



INTRODUCTION TO

3rd

ALGORITHMS

Cormen

Leiserson

Rivest

Stein

ch1 - ch35

53 pages

2015, 4, 26 - 2016, 11, 9

INTRODUCTION TO ALGORITHMS

Course

Prerequisite

Grade

Skills

Graphs - Trees

Algorithmic

Analysis - Design

- Insertion Sort: like sorting a hand of cards \rightarrow 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,  $\lfloor \frac{i+j}{2} \rfloor$ )
    A2 = mergeSort(A,  $\lceil \frac{i+j}{2} \rceil$ , j)
    return merge(A1, A2)
}

```

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

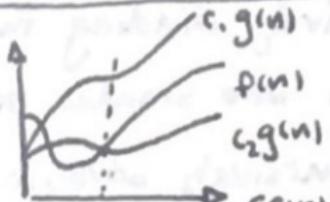
* SELECTION SORT : find the smaller one by one $\xrightarrow{O(n^2)}$

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

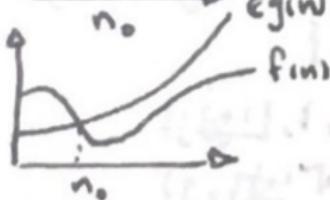
* BUBBLE SORT : large values sink down, smaller values bubble up $\xleftarrow{O(n^2)}$

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

ch 3

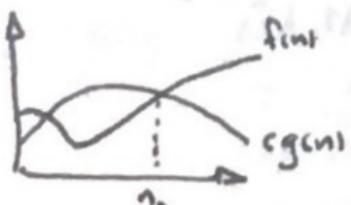


$$f(n) \in \Theta(g(n))$$



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

↳ O-notation is less tight



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

↳ \Omega-notation is less tight

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

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

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

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

$$F_i = F_{i-1} + F_{i-2} \rightsquigarrow F_i = \left[\frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right] \xrightarrow{\text{i}^{\text{th}} \text{ Fibonacci number}} \frac{1+\sqrt{5}}{2}$$

- $F_c^*(n)$: 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 ($\Theta(n)$)

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) \\ &\quad + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

* Matrix multiplication

precompute addition
matrices

$$\begin{aligned} \text{math: } & \Theta(n^3) \\ \text{break in quarters: } & \Theta(n^3) \\ \text{Strassen: } & \Theta(n^{2.37}) = \Theta(n^{2.3}) \end{aligned}$$

1 One way to solve recursions : substitute $T(n)$ 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. $f(n) = O(n^{\log_b^{a-\epsilon}}) \rightarrow T(n) = \Theta(n^{\log_b^a})$

①

3.2. $f(n) = \Theta(n^{\log_b^a}) \rightarrow T(n) = \Theta(n^{\log_b^a} \lg n)$

②

③

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

★ mind the gaps when using ^{the} master theorem

* We could also use generating functions to solve recursion

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

no find $p \in \mathbb{R} : \sum_{i=1}^k a_i b_i^p = 1$

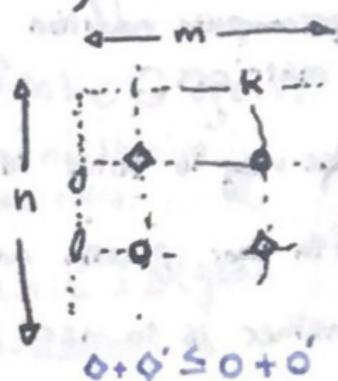
$$\Rightarrow T(n) = \Theta\left(n^p \cdot \left(1 + \int_1^n \frac{f(u) du}{u^{p+1}}\right)\right)$$

Monge arrays

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

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

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





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.

ch 5

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

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

$$\text{also: } E[\sum X] = \sum E[X]$$

⇒ 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	$\Theta(n \lg n)$	—
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$
stable CountingSort	$\Theta(k+n)$	$\Theta(k+n)$
Radix Sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$

(binary) heap data → min-heap:

Structure has a heap property. → max-heap:
 $\forall n: n.value \geq n.parent.value$

if implemented
as an array



$$\text{parent}(n) = \lfloor \frac{n}{2} \rfloor$$

$$\text{left}(n) = 2n$$

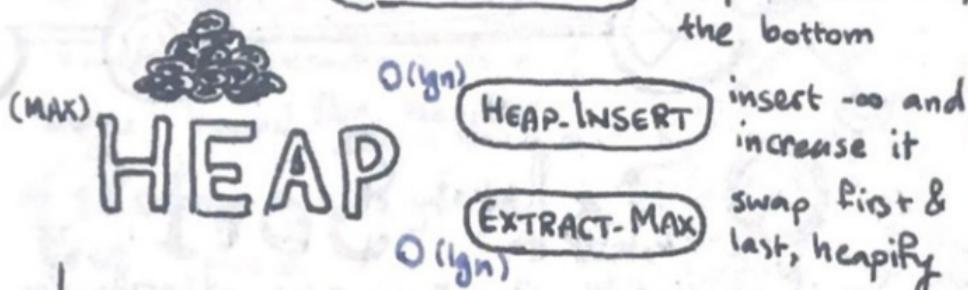
$$\text{right}(n) = 2n+1$$

MAX-HEAPIFY sinks down a violating node

$O(\lg n)$

$O(n)$ **BUILD-MAX-HEAP**

heapiify from bottom
the bottom



insert $-\infty$ and
increase it

swap first &
last, heapiify

float new value
upward

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

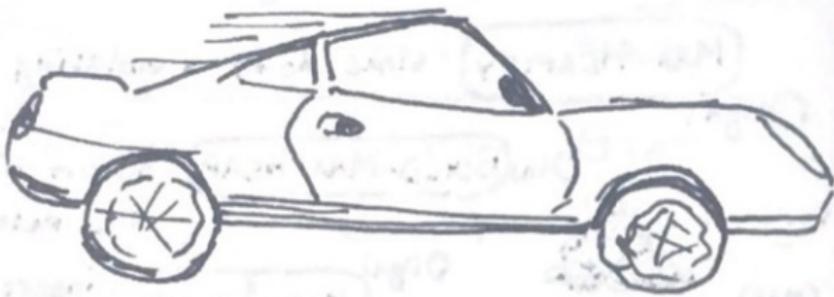
Young Tableau

Each row and each column is sorted.

an empty cell

2	3	4	5
4	6	12	19
7	8	13	22
14	∞	∞	∞

ch7

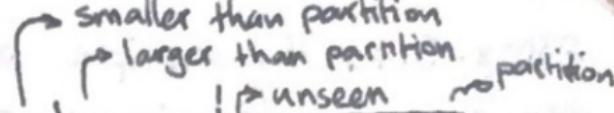


Quick Sort

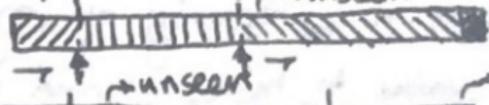
→ in practice the fastest sorting algorithm

- ▷ Sort one item in place
 - ▷ Call that the partition
 - ▷ Sort the halves recursively
- } Could choose a bad partition
⇒ $O(n^2)$

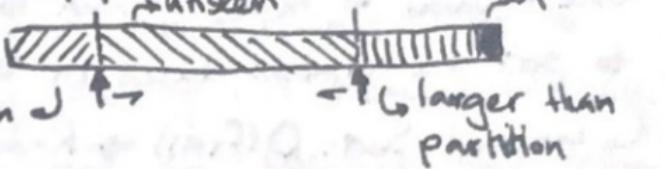
~ if we randomly choose an element
the partition we are increasing the chance
of an average case happening

Partitioning: 

1) CLRS:



2) Hoare:



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

ch 8

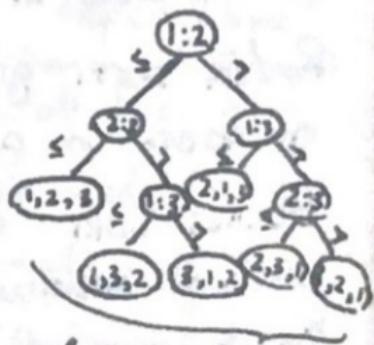
Comparison sort algorithms make no other assumption about their input than their being comparable.

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

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

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

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



l : reachable leaves

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 those digits.

↳ Internal Sort: $O(f(n)) \rightarrow$ Radix sort: $O(d f(n))$
Internally using counting sort: $\xrightarrow{\text{values of each digit}}$

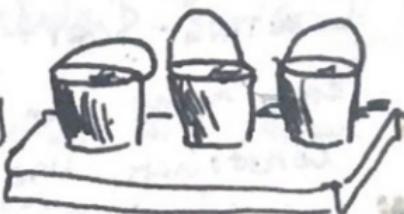
Radix sort: $O(d(n+k))$

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 $O(n)$



Selection: given a set of n distinct numbers, find i^{th} element that is larger than exactly $i-1$ elements in the set.

- ◆ Basic solution: sort ($O(n \lg n)$) and return the i^{th} element ($O(1)$) $\Rightarrow O(n \lg n)$

- ◆ Randomized selection: use randomized partitioning to figure out only the parts of the set we are interested in: $O(n)$

$\hookrightarrow \Theta(n^2)$: worst case expected time \hookrightarrow
 reason: $O(n!)$ $\rightarrow 3: O(n \lg n)$

- ◆ 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.

we choose
the pivot

\hookrightarrow We examine each position only once,
& sort arrays of constant size 5,

thus it is $O(n)$. worst

\hookrightarrow If array size is less than 6, just sort that and return the item

Set usage policies:

- First-in, first-out: Queue / Enqueue

- First-in, last-out: Stack \ Dequeue

Push

Pop



THE STACK

Linked lists:

- Singly linked →

- Doubly linked →

Insert $O(n)$ / Delete $O(1)$ / Search $O(n)$

→ if searching: $O(n)$

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 \leftarrow collision

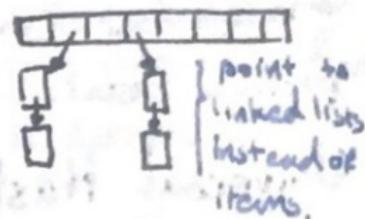
One way to handle

collisions is by chaining

$n \leftarrow$ items stored

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

$m \leftarrow$ 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"

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

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

Knowing something about the keys could help us.

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

- ◆ The multiplication method:

$h(k) = Lm(ka \bmod 1)$ $\rightarrow 0 < a < 1$; Knuth: $\phi = \frac{\sqrt{5}-1}{2}$

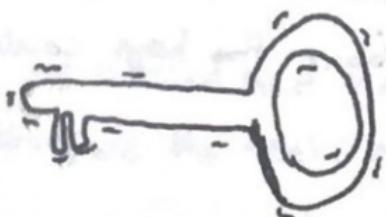
↳ Independent of m . If m is a power of 2, can perform faster.

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 $h \neq l$ for any function is at most $|\mathcal{H}|/m \rightarrow$ collision chance is $\frac{1}{m}$ for a random choosing. \rightarrow Restore simple uniform hashing assumption regardless of original keys

For prime "p": $\mathcal{H}_{pm} = \{h_{ab} : 1 \leq a \leq p, 0 \leq b < p\}$

$$\& h_{ab}^{(k)} = ((ak+b) \bmod p) \bmod m$$



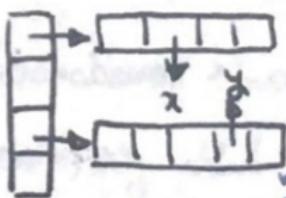
this set is always
guaranteed to be
universal

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.

~~key being hashed~~ $h(k, i)$ → 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: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$



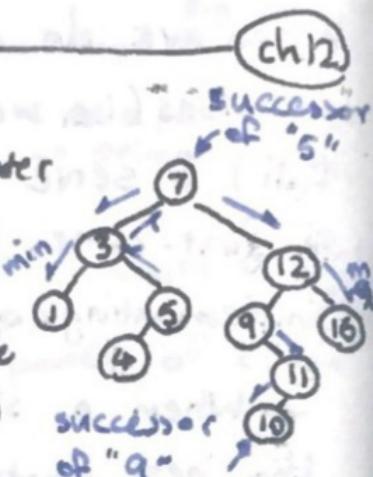
→ using two levels of hash tables with universal hashing → Perfect Hashing

- * Search has a worst-case perf of $O(1)$
- * Space complexity: $O(n)$ → number of stored items
- ▷ Keys have to remain static throughout application → if not search. recompute every time - 17 - in case of a change

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

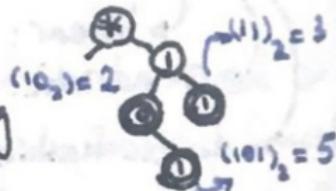
- pre order: parent before children
- post order: parent after children
- in order: parent between children



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

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

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



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

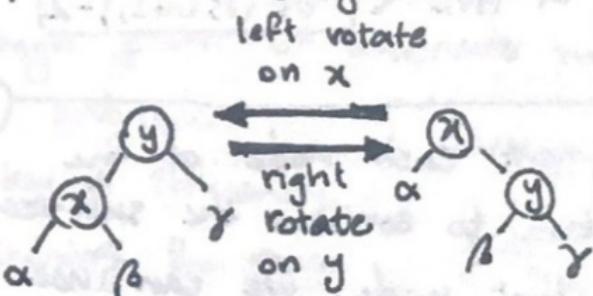
ch 13



- 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 (Search, Minimum, Maximum, Successor, & Predecessor) can perform in $O(\lg n)$



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 < $\log_2 (\sqrt{5(n+2)} - 2)$

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 $O(\lg n)$ time. ch 14

Augmenting Data Structures {

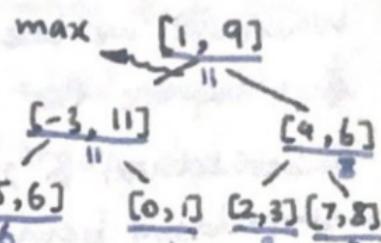
- ① Choose basic data structures
- ② Determine additional data
- ③ Modify existing operations to maintain this additional data
- ④ Develop new operations

⚠ the most important step

* If the new information for 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 n , we remove every m^{th} person until no one remains

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

$\hookrightarrow O(n)$ with dynamic programming

- 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 Programming → Trading off memory for speed

Notable problems:

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

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

ch16

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

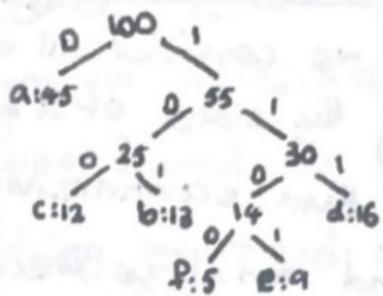
To solve the knapsack problem (0-1 version) we solve the problem for knapsacks of sizes 1..w over the first "i" items, always, considering if choosing item "i" & filling the rest of the knapsack with items 1..i-1 is better or not taking item i, by looking at how well we did with the first i-1 items.

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

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 \rightarrow 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: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$\xrightarrow{\text{amortized cost}}$ $\xrightarrow{\text{actual cost}}$ $\xrightarrow{\text{potential function}}$

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 contraction for data tables, the running time of n consecutive 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.

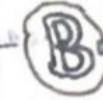


ch 18

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: $t-1 \leq n \leq 2t-1$

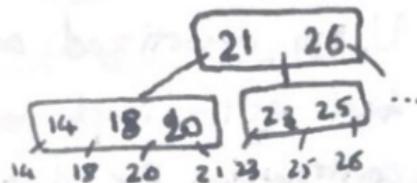
no one really knows why the B  Tree & the authors never explained!!

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

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

Sample B-Tree with

$$t = 3.$$



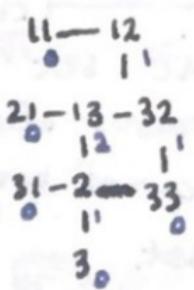
Tree height only increases when the root becomes "full".

- A B⁺-Tree has all leaves linked continuously and does not associate data with internal nodes.
- A B*-Tree has the property of: $\frac{2}{3}t \leq x \cdot n$

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

ch 19

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



The Fibonacci heap relies on a tree-like data structure 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	$\Theta(\lg n)$	$\Omega(\lg n)$
	<u>worst-case</u>	<u>amortized</u>

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

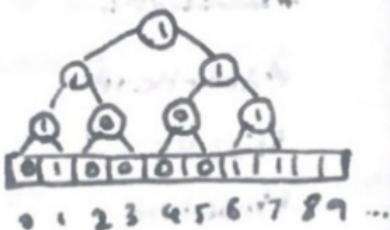
→ Necessary for priority queue.
 → won't work if we don't expose pointers to nodes



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.

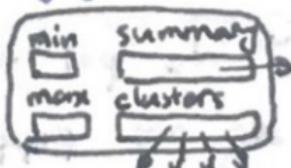
We can then superimpose this with with a binary tree to help with successor & other related operations

Applied to data structures



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

a given van Emde Boas tree node of size u has



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

We break index x into two parts:

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

segment: higher bits offset: lower bits

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

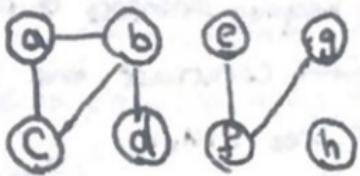
DS
isjoint sets

MAKE-SET
UNION
FIND-SET

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

it only has to be deterministic unless \rightarrow we change something.

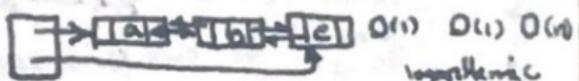
\Rightarrow finding connected components is one application.



$$\{a\} = \{a, b, c\}, \{e\} = \{e, f, g\}$$

$$\{h\} = \{h\} \quad \text{makeset find union}$$

• Linked list:



• Forest:



merge by concatenating lists \rightarrow weighted lists do better

merge by joining roots \rightarrow ranked trees do better.

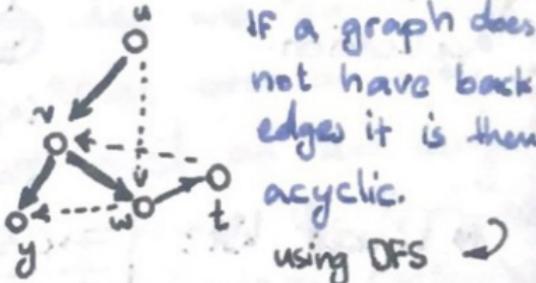
\rightarrow path compression: flattening child list in find path during FIND-SET

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 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.

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



► 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

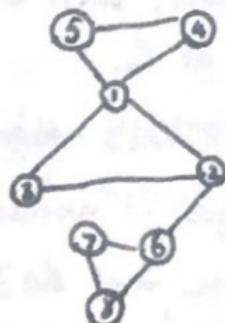
Forest forms strongly connected components

- 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.

$(5,4,1,2,3)$ no a biconnected component

① → an articulation point

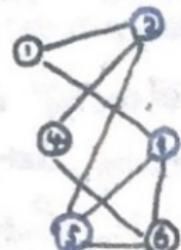
$(2,6)$ → a bridge



→ An Euler tour is a visitation that passes each edge exactly once → possible only if for all $v \in G.V$: $\text{in}(v) = \text{out}(v)$

ch 23

- 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 $W_A = \{w(u) | u \in A\}$



- 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 .

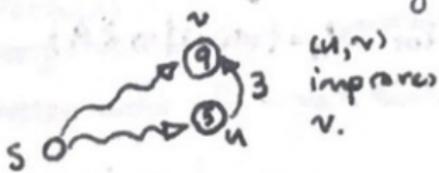
- * Kruskal's algorithm works by considering edges in nondecreasing order of weight & picking those that do not belong to a vertex previously picked. \Rightarrow grows a forest and joins the trees.
- * Prim's algorithm starts by traversing from a given node & always marking the best parent for a node,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

ch.24



To solve shortest path problem with destination 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 \rightarrow No solution $O(V \cdot E)$

We could also first sort all nodes topologically and go through them, relaxing all outgoing edges \rightarrow 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 $O(\lg V + E)$

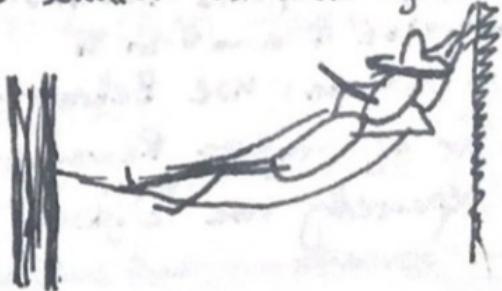
$$\begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \leq \begin{bmatrix} 4 \\ -2 \\ 3 \end{bmatrix} \rightarrow \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}$$

with Fibonacci heaps

$$\left\{ \begin{array}{l} v_2 - v_3 \leq 4 \\ v_3 - v_2 \leq -2 \\ v_3 - v_1 \leq 3 \end{array} \right.$$

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 $\rightsquigarrow O(VE \lg V)$

$O(V^3)$ • Finding matrix $L^{(n-1)}$ where L holds the shortest path, by equation $l_{ij} = \min_{\substack{L_i \\ L_j}} \{ l_{ik} + w_{kj} \}$

- Doing the above, only exploiting the fact that $W^{2m} = W^m \times W^m$ for square matrices

$\rightsquigarrow O(V^3 / gV)$ \rightarrow works better with adjacency matrix

- FLOYD-WARSHALL uses midpoint k to go from i , to j if it makes the path better

\rightarrow A transitive closure of graph $\rightsquigarrow O(V^3)$

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

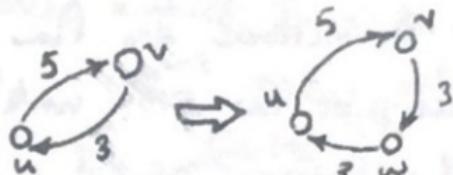
- Johnson: use Bellman-Ford to get an offset for all weights. Reweight all edges and then repeatedly use Dijkstra $\rightsquigarrow O(V^2 \lg V + VE)$

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:

$$|f| = \underbrace{\sum_{v \in V} f(s, v)}_{\text{what's flowing out of "s"}} - \underbrace{\sum_{v \in V} f(v, s)}_{\text{what's flowing into "s" usually zero}}$$

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

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

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

Flow augmentation

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

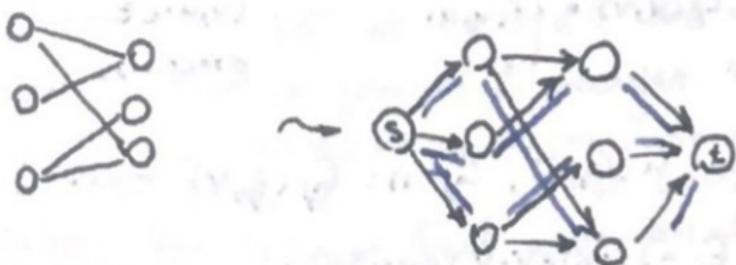
* Residual capacity of path p is:

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

(\rightarrow 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.

(\rightarrow Called EDMONDS-KARP when we use BFS to find augmenting path "p". $O(V.E)$)



- modelling bipartite matching using a maximum flow graph.

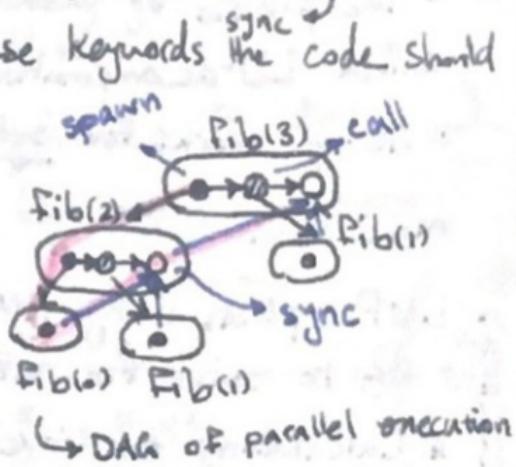
Another method for solving the network flow problem is a class of algorithms called the push-relabel method.

ch27

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

- after removing these keywords the code should function normally

- strand a is
reachable by
strand b : they
are logically in
series



- otherwise: they're logically in parallel



work: number of vertices

see
24.2

performance span: length of critical path

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

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

↪ Permutation matrix

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

Permutation array

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

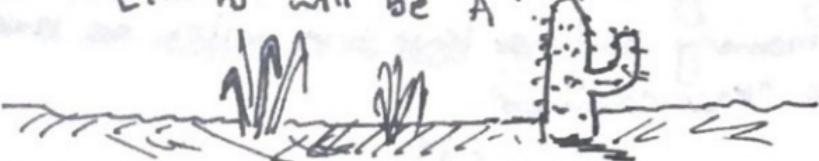
$$\rightarrow \begin{cases} Ly = Pb \\ Ux = y \end{cases}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} Pb \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

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

Always works for symmetric definite positive matrices.

- LUP pivots using permutation π_L 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 $(A^T A) \cdot x = (A^T b)$ for all A
- For X_i & I_i , which are column matrices we can solve "AX_i = I_i" n times for A_{n×n}.
If I_i is the i-th column of identity I_{n×n}
then $X = [X_1 \dots X_n]$ will be A⁻¹.



Elementary operations : switch: $R_i \leftrightarrow R_j$

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



add: $R_i + kR_j \rightarrow R_i$
multiply: $kR_i \rightarrow R_i$

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

- If $c = a+ib \Rightarrow \bar{c} = a-ib$ 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 $\bar{z}Mz$ 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.

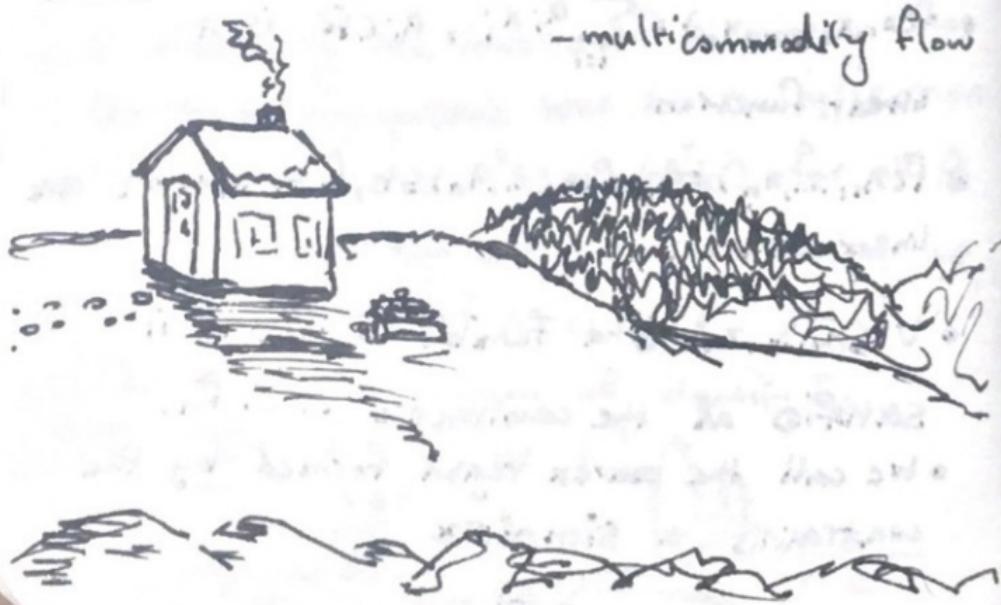
We select a nonbasic variable from the objective function that has a positive coefficient & increase it as much as possible without violating any of the constraints.

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

- Shortest paths
- maximum flow
- minimum cost flow
- multi-commodity flow



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

Fast Fourier transform

FFT

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

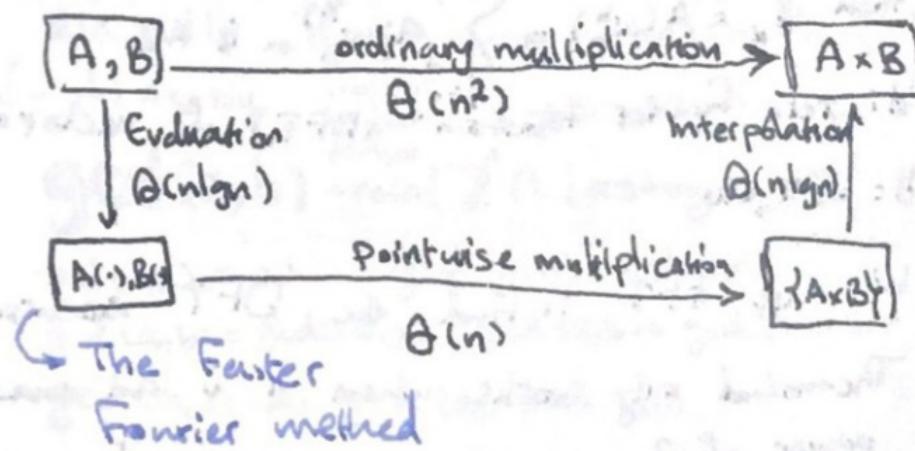
Polynomial representation coefficient representation

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots))$$

$$A(x_k) = \{(x_0, y_0), \dots, (x_n, y_n) \mid y_k = A(x_k)\}$$

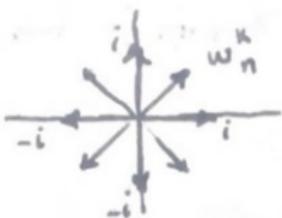
Evaluating $A(x_0)$ takes $\Theta(n)$ using Horner's method.

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



complex n^{th} root of unity: $\omega^n = 1$

$$\rightarrow \omega_n = e^{2\pi i/n} \Rightarrow \omega^n = 1 \rightarrow \omega \in \{\omega_n^k\}, \text{ where } k \in \mathbb{Z}$$



Principal n^{th} root of unity

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

Properties
of the complex
 n^{th} roots of
unity

cancellation

$$\textcircled{1} n > 0, k \geq 0, d \geq 0 \Rightarrow \omega_{dn}^k = \omega_n^k$$

$$(\omega_n^{n/2} = \omega_2 = -1)$$

halving:

\textcircled{2} $n > 0, n \equiv 0 \pmod{2}$, Squares of
 n complex n^{th} roots of unity
are the $n/2$ complex $(n/2)^{\text{th}}$ roots
of unity.

summation

$$\textcircled{3} n+k \Rightarrow \sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

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 a .

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

We use FFT to find the DFT recursive

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: $\Theta(\beta \lg \beta \cdot \lg \lg \beta)$

$$\{ \text{allo } \rightarrow b = ak \rightarrow |a| < |b| \vee |a| = 0 \}$$

$$\{ \text{allo } \rightarrow b = ak + r \rightarrow r = b \bmod a \}$$

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

$\begin{cases} a > 1 \& b \text{ has only trivial divisors} \rightarrow a \text{ is prime} \\ a > 1 \& b \text{ has other divisors} \rightarrow a \text{ is composite} \\ \text{otherwise} \rightarrow a \text{ is neither prime nor composite} \end{cases}$

$$b = aq + r \rightsquigarrow \text{remainder (residue)}$$

divisor $\xrightarrow{\quad}$ quotient

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

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

$d \mid a \wedge d \mid b \rightarrow d$ is a common divisor for $a \& b$
 $\hookrightarrow d \mid ax+by$ greatest common divisor \rightarrow positive integers

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

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

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

$$\gcd(a, 1) = 1 \mid a \quad \gcd(a, ka) = 1 \mid a$$

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

we call a & b :

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

"relatively prime"

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

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

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

→ Euclid's algorithm: $\gcd(a, b) = \begin{cases} a & b=0 \\ \gcd(b, a \bmod b) & \end{cases}$

↪ $k \geq 1$, if $a > b \geq 1$ & $b < F_{k+1}$, $\text{Euclid}(a, b)$ makes fewer than K recursive calls.

group (S, \oplus) $\left\{ \begin{array}{l} \text{closure: } a \oplus b \in S \Rightarrow a \oplus b \in S \\ \text{identity: } \exists e \in S \Rightarrow e \oplus a = a \oplus e = a \\ \text{associativity: } (a \oplus b) \oplus c = a \oplus (b \oplus c) \\ \text{inverses: } \forall a : \exists b \Rightarrow a \oplus b = b \oplus a = e \end{array} \right.$

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

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

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

$\{(Z_n^*, \cdot_n)\}$ is a finite, abelian group

↪ $\{[a] : \gcd(a, n) = 1\}$

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 $|S'| \mid |S|$

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

$$a^{(k)} = \underbrace{a \oplus a \oplus \dots \oplus a}_k = \bigoplus_{i=1}^k a$$

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

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

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

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

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

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

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

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

Euler's ϕ function
 $\phi(n) = n \prod_{p \text{ prime} \& p \mid n} (1 - \frac{1}{p})$

$$n \text{ prime} \Rightarrow \phi(n) = n - 1$$

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

D : set of all permissible messages

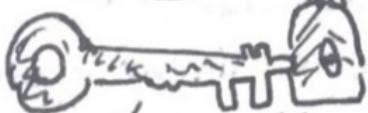
All messages such as M belong to D .

We have: $\begin{cases} M = S_A(P_A(M)) \xrightarrow{\text{public key for A}} \\ M = P_A(S_A(M)) \xrightarrow{\text{secret key for A}} \end{cases}$

(M, σ) digitally signed copy of M where

$$\sigma = S_A(M)$$

RSA



This is supposed to be a key!!!

$$\Rightarrow \begin{cases} P(M) = M^e \pmod{n} \\ S(C) = C^d \pmod{n} \end{cases} \quad \text{where } D = \varphi(n)$$

- ① Select ^{large} primes $p \neq q, p \nmid q$
- ② $n = p \cdot q$
- ③ $e = \text{small odd integer so that } \gcd(e, \varphi(n)) = 1 \quad (\varphi(n) = (p-1)(q-1))$
- ④ $d \equiv e^{-1} \pmod{\varphi(n)}$

$$\begin{cases} P_A = (e, n) \\ S_A = (d, n) \end{cases}$$

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 $K(K(M)) = M$.

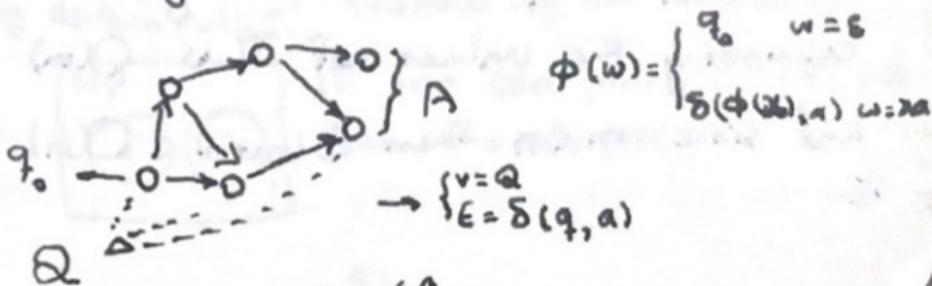
The big assumption behind the use of RSA is that factorizing n into $p \times q$ is infeasible. This rests on $p \times q$ being large enough. Generating large primes is difficult & at best we can be reasonably sure that the number we have generated is actually prime.

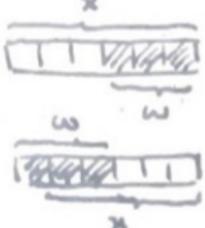
Pollard's rho heuristic is a decent attempt at factorizing in a reasonable time.

ch32

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

- naive: no preprocessing, matching: $O(nm)$
- 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





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

$w \sqsubset x : w$ is a ^{prefix} ~~suffix~~ 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. $\rightsquigarrow O(\max(n, m^3 |\Sigma|))$

complexity of
finding Δ

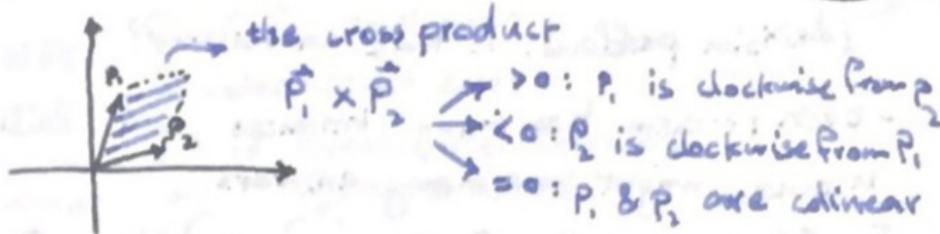
► Knuth-Morris-Pratt

We can define $\pi(x)$ to be the length of the longest prefix $P_{1..r}$ that $P_{1..r} \sqsupseteq P_{r..x}$.

Then if we ensure the $S_{k..k+m}$ matches $P_{1..m}$ ~~because~~ we know P is in S at k .

$\hookrightarrow \Theta(n)$ where $n = |S|$

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



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 $\vec{P}_1\vec{P}_2$ & $\vec{P}_2\vec{P}_3$?
- do $\vec{P}_1\vec{P}_2$ & $\vec{P}_2\vec{P}_3$ intersect? Graham

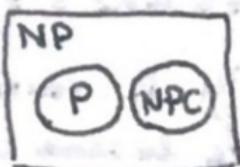


→ the convex hull $\textcircled{1} O(n \lg n)$

for these points $\textcircled{2} O(nh) \rightarrow$ Jarvis
↳ number of
convex corners

Problem Classes

P	: Polynomially solvable
NP	: Verifiable in polynomial time
NPC	: Hard to solve generally, not only decision problems



If one can prove $P \cap NPC \neq \emptyset$
then $P = NP = NPC$

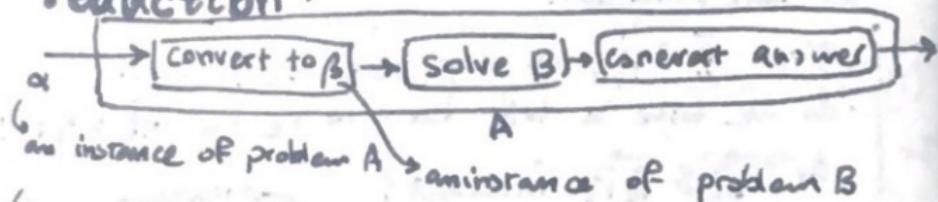
{ optimization problems: Find best answer
decision problems: is there an answer?

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

For "decision problems" the answers belong to to:

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

reduction:



(\hookrightarrow) Problem A is in the same class as B.

- NP-Complete**
- Clique of (V, E) is (S, F) so that $\subseteq V$ $\subseteq E$
 (S, F) is a complete graph. Finding largest clique is interesting.
 - A vertex cover $V' \subseteq V$: ensures that $E = \{(u, v) | u \in V' \vee v \in V'\}$
 - 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

approximate answer
 ③ Find approximate solutions
 \rightarrow actual Answer



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

\hookrightarrow a $\frac{1}{t}$ 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. $\rho \approx 2$.

\hookrightarrow only works if $c(u,v) + c(v,w) \geq \overbrace{c(u,w)}^{\text{cost function}}$

■ 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.