = 0, i = 3$

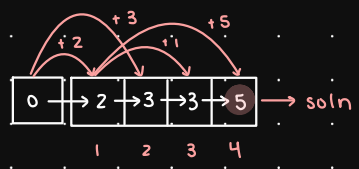
$i = 3, k = 1$ while: $2 = f_1 \leq s_{A[3]} = 2 < f_2 = 4$

$p_3 = 1, i = 4$

$2 = f_1 \leq s_{A[4]} = 3 < f_2 = 4$

$p_4 = 1, i = 5$

j	1	2	3	4
w	2	3	1	3
p	0	0	1	1



e.g. 0/1 knapsack

Input:

- items $1, \ldots, n$ with **weights** $w_1, \ldots, w_n$ and **values** $v_1, \ldots, v_n$
- a **capacity** W

Output:

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

Example:

- $w_1 = 3, w_2 = 4, w_3 = 6, w_4 = 5$
- $v_1 = 2, v_2 = 3, v_3 = 1, v_4 = 5$
- $W = 8$
- optimum $S = \{1, 4\}$ with weight 8 and value 7

See also:

- fractional knapsack (items can be divided), solved with a greedy algorithm

Either we choose item n or not.

Optimum $O[W, n]$ is max of 2 vals:

1) $v_n + O[W - w_n, n-1]$ if we choose n $(w_n \leq W)$
2) $O[W, n-1]$ if we don't choose n

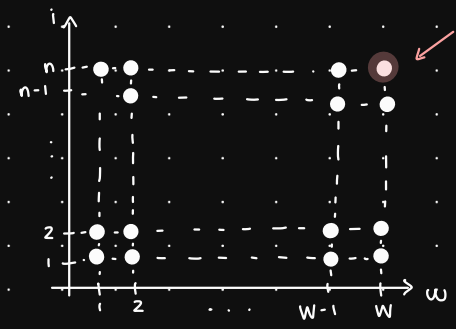
} either 1 of these 2 cases

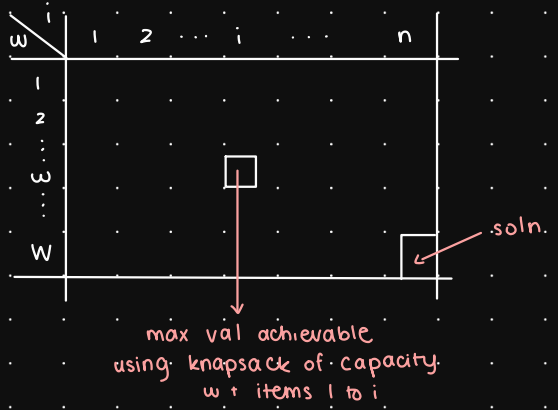
Let $O[w, i]$ = max val achievable using knapsack of capacity w + items $1, \ldots, i$

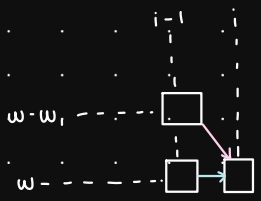
Init conds:

↳ $O[0, i] = 0$ for any i

↳ $O[w, 0] = 0$ for any w







1) Choose i so add $v_i + O[w - w_i, i-1]$
2) Don't choose i so just copy $O[w, i-1]$ over

↳ pseudocode:

01KnapSack$(v_1, \ldots, v_n, w_1, \ldots, w_n, W)$

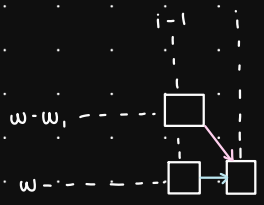
```
1.  initialize an array O[0..n, 0..W]
2.  with all O(0, j) = 0 and all O(w, 0) = 0
3.  for i = 1, ..., n
4.      for w = 1, ..., W
5.          if w_i > w        <-- check if we can acc put item i in knapsack w/its weight w_i
6.              O[w, i] ← O[w, i − 1]
7.          else
8.              O[w, i] ← max(v_i + O[w − w_i, i − 1], O[w, i − 1])
```

↳ example:

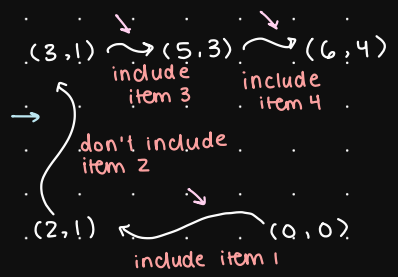
i	1	2	3	4
w	2	3	3	1
v	3	1	2	4

W = 6

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	4
2	0	3	3	3	4
3	0	3	3	3	7
4	0	3	3	3	7
5	0	3	4	5	7
6	0	3	4	5	9



- to recover soln, backtrack from soln cell:
  - → take item i if ↘
  - → don't take item i if →



↳ runtime is $\Theta(nW) = \Theta(n2^{\log W})$ (size of input is $\log W$ b/c it takes $\log W$ bits to store a word)

- exponential
- format runtime st we can always see size of input

↳ pseudo-polynomial algo

- in word RAM model, we assume all v_is + w_is fit in a word
- input size is $\Theta(n)$ words
- runtime also depends on vals of inputs
- 01-knapsack is NP-complete

e.g. longest inc subseq

Input: An array $A[1..n]$ of integers

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)

↳ order matters (i.e. subseq must match og order)

Let $L[i]$ be len of longest inc subseq of $A[1..i]$ that ends w/ $A[i]$ for $i = 1, \ldots, n$.

↳ $L[1] = 1$

↳ longest inc subseq S ending at $A[i]$ looks like

S = [..., A[j], A[i]] = S' cat [A[i]]

- S' is longest inc subseq ending at A[j]
  - → else, it's empty
- don't know j so we try all j<i for which A[j] < A[i]

↳ pseudocode:

LongestIncreasingSubsequence$(A[1..n])$

```
1.  L[1] ← 1
2.  for i = 2, ..., n do
3.      L[i] ← 1
4.      for j = 1, ..., i − 1 do
5.          if A[j] < A[i] then
6.              L[i] = max(L[i], L[j] + 1)
7.  return the maximum entry in L
```

↳ example:

i	1	2	3	4	5	6
A	7	1	3	10	11	5
L	1	1	2	3	4	3
j	0	0	2	3	4	3

i = 2 : ~~j = 1~~

i = 3 : ~~j = 1~~ | j = 2, 1 + 1

i = 4 : j = 1, 1 + 1 | j = 2, 1 + 1 | j = 3, 2 + 1

i = 5 : j = 1, 1 + 1 | j = 2, 1 + 1 | j = 3, 2 + 1 | j = 4, 3 + 1

i = 6 : ~~j = 1~~ | j = 2, 1 + 1 | j = 3, 2 + 1 | ~~j = 4~~ | ~~j = 5~~

↳ to recover soln:

- start at index i of max L[i]
- look at j for that i
- keep following j indices to trace back to base case
- for above example: 5 → 4 → 3 → 2 → 0

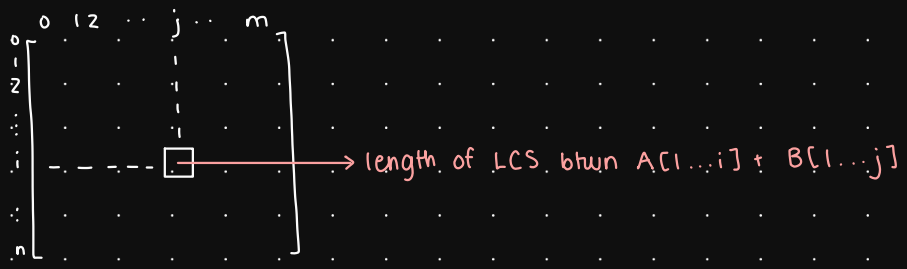
e.g. longest common subseq (LCS)

Input: Arrays $A[1..n]$ and $B[1..m]$ of characters

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

Example: A =**blurry**, B =**burger**, longest common subsequence is **burr**

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



In A[1...i] + B[1...j], 3 cases:

1) B[j] doesn't appear in LCS
   - common subseq is contained in A[1...i] + B[1...j-1]

2) A[i] doesn't appear in LCS
   - common subseq is contained in A[1...i-1] + B[1...j]

3) A[i] = B[j]
   - A[i] = B[j] must appear in LCS b/c they're same char (i.e. in common)
   - rest of seq is contained in A[1...i-1] + B[1...j-1]

↳ soln:

Let M[i,j] be longest subseq btwn A[1...i] + B[1...j].

Base cases:
- ∀j, M[0,j] = 0
- ∀i, M[i,0] = 0

Then M[i,j] is max of up to 3 vals:
- M[i,j-1]
  → don't use B[j]
- M[i-1,j]
  → don't use A[i]
- if A[i] = B[j], must use 1 + M[i-1,j-1]

↳ runtime is $\Theta(mn)$
- comparisons are O(1)
- arr is size mn + we compute all M[i,j] using 2 nested loops

↳ e.g.

		1	2	3	4	5	6
		A	b	l	u	r	r	y
	B	b	u	r	g	e	r

			b	u	r	g	e	r
	i \ j	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
b	1	0	1	1	1	1	1	1
l	2	0	1	1	1	1	1	1
u	3	0	1	2	2	2	2	2
r	4	0	1	2	3	3	3	3
r	5	0	1	2	3	3	3	4
y	6	0	1	2	3	3	3	4

↳ to recover soln
- walk back from M[n,m]
- each ↘ adds char to output

e.g. edit dist (ED)

Input: arrays $A[1..n]$ and $B[1..m]$ of characters

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

Example: A =**snowy**, B =**sunny**

s	n	o	w	y
s	u	n	n	y

3C

s	_	n	o	w	y
s	u	n	n	_	y

1A, 1C, 1D

_	s	n	o	w	_	y
s	u	n	_	_	n	y

2A, 1C, 2D

Examples: DNA sequences made of **a**, **c**, **g**, **t**

↳ when does misspelling happen?
1) char typed wrong → change
2) char is missed → add
3) extra char is typed → delete

We want to find ED btwn A[1...n] + B[1...m].

Types of alignment:

1) ... A[n]
   ... B[m]
   Last col adds 1 to dist if A[n] ≠ B[m] or adds 0 to dist if A[n] = B[m].
   ED of remaining cols is based on ED btwn A[1...n-1] + B[1... m-1].

2) ... A[n]
   ... — B is shorter than A
   ED is 1 + ED btwn A[1...n-1] + B[1... m]

3) ... — A is shorter than B
   ... B[m]
   ED is 1 + ED btwn A[n] + B[1...m-1]

↳ soln:

Let D[i,j] be ED btwn A[1...i] + B[1...j].

Base cases:
- ∀j, D[0,j] = j
  → add j chars
- ∀i, D[i,0] = i
  → delete i chars

Then, D[i,j] is min of 3 vals:

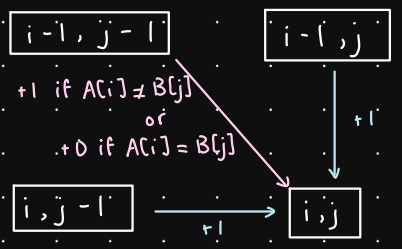
↳ D[i-1, j-1] if A[i] = B[j] or D[i-1, j-1] + 1 if A[i] ≠ B[j]

↳ D[i-1, j] + 1

- delete A[i] + match A[1..i-1] w/ B[1..j]

↳ D[i, j-1] + 1

- delete B[j] + match A[1..i] w/ B[1..j-1]



↳ runtime is $\Theta(m \cdot n)$

- D[i,j] computed using 2 nested loops

e.g. optimal bin search tree (BST)

Input:

- integers (or something else) $1, \ldots, n$
- probabilities of access $p_1, \ldots, p_n$, with $p_1 + \cdots + p_n = 1$

Output:

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

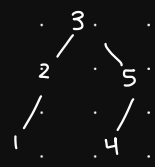
Example: $p_1 = p_2 = p_3 = p_4 = p_5 = 1/5$: ?

See also

- optimal static ordering for **linked lists**
- **Huffman trees**

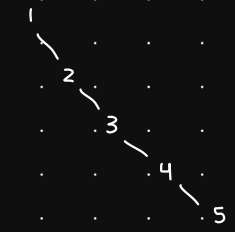
both built using greedy algorithms

↳ e.g.



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

$= \frac{11}{5}$



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

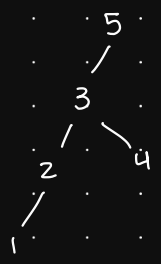
$= 3$

↳ greedy doesn't work b/c it puts key w/ highest probability in root

- e.g.

	1	2	3	4	5
p_i	0.1	0.2	0.25	0.05	0.4

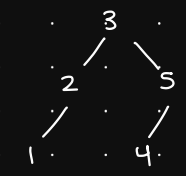
Greedy:



$0.4 + 2(0.25) + 3(0.2 + 0.05) + 4(0.1)$

$= 2.05$

Optimal:



$0.25 + 2(0.2 + 0.4) + 3(0.1 + 0.05)$

$= 1.9$

↳ soln:

Define $M[i,j]$ to be minimal cost for items i to j. Take item k b/wn $i+j$, then put it in the root

1) $M[i, k-1] + \sum_{\ell=i}^{k-1} (p_\ell)$ is min cost for i to $k-1$
   → all items shifted down 1 lvl so cost of access goes up by 1 for each node (which is why we need summation)
2) $M[k+1, j] + \sum_{\ell=k+1}^{j} (p_\ell)$ is min cost for $k+1$ to j
3) p_k for root

Recurrence is $M[i,j] = \min_{i \le k \le j} \left(M[i, k-1] + \sum_{\ell=i}^{k-1} p_\ell + p_k + M[k+1, j] + \sum_{\ell=k+1}^{j} p_\ell\right)$

$= \min_{i \le k \le j} \left(M[i, k-1] + M[k+1, j]\right) + \sum_{\ell=i}^{j} p_\ell$

- choose k so that it gives min val
- $M[i,i] = p_i$
- $M[i,j] = 0$ for $j < i$
- to compute $\sum_{\ell=i}^{j} p_\ell$, define $S[\ell] = \sum_{t=1}^{\ell} p_t$ so $\sum_{\ell=i}^{j} p_\ell = S[j] - S[i-1]$
  → $S[0] = 0$

↳ pseudocode:

OptimalBST$(p_1, \ldots, p_n, S_0, \ldots, S_n)$

```
1.  for i = 1, ..., n+1
2.      M[i, i-1] ← 0
3.  for d = 0, ..., n-1          d = j - i
4.      for i = 1, ..., n-d
5.          j ← d + i
6.          M[i, j] ← min_{i≤k≤j}(M[i, k-1] + M[k+1, j]) + S[j] - S[i-1]
```

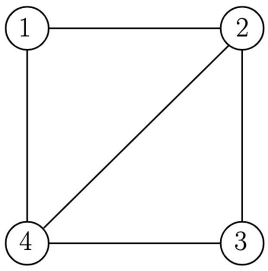
- runtime is $O(n^3)$

e.g. independent sets in trees

An independent set of a graph $G = (V, E)$, is $S \subseteq V$ if there are no edges between elements of S.

The maximum independent set problem (for a general graph):
input: $G(V, E)$
Output: An independent set of maximum cardinality.

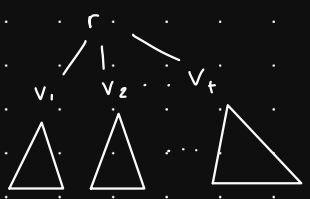
Example (not a tree):



$S = \{1, 3\}$.

↳ $u, v \in S \Rightarrow (u, v) \notin E$

Given a tree:



Either root appears in max indep set or it doesn't

1) $r \in S$: none of root's children can be in S so remaining part of S should come from subtrees rooted at grandchildren of r.
2) $r \notin S$: all elmts of S are from subtrees rooted at children of r

↳ soln:

I(u) is size of max indep set of subtree rooted at u.

$$I(u) = \max\left\{1 + \underbrace{\sum_{\text{grandchildren } w \text{ of } u} I(w)}_{u \in S}, \underbrace{\sum_{\text{children } w \text{ of } u} I(w)}_{u \notin S}\right\}$$

BELLMAN-FORD ALGORITHM

Bellman-Ford algo solves SSSP problem

↳ source is fixed

↳ if no -ve cycle, computes all dists $\delta(s, v)$

↳ can detect -ve cycles

↳ simple pseudocode, but slower than Dijkstra's algo

for $i = 0, \ldots, n-1$, set $\delta_i(s, v)$ = len of shortest path $s \rightsquigarrow v$ w/at most i edges

↳ if no such path, $\delta_i(s,v) = \infty$

↳ $\delta_0(s,s) = 0$

↳ $\delta_0(s,v) = \infty$ for $v \neq s$

↳ if there's no -ve cycles, $\delta_{n-1}(s,v) = \delta(s,v)$

- i.e. shortest paths are simple
- if $\delta(s,v) = \infty$, there's no path
- if $\delta(s,v)$ is finite #, then it's weight of shortest path P from s to v
  - → claim: P has at most n-1 edges
    We prove by contradiction. If claim is false, it's implied that P has a cycle. This cycle can't be -ve so it's a +ve cycle. We can remove cycle to get better path This is contradiction.

↳ $\forall i, v$, in any case, $\delta(s,v) \leq \delta_i(s,v)$

- if $\delta_i(s,v) = \infty$, then anything is $\leq \infty$
- if $\delta_i(s,v)$ is finite #, it means there exists a path from s to v
  - → $\delta(s,v)$ is that path or smth better so $\delta(s,v) \leq \delta_i(s,v)$

↳ a path from s to another vertex v has at most n-1 edges unless it contains a cycle (assuming no -ve cycles)

- if it has $> n-1$ edges, then 1 of the vertices appear more than once, meaning there exists a cycle

recurrence: $\delta_i(s,v) = \min(\delta_{i-1}(s,v), \min_{(u,v)\in E} \delta_{i-1}(s,u) + w(u,v))$

↳ idea:

Assume $\delta_i(s,v) < \infty$, which means $\delta_i(s,v)$ is len of path w/ at most i edges which has finite weight. The path either has exactly i edges or at most i-1 edges. If it has i-1 edges, then we use $\delta_{i-1}(s,v)$. Otherwise, decompose path to 1 edge (u,v) + path w/ i-1 edges to u. So, we use $\delta_{i-1}(s,u) + w(u,v)$.

However, to find shortest path, must consider all edges (u,v) as path may pass thru any neighbours

pseudocode (not efficient version):

BellmanFord(G, s)

```
1.  d_0 ← [0, ∞, ..., ∞] (s is the first index)
2.  parent ← [s, •, ..., •] (s is the first index)
3.  for i = 1, ..., n − 1 do
4.      for all v in V do
5.          d_i[v] ← d_{i−1}[v]
6.          for all (u, v) in E do
7.              if d_{i−1}[u] + w(u, v) < d_i[v] then
8.                  d_i[v] ← d_{i−1}[u] + w(u, v)
9.                  parent[v] ← u
```

↳ if no -ve cycle, $d_i[v] = \delta_i(s,v)$ so $d_{n-1}[v] = \delta(s,v)$

↳ there's d arr for every $i = 1, \ldots, n-1$

↳ use single d arr to save time + space:

BellmanFord2.0(G, s)

```
1.      d ← [0, ∞, ..., ∞]  (s is the first index)
2.      parent ← [s, •, ..., •]  (s is the first index)
3.      for i = 1, ..., n − 1 do
4.          for all (u, v) in E do
5.              if d[u] + w(u, v) < d[v] then
6.  relaxation      d[v] ← d[u] + w(u, v)
7.                  parent[v] ← u
```

- runtime: $O(mn)$

correctness for ver 2.0:

↳ claim: $\forall i = 0, \dots, n-1$, after i^{th} iteration, $\delta(s,v) \le d[v] \le \delta_i(s,v) = d_i[v]$ (1: $\delta(s,v) \le d[v]$; 2: $d[v] \le \delta_i(s,v)$)

If this is true, then after $(n-1)^{th}$ iteration, $\delta(s,v) \le d[v] \le \delta_{n-1}(s,v) = \delta(s,v)$

$\Rightarrow d[v] = \delta(s,v)$

2) $\forall i$, after iteration i, $\forall v$, $d[v] \le \delta_i(s,v) = d_i[v]$

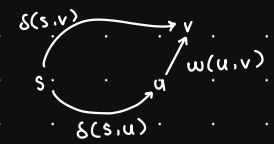
- proof by induction:
  - Base case: $i = 0$
  - True b/c $d_0[v] = \infty$
  - Inductive step: suppose true for index $i-1$, then prove for i
  - At beginning of loop, $\forall v$, $d[v] \le d_{i-1}[v]$ (inductive hypothesis). $d[v]$ can only dec so this stays true thruout loop. $d[v]$ is replaced by $d_i[v] = \min(d_{i-1}[v], \min_{(u,v)\in E}(d[u] + w(u,v)))$. We know $d[u] \le d_{i-1}[u]$ so after iteration i, $d[v] \le d_i[v]$.

1) $d[v]$ can only dec thru relaxation + if $\delta(s,u) \le d[u]$ and $\delta(s,v) \le d[v]$ before relaxation, then $\delta(s,v) \le d[v]$ post-relaxation

→ relaxation: $d[v] \leftarrow \min(d[v], d[u] + w(u,v))$

→ 1st item is true

→ 2nd item:



Due to triangle inequality, $\delta(s,v) \le \delta(s,u) + w(u,v)$ (where $\delta(s,u) = d[u]$) before relaxation

so after relaxation, $d[v] = \min(d[v], d[u] + w(u,v))$

Hence, $d[v] \ge \delta(s,v)$. (where $\delta(s,v) \le d[v]$ before relaxation)

→ if all $d[v]$ satisfy $\delta(s,v) \le d[v]$ + we apply any # relaxations, then all inequalities stay true

if there's no -ve cycle, $\forall v$, $d[v] = \delta[s,v]$ at end

↳ for any edge (u,v), $d[v] \le d[u] + w(u,v)$

- i.e. triangle inequality

claim: if there's -ve cycle, then there must exist edge (u,v)

w/ $d[v] \geq d[u] + w(u,v)$

↳ proof:

There exists -ve cycle $C = v_0, \ldots, v_k$ where $v_0 = v_k$ + $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

Assume that $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ $\forall i$. Take sum of inequalities around cycle.

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

since $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$ b/c $v_0 = v_k$

Thus, we get a contradiction b/c -ve cycle should have overall -ve weight.

↳ for extra $O(m)$, check for presence of -ve cycle

FLOYD-WARSHALL ALGORITHM

- computes all shortest paths $\delta(u,v)$ from any vertex u to any vertex v
  - ↳ no fixed source
- -ve weights are OK but no -ve cycles
- simple pseudocode, but slower than other algos
  - ↳ Bellman-Ford from all u is $O(mn^2)$
- comparisons of subproblems for DP:
  - ↳ Bellman-Ford uses paths w/ fixed # steps

  - ↳ Floyd-Warshall restricts which vertices can be used

- $\forall i = 0, \ldots, n$, set $D_i(v_j, v_k)$ = len of shortest path $v_j \rightsquigarrow v_k$ w/ all intermediate vertices in $v_1, \ldots, v_i$
  - ↳ for $i = 0$
    - $D_0(v_j, v_j) = 0$
    - $D_0(v_j, v_k) = \begin{cases} w(v_j, v_k) & \text{if there's edge } (v_j, v_k) \\ \infty & \text{otherwise} \end{cases}$
  - ↳ $D_n(v_j, v_k) = \delta(v_j, v_k)$

recurrence rltn: consider shortest path btwn v_j to v_k w/ all intermediate vertices in $\{v_1, \ldots, v_i\}$ + we call it P

1) v_i is not on P. All intermediate vertices on P are in $\{v_1, \ldots, v_{i-1}\}$ so we use $D_{i-1}[v_j, v_k]$

2) v_i is on P. Then, P looks like (v_i is only in P once b/c there's no cycles):

P_1 is shortest path from v_j to v_i. P_2 is shortest path from v_i to v_k. In both P_1 & P_2, all intermediate vertices are in $\{v_1, \ldots, v_{i-1}\}$. Thus, we also use D_{i-1}.

↳ $D_i(v_j, v_k) = \min(D_{i-1}(v_j, v_k), D_{i-1}(v_j, v_i) + D_{i-1}(v_i, v_k))$

$D_{i-1}(v_j, v_k)$: v_i not in P

$D_{i-1}(v_j, v_i) + D_{i-1}(v_i, v_k)$: v_i in P

pseudocode

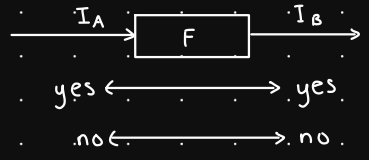
FloydWarshall(G)

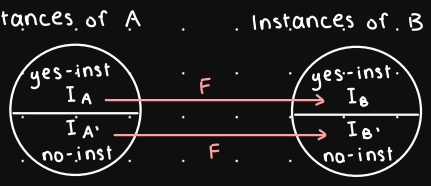
```
1.  set up D_0 above
2.  for i = 1, ..., n do
3.      for j = 1, ..., n do
4.          for k = 1, ..., n do
5.              D_i[v_j, v_k] ← min(D_{i-1}[v_j, v_k], D_{i-1}[v_j, v_i] + D_{i-1}[v_i, v_k])
```

↳ runtime & mem is $\Theta(n^3)$

POLYNOMIAL TIME REDUCTION

- only talk abt **decision problems** (i.e. bool ans of yes(Y) or no(N))
- **problem instance**: input for specified problem
- **problem soln**: correct ans (Y or N) for specified problem instance
  - ↳ I is yes-instance if correct ans is Y for instance
  - ↳ I is no-instance if correct ans is N for instance
- size of problem instance (**Size(I)**): #bits required to specify instance I
- **polynomial time reduction**: decision problem A is polynomial time reducible to B if there's a polynomial time algo F that transforms any instance I_A of A to instance I_B of B st **I_A is yes-instance of A $\iff$ $F(I_A) = I_B$ is yes-instance of B**
  - ↳ $A \leq_P B$ if such polynomial time reduction exists
  - ↳ $A =_P B$ if $A \leq_P B$ + $B \leq_P A$
  - ↳ i.e. goal is to find poly-time algo F that converts inputs of A to inputs of B





e.g.

Activity

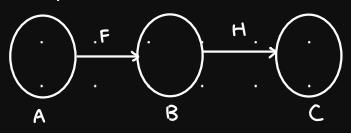
Assume A and B are decision problems and we are given:

- an algorithm Alg_B, which solves B in polynomial time,
- a polynomial reduction F, which gives $A \leq_P B$.

Design a polynomial time algorithm to solve A.

Input: instance I_A of A
Output: whether I_A is yes-instance
1) Use F to transform I_A into $I_B = F(I_A)$ which is instance of I_B
2) Return $Alg_B(I_B)$
Assume $Size(I_A) = n$. If runtime of F is $p(n)$, where p is polynomial, then output $F(I_A)$ has property $Size(F(I_A)) \leq p(n)$ since F must return $F(I_A)$.
Total runtime: if Alg_B has runtime $q(n)$, which is polynomial, then total runtime is in $O(q(p(n)))$

- **transitivity of poly-time reductions** lemma: $A \leq_P B$ and $B \leq_P C \Rightarrow A \leq_P C$
  - ↳ proof:
    - $A \leq_P B$ means there exists poly-time algo F which maps instances of A to B
    - $B \leq_P C$ means there exists poly-time algo H which maps instances of B to C



H ∘ F maps instances of A to C in poly-time. It also respects property of yes/no instances.

e.g.

Activity

Assume problem A is **known** to be impossible to be solved in polynomial time.

True/False:
$A \leq_P B \Rightarrow$ B Cannot be solved in polynomial time.

T

Assume that B can be solved in poly-time w/algo Alg_B. Use Alg_B for solving A by reducing it to B. However, it's impossible to solve A. Thus, we have a contradiction.

↳ looking at notation $A \leq_p B$:

- B is at least as hard as A
- if we could solve B, we could solve A

SIMPLE REDUCTIONS

following 3 problems are equiv in terms of poly-time solvability

↳ max clique:

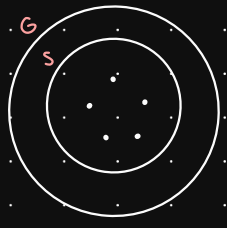
Maximum Clique (Clique)

$S \subseteq V$ is a clique if, $\{u, v\} \in E$ for all $u, v \in S$.
Input: $G = (V, E)$ and an integer k
Output: (the answer to the question) is there a clique in G with at least k vertices?

↳ max indep set (IS):

Maximum Independent Set (IS)

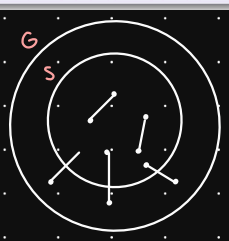
$S \subseteq V$ is an independent set if, $\{u, v\} \notin E$ for all $u, v \in S$.
Input: $G = (V, E)$ and an integer k
Output: (the answer to the question) is there an independent set in G with at least k vertices?



↳ min vertex cover (VC):

Minimum Vertex Cover (VC)

$S \subseteq V$ is a vertex cover if, $\{u, v\} \cap S \neq \emptyset$ for all $\{u, v\} \in E$.
Input: $G = (V, E)$ and an integer k
Output: (the answer to the question) is there a vertex cover in G with at most k vertices?



prove clique $\leq_p$ IS + IS $\leq_p$ clique:
↳ clique finds set of vertices w/ all edges in btwn
↳ IS finds set of vertices w/ no edges in btwn
↳ idea: find complement of graph
$\overline{G} = (V, \overline{E})$, $\{u, v\} \in \overline{E} \iff \{u, v\} \notin E$
F is algo to construct complement of input G. It can be done in poly-time + we can check $S \subseteq V$ is a clique in G iff S is indep set in G. Hence, $\langle G, k \rangle$ is yes-instance for clique iff $\langle \overline{G}, k \rangle$ is yes-instance for IS.

lemma: assume $G = (V, E)$ is graph, $S \subseteq V$ is vertex cover $\iff V - S$ is indep set in G
↳ proof:
$\Rightarrow$:
Assume S is vertex cover in G + we claim $V - S$ is IS. If it's not true, then we have $x, y \in V - S$ st $\{x, y\} \in E$. By defn of VC, at least 1 of x or y must be in S. This is contradiction.
$\Leftarrow$:
Assume that $V - S$ is IS + we claim S is VC. If it's not true, there's an edge $\{x, y\} \in E$ st $x \notin S$ and $y \notin S$. This means that $x, y \in V - S$ + there's edge btwn them. Since $V - S$ is IS, this is contradiction.
↳ from this lemma, reduction algo maps $\langle G, k \rangle$ for VC to $\langle G, n-k \rangle$ for IS
- it runs in poly-time + maps yes/no instances for VC to respective yes/no instances for IS
- above results + transitivity shows clique $=_p$ IS $=_p$ VC

Hamiltonian Cycle (HC) + Hamiltonian Path (HP) simple reduction:

Hamiltonian Cycle (HC)

A cycle is a Hamiltonian Cycle if it touches every vertex exactly once.
Input: Undirected graph $G = (V, E)$
Output: (the answer to the question) Does G have a Hamiltonian Cycle?

Hamiltonian Path (HP)

A path is a Hamiltonian Path if it touches every vertex exactly once.

Input: Undirected graph $G = (V, E)$

Output: (the answer to the question) Does G have a Hamiltonian path?

↳ HP $=_p$ HC

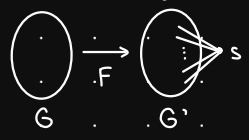
- proof:

HP $\leq_p$ HC:

Given $G = (V, E)$ for HP, we have to transform it to $G' = (V', E')$ for HC. We construct G' by:

→ add vertex s to V: $V' = V \cup \{s\}$

→ add edges (s, x) for $x \in V$ (i.e. s has edge to every vertex in G)



Then, F runs in poly-time b/c we're adding n new edges.

→ claim: G has HP iff G' has HC

- proof:

⇒:

Assume P is HP in G w/ endpoints u + w. Then, P + su + sw is HC in G'.

⇐:

Assume C' is HC in G'. There must exist 2 incident edges on s, name them su + sw. Then, by removing su + sw from C', we get C which is HP in G.

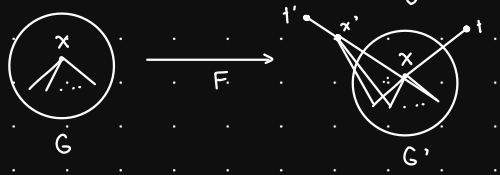
HC $\leq_p$ HP.

Assume $G = (V, E)$ is given for HC. We construct $G' = (V', E')$ by:

→ choose arbitrary vertex $x \in G$

→ add duplicate x' of x in G'

→ add vertices t' + t (w/ deg 1) w/ edges $t'x'$ + tx



F is poly-time transformation.

→ claim: G has HP iff G' has HC

- proof:

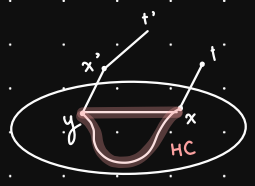
⇒:

Assume there's HC in G

P + x'u + x't' + xw + xt is HP in G'.

⇐:

Assume there's HP in G'. It should have form:



The 2 endpoints must be t' + t b/c they're deg 1 vertices. x' has 2 neighbours, t' + a vertex y. We also know that xy is edge in G as x' is a copy of x (and x' has edge to y). By removing t'u', x'y, and tx from HP + adding xy, we form HC in G.

$X = \{x_1, \ldots, x_n\}$ where x_i s are Bool vars

↳ literal term is either x_i or $\bar{x}_i$.

↳ clause: disjunction of distinct literals, $t_1 \vee t_2 \vee \ldots \vee t_\ell$

- $t_i \in \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$
- clause has len ℓ

↳ asgnt satisfies clause C if it causes C to eval to true

↳ conjunction of finite set of clauses, $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ is called formula in Conjunctive Normal Form (CNF)

3-SAT:

3-SAT

Input: A CNF-fomula in which each clause has at most **three** literals.

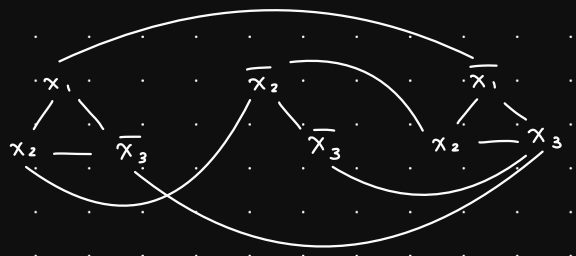
Output: (the answer to the question) is there a truth assignment to the variables that satisfies all the clauses?

↳ e.g.

$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$

C_1 C_2 C_3

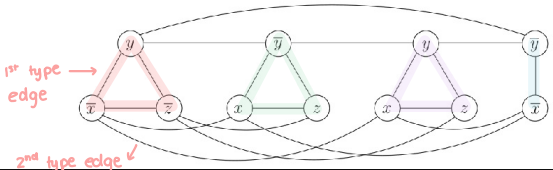
$x_1 = T$, $x_2 = F$, $x_3 = T$



NOTE
Connect all x_i + $\bar{x}_i$

↳ e.g.

Figure 8.8 The graph corresponding to $(x \vee y \vee z)\ (x \vee \bar{y} \vee z)\ (x \vee y \vee \bar{z})\ (\bar{x} \vee \bar{y})$.



↳ thm: $3\text{-SAT} \leq_p IS$

- proof:

A CNF w/clauses at most 3 is given. For each clause, we form triangle w/vertices labelled as literals of clause.

$(x \vee \bar{y} \vee \bar{z}) \longrightarrow$ 

A clause w/2 literals will be edge from 1 literal to another. A clause w/1 literal is a vertex. To force exactly 1 choice from each clause, set k to be # clauses. Make sure we're not choosing opp literals x & $\bar{x}$ in diff clauses, so we create edge btwn any 2 vertices that correspond to opp literals. This construction takes poly-time.

→ claim: suppose CNF formula has k clauses, then formula is satisfiable iff there's IS of size k in graph

- proof:

$\Rightarrow$:

If there's satisfying asgnt, then choose 1 literal that's set to true in each clause & corresponding vertex will be in IS. Since CNF is satisfiable, there's at least 1 true literal in each clause. So, the set has exactly k vertices. The k vertices form IS as there's no edges of 1st type btwn them since we choose only 1 literal vertex in each clause. There are no 2nd type edges b/c we won't choose x_i & $\bar{x}_i$ in satisfying asgnt.

$\Leftarrow$:

Assume there's IS of size k in graph G. Any IS can choose only 1 vertex from each clause since there are edges btwn them. We have only k clauses so IS of size k chooses exactly 1 vertex from each clause as there's edges btwn them. For each var, we choose at most 1 of x_i or $\bar{x}_i$ since there's 2nd type edge btwn them. If x_i is chosen in IS, set x_i to true & otherwise, set it to false. This asgnt satisfies CNF.

NP-COMPLETENESS

- for problem X, rep instance of X as bin str S
- problem X is in NP if there's poly-time verification algo ALG_X st input S is yes-instance iff there's a proof / certificate t which is bin str of len poly(|S|) st $ALG_X(S,t)$ returns yes
  - ↳ e.g.

Example 1: Vertex Cover

- S here is an input graph $G = (V, E)$ and an integer k
- t here is a subset U of V with $|U| \leq k$

$Alg_v(S,t)$: go through all E and check if t covers the edges and $t \leq k$.

Example 1: 3-SAT

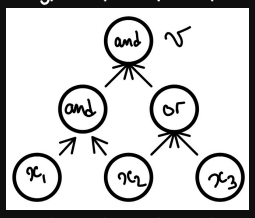
- S here is a 3-SAT formula
- t here is a truth assignment

$Alg_v(S,t)$: check whether t satisfies all clauses.

Exercise: Clique, IS, HC, HP, Subset-Sum are all in NP.

- not all problems are in NP
  - ↳ e.g. is given graph non-Hamiltonian (i.e. no Hamiltonian cycle)?
- co-NP: set of decision problems whose no-instances can be certified in poly-time
  every poly-time solvable decision problem is in NP
  - ↳ P is set of poly-time solvable decision problems
    - $P \subseteq NP$
- NP comes from Non-deterministic Polynomial time
  - ↳ non-deterministic machine has power to correctly guess soln
- problem $X \in NP$ is NP-complete if $Y \leq_p X$ for all $Y \in NP$
  - ↳ informally, problem is NP-complete if it's hardest problem in NP
  - ↳ fact: $P = NP \iff$ NP-complete problem can be solved in poly-time
- thm (Cook-Levin): 3-SAT is NP-complete
  - ↳ if we can prove 3-SAT $\leq_p X$, then X is NP-complete
    - e.g. IS ∈ NPC since 3-SAT $\leq_p$ IS
  - ↳ to prove a problem $X \in NP$ is NP-complete, find NP-complete problem Y & prove $Y \leq_p X$
- circuit-SAT
  - ↳ instance: a circuit = DAG w/ labels on vertices
  - ↳ inputs labelled by bool vars $x_1, \ldots, x_n$
  - ↳ internal vertices labelled by and, or, not
  - ↳ marked vertex v for output
  - ↳ problem: is there choice of bool x_i that makes v true?

↳ e.g.



to prove Cook-Levin thm:
- ↳ show circuit-SAT is NPC
- ↳ show circuit-SAT $\leq_p$ 3-SAT

to show Circuit-SAT is NPC:
- ↳ idea:
  - given: instance S of $A \in NP$
  - want: proof t st $ALG_A(S,t) = true$
  - $ALG_A(S,t)$ can be turned into circuit w/ t as input
    - → verification algo $ALG_A(S,t)$ is Circuit-SAT
  - call Circuit-SAT to find t st $ALG_A(S,t) = true$
- ↳ e.g.

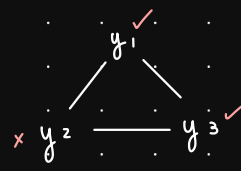
- problem A: IS
- instance S: complete graph with 3 vertices, $k = 2$
- certificate t: 3 bits y_1, y_2, y_3 (yes/no for each vertex)
- circuit for $ALG_A(S,t)$ computes the "formula"

$$(y_1 + y_2 + y_3 \geq 2) \wedge \overline{y_1 \wedge y_2} \wedge \overline{y_1 \wedge y_3} \wedge \overline{y_2 \wedge y_3}$$

at least 2 items in set — vertices in set aren't connected

y_1 y_2 y_3

1 0 1



$\{y_1, y_2, y_3\}$

✓ ✗ ✓

1 0 1

- no-instance

to show Circuit-SAT $\leq_p$ 3-SAT

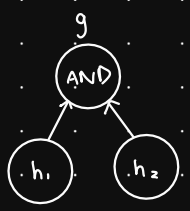
From logic, we know $(p \Leftrightarrow q) \equiv ((p \vee \bar{q}) \wedge (\bar{p} \vee q))$

Gate g

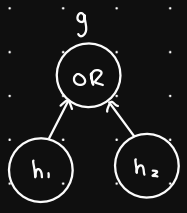
true ⟶ create (g) for this

g

false ⟶ create $(\bar{g})$ for this



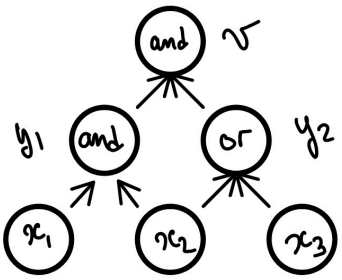
$$g \Leftrightarrow (h_1 \wedge h_2)$$
$$\equiv (\bar{g} \vee (h_1 \wedge h_2)) \wedge (g \vee \overline{(h_1 \wedge h_2)})$$
$$\equiv (\bar{g} \vee h_1) \wedge (\bar{g} \vee h_2) \wedge (g \vee \bar{h}_1 \vee \bar{h}_2)$$



$$g \iff (h_1 \vee h_2)$$
$$\equiv (\bar{g} \vee (h_1 \vee h_2)) \wedge (g \vee \overline{(h_1 \vee h_2)})$$
$$\equiv (g \vee \bar{h}_1) \wedge (g \vee \bar{h}_2) \wedge (\bar{g} \vee h_1 \vee h_2)$$

$$g \iff \bar{h}$$
$$\equiv (g \vee h) \wedge (\bar{g} \vee \bar{h})$$

↳ e.g.



gives

$$\Big(y_1 \iff (x_1 \wedge x_2)\Big) \wedge \Big(y_2 \iff (x_2 \vee x_3)\Big) \wedge \Big(v \iff (y_1 \wedge y_2)\Big) \wedge v$$

and

$$\begin{aligned} F = &(y_1 \vee \overline{x_1} \vee \overline{x_2}) \wedge (\overline{y_1} \vee x_1) \wedge (\overline{y_1} \vee x_2) \wedge \\ &(y_2 \vee \overline{x_2}) \wedge (y_2 \vee \overline{x_3}) \wedge (\overline{y_2} \vee x_2 \vee x_3) \wedge \\ &(v \vee \overline{y_1} \vee \overline{y_2}) \wedge (\overline{v} \vee y_1) \wedge (\overline{v} \vee y_2) \wedge v \end{aligned}$$

NPC problems:

↳ 3SAT, SAT

↳ IS, VC, + clique

↳ directed HC + HP

↳ traveling salesman

↳ subset sum

- input: +ve ints $a_1, \dots, a_n$ + K
- output: $S \subseteq \{1, \dots, n\}$ st $\sum_{i \in S} a_i = K$

↳ 0/1 knapsack

- input: $((v_1, w_1), \dots, (v_n, w_m))$, W, V
- output: $S \subseteq \{1, \dots, n\}$ st $\sum_{i \in S} w_i \leq W$ + $\sum_{i \in S} v_i \geq V$

DirectedHamiltonianCycle (DHC), HC, + HP are NPC

DHC

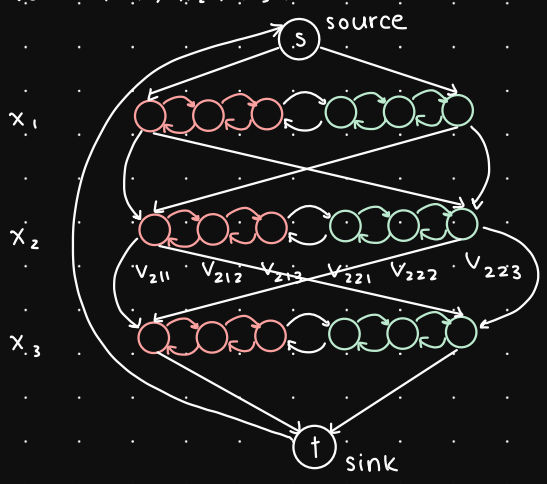
↳ input: directed graph G

↳ output: does G have directed cycle that visits each vertex once

thm: $3SAT \leq_p DHC \leq_p HC$

↳ start w/ $3SAT \leq_p DHC$ so we're given formula in 3CNF

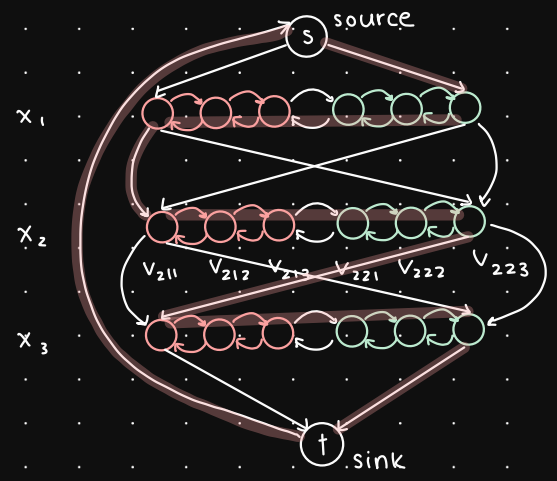
- clauses: $C_1 \wedge C_2$
- vars: $\{x_1, x_2, x_3\}$



v_{ijk} → x_i, C_j, $\{1,2,3\}$

NOTE
There's 2^n HCs (n is #vars) b/c there's 2 options (left or right) to traverse.

- source s + sink t
- 1 row of vertices per var x_i
- on row i, 3 vertices v_{ij1}, v_{ij2}, v_{ij3} per clause C_j
- convention:
  - → T = left to right
  - → F = right to left
- e.g. $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2)$

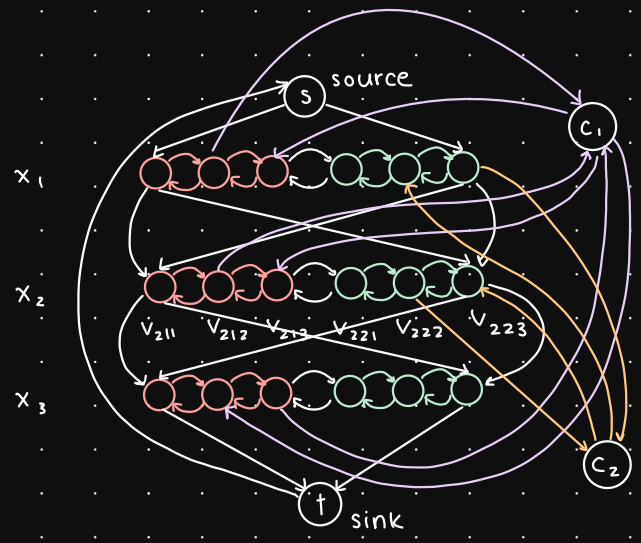


$x_1 = F$

$x_2 = T$

$x_3 = T$

- for any clause C_j:
  - → add new vertex c_j
  - → for any literal x_i in C_j, add edges (v_{ij2}, c_j) & (c_j, v_{ij3})
  - → for any literal $\bar{x}_i$ in C_j, add edges (v_{ij3}, c_j) & (c_j, v_{ij2})



$C_1 = (x_1 \vee x_2 \vee \overline{x_3})$

$C_2 = (\overline{x_1} \vee x_2)$

NOTE
We use a "gap node" in btwn b/c if not, we could take a path to c_i, then go to another row, then go back c_i, then og row (which is unwanted).

- **claim**: if formula is satisfiable, there's DHC in G
  - → var asgnt to either T/F, so we know dir to go in each row
  - → choose only 1 literal that's T per clause C_j (so that we visit c_j once)
  - → detour to visit c_j when we go thru corresponding row
- **claim**: if DHC in G, formula is satisfiable
  - → if cycle goes from v_{ij2} to c_j, it must come back to v_{ij3} (else, we can't put v_{ij3} in cycle) since each row only moves in 1 dir
  - → each row is visited LtoR or RtoL
  - → dir gives asgnt for $x_1, \ldots, x_n$
  - → by design, it satisfies all clauses

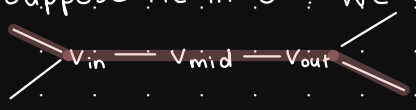
↳ **DHC $\leq_p$ HC**

- reduction:
  - → given: directed graph G
  - → build: undirected graph G'
  - → ensure: DHC in G $\Leftrightarrow$ HC in G'
- gadget:
  - → replace each vertex v by v_{in}, v_{mid}, v_{out}
  - → make all edges undirected

- **claim**: DHC in G $\Leftrightarrow$ HC in G'

$\Rightarrow$:
If DHC in G, then HC in G' by following cycle

$\Leftarrow$:
Suppose HC in G'. We can only have



but not

Thus, this gives DHC in G.

e.g. 3D matchings

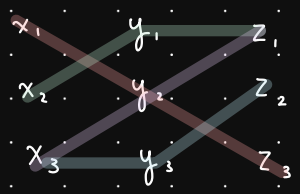
3D matching

- **input:** 3 sets X, Y, Z of size n and a family of **hyperedges** $E \subset X \times Y \times Z$
- **output:** is there a **perfect matching** (n hyperedges that cover X, Y and Z)?

  (each x_i (and each y_j, and each z_k) is in a unique hyperedge)
- **NP**

Remark: 2D version

- **input:** 2 sets X, Y of size n and a family of **edges** $E \subset X \times Y$
- **output:** is there a **perfect matching** (n edges that cover X, Y)?
- this is testing if a bipartite graph has a perfect matching



$E = \{(x_1, y_2, z_3), (x_2, y_1, z_1), (x_3, y_2, z_1), (x_3, y_3, z_2)\}$

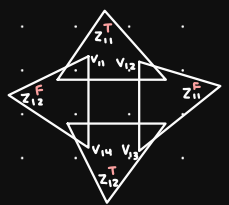
output $= \{(x_1, y_2, z_3), (x_2, y_1, z_1), (x_3, y_3, z_2)\}$

↳ claim: 3SAT $\leq_p$ 3DMatching

- given: formula F in 3CNF w/ s clauses $C_1, \ldots, C_s$
- want: build instance H of 3DMatching st F is satisfiable iff H admits perfect 3D matching
- reduction must be poly-time

↳ var gadget: build fidget spinner per var x_i, $i = 1, \ldots, n$

- $2s$ core vertices $v_{i1}, \ldots, v_{i2s}$
  → only used in gadget
- $2s$ tip vertices $z_{i1}^T, z_{i1}^F, \ldots, z_{is}^T, z_{is}^F$
  → will connect to clause
- e.g. $s = 2$, $i = 1$

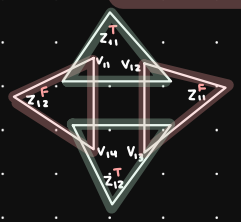


$C_1 = x_1 \vee x_2 \vee \overline{x_3}$

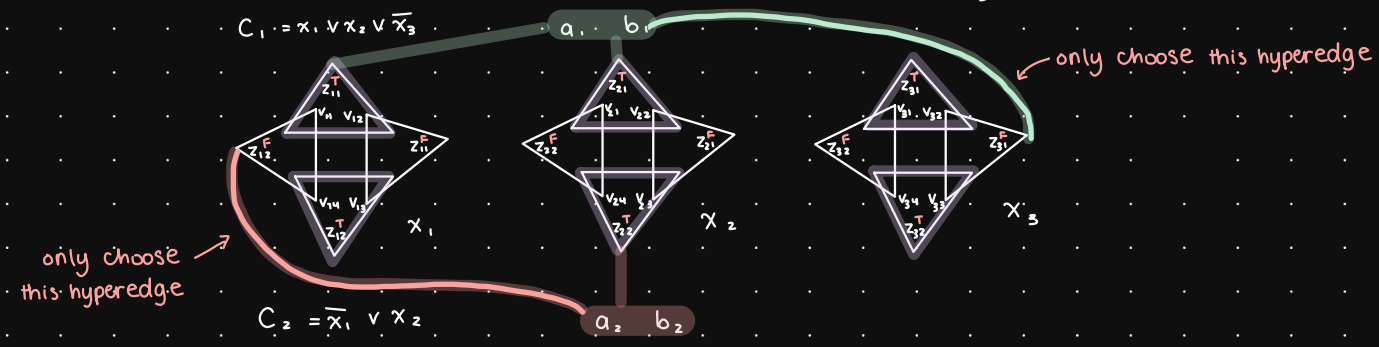
$C_2 = \overline{x_1} \vee x_2$

$C_1 \wedge C_2$

- hyperedges for $j = 1, \ldots, s$:
  - → z^T_{ij}, $v_{i,2j-1}$, $v_{i,2j}$
  - → z^F_{ij}, $v_{i,2j}$, $v_{i,2j+1}$
- to cover **core vertices**, choose to either assign var to T / F



  - → 2^n coverings (2 possibilities per var)
- for any **clause C_j**, choose 1 hyperedge to make C_j true
  - → add 2 new vertices a_j & b_j
  - → for any literal x_i in C_j, add hyperedge $\{a_j, b_j, z^T_{ij}\}$
  - → for any literal $\bar{x}_i$ in C_j, add hyperedge $\{a_j, b_j, z^F_{ij}\}$



- final adjustments:
  - → there's 2ns tips
  - → in perfect matching, each clause covers a tip so 2ns - s tips left
  - → add **2ns - s dummy pairs d_k, e_k**, + all hyperedges $\{z^T_{ij}, d_k, e_k\}$ + $\{z^F_{ij}, d_k, e_k\}$ to covering
    - $(2ns - s)(2ns)$ more hyperedges

↳ **claim**: F is satisfiable iff H admits perfect 3D matching.

- proof:

⇒:

Assume F is satisfiable. Cover gadgets for $x_1, \ldots, x_n$ according to truth vals. Pick exactly 1 true literal per clause C_j. If x_i, take hyperedge $\{a_j, b_j, z^T_{ij}\}$. If $\bar{x}_i$, take hyperedge $\{a_j, b_j, z^F_{ij}\}$. Match remaining tips w/ pairs of dummy vertices.

⇐:

Assume H has perfect 3D matching. Matching gives truth vals for each var. For each clause C_j, we picked a hyperedge $\{a_j, b_j, z^T_{ij}\}$ that corresponds to x_i or $\{a_j, b_j, z^F_{ij}\}$ that corresponds to $\bar{x}_i$. As such, all clauses are satisfied either way.

e.g. subset sum

Subset sum

- **given:** positive integers $a_1, \ldots, a_n$ and K
- **want:** is there a subset S of $\{1, \ldots, n\}$ with $\sum_{i \in S} a_i = K$
- **NP**

Claim

3DMatching $\leq_P$ SubsetSum

- **given:** sets X, Y, Z of size n, m hyperedges $E \subset X \times Y \times Z$
- **want:** integers $a_1, \ldots, a_s, K$ s.t. perfect 3D matching iff $\sum_{i \in S} a_i = K$ for some $S \in \{1, \ldots, n\}$
- reduction must be polynomial time

↳ claim: 3DMatching $\leq_p$ SubsetSum

(x_u, y_v, z_w)

↓ ↓ ↓

X Y Z

- define m 0/1 vectors (1 per hyperedge) of size 3n.
  - → j^{th} hyperedge = $\{x_u, y_v, z_w\}$, u, v, w in $\{1, \ldots, n\}$
  - → j^{th} vector given by:

$$\begin{bmatrix} 0 \cdots 1 \cdots 0 & 0 \cdots 1 \cdots 0 & 0 \cdots 1 \cdots 1 \\ x_1 \cdots x_u \cdots x_n & y_1 \cdots y_v \cdots y_n & z_1 \cdots z_w \cdots z_n \end{bmatrix}$$

1 u n n+1 n+v 2n 2n+1 2n+w 3n

- there's perfect 3D matching iff there's subset of $\{v_1, \ldots, v_m\}$ that adds up to $[111 \ldots 111]$
- define m ints (1 per hyperedge) of bit-size polynomial in n, m
  - → set base $b = m+1$
  - → given

$v_j = [0 \cdots 0\,1\,0 \cdots 0\,0 \cdots 0\,1\,0 \cdots 0\,0 \cdots 0\,1\,0 \cdots 0]$

u v+n w+2n

define $a_j = b^u + b^{v+n} + b^{w+2n}$

- since $b = m+1$, addition of vectors won't cause overflow

- $a_j \leq (m+1)^{3n+1}$ so $\log_2(a_j) \in (mn)^{O(1)}$
- set $K = b + b^2 + \ldots + b^{3n}$

↳ claim: for S subset of $\{1, \ldots, m\}$, $\sum_{i \in S} v_i = [1 \cdots 1] \iff \sum_{i \in S} a_i = K$

- proof:

We always have $\sum_{i \in S} a_i = \sum_{j=1}^{3n} c_j b^j$ w/ c_j = # v_is in S w/ $v_{ij} = 1$.

⇒:

Assume $\sum_{i \in S} v_i = [1 \cdots 1]$ and $c_j = 1$ for all j. Then, $\sum_{i \in S} a_i = \sum_{j=1}^{3n} b_j = K$.

⇐:

Assume $\sum_{i \in S} a_i = K$. Then, $\sum_{j=1}^{3n} b^j = \sum_{j=1}^{3n} c_j b^j$