* This set of notes follows several resources, including Prof. Lap Chi's (Waterloo CS341) and Prof. Halim's (NUS CS3230) notes.

We start by considering two problems: the traveling salesman problem and the Chinese postman problem.

Say we have an undirected graph on n vertices and m edges, where each edge has some non-negative cost.

Traveling salesman: find minimum cost tour visiting every vertex at least once. ⇝ naive: $O(n!)$ → DP $O(2^n)$ (NP-complete)

Chinese postman: find minimum cost tour visiting every edge at least once. ⇝ surprisingly can be solved in $O(n^4)$ (polynomial) time.

Goal:
- know well-known algorithms well ⇝ solve hard problems in polynomial time
- design new algorithms
- prove correctness and time complexity
- use reductions to solve problems and prove hardness

Note on recursion: Many modern languages have recursion depth limits due to memory and stack unwinding constraints.

We cover Master Theorem (and recursion trees), (D&C, randomized, graph, greedy) algorithms, DP, amortized analysis, and NP-hardness.

But how do we define time complexity? What operations count as primitive?

Computation Models

We generally assume the word-RAM model: we can access an arbitrary position (word) of an array (in RAM) in constant time, and that word-operations (addition, read/write, etc.) are constant-time.

Remark: Suppose we have a graph with n vertices. We would need $\log_2 n$ bits to identify each vertex ⇝ we assume this fits in a word and assume primitive operations (e.g. ID comparisons) are $\Theta(1)$.

Sometimes this model is unrealistic → we would analyse bit-complexity.

3SUM: Given numbers $a_1,\dots,a_n$ and c, we want to determine if there are i,j,k such that $a_i+a_j+a_k=c$.

Naive: enumerate all triples and check if sum is c. $\Theta(n^3)$

Algo.2: Write $a_i+a_j+a_k=c \iff c-a_i-a_j=a_k$. Enumerate all pairs a_i, a_j ($\Theta(n^2)$) and check if $c-a_i-a_j=a_k$ for some k. (sort + run binary search)

1) Sort array $A=a_1\le\dots\le a_n$. $O(n\log n)$

2) Enumerate a_i, a_j and binary search for $a_k=c-a_i-a_j$.

$\Rightarrow O(n\log n + n^2\log n) = O(n^2\log n)$.

Algo.3: Reduce to a simpler problem. Notice $a_i+a_j+a_k=c \iff a_i+a_j=c-a_k$.

For fixed k, this is a 2SUM problem ⇒ we reduce to n 2SUM problems. (one for each k)

reduce to 2SUM

1) Sort $A=a_1\le\dots\le a_n$. $O(n\log n)$

2) Enumerate a_k and solve the 2SUM problem $a_i+a_j=c-a_k=:b$. $*\ O(n)$

$\Rightarrow O(n^2)$

We claim that 2SUM on a sorted array can be solved in $O(n)$.

Double-pointer approach. Keep a left-index L and right-index R.

While $L\le R$:
- if $a_L+a_R=b$, we are done
- if $a_L+a_R>b$, R--
- if $a_L+a_R<b$, L++

Time complexity: (Prof.) Stops when $L>R$, and L and R get closer every iteration ⇒ at most n iterations ⇒ $O(n)$.

Correctness: If $\exists i,j$ such that $a_i+a_j=b$, we won't find any i,j.

Proof.

Say $\exists i,j$ such that $a_i+a_j=b$, for some $i \leq j$.

Algo. runs until $L=R \Rightarrow$ there is an iteration such that $L=i$ or $R=j$. → at some point, one of these must be true.

WLOG suppose $L=i$ occurs first, and so $R>j$.

As long as $R>j$, we have $a_i+a_R>b$ (since A sorted) $\Rightarrow$ we decrement R

$\Rightarrow$ eventually we will have $R=j$.

General k-SUM Algorithms Exercise (see CS341 material)

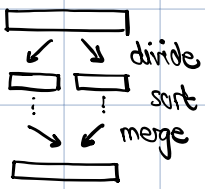
Whether $O(n^2)$ was optimal for 3SUM has been a long-standing open problem.

Recently, there was an $O\left(\frac{n^2}{(\frac{\log n}{\log\log n})^{2/3}}\right)$ algorithm discovered, but it remains to be proven if there is an $O(n^{2-\varepsilon})$ time algorithm.

Solving Recurrences

reduce problem to smaller instance of the same problem

Merge-Sort Classic divide-and-conquer sorting algorithm.




```
merge-sort:
  sort(A[1..n]):
    if n=1: return
    sort(A[1, ⌈n/2⌉])
    sort(A[⌈n/2⌉+1, n])
    merge(A[1, ⌈n/2⌉], A[⌈n/2⌉+1, n])
```

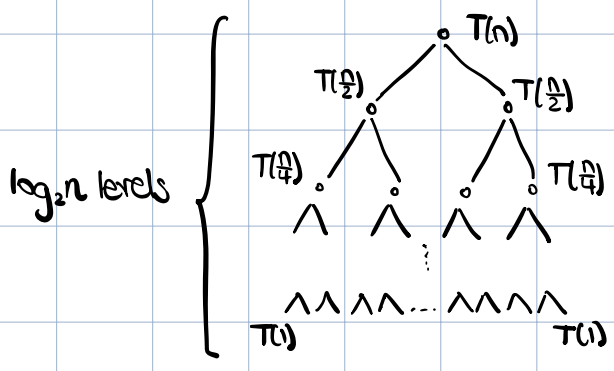
We have a recurrence

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

Assume $n=2^k$ for some k.

$\Rightarrow T(n) = 2T(\frac{n}{2}) + O(n)$ sloppy recurrence.

One way to solve the recurrence is to draw the recursion tree.



cn operations

$2(\frac{cn}{2})$ operations

$4(\frac{cn}{4})$ operations

$\vdots$

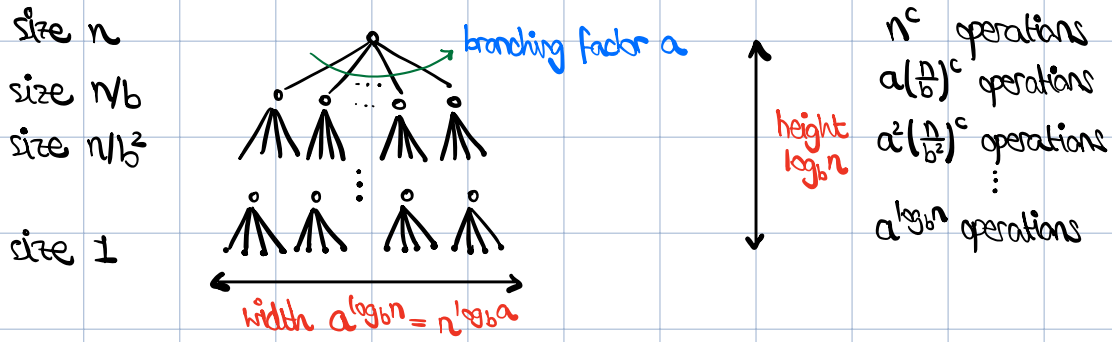
$n(\frac{cn}{n})$ operations

$\Rightarrow n$ operations for $\log_2 n$ levels $\approx O(n\log n)$ as expected

We can also solve this inductively.

Lets consider a more general setting.

Consider the relation $T(n) = aT(\frac{n}{b}) + n^c$ for some $a>0$, $b>1$, $c \geq 0$ constants.



We want to solve for $\sum_{i=0}^{\log_b n} \underbrace{a^i(\frac{n}{b^i})^c}_{\text{operations in step } i}$ } sum over all steps

$= \sum_{i=0}^{\log_b n} (\frac{a}{b^c})^i n^c$ a geometric series.

Case 1 $(\frac{a}{b^c})=1$: we get $\sum_{i=0}^{\log_b n} n^c = n^c(\log_b n + 1) \in \Theta(n^c \log_b n)$

Case 2 $(\frac{a}{b^c})<1$: decreasing geometric series dominated by the first term $\Rightarrow \Theta(n^c)$

Case 3 $(\frac{a}{b^c})>1$: increasing sequence dominated by last term $\Rightarrow \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$

Theorem (Master): If $T(n) = aT(\frac{n}{b}) + n^c$ for constants $a > 0$, $b > 1$, $c \geq 0$, then:

$$T(n) \in \begin{cases} O(n^c) & \text{if } c > \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^{\log_b a}) & \text{if } c < \log_b a \end{cases}$$

Remark: Method more important than result.

Examples: 1 Single subproblem.

$T(n) = T(\frac{n}{2}) + 1$, we have $T(n) \in O(\log n)$ (binary search)

$T(n) = T(\frac{n}{2}) + n$, we have $T(n) \in O(n)$ geometric sequence (quick-select)

$T(n) = T(\sqrt{n}) + 1$, we have $T(n) \in O(\log \log n)$ (counting levels)

2 Non-even subproblems.

$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + n \in O(n \log n)$

$T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n \in O(n)$

3 Exponential.

$T(n) = 2T(n-1) + 1 \in O(2^n)$

4 Fibonacci. Can we improve the runtime of $T(n) = T(n-1) + T(n-2) + 1$?

Consider recurrence $t_n = t_{n-1} + t_{n-2} \Rightarrow t_n - t_{n-1} - t_{n-2} = 0$. We use MATH 239 to create a closed-form expression to show that $t_n = (\frac{1+\sqrt{5}}{2})^n$ so $T(n) \in O((\frac{1+\sqrt{5}}{2})^n) \approx O(1.618^n)$.

This is faster than $O(2^n)$ standard exponential time.

These sorts of recurrences appear often when analysing algorithms (especially faster-exponential time algorithms).

Consider the maximum independent set problem: given a graph $G = (V, E)$, we want to find a maximum subset $S \subseteq V$ such that $u, v \in S \Rightarrow uv \notin E$.

Naive: enumerate all subsets $S \subseteq V$ to form power set $P(V)$, and check each $X \in P(V)$

$\Rightarrow \Omega(2^{|V|})$ time.

Variant: Pick $v \in V$ with maximum degree. Two possibilities: $v \in S$ or $v \notin S$.

$v \in S$: Reduce to $G - v$ (delete v) $\rightsquigarrow$ reduce graph size by 1.

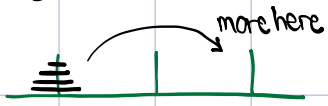
$v \notin S$: Then neighbours of v cannot be in S $\rightsquigarrow$ delete v and all its neighbours $\rightsquigarrow$ reduce graph size by ≥ 2.

$\Rightarrow T(n) \leq T(n-1) + T(n-2) + O(n) \rightsquigarrow T(n) \in O(n2^n)$.

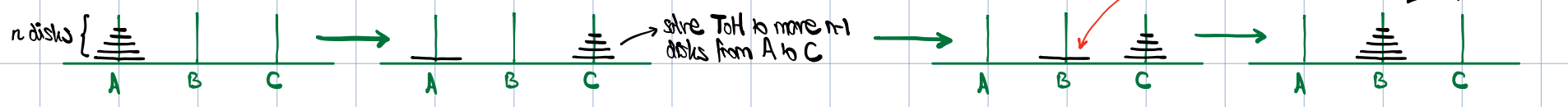
In general, we consider 4 methods for solving recurrences:

telescoping series, substitution, recursion trees, and master theorem.

Substitution: Towes of Hanoi. • only one disk may be moved at a time
• no disk may be placed on a smaller disk



Notice a ToH on n disks can be reduced to a ToH on $n-1$ disks:



runtime
$\Rightarrow T(n) = T(n-1) + T(n-1) + 1 = 2T(n-1)+1$ with algorithm ToH(n, A, B, C) (from to aux):

ToH(n-1, A, C, B)
move A[n] to B
ToH(n-1, C, B, A)

Intuitively, we can guess that $T(n) \in \Theta(2^n)$:

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

$= 2(2T(n-2)+1)+1 = 4T(n-2)+(2+1)$

$= \dots = 2^n T(n-n) + \sum_{i=0}^{n-1} 2^i = 2^n + \frac{2^n-1}{2-1} \in \Theta(2^n)$. We would then proceed by induction proof.

Consider $T(n) = aT(\frac{n}{b}) + f(n)$ where $f(n)$ is any generic function. We have three possible cases:

1. $f(n) \in O(n^{d-\varepsilon})$ for some $\varepsilon > 0 \Rightarrow T(n) \in \Theta(n^d)$
2. $f(n) \in \Omega(n^{d+\varepsilon})$ for some $\varepsilon > 0$ (and $af(\frac{n}{b}) \le cf(n)$ for some $c<1$ ⇝ regularity condition) $\Rightarrow T(n) \in \Theta(f(n))$
3. $f(n) \in \Theta(n^d \log^k n)$ for some $k \ge 0$

} where $d = \log_b a$

Proof (1): $T(n) = aT(\frac{n}{b}) + f(n)$. Let $n = b^m$ so $T(b^m) = aT(b^{m-1}) + f(n) = a^m T(b^{m-m}) + \sum_{i=0}^{m-1} a^i f(\frac{n}{b^i})$.

We have $\underbrace{a^m T(b^{m-m})}_{a^{\log_b n} T(1)} + \underbrace{\sum_{i=0}^{m-1} a^i f(\frac{n}{b^i})}_{a^i O((\frac{n}{b^i})^{d-\varepsilon})} \le a^{\log_b n} T(1) + \sum_{i=0}^{m-1} \underbrace{a^i c (\frac{n}{b^i})^{d-\varepsilon}}_{(\frac{a}{b^{d-\varepsilon}})^i c n^{d-\varepsilon}}$

$= n^{\log_b a} T(1) + \underbrace{cn^{d-\varepsilon} \sum_{i=0}^{m-1} (\frac{a}{b^{d-\varepsilon}})^i}_{cn^{d-\varepsilon} \sum_{i=0}^{m-1} (\frac{ab^\varepsilon}{b^d})^i = cn^{d-\varepsilon} \sum_{i=0}^{m-1} (\frac{ab^\varepsilon}{b^{\log_b a}})^i}$

$= cn^{d-\varepsilon} \sum_{i=0}^{m-1} (b^\varepsilon)^i$ ($a = b^{\log_b a}$)

$= cn^{d-\varepsilon} (\frac{b^{\varepsilon m}-1}{b^\varepsilon - 1})$ ($b^{\varepsilon m} = b^{\varepsilon \log_b n} = n^\varepsilon$)

$= Cn^{d-\varepsilon} n^\varepsilon = Cn^d$

$= n^d T(1) + Cn^d \in \Theta(n^d)$. (constant) Δ

how does this prove a Θ bound? consider the other case of removing rightmost sum term.

Clearly this method is tedious, so we will again use the recurrence tree proof.

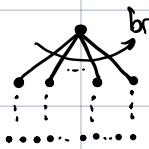
Theorem (Master): Let $T(n) = aT(\frac{n}{b}) + f(n)$ be a recurrence where $a>0$, $b>1$, ($T(1) \in O(1)$,) and $f(n)$ is any function.

1 (left-dominated) $f(n) \in O(n^{d-\varepsilon})$ for some $\varepsilon > 0 \Rightarrow T(n) \in \Theta(n^d)$

2 (root-dominated) $f(n) \in \Omega(n^{d+\varepsilon})$ for some $\varepsilon > 0$ and $af(\frac{n}{b}) \le cf(n)$ (for all n sufficiently large) for some $c<1$ $\Rightarrow T(n) \in \Theta(f(n))$ (regularity condition)

3 (balanced) $f(n) \in \Theta(n^d \log^k n)$ for some $k \ge 0 \Rightarrow T(n) \in \Theta(n^d \log^{k+1} n)$

where $d = \log_b a$.

Proof: Consider the recurrence tree for $T(n)$: 

. Since $n \to \frac{n}{b}$ each level, we have height $h = \log_b n - 1$.

Notice that each node has "a" children

$\Rightarrow$ #leaves $= a^{\log_b n} = n^{\log_b a} = n^d$. Since each node has >1 child, we have #nodes $\in \Theta(n)$. (CS240)

Each node has runtime $f(\frac{n}{b^i})$ at level $i = 0, \dots, h$

$\Rightarrow$ total runtime across node operations is $\sum_{i=0}^{h} \underbrace{a^i}_{\text{\#nodes at level } h} \underbrace{f(\frac{n}{b^i})}_{\text{per-node runtime}}$.

and total runtime across leaf operations is $n^d T(1)$.

1 $f(n) \in O(n^{d+\varepsilon})$. Dominant term becomes leaf operations $\Rightarrow T(n) = n^d T(1) + \sum_{i=0}^{h} a^i f\left(\frac{n}{b^i}\right)$

$\Rightarrow T(n) \in \Theta(n^d)$. $\triangle$

$c n^d$ from above

2 $f(n) \in \Omega(n^{d+\varepsilon})$ and $af\left(\frac{n}{b}\right) \le cf(n)$. Dominant term becomes node operations.

$$n^d T(1) + \underbrace{\sum_{i=0}^{h} a^i f\left(\frac{n}{b^i}\right)}_{\text{regularity}} \le n^d T(1) + f(n) \underbrace{\sum_{i=0}^{h} c^i}_{\le \sum_{i\ge 0} c^i = \frac{1}{1-c} \text{ constant since } c<0} \le n^d T(1) + \underbrace{\left(\frac{1}{1-c}\right) f(n)}_{\text{dominant}} \in \Theta(f(n))$$

regularity: $f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \dots \le f(n) + cf(n) + c^2 f(n) + \dots$

$\Rightarrow T(n) \in O(f(n))$.

and $n^d T(1) + \sum_{i=0}^{h} a^i f\left(\frac{n}{b^i}\right) = n^d T(1) + f(n) + \sum_{i=1}^{h} a^i f\left(\frac{n}{b^i}\right) \ge n^d T(1) + \underbrace{f(n)}_{\text{dominant}} \in \Theta(f(n))$

$\Rightarrow T(n) \in \Omega(f(n))$.

Thus $T(n) \in \Theta(f(n))$. $\triangle$

3 $f(n) \in \Theta(n^d \log^k n)$. Both terms are comparable.

We have $T(n) = n^d T(1) + \sum_{i=0}^{h} a^i f\left(\frac{n}{b^i}\right) = n^d T(1) + \sum_{i=0}^{h} a^i \left(\frac{n}{b^i}\right)^d \log^k\left(\frac{n}{b^i}\right)$ — remove this for UB

$$= n^d T(1) + n^d \sum_{i=0}^{h} \left(\frac{a^i}{b^{di}}\right) \left[\log^k n - i \log^k b\right]$$

$$\le n^d T(1) + n^d \log^k n \sum_{i=0}^{h} \left(\frac{a}{b^d}\right)^i = n^d \log^k n (h+1)$$

($\frac{a}{b^d} = a$)

$n^d \log^k n (h+1) = n^d \log^k n \log_b n$

$= n^d \log^k n \frac{\log n}{\log b} \in n^d \log^{k+1} n \Rightarrow T(n) \in O(n^d \log^{k+1} n)$.

($\frac{\log n}{\log b}$: constant)

★ how to formulate a meaningful LB?

Exercise. $\triangle$

Therefore, we are done through a case-by-case analysis. $\square$

Remark: The Master Theorem does NOT cover all recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

There are functions that will not satisfy any 3 of the cases.

Ex. Let $T(n) = T\left(\frac{n}{2}\right) + 2^{\sqrt{\log n}}$ so $a=1$, $b=2$, $d = \log_b a = 0$.

Claim: $T(n) \notin O(n^{d-\varepsilon})$, $T(n) \notin \Omega(n^{d+\varepsilon})$, $T(n) \notin \Theta(n^0 \log^k n)$ for any $k \ge 0$. ★ Exercise

Remark: We can write the general form of a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ as $\underbrace{n^d T(1)}_{\text{leaf ops.}} + \underbrace{\sum_{i=0}^{h} a^i f\left(\frac{n}{b^i}\right)}_{\text{internal ops.}}$.

Exercise (AIQ2) Resolve recurrence $T(n) = 16T\left(\frac{n}{4}\right) + n^4$.

We have (from recursive algo. general form) that $T(n) = n^{\log_b a} T(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$

$$= n^2 + \underbrace{\sum_{i=0}^{\log_b n - 1} 16^i \left(\frac{n}{4^i}\right)^4}$$

$$n^4 \sum_{i=0}^{\log_b n - 1} \left(\frac{16}{4^4}\right)^i = n^4 \underbrace{\sum_{i=0}^{\log_b n - 1} \left(\frac{1}{16}\right)^i}_{\text{bounded by constants}} \in \Theta(n^4 \log n)$$

Say we have a recurrence $T(n) = aT(\frac{n}{b}) + \underbrace{cn^j \log^k n}_{f(n)}$ for some $\begin{matrix} a>0 \\ b>1 \\ c>0 \\ j\geq 0 \\ k\geq 0 \end{matrix}$. When do we have some $\delta<1$ such that $af(\frac{n}{b}) \leq \delta f(n)$ for sufficiently large n (regularity)?

Notice $af(\frac{n}{b}) = a(\frac{n}{b})^j c\log^k(\frac{n}{b}) = ab^{-j}n^j c(\log n - \log b)^k < (ab^{-j})n^j c\log^k n = (ab^{-j})f(n)$ so we want $(ab^{-j})<1$.

In particular, $(ab^{-j}) < (ab^{-d}) = 1$ so $j>d$ needed.

Corollary: Say $T(n) = aT(\frac{n}{b}) + f(n)$ satisfying $f(n) \in \Omega(n^{d+\varepsilon})$ for some $\begin{matrix} \varepsilon>0, \\ a>0, \\ b>1, \end{matrix}$ where $f(n) = cn^j \log^k n$ for some $c>0$, $j\geq 0$.

Then $f(n)$ always satisfies the regularity condition.

Proof: Since $cn^j \log^k n \in \Omega(n^{d+\varepsilon})$ for some $\varepsilon>0$, the idea is that $n^j \underbrace{\log^k n}_{\in\omega(n^\varepsilon)} \geq n^d n^\varepsilon \Rightarrow j>d = \log_b a$.

Formally, we have $cn^j \log^k n \geq Cn^{d+\varepsilon} = Cn^d n^\varepsilon > Cn^d \log^k n$ for sufficiently large n

so $\Theta(f(n)) = \Theta(n^j \log^k n) \subseteq \Omega(n^{d+\varepsilon})$ and $\Theta(n^{d+\varepsilon}) \subseteq \underbrace{\Omega(n^{d+\varepsilon})}_{\text{grows} \geq n^{d+\varepsilon}} \subseteq \omega(n^d \log^k n)$ since $n^{d+\varepsilon} > n^d \log^k n$

$\Rightarrow \Theta(n^j \log^k n) \subseteq \omega(n^d \log^k n)$ so $j>d$.

Therefore, we have $af(\frac{n}{b}) = a(\frac{n}{b})^j \log^k(\frac{n}{b}) = ab^{-j} n^j (\log n - \underbrace{\log b}_{b>1\Rightarrow \log b>0})^k < ab^{-j} n^j \log^k n = \underbrace{(ab^{-j})}_{\delta} f(n) = \delta f(n)$

where $\delta<1$ as required. $\square$

Leetcode

1 merge-k-sorted-lists on k sorted singly-linked-lists.

hard.

Naïve: create a pointer for each list L_i (say pointer p_i). $O(k)$

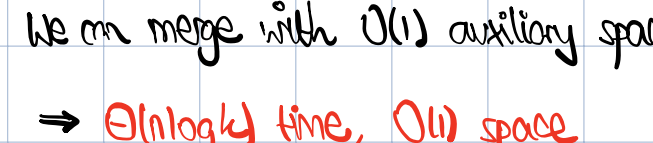
for each element, find $\min_i(L_i[p_i])$ and append element to a new linked list. $O(k)$ n times

increment that pointer p_i. $O(1)$

$\Rightarrow \Theta(nk)$ time, $\Theta(n)$ space

Merge: modify the merge step from merge-sort. We divide-and-conquer on k.

merge step from merge-sort: essentially naive algorithm on 2 lists.



since each SLL already sorted, we don't need to do the sorting step, just the merging step.

given k sorted SLLs:

base case $k \leq 2$: naive merge

first half → merge first $\lfloor \frac{k}{2} \rfloor$ lists

second half → merge last $\lceil \frac{k}{2} \rceil$ lists } each list has $O(n)$ elements

merge first and second half

We have a runtime $T(k) = \begin{cases} 2T(\frac{k}{2}) + cn & \text{if } k > 2 \\ O(1) & \text{if } k \leq 2 \end{cases}$ ← on average, n total in k lists ⇒ n/k per list

why cn here? because each iter reads a merge step

runtime analysis; would then prove by induction

⇒ let $k = 2^m$ for some m, so we have

$$T(2^m) = 2T(2^{m-1}) + cn$$

$$= 2(2T(2^{m-2}) + cn) + cn$$

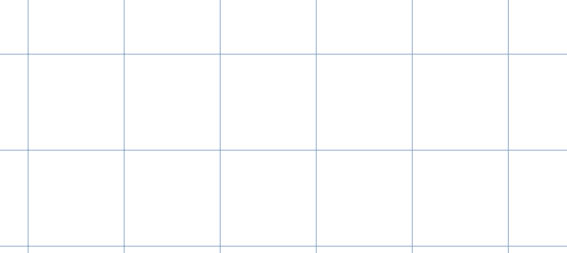
$$= \dots = 2^m T(2^{m-m}) + cn\sum_{i=0}^{m-1} 2^i \quad \} k = 2^m \Rightarrow m = \log k$$

$$\Rightarrow T(2^m) = 1 \cdot 2^{\log_2 k} + cn(2^m - 1) = n + cn\log k \in \Theta(n \log k)$$

↖ from base case

We can merge with $O(1)$ auxiliary space

⇒ $\Theta(n \log k)$ time, $O(1)$ space



Heaps Idea: in naive approach, we want to find smallest element from k pointers $p_1, \dots, p_k$.

We can do this using a min-heap instead of a linear search.

initialize all p_i to start of L_i

add all $L_i[p_i]$ to a min-heap

for n iterations:

say p_j is the min in heap

pop min. element from heap and add to new list $O(1)$

increment p_j $O(1)$

add $L_j[p_j]$ to heap $O(\log k)$ since heap always has k elements

⇒ $\Theta(n \log k)$ time, $O(k)$ space. ← can be reduced to $O(k)$

2 med **sort-colours**: given an array of n elements in $\{0,1,2\}$, design an algorithm to sort the array.

Naive: bucket-sort (or radix-sort). $\Theta(n)$ time, $\Theta(n)$ space.

↓

counting-sort: count occurrences of each category and override old array ⇒ $\Theta(n)$ time, $\Theta(1)$ space.

but what if each element $x \in A$ has additional information? counting-sort would lose info.

Modified: start with a first-pass counting each category ($\Theta(n)$), say n_0, n_1, n_2 counts.

then, initialize indices $i_0 = 0$, $i_1 = n_0$, $i_2 = n_0 + n_1$.

initialize temporary variable $t_1 = A[i_0]$, $t_2 = \text{NULL}$

for n iterations: notice $t_1 \in \{0,1,2\}$

$t_2 = A[i_{t_1}]$ the element that $A[t_1]$ should replace

$A[i_{t_1}] = t_1$

$t_1 = t_2$ next iteration will find space for the popped element

increment i_{t_1}

⇒ $\Theta(n)$ time, $\Theta(1)$ space without losing any information

We must show the correctness of this algorithm, since it is not immediately obvious.

* Exercise

swap-sort: we want to use swapping to design a 1-pass, $\Theta(n)$ time, $\Theta(1)$ space algorithm.

idea: initialize $i_0 = 0$, $i_2 = n-1$

for $j=0$, $j<n$, j++:

if $A[j]$ is 0: swap $A[j]$ and i_0, i_0++

if $A[j]$ is 2: swap $A[j]$ and i_2, i_2--

else pass it will naturally end up in the right place

3 med **remove-duplicates-sorted-ii**: A: sorted array of n elements. remove dupes such that each unique element appears at most twice without changing sorted order.

read/write approach: one pointer goes through entire array to read: when we hit a new element, we write.

initialize read index and write index $r_i = 1$, $w_i = 1$

for $r_i < n$, r_i++:

if $A[r_i] \neq A[w_i - 1]$: current element not same as previously written element

$A[w_i] = A[r_i]$ write

w_i++

otherwise r_i incremented (we hit a dupe ⇒ move on)

return $A[0 : w_i - 1]$

4 med **k-radius-subarray-averages**: array $A[0..n-1]$, some integer $k \geq 1$. Create array of average of subarray of A (subarray size k, centered at index i). Essentially sliding window average problem.

Divide and Conquer Algorithms

Counting Inversions

An inversion in an array A is a pair of indices (i,j) such that $i<j$ but $A[i]>A[j]$.

Consider the problem of counting inversions. ⇝ how "inverted" is a sequence?

⇒ can use D&C here? If we count #inversions in first half, then second half, can we use this info to count inversions of both halves put together?

⇝ must count inversions with one element in each half → cross-inversion pairs are easier to find because we know their relative positions

⇝ sort each half (not lossy because we already counted in-half inversions!)

then we can just count elements $>a_i$ in second half for each i → can do this equiv. on the other half instead

⇒ reduced the problem to efficient counting problem. Let halves be H_1, H_2.

Idea 1: H_1 sorted ⇒ binary search for number $>a_j$ where $a_j \in H_2$ (⇒inversion)

$\Theta(\log n)$ time for each a_j ⇒ $\Theta(n\log n)$ time to find all cross-inversions.

Idea 2: The info we want can be determined using a merge from merge-sort.

→ naive merge

When we insert $x \in H_2$ to the merged list

⇒ $\Theta(n)$ time.

```
count(A[1..n]):
    if n=1: return 0
    count(A[1..n/2])          ← T(n/2)   (divide)
    count(A[n/2+1..n])        ← T(n/2)   (divide)
    sort(A[1..n/2])           ← Θ(nlogn) (conquer)
    sort(A[n/2+1..n])         ← Θ(nlogn) (conquer)
    merge-count(A[1..n/2], A[n/2+1..n])  ← Θ(n)
```

is this step necessary? bottleneck...

Therefore $T(n) = 2T\left(\frac{n}{2}\right) + f(n)$ where $f(n) \in \Theta(n\log n)$

⇒ $T(n) \in \Theta(n\log^2 n)$ by Master Theorem.

Merge Count

```
count-and-sort(A[1..n]):
    if n=1: return 0
    S1 = count-and-sort(A[1..n/2])
    S2 = count-and-sort(A[n/2+1..n])
    S3 = merge-count(A[1..n/2], A[n/2+1..n])
    return S1+S2+S3
```

remove redundant sorting step because we already sort in recursive subprocess' merge step(s).

⇒ $T(n) = 2T(\lfloor\frac{n}{2}\rfloor) + O(n) \in \Theta(n\log n)$. Theoretically best (based on comparison-based LB).

*Exercise: write merge-count pseudocode

Maximum Subarray

Given array A of elements $a_1,\dots,a_n$, we want to find (i,j) that maximizes $\sum_{k=i}^{j} a_k$.

Naïve For every i, iterate j and compute $\sum_{k=i}^{j} a_k$.

Use sliding window for each addition to be $O(1)$ ⇒ we have time $O(n^2)$.

Would D&C help? If we had max subarrays for halves H_1, H_2, would this help solve for A?

→ optimal solution either contained in H_1, H_2, or crosses between H_1 and H_2.

$\Rightarrow$ after solving subproblem, we just need to find max. subarray that crosses $\frac{n}{2}$ (so $i \le \frac{n}{2} < j$).

Assume that (i,j) is optimal with $i \le \frac{n}{2} < j \Rightarrow [i, \frac{n}{2}]$ must be a max. subarray at $\frac{n}{2}$.

FAC not, so $[i', \frac{n}{2}]$ max. subarray.
then $\sum_{k=i'}^{j} a_k > \sum_{k=i}^{j} a_k$ contradicts (i,j)-maximality ↯

Claim: For $i \le \frac{n}{2} < j$, we have $[i,j]$ is a maximum subarray crossing midpoint
$\iff$ $[i, \frac{n}{2}]$ max. subarray ending at $\frac{n}{2}$ and $[\frac{n}{2}+1, j]$ max. subarray starting at $\frac{n}{2}+1$.

Proof: Above argument. □

Finding max. subarray with a fixed starting/ending point can be done in $O(n)$ time.

Therefore:
Start with $A[1..n]$
find (i_1, j_1) for H_1 and compute $s_1 = \sum_{k=i_1}^{j_1} a_k$ $T(\frac{n}{2})$
find (i_2, j_2) for H_2 and compute $s_2 = \sum_{k=i_2}^{j_2} a_k$ $T(\frac{n}{2})$
compute whether $[i_1, \frac{n}{2}]$ max. subarray
$[\frac{n}{2}+1, j_2]$ max. subarray } $\Theta(n)$
yes $\Rightarrow [i_1, j_2]$ max. for A.
else $\max(s_1, s_2)$ max. for A. } $\Theta(1)$

$\Rightarrow T(n) = 2T(\frac{n}{2}) + \Theta(n)$
$\in \Theta(n \log n)$.

* Exercise (HARD): Find an $\Theta(n)$ time algorithm for maximum subarray problem.

* Exercise: extend algo. to work in circular settings? → just run algorithm on concatenated array AA

Median Finding (randomized) quick-select (see CS240) works; $\Theta(n)$ expected time. But this is a randomized algo..

problem with quick-select: the pivot choice needs to be good, but might be terrible.

Deterministic Algo. There is an interesting deterministic algorithm that would always return a number with rank $\frac{3n}{10} \le r \le \frac{7n}{10}$ (somewhat close to median) in $O(n)$ time.
$\Rightarrow$ we can find a reasonable pivot in $O(n)$.

$\Rightarrow T(n) \le T(\frac{7n}{10}) + P(n) + cn \Rightarrow T(n) \in O(n)$.
($\frac{7n}{10}$: problem size reduces at least by $\frac{3n}{10}$; $P(n)$: pivot selection time)

Good Pivot Finding

The idea is to find median of medians.

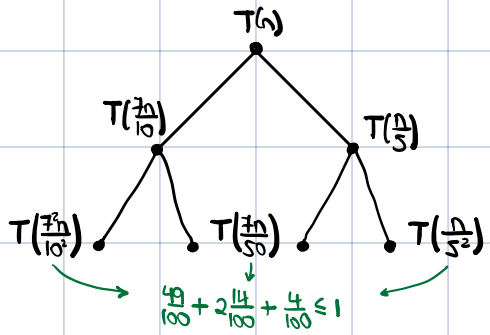
1) Divide $A[1..n]$ into $\frac{n}{5}$ groups (each group has 5 numbers) in $O(n)$.
2) Find medians $b_1, \dots, b_{\frac{n}{5}}$ for each group, each $O(1)$ time $\Rightarrow O(n)$ time.
3) Find the median of $b_1, \dots, b_{\frac{n}{5}}$ by recursive call $\Rightarrow T(\frac{n}{5})$ time. → we use mutual recursion between pivot and median finding

Lemma: Let r be the rank of the median of medians. Then $\frac{3n}{10} \le r \le \frac{7n}{10}$.

Proof: See L03.pdf from CS341 notes.

Therefore, we have $P(n) = T(\frac{n}{5}) + c_1 n \Rightarrow T(n) \le T(\frac{7n}{10}) + P(n) + c_2 n = T(\frac{7n}{10}) + T(\frac{n}{5}) + (c_1 + c_2) n$

Recursion Tree Analysis:



$c_3 n$ operations

$c_3\left(\frac{3n}{10} + \frac{n}{5}\right) = c_3\left(\frac{9n}{10}\right)$ operations

$\leq c_3 n$ operations

* Exercise: complete.

Closest Pair Input: n points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^2$.

Output: indices $1 \leq i < j \leq n$ such that Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ minimized.

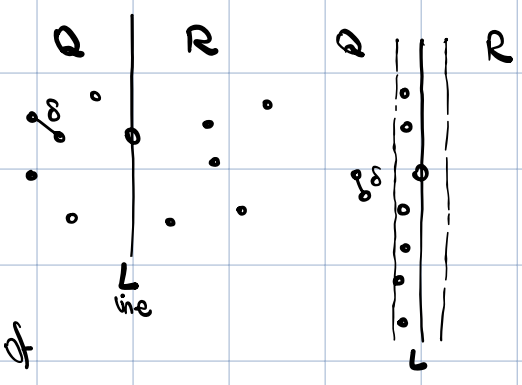
Naïve Enumerate all pairs, $O(n^2)$.

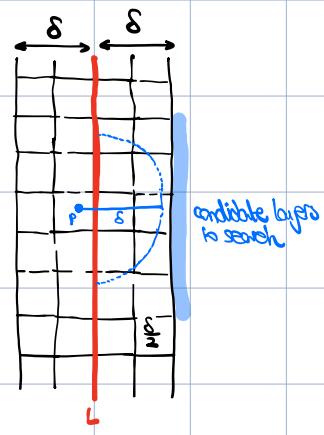
D&C: Split the problem in half. Would knowing the closest pair in each half help? → we can easily take the closer of the two pairs ⇒ suffices to find cross-half pairs

Suppose we halve by x-coordinate median.

Notice we only need to check crossing pairs for points that are $< \delta$ distance from L

but this can still be all the points!





Idea: divide the δ-neighbourhood of L into a grid of squares with side length $\frac{\delta}{2}$.

* We assume all points have distinct x-coord.

Lemma: Each box contains at most one point.

Proof: IFAC not ⇒ we have a pair within Q or R (non-crossing) with distance at most $\frac{\delta}{\sqrt{2}} < \delta$. ↯ □

Lemma: Each point only needs to compute distances with points within two horizontal layers. (up and down)

Proof: Vertical squares are non-crossing. Diagram above shows that ±2 horizontal layers stay δ-close, else not. □

⇒ each candidate point must compare with at most 11 other points ⇒ search space $O(n)$ squares.

Algorithm 1

1. Find line L by finding median. $O(n)$ ← use quick-select
2. Recursively solve problem in Q and R (halves) $2T(\frac{n}{2})$
3. Using a linear scan, remove all points outside δ-neighbourhood. $O(n)$
4. Sort points by y-value $O(n \log n)$ bottleneck ← we don't need to do this in the recursion
5. For each point, compare distance to next 11 points in y-ordering. $O(n)$
6. Return minimum distance found.

⇒ $T_1(n) = 2T_1(\frac{n}{2}) + \Theta(n \log n) \overset{\text{Master}}{\Rightarrow} T_1(n) \in \Theta(n \log^2 n)$.

If we sort beforehand we get $T(n) = 2T(\frac{n}{2}) + O(n) \in O(n \log n)$.
→ also allows us to remove the median-finding step

Remark: There is a randomized algorithm for closest pair with expected $O(n)$ time.

Arithmetic Operations with D&C

This is where D&C is most powerful. Eventually, this also allows us to parallelize operations for high performance.

See CUDA and tensor operations for instance.

A Integer Multiplication

Given n-bit numbers $a=a_1 \dots a_n$ and $b=b_1 \dots b_n$, we want to compute ab.

Grade-school multiplication requires $\Theta(n^2)$ bit operations.

D&C: Say we can do n bit multiplication efficiently. How can this help $2n$ bit multiplication?

Let x, y be $2n$ bit. Write $x=x_1x_2$ and $y=y_1y_2$ where x_1, x_2, y_1, y_2 are n-bit

$$\Rightarrow x = x_1 2^n + x_2 \quad \text{and} \quad y = y_1 2^n + y_2$$

$$\Rightarrow xy = (x_1 2^n + x_2)(y_1 2^n + y_2) = x_1y_1 2^{2n} \oplus (x_1y_2 + x_2y_1) 2^n \oplus x_2y_2.$$

(easy) (extra (use bit-shift)) (easy) (easy) (extra (bit-shift)) (easy)

So $T(n) = 4T(\frac{n}{2}) + O(n) \Rightarrow T(n) \in O(n^2)$ (Master; extra operations; bit) no improvement. Notice we did nothing clever to combine subproblems $\Rightarrow$ we get no speedup.

So what can we do?

Karatsuba's Algorithm Reduce to three subproblems: x_1y_1, x_2y_2, and $(x_1+x_2)(y_1+y_2)$.

Compute middle term using $(x_1+x_2)(y_1+y_2) - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1$.

$$\Rightarrow T(n) = 3T(\tfrac{n}{4}) + O(n) \overset{\text{Master}}{\Longrightarrow} T(n) \underset{\text{bit}}{\in} O(n^{\log_2 3}) = O(n^{1.59}).$$

B Polynomial Multiplication

Same as Karatsuba. * Exercise

C Matrix Multiplication ($n \times n$)

Given two $n \times n$ matrices, suppose we can multiply them efficiently.

Say we have two $2n \times 2n$ matrices A, B.

$$\Rightarrow A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \text{ block matrices where blocks are } n \times n.$$

$$\Rightarrow AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{12}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \Rightarrow \text{eight subproblems.}$$

$$\Rightarrow T(n) = 8T(\tfrac{n}{2}) + O(n^2) \in O(n^3) \quad \text{no improvement.}$$

Strassen's Algorithm

Smart reduction for matrix multiplication. Details omitted, but reduces matmul to $8 \to 7$ subproblems.

$$\Rightarrow T(n) = 7T(\tfrac{n}{2}) + \Theta(n^2) \Rightarrow T(n) \in O(n^{\log_2 7}) = O(n^{2.81}).$$

Many combinatorial problems can be reduced to matmul $\overset{\text{Strassen}}{\Longrightarrow}$ these can be solved in $O(n^3)$ time.

* Ex. Determine whether $G=(V,E)$ has a triangle.

Fast Fourier Transform solves integer and polynomial multiplication in $O(n \log n)$. See CS487.

D Exponentiation given $a, n \in \mathbb{N}$, compute a^n.

Naive: iterative algorithm $O(n)$ (slow).

D&C: we can consider the rough idea of $a^n = a^{\frac{n}{2}} a^{\frac{n}{2}}$. we have 2 main cases here:
n even and n odd.

if n even: $a^n = a^{\frac{n}{2}} a^{\frac{n}{2}}$ O(1) operation* → we assume these are O(1) word operations

if n odd: $a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lceil \frac{n}{2} \rceil}$ or $a^n = a a^{\lfloor \frac{n}{2} \rfloor} a^{\lfloor \frac{n}{2} \rfloor}$ O(1) operation*

$\Rightarrow T(n) = 2T\left(\frac{n}{2}\right) + O(1) \in O(\log n)$ exponential improvement over naive.

E Quake Fast Inverse Square Root

* Exercise.

General Structures for Proof of Correctness

In CS245, we analyzed computational correctness proofs for simple algorithms. We generalize here.

Iterative Algorithms

Of the form initial conditions → for/while loop (iterative process) → termination.

We do 3 things:
1. Show that the loop invariant is true at the start of the iterative process
2. Show that the loop invariant is maintained (inductively)
3. Prove that the iterative process terminates

⇝ choose a loop invariant such that the loop invariant itself helps prove correctness

Recursive Algorithms

Similar inductive approach.

1. Show correctness for base case of recursion
2. Show correctness for inductive step (usually strong induction)

More Arithmetic Operations

Polynomial Multiplication (Karatsuba's)

Say we have $A(x) = \sum_{k=0}^{n} a_k x^k$ and $B(x) = \sum_{k=0}^{n} b_k x^k$ so $\deg(A) = \deg(B) = n$.

We want to compute $C(x) = A(x)B(x)$ with $\deg(C) = 2n$ efficiently. Denote $C(x) = \sum_{k=0}^{2n} c_k x^k$.

Naive: Compute $c_k = \sum_{i+j=k} a_i b_j = \sum_{i=0}^{k} a_i b_{k-i}$ for every $k \in \{0, \dots, 2n\}$.

$\Rightarrow$ each c_k computation takes $k+1$ operations, so we have a runtime

$\sum_{k=0}^{2n} (k+1) \in \Theta(n^2)$.

Idea (D&C): Say we can compute $\deg(\frac{n}{2}) \times \deg(\frac{n}{2})$ polynomial multiplication fast. How can this help?

Let $A(x) = A_1(x) + A_2(x)$ where $A(x) = \underbrace{\sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} a_k x^k}_{A_1(x)} + \underbrace{\sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} a_k x^k}_{A_2(x)}$ and similarly $B(x) = B_1(x) + B_2(x)$.

$\Rightarrow C = (A_1 + A_2)(B_1 + B_2)$

$= \underset{n \times n}{A_1 B_1} + \underset{n \times \frac{n}{2}}{A_1 B_2} + \underset{\frac{n}{2} \times n}{A_2 B_1} + \underbrace{\underset{\frac{n}{2} \times \frac{n}{2}}{A_2 B_2}}_{\text{fast}}$. However, we can do better subproblem division.

The key idea is we want all subproblems to be $(\frac{n}{2}) \times (\frac{n}{2})$.

Notice if we instead write $A(x) = \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} a_k x^k + \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} a_k x^k = x^{\lfloor \frac{n}{2} \rfloor + 1} \underbrace{\sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} a_{k+\lfloor \frac{n}{2} \rfloor + 1} x^k}_{A_1(x)} + \underbrace{\sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} a_k x^k}_{A_2(x)}$

then we have $C(x) = (x^{\frac{n}{2}} A_1(x) + A_2(x))(x^{\frac{n}{2}} B_1(x) + B_2(x))$

$= x^n A_1B_1 + x^{\frac{n}{2}} A_1B_2 + x^{\frac{n}{2}} A_2B_1 + A_2B_2$ where each subproblem is $(\frac{n}{2}) \times (\frac{n}{2})$.

Since $x^n \sum_{i=0}^{n} d_i x^i = \sum_{i=0}^{n} d_i x^{i+n}$ can be done in $O(n)$ for arbitrary coeffs. d_i, we have

$T(n) = 4T(\frac{n}{2}) + \Theta(n)$ with $\log_2 4 = 2$ so $\Theta(n) \subseteq O(n^{\log_b a - \varepsilon}) \underset{\varepsilon = \frac{1}{2} > 0}{=} O(n^{\frac{1}{2}})$

$\xrightarrow[\text{Master (loof)}]{} T(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$. No improvement.

The issue is that our subproblem space is too large. Karatsuba's algorithm reduces this space.

Karatsuba's

Notice that it suffices to compute $(A_1 + A_2)(B_1 + B_2) = A_1B_1 + A_1B_2 + A_2B_1 + A_2B_2$

$\xrightarrow{\text{somehow}} x^n A_1B_1 + x^{\frac{n}{2}} A_1B_2 + x^{\frac{n}{2}} A_2B_1 + A_2B_2$.

However for the final transformation, we need access to each (almost!) subproblem term individually.

$= x^n \underbrace{A_1B_1}_{\text{term 1}} + x^{\frac{n}{2}} \underbrace{(A_1B_2 + A_2B_1)}_{\text{term 2}} + \underbrace{A_2B_2}_{\text{term 3}}$. We can reduce as follows:

- compute A_1B_1 term 1 and A_2B_2 term 3 $(2T(\frac{n}{2}))$
- compute $\underbrace{(A_1 + A_2)(B_1 + B_2)}_{T(\frac{n}{2}) + O(n)} - A_1B_1 - A_2B_2 = (A_1B_2 + A_2B_1)$ term 2

$\Big\} 3T(\frac{n}{2}) + O(n)$

$\Rightarrow$ we have $d = \log_2 3 > 1 \xrightarrow[\text{Master leaf}]{} T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.57})$.

Quake Fast Inverse Sqrt

we want to compute $\frac{1}{\sqrt{x}}$.

Idea: 1 alias x to an integer to compute approx. $\log_2 x$.

2 use $\log_2 x$ to compute $-\frac{1}{2}\log_2 x = \log_2 \frac{1}{\sqrt{x}}$.

3 alias back to float to approximate $2^{\log_2 \frac{1}{\sqrt{x}}} = \frac{1}{\sqrt{x}}$. ← by this step, we have a (bad) approx. of $\frac{1}{\sqrt{x}}$

4 refine using Newton's method.

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

The details are in the type aliasing steps that require strong understanding of float representations.

Addition Machine

Consider a restriction of the word-RAM model called the addition machine that supports only $(+, -)$ and comparison.

Fact: $\text{div}(x, y) = x // y$ can be implemented in $\Theta(\log(\frac{x}{y}))$ time.

is_odd(n) can be implemented using $\text{div}(n, 2) \Rightarrow \Theta(\log n)$ time.

Goal: implement $\text{square}(n) = n^2$.

Naive: iteratively add n to a variable n times $\Rightarrow \Theta(n)$.

D&C: how can we break n^2 into recursive subproblems? Say $(\frac{n}{2})^2$ can be computed fast.

$n^2 = 4(\frac{n^2}{4}) = 4(\frac{n}{2})^2$ where we can add $(\frac{n}{2})^2$ to itself 4 times in $O(1)$.

If $n = 2^k$ for some k, we clearly have $T(n) = T(\frac{n}{2}) + O(1) \Rightarrow T(n) \in \Theta(\log n)$.

However, if $n \neq 2^k$, then we would have to use is_odd(n) before recursive call, so

$$T(n) = T(\tfrac{n}{2}) + \Theta(\log n) \underset{\text{balanced}}{\Rightarrow} T(n) \in \Theta(\log^2 n).$$ This is the high-level idea.

Leetcode

5 med. binary search on offset sorted array (search-rotated-sorted-array)

Goal: Given sorted array rotated arbitrarily, search for an element k.

eg. [6, 7, 8, 1, 2, 3, 4, 5] — If we know shift value (e.g. smallest element index), we can solve easily

⇒ we want to find smallest element in O(log n) first.

D&C: Say we split into L and R, and can find minimum for both. Then $m = \min(m_L, m_R)$.

⇒ $T(n) = 2T(\frac{n}{2}) + O(1) \in O(\log n)$.

Then we simply offset our binary search by the index of this minimum element.

D&C 2: At least one half of A must be fully sorted. → possible for both halves sorted, e.g. [4, 5, 6, 1, 2, 3]

1 Check which half sorted (O(1) time by checking first and last elements). WLOG, A_L sorted.

2 If $k \in A_L$ sorted, regular binary search. Else recursive call on A_R.

⇒ $T(n) \leq \max(T(\frac{n}{2}), c\log n) + O(1)$. (max: asymptotic max.)

If $\max(T(\frac{n}{2}), c\log n) = c\log n$, then $T(n) \in \Theta(\log n) \Rightarrow T(\frac{n}{2}) \in \Theta(\log n)$. (max: asymptotic)

else $\max(T(\frac{n}{2}), c\log n) = T(\frac{n}{2}) \Rightarrow T(n) \in O(\log n)$.

so $T(n) \in O(\log n)$.

6 med. pow-x: calculate x^n given $x \in \mathbb{Z}$ and $n \in \mathbb{N}_{\geq 0}$ (not both zero).

Naive: O(n) by iteration.

D&C: If we know $x^{\frac{n}{2}}$ and $x^{\frac{n}{2}}$, then $x^n = x^{\frac{n}{2}} x^{\frac{n}{2}}$. Therefore we do such a binary divide.

This gives runtime $T(n) = 2T(\frac{n}{2}) + O(1) \in \Theta(\log n)$.

```
pow(x, n):
  if n=0: return 1
  if n=1: return x
  return pow(x, ⌈n/2⌉) * pow(x, ⌊n/2⌋)
```

7 med. perfect-square: given n, identify if n is a perfect square without using sqrt().

Naive: Iterate i from 1 to n. Calculate i^2, check against n.

$i^2 < n \Rightarrow$ continue
$i^2 = n \Rightarrow$ perfect square
$i^2 > n \Rightarrow$ terminate

$\Rightarrow T(n) \in O(\sqrt{n})$ because we stop at $i = \lceil \sqrt{n} \rceil$.

Binary search: search for $\sqrt{n}$ in $[1..n]$. Say search range is $[1, y]$.

Calculate $(\lceil \frac{y}{2} \rceil)^2$. If $> n$, then $\sqrt{n}$ in left half-range. If $> n$, in right half-range.
else $= n \Rightarrow$ done.

Recursive case: we search desired half-range.

$\Rightarrow T(n) = T(\frac{n}{2}) + O(1) \Rightarrow T(n) \in \Theta(\log n)$.

8 med. spells-and-potions: input: two positive integer arrays S and P, length n and m; integer "success".
a potion pair is successful if $S[i]P[j] \geq$ success.

output: integer array A of length n where A[i] = #potions that will form a successful pair with S[i].

Naive: Direct-address table $A[1..n]$. Enumerate all pairs and update $A \Rightarrow O(mn)$.

Sort First: Sort S and P. Then we know that $S[i]P[j] \geq$ success $\Rightarrow S[i..n]P[j..m] \geq$ success.

$\Rightarrow$ for each $S[i]$, we just need $P[j]$ such that $P[j] \geq \frac{\text{success}}{S[i]}$.

Binary search on P to find j such that $P[j] \geq \frac{\text{success}}{S[i]}$.

so $T(n) = \underbrace{\Theta(n \log n)}_{\text{sort } S} + \underbrace{\Theta(m \log m)}_{\text{sort } P} + \Theta(\overset{n \text{ times}}{n} \underbrace{\log m}_{\text{binary search}})$.

Notice we don't use the fact that $S[i]$ sorted. $\Rightarrow$ we can avoid sorting S.

reduce $\rightsquigarrow$ $T(n) = \Theta(m \log m) + \Theta(n \log m)$.

Sorting both and using some smart algorithm does not help, since $n \log n$ or $m \log m \geq m \log n$ and $n \log m$ based on $n > m$ or $n \leq m$.

What if S and P were sorted beforehand? Can we speed things up more?

Consider the observation that $S[i]P[j] \geq$ success $\Rightarrow S[i..n]P[j..m] \geq$ success. Let i be fixed.

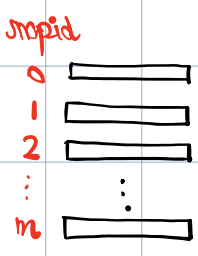
Let j_i be the smallest index such that $S[i]P[j_i] \geq$ success

$\Rightarrow j_i$ is a lower-bound index for $i-1$, so we only must look for $j_{i-1} \geq j_i$ from P. (in practice, potentially shrinks search space)

9 med **Snapshot-array:** data structure `SnapshotArray(int length)` that:

- starts with array of zeros of specified length
- `set(index, val)` sets `A[index] ← val`
  `snap()` takes a "snapshot" of the array and returns a snapshot ID (starts at 0, increases by 1 every snap)
  `get(index, snap_id)` gets index at `A[index]` at that snapshot

Binary Search every snapshot stored as an array $\Rightarrow$ we have a matrix.



$\Rightarrow$ binary search on snap_id to retrieve array, then return element at index idx.

`get()` time $O(\log m)$
`set()` time $O(1)$
`snap()` time $O(1)$ amortized
} space $\Theta(mn)$

Direct Addressing Store data in matrix (flattened representation as contiguous array) as above.

Then `get(idx, snap_id)` returns $M[n * \text{snap\_id} + \text{idx}]$.

`get()` time $O(1)$
`set()` time $O(1)$
`snap()` time $O(1)$ amortized
} space $\Theta(mn)$

Sparse Space Optimization Idea: each index i keeps a sorted list of pairs (snap_id, value) so

$A = [\downarrow \cdot \cdot \dots \cdot]$ where each $A[i]$ only updates when that value updates.

$[(s_1, v_1), \dots, (s_k, v_k)]$

for `get(i, s)` we get the sorted pairs list for $A[i]$ $(O(1))$
then binary search $A[i]$ for the largest snap_id $\leq s$, and return corresponding value

10 med **2D-matrix search:** $M \in \mathbb{Z}^{m \times n}$ with each row of M sorted, and last element of row $r_i <$ first element of r_{i+1}.
Search for value k in M. ⇝ we can binary search

1 Search largest $v \leq k$ along $r_i[0]$ (first element of each row)
2 Once some r_i identified, binary search through r_i's elements for k

⇒ $O(\log n + \log m) = O(\log mn)$ runtime

11 easy **find-peaks:** find local maxima of an array A (elements greater than their neighbours).

Observation: worst-case we have $A = [0, 1, 0, 1, \ldots, 0]$ so there are $\Omega(n)$ peaks

⇒ our algorithm will be at best $\Omega(n)$.

Linear search: l_i, r_i, m_i indices left, right, and middle.

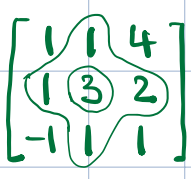
```
l_i = 0, r_i = 2, m_i = 1
peaks = []
while m_i < n-1:
    if A[l_i], A[r_i] < A[m_i]:
        peaks.append(m_i)
        notice r_i cannot be a peak ⇒ skip one
        l_i, r_i, m_i += 2
    else:
        not a peak
        l_i, r_i, m_i += 1
```

12 med **find-peak-element:** given A, find any peak of A.

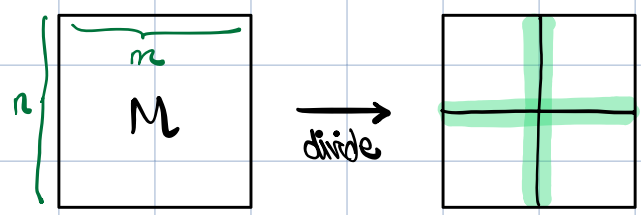
Naïve: same as 11 find-peaks, runtime $O(n)$.

Idea: D&C. Given arrays A_L and A_R, how can we find a peak?
either A_L has a peak ⇒ return
or A_R has a peak ⇒ return
or there is a peak in $A_L[-2]$, $A_L[-1]$, $A_R[0]$, $A_R[1]$ ⇒ runtime $T(n) = 2T(\frac{n}{2}) + O(1) \in \Theta(\log n)$
else no peaks

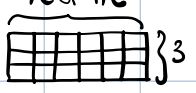
13 med **find-peak-element-2D:** find a local maximum of a matrix where neighbours are top, left, right, bottom.

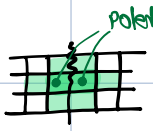


peak element 3



must search for cross-border peaks ⇒ $O(m+n)$ naïve

⇝ can we find cross-border peaks in $O(\log(m) + \log(n))$?
Re-use find-peak-element and modify for 2D case. Goal: given $S =$  shape, find peaks (in middle row)

D&C: split into S_L and S_R.
S_L or S_R has a peak ⇒ done
else peak must be crossing between: 

⇝ $O(1)$ check
or no peak

⇒ finding cross-block peaks is $2T_S(n) + 2T_S(m)$ where $T_S(n) = 2T_S(\frac{n}{2}) + O(1) \in \Theta(\log n)$
($2T_S(n)$: search block crossings (vert.); $2T_S(m)$: search block crossings (horiz.))

⇒ cross-border peaks searchable in $O(\log n + \log n)$ so $T(m,n) = 4T(\frac{n}{2}, \frac{m}{2}) + \Theta(\log n + \log m) \overset{\text{prove}}{\in} \Theta(\max(n \log m, m \log n))$ *

14 med **group-anagrams:** given an array of strings, group all anagrams together into sublists.

Say we have n strings and $m = \max_{s \in A} \text{len}(s)$ (longest string length).

Naive: Generate a character-count hashmap for each string $\Rightarrow n$ hashmaps $O(mn)$
Then check pairwise equality and group accordingly. $O(n^2)$

Prop.: Let h_1 and h_2 be hashmaps. Then we can compute whether $h_1 \neq h_2$ in $O(1)$ expected amortized time.
Idea: check equality of the hashes of h_1 and h_2 instead of element-wise "D"
Remark: holds for arbitrary size but since our hashmaps are $O(1)$ size, we can guarantee $O(1)$ equality-checking

$\Rightarrow \Theta(n^2 + mn)$ time; bottleneck is equality-checking.

Hashing Arrays: Initialize arrays $A_1, \dots, A_n$ of size 26 to act as our character counters.
Then construct a hashmap where keys are A_i and values are string arrays.

don't use sets here because we want to preserve duplicates

group-anagrams(A):
- Initialize $A_1, \dots, A_n = [0, \dots, 0]$ (26) character count arrays $O(n)$
- Initialize $H = \text{dict}()$
- $n\times$ for $s \in A$:
  - build A_s by character-counting s $O(n)$ due to linear scan
  - if $A_s \in H.\text{keys}()$: $O(1)$ check for existence
    - $H[A_s].\text{append}(s)$
  - else:
    - $H[A_s] = [s]$ $O(1)$

NOTE: arrays are an unhashable type so we ★ encode A_i as strings (eg. $[0,1,0] \mapsto$ "0-1-0") and use these as keys

$\Rightarrow O(mn)$ runtime and $O(n)$ space

15 med **top-k-frequent:** integer array $A[0..n-1]$, and some $k \leq n-1$. Return k most frequent elements in A.

Naive: linear scan and counting $\rightsquigarrow$ element count hashmap $\rightsquigarrow$ sort by frequency.

linear scan $O(n)$ $\Rightarrow O(n \log n)$ time
sort (k,v) pairs by v $O(n \log n)$

Idea: we must solve the sorting bottleneck.

1 randomized-quick-select

notice we don't need to fully sort, just finding element of rank-k suffices
$\Rightarrow$ modify randomized-quick-select to also return the left ($\geq$) subarray in $\Theta(n)$ expected time

r-q-s(A, k):
- pivot = rand(len(A)-1)
- pivot_val = A[pivot]
- $A_L = []$, $A_R = []$
- for $a_i \in A$:
  - add to A_L or A_R based on $\leq$ pivot_val
- return recursive call

expected $\Theta(n)$ time but worst-case $\Omega(n^2)$ $\rightarrow$ problem.

k-frequent(A, k):
- $H = \text{dict}()$ $O(1)$
- count { for num in A:
  - $H[\text{num}] \leftarrow H[\text{num}]+1$ else initialize $H[\text{num}] = 1$ } $O(n)$
- return r-q-s(H, k) $O(n)$ expected

2 bucket-sort frequency of an element can be at most n

$\Rightarrow$ create n buckets $B_1, \dots, B_n$

perform linear occurrence count using hashmap $O(1)$

iterate through hashmap adding element to corresponding occurrence bucket $n \times O(1)$

read top k non-empty buckets from n to 1 $O(n)$

So runtime $\Theta(n)$ and space $\Theta(n)$.

not algo. design

16 encode-decode: (med) A is a list of n strings (UTF-8) with $n = \sum_{s \in A} \text{len}(s)$. Encode s into one string and then decode.

Naive: choose some non-(UTF-8) character as a delimiter and concat/split naively.

we want to not depend on this.

Idea: use some information about string lengths and delimit accordingly.

what if we had numbers representing string block lengths?

$\rightsquigarrow$ how to avoid confusing with numbers in the string?

"$\ell_1 \# \underbrace{\quad}_{s_1} \ell_2 \# s_2 \dots \ell_n \# s_n$" where # delimiter and $\ell_i = \text{len}(s_i)$

number of non-encoding characters before ℓ_{i+1}

17 sqrt-x: (easy) given $x \in \mathbb{N}_{\geq 0}$, compute sqrt(x) rounding down.

Binary Search: Consider $A = [0..x]$. Split A halfway to form A_L and A_R. Then $\sqrt{x} \in A_L$ or $\sqrt{x} \in A_R$.

If $A_R[0]^2 > x$ then $\sqrt{x} \in A_L \Rightarrow$ recurse. Say $n = x$.

else $\sqrt{x} \in A_R \Rightarrow$ recurse.

$\Rightarrow T(n) = T(\frac{n}{2}) + O(1) \in \Theta(\log n)$

18 peak-in-mountain: (med) given an integer "mountain" array (values increase, peak, then decrease) with one peak, find index of peak.

Naive: linear scan $\Theta(n)$.

D&C: peak must either be in A_L or A_R. We can reduce this to a max. search.

```
search-max(A):
    L_max = search-max(A_L)
    R_max = search-max(A_R)
    return max(L_max, R_max)
```

$\Rightarrow T(n) = 2T(\frac{n}{2}) + O(1) \in \Theta(\log n)$

19 h-index: (med) given citations array C (unsorted), where $C[i] = \#$ citations for i^{th} paper, return h-index of C.

Naive: sort and then linear search $\Rightarrow \Theta(n \log n)$ time. Many similar approaches have $\Theta(n \log n)$ time.

"Quick-select": can we modify quick-select for this problem?

choose pivot p randomly

$A_L < p$ and $A_R \geq p$ (want largest p such that $A_R.\text{size} \geq p$)

if $A_R.\text{size} < p$: (so $h < p$)

$p \leftarrow p/2$

retry with new p

does not seem like this would be any faster

Binary Search $h \in [0, n]$ so we can run a binary search with some subroutine is-h(k) that determines if A has k elements $\geq k$ $\Rightarrow \Theta(n \log n)$ binary search.

Notice $h \in [0, \min(\max(A), n)]$ so if $\max(A) \ll n$ then this can be even faster ($\Theta(n \log M)$, $M = \max(A)$).

Counting Sort: buckets B[i] where we have $M = \min(\max(A), n)$ buckets

⇒ add papers to buckets and identify h-index using a linear scan. $\Theta(n)$ time, $\Theta(n)$ space.

20 med **product-array-except-i** array of integers A, we want to return an array P such that $P[i] = \prod_{j \neq i} A[i]$.

Naive: Precompute $\prod_{j=0}^{n-1} A[i]$ ALL product. Then calculate $B[i] = p / A[i]$ where $p = \prod_j A[i]$ pre-computed.

Constraint: cannot use division.

Idea: compute forward and backward product arrays $F[i] = \prod_{j=0}^{i-1} A[j]$ and $B[i] = \prod_{j=n-1}^{i+1} A[j]$ (both $\Theta(n)$ by rolling window)

⇒ $P[i] = F[i]\,B[i]$ $\Theta(n)$ time, $\Theta(n)$ space

Notice $|P| \in \Theta(n)$ ⇒ our algorithms will always be $\Omega(n)$ so $\Theta(n)$ is optimal.

21 med **longest-consecutive-sequence** array A of integers, determine the length of the longest consecutive numeric sequence that can be formed using A.

Naive: $\Theta(n^2)$ checking, or Naive: sort on check, $\Theta(n \log n)$. We want $\Theta(n)$. ← but without radix-sort, that would be too easy

Hashing: Convert A so that we have a hashmap H where $H[a_i] = 1$.

Find $m = \min(A)$ and start at H[m]. Pop H[m] and look for H[m+1].

If $m+1 \in H$, pop and continue with m+2

else: continue to next element? * how can this be done efficiently though

Identify all potential sequence starts: for $a_i \in A$, check if $a_{i-1} \in H$. If no, then a_i potential start. (O(1))

Let S be the set of sequence starts

for $s \in S$: ($\Theta(n)$)

do above pop-and-find process to identify sequence length

⇒ get longest such sequence

$\Theta(n)$ time, $\Theta(n)$ space

22 med **edit-distance** word1 and word2, and 3 valid operations: insert character, delete character, edit character

find least number of operations to convert word1 → word2.

Idea: If $|word1| < |word2|$ then we must add ⇒ must find most optimal place to add

$|word1| > |word2|$ we must remove ⇒ find most optimal deletions

ops = 0

Look for chars of word2 in-order within word1

ops += number of missing in-order chars

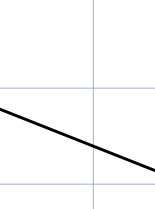
ops += length difference

Dynamic Programming Problem

cat, bat ⇒ 1+0 = 1 operation

cats, bat ⇒ 1+1 = 2 operations

intention
execution



PA1 Q1.1 bloom-day-bouquets (med) integer array A, and some $m, k \in \mathbb{Z}$.

We want to make m bouquets. To make a bouquet requires k adjacent flowers from a garden.

The garden has n flowers, and the i^{th} flower blooms at A[i] and can then be used in exactly one bouquet.

Find the minimum number of days to make m bouquets of size k, and return -1 if impossible.

(a) The largest output depends on the values in A. Consider A=[1, M] and m=2 and k=1.
Then the algorithm will return max(M, 1).

(b) Say can(d) is an $\Theta(n)$ subroutine that returns { True if can make m bouquets using k flowers on day d; False otherwise

We can binary search for $d \in [0, \overbrace{\max_{a \in A}(a)}^{M}]$

$\Rightarrow$ runtime $\Theta(T(\text{can}) \log M) = \Theta(n \log M)$.

1 Find $M = \max_{a \in A}(a)$ $\longrightarrow O(n)$

2 Set l, r = 0, M
s = (r-l)/2

$\log M \times$ {
while l≤r: → Θ(n)
if l==r and can(s) True:
return s
if can(s) True:
r=s → Θ(n)
else:
l=s
s = (r-l)/2
}

(c) can(d): need $\Theta(n)$ algorithm to decide whether we can make m k-bouquets on day d.

linear scan (?) keep a flower counter c, move linearly, reset if a flower not bloomed.

```
c=0   current flowers count
b=0   bouquet count
for i=0 to n-1:
    if A[i]≤d: c+=1
    else: c=0
    if c≥k:
        b+=1
        c=0
    if b≥m:
        return True
if b<m:
    return False
```

PA1 Q1.2 minimum-repair-time A: integers representing ranks of m mechanics.
mechanic with rank r can repair n cars in rn^2 minutes.
n = number of cars

largest possible output:
We have 1 mechanic of rank R ⇒ takes Rn^2 time to repair n cars. Make R arbitrarily large.

Say can(t) (→ Θ(n)) subroutine: whether n cars can be repaired by time t.

We can binary search $t \in [0, \max_{r \in A}(r)n^2]$ → problem: this might be large
this is okay because logarithm will absorb any exponents

Can(t):

```
cap=0   repair capacity by time t
for r in A:
    cap += ⌊t/(r·n²)⌋
    if cap ≥ n: return True
return False
```

```
    x = r // t        x = n²
    cap += floor(√x)   O(1) using quake sqrt
    if cap ≥ n:
        return True   } can make ≥ n cars by time t
return False
```

Binary search use subroutine to binary search $t \in [0, \max_{r \in A}(r)n^2]$.

$\Rightarrow$ runtime $\Theta(n \log(Rn^2)) = \Theta(n \log R + 2n \log n) = \Theta(n \log(nR))$

Randomized Algorithms We study more randomized algorithms beyond CS240.

Freivald's Algorithm Given three $n \times n$ matrices $A, B, C \in \mathbb{R}^{n \times n}$, we want to check whether $AB = C$.

Naive: Compute AB using Strassen's algorithm and check if $AB = C$ elementwise $\Rightarrow O(n^{2.81})$ time.

Idea: Use randomization to reduce the problem to a simpler equality checking.

Choose $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$ randomly such that $v_i \in \{0, 1\}$ for each i. ← e.g. $v_i = \begin{cases} 0 \text{ with prob. } \frac{1}{2} \\ 1 \text{ with prob. } \frac{1}{2} \end{cases}$

Check if $ABv = Cv$ (notice $\underset{n \times n}{A} \underset{n \times 1}{B} v \in \mathbb{R}^n$ and $Cv \in \mathbb{R}^n$) $\rightsquigarrow O(n^2)$ time with matrix-vector mult.

$ABv = Cv \Rightarrow$ predict $AB = C$
$ABv \neq Cv \Rightarrow AB \neq C$ deterministically

} notice $ABv = (AB)v = A(Bv)$ so AB need not be computed. instead we can check if $ABv - Cv = 0$ using $\underbrace{\underbrace{A(Bv)}_{O(n^2)} - \underbrace{Cv}_{O(n^2)}}_{O(n^2)} = 0$ in $O(n^2)$.

The non-trivial case is when $AB \neq C$ but $ABv = Cv$ which we must analyze

Correctness Analysis: Consider $D = (AB - C)$ so $AB = C \Rightarrow \ker(D) \supseteq \mathbb{R}^n$. Denote $Dv = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$.

If $\begin{pmatrix} u_1 \\ i \\ u_n \end{pmatrix} = Dv = 0 \in \mathbb{R}^n$ but $AB \neq C$, then Freivalds produces an incorrect answer.

Freivald's produces an incorrect answer $\Rightarrow ABv = C$ but $AB \neq C$.

so Freivald's incorrect $\Leftrightarrow ABv = Cv$ but $AB \neq C \Leftrightarrow v \in \ker(D)$ but $D \neq 0 \in \mathbb{R}^{n \times n}$

$\Rightarrow \exists (i, j)$ such that $D_{ij} \neq 0$. Notice $u_i = D_{i1}v_1 + D_{i2}v_2 + \cdots + D_{ij}v_j + \ldots + D_{in}v_n = x + D_{ij}v_j$.

and $Dv = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = 0 \in \mathbb{R}^n \Rightarrow u_i = x + D_{ij}v_j = 0$ with $D_{ij} \neq 0$.

Once we reveal our random choices $\{v_1, \ldots, v_n\} \setminus \{v_j\}$, term x is fixed. (Principle of Deferred Decision)

Notice at most one choice between $v_j = \{^0_1$ can yield $u_i = 0$ since $D_{ij} \neq 0$.

This choice has probability $\frac{1}{2} \Rightarrow \Pr(u_i \neq 0) \geq \frac{1}{2}$

$\Rightarrow \Pr(\text{Freivald's correctness}) \geq \frac{1}{2}$.

$\rightsquigarrow$ amplify success probability by repeating k times independently. Let $X_1, \ldots, X_k$ be results.

Then $P(\underbrace{X_1 \text{ failure}, X_2 \text{ failure}, \ldots, X_k \text{ failure}}) = \prod_{i=1}^{k} P(X_i \text{ failure}) \leq \prod_{i=1}^{k} \frac{1}{2} = \frac{1}{2^k}$

failure: algorithm produces wrong output (i.e. false positive)

$\Rightarrow P(\text{correctness}) \geq 1 - \frac{1}{2^k}$.

Therefore Freivalds has $\Theta(kn^2)$ time with $\geq 1 - \frac{1}{2^k}$ success probability.

Coupon Collector Problem There are n types of coupons. Each box of cereal contains one random coupon.

How many boxes of cereal to collect all n coupons?

≡ reframe

n balls randomly placed in n bins, what is $P(\text{every bin has} \geq 1 \text{ ball})$?

Consider one bin B_i. Notice $B_i \sim \text{Binomial}(n, \frac{1}{n})$ so $P(B_i = 0) = \underbrace{\left(1 - \frac{1}{n}\right)^m}_{\text{failure prob.}} = \underbrace{\left(\left(1 - \frac{1}{n}\right)^n\right)^{\frac{m}{n}}}_{1 + x \leq e^x} \leq e^{-m/n}$

$\Rightarrow 1 - x \leq e^{-x}$ for $x \leq 1$
$\Rightarrow (1-x)^m \leq e^{-xm}$

$$\Rightarrow P(B_1 = 0 \text{ or } B_2 = 0 \text{ or } \ldots \text{ or } B_n = 0) = P\left(\bigcup_{i=1}^{n} (B_i = 0)\right)$$
$$= \sum_{i=1}^{n} P(B_i = 0) \leq n\left(1 - \frac{1}{n}\right)^m \leq ne^{-\frac{m}{n}}$$

so $m = 2n\lceil \ln n \rceil \in \Theta(n \log n)$ boxes guarantees $P(\text{collect } n \text{ types}) \geq 1 - \frac{1}{n}$.

We introduce a useful result for calculating expected runtime and success probability.

Lemma (Markov's Inequality): Let X be a non-negative random variable and $a>0$. Then $P(X \geq aE[X]) \leq \frac{1}{a}$.

Proof: By definition $E[X] = \int_{\mathbb{R}} xf(x)\,dx \overset{X \text{ non-neg.}}{=\!=} \int_{[0,\infty)} xf(x)\,dx$ so we have

$$E[X] = \underbrace{\int_{[0,a]} xf(x)\,dx}_{\geq 0} + \int_{[a,\infty)} xf(x)\,dx \geq \int_{[a,\infty)} xf(x)\,dx \geq \min_{x\in[a,\infty)}(x) \int_{[a,\infty)} f(x)\,dx = a\int_{[a,\infty)} f(x)\,dx$$
$$\overset{\text{def.}}{=} aP(X \geq a).$$

Thus $E[X] \geq aP(X\geq a) \Rightarrow \frac{E[X]}{a} \geq P(X \geq a)$.

Using $b = \frac{a}{E[X]} > 0$ gives $\frac{E[X]}{a} = \underbrace{\frac{1}{b} \geq P(X \geq bE[X]) = P(X \geq a)}_{\text{done}}$. □

This is very useful in converting expected runtime to probabilistic runtime.

$P(T \geq aE[T]) \leq \frac{1}{a}$ where T is runtime.

$\Rightarrow P(T \geq 100E[T]) \leq \frac{1}{100}$. In particular, there is an upper bound on the probability of "bad" T.

If expected runtime $E[T] \in \Theta(n\log n)$ then $P(T \geq 100\,\Theta(n\log n)) \leq 0.01$

$\Rightarrow P(T < 100\,\Theta(n\log n)) \geq 0.99$ so $P(T \in \Theta(n\log n)) \geq 0.99$.

In general, this gives rise to two types of randomized algorithms:

Las Vegas algorithms: $P(\text{correctness}) = 1$ but time complexity guarantee is only in expectation.
randomized-quick-sort
i.e. we have $E[T] \in O(f(n))$

Monte Carlo algorithms: $P(\text{correctness}) \leq 1$ but time complexity guarantee always holds.
Freivald's algorithm
i.e. we have $T \in O(f(n))$

Lemma: All Las Vegas algorithms are Monte Carlo algorithms.

Proof: Convert using Markov's inequality. Let $\mathcal{A}$ be a Las Vegas algorithm.

$\overset{\text{def.}}{\Rightarrow} E[T(\mathcal{A})] \in O(f(n))$ and $P(\mathcal{A} \text{ correct}) = 1$

so we have $\frac{1}{a} \overset{\text{Markov inequality}}{\leq} P(T(\mathcal{A}) \geq a\underbrace{E[T(\mathcal{A})]}_{\leq cf(n)}) \leq P(T(\mathcal{A}) \geq acf(n))$

$\Rightarrow P(T(\mathcal{A}) \in O(f(n))) \geq 1 - \frac{1}{a}$.

Modify $\mathcal{A}$ by setting a time limit $a = 100E[T(\mathcal{A})]$ so we have a new $\mathcal{A}'$ with $P(\mathcal{A}' \text{ correct}) \geq 1 - \frac{1}{a}$ and $P(T(\mathcal{A}') \in O(f(n))) = 1$ so $\mathcal{A}'$ is Monte Carlo. □

from STAT 240 Notes

Proposition (Markov's Inequality): For any random variable X with $E[|X|^k] < \infty$, we have $P[|X| \geq c] \leq \frac{E[|X|^k]}{c^k}$ for all $c>0$, for all $k \in \mathbb{Z}_{>0}$.

Proof: Say X_i are continuous. Similar proof for X_i discrete.

We have $\frac{E[|X|^k]}{c^k} = \frac{1}{c^k}\int_{-\infty}^{\infty} |x|^k f_X(x)\,dx$ and $\mathbb{R} = \{|x| \geq c\} \cup \{|x| < c\}$:

$$= \frac{1}{c^k}\left(\int_{\{|x|\geq c\}} |x|^k f_X(x)\,dx + \underbrace{\int_{\{|x|<c\}} |x|^k f_X(x)\,dx}_{\geq 0}\right)$$

definition

$$\geq \frac{1}{c^k}\int_{\{|x|\geq c\}} |x|^k f_X(x)\,dx \geq \frac{1}{c^k}\underbrace{\min_{x\in\{|x|\geq c\}} |x|^k}_{x=c}\int_{\{|x|\geq c\}} f_X(x)\,dx = \int_{\{|x|\geq c\}} f_X(x)\,dx = P(|X|\geq c). \quad \square$$

Designing Random Algorithms

Reservoir Sampling

We are given some collection $(a_1,\dots,a_n)$ where n is unknown apriori, and want to return a value uniformly at random.

We can use a strategy called reservoir sampling: (eg. when sampling from a linked list)

```
ans = None
for a_i in (a_1,..,a_n):
    prob = 1/i
    r_i ~ Bernoulli(prob)
    if r_i:
        ans = a_i
return ans
```

This method gives $P(ans = a_i) = \frac{1}{n}$ at termination.

Notice $P(ans=a_i) = P(r_i=1, \underbrace{r_{i+1}=0,\dots,r_n=0}_{\text{future states}})$

$$\overset{\text{indep}}{=} P(r_i=1)\prod_{k=i+1}^{n} P(r_k=0)$$

$$= \frac{1}{i}\prod_{k=i+1}^{n}\underbrace{\left(1-\frac{1}{k}\right)}_{\frac{k-1}{k}} = \frac{1}{i}\left(\frac{i}{i+1}\right)\left(\frac{i+1}{i+2}\right)\cdots\left(\frac{n-1}{n}\right) = \frac{1}{n}.$$

Fisher-Yates Shuffle

We have an array $A=[a_0,\dots,a_{n-1}]$ and want to shuffle A such that any permutation is equally likely.

This is useful to generate random permutations of $[1,\dots,n]$ for arbitrary n.

```
for (i=n-1; i=0, i--):
    j ← randint({i,...,n-1})
    swap A[i] and A[j]
```

$\Rightarrow$ we shuffle in $O(n)$ with $O(1)$ auxiliary space (optimal).

It remains to show that the algorithm is correct.

Proposition (Fisher-Yates): The Fisher-Yates shuffle on $A=[1..N]$ is a random variable X such that

$$P(X=B_N) = \frac{1}{|\mathcal{I}_N|} = \frac{1}{N!} \text{ for all } B_N\in\mathcal{I}_N.$$

Proof: WLOG start with $A=[1,\dots,N]$. Consider arbitrary $B=[b_1,\dots,b_N]\in\mathcal{I}_N$.

Then $P(\bigcap_{i=1}^{N}\{b_i=i\}) \overset{\text{indep}}{=} \prod_{i=1}^{N} P(b_i=i \mid \bigcup_{k=0}^{i-1}\{b_k=k\})$ by conditional probability.

For the first random swap, clearly a_1 is only touched once (when $i=0$ in the algorithm)

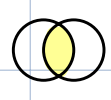
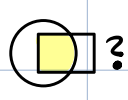
$\Rightarrow P(b_1=1)=\frac{1}{N}$. Clearly $P(b_2=2 \mid b_1=1) = \frac{1}{N-1}$ ($N-1$ spots left). Inductively, $P(b_i=i \mid \underbrace{b_{i-1}=i-1,\dots,b_1=1}_{i-1 \text{ spots filled}}) = \frac{1}{N-i+1}$.

$$\Rightarrow P(B=A) = P(\bigcap_{i=1}^{N}\{b_i=i\}) \overset{\text{indep}}{=} \prod_{i=1}^{N} P(b_i=i \mid \bigcup_{k=0}^{i-1}\{b_k=k\}) = \prod_{i=1}^{N}\frac{1}{N-(i-1)} = \prod_{i=0}^{N-1}\frac{1}{N-i} = \frac{1}{N!} \text{ as required. } \square$$

Thus the Fisher-Yates shuffle is correct.

Random Point Generation

Say we want to generate a (uniformly) random point in a circle of radius r centered at 0. There are two main approaches:

1. Generate (r,θ) pair. Notice we want the density of generated points to be uniform in the circle, so we sample $x\sim \text{Unif}(0,\sqrt{r})$ and $\theta\sim\text{Unif}(0,2\pi)$ so (x^2,θ) defines a random point.
2. What if we want some random point in a slightly more complex region?  or  ?

   We can use randomization.

Idea: Define a simple cover for our target region R (e.g. cover with a rectangle $[x_1, x_2] \times [y_1, y_2]$.

Sample $p \sim [x_1, x_2] \times [y_1, y_2]$ uniformly. If $p \in R$, return p. Else, retry.

---

target region R, area A_R
bounding box $B = [x_1, x_2] \times [y_1, y_2]$, area A_B
while True:
    sample $x \sim B$ uniformly
    if $x \in R$:
        return x } probability $\frac{A_R}{A_B}$

Therefore our runtime depends on the success $p = \frac{A_R}{A_B} \leq 1$.

Notice $E[\underbrace{T}_{\text{runtime}}] = \sum_{t=1}^{\infty} t P(T=t)$

$= \sum_{t=1}^{\infty} t P(x_t \in R, x_{t-1} \notin R, \dots, x_1 \notin R)$

$\overset{\text{indep}}{=} \sum_{t=1}^{\infty} t P(x_t \in R) \prod_{k=1}^{t-1} P(x_k \notin R)$

$= \sum_{t=1}^{\infty} t \frac{A_R}{A_B} \left(1 - \frac{A_R}{A_B}\right)^{t-1} = L$ converges.

Example: For instance, say R unit circle and B bounding square, so

$E[T] = \sum_{t=1}^{\infty} t \frac{\pi}{4}\left(1-\frac{\pi}{4}\right)^{t-1} = \frac{\pi}{4}\sum_{t=1}^{\infty} t\left(\frac{4-\pi}{4}\right)^{t-1} = \underbrace{\frac{\pi}{4}\sum_{t=1}^{\infty}(t-1)\left(\frac{4-\pi}{4}\right)^{t-1}}_{\frac{\pi}{4}\sum_{t=1}^{\infty} t\left(\frac{4-\pi}{4}\right)^{t} = \frac{\left(\frac{4-\pi}{4}\right)}{\left(1-\left(\frac{4-\pi}{4}\right)\right)^2}} + \underbrace{\frac{\pi}{4}\sum_{t=1}^{\infty}\left(\frac{4-\pi}{4}\right)^{t-1}}_{\frac{\pi}{4}\left(\frac{1}{1-\left(\frac{4-\pi}{4}\right)}\right)} \leq L \in \mathbb{R}.$

$\Rightarrow E[T] \in O(1)$ expected.

Moderately Small Element

Recall earlier, we saw a deterministic median-finding algorithm that runs in $O(n)$.
We also know that smallest-element search takes $\Omega(n)$.

→ can we find a moderately small element in $o(n)$?

Let A be an array of n integers. We say $a \in A$ is moderately small if $\text{rank}(A) \leq \frac{1}{10}$.

Notice $P(a \in A \text{ moderately small}) = P(\text{rank}(a) \leq \frac{n}{10}) = \frac{1}{10}$ for uniformly random $a \in A$.

Choose some set of indices $i_1, \dots, i_s \in \{0, \dots, n-1\}$ randomly with replacement, and returning $\min_k(a_{i_k})$

gives $P(\min_k(a_{i_k}) \text{ moderately small}) = P(\bigcup_{k=1}^{s} \{a_{i_k} \text{ moderately small}\})$ ← at least one is moderately small

$= 1 - P(\bigcap_{k=1}^{s} \{a_{i_k} \text{ not moderately small}\})$

$\overset{\text{indep}}{=} 1 - \prod_{k=1}^{s} P(a_{i_k} \text{ not moderately small}) = 1 - \left(1 - \frac{1}{10}\right)^s$

Say we want our algorithm to have a success rate $P(\text{success}) \geq 1 - \frac{1}{n}$.

$\Rightarrow$ we can use randomization and require $1 - \left(1 - \frac{1}{10}\right)^s \geq 1 - \frac{1}{n}$ so $\left(1 - \frac{1}{10}\right)^s \leq \frac{1}{n}$

$\Leftrightarrow \left(\frac{9}{10}\right)^s \leq \frac{1}{n} \Leftrightarrow s \log\left(\frac{9}{10}\right) \leq -\log n \Leftrightarrow s \log\left(\frac{10}{9}\right) \geq \log n \Leftrightarrow s \geq \log\left(\frac{9}{10}\right) \log n$

So choose $s = C \log n \in \Theta(\log n)$. Selecting $A[0:s]$ and returning $\min(A[0:s])$ has $P(\text{success}) \geq 1 - \frac{1}{n}$.

$\Rightarrow \Theta(\log n)$ runtime. We can randomize subarray selection and repeat to amplify success rate.

Graph Algorithms I

We study some simple graph searching algorithms. We start with **breadth-first search** (BFS) and **depth-first search** (DFS).

Let $G=(V, E)$ be an undirected graph.

From CS135/136, we know there are many representations of a graph symbolically.

We will use the convention $n=|V|$, $m=|E|$.

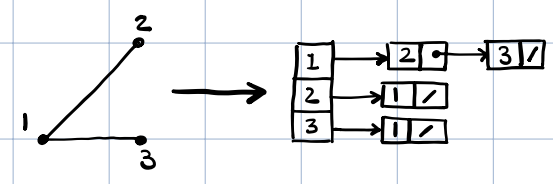
Adjacency matrix: represent vertices as rows/cols of an $n \times n$ matrix.

For G unweighted (edges do not have an associated weight), we can define $A \in F^{n \times n}$ by $a_{ij} = \begin{cases} 1 \text{ if } ij, ji \in E \\ 0 \text{ if } ij \notin E \end{cases}$ (undirected $\Rightarrow$ both same)

If G is weighted with $w_{ij} \in \mathbb{R}_{\geq 0}$ for all $i,j \in V$, then define $a_{ij} = \begin{cases} w_{ij} \text{ if } ij \in E \\ \infty \text{ if } ij \notin E \end{cases}$

$\Rightarrow \Theta(n^2)$ space.

Adjacency list: each vertex maintains a linked list of its neighbours.



$\Rightarrow |V| + \sum_{v \in V} \deg(v) = 2|E| + |V| \in \Theta(m+n)$ space.

we will frequently default to this representation

Some basic questions, given a graph G, are:

1) Is G connected?
2) What are the connected components of G?
3) Are some $u, v \in V$ connected? Find a shortest uv-path.

★ Graph BFS

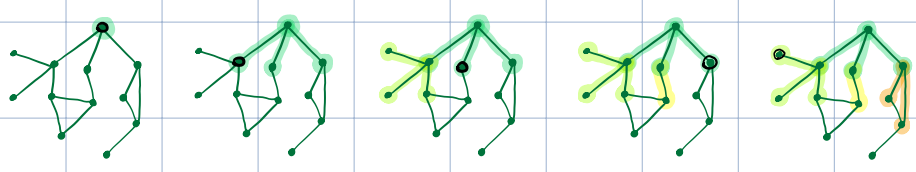
search through neighbours before depth

Input: graph $G=(V, E)$, source $s \in V$. Output: all vertices reachable by s (i.e. its component).

```
visited[v] = false ∀v∈V
queue Q empty
enqueue(Q, s), visited[s] = true
while Q ≠ ∅:
    u = dequeue(Q)
    for each neighbour v of u:
        if visited[v] = false:
            enqueue(Q, v)
            visited[v] = true
```

(how we know which vertex to search next in our BFS.)

(breadth-first)



Time: each $v \in V$ enqueued at most once.
when $v \in V$ dequeued, the loop executes $\deg(v)$ times.
$\Rightarrow O(n + \sum_{v \in V} \deg(v)) = O(n+m)$ time

Lemma: For $v \in V$, $\exists sv$-path in $G \iff$ visited[v] = true at the end of the algorithm.

Proof: ($\Leftarrow$) Induction on #steps of algorithm. We show visited[v] = true $\Rightarrow \exists sv$-path.

Base case step 0, when only visited[s] = true.

Suppose visited[v] is set to true at step k (within the for-loop for $u \in V$).

$\Rightarrow u \in V$ was dequeued $\Rightarrow u$ was enqueued at some previous step $\Rightarrow$ visited[u] = true

$\overset{IH}{\Rightarrow} \exists su$-path in G. Since v is added in the for-loop for u, we have $uv \in E$.

Extend su-path by $uv \Rightarrow \exists sv$-path. Done by induction. $\triangle$

($\Rightarrow$) Let $U = \{v \in V \mid$ visited[v] = true$\}$ (by algo. end). We want to show that s and $V \setminus U$ are disconnected.

Notice $\text{cut}_U(V \setminus U) = \emptyset$ since otherwise, some endpoint in $V \setminus U$ would have been enqueued

$\Rightarrow$ U and $V \setminus U$ are disconnected components $\Rightarrow$ $s \in U$ and $v \setminus U$ disconnected. $\triangle \square$

Correctness: Lemma above. Thus we see that this basic version of BFS can be used to answer:

- whether G is connected (check visited[v] = true $\forall v \in V$);
- what the component containing s is;
- whether $\exists sv$-path (check visited[v] = true).

* Exercise: Find all components of G in $O(m+n)$ time.

Path Backtracking (BFS):

add an array parent[v] that tracks which vertex enqueued v

$\Rightarrow$ parent[v] appears directly before v in the path found. We just need to keep finding vertex parents from the array to backtrack.

BFS Tree: (→ NOT unique!) Let H_s be the component of s.

Lemma: The graph formed by edges $(v, \text{parent}[v])$ is a spanning tree of H.

Proof: Exercise (easy). Show acyclic.

BFS finds shortest paths. This is a major feature of the algorithm.

(← relatively intuitive result) We can modify the BFS algorithm to compute path distances.

interesting

* Check Bipartedness (BFS):

Given source $s \in V$ in a connected graph G, notice if $s \in A$ then $N(s) \subseteq B$ and $N(N(s)) \subseteq A$, etc.

$\Rightarrow$ split V by path length parity from $s \in V$. Put even distances in A and odd in B.

$A = \{v \in V \mid \text{dist}(v) \text{ even}\}$, $B = \{v \in V \mid \text{dist}(v) \text{ odd}\}$
perform BFS starting at $s \in A$,
during BFS check if edge has endpoints in distinct partitions

---

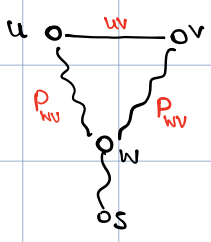
$\Rightarrow$ time $O(m+n)$ (same as BFS)

Correctness: Clearly correct when G bipartite. (true) How about when the algo. returns false?

Proof (Sketch): G bipartite $\Leftrightarrow$ G has no odd cycles.

Idea: algo. says non-bipartite $\Rightarrow$ there is an odd cycle in G.

WLOG say $\exists u, v \in A$ with $uv \in E$ (since algo. finds one).



Consider BFS tree T. Let $w \in V(T)$ be the lowest common ancestor of u, v in T. (exists because $s \in V$ is definitely one)

$\Rightarrow$ paths P_{wu} and P_{wv} are even length

$\Rightarrow$ cycle $P_{wu}\, uv\, P_{vw}$ odd length. $\square$

Remark: above is also an algorithmic proof of bipartiteness characterization.
also gives a linear-time algo. to find an odd cycle of an undirected graph.

* Graph DFS

In some situations, a DFS is more natural than a BFS. Consider solving a maze.

maze ↦ graph where squares of the maze are vertices, and $uv \in E \Leftrightarrow$ squares u and v are one step apart.

Finding a solution is equivalent to finding some st-path.

Moving back/forth makes a BFS inefficient. We want to explore a path until a dead end
⇝ backtrack to last "good" vertex
⇝ try new path

analogy: trace chalk as you solve maze ⇒ essentially DFS
→ dead-end → go back and only explore new paths

Input: undirected graph $G=(V,E)$, source $s \in V$. **Output:** all $v \in V$ reachable from s

← recursive

```
visited[v] = false ∀v∈V
visited[s] = true, explore(s)

explore(u):
  for each neighbour v of u:
    if visited[v] = false:
      visited[v] = true, explore(v)
```

Time: explore() is called at most once for each $v \in V$, with deg(v) subcalls

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

→ We can also write a DFS non-recursively using a stack

⇒ BFS and DFS are identical, except one uses a queue and the other uses a stack

(two fundamental search methods correspond to two fundamental data structures)

All basic results (e.g. connectivity) hold for DFS.

But we will see that DFS can solve some interesting problems that BFS cannot.

* DFS Tree: (important) → NOT unique!

Constructed in the same way as for BFS (using parent[v] relations).
Notice DFS trees may be different based on neighbour exploration order.

We induce tree ordering starting at root $s \in T$ and define regular terminology as in MATH 239.

A non-tree edge uv is called a back edge if either u is an ancestor or descendant of v in G.

Property (back edges): In an undirected graph, all non-tree edges are back edges.

Proof: SFAC $\exists uv \in E$ but uv is not an ancestor-descendant pair.

WLOG say u is visited before v.

$uv \in E \Rightarrow$ v explored before u is finished $\Rightarrow$ u is an ancestor of v. ↯ □

Starting Time and Finishing Time

We record the time when a vertex is first visited and the time when its exploration is finished.

→ information important in the design and analysis of algorithms.

We modify our algos. to include this:

```
visited[v] = false ∀v∈V
time = 1, visited[s] = true, explore(s)

explore(u):
  start[u] = time
  time ← time+1
```

Property (parenthesis): Let $u, v \in V$. Then either:

- One of [start[u], finish[u]], [start[v], finish[v]] is contained in the other
- [start[u], finish[u]] ∩ [start[v], finish[v]] = ∅

and the former occurs precisely when u,v are ancestor-descendant pair.

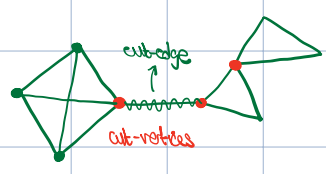
```
for v∈N(u):
    if visited[v] = false:
        visited[v] = true
        explore(v)
finish[u] = time
time ← time+1
```

Remark: we can generalize DFS on non-connected graphs.
In this case, DFS would generate a forest.

Let G graph. In MATH 239, we defined when $e \in E$ was a **bridge**: when $comp(G-e) > comp(G)$.

Similarly, we say $e \in E$ is a **cut-edge** if $comp(G-e) > comp(G)$, and

$v \in V$ is a **cut-vertex** if $comp(G-v) > comp(G)$.

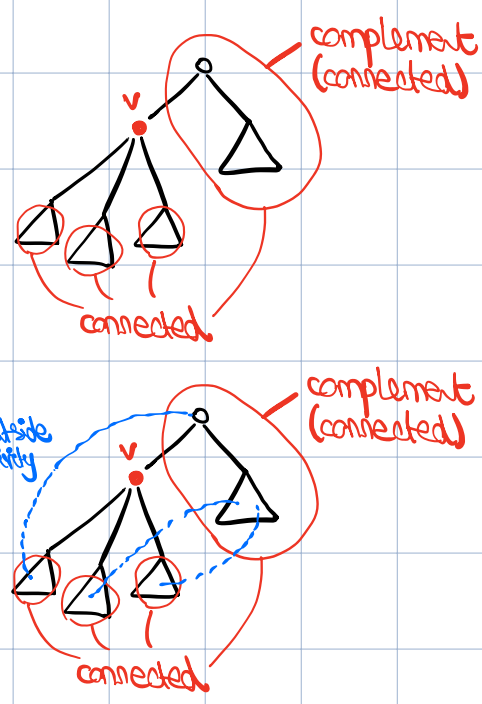


Idea: use a DFS tree to identify all cut-vertices and cut-edges.

Consider DFS tree $T \subseteq G$, and non-root $v \in V$. We want to know if v is a cut-vertex of G.

Recall Property (back-edges) $\Rightarrow$ all non-tree edges are back-edges.

$\Rightarrow$ the only way for a subtree below v to be connected outside that subtree is to have edges going to some ancestor of v.



Claim (cut-vertex): A subtree T_i below v is a connected component in $G-v$ $\iff$ $\exists e = uv \in T_i$ with one endpoint in T_i and another endpoint in a (strict) ancestor of v.

Proof: ($\Leftarrow$) all non-tree edges are back-edges $\Rightarrow$ there are no edges going to another subtree below v, nor edges going to another subtree of the root. $\triangle$

($\Rightarrow$) If $\exists e \in E$ with one endpoint in T_i and another in a (strict) ancestor of v, then T_i and the complement are still connected $\Rightarrow$ T_i not a component $\lightning$ $\triangle\square$

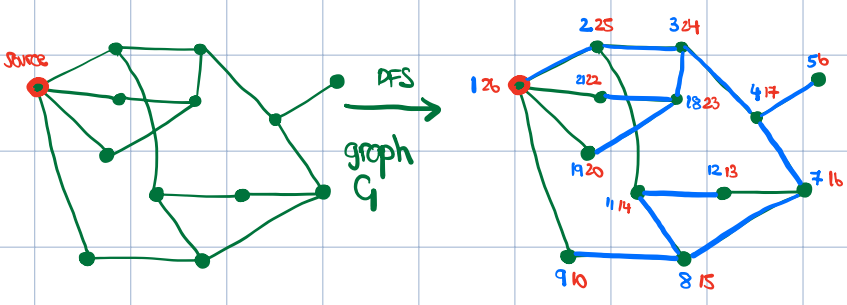
↳ components are maximally connected

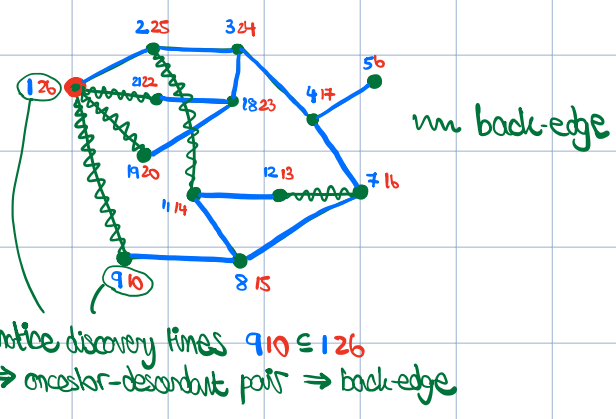
The same argument applies to all subtrees below v, so we get this characterization:

Lemma: For a non-root vertex v in a DFS tree,

v is a cut vertex $\iff$ there is a subtree below v (we only care about the subtrees directly below v) with no edges going to a strict ancestor of v.

Ex.





notice discovery times $9|10 \subseteq 1|26$
$\Rightarrow$ ancestor-descendant pair $\Rightarrow$ back-edge

Let T be a DFS forest.

In general, we consider 4 edge classes on directed G:

tree edges are $e \in E(T)$
back edges are edges that connect $v \in V$ to an ancestor
forward edges are edges that connect $v \in V$ to a descendant
cross edges are all other edges.

} where direction is implied: in undirected G, we call both classes back-edges

Proof (Lemma): ($\Rightarrow$) Say v is a cut vertex. SFAC every subtree below v has an edge going to an ancestor of v.

Then every subtree of v is connected to the complement structure $\Rightarrow$ $comp(G-v) = comp(G)$

$\Rightarrow$ v not cut vertex. ↯

Thus there must be at least one subtree below v with no edge going to an ancestor of v. △

($\Leftarrow$) Say some subtree T_i below v has no edges going to an ancestor of v

↓ Conn. $\Rightarrow$ T_i will be a connected component of G-v $\Rightarrow$ v is a cut vertex. △□

↓ see above

*Ex: extend this result to the root vertex:

Lemma: For root v of a DFS tree, v is a cut vertex $\Leftrightarrow$ v has at least 2 children.

Proof: (contrapos) Say v has 2 children in DFS tree. Say rooted subtrees of these children are T_1, T_2.

Since we search depth-first, WLOG, say T_1 generated first.

Notice T_2 created from v $\Rightarrow$ no exploration of T_1 brought us to T_2 (else T_2 not a subtree of v)

$\Rightarrow$ T_2 unreachable from T_1 in G-v $\Rightarrow$ v is a cut vertex. "□"

With these lemmas, we know how to determine if a vertex is a cut vertex by looking at a DFS tree.

Algorithm: find all cut vertices $O(n+m)$

Idea: process the vertices of a DFS following a bottom-up ordering, keep track of how "far up" the back edges of a subtree can go (i.e. how close to the root these back edges go).

$\Rightarrow$ use Lemma: for non-root v, we have v not a cut vertex $\Leftrightarrow$ all subtrees below v has an edge that goes above v.

How can we keep track of "how far up" we can go?
use starting/finishing time since descendant time $\subseteq$ ancestor time.

In particular, des. start $\geq$ anc. start

Define early[v] for each v in our DFS tree:

$$early[v] = \min\{\underbrace{start[v]}_{\text{current start}}, \underbrace{\min\{start[w] \mid uw \text{ is a back edge with } u \text{ a descendant of } v, \text{ or } u=v\}}_{\text{earliest start time among all vertices connected to somewhere below v through back edge}}\}$$

if this w has start[w] < start[v] then w ancestor of v $\Rightarrow$ found edge that goes above v in rooted subtree

Informally, early[v] records how far up we can go from the subtree rooted at v.

Suffices to prove/show: ① We can compute early[v] for all $v \in V$ in $O(n+m)$ time

② We can identify all cut vertices using early[v] in $O(n+m)$ time

Computing early[v]

① We compute early[v] values from leaves to root in the DFS tree.

Proof: (idea) Base case: $v \in V$ is a leaf of the DFS tree. We want to compute early[v].

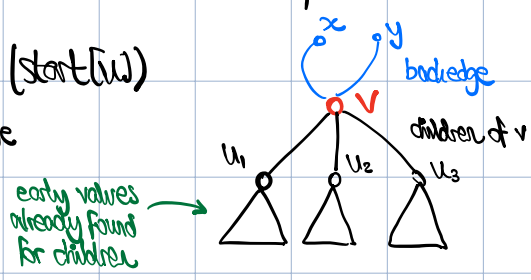
$\rightsquigarrow$ consider all the edges incident on v (so say $uv \in V$), and take

$early[v] = \min(start[v], \min_{uv \in E}(start[u]))$ in $O(\deg(v))$ time (base case)

Inductively suppose, for $v \in V$, all early[u] values for children of v has been computed.

$\Rightarrow$ early[v] = take minimum between $\min_{u \text{ child}}(early[u])$ and $\min_{uv \in E \text{ backedge}}(start[u])$

$= \min(\underbrace{\min_{u \text{ child}}(early[u])}_{\subseteq \text{ edges of } v}, \underbrace{\min_{uv \in E \text{ backedge}}(start[u])}_{\subseteq \text{ edges of } v})$ $O(\deg(v))$ time



early values of children correct $\Rightarrow$ early[v] correct * can be proven inductively

bottom-up ordering $\Rightarrow$ every $v \in V$ only computed once

early[v] = min{early[u₂], start[x], start[y]}

so total time complexity is $O(n+\underbrace{\sum_{v\in V}\deg(v)}_{2|E|}) = O(n+m)$. □

finding cuts

② To check if non-root $v\in V$ is a cut vertex, we need to check whether $early[u_i] < start[v]$ for all children u_i of v.

Proof: (Idea) $early[u_i] < start[v]$ for all u_i (u_i child) $\Rightarrow$ v not a cut vertex

→ i.e. we could have gone a different way from above v to each subtree.

since all subtrees below v have a back edge going above v.

Otherwise if $\exists u_i$ child with $early[u_i] \geq start[v]$ then u_i-rooted subtree will be a connected component of $G-v$ $\Rightarrow$ v cut vertex.

Handle root vertex separately. Convince yourself this is $O(n+m)$. □

* Exercise Extend algorithm to find cut edges of a graph G.

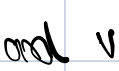
Remark: We looked at (undirected) graphs G and what BFS/DFS can do.

A highlight is very clever algorithm to detect strongly connected components in linear time.

Now we look at directed graph generalizations and algorithms

Directed Graph Algorithms

A directed graph is a graph G where $e\in E$ has an order property (in particular, $uv \neq vu$).

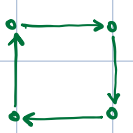
If $uv\in E$, we call u the tail and v the head of the edge. 

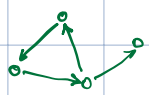
We define $indeg(v)$ and $outdeg(v)$ accordingly.

⇝ useful in modeling asymmetric relations. We are interested in connectedness properties of directed G.

G directed graph, $s,t\in V$. We say t is reachable from s if $\exists$ directed st-path in G. 

We say G is strongly connected if for all pairs $u,v\in V$, u reachable from v and v reachable from u.

 strongly connected

 not strongly connected

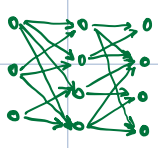
A subset $S\subseteq V$ is strongly connected similarly.

$S\subseteq V$ is a strongly connected component if S is a maximally strongly connected subset (i.e. $S\subseteq V$ strongly connected and $S+v$ not strongly connected for any $v\in V\setminus S$).

G directed is a directed acyclic graph if G has no directed cycles.

Remark G undirected and G acyclic $\Rightarrow$ $|E|$ is bounded.

The same does not hold for directed graphs because of edge direction property.

 acyclic directed graph

We want to design algorithms to answer basic questions about directed graphs:

1 Is G strongly connected? 2 Is G directed acyclic? 3 Find all strongly connected components of G.

Turns out there are $O(n+m)$ algorithms for these, but they are more complicated than those for undirected graphs.

Graph Representations

Adjacency Matrix Extends naturally to directed graphs: $A_{ij} = \begin{cases} 1 \text{ if } ij \in E \\ 0 \text{ otherwise} \end{cases}$ ← note $ij \neq ji$ (directed)

Finish directed graphs

Single-Source Shortest Paths

We study Dijkstra's algorithm for shortest paths from source vertex to all other vertices.

Shortest Paths **Input:** A directed graph $G=(V,E)$ with a weight $w_e \in \mathbb{R}_{\geq 0}$ for each $e \in E$, and two arbitrary vertices $s, t \in V$ (s: source, t: target)

Output: A shortest st-path P, where $\text{length}(P) = \sum_{e \in P} w_e$.

It will be more convenient to solve a more general problem.

Shortest single-source path algs.

Input: Directed $G=(V,E)$ with $w_e \in \mathbb{R}_{\geq 0}$ for $e \in E$, and some source $s \in V$.

Output: A shortest sv-path, for all $v \in V$.

This may seem harder, since each path could have $\Omega(n)$ edges, and so output size could be $\Omega(n^2)$.

↝ turns out that there is a succint representation of paths, and the single-source shortest path can be solved in the same complexity as shortest st-path problem.

Dijkstra's Algorithm

We have seen that BFS can be used to solve the single-source shortest path problem, where every edge has the same weight. ← so we essentially minimize the # edges in the path

We can reduce the non-constant $w_e \in \mathbb{R}_{\geq 0}$ case to the same-weight special case.

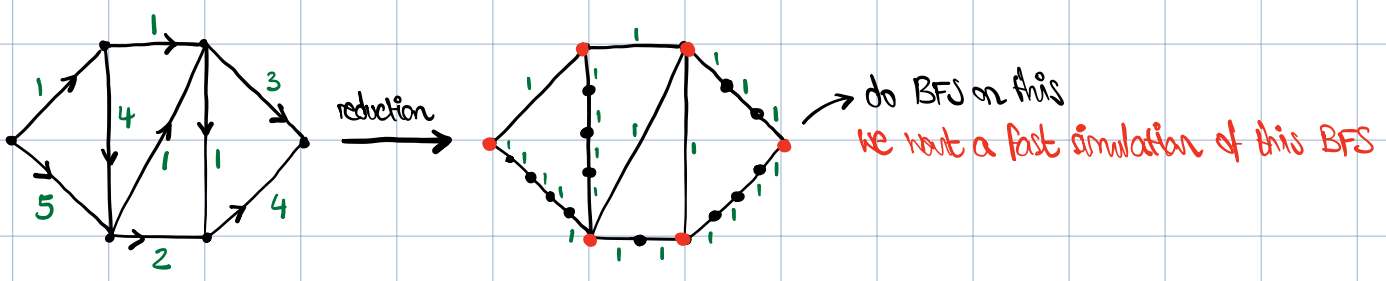
replace each edge of weight $w_e \in \mathbb{Z}_{\geq 0}$ by a path of w_e edges, forming G'

then $\exists$ st-path of length k in $G \iff \exists$ st-path of length k in G'

$\Rightarrow$ linear time algorithm for $G' \not\Leftrightarrow$ linear time algorithm for G

Problem: reduction might be inefficient since $|V'| \geq |V|$.
In particular, $|V'| = n + \sum_{e \in E} (w_e - 1) \Rightarrow$ large w_e problematic.

Idea: To design an efficient algorithm through this reduction, instead of constructing G' explicitly and running BFS, we simulate BFS on G' efficiently.



Physical Process Analogy

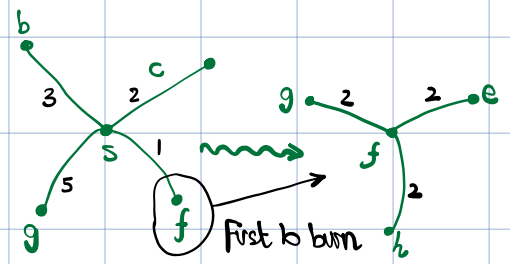
Construct graph G' with $w_e = 1$, $\forall e \in E'$.

Choose source $s \in V$ and start a fire that spreads one edge per unit time.

$\Rightarrow$ shortest path distance to $t \in V$ = amount of time until fire reaches t.

To simulate the process efficiently, we just need to keep identifying what the next vertex to be burned is, and when.

Initially, source $s \in V$ is burned.
We know b, c, g, f will be burned in time at most 3, 2, 5, 1.



f burns first. Then f burned at time 1 ⇒ g, h, e will burn in at most 2 more time.

⇒ we update the upper bound time for g to be burned to 3.

This is Dijkstra's algorithm. → We continue by finding the next vertex burned, and update upper bounds on its neighbours.

Dijkstra's Algorithm (BFS simulation)

```
dijkstra(G, s∈V):
    dist(v) = ∞ for every v∈V        initial upper bound ∞ for all vertices
    dist(s) = 0                      } initialize
    Q = priority-queue(V)            put all vertices in a priority queue with priority value dist(v)
                                     notice s∈Q as well.

    while Q ≠ ∅:
        u = delete-min(Q)            deque vertex with shortest dist(·) value (next to be burned)   ← next to burn

        for each neighbour v of u:
            if dist(u) + w_uv < dist(v):        shorter way to get to v through u
                dist(v) = dist(u) + w_uv        } shortest path update
                decrease-key(Q, v)              → the distance always only decreases here, so we update the
                                                  priority of the element using decrease-key in Q
                parent(v) = u
                update the parent of v to be u
                on the shortest path found      } this is where P is constructed
```

Proposition (Dijkstra): The algorithm finds shortest paths in $O((n+m)\log n)$ (binary heaps) or $O(m + n\log n)$ (Fibonacci heaps), where $m = \sum_{v\in V} \text{outdeg}(v)$.

Proof (Correctness): The algorithm is an efficient implementation of BFS on pseudo-G. One method is to prove the reduction holds. We proceed with a formal proof on an equivalent formulation.

1 The following formulation of Dijkstra's is equivalent to the one above:

* Equivalence: exercise (easy)

```
dijkstra(G, s∈V):
    dist(v) = ∞ ∀v∈V,  dist(s) = 0
    R = ∅
    while R ≠ V:
        pick u∈V\R that minimizes dist(u)      } equivalently, u∉R is the vertex such that
        R ← R ∪ {u}                              dist(u) = min_{v∉R, y∈R} {dist(y) + w_yv}
        for each uv∈E:                                    v∉R, y∈R
            dist(v) = min(dist(v), dist(u) + w_uv)
```

all y that have been burned; v that are not yet burned, neighbouring y ⇒ choose next to burn

2 The algorithm above is correct.

We proceed by induction. Invariant: for any $v\in R$, dist(v) is the shortest path distance from s to v.

Base Case: True when $R = \{s\}$.

Now assume that the invariant holds for current R. We want to show that the invariant holds after a new vertex u is added to R (in the $R \leftarrow R\cup\{u\}$ step).

We must show that $\text{dist}(u) = \min_{v\notin R,\, y\in R} \{\text{dist}(y) + w_{yv}\}$ is the shortest path distance from s to u.

Let $y\in R$ be the vertex in a minimizer,

i.e. $\text{dist}(u) = \text{dist}(y) + w_{yv} \le \text{dist}(a) + w_{ab}$ $\forall a\in R$ and $b\notin R$. (* what's v here?)

(≤) Since invariant holds for R, we have $y\in R \Rightarrow$ dist(y) shortest path from s to y

$\Rightarrow \text{dist}(u) \leq \underbrace{\text{dist}(y)}_{\text{sy-path}} + \underbrace{w_{yu}}_{\text{yu-edge}}$ using sy-path and following yu-edge. $\triangle \leq$

($\geq$) Now we show $\text{dist}(u) \geq \text{dist}(y) + w_{yu}$ for the minimizes $y \in R$.

Let P be any su-path. Since $s \in R$ and $u \notin R$, there must be some edge $xz \in P$ with

$x \in R$ and $z \notin R$

$\Rightarrow \text{length}(P) \geq \text{dist}(x) + w_{xz}$ (invariant holds for $R \Rightarrow$ this is minimized) lower bound (crucially uses fact that $w_e > 0 \ \forall e \in E$). ★ later we explore this more difficult case using DP

But $y \in R$ minimizes $\Rightarrow \text{dist}(y) + w_{yu} \leq \text{dist}(x) + w_{xz}$ since $\text{dist}(u) = \text{dist}(y) + w_{yv} \leq \text{dist}(a) + w_{ab}$ $\forall a \in R$ and $b \notin R$.

$\Rightarrow \text{length}(P) \geq \text{dist}(x) + w_{xz} \geq \text{dist}(y) + w_{yu}$.

Equality holds for any su-path $\Rightarrow \text{length}(P) = \text{dist}(u) \geq \text{dist}(y) + w_{yu}$. $\triangle \geq$

(shortest path choice; $\text{dist}(u)$ on shortest path)

So $\text{dist}(u) \leq \text{dist}(y) + w_{yu}$ and $\text{dist}(u) \geq \text{dist}(y) + w_{yu} \Rightarrow \text{dist}(u) = \text{dist}(y) + w_{yu}$ ☐ Correctness

Proof (Complexity): Every $v \in V$ is enqueued exactly once (at the start) and dequeued exactly once (when it burns).

When dequeuing, we check every outbound edge once and may use $\text{dist}(u) + w_{uv}$ to decrease-key.

```
dijkstra(G, s∈V):
    dist(v) = ∞ for every v∈V        O(n)
    dist(s) = 0                      O(1)
    Q = priority-queue(V)            ← O(n log n) inserting n items into Q
                                       suppose binary min-heap
    while Q ≠ ∅:                     } n iterations
        u = delete-min(Q)
        for each neighbour v of u:   } O(deg(u)) iterations
            if dist(u) + w_uv < dist(v):
                dist(v) = dist(u) + w_uv
                decrease-key(Q, v)   ← O(log n) for bubble-up
                parent(v) = u
```

Note: Fibonacci heaps are out of scope.

$$\Rightarrow T(n) = n\log n + \sum_{v=1}^{n}\Big(\underbrace{\sum_{uv\in E}}_{\deg(v)} \log n\Big) = n\log n + \sum_{v=1}^{n} \deg(v)\log n = \Big(n + \underbrace{\sum_{v\in V} \deg(v)}_{m}\Big)\log n. \quad \text{(outdeg(v))}$$

☐ Complexity

Allowing negative edge lengths makes the SS shortest-path problem considerably harder.

Before approaching this problem and introducing the Bellman-Ford algorithm, we introduce dynamic programming.

Intuition: although Dijkstra may not compute all distances correctly in one pass, it will compute distances to some of the vertices correctly (e.g. the first vertex on a shortest path).

Dynamic Programming (1)

On a high level, we can solve a problem by dynamic programming if there is a recurrence relation with only a small number of subproblems (i.e. polynomially many).

This is a general and powerful technique that is simple to use once learnt well.

A common toy problem is computing the Fibonacci sequence.

The function $F(n) = F(n-1) + F(n-2)$ is defined recursively. Using recursion gives a huge recursion tree

$\Rightarrow$ runtime $O(1.618^n)$ (MATH 239).

Notice many subproblems are computed redundantly (we only have n subproblems) $\rightsquigarrow$ why waste time?

Top-Down Memoization

Store visited/computed values in an array (or hashmap) visited[i] so we can avoid re-computation.

$\Rightarrow$ each subproblem computed once, and when it is computed, we do an $O(1)$ lookup $\Rightarrow O(n)$ computations.

Bottom-Up Computation

Straightforward bottom-up iterative approach $\Rightarrow O(n)$ additions.

This is the framework of DP: store intermediate values to avoid recomputation.

From the top-down approach, it should be clear that if we write a recursion with only polynomially many subproblems with some additional polynomial-time processing, the problem can be solved in polynomial time.

$\Rightarrow$ designing efficient DP algorithms boils down to designing a nice recursion (like D&C algorithms).

In principle, we just need to find some topological ordering of subproblems.

Ex. Weighted Interval Scheduling:

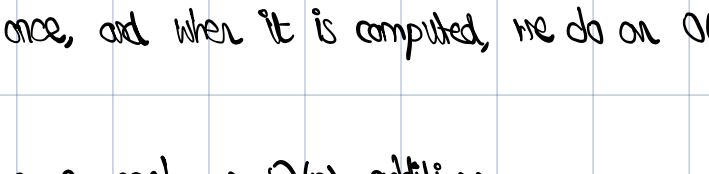
Input: n intervals $[s_1, f_1], \dots, [s_n, f_n]$ with weights $w_1, \dots, w_n \in \mathbb{R}$

Output: a subset n of disjoint intervals that maximizes $\sum_{i \in S} w_i$

We can think of the intervals as requests to book a room for time $[s_i, f_i]$ and cost w_i

$\Rightarrow$ we want to maximize our earnings by choosing good intervals

Naive Exhaustive search.

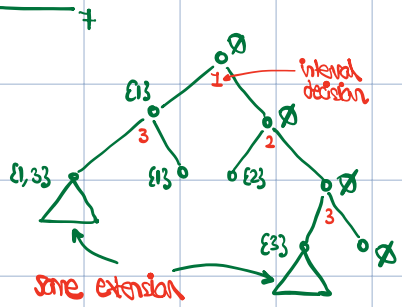


1 Decide whether to choose interval 1.

choose 1 $\Rightarrow$ reject 2 $\Rightarrow$ decide on interval 3

not choose 1 $\Rightarrow$ decide on interval 2

2 Proceed recursively.



$\Rightarrow$ lots of redundancy in tree

Notice: To determine how to extend partial solutions, what really matters is the last chosen interval of the current partial solution.

Nothing on the left matters, since they won't interact with anything on the right.

$\rightarrow$ we just need to keep track of the "boundary" of the solution (here, the boundary is the last interval) $\rightsquigarrow$ there should be a recursion with only one parameter (boundary)

Better Recurrence → use a good ordering of intervals and pre-compute some useful information

1 Sort intervals by starting time so $s_1 \le s_2 \le \dots \le s_n$.

2 For i^{th} interval, use next[i] to determine smallest j with $j>i$ and $f_i < s_j$
If no such j exists, set next[i] = n+1 (end)

strict ineq. because we want disjoint intervals

$f_i < s_j$: i ends before j starts

⇒ for interval i, we have that intervals $\{i, \dots, next[i]-1\}$ overlap with i, and $\{next[i], \dots\}$ are disjoint from i.

Now we are ready to write a recurrence with one parameter, resulting in n subproblems.

Define opt(i) = max income that we can earn using intervals in $\{i, \dots, n\}$ only. Goal: compute opt(1).

To compute opt(1) there are only 2 possibilities:

1) Solutions that choose interval 1.
⇒ we cannot choose intervals in $\{2, \dots, next[1]-1\}$ since these overlap with interval 1
⇒ the optimal value for solutions choosing interval 1 is $w_1 + opt(next[1])$

2) Solutions that exclude interval 1.
⇒ opt(2) is by definition the optimal value

⇒ $opt(1) = \max(opt(2),\ w_1 + opt(next(1)))$ and generally, $opt(i) = \max(opt(i+1),\ w_i + opt(next(i)))$ defines a recurrence.

Top-down weighted interval scheduling

1 Sort intervals by non-increasing starting time so $s_1 \le \dots \le s_n$
2 Compute next[i] for $1 \le i \le n$, set visited[i] = false
3 Return opt(1) ← O(n) we have n subproblems

} pre-process O(nlogn) ← why?

ready, sorting is O(nlogn).
remains to show that next[] array can be computed in O(nlogn).
consider using a binary search approach:

```
next(i, A, n):        A sorted array
    mid = n/2
    A_L, A_R = A (halved at middle)
    if A_L[-1] ≤ A[i]:
        next must be in A_R
    else:
        next must be in A_L
```

O(logn) algorithm
⇒ O(nlogn) when called over all elems

```
opt(i): ← subroutine definition
    if i ≥ n+1:
        return 0                    } base case
    if visited[i] true:
        return answer[i]            } opt(i) already computed
    answer[i] = max(opt(i+1), w_i + opt(next[i]))   } recursive
    visited[i] = true
    return answer[i]
```

We can also write an equivalent bottom-up variant:

Bottom-Up Weighted Interval Scheduling

some pre-processing in O(nlogn)

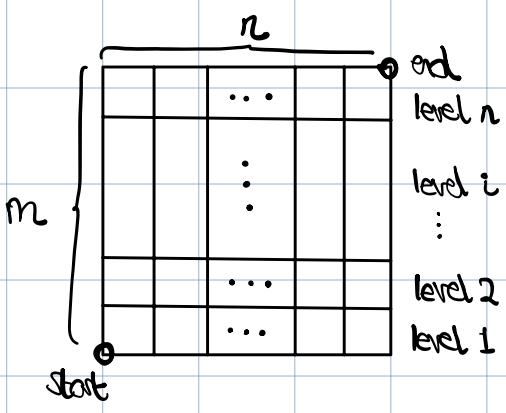
```
opt(n+1) = 0
for (i=n; i≥1; i--):
    opt[i] = max(opt(i+1), w_i + opt(next[i]))
```

potentially more natural solution working backwards to find an answer

both of these would already be pre-computed

We can see why dynamic programming is general and powerful, because it is very systematic and yields very competitive algorithms.

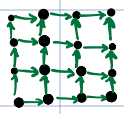
B Grid Path Counting say we have an m×n grid and start at the bottom left. We can move up or right each step. How many ways of getting from bottom left to top right?



Notice that to get to a particular location M[i][j], we must have either just moved up, or just moved right.

#ways to get to [i,j] location

$\Rightarrow$ $M[i][j] = \underbrace{M[i-1][j]}_{\text{moved up}} + \underbrace{M[i][j-1]}_{\text{moved right}}$ is a recurrence relation

Therefore compute dependencies look something like  $\Rightarrow$ we must first compute left and bottom edges.

Fortunately, this is very easy.

$\Rightarrow$ we have reduced to a recurrence with n^2 subproblems.

Remark: We can likely derive a combinatorial/analytic solution, but we are not interested in that

C. Subset-Sum Problem

input: $a_1,\dots,a_n \in \mathbb{N}_{>0}$ and $k \in \mathbb{Z}$

output: some collection $S \subseteq \{1,\dots,n\}$ such that $\sum_{i \in S} a_i = k$, or report that no such S exists

For each a_i, we can choose to either include it or exclude it. Say we start with a_1.

1) Include a_1. Then we must choose some subset $S' \subseteq \{2,\dots,n\}$ such that $a_1 + \sum_{i \in S'} a_i = k$ so $\sum_{i \in S'} a_i = k - a_1$.

2) Exclude a_1. Then we need $S' \subseteq \{2,\dots,n\}$ with $\sum_{i \in S'} a_i = k$.

Naive: Naïvely implement the above logic $\Rightarrow O(2^n)$ time since every subset will be enumerated.

We want to reduce the search space for the problem.

Observation: Say we have chosen two subsets $S_1 \subseteq \{1,\dots,i\}$ and $S_2 \subseteq \{1,\dots,i\}$ so far, and say that $\sum_{i \in S_1} a_i = \sum_{i \in S_2} a_i$. Notice the remaining $S' \subseteq \{i+1,\dots,n\}$ is agnostic of the choice of S_1 or S_2 thus far.

$\Rightarrow$ as long as subsets selected thus far have the same sum, we need not track them.

$\Rightarrow$ we can just keep track of the partial sum in the recursion.

{ reduce

consider the subproblem subsum[i, L] for $i \in [1, n]$ and $L \le k$, where subsum[i, L] = true $\iff \exists S' \subseteq \{i,\dots,n\}$ with sum L.

$\Rightarrow$ we would like to solve subsum[1, k]

We have a recurrence subsum[i, L] = (subsum[i+1, L−a_i] (include element a_i) OR subsum[i+1, L] (exclude element a_i))

$\Rightarrow$ there are $O(nk)$ subproblems (n choices for i, k choices for L)

$\Rightarrow$ time complexity $O(nk)$.

Top-down subset-sum

```
subsum(i, L):
    if L=0: return true   ← choose empty set ∅
    if (i>n or L<0): return false   ← ran out of numbers or partial sum too large
    return (subsum(i+1, L−a_i) or subsum(i+1, L))
```

Remark: $O(nk)$ is pseudo-polynomial since the runtime looks polynomial but depends on k.

If $k \in \Omega(2^n)$ this would be exponential, and would be even slower than exhaustive search.

This is likely unavoidable. We will soon see that the subset-sum problem is NP-complete.

* Exercise: bottom-up algorithm implementation (use matrix $M \in \mathbb{R}^{n \times k}$ for memorization.

optimize space and use a matrix $M \in \mathbb{R}^{2 \times k}$.

Notice that the bottom-up version has runtime $\Theta(nk)$ but top-down is $O(nk)$. (some inputs may return faster)

Dynamic Programming and Graph Search

We can consider representing the search space of a DP problem as a graph, where each vertex is a subproblem, and directed edges are added according to the DP recurrence relation.

Then, the problem is equivalent to finding some directed path from a starting (desired) state to a "true base state".

Top-down memoization is basically doing a DFS on this subproblem graph.

0/1 Knapsack

input: n items of weight w_i and value v_i, and some $W \in \mathbb{N}_0$.

output: a subset $S \subseteq \{1,\dots,n\}$ with $\sum_{i \in S} w_i \le W$ that maximizes $\sum_{i \in S} v_i$.

Think of a robber who can carry weight at most W, and wants to maximize the value of stolen goods.

Define subproblem knapsack(i, W, V) = true $\iff \exists S' \subseteq \{i,\dots,n\}$ with weight $\le W$ and total value V.

Two possibilities for object i: either include or exclude.

1) Include $\Rightarrow$ knapsack$(i+1,$ $W-w_i$ (weight of remaining), $V-v_i$ (value of remaining)$)$; 2) Exclude $\Rightarrow$ knapsack$(i+1, W, V)$.

Therefore knapsack(i, W, V) = (knapsack$(i+1, W-w_i, V-v_i)$ or knapsack$(i+1, W, V)$) recurrence relation.

Notice we can use a 3D tensor to encode this information.

Parameter / Dimensionality Reduction

Notice that, for the solution we intend to find, some subproblems dominate others.

In particular, if knapsack(i, W, V') is true for some $V' > V$, we can ignore subproblem knapsack(i, W, V)

$\Rightarrow$ define knapsack(i, W) = max. value achievable with $S \subseteq \{i,\dots,n\}$ of weight $\le W$.

More precisely, $\text{knapsack}(i, W) = \max_{S \subseteq \{i,\dots,n\}} \left\{ \sum_{j \in S} v_j \;\middle|\; \sum_{j \in S} w_j \le W \right\}$.

↓ recurrence

$$\text{knapsack}(i, W) = \max\{\underbrace{v_i + \text{knapsack}(i+1, W-w_i)}_{\text{include item } i}, \underbrace{\text{knapsack}(i+1, W)}_{\text{exclude item } i}\}.$$

Base cases: $W < 0 \Rightarrow$ return $-\infty$; $i > n \Rightarrow$ return 0

* Exercise: Implement pseudocode. $\Rightarrow O(nW)$ time (subproblem space size).

Remark: Knapsack is pseudo-polynomial and is actually NP-complete, proven in last chapter.

DP1 climbing-stairs: there are n stairs in a staircase. each step we can climb either 1 or 2 steps. how many distinct ways of climbing?

Notice climb(i) = climb$(i-1)$ (1 stair) + climb$(i-2)$ (2 stairs) defines a recurrence relation with climb$(0) = 1$, climb$(1) = 1$

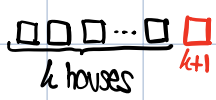
$\Rightarrow$ reduce to Fibonacci.

अतिथि
-Charlie Li

DP2 (meal) house-robber: some array A where $A[i]$ = money in house i. Houses are arranged contiguously in a straight line. can only steal from non-adjacent houses; what's the most we can steal?

Notice we can define a recursive subproblem decomposition:

2 choices: steal from $k+1$ and skip k, or steal from $[1..k]$ and skip $k+1$



$\Rightarrow$ value$[i]$ = max(get-value$(i-1)$ (skip i), $A[i]$ + get-value$(i-2)$ (skip $i-1$))

giving us $O(n)$ time and $O(n)$ space with memoization

DP3 med **coin-change**: given a target $t \in \mathbb{N}$ and a list of coin denominations A, find smallest number of coins needed to make t.

Notice greedy won't work here, since consider $t=105$ and $A=[1, 7, 98, 100]$.

Given some value v, we want to know what coin a_i to use to minimize total #coins.

Say coins[v] represents the smallest #coins needed for value v.

then we have a recursive definition $\text{coins}[v] = \underbrace{\min_i (1+\text{coins}[v-a_i])}_{O(n)}$ $\Rightarrow$ we can design an $O(tn)$ algorithm for coin-change.

```
need = dict()
need[0] = 0
def get_need(amt):
    if amt < 0: return ∞
    if amt in need: return need[amt]
    need[amt] = 1 + min{get_need(amt - a_i) | a_i ≤ amt}
return get_need(t)
```

Dynamic Programming (2)

In this section, we will use dynamic programming for basic sequence and string problems.

E Longest Increasing Subsequence

Given numbers $a_1, \dots, a_n$, a subsequence is a subset of the form $a_{i_1}, \dots, a_{i_k}$ where $i_1 < \dots < i_k$.

The subsequence is increasing if $a_{i_1} < \dots < a_{i_k}$.

subproblem: let $L(i)$ = length of a longest increasing subsequence starting at a_i formed using $\{a_i, \dots, a_n\}$

$\Rightarrow$ there are n subproblems.

once we compute $L(1), \dots, L(n)$, our final answer is $\max(L(i))$.

recurrence: say we are at a_i. Notice that we can prepend a_i to any subsequence $L(j)$ for $j > i$ and $a_j > a_i$

(so $L(i)$ forms some subsequence $a_i \underbrace{a_j \dots}_{\text{found}}$ giving us $L(i) = \max_{\substack{j>i \\ a_j > a_i}} (1 + L(j)) = 1 + \max_{\substack{j>i \\ a_j > a_i}} \{L(j) \mid a_j > a_i\}$.

* Implement

$\Rightarrow O(n^2)$ time.

Faster Algorithm: Notice some subproblems are "dominated" by others.

For each length k, we store the "best" position to start an increasing subsequence of length k.

$\Rightarrow$ these "best" positions satisfy a monotone property.

$\Rightarrow$ we can binary search and update these values in $O(\log n)$ time when considering a new a_i.

idea: say we have computed $L(i+1), \dots, L(n)$ and we want to compute $L(i)$.

For some length k, consider some indices $i_1, \dots, i_\ell$ such that $L(i_1) = \dots = L(i_\ell) = k$ and $i < i_1 < \dots < i_\ell$.

what's the best subproblem to loop for computations $L(i), L(i-1), \dots, L(1)$?

↳ notice since we are extending using some elements in $\{1, \dots, i\}$, the start indices $i_1, \dots, i_\ell$ are unimportant.

↳ the starting values are what matters.

if $L(i_1) = L(i_2)$ and $a_{i_1} > a_{i_2}$, then subproblem $L(i_1)$ dominates subproblem $L(i_2)$, since

among increasing subseq S_{i_1}, S_{i_2} of length k, one with largest start value is easiest to be extended

- any increasing $S_1 \subseteq \{a_1, \dots, a_{i_1}\}$ that can be extended by some $\underbrace{S_{i_2}}_{\text{length } k}$ where S_{i_2} starts at a_{i_2}
- can also be extended by $\underbrace{S_{i_1}}_{\text{length } k}$ starting at a_{i_1}.

$$a_i \underbrace{a_{i_1} \cdots}_{k} \text{ and } a_i \underbrace{a_{i_2} \cdots}_{k} \Rightarrow \text{we choose to extend by } a_{i_1} > a_{i_2}$$

Define $pos[k] = \underbrace{\underset{j>i}{\operatorname{argmax}}\{a_j \mid L(j) = k\}}_{\substack{j \text{ such that } a_j \text{ maximized} \\ \text{with some length-}k \text{ subsequence}}}$ where $L(i)$ is the current subproblem to be computed.

← best position $j>i$ to start a length-k increasing subsequence

Let $m_i = \max_{j>i}\{L(j)\}$ be the length of the longest future subsequence.

$\Rightarrow$ when computing $L(i)$, we just need to consider $L(pos(1)), \dots, L(pos(m_i))$

For instance, say our sequence is $\underset{1}{2}, \underset{2}{7}, \underset{3}{6}, \underset{4}{1}, \underset{5}{4}, \underset{6}{8}, \underset{7}{5}, \underset{8}{3}$. (here) Say we are computing $L(2)$. We have:

$L(3) = L(5) = 2 \rightsquigarrow pos[2] = 3 \quad (a_3 = 6)$
$L(4) = 3 \rightsquigarrow pos[3] = 4 \quad (a_4 = 1)$
$L(6) = L(7) = L(8) = 1 \rightsquigarrow pos[1] = 6 \quad (a_6 = 8)$

} extract best future index to start a subseq. of that length

we only keep these subproblems

Claim (Monotone Property): Once we only keep the above subproblems, we have

$A[pos[1]] > A[pos[2]] > \dots > A[pos[m_i]]$ at subproblem $L(i)$.

(largest element to start length 1; largest element to start length 2) → makes intuitive sense?

Proof: SFAC $\exists j$ with $A[pos[j]] \geq A[pos[j-1]]$. Let $a_{p_1} < \dots < a_{p_j}$ be an optimal increasing subsequence of length j where $p_1 = pos[j]$. Notice $a_{p_2} < \dots < a_{p_j}$ is an increasing subsequence of length $j-1$ and $a_{p_2} > a_{p_1}$.

Thus $A[pos[j-1]] \geq a_{p_2} > a_{p_1} = A[pos[j]] \Rightarrow A[pos[j]] < A[pos[j-1]]$. ↯ □

(definition)

Now we try to formulate a way to update the best subproblems using binary search when on a_i.

Suppose we have found best subproblems $pos[m_i], \dots, pos[1]$ in past computations. We have 3 cases:

1) $a_i < a_{pos[m_i]}$ best case. We extend the longest subsequence by 1, so $m_{i-1} = m_i + 1$ and $pos[m_{i-1}] = i$.

binary search to find such j → 2) $a_{pos[j]} \leq a_i < a_{pos[j-1]} \Rightarrow$ cannot use a_i to extend length-j subsequence, but can extend $j-1$.

Since $a_i \geq a_{pos[j]}$ the new index i for a length-j subsequence is better than previous $pos[j]$

$\Rightarrow pos[j] \leftarrow i$.

3) $a_{pos[1]} \leq a_i \Rightarrow a_i$ is larger than all other subsequence starting values, so we can extend no subseq.

but can update $pos[1] \leftarrow i$.

← the final algo. is very simple but may be hard to invent.

```
m = 1, pos[1] = n
for (i = n-1; i ≥ 1; i--):
    if a_i < a[pos[m]]: set m ← m+1 and pos[m] ← i
    else use binary search to find smallest j with a[pos[j]] ≤ a_i < a[pos[j-1]]
        and set pos[j] ← i
return m
```

$\Rightarrow$ time $O(n \log n)$

F. Longest Common Subsequence

We have two strings $a_1,\dots,a_n$ and $b_1,\dots,b_m$ where each a_i, b_j are symbols.

We want to find the largest k such that there are $\begin{matrix} i_1<\dots<i_k \\ j_1<\dots<j_k \end{matrix}$ such that $a_{i_1}\dots a_{i_k} = b_{j_1}\dots b_{j_k}$.

The E Longest Increasing Subsequence problem is a special case of this, where $b = \text{sorted}(a)$.

recurrence: Let $C(i,j)$ be the length of a longest common subsequence of $a_i\dots a_n$ and $b_j\dots b_m$. want: $C(1,1)$.

to compute $C(i,j)$ there are three cases:

1) $a_i = b_j$ use both $\Rightarrow$ put a_i and b_j at the start of any existing common subsequence $\Rightarrow$ remains to find $C(i+1, j+1)$

   $\Rightarrow$ let $S_1 = 1 + C(i+1, j+1)$. otherwise $S_1 = 0$.

2) not use a_i $\Rightarrow$ find a longest common subsequence for $a_{i+1},\dots,a_n$ and $b_j,\dots,b_m$.

   $\Rightarrow$ let $S_2 = C(i+1, j)$.

3) not use b_j $\Rightarrow$ find a longest common subsequence for $a_i,\dots,a_n$ and $b_{j+1},\dots,b_m$.

   $\Rightarrow$ let $S_3 = C(i, j+1)$.

Then we take $C(i,j) = \max\{S_1, S_2, S_3\}$ since these encompass all possibilities.

Time: there are mn subproblems, each doing $O(1)$ lookups (if we memoize) $\Rightarrow O(mn)$ time.

G. Edit Distance

Two strings $a_1,\dots,a_n$ and $b_1,\dots,b_m$.

We want to find smallest k such that k insert/delete/change operations can change $a_1\dots a_n$ to $b_1\dots b_m$.

recurrence: similar to E LCS. Let $D(i,j)$ be the edit distance of $a_i\dots a_n$ and $b_j\dots b_m$. want: $D(1,1)$.

to compute $D(i,j)$ there are four subcases:

1) insert b_j to the current string (eg. $\dots|abc/def \to \dots-|abc/def$)

   Then we match one more symbol of the target string and move on: if $j \le m$ then $SOL_1 = 1 + D(i, j+1)$ else $SOL_1 = \infty$.

2) delete a_i from the current string when $i \le n$ (eg. $\dots|abc/def \to \dots a|bc/def$)

   Then move forward one symbol in the current string: if $i \le n$ then $SOL_2 = 1 + D(i+1, j)$ else $SOL_2 = \infty$.

3) change a_i to b_j when $i \le n$ and $j \le m$. Thus if $i \le n$ and $j \le m$ then $SOL_3 = 1 + D(i+1, j+1)$ else $SOL_3 = \infty$.

4) match a_i and b_j if $i \le n$ and $j \le m$ and $a_i = b_j$, so if $i \le n$ and $j \le m$ and $a_i = b_j$ then $SOL_4 = D(i+1, j+1)$ else $SOL_4 = \infty$

Set $D(i,j) = \min\{SOL_1, SOL_2, SOL_3, SOL_4\}$.

There are mn subproblems, each requiring a constant number of operations $\Rightarrow$ time $O(mn)$.

Dynamic Programming on Graphs

Recall:

> Allowing negative edge lengths makes the SS shortest-path problem considerably harder.
> Before approaching this problem and introducing the Bellman-Ford algorithm, we introduce dynamic programming.
> Intuition: although Dijkstra may not compute all distances correctly in one pass, it will compute distances to some of the vertices correctly (e.g. the first vertex on a shortest path).

Dijkstra's

We study algorithms to solve the following problems:

1) Given a (weighted) directed graph G, check if there exists a negative cycle, i.e. $\sum_{e \in C} w_e < 0$.
2) If G has no negative cycles, solve the single-source shortest path problem.

Bellman-Ford Algorithm

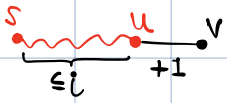
We use dynamic programming to compute the shortest path distance (weight/cost) from $s \in V$ to every $v \in V$ using at most i edges, from $i=1$ to $i=n-1$.

Subproblems: Let $D(v, i)$ be the shortest path distance from $s \in V$ to v using at most i edges.

Answers: For each $v \in V$, distance $D(v, n-1)$ is shortest from s to v, when G has no negative cycles.

Base case(s): $D(s, 0) = 0$ and $D(v, 0) = \infty$ for all $v \in V \setminus \{s\}$.

Recurrence: To compute $D(v, i+1)$, a path with $\leq i+1$ edges must be an extension of a path with $\leq i$ edges from s to u, for some in-neighbour u of v.



$\Rightarrow$ look for all paths $P_{s \to u}$ with $\text{len}(P_{s \to u}) \leq i$ to find the best candidate for extension to v.

We get the recurrence relation $D(v, i+1) = \min\{\underbrace{D(v, i)}_{\text{existing path}}, \underbrace{\min_{u:\, uv \in E}\{D(u, i) + w_{u,v}\}}_{\text{candidate for extension}}\}$

Time complexity: Given $D(v, i)$ for all $v \in V$, takes $\text{in-deg}(w)$ to compute $D(w, i+1)$

$\Rightarrow$ computing $D(w, i+1)$ for all $w \in V$ takes $O(\sum_{w \in V} \text{in-deg}(w)) = O(m)$ time.

$\Rightarrow$ takes $O(mn)$ time to compute for all $1 \leq i \leq n-1$.

Space: naively $\Theta(n^2)$, reduce to $O(n)$ by noticing we only need $D(v, i)$ $\forall v \in V$ to compute $D(w, i+1)$ $\forall w \in V$.

```
dist(s) = 0,  dist(v) = ∞  ∀v ∈ V\{s}
for i = 1 to i = n-1:
    for uv ∈ E:
        if dist(u) + w_uv < dist(v):
            dist(v) = dist(u) + w_uv
            parent(v) = u
```

Bellman-Ford algorithm

Shortest Path Tree As in Dijkstra's, we would like to return a shortest path $s \to v$ by following edges $(\text{parent}(v), v)_v \in E$. But Bellman-Ford has many outer-loop iterations, and it is not clear whether the edges form a tree. In fact, the path $(\text{parent}(v), v)_v$ may contain cycles. We show these cycles must be negative.

Lemma: If there is a directed cycle C in edges $(\text{parent}(v), v)$, then C must be negative ($\sum_{e \in C} w_e < 0$).

Proof: Let C be the directed cycle, and write $C = v_1 \dots v_k v_1$ with $v_i v_{i+1} \in E$.

Assume that $v_k v_1$ is the last edge formed by the Bellman-Ford algorithm in C, so the cycle C is formed when $\text{parent}(v_1) \leftarrow v_k$, while $\text{parent}(v_i) = v_{i-1}$ already for $2 \le i \le k$.

Consider values $\text{dist}(v_i)$ right before v_k becomes the parent of v_1.

Since $v_{i-1} = \text{parent}(v_i)$ we have $\text{dist}(v_i) \ge \text{dist}(v_{i-1}) + w_{v_{i-1}v_i}$ for $2 \le i \le k$.

Note we cannot have $\text{dist}(v_i) < \text{dist}(v_{i-1}) + w_{v_{i-1}v_i}$ since otherwise $\text{parent}(v_i)$ would be updated.

When we set $\text{parent}(v_1) = v_k$, we must have $\text{dist}(v_1) > \text{dist}(v_k) + w_{v_k v_1}$

$$\Rightarrow \sum_{i=1}^{k} \text{dist}(v_i) = \sum_{i=2}^{k} \underbrace{\text{dist}(v_i)}_{\ge \text{dist}(v_{i-1}) + w_{v_{i-1}v_i}} + \text{dist}(v_1) \ge \sum_{i=1}^{k-1} \text{dist}(v_i) + \sum_{i=2}^{k} w_{v_{i-1}v_i} + \underbrace{\text{dist}(v_1)}_{> \text{dist}(v_k) + w_{v_k v_1}}$$

$$> \sum_{i=1}^{k} \text{dist}(v_i) + \sum_{e \in C} w_e$$

$\Rightarrow \sum_{e \in C} w_e < 0$ so C is a negative cycle. $\square$

Thus if G has no negative cycles, then edges $(\text{parent}(v), v)$ have no directed cycles.

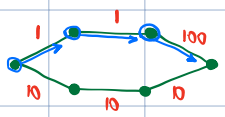
G connected $\Rightarrow$ every $v \in V$ has exactly one incoming edge in $(\text{parent}(v), v)$ and $\nexists$ directed cycles

$\Rightarrow$ edges $(\text{parent}(v), v)$ form a directed tree moving away from $s \in V$.

Greedy Algorithms

When does making the locally optimal choice at each stage result in a globally optimal solution? ← without backtracking

In dynamic programming, we perform an exhaustive search on all subproblems, and may reconsider the previous stage's solutions to improve future stages.

Anti-example Pathfinding. 

locally optimal choices are bad.

We say a problem exhibits the greedy choice property if locally optimal choices lead to a globally optimized solution. ← not necessarily unique!

Often, we express these problems recursively.

To prove that a problem satisfies the greedy property, we:

1. Show an optimal solution exists and define it
2. Show that the (greedy) recursive problem decomposition generates the optimal solution.

A Fractional Knapsack Problem: Recall DP.D Knapsack, input: n items of weight w_i and value v_i, and some $W \in \mathbb{N}_{>0}$. output: a subset $S \subseteq \{1,\dots,n\}$ with $\sum_{i\in S} w_i \le W$ that maximizes $\sum_{i\in S} v_i$.

What if we are allowed to take fractions $p_i \in [0,1]$ of each item such that $\sum_{i\in S} p_i v_i$ is maximized under $\sum_{i\in S} p_i w_i \le W$?

This is the fractional knapsack problem. Intuitively, filling the knapsack with highest value:weight ratio is best.

Define densities $d_i = \left(\frac{v_i}{w_i}\right)$ for $i \in \{1,\dots,n\}$ and sort $D = [d_i]_i$ in decreasing order.

Then choose high-density elements until we no longer can.

$D = [(\frac{v_i}{w_i})]_i$ $O(n)$

$D = \text{sorted}(D)$ $O(n \log n)$

Collect elements from D until adding one more would be overweight; add fractional part of that last element } $O(n)$

$\Rightarrow O(n \log n)$ time, $O(1)$ space.

* HARD (very)! Find an $O(n)$ algorithm for the problem that adopts a weighted median algorithm.

Claim: The fractional knapsack problem has the greedy choice property.

Proof: Say w_i, v_i are sorted such that $\frac{v_i}{w_i} > \frac{v_{i+1}}{w_{i+1}}$.

Let $G = (p_1,\dots,p_n)$ be the fractions produced greedily, and $OPT = \{\vartheta = (q_1,\dots,q_n) \mid \vartheta \text{ optimal}\}$. We claim $G \in OPT$, SFAC not.

By assumption, $\sum q_i v_i > \sum p_i v_i$ and so $(p_i)_i \neq (q_i)_i$.

Let i be the first index such that $q_i \neq p_i$.

By the design of our algorithm, we must have $p_i > q_i$. } G takes a larger proportion of this higher density element

By optimality of ϑ we must have some $j > i$ (lower density) with $p_i < q_j$. → ϑ takes larger prop. of some lower density elem.

Idea: Define ϑ' that takes slightly more i and less j, show $\vartheta' > \vartheta$.

Say $\vartheta' = (q'_1,\dots,q'_n)$ where $q'_k = q_k$ for $k \neq i,j$.

Define $q'_i = q_i + \varepsilon$ and $q'_j = q_j - \varepsilon \frac{w_i}{w_j}$.

Then $\sum_{k=1}^{n} q'_k v_k = \sum_{k\neq i,j} q_k v_k + (q_i+\varepsilon) v_i + (q_j - \varepsilon \frac{w_i}{w_j}) v_j$

$= \sum_{k\neq i,j} q_k v_k + q_i v_i + q_j v_j + \underbrace{\varepsilon v_i - \varepsilon \frac{v_j}{w_j} w_i}_{v_i - \frac{v_j}{w_j} w_i = w_i \frac{v_i}{w_i} - w_i \frac{v_j}{w_j} > 0} > \sum_{k=1}^{n} q_k v_k$

and $\sum_{k=1}^{n} q'_k w_k = \sum_{k=1}^{n} q_k w_k \le W$

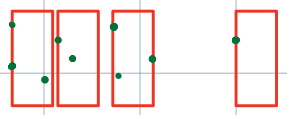
$\Rightarrow$ we have contradicted optimality of ϑ. ↯ $\Rightarrow G \in OPT$. □

Remark: The proof for greedy property we use is by contradiction and is constructive.

We can also formulate a recursive proof:

1. Identify recursive structure
2. Prove that optimal structure preserves optimal substructure

B minimum-rectangle-cover: some array $A = ((x_i, y_i))$ of points, some $w \in \mathbb{R}_{>0}$ width. Find minimum number of rectangles of width $\le w$ that covers A.



We can use a greedy algorithm here.

We claim that this algorithm produces the correct result.

A = sort A by x-value
cover the first point with rectangle R of width w
"remove" all points covered by R
do recursively

$\Rightarrow$ time $O(n \log n)$, space $O(1)$.

Substructure: Consider the decomposition $A = (p_1, \dots, p_k) \sqcup (p_{k+1}, \dots, p_n)$ such that $x_k - x_1 \leq w$ but $x_{k+1} - x_k > w$.

Then min-rect-cover$(A) = 1 +$ min-rect-cover$((p_{k+1}, \dots, p_n))$.

Proof: SFAC not true, so $m(A) \neq 1 + m(A')$. If $m(A) < 1 + m(A')$ then $m(A) \leq m(A')$ but if $x_{k+1} - x_1 > w$ then this cannot be true.

If $m(A) > 1 + m(A')$ we can construct a cover taking $m(A')$ and adding a rectangle covering $(p_1, \dots, p_k) \Rightarrow m(A) \leq 1 + m(A')$. ↯

Therefore the recursive substructure holds. □

Since the algorithm uses this substructure decomposition, we have proven correctness.

E min-halve-sum-operations

★ finish greedy algos.

Amortized Analysis

So far, we have been looking at static problems: solve a problem of size n.

We want to shift our focus to general sequences of operations, where we aim to analyze the cost per operation. (eg. heap use cost in Dijkstras)

Instead of being given n items, we might want to analyze the cost of n consecutive search/insert/delete operations.

The "amortized cost" over n operations is roughly $\frac{\sum_{i=1}^{n} \text{cost}(op_i)}{n}$ ← total cost / ← #operations

} our actual definition will be slightly more flexible

Aggregate Method

A (aggregate) std::vector dynamic resizing (aggregate method)

→ "weakest" but most intuitive

In C++, we know the std::vector structure provides dynamic arrays. How can we perform amortized analysis?

+1 resizing: Say our resizing strategy when pushing into an array of size n was to: 1 allocate size $n+1$ array, 2 copy old elements into new

Starting with an empty array and performing n insertions, we have

$$C_{Am} = \frac{1}{n}\sum_{i=0}^{n-1} i = \frac{1}{n}\cdot\frac{n(n-1)}{2} = \frac{n-1}{2}$$ ← amortized cost per operation

Of course, this resizing strategy is bad.

double resizing: Instead, we double array size whenever we reach capacity.

Let $k \in \mathbb{N}$ be such that $2^k < n \leq 2^{k+1}$, so our array has capacity 2^{k+1}. Then we have

$$C_{Am} = \frac{1}{n}\sum_{i=0}^{k} 2^i = \frac{1}{n}\left(\frac{2^{k+1}-1}{2-1}\right) \overset{\text{drop } -1}{\leq} \frac{1}{n}2^{k+1} = \frac{1}{n}2\cdot 2^k \overset{2^k<n}{<} \frac{1}{n}2n = 2$$ ← amortized cost per operation

Clearly, this strategy is better.

* Exercise: Hash tables have $O(1)$ expected amortized cost with doubling strategy.

But why do we need amortized analysis?

B Bitmask (motivating example): Consider the algorithm:

```
increment(A):                    # A: array of bits
    i ← 0
    while i < len(A) and A[i] = 1:
        A[i] = 0                 ← flip the bit 1→0
        i++
    if i < len(A):
        A[i] = 1                 ← flip 0→1
```

* Exercise: amortized cost of operations

What if we have mixed sequences of operations? Amortized bounds: assign an amortized cost per operation type, "preserve the sum" over all operations.

→ want: upper bound on actual cost

$$\underbrace{\sum_{op.} \text{amortized cost}}_{\text{amortized bound}} \geq \sum_{op.} \text{actual cost}$$

→ know nothing about actual per-operation cost, but gain information about total cost

→ we define amortized cost such that we form an amortized bound somehow

C (aggregate) binary trees: consider create-empty/insert/delete on binary trees.

Recall: create-empty: $O(1)$ worst-case; insert/delete: $\Theta(\log n)$

So say we have $c, i, d \in \mathbb{N}$ of each operation, starting empty.

Notice $d \leq i$ since we can only delete at most as many times as we insert.

$\Rightarrow$ total cost $O(c + i\log n^* + \underbrace{d}_{\leq i}\log n^*) = O(c + i\log n^*)$ so amortized bounds

create-empty $O(1)$
insert $\Theta(\log n)$.
delete $\varnothing$ ⇝ dub with insert

* but n changes, so how can we analyze properly? order of operations also matters
(eg. create → i → i → … → i → d → d → … → d expensive, create → i → d → i → … → d cheap)

Accounting Method

Imagine you have a bank account, and that every operation can store credit in that account (always ≥ 0 balance).

- allow operations to add credit to account.
- allow operations to pay for time using stored credit.

} eg. binary tree insertion adds $\log n^*$ money to account; deletion removes $\log n^*$ money from account

* Ex. show valid amortized bound.

* calculate amortized operation cost = actual operation cost + deposits − withdrawals → we can form amortized bound.

C (account) (continued): we claim that insert = $O(\log n)$ and delete = $\emptyset$ amortized.

Structure: add 1 coin worth $\Theta(\log n^*)$ per insert; consume 1 coin for each delete } amortized cost of operation = actual cost + deposits − withdrawals

Generalize: say n is the current structure size.

insert: deposit 1 coin worth $\log n$, $\Rightarrow$ amortized $\log n + \log n - 0$ $\in \Theta(\log n)$ amortized

delete: withdraw one coin worth $\log n$. $\Rightarrow$ amortized $\log n + 0 - \log n$ $\in \emptyset$ amortized

invariant: a structure of size n always consists of an account A with $\{\log 1, \log 2, \ldots, \log(n)\} \subseteq A$.

* include $\log n$ or not?

Proof: Induction. True for $n=0$. Say true for n.

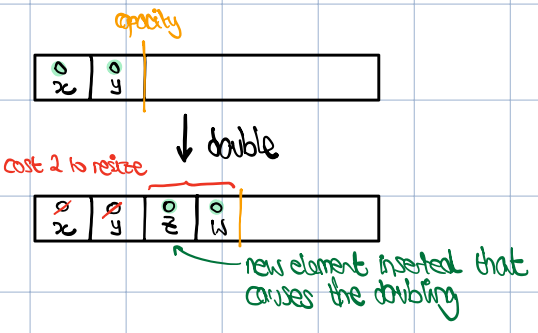
If we insert, then n++, $A \leftarrow A \cup \{\log(n+1)\}$ so holds. ($\cup \{\log 1, \ldots, \log(n)\}$) If we delete, then $A \leftarrow A \setminus \{\log(n)\}$, n-- holds. □

D (account) hash-table doubling (insertion-only)

When we insert an item, put a coin worth $c = O(1)$ on the item.

When doubling, use existing coins to pay for doubling operation. (last $\frac{n}{2}$ items have coins *)

$\Rightarrow$ table has at most half coins (at most $n/2$ coins).



amortized doubling: $\Theta(n) - c\frac{n}{2} = 0$ for c large enough constant

(actual; withdrawals)

* we would have to prove some spatial property/invariant to prove that each operation charged at most once

Charging Method

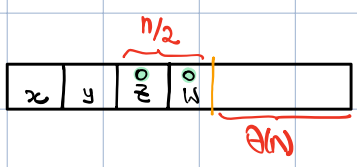
- allow operations to charge cost retroactively to the past (not future).

* calculate amortized operation cost = actual cost − total charge to past + total charge in future.

* Ex. show valid amortized bound.

D (charge) hash-table doubling (insertion-only)

For every doubling operation, charge all the insertions made since the last doubling.



$\rightsquigarrow$ uniformly charge each $\Theta(1)$. → key: each insert is only charged at most once over all time

$\Rightarrow$ amortized insert = $\Theta(1) - 0 + \Theta(1) \in \Theta(1)$

amortized double = $\Theta(n) - \frac{n}{2}\Theta(1) + 0 \in \emptyset$ zero cost.

D (charge) hash-table doubling and halving (insertion and deletion): double at capacity, halve at $\frac{1}{4}$ capacity

$\Rightarrow$ after any doubling or halving, our structure is $\frac{1}{2}$ full ("nice state"). Say $n = \Theta(n)$ capacity, where n items.

Suppose we are at the "nice" equilibrium state (50% full). Perform some sequence of insert/delete.

At some point, we reach a state of either 25% or 100%.

25%: we must have deleted 25% of items, i.e. $\geq \frac{n}{4}$ deletions.

100%: we must have inserted 50% of items, i.e. $\geq \frac{n}{2}$ insertions.

$\Rightarrow$ charge having to $\geq \frac{n}{4}$ deletions since last resize so charge $\Theta(1)$ per op.
charge doubling to $\geq \frac{n}{2}$ insertions since last resize so charge $\Theta(1)$ per op.
} each resize only charges past time window * of operations made (so each op. charged at most once)

* unlike accounting, we only prove a temporal property/invariant here.
$\Rightarrow$ each op. charged at most once is trivial to observe

Potential Method

idea is to smooth out the analysis of some algorithm

Define a potential function Φ mapping data-structure configuration $\longrightarrow \mathbb{N}_{\geq 0}$. Typically we use size $n \in \mathbb{N}_{\geq 0}$

Therefore $\Phi: \mathbb{N}_{\geq 0} \longrightarrow \mathbb{N}_{\geq 0}$, $n \longmapsto \Phi(n)$. We also constrain Φ to satisfy $\Phi(0)=0$.
↳ empty structure potential

potential difference $\Delta\Phi(n)$

* calculate amortized operation cost = actual cost + $(\Phi(n) - \Phi(n-1))$.
(Φ(after op.) − Φ(before op.))

$\Rightarrow$ amortized cost of n operations = $\sum_{i \leq n}$ amortized cost of i^{th} operation
↑ valid amortized bound
$= \sum_{i \leq n}$ actual i^{th} cost + $\sum_{i \leq n} (\Phi(i) - \Phi(i-1))$
telescoping
$=$ actual n operations cost + $\Phi(n) - \cancel{\Phi(0)}^{0} \geq$ actual n operations cost.

Choosing a good Φ is hard. In general, we want expensive i^{th} op. to have $\Delta\Phi(i) < 0$ to such an extent that it nullifies the effect of the actual expensive cost.

E Binary counter
potential

```
increment(A):                    ← array of bits
    i ← 0
    while i<len(A) and A[i]=1:
        A[i]=0   ← flip the bit 1→0
        i++
    if i<len(A):
        A[i]=1   ← flip 0→1
```

e.g. 00110101111 → (increment) → 00110111000 at most n bits change

In an amortized sense, how many bits change?

We can use the potential method.

Notice an increment has a very dear cost: $O(1 + \#(\text{trailing 1s}))$. We want constant amortized cost.

Try: Φ = #trailing 1s. But then 11110 has cost $O(1) + \underbrace{\Phi(11111)}_{\Theta(n)} - \underbrace{\Phi(11110)}_{O(1)} = \Theta(n)$ amortized (bad).
(won't work)

Define Φ(bitstring) = #1-bits. Then notice $\Delta\Phi$ is at most 1 since we add at most one 1-bit.

However for expensive operations (e.g. $\underset{\Phi=4}{01111} \longrightarrow \underset{\Phi=1}{10000}$) we have $\Delta\Phi$ large negative (expensive ops).

In general, t trailing 1s $\Rightarrow$ increment destroys t 1s, creates one 1, so $\Phi(i) = \Phi(i-1) - t + 1$
$\Rightarrow \Delta\Phi(i) = -t+1$

amortized op. cost = $\underbrace{(1+t)}_{\text{actual cost}} \underbrace{-t+1}_{\Delta\Phi(op.)} = 2$ constant amortized.

Ex. Consider the amortized cost of splitting on a 2-3 tree. The "problem cases" are where the tree has too many 3-nodes.
So we define Φ(state) = #3-nodes.

One strategy to define suitable Φ is to look for bad state properties.

E potential Stack-1 (Potential)

Consider a stack with three operations:
- push(S, x) $\Theta(1)$
- pop(S) $\Theta(1)$
- multi-pop(S, k) $\Theta(\min(|S|, k))$

} what are amortized operation costs?

Define potential $\Phi(S) = |S|$. Clearly $\Phi(0) = 0$ and $\Phi(S) \geq 0$ so Φ is a valid potential function.

POP: amortized cost = $\underbrace{\text{actual cost}}_{c_{pop}\text{ constant}} + (\underbrace{\Phi(S_{new})}_{|S_{new}|} - \underbrace{\Phi(S_{old})}_{|S_{old}|}) = \underbrace{c_{pop}}_{1} + \underbrace{|S_{new}|}_{|S_{old}|-1} - |S_{old}| = 0.$

PUSH: amortized cost = $\underbrace{\text{actual cost}}_{c_{push}\text{ constant}} + (\underbrace{\Phi(S_{new})}_{|S_{new}|} - \underbrace{\Phi(S_{old})}_{|S_{old}|}) = \underbrace{c_{push}}_{1} + |S_{old}| + 1 - |S_{old}| = 2.$

MULTI-POP: $\underbrace{\text{actual cost}}_{\min(|S_{old}|, k)} + \underbrace{\Phi(S_{new})}_{|S_{old}| - \min(|S_{old}|, k)} - \Phi(S_{old}) = 0.$
(#elements popped)

E account Stack-2 (Accounting)

Intuitively, elements can only be popped if they were inserted in the past. Thus, charge insertions now for their potential future deletion. Define:

- PUSH: charge \$1, deposit \$1 $\Rightarrow a_{cost} = \underbrace{\text{actual cost}}_{1} + \underbrace{\text{deposits}}_{1} - \underbrace{\text{withdrawals}}_{0} = \$2$
- POP: withdraw \$1 $\Rightarrow a_{cost} = 1 + 0 - 1 = \$0.$
- MULTI-POP: withdraw \$$\min(|S|, k)$ $\Rightarrow a_{cost} = \min(|S|, k) + 0 - \min(|S|, k) = \$0.$

It suffices to show that our bank balance $B \geq 0$ regardless of operation order. *Easy exercise

General Insertion/Deletion Structures

When we have a structure where single-element insertion costs $c_{insert}(n)$ and deletion costs $c_{delete}(n)$, we can generally charge the more expensive operation the cost of both.

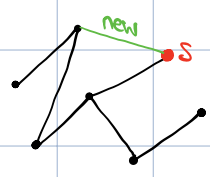
Things become tricky when the cost of deletion varies (eg. is context-dependent).

Dynamic BFS Distances

Consider directed G with fixed V and dynamic E (initially $E=\emptyset$).

Let $s \in V$ be some fixed source (constant over the sequence of operations). We can only add to E.

After an insertion $(u,v) \in E$ we update:
1. The BFS tree rooted at source $s \in V$
2. The distance $dist(x)$ from $s \to x$ for $x \in V$



Since naive BFS recomputation is slow, we want a better update algorithm.

Efficient BFS Update

Notice that an insertion $uv \in E$ has 2 possibilities:
1) $dist(v)$ reduces. ← $dist(u)$ cannot reduce since $uv \in E$ is a directed edge
2) $dist(v)$ is unchanged.

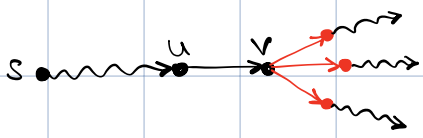
The new candidate path to v goes through u:  with length $dist(u)+1$.

$\Rightarrow$ we update all out-neighbours of v using a directed BFS

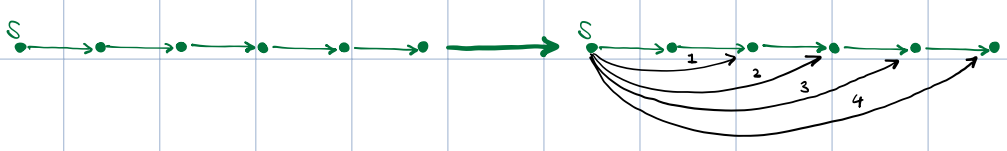
← update(G) function

```
if dist(v) ≤ dist(u)+1: return   ← no change condition
dist(v) ← dist(u)+1
Q = queue([v])
while Q ≠ ∅:
    x = Q.pop()
    for outgoing edge (x,y) ∈ E:
        if dist(y) > dist(x)+1:
            dist(y) ← dist(x)+1
            Q.push(y)
```
BFS

Notice this algorithm may have cost $\Omega(m)$ for particular updates (worst case).



Remark: Notice amortized update cost is not $O(1)$ since we can form an adversarial example:



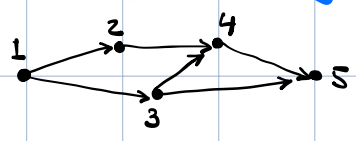
adding edges in specified order.

Lemma: G directed as above, $E=\emptyset$. For all $v \in V$ such that $\exists$ s-v path, say $dist(v) \in \{0,1,\dots,D\}$ for some $D \in \mathbb{N}$ constant independent of $|V|, |E|$, over n edge insertions. Then update(G) $O(1)$ amortized.

Incremental Topological Ordering

Let G be directed and acyclic (DAG) such that V is fixed and $|V|=n$.

A topological ordering of G is an enumeration $[v_1,\dots,v_n]$ of V such that $v_i v_j \in E \Rightarrow i<j$.



In general, topological orderings are not unique.

Suppose $\tau = (v_1,\dots,v_n)$ is a topological ordering for G we must maintain.

Consider the insertEdge(u,v) operation that:
1) adds $uv \in E$ (with the guarantee that G remains a DAG)
2) if τ satisfies $pos[u]<pos[v]$ do nothing
3) else edge $u \to v$ violates τ, so repair τ by reordering.

```
reorder(uv, τ):
    i = pos(u), j = pos(v)   so uv ∈ E but i ≥ j
    S = {τ[k] | j ≤ k ≤ i}
    R = {x ∈ S | ∃ u→x path in G}        } O(n)
    construct τ' such that:
        R appears after S\R in τ'
        everything else follows τ        } O(n)
    τ ← τ'
```
reorder algorithm for τ

correctness

Lemma: Upon insertEdge(u,v), calling reorder(uv, τ) maintains a valid top. ordering τ.

Proof: Let G be original DAG, and $G'=G+uv$, and τ' be the resulting reordered τ.

If $pos(u)<pos(v)$ then $\tau'=\tau$ and τ valid on $G \Rightarrow \tau'$ valid on G'.

Say $pos(u) \geq pos(v)$.

Define $S = \{\tau[k] \mid pos(v) \leq k \leq pos(u)\}$ and $R = \{x \in S \mid \exists u\text{-}x \text{ path in } G\}$.

For $y \in S \setminus R$ notice $\nexists u\text{-}y$ path in G, and $pos(u) \geq pos(y)$ in $\tau \Rightarrow \nexists y\text{-}u$ path in G.

$\Rightarrow$ relative ordering of $y \in S \setminus R$ and u does not matter.

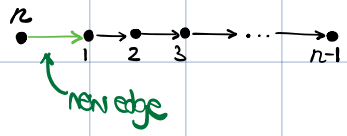
Let $z \in R$ so $\exists u\text{-}z$ path in $G \Rightarrow \nexists z\text{-}y$ path in G (else $\underbrace{u \rightsquigarrow z \rightsquigarrow y}_{u\text{-}y \text{ path}}$ ↯).

worst case $\exists y\text{-}z$ path

$\Rightarrow \underbrace{pos(y)}_{S \setminus R} < \underbrace{pos(z)}_{R}$ suffices to satisfy relations between R and $S \setminus R$

Therefore, moving $S \setminus R$ before R in τ' whilst maintaining internal R and $S \setminus R$ orders from τ defines a valid topological ordering τ' on G. □

Lemma: reorder(uv, τ) is worst-case $\Omega(n)$.

Proof: Consider 

requiring $\Omega(n)$ elements moved. □

amortized analysis

Lemma: Over m operations, the amortized cost of reorder() is $O(n)$.

Proof: Show that each operation is $O(n)$. □

Fibonacci Heap

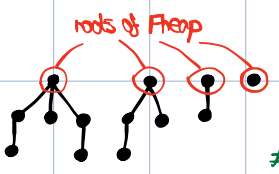
We introduce and analyse the amortized cost for Fibonacci heap operations.

The min-oriented Fheap supports:

find-min()	insert(x)	delete-min()	decrease-key(x, newkey)	union(H_1, H_2)
$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$ ← amortized

Notice the runtime of the operations above is quite different from a binary heap.

The main idea is an Fheap H is a collection of trees each satisfying the heap property.



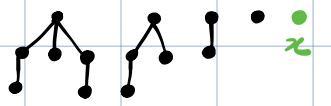
each node stores its degree (# children)
mark (whether a node is "marked")

We also define $t(H)$, $d(n)$, $m(H)$.
(#trees in H; max. node degree for heap with n nodes; #marked)

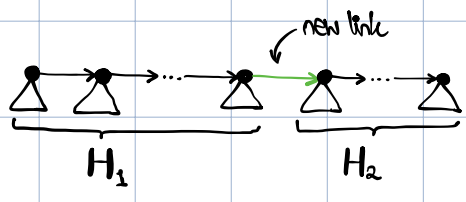
Lemma: $d(n) \leq \log_\gamma(n) \in \Theta(\log n)$ where γ is the golden ratio.

Proof: Beyond scope. □

find-min(): We can store/memoize the minimum value of $H \Rightarrow O(1)$ find-min.

insert(x): Add x as a new root to H  $\Rightarrow O(1)$ insertion.

union(H_1, H_2): Collect all trees of H_1 and H_2 into a new $H_1 \cup H_2$.
If trees are linked using a LL, this can be $O(1)$. (and update min)



delete-min(): 1. Delete minimum node and add its child subtrees as new trees in H. $O(\log n)$ since at most $d(n)$ nodes are children

2. Combine trees so no roots of trees have the same degree.
   - Iterate through roots and collect degrees in table D. Upon collision, merge colliders and update D with new degree.
   - at most $d(n) + t(H)$ roots (new + old) and at most $O(d(n))$ combines $\Rightarrow O(\log n + t(H))$
3. Update min.
   - $O(\log n)$ because only $O(\log n)$ trees after step 2

$\Rightarrow$ delete-min has worst-case $O(\log n + t(H))$.

decrease-key(x, K): decrease key[x] to K where $K \leq$ key[x].

```
key[x] ← K
y ← parent[x]
if y not None and key[x] < key[y]:        # heap order violation
    move x to be a new root               } cut x from y
    degree[y]--
    parent[x] = None
    mark[x] = False
cascade:
    while y not a root:
        if mark[y] is False, set mark[y] = True and exit
        else:  <- y is marked => fragile (could be cut)
            z ← parent[y]
            cut y from z (so y becomes a new root)
            y ← z  (continue on z)
    update H.min
```

---

marked nodes are nodes that have lost one child
$\Rightarrow$ they are somewhat "fragile"

also notice a node cannot lose >1 child without itself being cut to the root, so nodes cannot lose children too aggressively.

Lemma: (delete-min amortized) The amortized cost for delete-min over k operations is $O(\log n)$.

Proof: This is an interesting analysis. We use the potential method.

For delete-min, notice { "roots are a mess" that must be cleaned later.
marked nodes are "fragile" and may get cut easily.

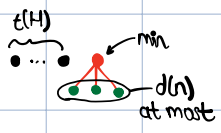
$\Rightarrow$ $t(H)$ (roots) and $m(H)$ (marked) encode bad state properties.

(One strategy to define suitable ϕ is to look for bad state properties.)

Thus define $\phi(H) = t(H) + 2m(H)$ so clearly $\phi: S \to \mathbb{N}_{\geq 0}$ and $\phi(0) = 0$.

We aim to analyze $c_{am} = c_{actual} + \Delta\phi$ for delete-min.

Notice upon delete-min, we create $O(d(n))$ new roots $\Rightarrow t(H_{new}) - t(H_{old}) \leq d(n)$.
$m(H)$ remains constant at worst case, so $m(H_{new}) - m(H_{old}) \leq 0$.



$\Rightarrow \Delta\phi = \phi(H_{new}) - \phi(H_{old}) \leq d(n)$.

After deleting the (dangling) min. node, we have at most $t(H) + d(n) - 1$ trees in H_{new}. (old, cut nodes, deleted min.)
Then we combine same-degree trees. Notice upon combining, $t(H)$ reduces by 1 and $m(H)$ at worst remains constant (cannot increase).

Claim: $t(H_{new}) \in O(\log n)$ after "combining" same-degree roots.

Notice $d(n) \in O(\log n)$ and each root has a distinct degree in $[0, \dots, d(n)] \Rightarrow O(\log n)$ roots. Δ

Therefore, if $t(H)$ is large, then $t(H_{new}) \in O(\log n) \Rightarrow \Delta\phi$ sufficiently kills $t(H)$.

$$\Rightarrow c_{am} = O(\log n + t(H)) + 2(m(H_{new}) - m(H)) + (t(H_{new}) - t(H))$$
$$\leq \Theta(\log n) + \Theta(t(H)) + \Theta(\log n) - \Theta(t(H)) \in \Theta(\log n). \quad \square$$

showing these cancel needs a bit more algorithmic rigour

NP-Hardness, Completeness, and Problem Reduction

Once we have learned more and more algorithms, they become our building blocks, and we may not need to design new algorithms from scratch every time. new problem $\overset{\text{reduction}}{\rightsquigarrow}$ known problem

we are interested in finding problem reductions

Decision Problems

To formalize the notion of a reduction, it is convenient to restrict our attention to decision problems, where the outputs are YES and NO.

Ex. Finding max. graph matching $\xrightarrow{\text{decision}}$ "does G have a size-k matching?"

For all the problems we will consider, solve decision problem in polynomial time $\Rightarrow$ solve search problem in polynomial time

↓

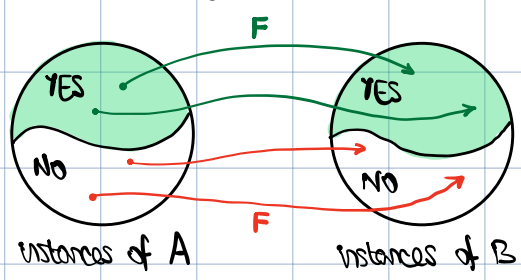
use decision algorithm as a subroutine

Let A be a decision problem. We say A is polynomial time reducible to a decision problem B if there is a polynomial time algorithm F that transforms $I_A \xmapsto{F} I_B$ such that $I_A \equiv \text{YES} \iff I_B = F(I_A) \equiv \text{YES}$.

(I_A: instance of A; I_B: instance of B)

We use $A \leq_P B$ to denote the existence of such a reduction.

Intuitively, this says A is not more difficult than B in terms of polynomial time solvability.



Essentially, say D_B is a polynomial-time decider for B, and D_A is some decider for A.

Then $I_A \in D_A^{-1}(\text{YES}) \iff D_B(\underbrace{F(I_A)}_{A \leq_P B}) = \underbrace{D_B(I_B)}_{\text{polynomial time}} = \text{YES}$.

so $I_A \in D_A^{-1}(\text{YES}) \iff I_B = F(I_A) \in D_B^{-1}(\text{YES})$.

Conversely, say $A \leq_P B$ and it is known that A cannot be solved in polynomial time. Then B cannot either.

Lemma (P-reduction): A and B decision problems. Then $A \leq_P B$ and B polynomial $\Rightarrow$ A polynomial.

Lemma (NP-reduction): A and B decision problems. Then $A \leq_P B$ and A non-polynomial $\Rightarrow$ B non-polynomial.

How can we prove that a problem is not solvable using a polynomial-time algorithm?

Ex. Consider traveling salesman problem TSP, and say this is almost-surely NP.
To show problem C is almost-surely NP, we can show $TSP \leq_P C$.

We will start with the following 3 problems and show they are equivalent in terms of polynomial-time solvability.

1 Maximum Clique (Clique): A subset $S \subseteq V$ is a clique if $u, v \in S \Rightarrow uv \in E$.

Input: $G = (V, E)$ and $k \in \mathbb{Z}$. Output: Is there a clique in G with $\geq k$ vertices?

2 Maximum Independent Set (IS): A subset $S \subseteq V$ is independent if $u, v \in S \Rightarrow uv \notin E$.

Input: $G = (V, E)$ and $k \in \mathbb{Z}$. Output: Is there an IS in G with $\geq k$ vertices?

3 Maximum Vertex Cover (VC): A subset $S \subseteq V$ is a vertex cover if $uv \in E \Rightarrow \{u, v\} \cap S \neq \emptyset$. (use or vee)

Input: $G = (V, E)$ and $k \in \mathbb{Z}$. Output: Is there a vertex cover for G with $\leq k$ vertices?

Proposition: Clique $\leq_p$ IS and IS $\leq_p$ Clique.

Proof: Clique → IS needs us to change edges to non-edges,
and IS → Clique needs us to change non-edges to edges.

Clique $\leq_p$ IS. To solve clique on G, construct complement $\bar{G}$ and solve IS on $\bar{G}$,
so $uv \in E \Leftrightarrow uv \notin \bar{E}$. Clearly, the reduction (constructing $\bar{G}$) is polynomial. Δ

IS $\leq_p$ Clique. Same construction. Δ

Therefore $\{G, k\} \in \text{Clique}^{-1}(\text{YES}) \Leftrightarrow \{\bar{G}, k\} \in \text{IS}(\text{YES})$. $\square$

Lemma: $S \subseteq V$ is a vertex cover for $G \Leftrightarrow V \setminus S$ is independent in G.

Proof: ($\Rightarrow$) $S \subseteq V$ vertex cover $\Rightarrow$ for $uv \in E$ we have $\{u, v\} \cap S \neq \emptyset$. SFAC $V \setminus S$ not independent.
$\Rightarrow \exists u, v \in V \setminus S$ with $uv \in E$. So $\exists uv \in E$ with $u, v \in V \setminus S$ so $\{u, v\} \cap S = \emptyset$. $\lightning \Delta$

($\Leftarrow$) $V \setminus S$ independent $\Rightarrow$ all $uv \in E$ satisfy $\{u, v\} \cap S \neq \emptyset$ since $u, v \in V \setminus S \Rightarrow uv \notin E$. $\Delta\square$

Proposition: VC $\leq_p$ IS and IS $\leq_p$ VC.

Proof: Since $S \subseteq V$ vertex cover $\Leftrightarrow V \setminus S$ independent, we have that $\exists |S|$ vertex cover (S can always be made bigger) $\Leftrightarrow \exists |V \setminus S|$ IS.

G has a VC of size $\leq k \Leftrightarrow G$ has an IS of size $\geq n-k$.

$\Rightarrow$ the reduction maps $\{G, k\}$ VC $\mapsto \{G, n-k\}$ IS. Clearly polynomial time. $\square$

Lemma ($\leq_p$ Transitive): If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.

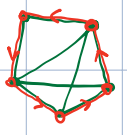
Proof: Say $A \xmapsto{F} B$ and $B \xmapsto{H} C$. Then $A \xmapsto{H \circ F} C$ and $H \circ F$ polynomial reduction. $\square$

Therefore, Clique, IS and VC are equivalent in polynomial-time solvability.

The above is an example of simple problem reductions. We see more reductions below.

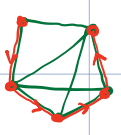
1 Hamiltonian Cycle (HC): A cycle is a Hamiltonian cycle if it touches every vertex exactly once.

Input: $G = (V, E)$ undirected. Output: Does G have a Hamiltonian cycle?



2 Hamiltonian Path (HP): A path is a Hamiltonian path if it touches every vertex exactly once.

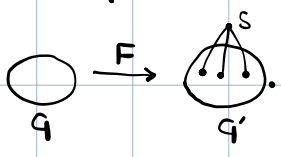
Input: $G = (V, E)$ undirected. Output: Does G have a Hamiltonian path?



It is not surprising that HC $=_p$ HP but is a good exercise to formalize the reductions.

Proposition: HP $\leq_p$ HC.

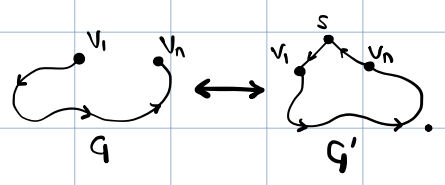
Proof: Given $G = (V, E)$ for HP, construct $G' = (V \cup \{s\}, E')$ by adding a new vertex s that neighbours every $v \in V$.



Clearly, reduction F runs in polynomial time. We claim G has HP $\Leftrightarrow G'$ has HC.

($\Rightarrow$) G has HP $= v_1 \dots v_n$, then $\underbrace{sv_1}_{\in E'} \dots \underbrace{v_n s}_{\in E'}$ is a HC in G'.

($\Leftarrow$) G' has HC $\Rightarrow$ there is a HC $sv_1 \dots v_n s$ by choosing to start at $s \in V' \Rightarrow v_1 \dots v_n$ HP in G'. $\square$
($\subseteq E$ since HC $\Rightarrow$ s only visited once)

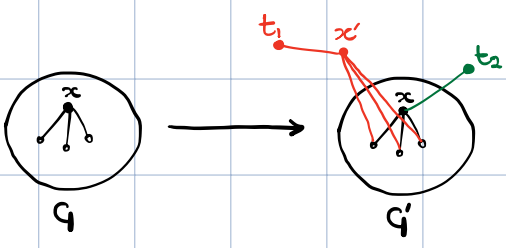


Remark: It may be tempting to ask why the above $\Rightarrow HC \leq_p HP$ directly. Note that we have built a reduction G for HP $\xmapsto{F}$ G' for HC, so given G for HP, we know how to build G' for HC.

However, given G' for HC, we don't yet know how to reduce to G for HP (reverse).

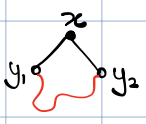
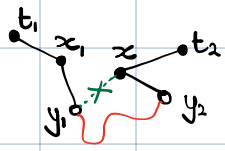
Proposition: $HC \leq_p HP$

Proof: Given $G=(V,E)$ for HC, choose arbitrary $x \in V$ and construct G' by duplicating x and two vertices:

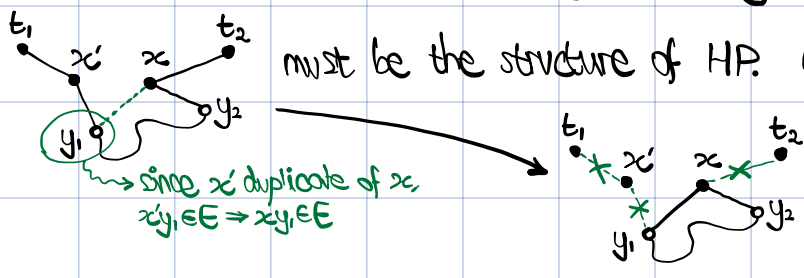


clearly, this reduction runs in polynomial time.

We claim that G has HC $\Leftrightarrow$ G' has HP.

($\Rightarrow$) G has HC  → construct  HP on G'.

($\Leftarrow$) G' has HP. Since $t_1, t_2 \in V'$ are degree-1, they must be endpoints of HP and $t_1 x' \in HP$.

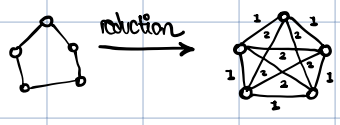


must be the structure of HP. Clearly we can construct a HC in G:

as shown. □

One technique to do reduction is to show that problem A is a special case of problem B.

This technique is called specialization.



Proposition: $HC \leq_p TSP$.

Proof: Given $G=(V,E)$ for HC, construct weighted G' such that $G'=(V,E')$ is complete, and for $uv \in E'$ we have $w_{uv} = \begin{cases} 1 \text{ if } uv \in E \\ 2 \text{ if } uv \notin E \end{cases}$. Then G has HC $\Leftrightarrow$ G' solves TSP with a tour of cost n. □

3-SAT and nontrivial reductions

The 3-SAT problem is important in the theory of NP-completeness.

In this problem we have n boolean variables $x_1, \dots, x_n$ that can each be set to True or False.

We are also given a formula in CNF (CS 245) where literals are either x_i or $\bar{x}_i = \neg x_i$.

$\underbrace{(x_1 \vee x_2 \vee \bar{x}_3)}_{\text{clause}} \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_3)$. } each clause has at most 3 literals and the formula only has x_1, x_2, x_3

3-Satisfiability (3-SAT):

Input: CNF-formula where each clause has at most 3 literals.

Output: Is there a truth valuation for $x_1, \dots, x_n$ that satisfies all the clauses?

This problem looks very different from the ones seen so far. Finding reductions between seemingly unrelated problems often requires new ideas.

Theorem: 3-SAT $\leq_p$ IS.

Proof: Given a 3-SAT formula, we would like to construct a graph G such that the formula is satisfiable $\iff$ G has an independent set of a certain size. Let F denote the CNF.

Idea: For each literal in F, construct a new vertex in G.

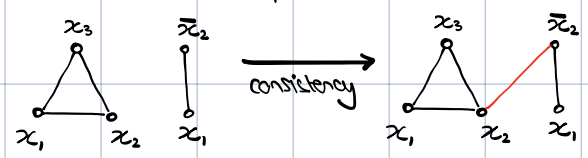
We want satisfying truth valuation $\longleftrightarrow$ independent set in G.

For F (CNF) to be True, at least one literal in every clause of F must be true, but we must be consistent in literal valuations across clauses.
(both x_i and $\bar{x}_i$ cannot be True, exactly one is True)

$\Rightarrow$ for each appearance of x_i and $\bar{x}_i$, add an edge between them, so x_i and $\bar{x}_i$ cannot exist in the same independent set.

We also add edges between literals of the same clause $\rightsquigarrow$ the idea here seems to be to reduce redundant "true" valuations where possible

so each clause can have at most one literal in the independent set.

$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2)$ $\xrightarrow{\text{clauses}}$  $\xrightarrow{\text{consistency}}$

$\Rightarrow$ G has two types of edges: clause-edges and consistency edges.

Claim: Say F has k clauses. Then F satisfiable $\iff$ there is an IS of size k.

The intuition is above: we constructed G such that each clause can have at most one vertex in the IS, and if x_i is in the IS then no other occurrence of $\bar{x}_i$ can be.

($\Rightarrow$) Say there is a satisfying truth valuation. Then at least one literal in each clause is True $\Rightarrow$ choose exactly one per clause to form a set $S = \{x_{a_1}, \dots, x_{a_k}\}$ of size k.

We show S is independent. Suffices to show that $uv \in E \Rightarrow \{u, v\} \not\subseteq S$. Say $uv \in E$.

1. uv clause-edge. Then $\{u, v\} \not\subseteq S$ since at most one vertex per clause is in S.
2. uv consistency-edge. The valuation $\{x_{a_1}, \dots, x_{a_k}\}$ well-defined $\Rightarrow$ $\{x_i, \bar{x}_i\} \not\subseteq S$.

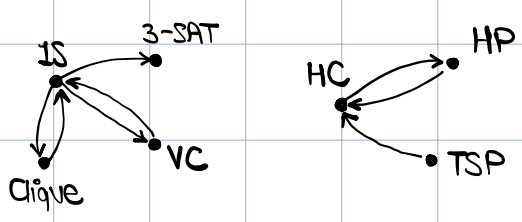
Therefore S independent of size k. $\triangle$

($\Leftarrow$) Let $S \subseteq V$ be independent of size k. By construction, each clause has exactly one vertex (literal) in S, and $x_i \in S \Rightarrow \bar{x}_i \notin S$ since S independent and $x_i \bar{x}_i \in E$.

Setting each $x_i \in S$ True gives a truth valuation where each clause of F is True $\Rightarrow$ satisfied. $\triangle$

The graph construction is clearly polynomial in n, so 3-SAT $\leq_p$ IS from the construction above. $\square$

We have introduced the notion of polynomial time reduction. So far, we have



where $A \to B$ means $B \leq_p A$.

In principle, we can keep growing this computational web by applying reductions for more and more problems.

If we define the relation $A =_p B$ to mean $A \leq_p B$ and $B \leq_p A$, we can show that $=_p$ defines an equiv. relation.

What if we could identify the hardest problem X of a large class, so for a new problem Y, it suffices to show that $X \leq_p Y$ to show that Y is at least as hard as others in the class?

NP class

Let A be a decision problem. Determining whether an instance I_A is satisfiable may be hard, but given some "solution" it may be easy to check if the solution satisfies I_A.

Ex. $A = HC$, $I_A = G$ graph. Given a cycle C we can "easily" check if it is Hamiltonian.

Let X be a problem, and represent each instance of X by some binary string s.

Then $X \in NP$ if there is a polynomial time verifier B_X such that

s is a YES-instance $\iff$ there is a proof t of length poly($|s|$) such that $B_X(s,t)$ is YES.

- s is a YES-instance: the problem $s = I_X$ is satisfiable
- t: binary string (typically a solution to I_A)
- $B_X(s,t)$: typically poly-time algorithm to verify that solution t "works"

Ex. Consider VC problem. Then

X = VC the problem (vertex cover of size $\leq k$)

s = (binary representation of some graph G) the input string

t = (binary representation of $S \subseteq V$ with $|S| \leq k$)

$B_X(s,t)$ checks if t is a vertex cover for G

* Exercise: write B_X to show that VC $\in$ NP.

* Ex. VC, IS, Clique, HC, HP, subset-sum, 3-SAT $\in$ NP.

Remark: Say the problem is to determine if a graph G is non-Hamiltonian (i.e. no Hamiltonian cycles). (HC $\in$ NP but nHC $\in^?$ NP)

non-example

Whether nHC $\in$ NP is unsolved. We don't know much better than to enumerate all candidate cycles and check.

Remark: The problem of whether a graph has no bipartite matching of size $> k$ can be efficiently validated by checking any vertex cover of size $\leq k \Rightarrow$ a fast verifier exists. Thus nBM $\in$ NP and BM $\in$ NP.

co-NP

A problem X with polynomial verifiers for YES- and NO-instances are in NP $\cap$ co-NP.

Clearly, $P \subseteq NP$, where P is the class of problems solvable in polynomial time.

NP is the class of problems solvable by a non-deterministic polytime (NP) algorithm. * as long as there is a short solution, a non-deterministic machine will "magically" find the solution.

Whether $P = NP$ or $P \subset NP$ is unsolved.

NP-completeness

A problem X is NP-complete if $Y \leq_p X$ for all $Y \in NP$. (X is a "hardest" problem in NP)

Proposition: $P = NP \iff$ some NP-complete problem can be solved in polynomial time. * Trivial by def'n.

Once we know one problem is NP-complete, we can use reductions to show that other problems are too.

But how do we prove NP-completeness for that first problem?

Theorem (Cook-Levin): 3-SAT is NP-complete. * We prove this below.

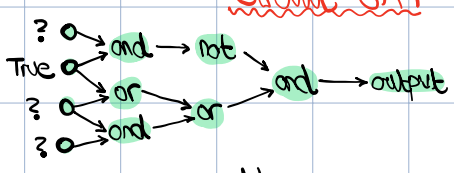
Therefore, since 3-SAT $\leq_p$ IS $=_p$ VC $=_p$ Clique, we have: **Corollary:** IS, VC and Clique are NP-complete.

To show some problem X is NP-complete, we must show $X \in NP$ then show $Y \leq_p X$ for some NP-complete Y.

The experience of proving $Y \leq_p X$ is often magical. We don't even know what Y to choose to reduce, and then we must find some polynomial reduction of Y into our target X.

Proof (Cook-Levin): We will prove that 3-SAT is NP-complete. We will introduce some intermediate problems to do so.

Circuit-SAT Input: circuit with AND/OR/NOT gates with some subset of unknown boolean inputs.
Output: is there a truth valuation to the unknown subset that outputs True?



We can assume the circuit is a DAG with each AND/OR gate having two incoming edges.

Claim: Circuit-SAT is NP-complete.

Proof sketch: The statement is very abstract, so we must start from definitions. We show $X \leq_p$ C-SAT $\forall X \in NP$.

$X \in NP \overset{\text{def.}}{\Rightarrow} \exists B_x$ polytime verifier for YES-instances such that s is a YES-instance, $\exists t$ (proof) short such that $B_x(s,t)$ is YES. What is an algorithm?

If B_x runs in poly(|s|) then there is some machine executing B_x in poly(|s|) time

$\Rightarrow$ takes at most poly(|s|) atomic operations to execute B_x

$\Rightarrow$ there is a circuit of size poly(|s|) implementing B_x.

} this reduction is sketchy: we have to return to Turing machines and do a much more tedious reduction here

The main conceptual idea is that a circuit is as general as an algorithm.

The reduction above can be carried out in polynomial time poly(|s|). } *figure out details

So s YES-instance $\Rightarrow$ some proof t makes $B_x(s,t)$ YES $\Rightarrow$ circuit outputs True.

s NO-instance $\Rightarrow$ no input t to the circuit outputs True.

} $\Rightarrow$ C-SAT reduction

Thus $X \leq_p$ C-SAT, so C-SAT is NP-complete. $\triangle$

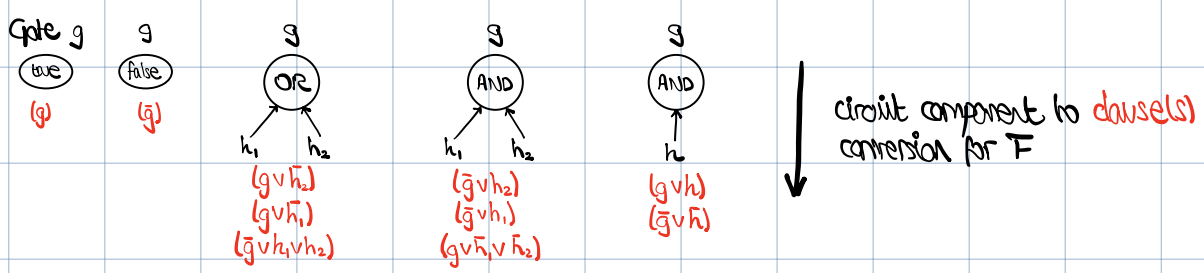
It remains to show that 3-SAT CNFs are as expressive as circuits from C-SAT.

Claim: C-SAT $\leq_p$ 3-SAT.

Proof: Given a circuit of n gates, we want a polytime reduction to a meaningful 3-CNF F for 3-SAT.

Let C denote the circuit. For each gate in C, we create one variable in F.

Then, we add clauses to F and modify it as shown:



Finally, to ensure that C outputs True only if C is True, add an output clause (o) to F.

This is a faithful simulation of C using F, and can be constructed in polytime using local circuit replacement at each construction step. $\triangle\square$

It remains to show that 3-SAT CNFs are as expressive as circuits from C-SAT.

Hard Graph Problems

We use reductions to show that Hamiltonian cycle and graph colouring are NP-complete.

Hamiltonian Cycle We introduce an intermediate problem for the reduction.

Directed Hamiltonian Cycle (DHC) Input: Directed $G=(V,E)$.
Output: Does G have a directed Hamiltonian cycle?

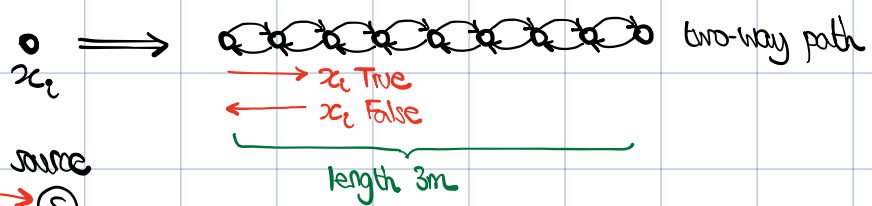
We show $3\text{-SAT} \leq_p \text{DHC} \leq_p \text{HC}$.

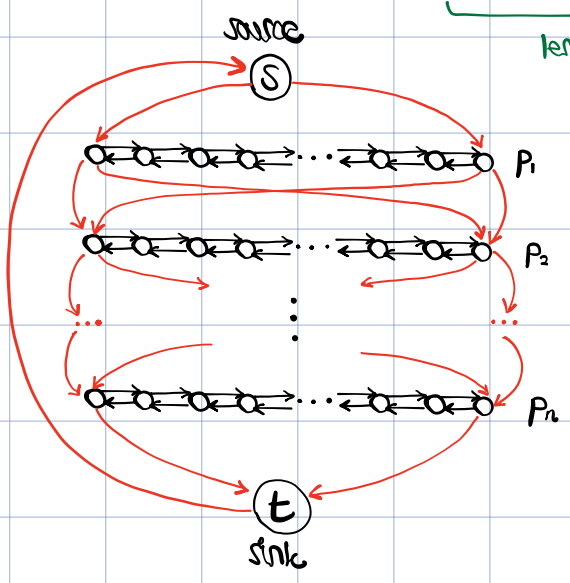
Theorem: DHC is NP-complete.

Proof: Clearly $\text{DHC} \in \text{NP}$. We show $3\text{-SAT} \leq_p \text{DHC}$.

Given a 3-SAT instance F of n variables $x_1,\dots,x_n$ and m clauses $c_1,\dots,c_m$ we would like to construct directed G such that F satisfiable $\iff$ G has a Hamiltonian cycle.

Idea: associate a long two-way path to a variable and traveling the path one way corresponds to setting the variable to True, other way corresponds to setting False.





At traversing each path, we can choose to go left→right or right→left.

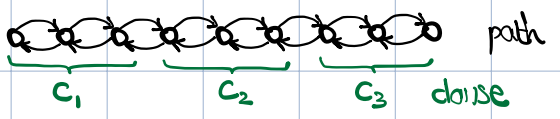
n binary choices

$\Rightarrow 2^n$ possible Hamiltonian cycles

$\Rightarrow$ bijection between truth valuations and Hamiltonian cycles.

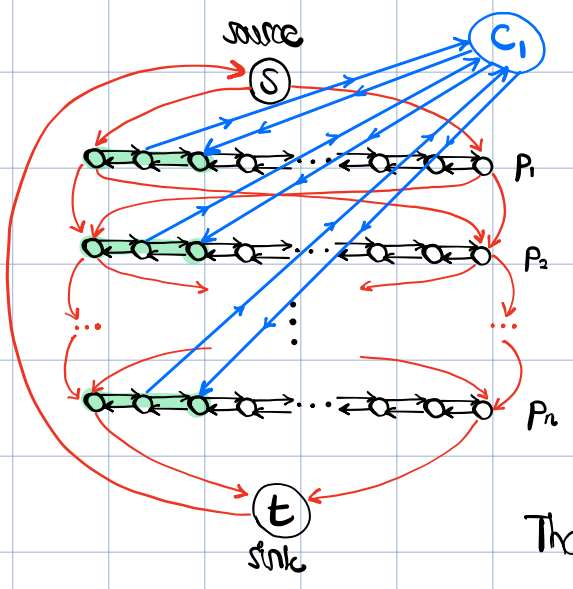
This is a good start.

Now, we would like to add some clause structures to "kill" all HCs that do not correspond to arrangements satisfying F. We decompose each path P_i:



For some clause, eg. $(x_1 \vee \bar{x}_2 \vee x_3)$, we want at least one of the literals to be True, so we want HCs to either: (1) move L→R in P_1, (2) move R→L in P_2, (3) move L→R in P_3 } we want a way to track which clauses our potential HC choice is able to satisfy

$\Rightarrow$ add clause vertices to G, and connect them to variable paths where meaningful.



if x_i appears in clause c_j, connect the corresponding decomposition in P_i to c_j L→R.

if $\bar{x}_i$ appears in clause c_j connect R→L.

traverse P_i L→R and $x_i \in c_j \Rightarrow c_j$ visited.
traverse P_i R→L and $\bar{x}_i \in c_j \Rightarrow c_j$ visited.

That's the whole construction for graph G.

Clearly, G can be constructed in polynomial time.

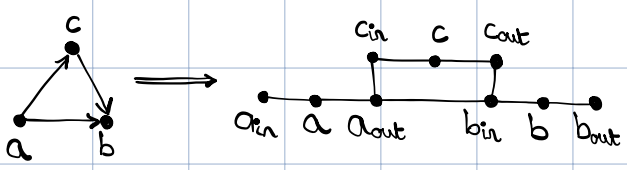
The argument above provides a rough sketch as to prove F satisfiable $\Leftrightarrow$ G has a DHC.

The details of formalizing this claim are left to the reader. □

To prove DHC $\leq_p$ HC, we will use length-3 paths to simulate directed edges.

Proposition: DHC $\leq_p$ HC.

Proof: Given directed G for DHC, construct undirected G' for HC using v_{in} and v_{out} vertices for direction.



Clearly, G' can be constructed in polynomial time.

G has a DHC $\Leftrightarrow$ G' has a HC.

($\Rightarrow$) Immediately true by construction: follow the cycle.

($\Leftarrow$) Start at v_{in}. We must go to v, since otherwise v becomes a dead-end. From v, the only place left to go is v_{out}. Then we must go to some w_{in} by construction. Process continues.

Reconstruct DHC on G based on the above structure by traversing $v \to w$. □

Corollary: TSP is NP-complete. Trivial: we proved earlier HC $\leq_p$ TSP.

Graph Colouring

Input: Undirected G, and $k \in \mathbb{N}$.
Output: Does G admit a k-colouring?

$k=1$: True $\Leftrightarrow E=\emptyset$ no edges.
$k=2$: True $\Leftrightarrow$ G bipartite.
$k=3$: NP-complete.

} graph colouring problem is very useful in resource allocation problems, eg. interval colouring

Theorem: 3-colouring is NP-complete

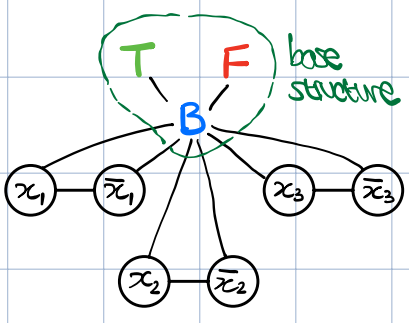
Proof: Clearly 3C $\in$ NP. We prove 3-SAT $\leq_p$ 3C. Say 3-SAT instance is CNF F with $x_1, \dots, x_n$ variables, $c_1, \dots, c_m$ clauses.

We want to construct G such that F satisfiable $\Leftrightarrow$ G 3-colourable.

Naturally, let one colour represent True and another represent False. Add $x_i \bar{x}_i \in E$ to ensure that not both are same truth valuation.

To enforce that x_i and $\bar{x}_i$ get the True/False colours,

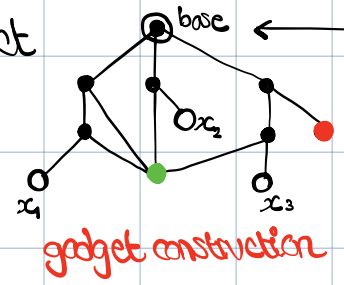
connect them to a base vertex structure (since we have 3 colours).



Now, we would like to add some clause structures to "kill" the 3-colourings that don't correspond to satisfying F.

We want some "gadget" for clauses such that the gadget is 3-colourable $\Leftrightarrow$ at least one of the clause literals are True.

Given $(x_1 \vee \bar{x}_2 \vee x_3)$ we can construct such that ⊙ base is only colourable if the clause is True.

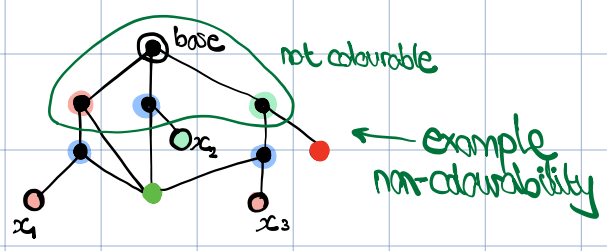


gadget construction

Note: The colour of ⊙ base does not matter: the only thing that matters is the colourability of the constructed gadget.

Clearly, this reduction can be done in polynomial time.

By trial-and-error, we can create similar gadgets for other clause configurations.



Remains to prove F satisfiable $\iff$ G has a 3-colouring. Exercise. $\square$

variables $x_1, \dots, x_n$

U-3-SAT Input: 3-CNF F and some $\mathcal{U} \subseteq \{0,1,2,3\}$.

Output: Does there exist a truth valuation for F such that #(True literals in each clause) $\in \mathcal{U}$ for every clause of F?

Lemma: $\mathcal{U}$-3-SAT $\in$ NP.

Proof: Say s is an instance, and t is some truth valuation.

Clause-wise, evaluate literals using valuation t, and verify that #True $\in \mathcal{U}$.

This verifier is polytime. $\square$

Lemma: $\{0\}$-3-SAT is solvable in polynomial time.

Proof: Start with clause c_1 of F, assuming $F = \bigwedge_{i=1}^{m} c_i$ decomposition.

Enforce all literals in c_1 to evaluate to False and track the necessary truth valuation in a hashmap.

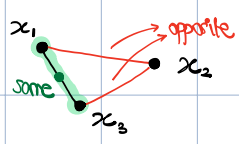
Iterate over all clauses. If there is a collision, then F not $\{0\}$-3-SAT. Else, True. $\square$

Lemma: $\{0,3\}$-3-SAT is solvable in polynomial time.

Proof: Let F be a CNF. Notice for each clause in F, all literals must have the same truth valuation.

Build a signed graph G with $V = \{x_1, \dots, x_n\}$ encoding this constraint:

and complete the graph by growing to include all x_i.

$(x_1 \vee \bar{x}_2 \vee x_3) \longrightarrow$ 

Then G is 2-colourable $\iff$ F $\{0,3\}$-3-SAT, solvable in polytime ($\iff$ G bipartite). $\square$

BFS/DFS works

Theorem: $\{0,1,2\}$-3-SAT is NP-complete.

Proof: We show 3-SAT $\leq_p$ $\{0,1,2\}$-3-SAT. Recall 3-SAT = $\{1,2,3\}$-3-SAT.

Given a 3-SAT instance F with variables $x_1, \dots, x_n$ and clauses $c_1, \dots, c_m$, we want to find a polynomial reduction to a new CNF F' for $\{0,1,2\}$-3-SAT.

For a clause c_j, define $\gamma(c_j)$ = #(True literals in c_j) under some truth valuation for F.

Intuitively if $\gamma(c_j) \in \{1,2,3\}$ for $c_j \in F$, then $\gamma(c_j') = 3 - \gamma(c_j) \in \{0,1,2\}$ where

$c_j' = (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ if $c_j = (x_1 \vee x_2 \vee x_3)$.

Therefore we apply the above transformation for each $c_i \in F$ and form $F' = \bigwedge_{i=1}^{m} c_i'$. Polytime.

F is $\{1,2,3\}$-3-SAT $\iff$ F' is $\{0,1,2\}$-3-SAT.

($\Rightarrow$) Let ν be a valuation for F so $\gamma(c_j) \in \{1,2,3\}$ for $c_j \in F$. Notice ν is well-defined for F'.

Then $\gamma(c_j) \in \{1,2,3\} \Rightarrow \gamma(c_j') = 3 - \gamma(c_j) \in \{0,1,2\}$. $\triangle$ Same argument for ($\Leftarrow$). $\triangle\square$

We consider an extension to the Independent Set (IS) problem. Let G be a graph.

An independent set $I \subseteq V$ is maximally independent if it cannot be enlarged to a larger IS.

Extend-MIS-out Input: G graph, $S \subseteq V$ set.
Output: Is there a maximally independent $I \subseteq V$ with $I \cap S = \emptyset$?

Lemma: E-MIS-out $\in$ NP.

Proof: Let (G, S) be a problem instance, and $I \subseteq V$ be a short proof.

1. Check $I \subseteq V$ is independent. Polytime from IS.
2. Check $S \cap I = \emptyset$. Polytime.
3. For all $v \in V \setminus I$ check that $I \cup \{v\}$ is not independent. $n \times$ polytime.

The above defines a polytime verifier $B_{\text{E-MIS-out}}$. □

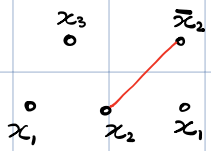
Theorem: Extend-MIS-out is NP-complete.

Proof: We provide a reduction from 3-SAT.

Let F be a CNF for 3-SAT with variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$.

Similar to the reduction 3-SAT → IS, construct consistency edges between every $x_i, \bar{x}_i$ occurrence.

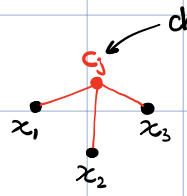
$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2)$ —consistency→



In IS, we asked if G had an IS of size m. → enforce exactly one true literal per clause and have m total clauses

Here, we need a different way to check this size condition.

In particular, each clause needs one True evaluation ⇒



we can add a clause node such that if $x_1, x_2, x_3 \in C_j$ then $x_1C_j, x_2C_j, x_3C_j \in E$.

Add clause nodes to S. The above reduction is clearly polytime.

We claim F satisfiable ⟺ G has a maximal IS outside of S.

(⇒) F satisfiable ⇒ there is some truth valuation such that $C_1, \dots, C_m$ each have at least one True literal. Add all corresponding True nodes to I.

Since each clause C_j has at least one neighbour in I, we have $I \cup \{C_j\}$ dependent for any C_j ⇒ there is a maximal IS for G outside $S = \{C_1, \dots, C_m\}$. △

(⇐) Let I be a maximal IS for G, and $S = \{C_1, \dots, C_m\}$, so $I \cap S = \emptyset$.

SFAC I does not define a satisfying valuation for F. Then $\exists j$ with C_j False so no neighbours of C_j (i.e. the clause literals) are True ⇒ none are in I.

Therefore $I \cup \{C_j\}$ independent, contradicting maximality outside S. ↯ △

Thus 3-SAT $\leq_p$ E-MIS-out, so E-MIS-out is NP-complete. □

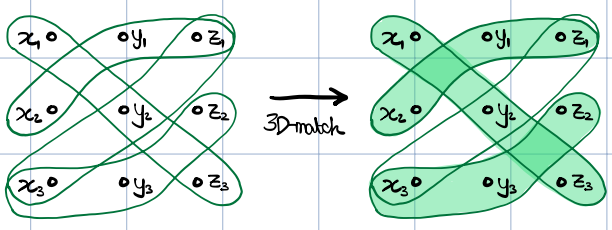
Hard Partitioning Problems

We will show that 3D-matching (→ generalization of bipartite matching) and subset-sum are NP-complete.

3D-matching Input: Disjoint X, Y, Z each of size n, and some set $T \subseteq X \times Y \times Z$ of tiples.

Output: Is there a subset of n tiples in T s.t. each $x \in X \cup Y \cup Z$ is contained in exactly one of the tiples?

Ex. Say $X = \{x_1, \dots, x_n\}$, $Y = \{y_1, \dots, y_n\}$ and $Z = \{z_1, \dots, z_n\}$. Then $X \cup Y \cup Z = \{x_1, \dots, x_n, y_1, \dots, y_n, z_1, \dots, z_n\}$.
Given T, we want $S \subseteq T$ with $|S| = n$ such that every $w \in X \cup Y \cup Z$ appears in exactly one triple in S.



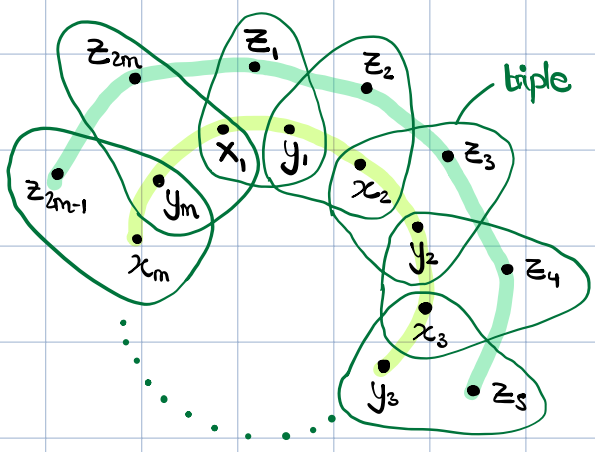
Theorem: 3DM is NP-complete.

Proof: Clearly 3DM $\in$ NP. We will show that 3-SAT $\leq_p$ 3DM.

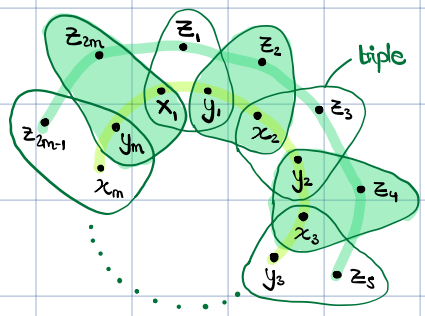
Given 3-SAT CNF F with variables $v_1, \dots, v_n$ and clauses $c_1, \dots, c_m$, we want to build a 3DM instance.

As in the Hamiltonian cycle problem, we would like to build some "gadgets" to capture binary decisions.

Gadget construction. For each variable v_i, we build the following gadget:

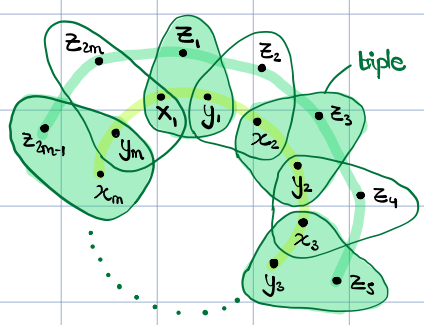


In our construction, we will ensure that each x_i, y_i are only contained in two tiples each, and are not contained in any other tiples ⇒ only 2 possibilities for matchings in the gadget



corresponds to v_i = True

OR



corresponds to v_i = False

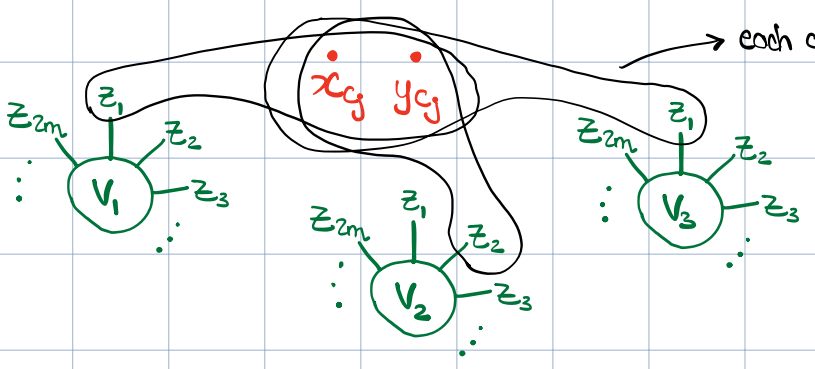
⇒ gadget captures binary decision for v_i.

It remains to add some clause structure.

We use gadget tips z_i to enforce clause structure. Say $c_j = (v_1 \vee \bar{v}_2 \vee v_3)$.

We create new vertices x_{c_j}, y_{c_j} for c_j. We want x_{c_j}, y_{c_j} to be matched ⇔ c_j evaluates to True.

Thus one of gadgets v_1, v_3 must be True or v_2 must be False.



⇒ each clause uses different tips (which is why each variable needs $2n$ tips)

so clause nodes x_{c_j}, y_{c_j} can only be matched if clause c_j evaluates to True.

For uncovered tips, we create dummy vertices such that these tips are matched.

Total $2nm - n = (2n-1)m$ uncovered tips ⇒ add $(2n-1)m \cdot 2nm < 4m^2n^2$ dummy triples.

We can construct this in polytime.

F satisfiable $\Leftrightarrow$ there is a perfect 3D-matching.

($\Rightarrow$) Say F has a satisfying valuation. If $v_i = T$, then v_i gadget covered using odd tips, else even.

All c_j satisfied $\Rightarrow$ each c_j has a literal satisfied $\Rightarrow$ there will be a tip available for v_i to cover x_{c_j}, y_{c_j}.

Other uncovered tips use dummy tips. $\triangle$

($\Leftarrow$) Perfect 3DM $\Rightarrow$ each clause pair x_{c_j}, y_{c_j} covered $\Rightarrow$ reverse-construct truth valuation. $\triangle$

Therefore 3-SAT $\leq_p$ 3DM so 3DM NP-complete. $\square$

Subset-sum Input: $a_1, \dots, a_n \in \mathbb{N}$ and $k \in \mathbb{Z}$.

Output: Is there a subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = k$?

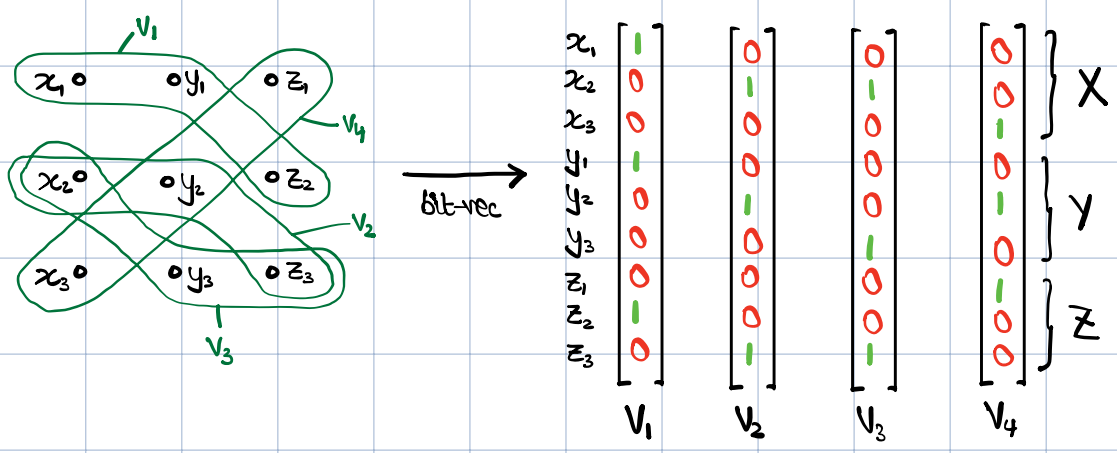
This is a problem about numbers, so some new ideas are needed to connect to previous (combinatorial) problems.

Theorem: Subset-sum is NP-complete.

Proof: Clearly subset-sum $\in$ NP. We show 3DM $\leq_p$ subset-sum.

Given an instance (X, Y, Z, T) of 3DM, we construct an instance of 3DM.

The idea is quite natural: $t \in T \mapsto$ bit vector $\mapsto$ number. We show an example first.



We essentially create masks for each $t_i \in T$ that identify the X, Y, Z elements in t_i.

Clearly, the construction can be done in polytime.

Denote $V = \{v_1, \dots, v_{|T|}\} \subseteq \mathbb{R}^{3n}$ to be our set of constructed bit-vectors.

Clearly, we see that if $\exists S \subseteq V$ such that $\sum_{v \in S} v = \mathbb{1} \in \mathbb{R}^{3n}$ then each row-sum is 1,

so exactly one of each x_i, y_i, z_i was included $\Rightarrow$ there is a perfect 3DM.

$\Rightarrow \exists S \subseteq V$ with $\sum_{v \in S} v = \mathbb{1} \in \mathbb{R}^{3n} \Leftrightarrow (X, Y, Z, T)$ has a perfect 3D-matching.

We would like to reduce this subproblem to subset-sum.

won't quite work yet! Idea: Think of each $v_i \in V$ as the binary representation of some number,

so $(x_i, y_j, z_k) \mapsto 2^i + 2^{n+j} + 2^{2n+k}$ so $\exists$ matching $\overset{?}{\Rightarrow}$ some subset sums to $\sum_{i=1}^{3n} 2^i$.

Problem: Above won't work because bits may be "carried" during addition: $\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$ but $2^1 + 2^1 = 2^2 \mapsto \begin{bmatrix}0\\1\end{bmatrix}$

We work around this by working in a large enough base b such that carrying can never happen.

Say there are at most $n = |T|$ vectors constructed. Choose base $b = n+1$ so a carry never occurs.

Thus we define our reduction $\varphi: T \longrightarrow \mathbb{Z}$, $(x_i, y_j, z_k) \mapsto b^i + b^{n+j} + b^{2n+k}$ and define $k = \sum_{i=1}^{3n} b^i$ the target.

1 $\varphi(T)$ is polytime.

Recall we define time complexity in terms of input size in bits, not as $n = |X| = |Y| = |Z|$.

Thus although m may be exponential in n, the input size is $\Omega(n)$ since T is part of the input.

For $t \in T$, clearly $\varphi(t)$ is polytime, so $\varphi(T)$ polytime.

2 (X, Y, Z, T) has a perfect 3D-matching $\Leftrightarrow$ there is $S \subseteq \varphi(T)$ with $\sum_{s \in S} s = k$.

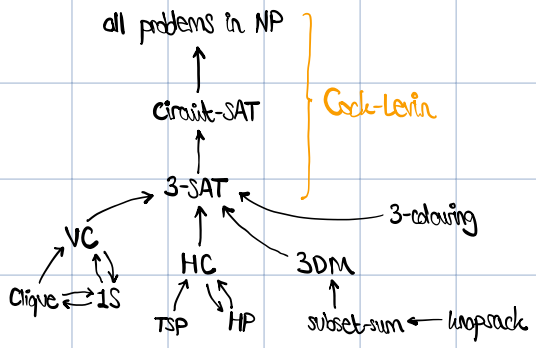
($\Rightarrow$) Proven above. $\triangle$

($\Leftarrow$) Each position ℓ in the base-b representation records the number of times we cover ℓ-th vertex in the 3DM instance. All positions 1 base-b $\Rightarrow$ perfect 3DM. $\triangle$

Therefore 3DM $\leq_p$ subset-sum, so subset-sum is NP-complete. $\square$

Corollary: Knapsack is NP-complete.

So far, we have built the following relations:



It is an interesting phenomenon to observe that 2 is usually easy but 3 is hard.

2-SAT is easy, but 3-SAT is hard. 2-colouring is easy but 3-colouring is hard. 2DM easy, 3DM hard.

Usually "2" is easy since once a decision is made, the problem space becomes much smaller (forced). However, with "3", we are still left with a binary decision.

Decision v.s. Search Problems

Does restricting our scope to decision problems limit our characterization of NP-completeness?

Many search problems can be easily reduced to decision problems. We briefly formalize this.

A problem X is said to be self-reducible if $X_{search} \xrightarrow{F} X_{decision}$ for polytime reduction F.

↳ polytime in the runtime of the solver for $X_{decision}$

Theorem: 3-SAT is self-reducible.

Proof: Let F be a 3-CNF with variables $x_1, \dots, x_n$ and clauses $c_1, \dots, c_m$. Say φ is a solver for 3-SAT$_{decision}$.

1) Set $x_1 = 0$ and obtain new formula F_1. Check $\varphi(F_1)$.
   If $\varphi(F_1)$ True, recursively find assignments for $x_2, \dots, x_n$ that satisfy F_1.

2) Else, F_1 was not satisfiable, so try with $x_1 = 0$ to obtain F_2. Check $\varphi(F_2)$.
   If $\varphi(F_2)$ True, recursively find assignments for $x_2, \dots, x_n$ that satisfy F_2.

3) F_1 and F_2 not satisfiable $\Rightarrow$ no assignment for x_1 satisfies $F \Rightarrow F$ not satisfiable.

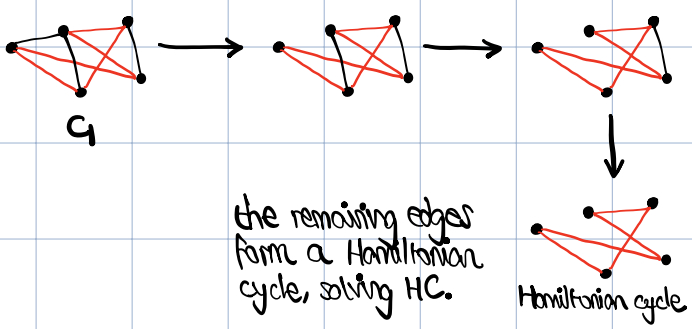
The above algorithm makes at most $3n$ (polynomially many) calls to $\varphi \overset{\text{def}}{\Rightarrow}$ 3-SAT self-reducible. $\square$

Theorem: HC is self-reducible.

Proof: Let G be an instance for HC, and $\mathcal{V}$ be a solver for $HC_{decision}$. Consider the following algorithm:

---

for $e \in E$:
- define $G' = G - e$ (remove the edge)
- check if $\mathcal{V}(G')$ is True
- $\mathcal{V}(G')$ True $\Rightarrow$ update $G = G'$ and continue
- $\mathcal{V}(G')$ False $\Rightarrow$ skip and continue (since e must be in all HCs of G)
- stop when G has $\leq |V|$ edges remaining



The algorithm calls $\mathcal{V}$ at most $\Theta(|E|)$ times, and $|E| \in poly(|V|) \Rightarrow HC \xrightarrow{P} HC_{decision}$. □

Theorem (Self-Reducibility): Let A be an arbitrary decision problem.

1) $A \in NP \Rightarrow A \xrightarrow{P} A_{search}$. 2) A is NP-complete $\Rightarrow A_{search} \xrightarrow{P} A$.

Proof (1): Trivial. To solve A, run A_{search} solver once to obtain a short proof for A, then verify in polytime. △

(2): Hard, beyond scope. △□

Therefore, for NP-complete problems, search and decision are identical in terms of solvability complexity.