Basics from CS 240

time space

- Algorithms focus on correctness, efficiency,
- Runtime - count "elementary operations"
  - function of a measure of the size of the input n.
  - asymptotic notation: $O, \Omega, \Theta, o, \omega$
  - worst case, average case, ...
  - recurrence · induction proofs
- Pseudocode
  - try to be realistic about "elementary operations"
- Model of Computation

Algorithm Paradigms (CS 341 Overview)

- Divide and Conquer
- Greedy
  - graphs
- Dynamic Programming
  - memoization
- Reductions: solving new problems based on things we already know
  - NP-completeness, undecidability, trackability
  - P - polynomial
  - P = NP ?

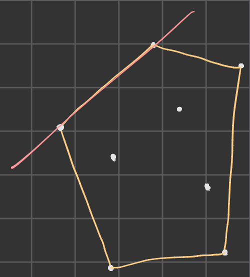
Lower Bounds - Do we have the best algorithm?

- Decision Tree (comparison based)

Convex Hull

Problem: Given n points, find convex hull (smallest convex set containg the points)

Equivalent: The convex hull is a polygon whose sides are formed by lines L that go through at least 2 points and have no points on one side of L.

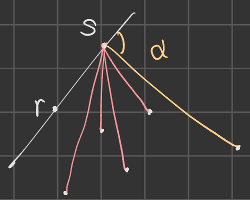


Algorithm 1: Find all edges st. $*$ holds $O(n^3)$

Find a line, check $*$ (whether all points lie on just one side of L)
$\binom{n}{2} = O(n^2)$. $O(n)$
lines points

Algorithm 2: Jarvis March $O(n^2)$ or $O(kn)$

- Once we found one line L, there is a natural next line L'.
- Find extreme one - minimize angle α $O(n)$

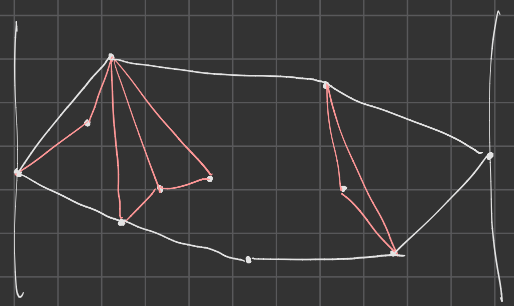


- Repeat until we have all edges of the convex hull $O(n)$

If we let k be # edges on convex hull $\Rightarrow$ $O(k)$

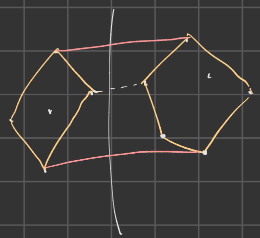
Algorithm 3: Reduction $O(n\log n)$

- Sort points by x-coord $O(n\log n)$
- Start at leftmost point to build the edges of convex hull $O(n)$



Algorithm 4: Divide and Conquer

- Partition into 2 sets
- Find convex hull on each side
- Merge into single convex hull



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

$O(n\log n)$

Can we do better ?

Convex Hull

Can we do better ?

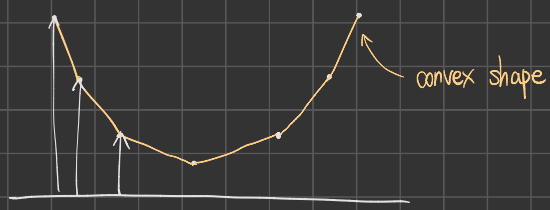
<u>Reduction:</u> If we can find a faster algorithm for C.H. then we can also sort faster.

Given n numbers to sort $(x_1, \dots, x_n)$

Map points x_i to x_i^2 (x_i, x_i^2). $\Theta(n)$

Call C.H., $f(n)$ for C.H

Start leftmost, go right on lower C.H $\Rightarrow$ sorted order $\Theta(n)$



$\Omega(n\log n)$

$\Theta(n + f(n))$ total for sorting

C.H. is comparison-based $\Rightarrow \Omega(n\log n)$

Timothy Chan: $O(n\log h)$ better than $O(n \cdot h)$ and $O(n\log h)$

Model of Computation

Pseudocode

$A[1 \dots n] = \{0\}$ // initialize array to 0, $O(n)$

Be mindful of operations that look like constant but are not.

Integers can get large, e.g. Fibonacci numbers

Can use bit counts. Storing n takes $\Theta(\log n)$ bits.

Multiplication

Computing $a * b$ takes $O(\log a \cdot \log b)$ // normal math

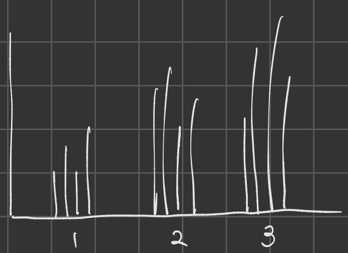
Can we do better ? There exists a way to do it in $O[(\log a)(\log b)^{0.59}]$ // faster

Random Access Machine

Accessing memory location takes constant time

Runtime

- Average, worst-case, ...
- Want simple functions, e.g. $n\log n$, n^2, etc.



Examples:

- $(n+1)! = (n+1)n! \notin O(n!)$
- $\log(n!) \in \Theta(n\log n)$

Suppose worst-case runtime of

- Alg 1 is $O(n^2)$
- Alg 2 is $O(n\log n)$

Which is better?

Difficult to say. To compare, use Theta bounds.

O is an upper bound, may not be tight.

Divide and Conquer

Example 1: Binary Search

$$
\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + c \\
&= T(\tfrac{n}{4}) + c + c \\
&= T(1) + \underbrace{c + \cdots + c}_{O(\log n)} \\
&= d + c\,O\log(n) \\
&\in O(\log n)
\end{aligned}
$$

Example 2: QuickSort

Worst case: $T(n) = T(n-1) + cn \in O(n^2)$

Best case: $T(n) = 2T(\frac{n}{2}) + cn \in O(n\log n)$

Example 3: MergeSort.

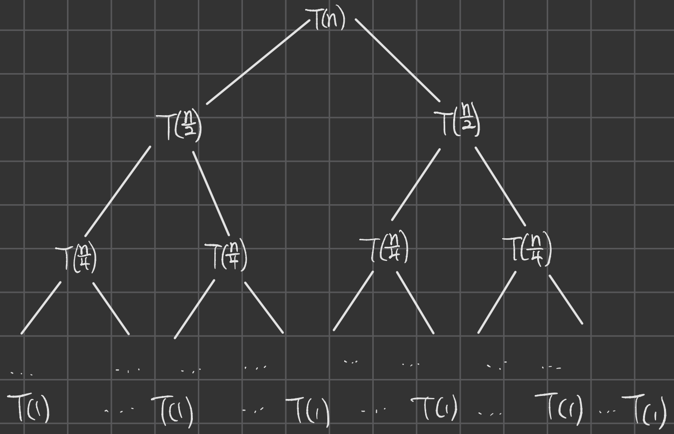
$T(n) = 2T(\frac{n}{2}) + cn \in O(n\log n)$

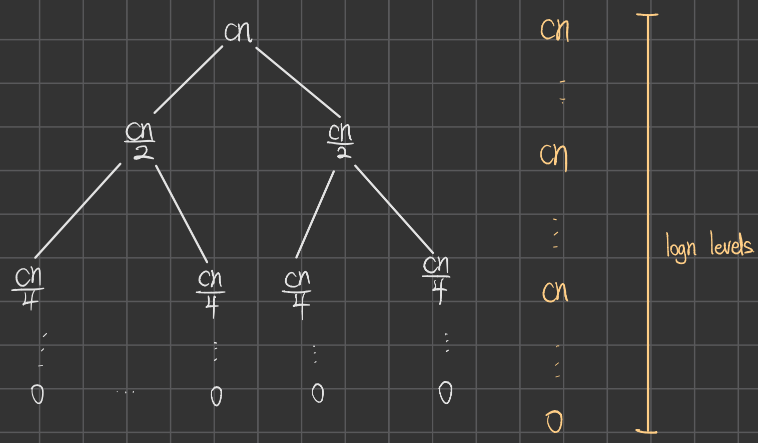
Rigorously, $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$

Recursion Tree for MergeSort

$T(n) = 2T(n/2) + cn$, if n is even

$T(1) = 0$, if counting number of comparisons





$$T(n) = c\left(n + 2\cdot\frac{n}{2} + 4\cdot\frac{n}{4} + \cdots + \frac{n}{2}\cdot\frac{n}{n/2}\right) + n\cdot 0$$
$$= cn\log n$$
$$\in \Theta(n\log n)$$

Solving Recurrences by Substitution

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + (n-1) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

Base Case: $0 = T(1) = cn\log n = c\cdot\log(1) = 0$

Induction step: separate into 2 cases - n even and n odd

For n even:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n - 1 \\ &\le 2c\cdot\frac{n}{2}\log\frac{n}{2} + n - 1 \\ &= cn\log\frac{n}{2} + n - 1 \\ &= cn\log n - cn\log 2 + n - 1 \\ &= cn\log n - cn + n - 1 \\ &= cn\log n - (c-1)n - 1 \\ &\le cn\log n \quad \text{if } c \ge 1 \end{aligned}$$

For n odd:

$$\begin{aligned} T(n) &= T\left(\frac{n-1}{2}\right) + T\left(\frac{n+1}{2}\right) + n - 1 \\ &\le c\left(\frac{n-1}{2}\right)\log\left(\frac{n-1}{2}\right) + c\left(\frac{n+1}{2}\right)\log\left(\frac{n+1}{2}\right) + n - 1 \\ &\le c\left(\frac{n-1}{2}\right)\log\left(\frac{n}{2}\right) + c\left(\frac{n+1}{2}\right)\left[\log\left(\frac{n}{2}\right) + 1\right] + n - 1 \\ &= c\left(\frac{n-1}{2}\right)\log\left(\frac{n}{2}\right) + c\left(\frac{n+1}{2}\right)\log\left(\frac{n}{2}\right) + c\cdot\frac{n+1}{2} + n - 1 \\ &= cn\log\left(\frac{n}{2}\right) + c\cdot\frac{n+1}{2} + n - 1 \\ &\le cn\log n + c\cdot\frac{n+1}{2} + n - 1 \\ &\le 2cn\log n \end{aligned}$$

* Fact: $\log\left(\frac{n+1}{2}\right) < \log\left(\frac{n}{2}\right) + 1 \quad \forall n \ge 3$

The constant is growing.

alternative:

$$\begin{aligned} &\le cn\log n - cn\log 2 + \frac{cn}{2} + \frac{c}{2} + n - 1 \\ &= cn\log n - cn + \frac{cn}{2} + n + \frac{c}{2} - 1 \\ &= cn\log n - \left(c - \frac{c}{2} - 1\right)n + \frac{c}{2} - 1 \\ &= cn\log n - \left(\frac{c}{2} - 1\right)n + \frac{c}{2} - 1 \\ &\le cn\log n \quad \text{if } c \ge 2 \text{ and } n \ge 1 \end{aligned}$$

$$\left(\frac{c}{2} - 1\right)n - \left(\frac{c}{2} - 1\right) \ge 0$$
$$\Leftrightarrow \left(\frac{c}{2} - 1\right)n \ge \frac{c}{2} - 1$$
$$\Leftrightarrow \frac{c}{2} - 1 \ge 0 \quad \text{for } n \ge 1$$

Watchout for Common Pitfall

$T(n) = 2T(n/2) + n$

Claim : $T(n) \in O(n)$.

Prove $T(n) \leq cn$, $\forall n \geq n_0$

$$
\begin{aligned}
T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\
&\leq 2c \cdot \tfrac{n}{2} + n \\
&= cn + n \\
&= \underline{(c+1)n}
\end{aligned}
$$

Problem: The constant is growing.

Substitution - Changing the Guess

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

$$
\begin{aligned}
T(n) &= T\left(\left\lceil \tfrac{n}{2} \right\rceil\right) + T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 1 \\
&\leq c \cdot \left\lceil \tfrac{n}{2} \right\rceil + c \cdot \left\lfloor \tfrac{n}{2} \right\rfloor + 1 \\
&= cn + \underline{1}
\end{aligned}
$$

Try proving $T(n) = cn - 1$ instead.

$$
\begin{aligned}
T(n) &= T\left(\left\lceil \tfrac{n}{2} \right\rceil\right) + T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 1 \\
&\leq c\left\lceil \tfrac{n}{2} \right\rceil - 1 + c\left\lfloor \tfrac{n}{2} \right\rfloor - 1 + 1 \\
&= cn - 1
\end{aligned}
$$

Example - Similarity Between Rankings

Ranking 1: B, $\boxed{D}$, C, A
Ranking 2: A, $\boxed{D}$, B, C
Exact match = 1

$\binom{4}{2} = 6$ pairs
How many have the same order? 2 - BC and DC
not very similar

Counting Inversions

1	2	3	4
B	D	C	A
A	D	B	C
4	2	1	3

Problem: How many pairs are in order? e.g. (2, 3) and (1, 3)
Brute force: check every pair - $\Theta(n^2)$

Divide and Conquer

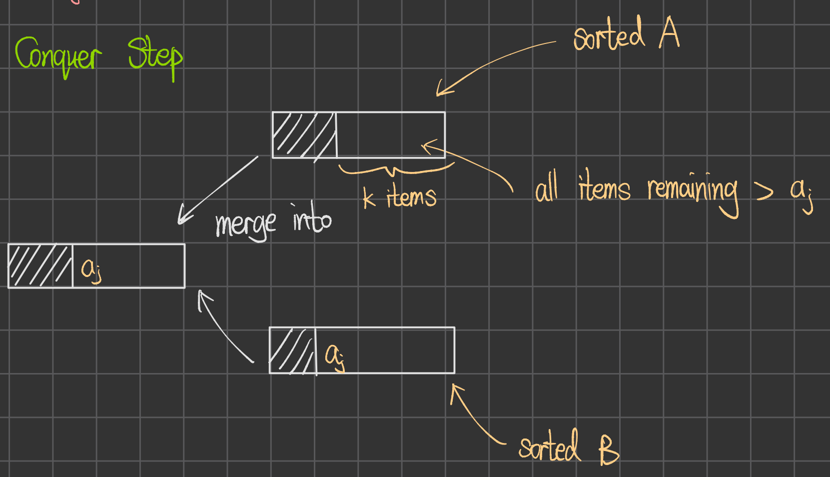
Divide into 2 halves.

answer from A (left half)
answer from B (right half)

answer $= r_A + r_B + r$ For each $a_j \in B$, count the number of items, r_j, in A that are larger than a_j

$r = \sum_{a_j \in B} r_j$

Conquer Step



When a_j is merged: $r_j = k$

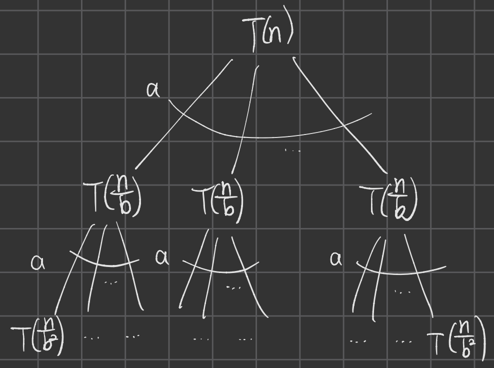
$r = \sum r_j$

Analysis

$T(n) = 2T(\frac{n}{2}) + O(n) \in O(n \log n)$ (similar to mergesort).

Master Theorem - Recursion Tree

$$T(n) = \begin{cases} aT(\frac{n}{b}) + cn^k & n \leq 1 \\ c & n > 1 \end{cases}$$



$cn^k = cn^k$

$a \cdot c \cdot (\frac{n}{b})^k = cn^k \cdot (\frac{a}{b^k})$

$a^2 \cdot c \cdot (\frac{n}{b^2})^k = cn^k (\frac{a}{b^k})^2$

How many levels? $\log_b n$

$\Rightarrow T(n) = aT(\frac{n}{b}) + cn^k$

$= a[aT(\frac{n}{b^2}) + c(\frac{n}{b})^k] + cn^k$

$= a^2 T(\frac{n}{b^2}) + ac(\frac{n}{b})^k + cn^k$

$= a^3 T(\frac{n}{b^3}) + a^2 c(\frac{n}{b^2})^k + ac(\frac{n}{b})^k + cn^k$

$\cdots$

$= a^{\log_b n} T(1) + \sum_{i=0}^{\log_b n - 1} a^i c (\frac{n}{b^i})^k$

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

Case 1: $a < b^k \Leftrightarrow \log_b a < k$

Then $\sum (\frac{a}{b^k})^i$ is a geometric series with $\frac{a}{b^k} < 1$.

$\Rightarrow$ Sum is constant.

$\Rightarrow T(n) = n^{\log_b a} T(1) + \Theta(n^k)$

$\Rightarrow \boxed{T(n) \in \Theta(n^k)}$ since $\log_b a < n^k$

Case 2: If $a = b^k \Leftrightarrow \log_b a = k$

Then $\sum_{i=0}^{\log_b n - 1} (\frac{a}{b^k})^i = \sum_{i=0}^{\log_b n - 1} 1 = \Theta(\log_b n)$

$\Rightarrow T(n) = n^{\log_b a} T(1) + cn^k \cdot \Theta(\log_b n)$

$= n^k \cdot \Theta(1) + cn^k \cdot \Theta(\log_b n)$

$= \boxed{\Theta(n^k \log n)}$

Case 3: If $a > b^k \Leftrightarrow \log_b a > k$.

Then $\sum_{i=0}^{\log_b n - 1} (\frac{a}{b^k})^i$ is a geometric series with $\frac{a}{b^k} > 1$.

The last term of $\sum$ dominates.

$\Rightarrow T(n) = n^{\log_b a} T(1) + \Theta(n^k (\frac{a}{b^k})^{\log_b n})$

$= n^{\log_b a} T(1) + \Theta(n^k \cdot a^{\log_b n} \cdot \frac{1}{(b^{\log_b n})^k})$

$= n^{\log_b a} T(1) + \Theta(\cancel{n^k} \cdot a^{\log_b n} \cdot \frac{1}{\cancel{n^k}})$

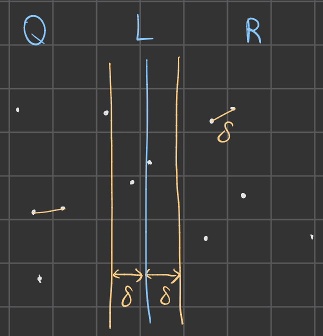
$= n^{\log_b a} T(1) + \Theta(a^{\log_b n})$

$= n^{\log_b a} T(1) + \Theta(n^{\log_b a})$

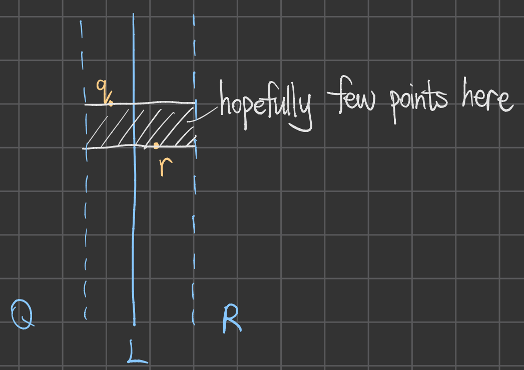
$= \boxed{\Theta(n^{\log_b a})}$

Closest Pair - Divide and Conquer

General Idea



Sort by x-coord once $O(n \log n)$
Can access the points we want in $O(n)$ time.
Let $\boxed{\delta}$ be the min-distance of closest pair both in L and R.
Need to find a pair (q, r), $q \in Q$ and $r \in R$ such that $d(q,r) < \delta$
Candidates: $d(q, L) < \delta$ and $d(r, L) < \delta$
If one point p is outside the band of width 2δ, then $d(p, L) > \delta$
* S may contain $O(n)$ or all the points



Sort by y-coord - only once
On the recursive subproblems, pull out relevant points in $O(n)$.
Once extracted they are in sorted order.

Algorithm Overview

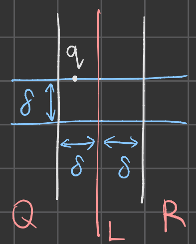
$X \leftarrow$ points sorted by x-coord $\quad O(n\log n)$
$Y \leftarrow$ points sorted by y-coord $\quad O(n\log n)$

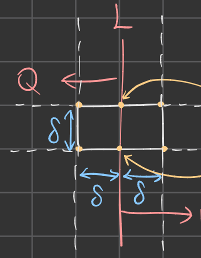
ClosestPair(X, Y) returns distance between closest pair of points.

- $L \leftarrow$ dividing line (middle of X)
- Extract X_Q, X_R (sorted points by x-coord in region Q, R) $\quad O(n)$
- Extract Y_Q, Y_R (sorted points by y-coord in region Q, R) $\quad O(n)$
- $\delta_Q \leftarrow$ ClosestPair(X_Q, Y_Q) $\quad T(\frac{n}{2})$
- $\delta_R \leftarrow$ ClosestPair(X_R, Y_R) $\quad T(\frac{n}{2})$
- $\delta \leftarrow \min\{\delta_Q, \delta_R\}$
- Find S (vertical strip of width 2δ around L)
- $Y_S \leftarrow S$ sorted by y-coord (extracted from Y) $\quad O(n)$

What do we do with S, Y_S?

Hope: If $q, r \in S$ and $q \in Q$ and $r \in S$ and $d(q,r) < \delta$, then q, r are near each other in Y_S





At most 8 points ahead to check.
(7 other)

For each $s \in Y_S$, check distances with the next 7 points in Y_S.

Running Time

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + dn & n > 2 \\ d & n = 2 \end{cases}$$

$\Rightarrow T(n) \in O(n\log n)$

QuickSelect

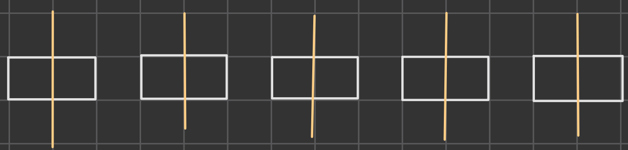
Runtime based on where pivot falls

Average: $T(n) \leq \begin{cases} cn + \frac{1}{2}T(n) + \frac{1}{2}T(\frac{3n}{4}) & n \geq 2 \\ d & n = 1 \end{cases}$

BFPRT - Blum Floyd Pratt Rivert Tarjan

- worst case $\Theta(n)$
- Choose pivot so that it is close to the middle
- $n = 10r + 5 + \theta$ where $r \geq 1$ and $0 \leq \theta \leq 9$
- Blocks of size 5 odd number of them

MOM - median of medians



- Find median of each block of size 5.
- Find median of medians m, choose as pivot
  - r blocks have median less than m
  - 3 elements of each block $< m$

  $\Rightarrow$ $3r + 2$ elements $< m$

  $\Rightarrow$ $n - (3r+2) - 1 = n - 3\left(\frac{n-5-\theta}{10}\right) - 2 - 1$

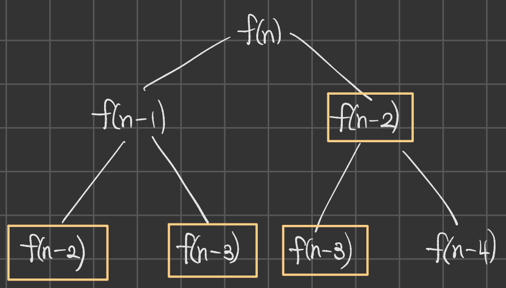
$$\frac{10n}{10} + \frac{-3n+15+3\theta}{10} - \frac{30}{10} = \frac{7n-15+3\theta}{10} \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$$

$$\begin{cases} T(n) \leq T(\frac{n}{5}) + T(\lfloor \frac{7n+12}{10} \rfloor) + \Theta(n). & n \geq 15 \\ d & n < 15. \end{cases}$$

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

Dynamic Programming

Fibonacci



Repetition of subproblems.

Text Segmentation

A string of letters $x[1..n]$ where $x[i] \in \{A..Z\}$. Can A be split into (2 or more) words?

Build up a solution for A[n] from smaller subproblems

	1	2	3	4	5	6	7	8	9	
	T	H	E	M	E	M	P	T	Y	
	0	0	1	1	1	0	0	0	1	// can be broken into words or not.
	-1	-1	1	1	1	-1	-1	-1	5	// starting index.

Runtime: $O(n^2)$

$$S[i] = \begin{cases} 1 & \text{if } (S[0]=1 \wedge \text{Word}(x[1..n])) \vee (S[1]=1 \wedge \text{Word}(x[2..n])) \vee \cdots \vee (S[n-1]=1 \wedge \text{Word}(x[n..n])) \\ 0 & \text{otherwise} \end{cases}$$

Longest Increasing Subsequence

Input: a sequence of numbers $A[1..n]$ where $A[i] \in \mathbb{N}$

	0	1	2	3	4	5	6	7	8	9	
	∅	5	2	1	4	3	1	6	9	2	∞ — dummy
M[i] = length	0	1	1	1	2	2	1	3	4	2	5
S[i] = j used	0	0	0	0	3	3	0	5	7	6	8

M[i] = length of LIS that includes the element M[i]

$$M[i] = \max \begin{cases} \text{if } Inc(1,i) = 1 \text{ then } M[1]+1 \text{ else } 1. \\ \text{if } Inc(2,i) = 1 \text{ then } M[2]+1 \text{ else } 1. \\ \cdots \\ \text{if } Inc(i-1,i) = 1 \text{ then } M[i-1]+1 \text{ else } 1. \end{cases}$$

Runtime: $O(n^2)$

Longest Common Subsequence

Input: $A[1 \cdots n]$ and $B[1 \cdots m]$ of characters

Base cases: $M(i, 0) = 0$ and $M(0, j) = 0$

$M(i,j) = M[i][j] =$ length of LIS of $A[1 \cdots i]$ and $B[1 \cdots j]$ (prefixes)

$$M(i,j) = \begin{cases} 1 + M(i-1, j-1) & \text{if } x_i = y_j \\ \max\{M(i-1,j),\ M(i, j-1)\} & \text{otherwise} \end{cases}$$

	Ø	C	A	T	A	M	A	R	A	N
Ø	0	0	0	0	0	0	0	0	0	0
T	0	0	0	$\boxed{1}$	1					
A	0	0	0	1	$\boxed{2}$					
M	0									
A	0									
R	0									
A	0									
C	0									

Algorithm Overview

```
for i = 0...n,  M(i,0) ← 0                  // initialize first row with 0.
for i = 0...n,  M(0,j) ← 0                  // initialize first column with 0.
for i = 1...n:
    for j = 1...m
        if x_i = y_j then                   // match a character
            M(i,j) ← 1 + M(i-1, j-1)
        else                                // skip one character
            M(i,j) ← max{ M(i-1, j), M(i, j-1) }
```

Finding Actual Solution

Work backwards

```
OPT(i,j)
   if i=0 or j=0 then return          // base case
   if M(i,j) = M(i-1,j) then
      OPT(i-1, j)
   else if M(i,j) = M(i, j-1)
      OPT(i, j-1)
   else                               // x_i = y_j
      output i,j
      OPT(i-1, j-1)                   swap?
```

Edit Distance

Input: 2 strings $A[1...n]$, $B[1...m]$. Find edit distance between x and y.
min # of changes to match x and y

A change is one of:
- insert a letter
- delete a letter
- replace with another letter

Change 1

replacement

S	_	N	O	W	Y
S	U	N	N	_	Y

insertion deletion

3 changes (edits)
- insert U
- replace O with N
- delete W

Change 2

_	S	N	O	W	_	Y
S	U	N	_	_	N	Y

5 changes (edits)

Subproblem: $M(i,j) = M[i][j]$ = min # of changes to match $x[1...i]$ and $y[1...j]$

Base Cases: $M(i,0) = i$ and $M(0,j) = j$

Recurrence:

$$M(i,j) = \min \begin{cases} M(i-1, j-1) & \text{if } x_i = x_j \\ 1 + M(i-1, j-1) & \text{replace} \\ 1 + M(i-1, j) & \text{delete} \\ 1 + M(i, j-1) & \text{insert} \end{cases}$$

no change
ABCD ⟶ ABCC
ABCD ⟶ ABC_
ABC_ ⟶ ABCD

Weighted Interval Scheduling

Subproblem: $M(i)$ = maximum weight subset of intervals $I_1, \dots, I_i$.

Let $w(i)$ be the weight of I_i, and $p(i)$ be the latest interval among the set of intervals before i and disjoint from i.

$$M(i) = \max \begin{cases} M(i-1) & \text{not choosing } I_i \\ w(i) + M(p(i)) & \text{choosing } I_i \end{cases}$$

Order intervals $1 \dots n$ by right endpoint. $p(i)$ is the rightmost interval that does not overlap i.

Improvement 1

Compute all $p(i)$ first.

Sort intervals by right endpoints, relabel to get $r_1, \dots, r_n$. AND sort by left endpoints to get $\underbrace{l_1, \dots, l_n}_{\text{endpoints}}$.

$j \leftarrow n$
for k from n down to 1:
  while r_j overlaps l_k:
    $j \leftarrow j-1$ $O(n)$ since j is always decreasing.
  $p(l_k) \leftarrow j$

Overall runtime for Weighted Interval Scheduling: $O(n\log n)$

Revised Algorithm

Sort by finish time
Sort by start time
Compute all $p(i)$
$M(0) \leftarrow 0$
for $i = 1 \dots n$
  $M(i) = \max\{M(i-1),\ w(i) + M(p(i))\}$

Improvement 2

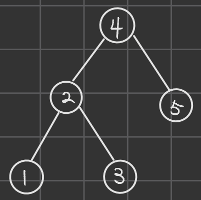
Recover S by recursive backtracking - save space

S-OPT(i)
 if $i=0$ then return $\emptyset$
 else if $M(i-1) \geq w(i) + M(p(i))$ not choosing I_i (skipped i)
  return S-OPT(i-1)
 else   choosing I_i
  return $\{i\} \cup$ S-OPT(p(i))

Optimal Binary Search Trees

$\# \text{ nodes} \cdot \text{ProbeDepth} \cdot \text{probability}$

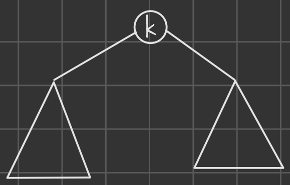
Example: $p_1 = p_2 = p_3 = p_4 = p_5 = \frac{1}{5}$



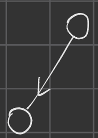
Search Cost $= 1 \cdot \frac{1}{5} + 2 \cdot 2 \cdot \frac{1}{5} + 2 \cdot 3 \cdot \frac{1}{5} = \frac{11}{5}$

Dynamic Programming Approach

Choose key k to be root:



$M(i,j)$ - subproblem of $i \dots j$



$1 \cdot 1 \cdot p_i$

$\Downarrow$

$1 \cdot 2 \cdot p_i$

depth $+1 \Rightarrow$ add one more p_i to search cost.

Let $M[i,j]$ be the optimal BST search cost on items $i \dots j$

$$M[i,j] = \min_{i \le k \le j} \{ M[i, k-1] + M[k+1, j] \} + \boxed{\sum_{t=i}^{j} p_t}$$

L and R subtrees have nodes 1 deeper, so add one more p_l for all $i \le l \le j$. This is independent of choice k.

Improvement

$$\sum_{t=i}^{j} p_t = p_i + p_{i+1} + \dots + p_j \qquad O(j-i+1) = O(n)$$

Let $P[i] = \sum_{t=1}^{i} p_t$ where $P[0] = 0$, so $\sum_{t=i}^{j} p_t = P[j] - P[i-1]$ $\quad O(1)$ - one subtraction

0-1 Knapsack

Input A set of items $\{1 \dots n\}$ where item i has weight w_i and value v_i and a knapsack with capacity W.
Goal A subset of items S such $\sum_{i \in S} w_i \le W$ so that $\sum_{i \in S} v_i$ is maximized.

Note: "0-1" means you either take an item or not.

Dynamic Programming Approach

$M[i]$ = maximum value from items 1 to i.
- do not take i: $M[i-1]$
- take i: $M[i-1] + v_i$ not considering weight restriction

2-dimensional subproblem

$M[i, w] = \max \sum_{i \in S} v_i$
Goal: compute $M[n, W]$

To find $M[i, w]$:
- * if $w_i > w$, then $M[i, w] \leftarrow M[i-1, w]$ no room; not choosing item i
- else $M[i, w] \leftarrow \max \begin{cases} M[i-1, w], & \text{not choosing item } i \\ M[i-1, w - w_i] + v_i & \text{choosing item } i \end{cases}$

  option of choosing i

Pseudocode

```
M[0, w] ← 0 for w = 0 ... W
for i = 1 ... n:
    for w = 0 ... W:
        compute M[i, w]   *
```

Runtime: $O(nW)$ pseudopolynomial

Runtime is polynomial in value of W rather than the size of W.

Recover Items

① Backtracking - use M to recover decision made

```
i ← n, w ← W, S ← ∅
while i > 0
    if M[i, w] = M[i-1, w] then        // did not choose i.
        i ← i-1
    else                               // chosen i.
        S = S ∪ {i}.
        w = w - w_i
```

② Store decision - also use Take(i, w) ; 0 (not take) or 1 (take)

- do not need to do comparison
- still need to backtrack through subproblems

③ Store the set of items

- more space
- faster

Memoization

0-1 Knapsack

$$M(i,w) = \max \begin{cases} M(i-1, w) \\ v_i + M(i-1, w-w_i) \end{cases}$$

Compute $M(i,w)$ for $0 \le i \le n$, $0 \le w \le W$.

Do we need all of the subproblems we compute?

Interval Scheduling

Input: A set $I = \{1 \dots n\}$ of intervals
Goal: A subset $S \subseteq I$ of pairwise disjoint intervals of maximum size

Greedy Choice

Select intervals with earliest finish time.

Sort intervals $1 \dots n$ by finish time and relabel so $f_1 \le \dots \le f_n$
$S = \emptyset$
for $i \leftarrow 1$ to n
    if interval i is pairwise disjoint with intervals in S then
    $S \leftarrow S \cup \{i\}$.

} Total work is $O(n)$ since we only examine each interval once

Runtime: $\underbrace{O(n\log n)}_{\text{sort}} + \underbrace{O(n)}_{\text{construct } S} \in O(n\log n)$

Proof of Correctness

Suppose greedy algorithm returns $a_1 a_2 \dots a_k$
- sorted by end time

Suppose an optimal solution is $b_1 \dots b_k b_{k+1} \dots b_l$ ($k \le l$ since $b_1 \dots b_k b_{k+1} \dots b_l$ is optimal)
- sorted by end time

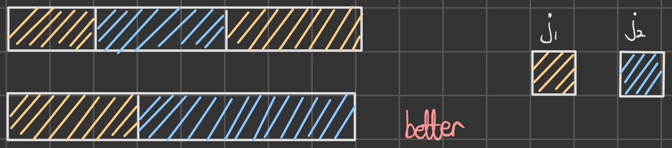
a_1 ends before b_1 - greedy always choose interval that ends first.
$\Rightarrow a_1 b_2 \cdots b_k b_{k+1} \dots b_l$ is a correct and optimal solution.
$\because$ end$(a_1) \le$ end(b_1) so a_1 does not overlap $b_2, b_3, \dots$
$\Rightarrow a_1 a_2 b_3 \dots b_k b_{k+1} \dots b_l$ is optimal by a similar argument.
$\because$ b_2 does not intersect with a_1 - otherwise greedy would not have chosen it.
Greedy chose a_2 over b_2, so end$(a_2) \le$ end$(b_2) \le$ start(b_3) and these intervals are disjoint.
Continue replacing to get $a_1 \dots a_k b_{k+1} \dots b_l$ is an optimal solution

At every step, we can do at least as good as a_i (Greedy stays ahead)

To finish, we argue that if $k < l$ then $a_1 \dots a_k b_{k+1} \dots b_l$ is an optimal solution, but then greedy algorithm would have chosen b_{k+1}.
Therefore $k = l$. (We had that $k \le l$)

Scheduling to Minimize Lateness

Observation 1: Once you start a job, always complete it.

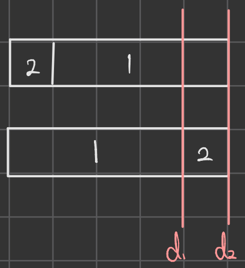


Observation 2

There is never any value in taking a break.

Greedy Approaches

1. Shortest task first ✗



job 1 is late

no lateness (maximum lateness = 0)

2. Do task with least slack

3. Optimal: Do task with <u>earliest deadline</u>

Order jobs by deadline (relabel) so $d_1 \le d_2 \le \dots \le d_n$.

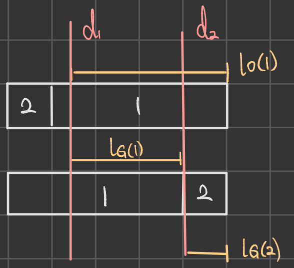
Proof Advice:
- Do not be general at first
- Try special cases
- Compare wrong/other solutions with greedy solution

Scheduling to Minimize Lateness

Greedy Choice: order jobs by deadline.

Other Solution O:

Greedy Solution G:



$l_O(x)$ = lateness of job x in O, $l_G(x)$ = lateness of job x in G

$l_G = \max\{l_G(1), l_G(2)\}$

$l_O = \max\{l_O(1), l_O(2)\}$

$l_G(1) \le l_O(1)$ since G does 1 earlier

$l_G(2) \le l_O(1)$ since $d_1 \le d_2$, and job 1 in O and job 2 in G finish at the same time

$\Rightarrow l_G \le l_O(1) \le l_O$

Theorem: Greedy solution is optimal.

Let $1 \dots n$ be ordering of jobs by greedy algorithm. $d_1 \le d_2 \le \dots \le d_n$.

Consider an optimal ordering.

- If it's the same as G then we're done.
- If not, there must be 2 jobs that are consecutive but in different order.

$\Rightarrow$ jobs i, j with $d_j \le d_i$

Claim: swapping i and j gives a new optimal solution.

- new optimal solution has fewer inversions.

Proof of claim:

$l_O(x)$ = lateness of job x in O, $l_G(x)$ = lateness of job x in G

$l_G = \max\{l_G(1), l_G(2)\}$

$l_O = \max\{l_O(1), l_O(2)\}$

$l_G(1) \le l_O(1)$ since O does 1 later

$l_G(2) \le l_O(1)$ since $d_1 \le d_2$, and job 1 in O and job 2 in G finish at the same time

$\Rightarrow l_G \le l_O(1) \le l_O$

Replace 1 by i and 2 by j.

Fractional Knapsack

Proof of Correctness:

Contradiction. Suppose greedy solution is not Optimal. Compare Greedy with Optimal, Show that Greedy is better than Optimal.

Let k be the first index where $x_k \neq y_k$
Then $x_k > y_k$ since greedy maximizes x_k.
Since $\sum x = \sum y = W \Rightarrow$ there is a later item index l where $l > k$ such that $y_l > x_l$. (can prove by contradiction?)

$$
\begin{array}{lccccccc}
G: & x_1 & \dots & \boxed{\begin{matrix} x_k \\ > \\ y_k \end{matrix}} & \dots & \boxed{\begin{matrix} x_l \\ < \\ y_l \end{matrix}} \\
 & \parallel & \parallel & & & \\
O: & y_1 & \dots & & \dots &
\end{array}
$$

Note that $\frac{v_k}{w_k} \geq \frac{v_l}{w_l}$ because of ordering.

Exchange some of item l for equal weight of item k in the optimal solution.
$y'_k \leftarrow y_k + \Delta$
$y'_l \leftarrow y_l - \Delta$

choose $\Delta \leftarrow \min \begin{cases} y_l - x_l \\ x_k - y_k \end{cases}$

Then either $x_k = y_k'$ or $x_l = y_l'$

Change in value: $\Delta\left(\frac{v_k}{w_k}\right) - \Delta\left(\frac{v_l}{w_l}\right) = \Delta\left(\frac{v_k}{w_k} - \frac{v_l}{w_l}\right) \geq 0$ (since $\frac{v_k}{w_k} \geq \frac{v_l}{w_l}$)

y was optimal, so this could not be better, contradiction is found.
New optimal matches on one more index (k or l).

Graph Representation

$|V| = n$ $\qquad m \leq n^2$

$|E| = m$

Matrix Representation:

(1, 2) is an edge.

$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & \cdots \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ \vdots \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & \\ & & & \\ & & & \\ & & & \end{bmatrix} \end{array}$$

Adjacency List

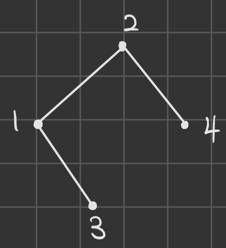
$1 \rightarrow 2 \rightarrow 3$

$2 \rightarrow 4 \rightarrow 1$

$3 \rightarrow 1$

$\vdots \qquad \vdots$

n



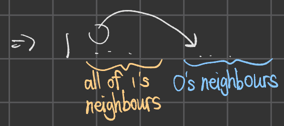
each edge appears in 2 nodes

Space: $O(n + 2m) \in O(n+2m)$

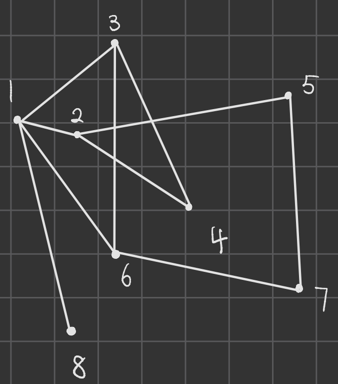
\# vertices

BFS

· Start at 1, find its neighbours, explore these next
  - find neighbours of neighbours

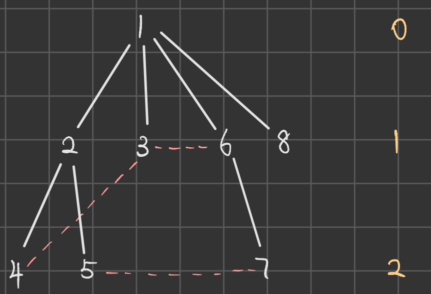


Data Structure: queue



BFS starting from 1.
⇒

non-tree edges



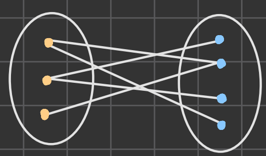
Order of discovery: 1 2 3 6 8 4 5 7

2m (Hand-shaking Lemma)

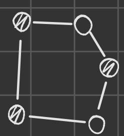
Runtime: $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

Note:
If (u,v) is a non-tree edge, then either $u.d = v.d$ or $|u.d - v.d| = 1$. levels differ by at most 1.

Applications
- Find the shortest path from v_0 to any node v (length = depth of v_0).
- Find cycles (any non-tree edge)
- Test if graph is bipartite
  - A: even levels in the BFS tree
  - B: odd levels in the BFS tree.
  - non-tree edges can only be on the same level or in levels with depth differing by 1.
    same-level non-tree edges ⇒ odd simple cycle exists ⇒ graph is not bipartite



bipartite graph.

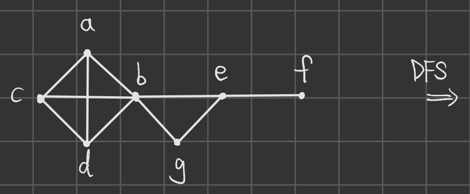


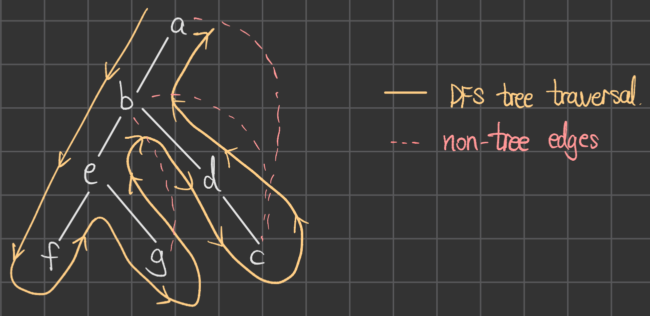
odd simple cycle.

least common ancestor ⇒ simple cycle.

non-tree edge

DFS





Order of discovery: a b e f g d c
1 2 3 4 6 9 10

Finish order: f g e c d b a
5 7 8 11 12 13 14

Lemma. All non-tree edges join an ancestor and a descendant in the DFS tree.

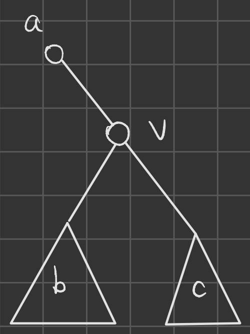
Parantheses structure.

Let $d(v)$ be the discovery time of v, and $f(v)$ be the finish time of v for all $v \in V$.

Discovery and finish times form a parenthesis system

If $d(v) < d(u)$, then $\underbrace{d(v) < d(u) < f(u) < f(v)}_{\text{nested}}$ OR $\underbrace{d(v) < f(v) < d(u) < f(u)}_{\text{disjoint}}$

Cut Vertex



$d(a)$ $d(v)$ everything in T $f(v)$ $f(a)$.

Definitions:

A vertex v is a cut vertex if $G-v$ is not connected

An edge e is a cut edge if $G-e$ is not connected.

can be u itself.

Let u be the root of a subtree T of v, x be a descendant of u.

at least one

Define: $low(u) = \min\{d(w) : (x, w) \text{ is a non-tree edge}\}$

Claim. A non-root vertex v is a cut vertex if and only if v has a child u with $\underline{low(u) \geq d(v)}$

no edge going above v.

Compute $low(u)$ recursively

$$low(u) = \min \begin{cases} \min\{d(w) : (u, w) \in E\} \\ \min\{low(x) : x \text{ is a descendant of } u\} \end{cases} \quad (*)$$

low(u) records how far up we can go from the subtree rooted at u.

Runtime: $O(n+m)$

Compute all cut vertices

Enhance DFS to compute low, or

finish first ⇒ higher depth.

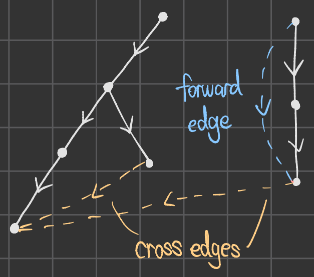
Run DFS to compute discovery times,

Then, for every vertex u in finish time order, use $(*)$ to compute $low(u)$

For every non-root v, if v has a child u with $low(u) \geq d(v)$, then v is a cut vertex.

The root is a cut vertex if and only if it has more than one child.

DFS on directed graphs

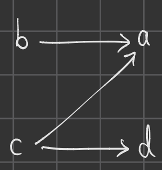


Lemma: A directed graph has a (directed) cycle $\Leftrightarrow$ DFS tree has a back edge

Proof: Suppose there is a directed cycle.
Let v_1 be the first vertex discovered in the DFS.
Number vertices in cycle $v_1, \dots, v_k$.
Claim: (v_k, v_1) is a back edge
Proof of claim: we must discover and explore all v_i before we finish v_1 since $v_1 \rightsquigarrow v_i$, for all $i = 2, \dots, k$
When we test edge (v_k, v_1), we label it a back edge since v_1 is an ancestor of v_k in the DFS tree

Topological Sort

Example:

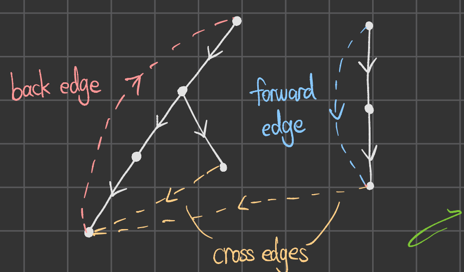


Possible Results:

b	c	a	d
b	c	d	a
c	b	d	a
c	d	b	a

One solution: choose a vertex with no in-edges and remove it ; repeat.

DFS solution: use reverse of finish order



finish order: u v y x r z w s

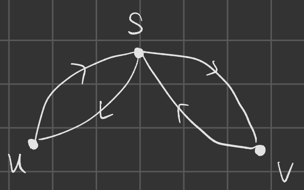
reverse: s w z r x y v u

Proof: For every directed edge (u, v), $\text{finish}(u) > \text{finish}(v)$

tree edge & forward edge → Case 1: u discovered before v. Then because of $(u,v) \in E$, v is discovered and finished before u is finished.

cross edge → Case 2: v discovered before u. G has no directed cycle we cannot reach u in DFS(v).
so v finish before u is discovered and finished.

Strongly Connected



$u \to v$: $u \to s \to v$

$v \to u$: $v \to s \to u$

Finding Strongly Connected Components in a Directed Graph

Don't need to test all pairs.

Run DFS $\Rightarrow$ finish order $f_1, \dots, f_n$ the root

Reverse edges of G, call it H.

Run DFS again with vertex order $f_n, \dots, f_1$

Lemma: Trees in the 2nd DFS are exactly the strongly connected components.

Runtime: $O(n+m)$ 2 DFS

Proof: Vertices u, v are strongly connected iff they are in the same DFS tree in 2nd DFS

($\Rightarrow$) Suppose u is discovered first in the 2nd DFS. Then since there is a path $u \rightsquigarrow v$ in the reverse graph, v is discovered before u is finished in the 2nd DFS.

($\Leftarrow$) Suppose u, v are in the same tree. Let r be the root.

Claim: r and u are strongly connected.

r is the root of tree containing u.

$\exists$ path $r \rightsquigarrow u$ in reverse graph $\Rightarrow$ $u \rightsquigarrow r$ in original graph.

We must show $\exists$ path $r \to u$ in original graph

When we started the tree rooted at r, u was undiscovered

Why did we pick r? It has a higher finish value in first DFS than u.

We have $u \to r$ path in original.

If u is discovered before r in the 1st DFS, then r has a finish time that comes before u. Contradiction.

So, r is discovered before u and finishes later.

$\Rightarrow$ u is a descendant of r

$\Rightarrow$ $\exists$ a path $r \to u$ in original graph.

Minimum Spanning Tree (MST)

n vertices $\Rightarrow$ $n-1$ edges in MST

Kruskal's Algorithm:

Correctness:

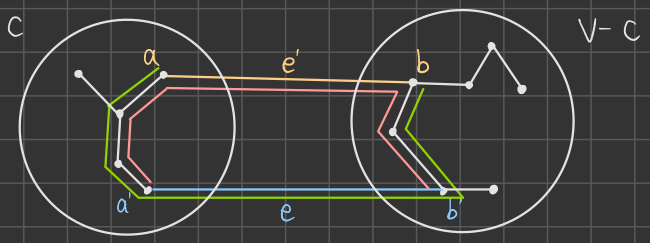
Base case: $i=0$ trivially true

Assume by induction there is a MST matching T on the first $i-1$ edges

Greedy T: $t_1, \dots, t_{i-1}, t_i, \dots, t_{n-1}$

MST M: $m_1, \dots, m_{i-1}, m_i, \dots, m_{n-1}$

Let $t_i = e(a,b)$ and let C be the connected component of T containing a



When the algorithm considers all edges of weight $< w(t_i)$ have been considered.

None of them go between C and $V-C$

Look at red path in M from a to b it must cross from C to $V-C$ (on an edge e')

Then $w(e) \le w(e')$ by ordering.

Exchange: Let $M' = (M - \{e'\}) \cup \{e\}$

Claim: M' is a MST since M' now matches T on i edges.

Proof: 1. M' is a spanning tree

2. $w(M') = w(M) - w(e') + w(e) \le w(M) \Rightarrow M'$ is MST, contradiction.

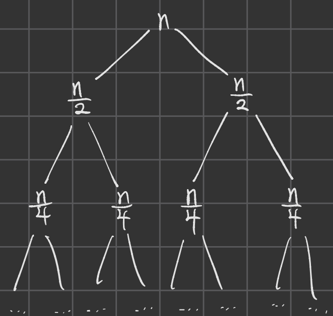
Union- Find

$O(m\log m) = O(m \log n^2) = 2O(m\log n) = O(m\log n)$

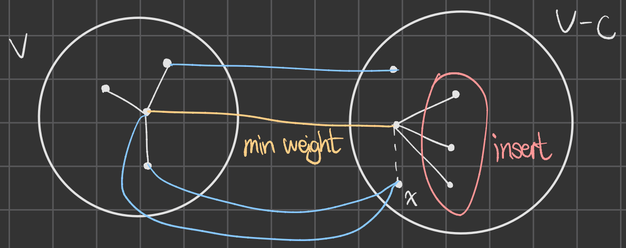
Kruskal's Runtime: $\underbrace{O(m\log n)}_{\text{sort}} + \underbrace{O(m)}_{\text{find}} + \underbrace{O(n \cdot \text{merge cost})}_{n\cdot\log n} \in O(m\log n)$

Update S: $O(\text{size of set})$

- always change smaller set
- new set is at least twice size of the smaller (at least doubling)
- can only do this at most $\log n$ times.



Prim's Algorithm



Build a single connected component T (eventually MST) by choosing a vertex v not in T that has a minimum weight edge (u, v) where $u \in T$

Implementation.

Need to find a vertex in V-C connected to C using a minimum weight edge.
For $v \in V-C$ define

$$\boxed{\text{weight}(v) = \begin{cases} \infty & \text{if no } (u,v) \text{ edge. } u \in C \\ \min\{w(e) \mid e = (u,v)\},\ u \in C & \text{otherwise} \end{cases}}$$

Priority Queue

Maintain a set V-C as an array in heap order according to the weight.
ExtractMin: remove and return vertex with min weight.
Insert(v, weight(v)): insert vertex with weight(v)
Delete(v): delete v from PQ

Go through Adj[v]
- Find endpoints in C where (u,v) = weight(u). $u \in C$.
- Insert new vertices in PQ
- Not in PQ yet, but will now be
- endpoints in C: w(u, v) = weight(v), choose 1 to be the tree edge
  - ignore, any others in C
  - $x \notin C$, not in PQ - add in PQ
  - $x \notin C$, but in PQ - check to update weight(x)
- Need a data structure to store where v is in PQ so that finding it takes O(1).

Create array $\bar{C}[1 \ldots n]$

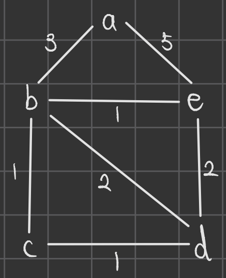
$$\bar{C}[v] = \begin{cases} -1 & \text{if } v \notin V-C \\ \text{"location" of } v \text{ in heap} & \text{otherwise} \end{cases}$$

Analysis

- 1 ExtractMin to add each v to C
- Scan Adj[v] to find e = (u, v) with w(e) = weight(e), add to MST.
- Need to update / reduce weight of vertices v'. s.t. (v, v') with $v' \in V-C$.
- Size of heap O(n)
  - n-1 ExtractMin $n \cdot O(\log n)$
  - O(m) reduce weight — delete and insert
  - Total $O(m \log n)$

Shortest Paths in Edge Weighted Graphs

Does MST always work?



Dijkstra (1959)

Input: graph or digraph $G \in (V, E)$, $w: E \to \mathbb{R}_{\geq 0}$, $s \in V$.
Output: shortest paths from s to every other vertex v.
Idea: grow tree of shortest paths starting from s.

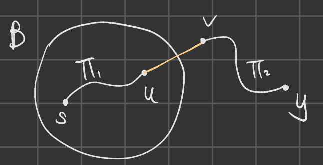
General Step:

We have a tree of shortest paths to all vertices in set B
Initially $B = \{s\}$.
Choose edge (x, y), $x \in B$, $y \notin B$ to minimize $d(s,x) + w(x,y)$, where $d(s,x)$ is known minimum distance from s to x
Call this minimum distance d. $d(s,y) \leftarrow d$
Add (x,y) to tree $\quad$ parent$(y) = x$
Greedy (in a sense): always add vertex with min weight distance from S.

Claim: d is the min distance from s to y

Proof

Any path π from s to y consists of



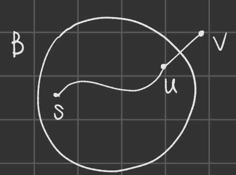
π_1: initial part of π in B $(s \rightsquigarrow u)$
$e = (u,v) \in E$, $u \in B$, $v \notin B$ (first edge leaving B)
π_2: rest of path

$$w(\pi) \geq w(\pi_1) + w(u,v) \geq \underbrace{d(s,u) + w(u,v)}_{\text{we chose the minimum}} \geq d = d(s,y)$$

Proof breaks if negative weight edges exist.
By induction on $|B|$ the algorithm correctly finds $d(s,v)$ for all $v \in V$.

Implementation

Keep "tentative" distance $d(v)$ $\forall v \notin B$,
$d(v)$ = min weight path from s to v with all but 1 edge in B.
PQ/heaps: similar to Prim's
Store d-values in a heap of size $\leq n$.
Modify a d-value: $O(\log n)$ to adjust heap



Runtime: $\underbrace{O(n\log n)}_{\text{ExtractMin}} + \underbrace{O(m\log n)}_{\text{Adjust Heap}} \in O(m\log n)$

Single Source Shortest Paths: Bellman-Ford

The original application of dynamic programming
Edge weights may be negative but no negative weight cycle is allowed.

$d_i(v)$ = weight of the shortest path from s to v using $\leq i$ edges.

$$d_1(v) = \begin{cases} 0 & v = s \\ w(s,v) & (s,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

can be merged ?

$$d_{i-1}(v) = \min \begin{cases} d_{i-1}(v) & \text{using at most } i-1 \text{ edges} \\ \min\limits_{u \in V} \{ d_{i-1}(u) + w(u,v) \} & \text{using at most } i \text{ edges} \\ \infty & \text{otherwise} \end{cases}$$

similar to 0-1 knapsack ?

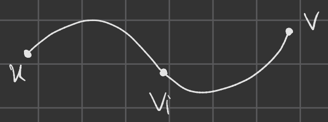
We want $d_{n-1}(v)$

n-1 is the max possible shortest length.

Runtime: $O(n \cdot (n+m))$

All Pairs Shortest Paths - Floyd-Warshall

step $i-1$: intermediate path may contain $v_1, \ldots, v_{i-1}$



2 choices: use v_i or not

$$D_0[u,v] = \begin{cases} 0 & \text{if } u=v \\ w(u,v) & \text{use } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

$$D_i[u,v] = \min \begin{cases} D_{i-1}[u,i] + D_{i-1}[i,v] & \text{using } v_i \\ D_{i-1}[u,v] & \text{not using } v_i \end{cases}$$

for i from 1 to n do:
    for $u \in V$ do
        for $v \in V$ do
            $D_i[u,v] \leftarrow \min\{D_{i-1}[u,v],\ D_{i-1}[u,i] + D_{i-1}[i,v]\}$

Analysis: $O(n^3)$ for both runtime and space.

Note: Can reduce the space usage to $O(n^2)$

for i from 1 to n do:
    for u from 1 to n do
        for v from 1 to n do
            $D[u,v] \leftarrow \min\{D[u,v],\ D[u,i] + D[i,v]\}$.

Exhaustive Search

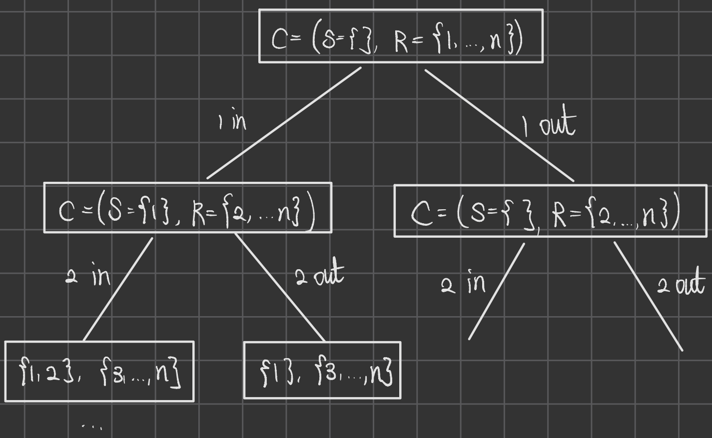
Alternative options
- approximations - often know error factor, quality of solution is based on error factor.
- heuristics - often okay, but no guarantee on quality or runtime.
- exact solution - very expensive

Example - Subset Sum

- No known polynomial time algorithm
- Explore all subsets

Backtracking to Explore all Subsets

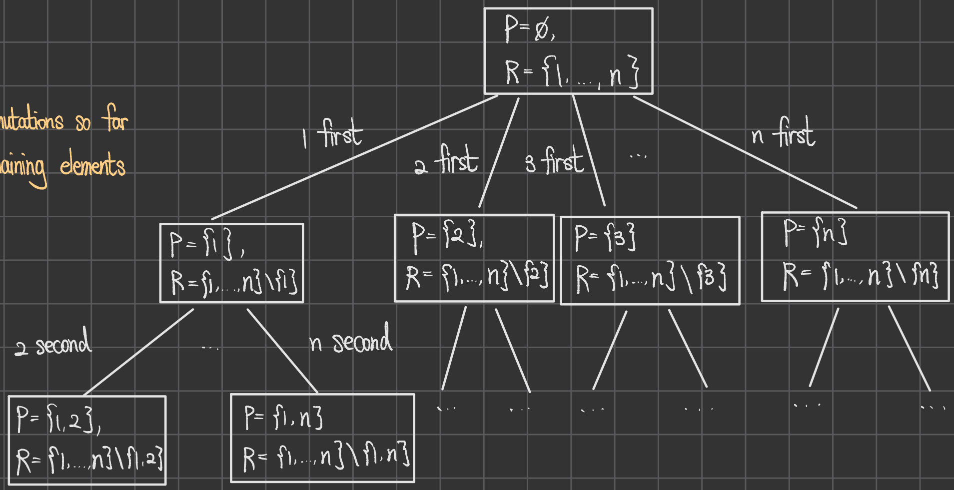
C: configuration
S: state
R: remaining items



height = n
\# nodes $\in O(2^n)$
\# leaf nodes = 2^n (when R = { }) — # subsets of {1, ..., n}

Backtracking to explore all permutations

P = permutations so far
R = remaining elements



height = n
# leaves = n! — # of permutations of $\{1,\dots,n\}$.

Maintain: $w=\sum_{i\in S} w_i$ and $r=\sum_{i\in R} w_i$

If $w=W$, problem is solved
If $w>W$, we have a dead-end.
If $r+w<W$, we have a dead-end.

Runtime: $O(2^n)$, better than DP if $W=2^n$
$O(nW)$

Backtracking on Graphs

$C=(N,X)$ — edges to exclude.
edges to include

$N\cap X=\emptyset$

General Algorithm

Let A be the set of active configurations.

while $A\neq\emptyset$ do
  $C\leftarrow$ remove from A
  Explore C
  if C solves the problem then we are done.
  if C is a dead-end then discard it
  else expand C to child configurations $C_1,\dots,C_t$ (by making choices)
    $A\leftarrow A\cup\{C_i\}$

Branch-and-Bound

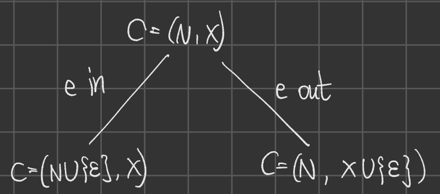
```
Let A be the set of active configurations
Initially A starts with a single configuration
Best-cost ← ∞
while A ≠ ∅ do
    C ← remove "most promising" configuration from A.
    Expand C to C1, ..., Ct (by making additional choices)        // Branch
    for i from 1 to t do.
        if Ci solves the problem then
            Best-cost ← min (Best-cost, cost(Ci))
        else if Ci is a deadend then discard Ci
        else if lower-bound(Ci) < Best-cost then add Ci to A      // Bound
```

Branch-and-Bound for TSP

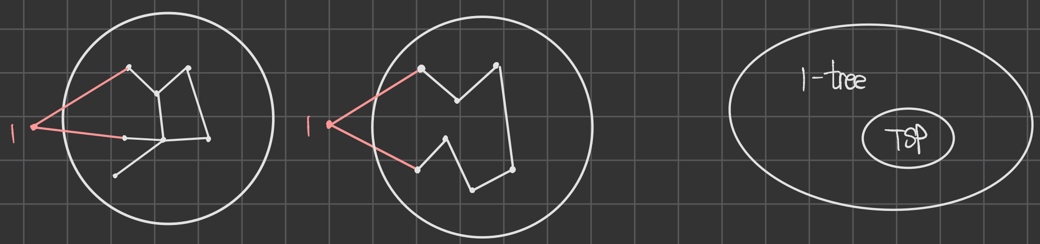
Necessary Conditions (used to detect dead-ends)

- $E - X$ is biconnected
- N has ≤ 2 edges incident to each vertex
- N contains no cycle (except on all vertices)

Branch



Claim: Any TSP tour is a 1-tree.



If the min-weight 1-tree is a TSP, then the TSP tour is optimal.

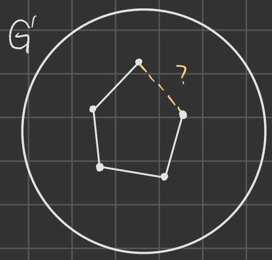
Given a configuration (N, X), we can efficiently compute a minimum weight 1-tree that includes edges in N and discarding the edges in X.
assigning a weight of 0

Find MST on vertices $2, \ldots, n$. Add the two minimum weighted edges incident to vertex 1.
Then compute the weight of the 1-tree (sum real weights)

Hamilton Path / Cycle

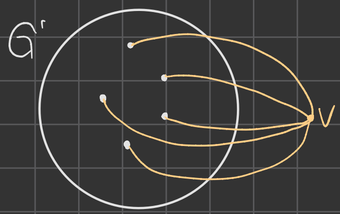
Construct G' s.t. G has a Hamilton Path iff G' has Hamilton Cycle

Idea 1: add one edge to get G'; don't know which edge



We will need to test between every pair of vertices. not a many-one reduction

Idea 2: add one new vertex adjacent to all vertices in G.



Runtime: $n+1$ vertices, $m+n$ edges $\Rightarrow$ $2n+m+1$
G' is linear in size compared to G

Correctness
G has a Hamiltonian Path iff G' has a Hamiltonian Cycle

Proof:
($\Rightarrow$) Suppose G has a Hamiltonian Path $u_1 \dots u_n$
Then G' has a Hamiltonian Cycle $v u_1 \dots u_n v$
($\Leftarrow$) Suppose G has a Hamiltonian Cycle.
Then remove v to get a Hamiltonian Path

Note
This is a special case of reduction, called a many-one reduction.
The subroutine is called only once.

Equivalence of Optimization and Decision Problems

Maximum Independent Set

An independent set is a set of vertices s.t. no two are joined by an edge in G.
Opt: Find max IS.
Dec: Given integer k, is there an IS of size $\leq k$?

Dec $\leq_P$ Opt
- use algorithm for Opt to solve Dec.

Opt $\leq_P$ Dec
- a bit harder.

Example - IS
- Find the max k_{OPT} by testing $k = 1, 2, \ldots, n$ using Dec.
- Then find the set with size k_{OPT}.
- Delete vertex one at a time
- If $\text{Max-IS}(G - v) = k_{OPT}$, then $G \leftarrow G - v$.
- Repeat until no vertex can be deleted

Runtime: polynomial (assuming algorithm for Dec takes polynomial time.)

Certificate/ Verification

For decision problems, not optimization problems.

TSP (Decision) $\in$ NP

Theorem: TSP is in NP.

Certificate: a permutation of n vertices
Verification:
1. check if it is a permutation of the nodes
2. check edges exist in G
3. check edges form a cycle.
4. sum of weights $\leq k$.

Verifiers need to be in polynomial time.

$X \leq_P Y$

- X reduces to Y
- "X is easier than Y", reducing to harder problem.
- Suppose we have a polynomial time subroutine for Y, create a polynomial time algorithm for X.

3-SAT $\leq_P$ Independent Set

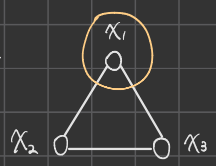
Theorem. Independent Set is in NP

Reduction

Construction $(x_1 \vee x_2 \vee x_3)$

One of x_1, x_2 and x_3 must be true

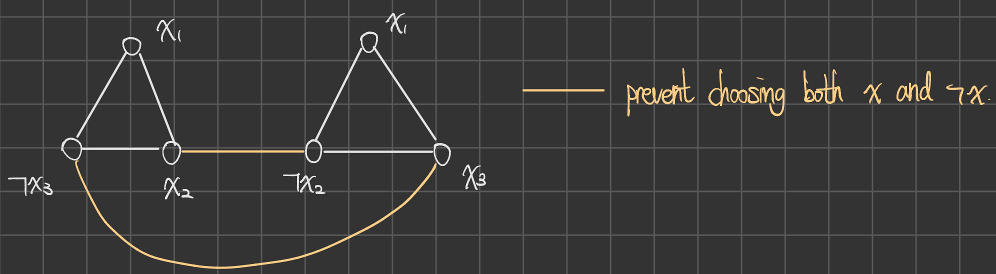
Graph G:



Choose 1 vertex for the Independent Set

Example

$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$



In general, to prove a problem Z is NP-complete

1. Prove Z in NP.
2. Prove $X \leq_P Z$ for some known NP-complete problem X.

since we knew X is NP-complete,
we don't have a polytime algorithm for it,
so we can construct a contradiction.

Clique

A clique is a subset V' of V where every pair of vertices is joined by an edge. a complete subgraph.
Input: an undirected subgraph $G = (V, E)$ and an integer k
Output: Does G have a clique of size $\geq k$?

Theorem: Clique is in NP.

Certificate: a set of vertices C do not use "a set of vertices that forms a clique; needs to be general"
Verifier:
1. each v is a valid vertex in G.
2. edges between all pairs of C exist in G.
3. $|C| \geq k$

Theorem: Clique is NP-complete

Proof: 1. Clique is in NP
2. [A known NP-complete problem] $\leq_p$ Clique
Independent Set

Reduction
Suppose we have a polytime algorithm for Clique, give a polytime algorithm for Independent Set

G has a clique of size $\geq k$ iff G^c has an IS of size $\geq k$ *
return clique($G' = G^c$, $k' = k$)
completement graph.

Runtime: Polynomial.
Correctness: Known property *

Proof of *
($\Rightarrow$) Let C be a clique in G with $|C| \geq k$, then since C is complete
C^c (the subgraph in G^c with the same vertices) has no edge between any two vertices.
($\Leftarrow$) Let C^c be an IS in G^c with $|C| \geq k$, then since C has no edge between any pair of vertices,
C has an edge for every pair of vertices in G, so C is a clique in G.

Vertex Cover

A vertex cover is a set $V' \subseteq V$ such that every edge $(u,v) \in E$ has $u \in V'$ or $v \in V'$ (or both)

Theorem: Vertex Cover is NP-complete

Proof: 1. VC is in NP
2. [A known NP-complete problem] $\le_p$ VC
Independent Set

Reduction

Suppose there is a polytime algorithm for VC, give a polytime algorithm for IS.
Construct $G' = G$, $k' = n-k$.

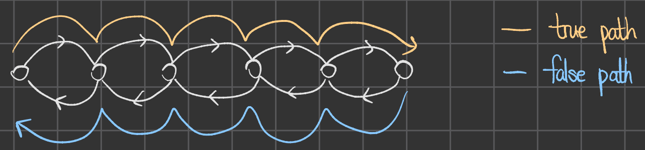
Runtime: polynomial
Correctness: G has a VC with size $\le k'$ iff G' has an IS of size $\ge k$
($\Rightarrow$) Let V' be a VC with size $\le k' = n-k$. Then $G - V'$ is an IS of size $\ge k$.
Why? Each $v \in G - V'$ is not in V' $\Rightarrow$ no edge between them.
($\Leftarrow$) Let V' be an IS with size $\ge k$. Then $G - V'$ is a VC of size $\le k'$.
Why? Let $(u,v) \in E$ be any edge, then either $u \in V'$ and $v \in G - V'$ or both $u, v \in G - V'$
Hence $G - V'$ is a VC.

3-SAT $\leq_p$ Directional Hamiltonian Cycle

Input: Boolean formula F clauses $C_1, \ldots, C_m$ on variable $v_1, \ldots, v_n$
Output: Is there an assignment that satisfies F?

Reduction

Construct directed graph G such that F is satisfiable iff G has a directed Hamilton Cycle
Idea: For each variable x_i, there is a part of G ("variable gadget") that chooses whether x_i is true or false.

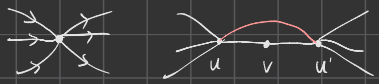
x_i 

We need a gadget like this for all variables.

Directed Hamiltonian Cycle $\leq_p$ Hamiltonian Cycle

Input: directed graph.

Suppose we have a polytime alg for (undirected) Ham Cycle.
$\Rightarrow$ need to convert graph G into undirected graph G'.



Why is v required?

$3\text{-SAT} \le_p$ Subset Sum

Subset Sum
- items $1 \dots n$
- weights $w_1, \dots, w_n$
- target W

Reduction:

Encode the info of boolean formula F into the "bits" of the numbers we use for weights.

	x_1	x_2	...	x_n	C_1	C_2	...	C_m
x_1	1				0			
$\neg x_1$	1				$\boxed{1}$			
x_2		1			$\boxed{1}$			
$\neg x_2$		1			0			
$\vdots$					$\vdots$			
x_n				1	0			
$\neg x_n$				1	$\boxed{1}$			
S_1					1			
S_1'					2			
S_2						1		
S_2'						2		
$\vdots$								
S_m								1
S_m'								2

encode clause info here

Ex. $C_1 = (\neg x_1 \vee x_2 \vee \neg x_n)$

Subset:
- weights: interpret each row as a number.
- choose a subset of weights
  - choosing rows
- use base-10 to avoid carry-over

Target:

both: $1+3$ or S_i': $2+2$ or S_i: $3+1$

(satisfied + slack)

$\boxed{\underbrace{1\dots1}_{n}}\ \boxed{\underbrace{4\dots4}_{m}}$

can only choose one between x_i and $\neg x_i$, for $i = 1, \dots, n$

- Purpose of slack: target cannot be variable, need to be fixed

Approximation Algorithms

- Heuristics - there might be no guarantee on the runtime or the quality of the solution
- Approximation algorithms
  - near optimal solution
  - polynomial time and a guarantee on the quality of the solution
  - for a minimization problem, might guarantee
- The cost of the approximate solution: C
- The cost of the optimal solution: C^*
- Approximation ratio of an approximate algorithm: $\rho(n)$.
  - maximization problem: $C^* \leq \rho(n)\, C$
  - minization problem: $C \leq \rho(n)\, C^*$
  - In both cases, $\rho(n) \geq 1$

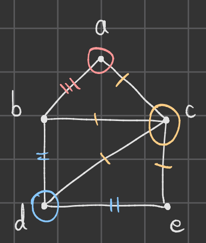
Vertex Cover

Optimization problem: find a minimum size vertex cover

Greedy Algorithm 1

$C \leftarrow \emptyset$
$E' \leftarrow G.E$
while $E' \neq \emptyset$
    $v \leftarrow$ vertex with maximum degree
    $C \leftarrow C \cup \{v\}$
    $E' \leftarrow E' - \{$edges covered by $v'\}$
return C.

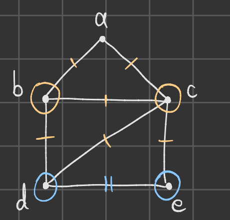
Runtime: polynomial
Approximation factor: $\Theta(\log n)$ not constant



$C = \{c, d, e\}$.

Greedy Algorithm 2

APPROX - VERTEX - COVER
$C = \emptyset$
$E' = G.E$
while $E' \neq 0$:
  Let $(u,v) \in E$ be any arbitrary edge of E'
  $C \leftarrow C \cup \{u,v\}$
  $E' \leftarrow C \setminus \{$every edge incident to either u or $v\}$
return C



$A = \{(b,c), (d,e)\}$
$C = \{b, c, d, e\}$

Runtime: polynomial
Approximation factor: 2

Theorem: APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

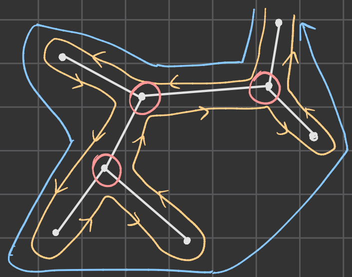
Proof: Let A denote the set of edges chosen by the greedy algorithm.
In order to cover the edges in A, any vertex cover, including optimal cover C^*, must include at least one endpoint of each edge in A.
We have $|C^*| \geq |A|$ on the size of an optimal vertex cover.
Each iteration picks an edge for which neither of its endpoint is already in C, so we have $|C| = 2|A|$
Therefore, we have $|C^*| \geq |A| \Rightarrow 2|C^*| \geq 2|A| = |C| \Rightarrow |C| \leq 2|C^*|$

Note:
not a proper subset of any other matching
A is a maximal matching of G.
$|A|$ gives a lower bound on the size of a vertex cover.

TSP Approximation

Euclidean TSP

- For complete graphs on points in the plane
- Weight is the Euclidean distance between the two endpoints.



— MST walk

— shortcut (TSP tour)

• skipped

① Fin

② Take a tour of MST

- avoid repeating vertices

③ Take shortcuts ⇒ TSP tour

- shortcuts are shorter (triangle inequality)

$|V|$ # edges in MST

$O(n + n - 1) = O(n)$

the weights are Euclidean distances

APPROX-TSP-Tour (G, c)

1. Select a vertex $r \in G.V$ to be a "root" vertex
2. Compute a MST T for G from root r using MST-Prim (G, c, r)
3. Let H be a list of vertices, ordered according to when they are first visited.
4. return H

Runtime: Even with a simple implementation of MST-Prim, the running time of APPROX-TSP-TOUR is $\Theta(n^2)$

Theorem: APPROX-TSP-TOUR is a polynomial 2-approximation algorithm for the TSP problem with triangle inequality.

Proof: Let H be the hamiltonian cycle. Let H^* be the optimal hamiltonian cycle.
Let T be any MST
We have $w(H^*) > w(T)$, since if we remove an edge from H, we get an MST.
We also have $w(H) \leq 2w(T)$ since we might skip some vertices during the MST walk.
The total weight is reduced if we do so because of the triangle inequality.
Now, we have $2w(T) < 2w(H)^*$ and so $w(H) \leq 2w(H^*)$

The general TSP

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation factor ρ.

Show a Decision Problem is in NP

- Define a certificate C - polynomial size
- Define a verifier (I, C)
  - polynomial time in terms of # bits in input.
  - give an algorithm for each step.

In terms of the bits

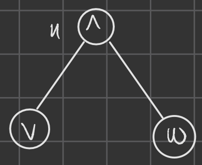
Given 2 numbers n and m.
- n is represented using $\log n$ bits
- m is represented using $\log m$ bits

Addition: $O(\max\{\log n, \log m\})$ polytime
Multiplication: $O(\log n \cdot \log m)$

0-1 Knapsack
- items $1 \ldots n$
- weights $w_1, \ldots, w_n$
- values $v_1, \ldots, v_n$
- Capacity W ~ need $\log W$ bits
- $W = 2^{\log_2 W}$ exponential in the # of bits
- pseudopolynomial

Circuit-SAT $\leq_P$ 3-CNF-SAT

Intuitively circuits and Boolean formulas are the same.
Not polynomial - formula "doubles" in size as we go up a level.



$x_u \equiv x_v \wedge x_w$
(convert each node to a variable)

Reduction

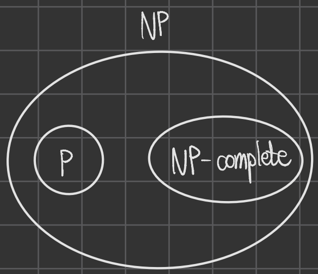
$(a \wedge b) \vee (\neg a \wedge \neg b)$

$a \equiv b$ means $(\neg a \vee b) \wedge (a \vee \neg b)$

$(\neg x_u \vee x_v \wedge x_w) \equiv (\neg x_u \vee x_v) \wedge (\neg x_u \vee x_w)$

$(x_u \vee \neg(x_v \wedge x_w)) \equiv (x_u \vee \neg x_v) \wedge (x_u \vee x_w)$

Convert clauses of 2 variables into 3 variables:
$(a \vee b) \equiv (a \vee b \vee x_{new}) \wedge (a \vee b \vee \neg x_{new})$



$X \leq_p Y \leq_p Z$

NP first NPC new NPC

SubsetSum $\leq_p$ 0-1 Knapsack

known NPC show this is NPC

Assume we have a polytime algorithm for 0-1 knapsack.
Give a polytime algorithm for Subset Sum.

Reduction

0-1 Knapsack	Subset Sum
items $1, \dots, n$	items $1, \dots, n$
$w_1, \dots, w_n$	$w_1, \dots, w_n$
$v_1, \dots, v_n$	$w_1, \dots, w_n$
W	W
k	W

Runtime: instance of 0-1 Knapsack is 2 x size of instance of Subset Sum

First NP-Complete Problem

1st NPC

$\forall Y \in NP, Y \leq_p X$. Then $X <_p Z$, where Z is a new NPC problem.

Circuit Satisfiability

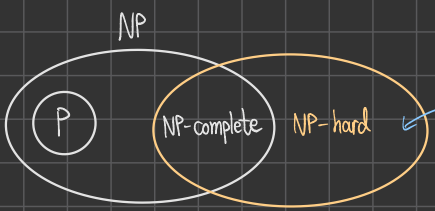
A circuit C

- sources/inputs (variables or 0,1) - no incoming edges.
- one sink / output - no outgoing edges.
- internal nodes are $\wedge, \vee, \neg$

Theorem: $\forall Y \in NP, Y \leq_p$ Circuit-SAT

For every Y in NP, there is an algorithm that maps any input y to a circuit C

Decidability



include optimization problems.

$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq$ Decidable

polynomial space   runtime $O(2^{n^k})$

Known: $P \neq EXP$

Unknown: Everything else.