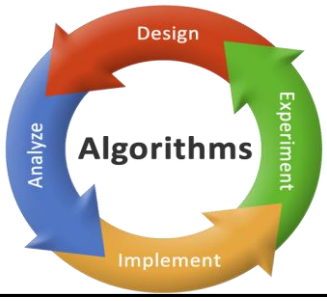
Algorithm Design and Analysis

Victor Adamchik | CS 15-451 | Spring 2015

Lecture 27 | Apr 29, 2015 | Carnegie Mellon University

Epilogue



We Had Some Lectures

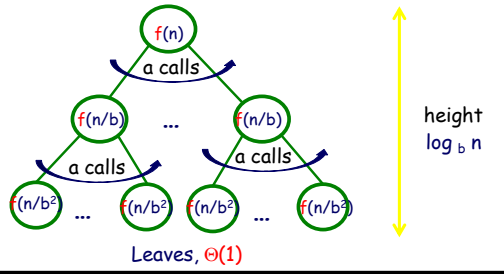
1. Solving Recurrences
2. Karatsuba/Strassen
3. FFT
4. DP I
5. DP II
6. Amortized Analysis
7. Splay Trees
8. Biconnected Graphs
9. Tarjan's SCC
10. Arborescences
11. Blossom Algorithm
12. Max Flow
13. Min Cost Max Flow
14. Fibonacci Heaps
15. Computational Geometry I
16. Computational Geometry II
17. Voronoi Diagrams
18. Linear Programming I
19. Linear Programming II
20. NP-Completeness
21. Approximation Algorithms
22. Online Algorithms
23. Randomized Online Algorithms
24. String Matching
25. Suffix Trees
26. Lowest Common Ancestor
27. Epilogue

#1. Solving Recurrences

Tree method:

$$T(n) = a \cdot T(n/b) + f(n)$$

$$T(1) = \Theta(1)$$



#1. Solving Recurrences

The Akra-Bazzi generalization:
(did not cover)

$$T(x) = \sum_{k=1}^{n} a_k \cdot T(b_k\, x) + f(x)$$

$$T(1) = \Theta(1)$$

#2. Karatsuba's Algorithm

Fast Integer multiplication: $\Theta(n^{1.58})$

3-way splitting: $\Theta(n^{1.46})$

k-way splitting: $\Theta(n^{\log_k(2k-1)})$

$$\log_k(2k-1) = \frac{\ln(2k-1)}{\ln k} = 1 + \frac{\ln(2 - 1/k)}{\ln k} > 1 + \varepsilon$$

#2. Strassen's Algorithm

For 2 x 2 matrices

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} s_1 + s_2 - s_4 + s_6 & s_4 - s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{pmatrix}$$

$s_1 = (a_{12}-a_{22})\,(b_{21}+b_{22})$
$s_2 = (a_{11}+a_{22})\,(b_{11}+b_{22})$
$s_3 = (a_{11}-a_{21})\,(b_{11}+b_{12})$
$s_4 = (a_{11}+a_{12})\,b_{22}$
$s_5 = a_{11}\,(b_{12}-b_{22})$
$s_6 = a_{22}\,(b_{21}-b_{11})$
$s_7 = (a_{21}+a_{22})\,b_{11}$

$\Theta(n^{\log 7}) = \Theta(n^{2.807})$.

#3. FFT

To compute the product $A(x)B(x)$ of polynomials (of order n)

$O(n \log n)$

1) evaluate $A(x)$ and $B(x)$ at $(2n+1)$ roots of unity, using the Vandermonde matrix
2) multiply $A(x_k)B(x_k)$, $O(n)$
3) then find the polynomial using Lagrange's interpolation via the Vandermonde matrix

$O(n \log n)$

#4-5. Dynamic Programming

Basic Steps of DP

1. Define subproblems.
2. Write the recurrence relation.
3. Prove that an algorithm is correct.
4. Compute its runtime complexity.

Examples:
The Longest Common Subsequence
The Knapsack Problem
The Matrix product
The Optimal BST
The Bellman-Ford algorithm
The Floyd-Warshall algorithm

#6. Amortized Analysis

Consider a sequence of n operations $\sigma_1, \sigma_2, \ldots$ on the data structure. Let the sequence of states through which the data structure passes be $s_0, s_1, \ldots$, and let the cost of operation σ_i be c_i.

Potential function Φ: $s_k \longrightarrow \mathbb{R}$

Define the amortized cost ac_i of operation σ_i by

$$ac_i = c_i + \Phi(s_i) - \Phi(s_{i-1})$$

#7. Splay Trees

BST with the splaying rules (zig, zig-zig and zig-zag)

Potential function $\Phi(T) = \sum_{x \in T} r(x) = \sum_{x \in T} \lfloor \log(s(x)) \rfloor$

where $s(x)$ = # nodes in subtree rooted at x.

If T is a linked list: $\Phi(T) = \sum_{k=1}^{n} \lfloor \log k \rfloor = \Theta(n \log n)$

If T is balanced, of height H

$$\Phi(T) = \sum_{k=0}^{H} 2^k \lfloor \log(2^{H+1-k} - 1) \rfloor = \Theta(2^H) = \Theta(n)$$

Access lemma: $AC(\text{splaying x to root t}) \leq 3(r(t)-r(x))+1$

Cor.: $AC(\text{splaying x to root t}) \leq 3 \log n + 1$

#8. Application of DFS

Classification of Edges:

Tree edges - are edges in the DFS

Forward edges – edges (u,v) connecting u to a descendant v in a depth-first tree

Back edges – edges (u,v) connecting u to an ancestor v in a depth-first tree

Cross edges – all other edges

#8. Biconnected Graphs

A vertex is an articulation point if its removal (with edges) disconnect a graph.

A connected graph is biconnected if it has no articulation points.

For each vertex we store two indexes. One is the counter of nodes we have visited so far dfs[v]. Second - the back index low[v].

low[v] is the DFS number of the lowest numbered vertex x (i.e. highest in the tree) such that there is a back edge from some descendent of v to x.

#9. Strongly Connected Components

Def. low[v] is the smallest dfs-number of a vertex reachable by a back edge from the subtree of v.

Def. A vertex is called a base if it has the lowest dfs number in the SCC.

Trace the differences between SCC and biconnected algorithms.

#9. APSP: Johnson's algorithm

It improves the runtime only when a graph has negative weights.

Algorithm:
- Reweight the graph, so all weights are nonnegative (by running Bellman-Ford's from a newly created vertex)
- Run Dijkstra's on all vertices

Complexity: $O(V E + V E \log V)$

#10. Arborescences

The Minimum Spanning Tree for Directed Graphs

Def. Given a digraph $G = (V, E)$ and a vertex $r \in V$, an arborescence (rooted at r) is a tree T s.t.
- T is a spanning tree of G if we ignore the direction of edges.
- There is a directed unique path in T from r to each other node $v \in V$.

Given a digraph G with a root node r and with a cost on each edge, compute an arborescence rooted at r of minimum cost.

The Algorithm

For each $v \neq r$ compute $\delta(v)$ – the mincost of edges entering *v*.

For each $v \neq r$ compute $w^*(u, v) = w(u, v) - \delta(v)$.

For each $v \neq r$ choose 0-cost edge entering v.

Let us call this subset of edges T.

If T forms an arborescence, we are done.

else

Contract every cycle C to a supernode

Repeat the algorithm

Extend an arborescence by adding all but one edge of C.

Return

#11. Blossom Algorithm

A maximum matching in an undirected graph

The approach is to search for an augmenting path. If one exists, we increase the size of current matching, o.w. ???

The algorithm finds either such a path or a blossom (cycle). Runtime $O(n^2 m)$.

If we find a blossom, we contract the graph and continue.

The algorithm constructs a series of BFS layers: the edges from an even layer are unmatched, and the edges from an odd layer are all matched.

#12. Max Flow

Given a directed graph with edge capacities

$c(u,v) \geq 0$ if $(u,v) \in E$

$c(u,v) = 0$ if $(u,v) \notin E$

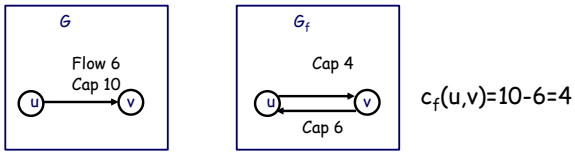
s is the source and t is a sink

Goal: push a max flow f(u,v) from s to t such that
1) $f(u, v) \leq c(u, v)$
2) $f(u, v) = - f(v, u)$
3) at a vertex $V-\{s, t\}$: flow-in = flow-out

Residual Network

Network: $G = (V, E)$ and flow f.
Residual capacity: $c_f(u,v) = c(u,v) - f(u,v)$.
Residual graph: $G_f(V, E_f)$, where

$$E_f = \{(u,v) \in V^2 \mid c_f(u,v) > 0\}$$



$c_f(u,v)=10-6=4$

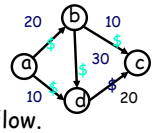
$c_f(v,u)=0-f(v,u)=f(u,v)=6$

The Ford-Fulkerson Algorithm

Given (G, s, t, c)
1) Start with $|f|=0$, so $f(e)=0$
2) Find an s-t path in G_f
3) Augment the flow along this path
4) Repeat until you stuck

Runtime: $O(|f|(E+V))$

#13. Min Cost Max Flow



Each edge has a capacity $w(e) \geq 0$

Each edge has a cost $c(e) \geq 0$ per unit flow.

The cost of flow is given by $|f| = \sum_{e \in E} f(e)\, c(e)$

The goal is to find the minimum cost flow

Residual Network:
$w_f(u,v) = w(u,v) - f(u,v)$
If $(v, u) \in E$ and $(u, v) \in E_f$ then $c(u, v) = -c(v, u)$.

Algorithm 1 (cycle canceling)

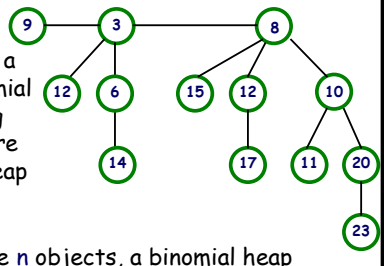
Given (G, s, t, w, c)
1) Find a max-flow ignoring cost (Ford-Fulkerson)
2) Construct G_f
3) Search G_f for a negative cost cycle (Bellman-Ford)
4) If $\nexists$ neg. cost cycle ⇒ done!
5) If $\exists$ neg. cost cycle ⇒ push a flow around.
6) Go to 2).

Runtime: $O(V E^2 \, cap_{max} \, cost_{max})$

#14. Fibonacci Heaps

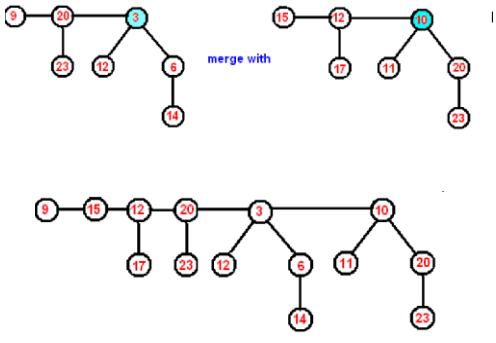
Binomial Heaps

A binomial heap is a collection of binomial trees in increasing order of size where each tree has a heap property.

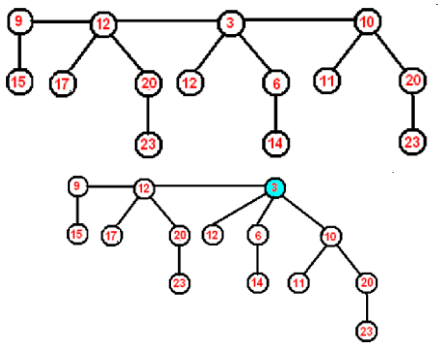


In order to store n objects, a binomial heap requires at most $\log n$ binomial trees.
$11_{10} = 1011_2$, thus we use B_3, B_1 and B_0

#14. Fibonacci Heaps - merging



#14. Fibonacci Heaps - merging



#14. Fibonacci Heaps - decreaseKey

We want it in O(1) time and deleteMin in O(log n).

Simple but wrong: take the node you want to decrease, and change its key, and disconnect it and its entire subtree from where it is, and attach it to the tree root list. This may result in O(n) for deleteMin.

Fredman-Tarjan marking algorithm, by implementing cut(k) function.

Theorem: Fibonacci heaps use O(1) amortized time for makeheap, insert, meld, and decreasekey. They use O(log n) amortized time for deletemin..

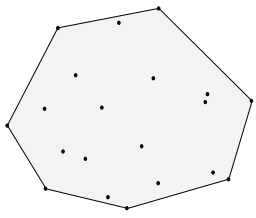
#15. Computing the Convex Hull

Given a set $S = \{p_1, p_2, \ldots, p_N\}$ of points in the plane, the convex hull $CH(S)$ is the smallest convex polygon in the plane that contains all of the points of S.

Convex Hull cannot be computed faster than

O(n log n)

in the worst-case



Graham's Scan

This was the first algorithm that showed convex hull computation in O(n log n) time.

Interesting, the algorithm has no obvious extension to three dimensions…

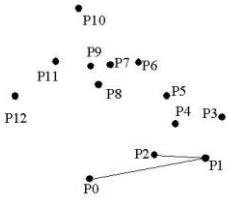
Graham's Scan

Take starting point p_0 to be bottom-most point
Connect it to all points and sort by angle wrt p_0.

Iterate over points in order

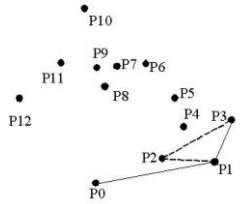
- If the next point forms a "left turn" with the previous two, add it to the list and continue
- If "right turn", remove the previous points until the angle becomes a left turn

Graham's Scan



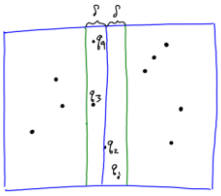
At this step, it realizes that it takes a right turn, so it backtracks and selects $P_1 P_3$

It selects the point P_1 with the least angle.
Then connect P_1 to P_2 & from P_2 to P_3



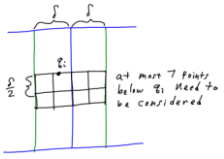
#16. Closest Pair

Given n points in the plane, find the pair of points that is the closest together.

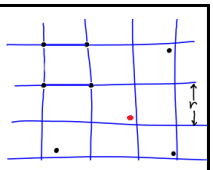


Divide and Conquer Algorithm:

the closest pair is either within the left half, within the right half, or it has one endpoint in the left half and one in the right half.



#16. Closest Pair



Randomized Algorithm:

Randomly permute the points.

Call the new ordering $p_1, p_2, \ldots, p_n$.

G = Makegrid(p_1, p_2) ← A grid of size $r=|p_1-p_2|$

for i = 3 to n do

  r = Insert(G, p_i) ← Inserts p_i into the grid returns a new grid size by checking 9 x 4 =36 points at most

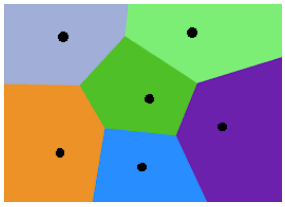
done

return r

#17. Voronoi Diagrams

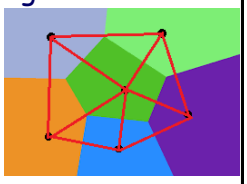
Given a set of (points in the plane, $s_1, s_2, \ldots s_n$

The Voronoi diagram partitions the plane into regions where the region associated with s_i is the set of points in the plane that are closer to s_i than any other.



#17. Delaunay Triangulations

A triangulation is a way of partitioning the convex hull of the points into triangles, where the vertices of the triangles are those points.



There are several ways by which to triangulate any given set of points:

AB is an edge of the Delaunay triangulation iff there is a circle passing through A and B so that all other points lie outside the circle.

Assuming no four sites are co-circular, then the Deleunay triangulation is unique.

#17. Voronoi and Delaunay

The Delaunay triangulation is the "dual" of the Voronoi diagram.

Given a Delaunay triangulation for a set of points we can compute the Voronoi diagram in O(n) time. And conversely.

We discussed two algorithms to compute the Delaunay triangulation:

- 3D convex hull
- divide and conquer

#18-19. Linear Programming

Given:

n variables $x_1, x_2, \ldots, x_n$

m linear constraints, $3x_1 + 4x_2 \leq 6$, $0 \leq x_3 \leq 3$ etc.

linear objective function, $3x_1+x_2+5x_3$

Goal:

Find values for $x_1, x_2, \ldots, x_n$ that satisfy constraints (*feasible*) and maximize the objective function.

Standard Form: $\max c^T x$, $Ax \leq b$, $x \geq 0$

If there is no max, it's *unbounded*.

#18-19. Linear Programming

Modeling Max Flow:

variables: one for each edge X_{uv} (a flow on that edge)

constraints: $0 \le X_{uv} \le c(u,v)$ and flow conservation

objective: max $\sum_u X_{ut} - \sum_u X_{tu}$

Modeling Min Cost Max Flow:

Solve the max flow ignoring the cost.

Add a new constraint flow=max and minimize the cost function.

The best algorithm : $O(n^{3.5} L)$ in L input bits which is not strongly polynomial

#18-19. Linear Programming

Modeling Dijkstra's algorithm:

variables: one for each vertex d_{sv} (a distance from the source s to vertex v)

constraints: $d_{sk} \le d_{si} + w_{ik}$, $d_{ss}=0$

objective: max d_{st} (max the dist. to the target)

The dual is min-cost max-flow:

variables: the flow on each edge f_{uv}

capacities are all 1s

costs are d_{uv}

#18-19. Integer Programming

Modeling MST:

variables: one for each edge $X_e \in \{0,1\}$

constraints: it must be a tree

1) with V-1 edges $\sum_{e \in E} X_e = |V| - 1$

2) (no cycles) for any subset S of edges $\sum_{e \in S} X_e \le |S| - 1$

objective: min $\sum_{e \in E} w_e X_e$

This is an exponential size IP.

#18-19. 2D-LP

We solve LP with 2 variables and m constraints in O(m)

Each constraint is a line (half-space), we are to find a point which is farthest in the given direction.

We make the following simplifications

- no constraint lines are normal to c
- we know a bounding box of the feasible region

Read the details of Seidel's algorithm…

The algorithm works in any fixed dimension.

#20. P vs. NP

P : a set of all languages L s.t. $\exists$ a polynomial time algorithm that decides on L

NP : a set of all languages L s.t. $\exists$ a polynomial time algorithm that verify $x \in L$.

Mapping reduction $A \le_p B$:

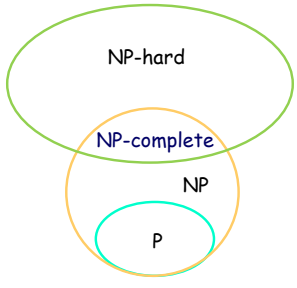
1. f is a polynomial time computable
2. $x \in A$ if and only if $f(x) \in B$.

NP-hard = $\{ L \subseteq \{0,1\}^* \mid \forall X \in NP \text{ and } X \le_p L \}$

NP-complete iff
1) $L \subseteq NP$
2) $L \subseteq$ NP-hard

Cook-Levin Theorem:
SAT is NP-complete

#20. P vs. NP



#20. P vs. NP

A recipe for proving any L ∈ NP-complete:

1) Prove L ∈ NP

2) Choose A ∈ NPC and reduce it to L

2.1) Describe mapping f:A -> L

2.2) Prove x ∈ A iff f(x) ∈ L

2.3) Prove f is polynomial

#21. Approximation Algorithms

Suppose we are given an NP-complete problem to solve. Can we develop polynomial-time algorithms that always produce a "good enough" solution?

Let P be a minimization problem, and I be an instance of P. Let ALG(I) be a solution returned by an algorithm, and let OPT(I) be an optimal solution. Then ALG(I) is said to be a c-approximation algorithm, if for ∀I, $ALG(I) \le c \cdot OPT(I)$.

Examples: Vertex Cover, Metric TSP.

Problems can be categorized to the best accuracy achieved by an approximation algorithm.

For some optimization problems, the approximation algorithms are unlikely to be possible. It is NP-hard to approximate them.

#22. Online Algorithms

Algorithms which have to make their decisions gradually as data arrives are called online algorithms.

The input can even be generated by an *adversary* that creates new input portions based on the system's reactions to previous ones. We seek algorithms that have a provably good performance.

Online Algorithms

Formally, an online algorithm receives a sequence of requests $\sigma_1, \sigma_2, \ldots, \sigma_m$.

When serving request σ_k, an online algorithm does not know requests σ_j with $j > k$.

Serving requests incurs cost and the goal is to minimize the total cost paid on the entire request sequence.

Let C_{OPT} be the optimum cost of offline algorithm, and C_{ALG} – the cost of online algorithm, then

An ALG is c-competitive, if ∃δ, that ∀I

$$C_{ALG}(I) \le c \cdot C_{OPT}(I) + \delta$$

#23. Randomized Online Algorithms

Intuition: we do better by going random!

Paging problem:
n pages in slow memory
k<n pages in fast memory

The algorithm must ensure that each requested page is in fast memory. Each time a page is moved into fast memory a cost of 1 is incurred. When a request is processed if the requested page is already in fast memory, cost = 0. OW, it must be moved into fast memory, and a page that is in fast memory must be "evicted" to make room for it.

#23. Randomized Online Algorithms

Paging Algorithms:
LRU (Least Recently Used)
LFG (Longest Forward Distance)

Cat and Mouse Game…

Claim: Randomized competitive ratio for Marking is O(log N)

#24. String Matching

The KMP Algorithm:

pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?

Theorem: At most 2N comparisons in total

KMP

How much can a string overlap with itself at each position?

a	b	a	b	b
0	0	1	2	0

Compute the length of the longest prefix of P that is a proper suffix of P.

It determines where to go whenever there is a mismatch in the next letter.

Failure Function

Consider all prefixes w[] of a pattern, define
$\pi[k] = \max(j<k \mid w[j] \text{ is a suffix } w[k])$

$\pi[k]$ is called a failure function, since it represents only backward transitions, in other words, it determines where to go whenever there is a mismatch in the next letter.

"aabaab", $\pi = \{0, 1, 0, 1, 2, 3\}$

"aabaaab", $\pi = \{0, 1, 0, 1, 2, 2, 3\}$

"aaabaabaaa", $\pi = \{0,1,2,0,1,2,0,1,2,3\}$

#24. String Matching: The Rabin-Karp Algorithm

We do not match a string against a given pattern, but rather compare their hash codes.

Theorem.
There's a pattern of length M and a text of length N. Pick a random prime $\in [2,\ldots, M N^2]$. The probability of getting a false match anywhere in the string is at most 2.53/N.

Rabin-Karp formalized

Let P[1 ... m] be a pattern and T[1 ... n] be a text. We define a pattern

$$P = 10^{m-1} P[1] + 10\, P[m-1] + \ldots + P[m]$$

and a shift in the text:

$$t_s = 10^{m-1} T[s+1] + 10\, T[s+m-1] + \ldots + T[s+m]$$

The value t_{s+1} can be obtained from t_s by

$$t_{s+1} = (t_s - 10^{m-1} T[s+1])\, 10 + T[s+m+1]$$

#25. Suffix Trees

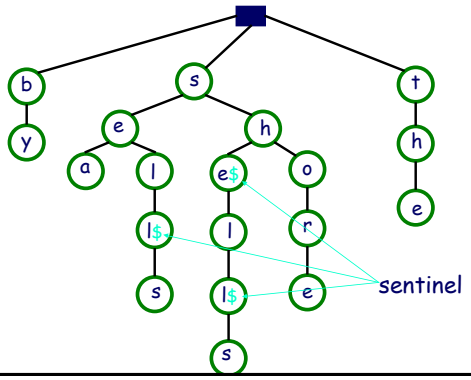
A trie is a data structure for storing a set of strings.

Each edge of the tree is labeled with a character.

Each node then implicitly represents a certain string of characters.

So to determine if a pattern occurs in our set we simply traverse down from the root of the tree one character at a time until we either (1) walk off the bottom of the tree, or (2) we stop at some node.

sells sea shells by the sea shore



#25. Suffix Trees

A suffix tree is a trie that stores all suffixes terminated by a special character.

Note, no string occurs as a prefix of any other

The space to store this data structure could be as large as $O(n^2)$.

but we can get it down to $O(n)$ by setting pointers to each substring.

Building suffix tree: can be done in linear time, Ukkonen's algorithm, though it was not discussed in lecture.

#26. Suffix Trees

Longest Common Substring:

Given two strings a and b, what is the longest substring that occurs in both of them?

Construct a new string s = a%b.

Now construct the suffix tree for s.

Every leaf of the suffix tree represents a suffix that begins in a or in b.

Mark every internal node with two bits: where each bit indicates where a suffix starts.

Now take the deepest node in the suffix tree that has both marks. This tells you the longest common substring.

#26. Lowest Common Ancestor

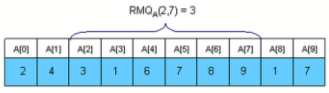
Given a rooted tree **T** and two nodes **u** and **v**, find the furthest node from the root that is an ancestor for both **u** and **v**.

It has been shown how to use RMQ (Range Min Query) for computing LCA.

LCA can be reduced to RMQ in linear time, so every algorithm that solves the RMQ problem will solve the LCA problem too.

#26. Range Min Query

Given an array, find the position of the item with the min value between two given indices.



Brute Force: for each {i,j} store its RMQ value.

Brute Force: $O(n^3)$ to build that new table

DP: $O(n^2)$ to build that table

#26. Range Min Query

Sparse Table Algorithm: $O(n \log n)$ to build the table

We will keep an array M[0, n-1][0,log n] where M[i][j] is the index of the minimum value in the sub array starting at i having length 2^j.

We do this preprocessing using DP.

To calculate $RMQ_A(i, j)$ we take two blocks: one starting at i to the right, another, from j to the left, and find the min.

$$RMQ_A(i,j) = \min\begin{cases} M[i][k] \\ M[j-2^k+1][k] \end{cases} \text{ where } k = \log(j-i+1)$$