INTRODUCTION TO 3rd

ALGORITHMS

Cormen
Leiserson
Rivest
Stein

ch1 – ch35
53 pages
2015,4,26 – 2016,11,9

• Insertion Sort: like sorting a hand of cards ⇝ insert everything in its proper place

incremental approach

"initialization", "maintenance", & "termination" must be elaborated for the "loop invariant".

We analyze worst & average running times for a given algorithm.

we usually want the order of growth, not the actual complexity ←

Divide progressively dividing the problem into smaller bits
Conquer and recursively solving it.

```
mergeSort (A, i, j) {
    A1 = mergeSort(A, i, ⌊(i+j)/2⌋)
    A2 = mergeSort(A, ⌈(i+j)/2⌉, j)
    return merge (A1, A2)
}
```

* To find $(x,y) \in S$ so that $x+y=z$, we first sort S ($O(n \lg n)$) and then move from both ends ($O(n)$) $\rightarrow O(n \lg n)$

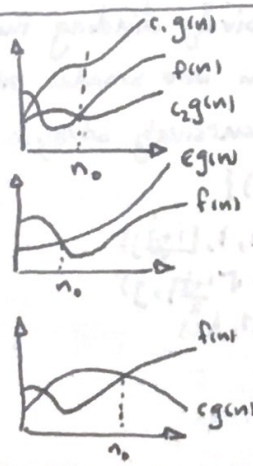
* SELECTION SORT : find the smallest one by one $\rightarrow O(n^2)$

→ for small "n", merge sort is shower than insertion sort.

* BUBBLE SORT : large values sink down, smaller values bubble up ← $O(n^2)$

• When we say array S has inversion (i,j) it means $i<j$ and $A_i > A_j$

ch 3



$f(n) \in \Theta(g(n))$

$f(n) \in O(g(n))$
↳ O-notation is less tight

$f(n) \in \Omega(g(n))$
↳ Ω-notation is less tight

$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right)$ ⇝ Stirling's approximation

$\Rightarrow \lg(n!) = \Theta(n \lg(n))$

$\Rightarrow n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} : \frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$

$f^{(i)}(n) = \begin{cases} n & i=0 \\ f(f^{(i-1)}(n)) \end{cases}$ ⇝ $\lg^*(n) = \min\{i \geq 0 : \lg^{(i)}(n) \leq 1\}$

$F_i = F_{i-1} + F_{i-2}$ ⇝ $F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor$ $\frac{1+\sqrt{5}}{2}$

↳ i^{th} Fibonacci number

• $f_c^*(n)$: number of iterations of "f" to make $n \leq c$

ch4

* Finding a max subarray :
1. Find in left half ($T(\frac{n}{2})$)
2. Find in right half ($T(\frac{n}{2})$)
3. Find containing midpoint ($\Theta(n)$)

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

* Matrix multiplication
- precompute addition matrices
- math : $\Theta(n^3)$
- break in quarters $\Theta(n^3)$
- Strassen : $\Theta(n^{\lg 7}) = \Theta(n^{2.8})$

1 One way to solve recursions : substitue $T(n)$ with our guess, and use induction

2 Another is to use recursion trees.

3 the master theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

→ subproblem sizes are equal

3.1 • $f(n) = O\left(n^{\log_b a - \varepsilon}\right) \rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$ ①

3.2 • $f(n) = \Theta\left(n^{\log_b a}\right) \rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$ ②

3.3 • $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ & $c<1 : a f\left(\frac{n}{b}\right) \leq c f(n) \rightarrow T(n) = \Theta(f(n))$ ③

☆ mind the gaps when using the master theorem

* We could also use generating functions to solve recursion

* The Akra-Bazzi method : $T(x) = \begin{cases} \Theta(1) & 1 \leq x < x_0 \\ \sum_1^k a_i T(b_i x) + f(x) & \end{cases}$

~ find $p \in \mathbb{R} : \sum_1^k a_i b_i^p = 1$

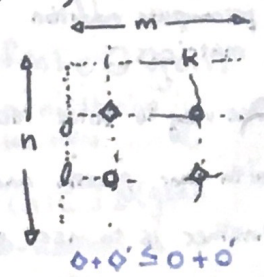
$\Rightarrow T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)\,du}{u^{p+1}}\right)\right)$

monge arrays

$\forall\, 1 \leq i < k < m \wedge 1 \leq j < l < n$

$: A[i,j] + A[k,l]$

$\leq A[i,l] + A[k,j]$



! a square monge matrix that's symmetric about its main diagonal is a **Supnick**. if distance matrix is Supnick, travelling salesman has an easy solution.

---

ch5

• For event A, indicator variable $I\{A\}$ is defined as: $I\{A\} = \begin{cases} 0 & \text{if A doesn't happen} \\ 1 & \text{if A happens} \end{cases}$

• ~~Expectation~~ Expection for value of A is defined as:

$$E[X_A] = E[I\{A\}] = Pr\{A\} \times I\{A\} + Pr\{\bar{A}\} \times I\{\bar{A}\}$$

• also: $E[\sum X] = \sum E[X]$

⇒ to perform randomized analysis on our algorithms we must know/assume something about their input.

* unified random permutation: generated by a function that generates every permutation with equal likelihood.

Comparison of Sorting Algorithms

Algorithm	worst	expected average
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
stable Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap Sort	$O(n \lg n)$	—
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$
stable Counting Sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix Sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$

(binary) heap data structure has a "heap property"
- → min-heap: $\forall n: n.value \geq n.parent.value$
- → max-heap: $\forall n: n.value \leq n.parent.value$

if implemented as an array ⇨
- $parent(n) = \lfloor \frac{n}{2} \rfloor$
- $left(n) = 2n$
- $right(n) = 2n+1$

MAX-HEAPIFY sinks down a violating node $O(\lg n)$

$O(n)$ BUILD-MAX-HEAP heapif from the bottom

HEAP (MAX)

$O(\lg n)$ HEAP-INSERT insert $-\infty$ and increase it

$O(\lg n)$ EXTRACT-MAX swap first & last, heapify

$O(\lg n)$ INCREASE-KEY float new value upward

$\Theta(1)$ HEAP-MAX return top element

→ a heap is used to do heap sort, which is in place, but is not stable

→ build max heap, extract max until heap is empty, the data is sorted

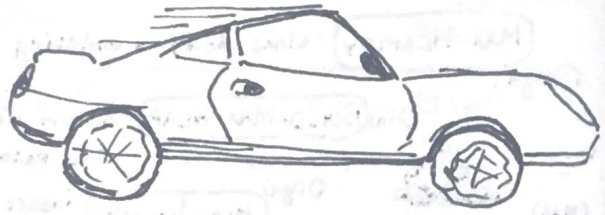
Young Tableau

Each row and each column is sorted.

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 4 & 6 & 12 & 19 \\ 7 & 8 & 13 & 22 \\ 14 & \textcircled{80} & \infty & \infty \end{bmatrix}$$

an empty cell

ch7



Quick Sort

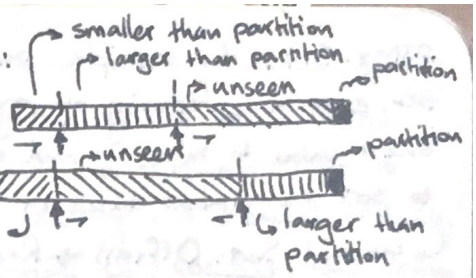
↳ in practice the fastest sorting algorithm

- ▷ Sort one item in place
- ▷ Call that the partition
- ▷ Sort the halves recursively

} Could choose a bad partition ⇒ $O(n^2)$

↝ if we randomly choose an element as the partition we are increasing the chances of an average case happening

Partitioning:



↳ if we find partition at $\lfloor \frac{n}{2} \rfloor$ by linear selection the worst case becomes $O(n \lg n)$

ch8

Comparison sort algorithms make no other assumption about their input than their being comparable.

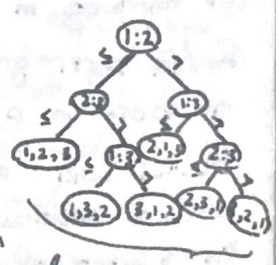
- in the decision three, lower bound for execution is a lower bound for height.

$n! \leq l \leq 2^h \rightarrow \lg(n!) \leq \lg l \leq h$

(h: # of permutations; l: max # of leaves)

$\rightarrow h \geq \lg n! \geq n \lg n$

$\rightarrow h = \Omega(n \lg n)$ → lower bound for any comparison sort



COUNTING-SORT: finds out how many times each array item in the input occurs & then constructs the answer in $O(k)$ where k is $\max\{A\} - \min\{A\}$. If $k < n \Rightarrow$ it is $O(n)$

RADIX-SORT: do multiple passes on the number set, each time focusing on one digit, going from lowest value to highest, and use a stable sort to sort the numbers based on these digits.

↳ Internal Sort: $O(f(n))$ → Radix sort: $O(df(n))$ (values of each digit)

Internally using counting sort:

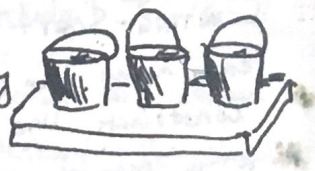
Radix sort: $O(d(n+k))$

Counting sort: great for sorting large # of numbers in a limited range

Radix sort: great for sorting small # of numbers in a wide range.

BUCKET-SORT: assume the numbers have a uniform distribution in a given range & create "n" buckets of equal sizes. Distribute numbers into their buckets, sort each bucket via insertion sort and concatenate everything.

↳ Since the numbers are distributed uniformly it is $O(n)$



ch 9

Selection: given a set of n distinct numbers, find ith element that is larger than exactly i-1 element in the set.

- Basic solution: sort ($O(n\lg n)$) and return the ith element ($O(1)$) → $O(n\lg n)$
- Randomized selection: use randomized partitioning to figure out only the parts of the set we are interested in: $O(n)$

↳ $\Theta(n^2)$: worst case — expected time

Baron: $O(n)$ ← → 3: $\Omega(n\lg n)$

- Linear: break into groups of five, sort each group, collect their medians, & recursively find the median of those found medians; then partition around it and look at one half of the remaining array until we find our item.

this ensures we choose a good pivot — See 9.3

↳ We examine each position only once, and sort arrays of constant size 5, thus it is $O(n)$. ← worst

(!) ↳ if array size is less than 6, just sort that and return the item

ch 10

Set usage policies:

- First-in, Fist-out: Queue / Enqueue / Dequeue
- First-in, Last-out: Stack / Push / Pop

THE STACK

Linked lists:

- Singly linked →
- Doubly linked →

Insert: $O(n)$ / Delete: $O(1)$ / Search: $O(n)$ → if searching: $\Theta(n)$

ch 11

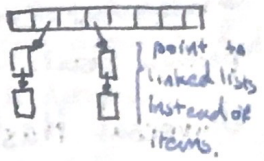
Hash tables are dictionary data structures that can handle delete, insert, and retrieval in expected $O(1)$.

* Direct addressing is a glorified name for a pre-allocated array of fixed size wherein array indices correspond to keys. → only useful if U (universe of possible keys) is small.

If we instead use hash function $h(k)$ to determine the real, stored key for any given element, we can reduce the storage required to $O(k)$.

Since $|U| > |k|$, we will have multiple keys hashing to the same slot ← collision

One way to handle collisions is by chaining



$\frac{n}{m} = \alpha$: load factor

n ← items stored

m ← available slots

α correlates directly with the probability of a collision.

Search performance is $\Theta(1+\alpha)$ for both misses & hits, if we have "simple uniform hashing"

↳ all keys are equally likely to fall in any of the m slots.

A good hash function follows simple uniform hashing. Possibly other characteristics as well.

Knowing something about the keys could help us.

◆ The division method: $h(k) = k \bmod m$ (e.g. 701)
↳ fast, depends on m
- not a power of 2
- a prime is good

◆ The multiplication method:
$h(k) = \lfloor m(ka \bmod 1) \rfloor$ → $0 < a < 1$; Knuth: $\varphi = \frac{\sqrt{5}-1}{2}$
↳ independent of m. If m is a power of 2, can perform faster.

Universal Hashing: choosing from a set of hash functions with a uniform random distribution to remove the need for data being uniformly distributed

* given set $\mathcal{H}$ of functions, it is universal if chances of $h(k) = h(l)$ for $k \neq l$ for any function is at most $|\mathcal{H}|/m$ → collision chance is $\frac{1}{m}$ for a random choosing. → Restore simple uniform hashing assumption regardless of original keys

For prime "p": $\mathcal{H}_{pm} = \{h_{ab} : 1 \le a < p, 0 \le b < p\}$
& $h_{ab}(k) = ((ak+b) \bmod p) \bmod m$

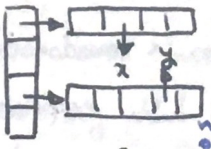
this set is always guaranteed to be universal

When we do not allow for handling of collisions (i.e. we let the hash table become full) & serve empty spaces to entries on a first-come, first-served basis, we are implementing «open addressing».

→ When a slot is full, we probe to find the next empty space in a deterministic fashion:

$h(k, i)$ ← key being hashed; ← attempt number
↳ unsuccessful: $O(\frac{1}{1-\alpha})$
↳ successful: $O(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$

▷ linear: $h(k,i) = (h'(k) + i) \bmod m$
▷ quadratic: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
▷ double hashing: $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

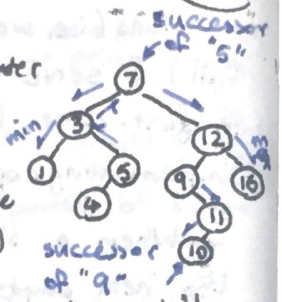


→ using two levels of hash tables with universal hashing → Perfect Hashing
no insert or delete, just insert

* Search has a worst-case perf of $O(1)$
* Space complexity: $O(n)$ ← number of stored items
▷ Keys have to remain static throughout application → your search recomputes everything in case of a change

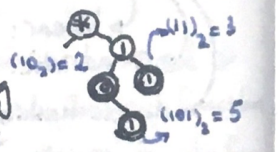
ch12

A binary tree organized so that every node is greater than all nodes on its left subtree and smaller than all nodes on its right subtree is a "binary search tree" (BST)



WALKING A TREE
- pre order: parent before children
- post order: parent after children
- in order: parent between children

Radix tree: go left if bit is zero otherwise go right.



→ helps with lexicographical sorting of numbers based on their bits in $\Theta(n)$ ⇝ pre order walk

→ Creating by insertion from a randomized permutation creates BST with expected height of $O(\lg n)$

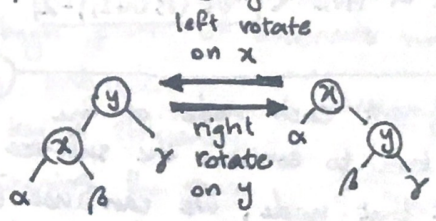
ch13

Red-Black trees ensure that balance is kept when restructuring the tree using rotations

- nodes are either red or black
- root is black
- leaves are black
- children of red nodes are black
- all paths from a node to its leaves have same number of black nodes

↳ Given "n" internal nodes has height of at most $2\lg(n+1)$

↳ Dynamic set operations can perform in $O(\lg n)$ (Search, Minimum, Maximum, Successor, & Predecessor)



After both insertion and deletion we use recoloring and rotation to reinstate the red-black properties of the tree.

↳ In case of deletion we only have an issue if the deleted node was black.

AVL trees → auto-balanced on each insert & delete
better retrieval than red-black but slower insert & update

↳ we define balance factor on each node as difference between left child height & right child height → [-1, 1] is okay.
We have to fix by walking up from newly modified node (inserted or deleted) & do rotations wherever ~~pos~~ required

↳ Height of AVL < $\log_\varphi(\sqrt{5}(n+2))-2$

---

ch14

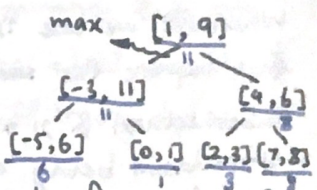
If we augment each node of the red-black tree to contain the sub-tree size rooted at that node, we can use this additional information to find the order statistics of an item in O(lgn) time.

Augmenting Data Structures
① Choose basic data structure
② Determine additional data
③ Modify existing operations to maintain this additional data
④ Develop new operations

⚠ the most important step

* If the new information f on every node of a RED-BLACK TREE only depends on that node & its children, we can maintain it during the insert & delete operations without affecting the original performance of these operations.

- We can also augment RED-BLACK TREES to implement interval trees that are used for finding which intervals overlap with a given interval. The key is the low point of each interval, and we also store the max of high points of intervals hosted on each subtree.
- the key is efficiently maintaining the max



Josephus Permutation ⟩⟩ given a circle of n people & constant m, we remove every m^{th} person until no one remains

$$g(n,m)=\begin{cases}(g(n-1,k)+k) \bmod n & n>1\\ 0 & n=1\end{cases}$$

↳ O(n) with dynamic programming

ch 15

▶ Dynamic programming is the practice of memoizing solutions to subproblems to avoid solving them again.

* When solving a problem, drawing a graph of how a sub-problem is referenced is a good idea & can help us figure out how to reconstruct the solution

↝ We can usually rewrite recursive memoization in the form of a bottom-up solution that starts with the smallest problem & fills up the memory. ← 15.1 rod cutting problem

Whenever we see 1) a set of subproblems that require the same optimization (optimal substructure) & 2) a high frequency in repetitive subproblems being referenced we should use dynamic programming. ← 15.2 matrix chain multiplication

* If after calculating the optimum results we require to be able to construct the steps that got us there, we usually will need to use more memory.

Dynamic Programing → Trading off memory for speed

Notable problems:
- Matrix chain ordering
- Longest common sub-sequence
- The rod-cutting problem
- Optimal binary search trees.
- Longest ascending subsequence

> whenever our solution depends on "k" data points in its substructure we need a memory of size $O(n^k)$

ch 16

Problems that can be solved by choosing the locally best available option and elicit one subproblem have a greedy solution.

If the solution depends on more than one subproblem it cannot be solved greedily & we might be better off trying dynamic programming.

If the overlap frequency of subproblems is low, we should avoid dynamic programming since it will only eat memory.

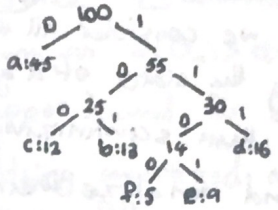
⇒ Activity selection: greedy solution performs better

To solve the knapsack problem (1-0 version) we solve the problem for knapsacks of sizes 1..w over the first "i" items, always considering if choosing item "i" & filling the rest of the knapsack with items 1..i-1 is better or not taking item i, by looking at how well we did with the first i-1 items.

We can use linear order statistics selection using value per pound of each item as the key to fill up the knapsack, if we are allowed to fill it by picking fractions of each item.

↳ We need to solve the problem of finding weighted medians.

One way to compress character data is to change their encoded value to take a variable length rather than a fixed length.



Huffman

coding is a way of getting a variable, prefix code for each character based on their frequency

the **prefix** property ensures that we can identify the first char. of any encoded sequence as a unique character.

a	b	c	d	e	f	
45	13	12	16	9	5	freq.
0	101	100	111	1101	1100	code

▶ We use min. priority queue and use the frequency as the key, and merge the first two under a new root with a key of the sum of its two children, while putting that root back into the queue, until we have only one item left.

▶ The tree is as close to full as possible and the keys are very well designed.

→ Every time we take a turn we take it as a "0" for left and "1" for right.

↳ Gives us 25% to 90% compression

ch17

In amortized analysis, we consider all the operations that modify the state of a given data structure rather than scrutinizing them independently and analyze their time/memory complexity in terms of the whole operations.

① → aggregate: consider complexity of "n" possible operations of all kinds and divide that by "n", assigning it to the average cost of every operation → all operations will end up with the same value.

② → accounting: in accounting method we associate each operation with a cost & the total of this cost has to be an upper bound for the actual cost & has to result in always positive "credit".

③ → potential: given operation i that transforms data structure D from state D_{i-1} to D_i we define: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

($\hat{c}_i$ ← amortized cost; c_i ← actual cost; Φ → potential function)

Since $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$ as long as $\Phi(D_n) > \Phi(D_0)$ we can find an upper bound for D.

Using amortized analysis we can show that after implementing expansion and contraption for data table, the running time of n consequtive operations on the table remains $O(n)$.

We use amortized analysis to have a better understanding of a data structure or algorithm's performance in practical situations.

ch18

B-Trees are balanced search trees that are designed to work well on disks.

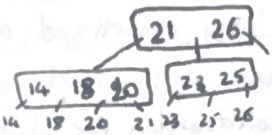
Instead of being binary search trees they have a degree "t" and each node stores a minimum of t-1 & a maximum of 2t-1 keys: $t-1 \le x.n \le 2t-1$

no one really knows why the B ← B-Tree & the authors never explained!!

Each node with $x.n$ keys has $x.n+1$ pointers to internal nodes.

For such a b-tree we have: $h \leq \log_t \frac{n+1}{2}$

Sample B-Tree with $t = 3$.



Tree height only increases when the root becomes "full".

→ A B^+-Tree has all leaves linked continously and doesnot associate data with internal nodes.

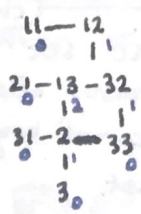
→ A B^*-Tree has the property of: $\frac{2}{3}t \leq x.n$

↝ When implementing data structures that work directly with secondary storage devices the main goal becomes minimizing storage access.

---

ch19

Fibonacci heaps are heaps that achieve a really good <u>amortized</u> time for most of the mergeable heap operations.

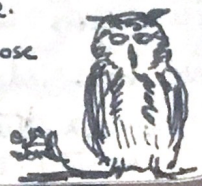


The Fibonacci heap relies on a tree-like data str-ucture wherein all sibling nodes are in a doubly linked circular list.

PROCEDURE	BINARY-HEAP	FIBONACCI
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$
	worst-case	amortized

↳ We put off rearranging until when we are popping a key or decreasing a key.

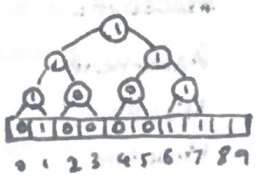
→ Necessary for priority queue.
↳ won't work if we don't expose pointers to nodes



ch20

One way to implement dynamic set operations is to have a bit array of size n to signify the presence of any number within [0, n) in the set.

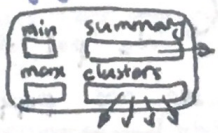
We can then super impose this with with a binary tree to help with successor & other related operations



Based on this same idea we can create m levels u-ary trees over an array, but we then will need an auxiliary structure to tell us about the distribution of the keys. fixed size of m → O(lg lg m) performance

DBC applied to data structures

a given van Emde Boas tree node of size u has



a summary tree of size $\uparrow\sqrt{u}$ and $\downarrow\sqrt{u}$ clusters where $\uparrow\sqrt{u} \cdot \downarrow\sqrt{u} = u$ & $\downarrow\sqrt{u}, \uparrow\sqrt{u} \in \mathbb{N}$

We break index x into two parts:

$x = (\lfloor x / \downarrow\sqrt{u} \rfloor) \times \downarrow\sqrt{u} + (x \bmod \downarrow\sqrt{u})$

segment: higher bits — offset: lower bits

ch21

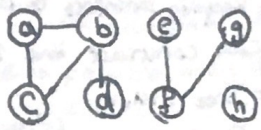
When we need to partition items based on a common trait we are facing the problem of maintaining disjoint sets.

DS (disjoint sets)
- MAKE-SET
- UNION
- FIND-SET

} it is expected that a set's representative can be chosen randomly

it only has to be deterministic unless we change something.

⇨ finding connected components is one application.

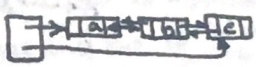


{a} = {a, b, c, d}, {e} = {e, f, g}

{h} = {h}

makeset, find, union

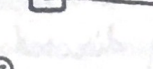
- Linked list:



O(1) O(1) O(n) worst-case

O(1) O(1) O(1)

- Forest:



→ merge by concatenating lists → weighted lists do better

→ merge by joining roots → ranked trees do better.

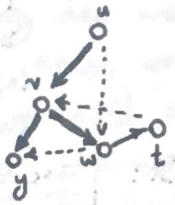
→ path compression: flattening child list in find path during FIND-SET

ch22

There are various ways for representing a graph → adjacency matrices → better when dense
→ adjacency list → better when sparse

▶ Breadth-First search is a form of visiting all nodes from given source vertex s so that nodes with distance k are seen before any node with distance k+1 or more from the source.
↳ by keeping pointers to parent nodes during a search we can construct the shortest path tree for BFS.

▶ Edge Types
- Tree : (u,v)
- Back : (t,v)
- Forward : (u,v)
- Cross (w,y)

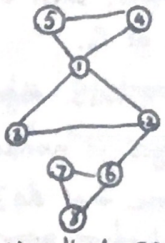


If a graph does not have back edges it is then acyclic. using DFS ↩

▶ We can sort any directed graph topologically.
↳ at each step we must be able to pick one vertex without any input edges.
⇒ In a strongly connected component there is a path between any two vertices.
↳ Do DFS on G → every existing edge is reversed
Do DFS on G^T → reverse order of finish time
Forest forms strongly connected components

→ articulation point: vertex whose removal will disconnect the graph.
→ bridge: edge whose removal disconnects the graph.
→ biconnected component: maximal set of edges that forms a simple loop.

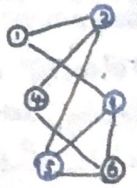
(5,4,1,2,3) ⇝ a biconnected component
(1) → an articulation point
(2,6) → a bridge



⇝ An **Eulertour** is a visitation that passes each edge exactly once → possible only if for all $v \in G.V$: $in(v) = out(v)$

ch23

▶ **Cut** $(S, V-S)$ for $G(E,V)$ is a partition for G.
▶ Edge (u,v) **crosses** the cut if one of u,v is in S and the other is not.
▶ Cut S **respects** set $A \subseteq E$ if no edge in A crosses it.
▶ Given $A \subseteq E$ we call edge $e \in A$ (u,v) **light** if $w(e) = \min\{W_A\}$ for $W_A = \{w(u) \mid u \in A\}$



- cut: 2,3,5
- (2,1) & (1,3) cross it.
- cut respects {(2,5),(5,3)}

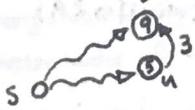
A generic algorithm for finding a minimum spanning tree (MST) for $G(E,V)$ works by maintaining a set $A \subseteq E$ and iteratively finding an edge (u,v) that crosses cut C which A spans, such that (u,v) is light, until C contains all of G.

* Kruskal's algorithm works by considering edges in nondecreasing order of weight & picking those that do not belong to a vertex previously picked. ⇒ grows a forest and joins the trees
* Prim's algorithm starts by traversing from a given node & always marking the best parent for a node, until it is done.

ch. 24

To solve shortest path problem with distination u, we transpose G and then solve same problem with source u in G^T.

relaxing an edge means checking to see if taking detour via that edge improves the length of the path known so far for reaching the endpoint vertex.



Considering that in a graph, the maximum number of edges in a simple path is $|V|$, the Bellman-Ford algorithm relaxes every edge $V-1$ times. If after this, any potential improvement is found, it means we have a negative loop ⇝ No solution $O(VE)$

select the closest unselected u and relax it's neighbors

We could also first sort all nodes topologically and go through them, relaxing all outgoing edges ⇝ DAG handler $\Theta(V+E)$

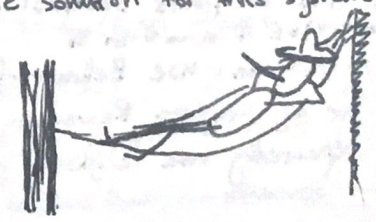
Dijkstra's algorithm works like the previous one except instead of topologically sorting it maintains a priority queue over the distances $O(V \lg V + E)$ (with Fibonacci heaps)

$$\begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \leq \begin{bmatrix} 4 \\ -2 \\ 3 \end{bmatrix} \rightsquigarrow$$

$$\rightsquigarrow \begin{cases} v_1 - v_3 \leq 4 \\ v_2 - v_3 \leq -2 \\ v_3 - v_1 \leq 3 \end{cases}$$

By executing Bellman-Ford on the graph representing the system of differences we can find a feasible solution for this system.

Just Relax!



We can find the shortest path between any two vertices in various ways:

- Finding single-source shortest path from all vertices $\rightsquigarrow O(VE \lg V)$
- Finding matrix $L^{(n-1)}$ where L holds the shortest path, by equation $l_{ij} = \min_k \{l_{ik} + w_{kj}\}$ $O(V^4)$
- Doing the above, only exploiting the fact that $W^{2m} = W^m \times W^m$ for square matrices $O(V^3 \lg V)$ → works better with adjacency matrix
- FLOYD-WARSHALL uses midpoint k to go from i, to j if it makes the path better $O(V^3)$

→ A transitive closure of graph

$G(E,V)$ is a subset E' where all u & v who have $(u,v) \in E'$ or $(v,u) \in E'$ are reachable from each other → give edges weight 1 & find all pairs shortest path $\{\infty$: not connected, else: connected$\}$

$G(u,v) \in E'$ if $u \rightsquigarrow v$ in G

- Johnson: use Bellman-Ford to get an offset for all weights. Reweight all edges and then repeatedly use Dijkstra $\rightsquigarrow O(V^2 \lg V + VE)$

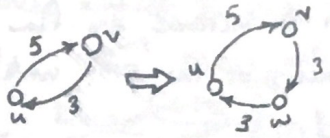
A flow network is a directed graph wherein all edges have a nonnegative capacity $c(u,v)$. $f(u,v)$ is then the used capacity of the edge.

The value of the flow is:

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

what's flowing out of "s" — what's flowing into "s" (usually zero)

We can remove anti-parallel edges very easily



Given flow network G, we construct "residual network" G_f where we have:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(v,u) & (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Given $G(E,V)$, G_f is: $G_f(E_f, V)$ where

$E_f = \{(v,u) \in V \times V : c_f(u,v) > 0\}$

Flow augmentation:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

↓ residual flow

A path from s to t in G_f is called an augmenting path.

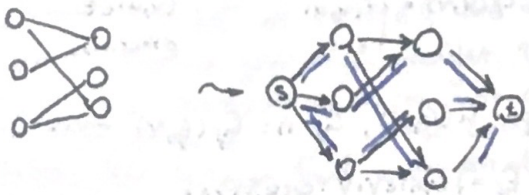
* Residual capacity of path p is:

$$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$$

↳ This is the max amount by which we can increase the flow on each edge in path p.

In FORD-FULKERSON method we construct G_f from G continually and find an augmenting path & increase the flow by the residual capacity of the path until no path can be found.

↳ Called EDMONDS-KARP when we use BFS to find augmenting path "p". → $O(V.E^2)$



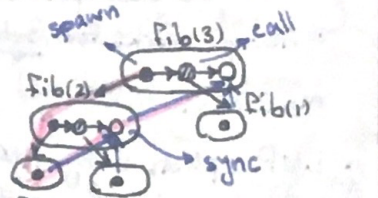
■ modelling bipartite matching using a maximum flow graph.

another method for solving the network flow problem is a class of algorithms called the push-relabel method.

ch27

if we have a parallelization framework that lets us "spawn" pieces of code and "sync" on their execution, we have multi-threaded algorithms: fib(n) = spawn(fib(n-1)) + fib(n-2)

- after removing these keywords the sync code should function normally



↳ DAG of parallel execution

- strand a is reachable by strand b: they are logically in series
- otherwise: they're logically in parallel

performance
- work: number of vertices (see 24.2)
- span: length of critical path

→ when 2 or more threads access a shared memory and at least one writes we have a "race condition"

ch28

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \equiv \pi = [3,2,4,1]$$

↳ Permutation matrix — Permutation array

$\Theta(n^2)$ with LUP-solve

$A \cdot x = b \rightarrow PA \cdot x = Pb \rightarrow LU \cdot x = Pb$

$\rightarrow \begin{cases} Ly = Pb \\ Ux = y \end{cases}$ $\quad \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$ $\quad y \; \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$

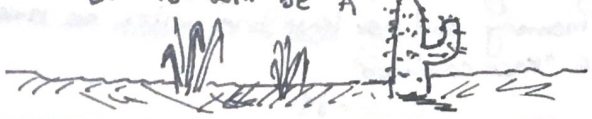
↳ The process of creating such L & U is called LU-decomposition

↳ Always works for symmetric definite positive matrices.

- LUP pivots using permutation π to be able to solve for all solvable systems.

* Calculating the inverse of a matrix has the same asymptotic time as multiplication
⋯→ we can solve $(A^TA) \cdot x = (A^Tb)$ for all A

For X_i & I_i which are column matrices we can solve "$AX_i = I_i$" n times for $A_{n \times n}$.
If I_i is the i-th column of identity $I_{n \times n}$ then $X = [X_1 \dots X_n]$ will be A^{-1}.



Elementary operations: { swith: $R_i \leftrightarrow R_j$; add: $R_i + kR_j \rightarrow R_i$; multiply: $kR_i \rightarrow R_i$ }

if we perform one of these on $I_{n \times n}$ we get an elemantary matrix.

$M = M_1 \times M_2 \times \dots M_n$ is invertible if M_1 to M_n are invertible.

▸ if $c = a + \hat{i}b \rightarrow \bar{c} = a - \hat{i}b$ is the conjugate

▸ if M is a matrix, M^* is M transposed with every element conjugated and is called conjugate transpose of M.

M is positive-definite if z^*Mz is real and positive for every non-zero vector z.

ch29

- $f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$, $a_i \in \mathbb{R}$ is a linear function
- $f(x_1, \dots, x_n) = b$, $f(x_1, \dots, x_n) \leq b$, $f(x_1, \dots, x_n) \geq b$ are linear constraints.
- $V = (x_1, \dots, x_n)$ is a feasible solution if it satisfies all the constraints $f_1 \dots f_n$.
- we call the convex region formed by the constraints a **simplex**.

We select a nonbasic variable from the objective function that has a positive coefficient & increase it as much as possible without violating any of the constraints.

Some other variable becomes basic & we rewrite everything so that the objective function is one step closer to truth.

When this process stops the value of the objective function is maximized.

Problems that can be expressed as a linear program ⇒
- shortestest paths
- maximum flow
- minimum cost flow
- multicommodity flow

- The straightforward method of adding two polynomials takes $\Theta(n)$
- The straightforward method of multiplying two polynomials takes $\Theta(n^2)$

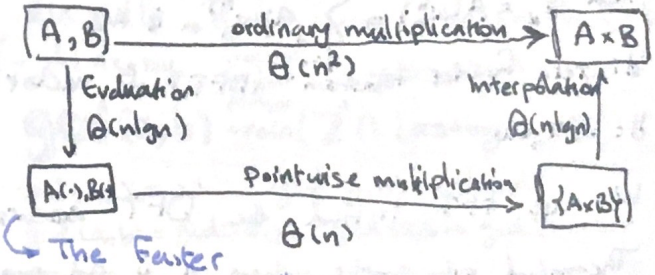
faster Fourier transform

FFT helps us multiply in $\Theta(n \lg n)$

Polynomial representation
- coefficient: $A(x_0) = a_0 + x_0(a_1 + x_0(\dots))$
- point-value: $A(x_0) = \{(x_0, y_0), \dots (x_n, y_n) \mid y_k = A(x_k)\}$

Evaluating $A(x_0)$ takes $\Theta(n)$ using Horner's method.

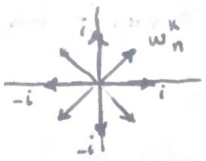
▶ For any n-point representation of a function the coefficient representation is unique (p.901)



↳ The Faster Fourier method

complex nth root of unity: $\omega^n = 1$

$\rightarrow \omega_n = e^{2\pi i/n} \Rightarrow \omega^n = 1 \rightarrow \omega \in \{\omega_n^k\}, 0 \le k < n$



↳ Principal nth root of unity

where: $e^{iu} = \cos u + i \sin u$

Properties of the complex nth roots of unity

- **cancellation**
  1. $n \ge 0, k \ge 0, d > 0 \Rightarrow \omega_{dn}^{dk} = \omega_n^k$
     $(\omega_n^{n/2} = \omega_2 = -1)$
- **halving**
  2. $n > 0, n \equiv 0 \bmod 2$, squares of n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.
- **summation**
  3. $n \nmid k \Rightarrow \sum_{j=0}^{n-1} (\omega_n^k)^j = 0$

When $A(x) = \sum_{j=0}^{n-1} a_j x^j$; $a = (a_0, a_1, \dots, a_{n-1})$

then $y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$ is the "discrete Fourier transform" (DFT) of vector $\vec{a}$.

$y = \mathrm{DFT}_n(a)$

We use FFT to find the DFT recursively

The method only works when n is an exact power of 2.

Normal method for multiplying two β-bit numbers takes $\Theta(\beta^2)$.

Fastest known method: $\Theta(\beta \cdot \lg\beta \cdot \lg\lg\beta)$

$a \mid b \rightarrow b = ak \rightsquigarrow |a| < |b| \vee b = 0$

$a \nmid b \rightarrow b = ak + r,\ r = b \bmod a$

For $a > 0$, 1 & a are the trivial divisors.

- $a > 1$ & has only trivial divisors $\rightarrow a$ is prime
- $a > 1$ & has other divisors $\rightarrow a$ is composite
- otherwise $\rightarrow a$ is neither prime nor composite

$b = aq + r$ ↝ remainder (residue)

divisor ↳ quotient

$[a]_n = \{a + kn \mid k \in \mathbb{Z}\}$ ↝ equivalence class modulo n

$\mathbb{Z}_n = \{[a_n] \mid 0 \le a \le n-1\}$

$d \mid a \wedge d \mid b \rightarrow d$ is a common divisor for a & b

↳ $d \mid ax + by$

$\gcd(a,b) = \min(\mathbb{Z}^+ \cap \{ax + by \mid x, y \in \mathbb{Z}\})$

(greatest common divisor; $\mathbb{Z}^+$ → positive integers)

$\gcd(0,0) = 0 \qquad \gcd(a,b) = \gcd(b,a)$

$\gcd(a,b) = \gcd(-a,b) \qquad \gcd(a,b) = \gcd(|a|,|b|)$

$\gcd(a,0) = |a| \qquad \gcd(a, ka) = |a|$

$d|a \wedge d|b \Rightarrow d|\gcd(a,b)$

$\gcd(na,nb) = n\cdot\gcd(a,b)$

we call a & n "relatively prime"

$n|ab \wedge \gcd(a,n)=1 \Rightarrow n|b$

Each composite integer has a unique factorization into its prime elements: $a = p_1^{e_1} p_2^{e_2} \cdots p_n^{e_n}$

$\gcd(a,b) = \gcd(b, a \bmod b)$

↳ Euclids algorithm: $\gcd(a,b) = \begin{cases} a & b=0 \\ \gcd(b, a \bmod b) & \end{cases}$

↳ $k \geq 1$, if $a>b\geq 1$ & $b < F_{k+1}$, Euclid(a,b) makes fewer than k recursive calls. (F_{k+1}: $k+1^{th}$ Fibonacci number)

group $(S, \oplus)$

- closure: $a,b \in S \Rightarrow a \oplus b \in S$
- identity: $\exists e \in S \Rightarrow e \oplus a = a \oplus e = a$
- associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- inverses: $\forall a : \exists b \Rightarrow a \oplus b = b \oplus a = e$

— If $(S, \oplus)$ satisfies $a \oplus b = b \oplus a$ it abelian

— If $|S| < \infty \Rightarrow$ it is finite

$(\mathbb{Z}_n, +_n)$ is a finite, abelian group

$(\mathbb{Z}_n^*, \cdot_n)$ is a finite, abelian group

↳ $\{[a_n] : \gcd(a,n)=1\}$

If $S' \subseteq S$ is a group, $(S', \oplus)$ is a subgroup.

$S' \neq S \Rightarrow (S', \oplus)$ is a "proper" subgroup.

Lagrange: $(S, \oplus)$ is finite, $(S', \oplus)$ is a subgroup then $|S'| \mid |S|$

$|\mathbb{Z}_n^*| = \phi(n)$

↳ Eulers ϕ function

$\phi(n) = n\prod_{p \text{ prime} \,\&\, p|n} (1 - \frac{1}{p})$

n prime $\Rightarrow \phi(n) = n-1$

$a^{(k)} = \overbrace{a \oplus a \oplus \cdots \oplus a}^{k} = \bigoplus_{i=1}^{k} a$

For $(\mathbb{Z}_n, +_n)$: $a^{(k)} = ka \bmod n$

For $(\mathbb{Z}_n^*, \cdot_n)$: $a^{(k)} = a^k \bmod n$

$\langle a \rangle = (\langle a \rangle, \oplus) = \{a^{(k)} : k \geq 1\} \Rightarrow$ the subgroup generated by "a"

$\text{ord}(a) = \min\{t : a^t = e\}$

$= |\langle a \rangle|$

if $d = \gcd(a,n) \Rightarrow \langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, (\frac{n}{d}-1)d\}$ in $\mathbb{Z}_n$

The Chinese remainder theorem lets us formulate the style of any number based on a set of prime numbers ($n = p_1 p_2 p_3 \cdots p_u k + r$)

$\Rightarrow$ It is possible to solve a linear equation using an extended form of Euclid's algorithm in modular form.

For $n = n_1 \cdot n_2 \cdot \ldots \cdot n_k$ when n_i & n_j are relatively prime the equations $x \equiv a_i \bmod n$ has a unique solution for x.

It is possible to calculate $a^b \equiv x \pmod{n}$ in $O(\beta)$ arithmatic & $O(\beta^3)$ bitwise operation if $a, b,$ & n are β-bit numbers. This is achieved by using repeated squaring.

$\mathcal{D}$: set of all permissible messages.
All messages such as M belong to $\mathcal{D}$.
We have: $\begin{cases} M = S_A(P_A(M)) \\ M = P_A(S_A(M)) \end{cases}$ (P_A: public key for A; S_A: secret key for A)

(M, σ) digitally signed copy of M where $\sigma = S_A(M)$

this is supposed to be okay!!!

① Select large primes p & q, $p \neq q$
② $n = p \cdot q$
③ e = small odd integer s.t. $\gcd(e, \phi(n)) = 1$ $(\phi(n) = (p-1)(q-1))$
④ $d \equiv e^{-1} \bmod \phi(n)$
↳ $\begin{cases} P_A = (e, n) \\ S_A = (d, n) \end{cases}$

$\Rightarrow \begin{cases} P(m) = m^e \bmod n \\ S(c) = c^d \bmod n \end{cases}$ where $\mathcal{D} = \mathbb{Z}_n$

A hybrid method for signing large messages is to sign M with k (k is small), then set $h = S(k)$ and send $(k(K), h)$, where $k(K(m)) = M$.

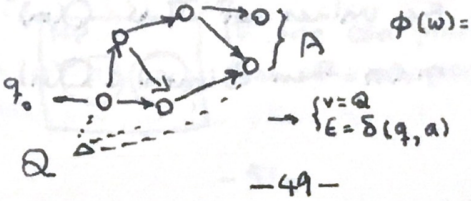
The big assumption behind the use of RSA is that factorizing n into p & q is infeasible. This rests on p & q being large enough. Generating large primes is difficult & at best we can be reasonably sure that the number we have generated is actually prime.
Pollard's rho heuristic is a decent attempt at factorizing in a reasonable time.

---

ch32

String Processing & Matching
- preprocessing: gather some data about text / pattern
- matching: find offset s of pattern in text

- naive: no preprocessing, matching: $O((n-m+1)m)$
- Rabin-Karp: compute hash of pattern, if hash of $S[i..i+m]$ is the same, compare $P[1..m]$ with $S[i..i+m]$ to check for a match
  - use a "rolling" hash to speed this up
- using finite automata



$\phi(w) = \begin{cases} q_0 & w = \varepsilon \\ \delta(\phi(w_k), a) & w = w_k a \end{cases}$

$\rightarrow \begin{cases} v = Q \\ \epsilon = \delta(q, a) \end{cases}$

$w \sqsupset x$: w is a suffix of x

$w \sqsubset x$: w is a prefix of x

$\delta(s, a)$ = next state after "s" when seeing "a"

↳ we go to the state wherein input "a" produces the longest prefix of the pattern.

→ using this, we iterate over the string to see if a given input takes us to the end of the pattern. ⇝ $O(\max(n, m^3 |\Sigma|))$ ← complexity of finding δ

▶ Knuth-Morris-Pratt

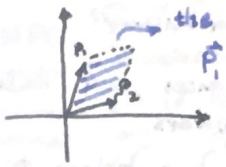
We can define $\pi[x]$ to be the length of the largest prefix $P_{..r}$ that $P_{..r} \sqsupset P_{..x}$.

Then if we ensure the $S_{k..k+m}$ matches $P_{..m}$ we know P is in S at k.

↳ $\Theta(n)$ where $n = |S|$

⇒ The precomputation necessary for calculating the values of π is $O(m)$ and since $m < n$ then $O(m+n) \in O(n)$

ch33



the cross product

$\vec{P}_1 \times \vec{P}_2$ → > 0: P_1 is clockwise from P_2

→ < 0: P_2 is clockwise from P_1

→ $= 0$: P_1 & P_2 are colinear

For $\vec{P}_1 = (x_1, y_1)$ & $\vec{P}_2 = (x_2, y_2)$: $\vec{P}_1 \times \vec{P}_2 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$

↳ this method is $O(1)$ and is not sensitive to division precision.

- given $\vec{P}_1$ & $\vec{P}_2$, are they clockwise? $O(1)$
- do we take a left turn at P_2 for $\vec{P_1P_2}$ & $\vec{P_2P_3}$?
- do $\vec{P_0P_1}$ & $\vec{P_2P_3}$ intersect? ↳ Graham

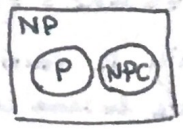
→ the convex hull ① $O(n \lg n)$ for those points ② $O(nh)$ → Jarvis ↳ number of convex corners

ch34

Problem Classes

- **P** : Polynomially solvable
- **NP** : Verifiable in polynomial time
- **NPC** : Hard to solve generally, not solvable by our computers

only decision problems



IF one can prove $P \cap NPC \neq \phi$ then $P = NP = NPC$

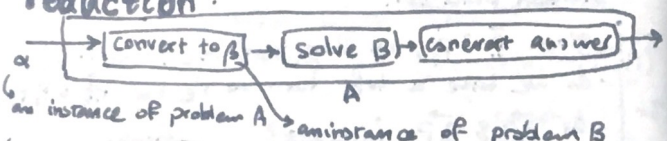
{optimization problems: Find best answer
{decision problems: is there an answer?

- Each problem has many instances and each instance might have many answers.

For "decision problems" the answers belong to $\{0,1\}$

k-CNF: ANDs of ORs of exactly k variables or their negations. $\Rightarrow$ 2-CNF $\in P$, 3-CNF $\in$ NPC

reduction:



↳ Problem A is in the same class as B.

NP-Complete

- Clique of (V,E) is (S,F) $S \subseteq V$, $F \subseteq E$ so that. (S,F) is a complete graph. Finding largest clique is interesting.
- A vertex cover $V' \subseteq V$: ensures that $E = \{(u,v) \mid u \in V' \vee v \in V'\}$
- Hamiltonian: cycle with all of V.
- Traveling salesman
- Subset sum: Find $S' \subseteq S$ the $\sum S' = t > 0$

k-Coloring: color vertices with k colors so that no two adjacent nodes are the same color.

NPC problems
① OK for small input sizes
② Solve special cases
③ Find approximate solutions

approximate answer → C, C^* ← actual answer

if $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$ it is $\rho(n)$-approximation.

↳ a 1-approximate answer is an optimal solution.

■ vertex cover: go over each edge, add both ends to C, remove incident edges of these ends, until the graph is covered. Return C: $|C| \leq 2|C^*|$

■ traveling salesman: Find the MST of the graph using PRIM, create sequence H of the preorder walk of the MST & return it. $\rho \approx 2$.
↳ only works if $c(u,v) + c(v,w) \geq c(u,w)$ (cost function)

■ set covering: with universe "U" and covers "S", keep selecting $s_i \in S$ that offers the most coverage on the uncovered part of U until it is fully covered.