richardwu.ca

CS 466/666 Course Notes

Design and Analysis of Algorithms

Anna Lubiw • Fall 2018 • University of Waterloo

Last Revision: October 22, 2018

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 September 10, 2018

1.1 Overview

How to design algorithms Assume: greedy, divide-and-conquer, dynamic programming

New: randomization, approximation, online algorithms

For one's basic repertoire, assume knowledge of basic data structures, graph algorithms, string algorithms.

Analyzing algorithms Assume: big Oh, worst case asymptotic analysis

New: amortized analysis, probabilistic analysis, analysis of approximation factors

Lower Bounds Assume: NP-completeness

New: hardness of approximation

1.2 Travelling Salesman Problem (TSP)

Given graph (V, E) with weights on edges $W : E \to \mathbb{R}^{\geq 0}$ find a *TSP tour* (i.e. a cycle that visits every vertex exactly once and has minimum weight or $\min \sum_{e \in C} w(e)$).

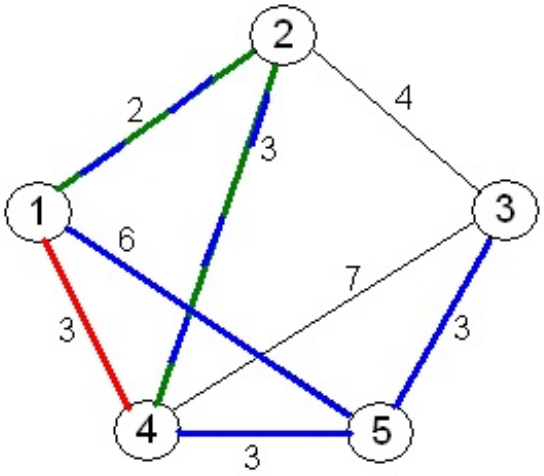


Figure 1.1: TSP tour is highlighted in blue.

Usually assume a **complete** graph (all possible $\binom{n}{2}$ edges exist). We can add missing edges with *high weight* to convert non-complete to complete.
Applications of TSP:

- School bus routes

- Delivery
- Tool path in manufacturing

To show the *decision version* of TSP (exist a tour of total weight $\leq k$) is NP-complete:

1. Show it is in NP (i.e. provide evidence (the tour itself) that there exists a TSP tour and show weights add up to $\leq k$ in polynomial time)
2. Show a known NP-complete problem reduces in polynomial time ($\leq_p$) to TSP (the Hamiltonian cycle problem can be reduced to TSP)

1.3 Approach to NP-complete problems

For NP-complete problems we want to:

- Find exact solutions
- Find fast algorithms
- Solve hard problems

We can in effect only choose two: for hard problems we give up on either *fastness* (exponential time algorithms) or *exactness* (approximation algorithms).

1.4 Metric TSP

An appproximation exists for the **metric TSP** version, where:

- $w(u, v) = w(v, u)$
- $w(u, v) \leq w(u, x) + w(x, v) \quad \forall x$

An algorithm (1977) was proposed for metric TSP:

1. Find a minimum spanning tree (MST) of the graph
2. Find a tour by walking *around* the tree.

   Think of doubling edges of MST to get Eulerian graph (i.e. every vetex has even degree), which lets us find an Eulerian tour traversing every edge once.

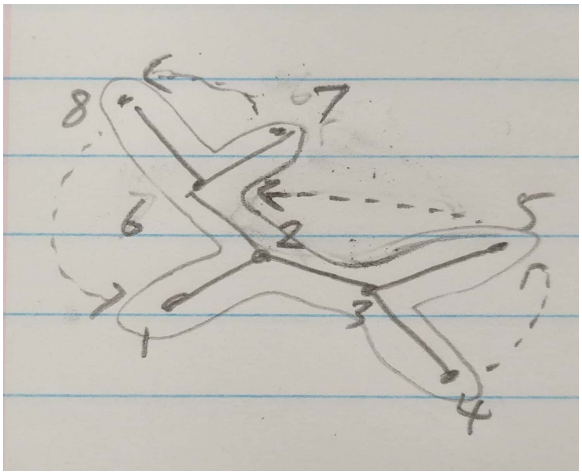


Figure 1.2: Eulerian tour is the solid line around the MST, the dotted lines show shortcuts taken, and the nodes are labelled in order.

3. Take shortcuts to avoid re-visiting vertices

   Instead of traversing a node twice (when walking around the MST), we take shortcuts and jump directly to the next unvisited node. By the triangle inequality our path should have a shorter path than if we actually traversed the MST edges twice, i.e.:

   $$l \leq 2l_{MST}$$

   where l is the length of our tour and l_{MST} is the length of the MST (remember we doubled every edge).

   Note the total path length we get will differ depending on which node we start with: thus one *could* attempt all paths to find the best path out of all approximated paths.

   Lemma 1.1. This algorithm is a **2-approximation**, i.e.:

   $$l \leq 2l_{TSP}$$

   where l_{TSP} is the minimum length of TSP.

   Proof. We need to show $l_{MST} \leq l_{TSP}$.

   Take the minimum TSP tour. Throw out an edge. This is a spanning tree T. Since

   $$l_{MST} \leq l_T \leq l_{TSP}$$

   the result follows. □

   Exercise 1.1. Show factor 2 can happen.

Analyzing/implementing this algorithm (let n # of vertices, m # of edges):
Steps 2 and 3 take $O(n + m)$.
Step 1 is our bottleneck: we've seen *Kruskal's* (sorted edges with union find to detect cycles) and *Prim's* (add shortest edge to un-visited vertex) MST algorithms.
Prim's took $O(m \log n)$ using a heap. An improvement is using a *Fibonnaci heap* (1987) which improves runtime for MST to $O(m + n \log n)$. A further improvement uses a randomized linear time algorithm for finding the MST (1995).

Theorem 1.1. For general TSP (no triangle inequality) if there is a polynomial time algorithm k-approximation for any constant k, then $P = NP$.

Proof. Exercise (hint: start with $k = 2$ and the Hamiltonian cycle problem. Show the 2-approximation can be used to solve the HC problem). □

Can we improve factor of 2 for metric case? Yes (Christofides 1996):

1. Compute MST
2. Look at vertices of odd degree in MST (there will be an even number). Find a minimum weight *perfect matching* of these vertices.

   The MST and perfect matching is Eulerian: take an Eulerian tour and take shortcuts (as before).

Implementation: we need a matching algorithm - the best runtime (in this situation) is $O(n^{2.5}(\log n)^{1.5})$ (1991).

Lemma 1.2. We claim $l \leq 1.5 l_{TSP}$. Note that $l \leq l_{MST} + l_M$ (where l_M is the total length of the minimum weight perfect matching). We must show that $l_{MST} \leq l_{TSP}$ and $l_M \leq \frac{1}{2} l_{TSP}$.
Sketch: to show $l_M \leq \frac{1}{2} l_{TSP}$, we show the smallest matching is $\leq \frac{1}{2} l_{TSP}$.

Open question: do better than 1.5 for metric TSP. We know the lower bound is 1.0045 (if we could get 1.0045-approximation then $P = NP$).
There is also the **Euclidean TSP** version where $w(e) =$ Euclidean length. We can get ϵ-approximation $\forall \epsilon > 0$.

2 September 12, 2018

2.1 Data structures

Every algorithm needs data strutures. Assume knowledge of:

- Priority queue (heap)
- Dictionary (hashing, balanced binary search trees)

In this course, we look at fancier/better DSes and also amortized analysis.

2.2 Priority queue

Operations supported by a priority queue (PQ) are: insert, delete min (delete), decrease-key, build, merge.
We usually implement PQs with a **heap**: a binary tree of elements where the parent is $\leq$ than the left and right children (min-heap), therefore the min. is at the root.

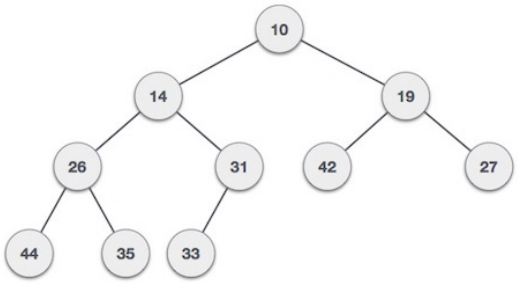


Figure 2.1: An example of a (min-)heap.

We assume that the shape is that of an almost perfect binary tree, where we always add a new element to the bottom right of the tree (if incomplete level) or the bottom left (start of a new level). We can store heaps in *level order in an array*, where for a given element indexed at i, acessing the parent via index $\lfloor \frac{i}{2} \rfloor$ and accessing the children via indexes $2i+1$ and $2i+2$.
The height of the tree is obviously $\theta(\log n)$. To implement each operation:

Insert Add new element in last position and bubble/sift up to recover ordering property. $\theta(\log n)$.

Delete min Remove root, it's the minimum. Move last position element to root and bubble/sift down (swap with smaller child). $\theta(\log n)$.

Derease-key Need only bubble/sift up (if $<$ parent). $\theta(\log n)$.

Build Repeated insertion is $\theta(n \log n)$.

Better approach: from bottom to top (after newly initialized heap in-place) bubble/sift down each element. $\theta(n)$.

2.3 Prim's algorithm

An application of heaps/PQs is for **Prim's MST algorithm**. Given graph with weights on edge, find spanning tree of minimum sum of edge weights.
For our given tree T so far, we find the minimum weight edge connecting to a new vertex. We begin with a PQ of all the edges and we will need to delete any newly added edge and any edges that lead to a newly added vertex.
Let $m = |E|$ number of edges and $n = |V|$ number of vertices. Every edge joins and leaves the heap once for a total of m times. We do delete min n times, so we have $\theta((m+n)\log m)$.
A better approach by using a heap of vertices and keeping track of shortest distance from T to vertex v gives us $\theta((m+n)\log n)$, which is only a constant time improvement since $\log m = \log n^2 = 2\log n$. We require the decrease-key operation here.
An even better approach for Prim's yields $\theta(m + n\log n)$ via *Fibonacci heaps*.

2.4 Binomial heaps

Binomial heaps improve the merge operation for heaps (which we can use for all other operations).
We use pointers to implement trees and each parent has an arbitrary number of k children (not necessarily binary tree) while maintaining heap order where parent is $\leq$ key of all children. We thus need to relax the shape; we also allow *multiple trees* for a given heap.
We define binomial trees B_k in terms of their rank k, which also coincides with the degree of the root.

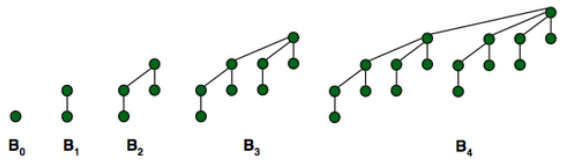


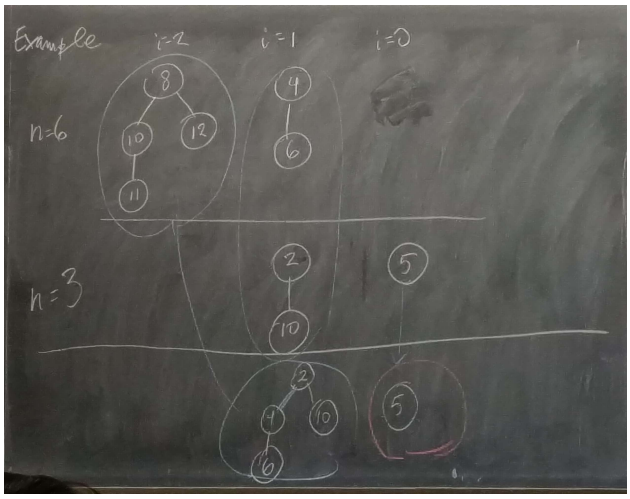
Figure 2.2: Example of five binomial trees with ranks $0, 1, \ldots, 4$.

In general, the number of node in B_k is $2^k = 2^{k-1} + 2^{k-1}$.
The height of B_k is k since, by induction, we have the recurrence $height(k) = 1 + height(k-1)$.

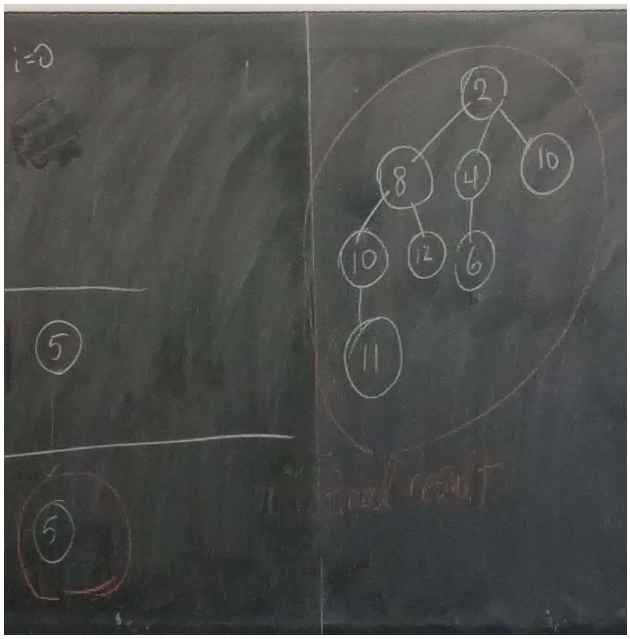
The number of nodes at depth i in B_k is

$$\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

(show by induction).
Brief counting proof of binomial equivalence: given k items from which we want to choose i items, we can either picked the first item or not: if we did not pick the first element, then we need to pick i items from the remaining $k-1$ items; if we did pick the first element, we need to pick $i-1$ items from the remaining $k-1$ items.
Note that B_k's only permit powers of 2 number of elements: thus a **binomial heap** for n elements use a collection of B_is (heap ordered), at most one for each rank.
E.g. for $n = 13$, we have $13 = 2^3 + 2^2 + 2^0$ (from binary 1101 so we use B_3, B_2, B_0).
It *does not matter* which B_i's contain a particular element.
We use $\theta(\log n)$ trees for a given binomial heap with n elements (from binary number expansion).
For merging two binomial heaps, we follow addition of two binary numbers (which are our bitmasks of trees). E.g. for adding a heap with 6 elements to a heap with 3 elements, we add $110 + 11 = 1001$ resulting in a binomial heap with a B_3 and B_0 tree.
When merging two trees of the same rank k, we simply take the tree with the larger root and add it as a children of the root of the other tree.



(a) We carry through the B_0 from the second heap and merge the B_1 trees from two binomial heaps by making the tree with the larger root the children of the other root.



(b) After merging the newly B_2 with the B_2 tree from the first heap into a B_3 tree, we get our final resulting binomial heap with a B_0 and B_3 tree.

Analysis of operations:

Merge Joining two B_i's take $\theta(1)$. We join up to $\log n$ trees so we have $\theta(\log n)$.

Insert Merge binomial heap with single B_0 (one new element): again $\theta(\log n)$ (since we have the worst case when we insert into a heap with $2^m - 1$ elements).

Delete min Takes $\theta(\log n)$ to find the minimum by checking roots of all trees. Once removing the root from the

tree B_k, we end up with $k-1$ trees $(B_{k-1}, \ldots, B_0)$ which we need to merge with the other trees (merge operation is also $\theta(\log n)$) so we have $\theta(\log n)$ overall.

Decrease-key After decreasing key, we bubble/sift up as necessary like before, which takes $\theta(\log n)$ since each tree has height at most $\log n$.

Build Repeated insertion appears to be $O(n \log n)$, but in fact it is $\theta(n)$. This can be seen by repeated addition of 1 in binary to our cumulative total: we only do merges at certain times.

Proof. In general, the cost of incrementing a k-bit counter has a *worst-case cost* of $k+1$ ($\theta(k)$ bit operations) where you add 1 to $11\ldots1$.

We show that incrementing from 0 to n is $\theta(n)$:

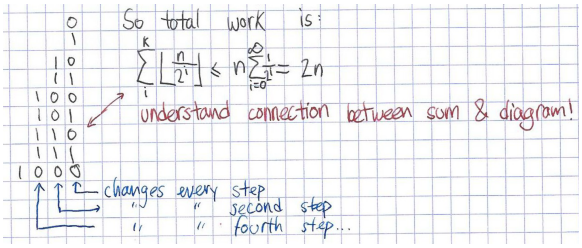


Figure 2.4: A sequence of incrementing from 0 to 1000_2 or 8_{10}. We observe that for binary bit i (rightmost is bit 0) the value of it changes every 2^i step. Thus for incrementing up to n, we sum up the number of times each bit i changes for k bits, which gives us $2n$ changes.

□

3 September 17, 2018

3.1 Amortization

Definition 3.1 (Amortized cost)**.** A sequence of m operations takes total cost $T(m)$: the **amortize cost** of one operation is thus $\frac{T(m)}{m}$.

3.2 Amortization "potential" method

Idea: use an "accounting trick" ("potential" in the physics sense)

Potential "savings" in bank account.

Cost the "true" cost.

Charge artificial: over/under-estimate cost at time of operation.

If charge > cost, we put excess in the bank (add to potential).

If cost > charge, extra has to come out of bank account.

Let Φ_i denote the potential after the ith operation, thus

$$\Phi_i = \Phi_{i-1} + \text{charge}(i) - \text{cost}(i)$$

Theorem 3.1. If the final potential $\geq$ initial potential (almost always 0) then the amortized cost per operation $\leq$ max charge.

Proof. The total charge we've introduced is

$$\begin{aligned}\sum_{i=1}^{m} \text{charge}(i) &= \sum_{i=1}^{m} \text{cost}(i) + \sum_{i=1}^{m} \Phi_i - \sum_{i=1}^{m} \Phi_{i-1} \\ &= \sum_{i=1}^{m} \text{cost}(i) + \Phi_m - \Phi_0\end{aligned}$$

Since $\Phi_m - \Phi_0 \geq 0$ then $\sum_{i=1}^{m} \text{charge}(i) \geq \sum_{i=1}^{m} \text{cost}(i)$.
Recall that we have for amortized cost

$$\frac{\sum \text{cost}(i)}{m} \leq \frac{\sum \text{charge}(i)}{m} \leq \text{ max charge}$$

□

To do potential analysis, devise potential/charge for each operation such that $\Phi_m \geq \Phi_0$ (the bank is never "in the red") and max charge is *small*.

Example 3.1. Applying the potential method to the binary counter example, add an extra "\$1" to each basic bit operation to compensate for when we roll over from string of all ones i.e. charge$(i) = 2$ (1 for the bit operation and "storing" the other 1 for the future).
By the theorem (assuming hypothesis holds), our amortized cost should be 2 per op.
Let's verify, initial potential $\Phi_0 = 0$

counter	cost	charge	potential
0 0 0 0	1		0
0 0 0 1	1	2	1
0 0 1 0	2	2	1
0 0 1 1	1	2	2
0 1 0 0	3	2	1
0 1 0 1	1	2	2

Intuition: potential is equal to the # of ones in counter, so final potential ≥ 0 (initial potential). Note that with cost of $\frac{3}{2}$ this fails.

Claim. Potential = # of ones in binary expansion of counter.
If this claim holds, final potential always ≥ 0 since we have at least one 1 in binary counter.

Proof. Proof by induction. Suppose current counter is

$$\begin{aligned}&01011\ldots011\ldots1 \\ &\quad\ldots100\ldots0\end{aligned}$$

where we have t_i ones at the end.

We thus have

$$\begin{aligned}\phi_i &= \phi_{i-1} + \text{charge}(i) - \text{cost}(i) \\ &= \phi_{i-1} + 2 - (t_i + 1) \\ &= \phi_{i-1} - t_i + 1\end{aligned}$$

Where we zeroed out our t_i ones (subtract) and added one 1. □

While potential method seems harder than previous sum argument, it is much more powerful in general.
This gives us $\theta(n)$ since if amortized cost is p (some constant), then n ops cost $pn \in \theta(n)$.

3.3 Summary of mergeable heaps

	binomial heap	lazy binomial heap	Fibonacci heaps
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ (amortized)	$O(\log n)$ (amortized)
merge	$O(\log n)$	$O(1)$	$O(1)$
decrease-key	$O(\log n)$ (bubble-up)	$O(\log n)$	$O(1)$ (amortized; improves MST time)
build	$\theta(n)$	$O(n)$	$O(n)$

3.4 Lazy binomial heaps

Idea: be lazy on merge/insert i.e. allow *multiple trees of same size*. Catch up on delete-min: re-combine to form a proper binomial heap (i.e. when delete-min occurs).
For delete-min with lazy binomial heaps:

1. Look at all roots to find min, remove this root.
2. Consolidate trees:

```
1    for rank = 1 to max rank:
2      while there are >= 2 trees of this rank:
3        link them into one tree
```

   where max rank is $\theta(\log n)$

Recall that rank = degree of root = height of tree.
It seems the worst case cost is $\theta(n)$ (after inserting n singletons).

Theorem 3.2. Lazy binomial heaps have $O(\log n)$ amortized cost for delete-min and $O(1)$ for insert/merge.

Proof. Let the potential be the # of trees and $\Phi_0 = 0$.
Clearly $\Phi_m \geq 0$ (we never have negative # of trees) so our previous result for the potential method applies.
We need to determine the charge per operation. Recall

$$\text{charge}(i) = \text{cost}(i) + \Phi_i - \Phi_{i-1}$$

Merge cost is 1 (not doing anything), and # of trees is the same (we combine the potentials of the two trees, no additional trees). So we have charge of 1.

Insert cost is 1, # of trees increases by 1, so we have charge of 2.

Delete-min Let r be the degree of the min node ($O(\log n)$), t be the number of trees before delete-min (Φ_{i-1}).

Thus consolidation will be invoked on $t - 1 + r$ number of trees.

Thus the cost will be $\leq t - 1 + r + O(\log n)$, where we have to merge/link at most $t - 1 + r$ times.

$O(\log n)$ is our loop from rank 1 to max rank ($O(\log n)$) and also keeping track of the # of trees of each rank.

Ultimately $\Phi_i \in O(\log n)$ since we end up with a Binomial heap with $O(\log n)$ trees.

We have

$$\begin{aligned}\text{amortize cost} &\overset{theorem}{\leq} \text{max charge} \\ &\leq \text{cost}(i) + \Phi_i - \Phi_{i-1} \\ &\leq t - 1 + r + O(\log n) - t \\ &\leq r + O(\log n) \\ &\in O(\log n)\end{aligned}$$

since $r \in O(\log n)$.

Thus delete-min has $O(n)$ worst case but $O(\log n)$ amortized.

□

3.5 Fibonacci heaps

Improvement on lazy binomial heaps by reducing `decrease-key` to $O(1)$ amortized time (vs $O(\log n)$).
Recall `decrease-key` bubbles-up O(height of tree): instead cut the tree. However we must be careful of the # of resulting trees and the nodes per tree changes (not 2^i anymore).
Instead, the number of nodes in a Fiobnacci tree of rank k is $\geq (k+2)$th Fibonacci number (f_{k+2}), where the Fibonacci numbers are $0, 1, 1, 2, 3, 5, \ldots$ with $f_0 = 0$.
This gives rank $\in O(\log n)$ because $f_k \in \theta(\phi^k)$.

4 September 19, 2018

4.1 Splay trees

Recall: a dictionary has keys from *totally ordered* universe and supports the operations **insert**, **delete**, and **search** (by key).
They can be implemented via hashing or balanced binary search trees. Recall for a **binary search tree** we have:

Search follow search tree invariant (left subtree $<$ root, right subtree $>$ root)

Insert insert where search fails

Delete Replace node to be deleted by either in-order successor (left-most child in right sub-tree), OR in-order predecessor (right-most child in left sub-tree).

Recursively delete chosen successor or predecessor, respectively.

Obviously if node to be deleted has 0 or 1 child, one can simply attach the child to the parent of the deleted node.

All operations for a binary search tree take $O(\text{height of tree})$; if balanced then height $\in O(\log n)$.
Some balance search trees variants include the **AVL tree** and **red-black** tree, which both employs rotation to balance the tree.

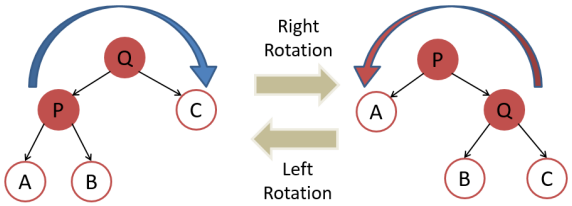


Figure 4.1: Right and left tree rotations. Note that the shorter paths that do not go through both P and Q become longer after a rotation. This helps balances out the height whenever a violation of the search tree invariant (AVL vs red-black) is violated.

Splay trees (Sleator & Tarjan, 1985) are a variant of balanced binary search trees

- $O(\log n)$ amortized cost per operation
- Easier to implement than AVL and red-black trees
- Do not need to keep balance information
- Careful: tree may become unbalanced
- Danger: repeated search for deep nodes.
  Fix: adjust tree whenever node is "touched".

The operations for a splay tree are

Splay(x) repeat until some target x node is root. We have up to 3 cases for where x is relative to its parent and grandparent:

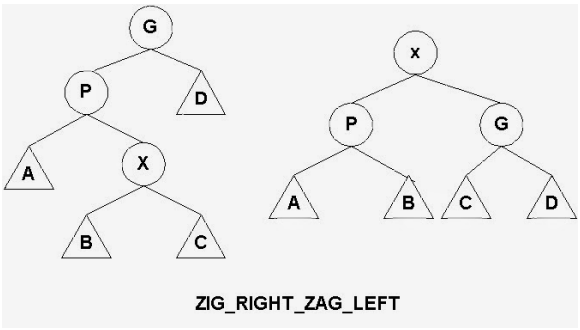


Figure 4.2: Zig-zag case: We want to lift x to the root where there is a zig-zag pattern up though parent P and grandparent G. We perform a left-rotation on x first, then perform a right rotation on x.

Case 1: zig-zag Equivalent to two rotations on x.

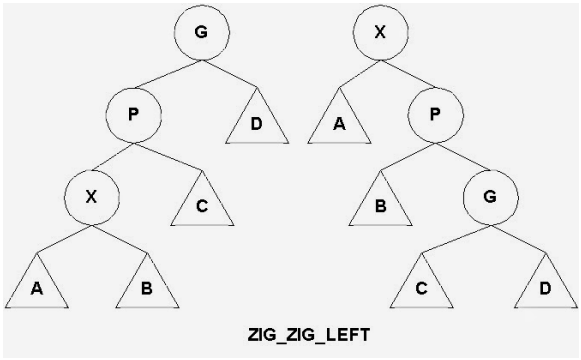


Figure 4.3: Zig-zig case: We want to lift x to the root where there is a straight pattern up though parent P and grandparent G. We perform a right rotation on P first then a right-rotation on x.

Case 2: zig-zig Equivalent to one rotation on y then one rotation on x.

Case 3: no grandparent or "zig" Single rotation on x.

Search After finding x (or place where search fails) using the usual search algorithm, we splay on x.

Insert Do usual insert on x, then splay(x).

Delete Do usual delete then splay parent of removed node.

4.2 Amortized analysis of splay trees

Goal: $O(\log n)$ amortized cost per operation. Recall that

$$\text{charge} = \text{cost} + \Delta\Phi$$

wher $\Delta\Phi = \Phi_i - \Phi_{i-1}$ or the change in potential.
Denote $D(x)$ as the # of descendants of x (including x itself).
Denote $r(x) = \log D(x)$, which is the best height possible for subtree rooted at x given $D(x)$.
Define our potential $\Phi(\text{tree}) = \sum_{x \text{ a node}} r(x)$.
For a degenerate tree with one single path of nodes down (height n), we have

$$\Phi = \sum_{i=1}^{n} \log i \in O(n \log n)$$

For a perfectly balanced tree, note that for a given node at height h, it has 2^h descendants and thus $r(x_h) = \log 2^h = h$.
So we have

$$\begin{aligned}
\Phi &= \sum_{\text{all nodes}} \text{height} \\
&= \sum_{h=1}^{\text{height of tree}} h \cdot \frac{n}{2^h} \qquad\qquad \frac{n}{2^h} = \text{ \# of nodes at height } h \\
&= n \sum_{h=1}^{\text{height of tree}} \frac{h}{2^h} \\
&\in O(n)
\end{aligned}$$

where we use the identity $S = \sum \frac{i}{2^i} \in O(1)$ (proof: take $2S - S$ and cancel out individual terms of the expanded series).
We need to analyze each of

1. zig, zig-zag and zig-zig
2. splay(x)
3. insert, delete and search

Denote r' and D' as the new rank and new # of descendents, respectively.

Claim. Amortized cost of one operation (i.e. our charge per operation) on a node x is

$$\text{charge} \leq \begin{cases} 3(r'(x) - r(x)) & \text{for zig-zag and zig-zig} \\ 3(r'(x) - r(x)) + 1 & \text{for zig} \end{cases}$$

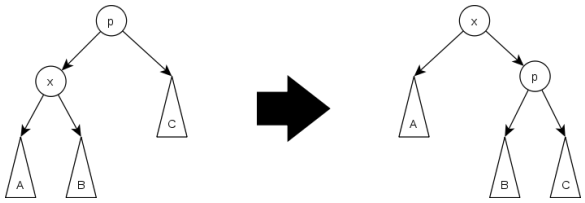


Figure 4.4: Zig (case 3) from before.

Proof. **Zig** Notice that $r'(x) = r(p)$ (i.e. x in the new tree has the same # of descendants or rank as old p).

Thus we have

$$\begin{aligned} \text{charge} &= \text{cost} + \Delta\Phi \\ &= 1 + r'(x) + r'(p) - r(x) - r(p) \\ &= 1 + r'(p) - r(x) && r'(x) = r(p) \\ &\leq 1 + r'(x) - r(x) && r'(p) \leq r'(x) \text{ since p child or x now} \\ &\leq 1 + 3(r'(x) - r(x)) && r'(x) - r(x) \geq 0 \text{ since x has at least the same subtree heights} \end{aligned}$$

where cost = 1 since we do 1 rotation.

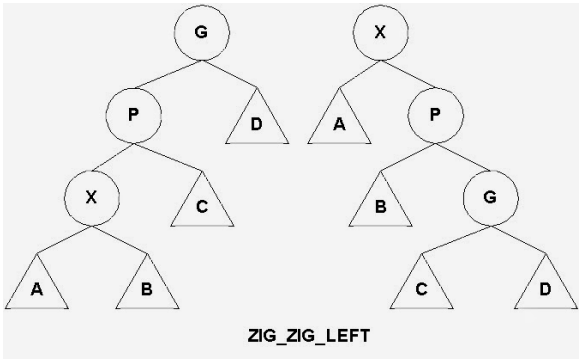


Figure 4.5: Zig-zig (case 1) from before.

Zig-zig Notice that $r'(x) = r(g)$ (same argument as before). Furthermore, $D(x) + D'(z) \leq D'(x)$.

Thus we have

$$\begin{aligned}
\text{charge} &= \text{cost } + \Delta\Phi \\
&= 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\
&= 2 + r'(p) + r'(g) - r(x) - r(p) && r'(x) = r(g) \\
&\leq 2 + r'(x) + r'(g) - r(x) - r(p) && r'(p) \leq r'(x) \\
&\leq 2 + r'(x) + r'(g) - 2r(x) && -r(p) \leq -r(x)
\end{aligned}$$

where cost $= 2$ since we do 2 rotations.

If we show $2 + r'(x) + r'(g) - 2r(x) \leq 3(r'(x) - r(x))$ then we are done, i.e.

$$\begin{aligned}
& 2 + r(x) + r'(g) \leq 2r'(x) \\
\iff & \log D(x) + \log D'(g) \leq 2 \log D'(x) - 2
\end{aligned}$$

Note that $D(x) + D'(g) \leq D'(x)$ (from diagram), thus we essentially need to show

$$\log a + \log b \leq 2 \log c - 2$$

if $a + b \leq c$, which holds (proof left as exercise).

Zig-zag Similary proof as zig-zig.

Therefore we have charge $\leq 3(r'(x) - r(x)) + 1$ for all of zig, zig-zig, and zig-zag.

Splay(x)

Claim. Charge of splay(x) is $O(\log n)$.

If we add up $3(r'(x) - r(x)) + 1$ (charge for each individual zig, zig-zig, or zig-zag) as x goes up the tree, we get a telescoping sum $3(r(\text{root}) - r(\text{original x})) + O(\log n) \leq O(r(\text{root})) + O(\log n) = O(\log n)$.

Search, insert, delete

Theorem 4.1. The amortized cost of search, insert and delete are $O(\log n)$.

Proof. **Search** Note that

$$\text{charge } = \text{charge(splay}(x) + \text{ cost } + \Delta\Phi$$

where cost $+\Delta\Phi$ is the cost of other work (i.e. walking the path from root to x), which is $\leq$ cost of splay ($O(\log n)$), thus charge is $O(\log n)$.

Delete One node disappears, Φ goes down which is okay.

Insert Increase D for all nodes on path root to x.

We can prove this is $O(\log n)$.

□

□

4.3 Optimality conjecture for splay trees (open problem)

Splay trees are (within big Oh) as good as we can get with binary search trees, even by looking ahead at sequence of operations and planning rotations for any binary search tree variant.

5 September 25, 2018

5.1 Union find

Also known as **disjoint sets**: data structure for representing disjoint sets that supports efficient lookup of elements (and which set they belong to) and insertions/merges (between multiple disjoint sets).
Motivation: find all connected components of a graph. Note that depth first search would take $O(n+m)$ where n, m represents the number of node and edges, respectively.
Dynamic graph connectivity: able to maintain connected components as the graph changes. It allows us to answer queries such as "given vertices u, v are they connected"?
Some application examples include:

1. Social networks: relationships added/deleted
2. Minimum spanning tree: recall Kruskal's algorithm involves ordering the edges by weight $e_1, \ldots, e_m$ where we add e_i to our collection of components T iff e_i joins two different components.

   This is a special case of **incremental dynamic connectivity**: we add edges but don't delete any.

Supports two operations:

1. `Union(A,B)` - unite two sets A and B (destroys A, B)
2. `Find(e)` - which set contains element e

Analysis of Kruskal's using union find:

$$\text{sort} + 2m\text{Finds} + n\text{Unions}$$

where sort takes $O(m \log m) = O(m \log n)$. We want the `Find`s and `Union`s to take $\leq O(m \log n)$.
In this analysis, we care about the sequence of `Union` and `Find` so amortized analysis is relevant.
From herein, let n denote the number of elements and m the number of operations where the # of unions $\leq n-1$.
There are multiple ways to implement union find.
Using an **array** $S[1 \ldots n]$ where $S[i]$ is the name of set containing element i. This means that `Find` is $O(1)$ but `Union` has worst case $O(n)$ (since we need to iterate through array and update all elements to new set name).
Tiny improvement: for `Union(A,B)` we update $S[i]$ for i in the smaller set of A, B. Thus if

$$A : 1, 3$$
$$B : 2, 7, 6, 5$$
$$C : 4$$

and we perform a union between A and B we only update $S[1], S[3]$ to B.
Note that the cost of all possible unions for n elements is $\leq O(n \log n)$ since each element changes its set $\leq \log n$ time (tree with leaves as each individual element; height is number of times an element changes set). Thus the cost of m operations is $O(m + n \log n)$ if # of finds is $\Omega(n \log n)$.
Thus for Kruskal's we have $O((m+n) \log n)$.
Abstractly, union find can be represented as a forest of n-ary trees where each tree is a disjoint set. The root of the tree is the "representative member". Second method: we use **pointers** to construct our forest of elements. Let *rank* of a node i be the length of the longest path from any element to i.

Find Walk up the tree from element e to get set name from root.

Union On union, add pointer from root of "smaller" tree to root of "larger" tree.

We can introduce an optimization: **path compression**. On `Find`, update pointer of every element in the path to point directly to the root. Note we could have done this during `Union`, but this is pre-emptive since we may not perform many `Find`s after. While this doubles the work of `Find`, we have the same $O(\cdot)$.
How do we define "smaller" and "larger"?
We can keep track of a tree's rank r where $r(\text{single node}) = 0$ and the union of two trees T_1 and T_2 with rank $r_1 \geq r_2$ would create a tree of rank $r = \max\{r_1, r_2 + 1\}$. Without path compression rank is equivalent to tree height.

Exercise 5.1. Show that an element of rank r has $\geq 2^r$ descendants ($r \geq 1$).
Proof by induction.

Analysis is a bit harder:

Theorem 5.1. (Tarjan 1975). The cost of m operations on the above union find data struture is $O(m \cdot \alpha(m, n))$ i.e. the amortized cost is $O(\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function, which is ≤ 5 for all practical purposes, so we have effectively $O(1)$ amortized cost.
This bound is tight (infinite examples where algorithm takes this runtime).

There is an easier bound to prove with path compression using a charging scheme with $O(m \log^* n)$. Note that $\log^* n$ is defined as

$$\log^* n = \min_i \{\log(\log(\ldots \log n)) \leq 1\}$$

where $\log^* n = i$ is the number of logs required such that the above expression is ≤ 1. How quickly does $\log^* n$ grow?
Note that the tower function is defined as $2 \uparrow n = 2^{2^{2^{\cdots}}}$. Thus we have $\log^*(2 \uparrow n) = n$, and note that

n	0	1	2	3	4	5
$2 \uparrow n$	1	2	4	16	65536	big number

So this bound is very good.
Note the cost of `Find(e)` is the distance from e to the root. We charge some of this cost to `Find` and some to the nodes along the path.

Claim. We claim $rank(e) < rank(parent(e))$.

Claim. The # of vertices of rank r is $\leq \frac{n}{2^r}$.

Proof. Using our previous claim that rank r has $\geq 2^r$ descendants and that vertices of rank r have disjoint descendants. □

Proof of runtime: for a given vertex of rank r, assign to group $\log^* r$. The number of groups is $\log^* n$. Note that group g contains ranks $2 \uparrow (g-1) + 1, 2 \uparrow (g-1) + 2, \ldots, 2 \uparrow g$ which has $\leq 2 \uparrow g$ ranks.
Then for `Find(e)`, for each vertex u on path from e to root, if u has parent and grandparent and $group(u) = group(parent(u))$, then charge 1 to u. Otherwise charge 1 to `Find(e)`.
Note that this covers the cost of the actual operation itself `Find(e)` (we've allocated enough charge).
The total times we charge to `Find(e)` (instead of a vertex u) is $\leq \log^* n + 1$ since group changes at most $\log^* n - 1$ times (and we add 2 for the root and child of root).
The total charge to each vertex u in group g: if u is charged then path compression will give it a new parent of higher rank than the old parent by claim 1.
So u in group g is charged

$$c(g) = (\#\text{ of ranks in group g}) - 1$$

times before it acquires a parent in a higher group and after then it is not charged, thus $c(g) \leq 2 \uparrow g$.

Total charge of all vertices in group g is $c(g) \cdot N(g)$ where $N(g)$ is the # of vertices in group g. Note that

$$\begin{aligned} N(g) &\leq \sum_{r=2\uparrow(g-1)+1}^{2\uparrow g} \frac{n}{2^r} \\ &\leq \frac{n}{2^{2\uparrow(g-1)+1}} \sum_{0}^{\infty} \frac{1}{2^i} \\ &= \frac{n}{2 \uparrow g} \end{aligned}$$

Thus $c(g) \cdot N(g) \leq n$.
Thus the charge to all vertices is $n \cdot \log^* n$ where n is the charge to 1 group and $\log^* n$ is the # of groups, thus we have the total charge for m `Find`s that are allocated to `Find`s and vertices

$$O(m(\log^* n + 1) + n \log^* n) = O(m \log^* n)$$

6 October 1, 2018

6.1 Geometric data

There are multiple problems associated with geometric data (i.e. multi-dimensional data in $\mathbb{R}^n$):

Range searching Given points in space, query a region R to find points in R.

In 2D, given a set of points pre-process them to handle range query for rectangle R.

Let us denote 3 measures:

1. P - preprocessing time S - space Q - query time ($\geq$ output size)

We may also consider a measure U for the update time to add/delete points.

Note that in $\mathbb{R}^1$, to find points in a given interval $[x_1, x_2]$, we can sort the points and find all points in between via binary searching for x_1, x_2. Thus we have

$$\begin{aligned} P &= O(n \log n) \\ S &= O(n) \\ Q &= O(\log n + t) \end{aligned}$$

is the output size.

To handle updates, we could instead use a balanced binary search tree, where P, S, Q remain the same. Update time $U = O(\log n)$.

In $\mathbb{R}^2$: in the static case (no updates) we have quad trees, kd trees, and range trees.

Quad trees Divide square into 4 subsquares recursively until each square has 0 or 1 points

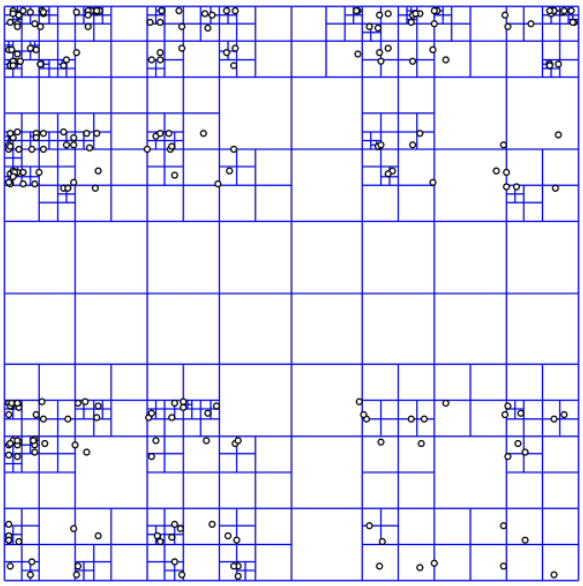


Figure 6.1: Example of quad tree partitioning space into quadrants/subsquares.

For our runtimes/space we have

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = \theta(\sqrt{n} + t)$$

(intuition for $\sqrt{n}$: it is equivalent to $2^{\log \sqrt{n}} = 2^{\log n/2}$ where we may need to check up to $\log n/2$ levels of nodes, and our branch factor is 2).

kd trees Alternately divide points in half vertically and horizontally.

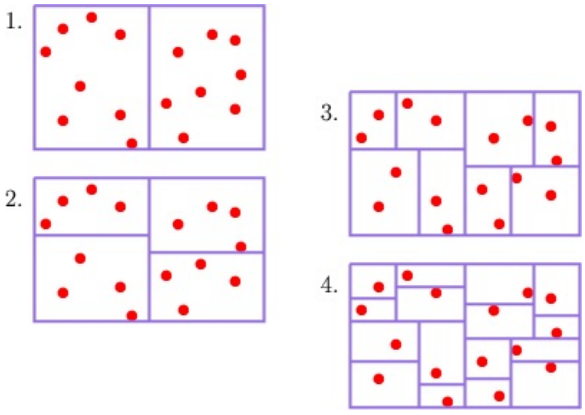


Figure 6.2: Example of kd tree dividing in half the points vertically first, then horizontally, then vertically, and finally horizontally.

We can first sort all points (each by both dimensions) and find our median/mid-point dividing lines for

each iteration. We then construct an binary search tree of our dividing lines. Thus we have

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = \theta(\sqrt{n} + t)$$

To do the actual query, we check if our rectangle endpoints if they belong in either side of each split-point. We then recurse into the side(s) that our rectangle is contained in.

Note that our query time with $\sqrt{n}$ is much worse than $\log n$.

Range trees Improve Q at the expense of S.

We construct a balanced BST on x-coordinates where the **leaves** are the points sorted by x-coordinates.

For a given internal node v, its descendants $D(v)$ is associated with a **slab**: that is we store at v a list $A(v)$ of points in $D(v)$ sorted by their y-coordinates.

Note an upper bound on space is $O(n^2)$: each internal node may store up to n nodes and we have $\frac{n}{2}$ internal nodes. However, a tighter bound is $O(n \log n)$ where we notice each leaf node can be a part of at most $O(\log n)$ ancestor nodes.

For processing: we first sort by x-coordinates. We maintain a y-coordinate sorted list as well. For each internal node we can simply extract the corresponding nodes in the slab from the sorted y-coordinate list. Thus $P = O(n \log n)$.

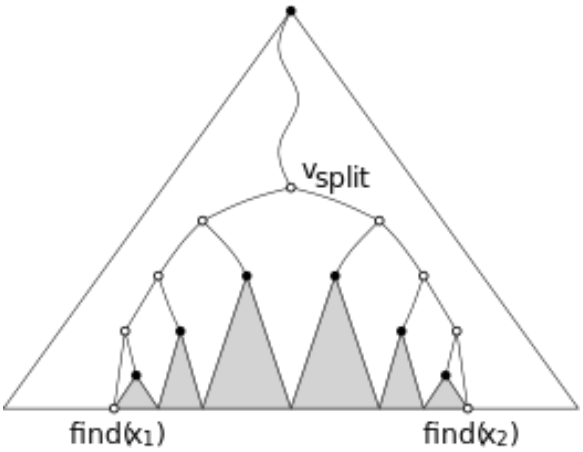


Figure 6.3: Example of a range tree where we are querying for points in between x_1 and x_2.

For searching: we search for x_1, x_2. We want the subsets of leaves between, which we can then filter by y-coordinate.

To find the leaves in between, we look at the internal nodes z, which are the **right children** of nodes on search path to x_1 and the **left children** of nodes on search path to x_2 (after paths split). Thus z corresponds to slabs with union $[x_1, x_2]$.

Remark 6.1. Why couldn't we just use $A(v_{split})$, where v_{split} is the common ancestor of x_1, x_2? Note that while v_{split} is the common ancestor, x_1 may actually be in the right children of a node in the left path further down (and similarly x_2 in the left children), so we don't want to include nodes outside this range (see figure).

For each slab z in $[x_1, x_2]$, we perform binary search on $A(z)$ to get points between $[y_1, y_2]$. Note that we have query time $O(\log n + t)$ per slab thus we have total query time $Q = O(\log^2 n + t)$ where t is the

output size (we have $O(\log n)$ slabs and since slabs are disjoint and each output is counted only once). How can we reduce to $Q = O(\log n + t)$? We can save work on repeated searches for y_1, y_2 using **fractional cascading**. For a given node z and child node w (of which we have sorted lists $A(z)$ and $A(w)$ by y-coordinate), we keep ponters from each element in $A(z)$ to the same (or next higher) element in $A(w)$. We perform binary search on the parent $A(z)$, then when we search in $A(w)$ we continue searching from the pointers we left off at. We thus only binary search on at most n elements so we have $O(\log n)$.

Point location Query a point p to find which region contains p.

The plane is generally divided into disjoint regions and we want to query for a point p and its region.

Remark 6.2. A special case of these regions are the regions generated by the "closest to center": given several points of interest (e.g. Tim Horton's locations) and a query point p, which is the closest point of interest?

We can split the plane at the midpoint between two or more arbitrary POI which ends up creating a plane of disjoint regions: this is the **Voronoi diagram**.

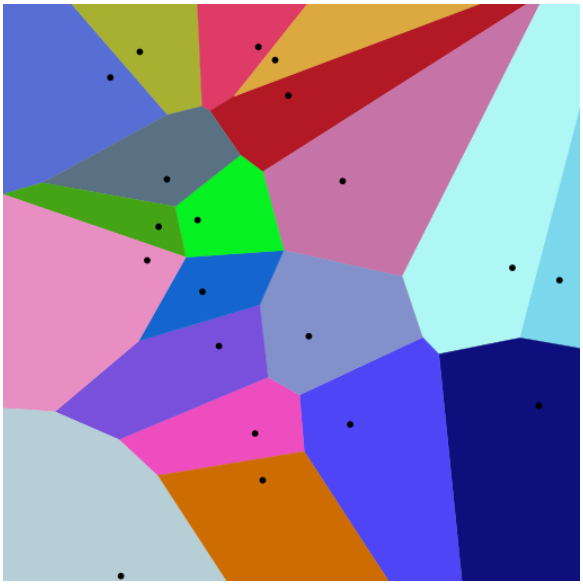


Figure 6.4: Voronoi diagram where each shaded region is mapped to a POI.

In $\mathbb{R}^1$, we keep track of the endpoints of the regions and binary search on our query point.

In $\mathbb{R}^2$, we divide our disjoint regions into slabs with vertical lines at every vertex of the regions. Let n denote the number of resulting slabs.

To search for a query point, we find the slab containing the x-coordinate $O(\log n)$. We then do a binary search inside the slab for the region containing the y-coordinate (this works for arbitrary line segments for a fixed x for our point since we can solve for $y = mx + b$), which is also $O(\log n)$. Thus we have in total $Q = O(\log n)$.

Exercise 6.1. Find a solution where we have space $S = O(n^2)$.

Hint: we have up to $n + 1$ vertical slabs. In each slab, we have up to $n + 1$ dividing line segments, thus we require $O(n^2)$ space.

Improvement to space: changes from one slab to the next are few (at each boundary vertex, some line segments end and some begin per each slab, while some remain the same).

We can thus think of our slabs as a **sequence of binary search trees** that have few changes in between them.

We can initially construct search tree containing line segments in the leftmost slab, then sweep left to right to introduce/take away line segments in each subsequent slab. A *persistent* tree will let us use the same sub-tree from previous steps instead of constructing an entirely new tree. We thus end up storing at most as many elements as there are line segments or $O(n)$ space.

Remark 6.3. Data structures that can be updated and queried in the past are called **persistent data strucutres** (e.g. querying for Facebook friends in the past).

Partial persistence only allows updates in the present while **full persistence** allows updates to the past.

We require partial persistence for range trees.

With the improvement, we end up with

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = O(\log n)$$

7 October 3, 2018

7.1 Randomized algorithms

Randomized alorithms are algorithms that use *random numbers*. The output and/or run time depend on random numbers. We must do **expected case analysis** for analyzing run time.
Advantages of randomized algorithms:

1. practical: faster/simpler algorithms in general
2. theoretical: can we even prove randomness helps? e.g. can randomness give poly-time for NP-hard problems? The evidence is slight.

Examples of randomized algorithms include hashing, quicksort, and quickselect.

Example 7.1 (Quicksort)**.** Let

```
1   Input: S = {s_1, ..., s_n}
2
3   if n = 0,1 return S
4   else
5     i = random[1..n]              // s_i is our pivot
6     L = {s_j : s_j < s_i}         // size l
7     M = {s_j : s_j = s_i}         // size m
8     B = {s_j : s_j > s_i}
9
10    return (Quicksort(L), M, Quicksort(B))
```

The worst case run time is $O(n^2)$ i.e. when $l = n - 1$ (and $n_i - 1$ on subsequent iterations i).
Intuition: we "expect" s_i to be in the middle hence

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

thus we have expected case $O(n \log n)$.

Remark 7.1. There is a slight difference between *average* and *expected* case:

average case analysis No random numbers, assume all inputs equally likely

expected case analysis Algorithms use random numbers, NO assumption on input

More formally, we have the model that includes the random number generation

$$x = rand[1, \ldots, n]$$
$$x = rand[0, 1]$$

which both have $O(1)$ cost each.
Some terminology:

sample space all possible "runs" of algorithms for fixed input

random variable maps sample space to run time (integer)

expected value Expectation of random variable X where

$$E[X] = \sum_x xP(X = x)$$

Example 7.2. Biased coin where $P(H) = \frac{1}{3}$. The expected number of coin tosses to get a head is:

$$\sum iP(\text{i tosses}) = 1 \cdot \frac{1}{3} + 2 \cdot \frac{2}{3}\frac{1}{3} + 3 \cdot \left(\frac{2}{3}\right)^2\frac{1}{3} + \ldots$$

Properties of expected values include:

$E(X+Y) = E(X) + E(Y)$	linearity
$E(cX) = cE(X)$	constant multiplication
$E(XY) = E(X)E(Y)$	X and Y are independent $P(X = x, Y = y) = P(X = x)P(Y = y)$
$E(X) < E(Y)$	if $X < Y$
$\max\{E(X), E(Y)\} \leq E(\max\{X, Y\})$	

The run time depends on the input *and* random numbers. For a randomized algorithm, our run time can be represented as $T(I, R)$ where I is a fixed input and R is a sequence of results of $rand[\ldots]$. We eventually want T as a function of the input size n.
Thus the **worse case** run time is the *max* over all inputs I where $|I| = n$, and the **expected case** run time is the *average* over all random operations R.
The expected case is formally $E(T(I, R)) = \sum_R P(R)T(I, R)$ and the worst case can be expressed as

$$T(n) = \max_{|I|=n} E(T(I, R)) \leq E(\max_{|I|=n} T(I, R))$$

Example 7.3. We can perform analysis on quicksort without using recurrences using expected case analysis.
We want to find $E(X)$ where X is the # of comparisons and $X(u, v)$ is the number of comparison between u and v.

Thus

$$\begin{aligned} E(X) &= E(\sum_{u,v \in S} X(u,v)) \\ &= \sum_{u,v \in S} E(X(u,v)) \end{aligned}$$

Note that for any pair u, v, we compare them either 0 or 1 times since given parts L, M, B as above:

- u, v are initially in the same part
- they are in different parts after partitioning
- they are never compared after they go in different parts

For $E(X(u,v))$ we can look at the step where u, v are separated. It is obvious that we only compare u, v if the pivot is either u or v. WLOG if $u < v$ and we sort all our number such that u has rank r and v has rank $r + k$ (rank is the order of an element when sorted):

$$\begin{aligned} E(X(u,v)) &= 1 \cdot P(\text{compare u,v}) + 0 \cdot P(\text{no comparison}) \\ &= \frac{2}{k+1} \end{aligned}$$

where $k + 1$ is the number of choices we have when u is separated from v because we chose something between u, v, inclusively. Thus we have

$$\begin{aligned} E(X) &= \sum_{r=1}^{n-1} \sum_{k=1}^{n-r} \frac{2}{k+1} \\ &\leq 2 \sum_{r=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\ &\leq 2 \sum_{r=1}^{n} O(\log n) && \text{harmonic series} \\ &= O(n \log n) \end{aligned}$$

7.2 Selection and Quickselect

The selection problem is as follows: given $S = \{s_1, \ldots, s_n\}$ numbers and $k \in [1, \ldots, n]$, return s_i of rank k i.e. $k = 1$ (min), $k = n$ (max), $k = \lfloor \frac{n}{2} \rfloor$ (median).
The Quickselect algorithm is as follows

```
1  if n <= constant
2    sort and return kth item
3  else
4    i = rand[1, ..., n]            // s_i pivot
5    partition S into
6      L: smaller than s_i          // size l
7      M: equal to s_i              // size m
8      B: bigger than s_i           // size b
9    recurse on appropriate set
```

The worst case is $O(n^2)$, but expected case is $O(n)$ using recurrence relations.
Open question: can we do expected case analysis for quickselect like we did for quicksort?
History for selection problem:

1960: Hoare Quickselect has $E(\#\text{ of comparison}) = 3n + o(n)$.

1973: Blum A *non-randomized* algorithm with expected case $O(n)$ where $E(\#\text{ of comparison}) = 5.43n + o(n)$.

1975: Floyd-Rivest Another randomized algorithm with $E(\#\text{ of comparison}) = 1.5n + o(n)$.

1985 Proved lower bound is $2n$ for deterministic/non-randomized algorithm.

1989: Munro & Cunto Proved *any* randomized algorithm has lower bound $E(\#\text{ of comparison}) \geq 1.5n + o(n)$, so Rivest's algorithm is tight.

1999: current determinisitic upper bound Deterministic algorithm with $2.95n$ comparisons

2001: current deterministic lower bound Proved lower bound is $(2 + \epsilon)n$ where $\epsilon = 2^{-80}$.

Conclusion: randomness provably helps.

7.3 Lower bound on median selection

Theorem 7.1. Proposed by Blum et al. in 1975, finding the median ($k = \lfloor \frac{n}{2} \rfloor$) requires $\geq 1.5n$ comparisons in worst case.

Proof. The proof uses an **adversarial argument**: i.e. what is the worst possible case for our algorithm?
Let m be the median, L be elements $< m$ and H be elements $> m$, each with $\frac{n-1}{2}$ elements.

Claim. We claim the number of comparisons between elements within L (#LL) and elements within H (#HH) is $\geq n - 1$ i.e. $\#LL + \#HH \geq n - 1$.
Note each element in L must "lose" a comparison (i.e. $<$) to an element in L or m itself.
Similarly each element in H must "win" a comparison (i.e. $>$) to an element in H or m itself.
This forms a tree of comparisons (each edge is a comparison) with at least $n - 1$ edges.

Claim. The worst case number of comparisons between elements in L and in H (#LH) is $\geq \frac{n-1}{2}$.
As the algorithm is computing comparisons (i.e. whether $s_i < s_j$), the adversary would construe/manufacture the worst possible case for these comparisons. The adversary maliciously tries to maximize the # of comparisons required by putting elements in L and H such that the number of #LH comparisons is maximized.
The adversary algorithm is as follows

```
1  on comparison x,y:
2    if x and y are set (in L/H)
3      continue
4    if x is set, y not set
5      if x in L, put y in H
6      if x in H, put y in L
7    if x,y are unset
8      put one in L, one in H
```

But we still require $\frac{n-1}{2}$ elements in each of L and H, thus the adversary stops when either $|L|$ or $|H|$ is $\geq \frac{n-1}{2}$.
Thus the adversary forces at least $\frac{n-1}{2}$ comparisons.

□

8 October 12, 2018

8.1 Las Vegas vs Monte Carlo

There are a few distinctions between randomized algorithms:

Las Vegas algorithm Always produces the correct output, regardless of random numbers generated. Expected polynomial runtime.

Quicksort is one such example.

Monte Carlo algorithm Produces the correct output with high probability (that can be bounded by number of trials). The runtime should always be polynomial.

How are these related?

Las Vegas to Monte Carlo Stopping algorithm after some time and outputting junk.

Monte Carlo to Las Vegas Given a correctness test (with good run time), we test the output of the Monte Carlo and if incorrect, repeat until we get the correct answer.

8.2 Primality test

Given an odd number n, is n composite (i.e. not prime)?
We phrase it this way so we have a problem in NP - verify YES answers with "proof" i.e. the factors of the composite number.
Note that if the input is n, the input size is $\log n$ (# of bits), so trial division up to $\sqrt{n}$ takes $O(\sqrt{n})$ which is **not** polynomial time wrt to the input size.
There does exists a polynomial time (non-randomized) algorithm to test primality (Agrawal, Kayal, and Saxena, 2002: AKS primality test).

Theorem 8.1 (Fermat's Little Theorem)**.** If p is prime then $a^{p-1} \equiv 1 \mod p$ for all $0 < a < p$.

Remember that the contrapositive states that $A \Rightarrow B$ is equivalent to $\neg B \Rightarrow \neg A$, thus FLT restated says that if there exists $0 < a < n$ and $a^{n-1} \not\equiv 1$ then n is composite. We call such an a a **Fermat witness** to n's compositeness.
Idea: to test if n is composite:

- Generate random a in $[1, \ldots, n-1]$
- Test if a is a Fermat witness (this can be done efficiently)
- If it is, output YES n is a composite
- otherwise, MAYBE n is prime

For this to work (efficiently), we require that if n is composite then there are many Fermat witnesses.
However, there are composite numbers with *no Fermat witnesses*: the **Carmichael numbers** e.g. 561, 1105, 1729, etc.
We thus need **strong witnesses**.

Definition 8.1 (Strong witness)**.** Let $n - 1 = 2^t \cdot u$ (n is even) where u is odd. Then $a \in [1, \ldots, n-1]$ is a **strong witness** if for some $0 \leq i < t$ we have $k = 2^i \cdot u$ and

$$a^k \not\equiv +1, -1 \mod n$$
$$a^{2k} \equiv 1 \mod n$$

That is a^k is a non-trivial square root of $1 \mod n$. Note that $-1 \equiv n - 1 \mod n$.

Theorem 8.2. If n is prime then there are no strong witnesses.
Idea: integers mod prime p form a finite field in which 1 has exactly two square roots, 1 and -1.

Theorem 8.3. If n is composite then there are $\geq \frac{n-1}{2}$ strong witnesses.
That is the probability that $a \in [1, \ldots, n-1]$ is a strong witness is $\geq \frac{1}{2}$.

Refer to CLRS for proofs of the two theorems above.
We thus define the pseudocode for our procedure `witness(a,n)` that tests if a is a strong witness of n

```
1   witness(a,n):
2     compute t,u where n - 1 = 2^t u, u odd
3     x_0 = a^u mod n
4
5     for i = 1...t
6       x_i = (x_{i-1})^2 mod n
7       if x_i = 1 and x_i-1 != 1 and x_{i-1} != n-1
8         return TRUE                                  // a is a strong witness
9     if x_t != 1 return TRUE                          // a is a Fermat witness
10
11    return FALSE
```

The runtime of `witness` is polynomial in $\log n$ since $t \leq \log n$ so the loop is executed $\log n$ times and squaring in mod n takes $\log n$ time.
Thus we have the **Miller-Rabin algorithm** which applies `witness` in a Monte Carlo approach

```
1   repeat s times
2     x = rand[1, ..., n-1]
3     if witness(x,n) then return YES: n is composite
4
5   return NO n is not composite                       // this is really a "MAYBE"
```

The runtime is therefore $O(s \log^k n)$ which is polynomial in $\log n$.
If n is prime then the algorithm is *always correct*.
If n is composite then

$$P(\text{alg outputs NO}) \leq P(\bigcap_{j=1}^{s} \text{ at trial j x is not a strong witness}) \leq \frac{1}{2^s}$$

This is a Monte Carlo algorithm with a **one-sided error** where:

- If the algorithm outputs YES (n is composite) it is correct
- If the algorithm outputs NO (n is prime) the probability of error is $\leq \frac{1}{2^s}$

8.3 Complexity classes

Recall we have the P and NP classes

P decision problems solvable in polynomial time

NP non-deterministic polynomial time: decision problems where YES answers can be verified in polynomial time given a certificate or proof

co-NP complement of problem is in NP i.e. decision problems where NO answers can be verified in polynomial time given a certificate

Some open questions include $NP = co - NP$, $P = NP$ and $P = NP \cap co - NP$.

Definition 8.2 (RP complexity class)**.** The RP or **randomized polynomial time class for one-sided Monte Carlo algorithms** are decision problems that have a randomized algorithm A running in worst-case polynomial time such that for any input x

$$x \text{ IS YES} \Rightarrow P(A(x) \text{ outputs YES}) \geq \frac{1}{2}$$
$$x \text{ IS NO} \Rightarrow P(A(x) \text{ outputs YES}) = 0$$

Thus YES is always correct and NO is wrong with probability $\leq \frac{1}{2}$.

Similarly, co-RP ("complement") are decision problems with randomized algorithms where NO is always correct and YES is wrong with probability $\leq \frac{1}{2}$.

Definition 8.3 (ZPP complexity class)**.** The ZPP or **zero error probability polynomial time** class are decision problems that have Las Vegas algorithms with expected polynomial run time.

Lemma 8.1. We claim $P \subseteq ZPP \subseteq RP \subseteq NP$.

Proof. $P \subseteq ZPP$ A polynomial time algorithm is a Las Vegas algorithm with no use of randomness.

$ZPP \subseteq RP$ We require

Theorem 8.4 (Markov's inequality)**.** If X is a random variable $X \geq 0$ with $E(X) = \mu$ then $P(X \geq c\mu) \leq \frac{1}{c}$.

Proof. Note that

$$\begin{aligned} \mu = E(X) &\geq \sum_{x \geq c\mu} xP(X = x) \\ &\geq c\mu \sum_{x \geq c\mu} P(X = x) \\ &= c\mu P(X \geq c\mu) \end{aligned}$$

Thus we have $\frac{1}{c} \geq P(X \geq c\mu)$. □

Suppose we had a ZPP decision algorithm A with expected run time $T(n)$ bounded by a polynomial in n. Define a Monte-Carlo algorithm A' as follows:

- on input x of length n, run $A(x)$ for time $2T(n)$
- if $A(x)$ produces YES/NO answer in that time, output it
- else output NO

Then A' runs in polynomial time (always), and

- if A' outputs YES this is correct
- if A' outputs no then

$$\begin{aligned} P(\text{error}) &\leq P(A(x) \text{ takes more than } 2T(n) \text{ time}) \\ &\leq \frac{1}{2} \qquad \text{Markov's inequality} \end{aligned}$$

$RP \subseteq NP$ Suppose we have a decision problem and an RP algorithm A for it.

An execution of A depends on the input x and random numbers y, which we can denote as $A(x, y)$ where $A(x, y)$ runs in time polynomial in $|x|$.

From the definition of RP, if x is a YES input then there is a y with $|y|$ bounded by polynomial in $|x|$ such that $A(x, y)$ outputs YES (in fact many y's).

If x is a NO input then there is no y such that $A(x, y)$ outputs YES.

Thus y acts as the certificate to verify a YES input (by running $A(x, y)$) in polynomial time.

□

It remains an open problem whether the containments are proper.

Lemma 8.2. We claim $ZPP = RP \cap co - RP$.

Proof. From above, $ZPP \subseteq RP$ and similarly $ZPP \subseteq co - RP$, thus $ZPP \subseteq RP \cap co - RP$.
It remains to prove that $RP \cap co - RP \subseteq ZPP$ (assignment 4). □

9 October 15, 2018

9.1 More Monte Carlo primality

Recall from last day: a randomized Monte Carlo algorithm to test if n is prime:

- Polynomial time
- One-sided error: if alg. claims n is composite it must be correct. If alg. claims n is prime, prob(error) $\leq \frac{1}{2}$ (larger fraction would be okay).
- Can improve with repeated trials (such that error is then bounded by $\frac{1}{2^n}$ for n trials).

Note: there is a non-randomized polynomial time algorithm to test primality (from last day).
Follow-up:

Question how do we generate a large random t-bit prime?

Answer generate a random t-bit number and test if it's prime. If not, generate a new number.

For deriving the expected runtime, we need to know the distribution of primes.

Application: RSA cryptosystem.

- Depends on hardness of factoring $n = pq$ where p, q are primes
- Factoring: given a number n, find prime factorization
- No known polynomial time non-randomized nor randomized algorithm, not known to be NP-hard
- Decision version: given n, m does n have a (prime) factor $\geq m$?

  Not known to be in NP-complete, but it is in NP.

9.2 Fingerprinting

Example 9.1. Suppose we wanted to test equality of strings and it is too expensive to send/compare the entire string (e.g. two databases in different locations).

Solution. Send/compare a smaller "fingerprint".

Let x be an n-bit binary number ($< 2^n$).
Compute $H_p(x) \equiv x \mod p$ where p is a prime chosen at random in $1, \ldots, M$ (we choose M), thus $H_p(x)$ has size $\log M$.
Note that if $x = y$ for some y, then $H_p(x) = H_p(y)$ must be true.
But we can have $x \neq y$ but $H_p(x) = H_p(y)$ which happens if $|x - y|$ is divisible by p, which is our "failure". What is error bound on prob(failure)?
Need two results from number theory:

1. Prime Number Theorem: Let $\pi(N)$ denote the # of primes $< N$.
   Then $\pi(N) \sim \frac{N}{\ln N}$.
2. # of primes dividing $A < 2^n$ is $\pi(n)$.

Thus the error rate is

$$\begin{aligned} prob(failure) &\leq \frac{\text{\# of primes } p < M, \text{ p divides } |x - y < 2^n|}{\text{\# of primes } < M} \\ &\sim \frac{\pi(n)}{\pi(M)} \end{aligned}$$

If we choose $M = n^2$, then

$$prob(failure) = \frac{n}{\ln n} \cdot \frac{\ln n^2}{n^2} = \frac{2}{n}$$

So by comparing fingerprints of length $O(\log n)$ (since we chose $M = n^2$, our resulting fingerprints $< n^2$ thus will have $O(\log n^2) = O(\log n)$ length), we get a good randomized test with prob(error) $= \frac{2}{n}$ (one-sided error).
Note by choosing p at random every time, we generate good behaviour for *all* x, y (as opposed to fixing p and have it fail with certainty for some x, y).

9.3 Verifying polynomial identities

Example 9.2. The **Vandermonde matrix** is given by

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix}$$

Theorem 9.1.

$$\det(M) = \prod_{i,j \quad j<i} (x_i - x_j)$$

We could verify this theorem by plugging in arbitrary values for $x_1, \ldots, x_n$ and computing $\det(M)$ as fast as matrix multiplication (obvious runtime is $O(n^3)$ but in reality $O(n^\omega)$ where ω is the matrix multiplication constant, currently 2.373, a slight improvement to the former best **Coppersmith-Winograd** algorithm).

Can compute $\det(M) \mod p$ for some prime p to do comparison(?)

In the above Vandermonde example, there was a theorem, but in general this arbitrary testing has other applications such as in *symbolic math* and *automatic theorem proving.*
General problem: given multivariate polynomial, test if $f(x_1, \ldots, x_n) \equiv 0$ (identically 0 i.e. all coefficients of terms are 0; we could multiply out the polynomial and check the coefficients but this is exponential time).

Example 9.3. The polynomial $x_1x_2^3 + x_3^2 + x_1x_2$ is clearly not identically 0.

Definition 9.1. The **degree of a term** $x_1^{i_1}x_2^{i_2}\ldots x_n^{i_n}$ is $\sum_{j=1}^{n} i_j$.
The **degree of a polynomial** is the max over all its terms.

We can thus use a randomized algorithm that plug in random values $x_1, \ldots, x_n$ to test if it's 0. To calculate the error probability, we will require the following theorem:

Theorem 9.2 (Schwartz-Zippel)**.** Let $f(x_1, \ldots, x_n)$ be a multi-variate polynomial of total degree d, $f \not\equiv 0$ (f not identically 0).
If we choose values $a_1, \ldots, a_n$ for $x_1, \ldots, x_n$ independently and uniformly from finite set $S \subseteq \mathbb{F}$ then

$$prob(f(a_1, \ldots, a_n) = 0) \leq \frac{d}{|S|}$$

If we choose $S = \{\pm 1, \pm 2, \ldots \pm d\}$ then our probability of error is $\leq \frac{1}{2}$.

Proof. By induction on # of variables n.
Base case when $n = 1$: note that the # of roots of $f(x) \leq d$ where d is our degree, thus we have probability of $\frac{d}{|S|}$ of choosing one of those d roots.
Induction case: we can write

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_1^i f_i(x_2, \ldots, x_n)$$

where f_i has at most degree $d - i$ (we rewrite f as a sum of terms with x_1 for all possible powers of x_1).
Since $f \not\equiv 0$, at least one term for $i > 0$ is $\not\equiv 0$. Take $k = \max i$ over these i's. Then

$$f_k(x_2, \ldots, x_n) \not\equiv 0$$

where f has total degree $\leq d - k$. By induction

$$P(f_k(a_2, \ldots, a_n) = 0) \leq \frac{d-k}{|S|}$$

If $f(a_2, \ldots, a_n) \neq 0$ then $f(x_1, a_2, \ldots, a_n)$ is a degree k polynomial in terms of x_1 (we plug in constants for all other $x_2, \ldots, x_n$).
Thus

$$P(f(a_1, \ldots, a_n) = 0 \mid f_k(a_2, \ldots, a_n) \neq 0) \leq \frac{k}{|S|}$$

from our base case when $d = 1$.
Finally, we use the identity

$$P(A) \leq P(B) + P(A \mid B^c)$$

thus we have

$$\begin{aligned} P(f(a_1,\ldots,a_n)=0) &\leq P(f_k(a_2,\ldots,a_n)=0) + P(f(a_1,\ldots,a_n)=0 \mid f_k(a_2,\ldots,a_n)\neq 0) \\ &\leq \frac{d-k}{|S|} + \frac{k}{|S|} \\ &\leq \frac{d}{|S|} \end{aligned}$$

□

We thus have a Monte Carlo algorithm to test polynomial identities (i.e. identically 0 polynomials) where we have a one-sided error: if algorithm claims NO (not identically 0) then it is correct. If it claims YES (identically zero) then prob(error) can be arbitrarily small (depending on the # of trials and our set S).
Open problem: Testing polynomial identities in polynomial time deterministically (i.e. no randomness).
Applications:

Verifying matrix multiplication We are given matrices A, B and we compute C as the product of the two matrices (e.g. C was computed with fast matrix mulplication that is complicated and prone to implementation error).

We'd like to verify if $C = AB$.

Suppose $A, B \in \mathbb{R}^{n\times n}$ for simplicity. Let $x = (x_1,\ldots,x_n)$ thus we can verify

$$A(Bx) = Cx$$

where we end up with n multivariate polynomials with (up to) n variables of total degree 1. For example

$$Cx = \begin{bmatrix} 5 & 15 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5x_1 + 15x_2 & 2x_1 - x_2 \end{bmatrix}$$

Since $d = 1$, we can pick a small $S = \{0, 1\}$, thus the probability of error is $\leq \frac{d}{|S|} = \frac{1}{2}$.

10 October 17, 2018

10.1 Linear programming

Linear programming is when we have an optimization problem with linear inequalities. Given variables $x_1,\ldots,x_d$ in d-dimensions we may have

$$\max c_1x_1 + c_2x_2 + \ldots + c_dx_d$$

subject to

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1d}x_d \leq b_1 \\ &\vdots \\ &a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nd}x_d \leq b_n \end{aligned}$$

or simply

$$\max c^t x$$

subject to

$$Ax \leq b$$

where $c \in \mathbb{R}^{d\times 1}$, $x \in \mathbb{R}^{d\times 1}$, $A \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}n \times 1$.
TODO convex hull picture
Each constraint $a_1x_1 + a_2x_2 \leq b$ forms a half-space. The intersection of all half-spaces is our feasible region (i.e. region that satifies our constraints which is a convex polyhedron or a convex hull).
Note that we have a few cases:

- Optimum occurs at a **vertex**: the intersection of two (or more) lines of constraints.
- We can have multiple optima.
- We can have an unbounded solution.
- The problem may also be infeasible.

Lemma 10.1. In any dimension d if the feasible region is non-empty and bounded then there is an optimum solution at a vertex.

This lemma implies a finite algorithm where we test all vertices see which gives the optimal value (i.e. maximum or minimum value): that is we test all sets of $\binom{n}{d}$ constraints where we set them to equality to find the vertices.
For each vertex, we test if x is feasible. If it is then we find c^tx and compare all such x.
This takes $O(n^d)$ time. Linear programming (e.g. the Simplex algorithm) attempts to make this more efficient.
Some applications of linear programming include:

- Planning diets: where x_i is the amount and c_i is the cost of food type i. We also need to meet minimum dietary requirements such as
$$a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jd}x_d \geq b_j$$
for various nutrients $j = 1, \ldots, n$.

The Simplex method (Dantzig 1940s) is a more efficient algorithm for solving linear programs (which subsequently spurred the development of computers).
At a high-level It starts at some vertex that is the intersection of two constraints. It chooses and removes one of the equality constraints and adds one new equality constraint, which effectively causes us to "walk along" a constraint edge to find our next vertex.
It however requires a rule (our "pivot rules") for which constraint to remove and which constraint to add.
Simplex is very good in practice but we do not know a pivot rule that guarantees polynomial time.
The run time is related to the diameter (minimum # of edges between two vertices) of our feasible convex polyhedron: intuitively we need to walk along an order of our diameter to reach our optimum solution.
The **Hirsch conjecture** states that the diameter is $\leq n - d$ where n is the number of constraints and d is the number of dimensions. This conjecture was however disproved in 2012.
There does exists polynomial time algorithm for linear programming:

Khachiyan 1980 ellipsoid method

Karmarkan 1984 interior point method

At a high level the algorithms operate on the bit representations of numbers.
Thus there remains the open problem if there exists a polynomial time algorithm that uses only arithmetic steps.
In the 1970s and 1980s, linear programming was applied to many small dimensional problems. In 2D, we can apply linear programming to find the best fit line for some points. In 3D, we can determine whether a cast can be removed from a mold.
An algorithm by Megiddo in 1983 takes $O(n)$ for fixed d: it is actually $O(2^{2^d} \cdot n)$.

We will look at **Seidel's randomized incremental linear programming algorithm**: at a high level, we add half-plane constraints one-by-one and update our (current) optimum solution v: how exactly do we update our v? Note if we add our half-planes in random order then the expected runtime is $O(n)$ (# of constraints).
To update, we add a half-plane h_i. There are multiple cases:

1. If $v \in h_i$ (v is still in the half-space of our half-plane), then no update is necessary.

   TODO picture

2. If $v \notin h_i$, then we claim the optimum solution lies on line l_i of h_i.

   We essentially have a 1-D linear programming problem where we have for example constraints $x_1 \leq 2$, $x_1 \leq 5$ and $1 \leq x_1$ and we find the optimal x that maximizes $c^T x$ of this 1D problem.

The algorithm $LP_2(H)$ where $H = \{h_1, \ldots, h_n\}$ or the set of half-planes is thus as follows:

```
1  LP_2(H):
2    Let h_1 ... h_n be in random order
3    v = point at infinity                              // init to unbounded optimum
4    for i = 1 ... n
5      if v not in h_i:
6        v = LP_1({h_1, ..., h_i-1} intersect l_i})   // l_i is the line at h_i
```

Note the 1D linear programming with i constraints runs in $O(i)$, thus our worst case is $\sum_{i=1}^{n} O(i) = O(n^2)$.
To find our expected run time: the idea is to note that case 1 (where $v \in h_i$) happens often.
Consider the situation after adding h_i: did we ecounter case 1 or case 2? (this technique of analysis is called **backwards analysis**).
After adding h_i we have i half-planes. Note that the probability that the h_i we just added is any particular h is equally likely thus $P(h_i = h) = \frac{1}{i}$.
Case 2 where $v' \notin h_i$ happens if we had to update our v because our h_i made our previous v' infeasible. The probability that case 2 happens is choosing one of the two h's that form our new vertex v i.e. v is determined by two lines l, l' where $P(h_i = l) = P(h_i = l') = \frac{1}{i}$. Thus $P(\text{case two}) \leq \frac{2}{i}$.
The expected work across all our LP_1 is

$$\sum_{i=1}^{n} \frac{2}{i} O(i) = O(n)$$

where $\frac{2}{i}$ is the probability of having to invoke LP_1 and $O(i)$ is the runtime of our LP_1 for i constraints.
In higher dimensions, $\frac{2}{i}$ becomes $\frac{d}{i}$ since d constraints determine a vertex v.
Thus we have the recurrence

$$T_d(n) = T_d(n-1) + \frac{d}{n}(T_{d-1}(n))^n$$

which solves to $T_d(n) = O(d!n)$.

11 October 22, 2018

11.1 SAT

Satisfiability or SAT: Given a logical formula with n variables, m clauses can we assign true/false to variables to satisfy the formula.

3-SAT all clauses have 3 literals (variable or negation). This is NP-complete.

2-SAT 2 literals per clause. There exist polynomial time algorithms.

Some applications of SAT:

- AI
- All problems with quantified Boolean formulas e.g. $\forall x \exists y \forall z F(x, \ldots)$.

  SAT is the case with one existential quantifier for all variables.

Some techniques for solving SAT problems:

- heuristics or "resolution"
- brute force worst case run time with $O(1.5^n)$ (best algorithm known thus far); better than the obvious $O(2^n)$ brute force

Question 11.1. Can we get randomized polynomial time for 3-SAT?
This would give randomized polynomial time for all NP-complete problems.
We will thus show a randomized algorithm that can beat the best known deterministic (non-randomized) algorithm.

A randomized SAT algorithm (Papadimitriou 1991) gives us a one-sided error approach:

```
1   input: Boolean formula E in CNF
2   idea: local improvement (hill climbing)
3
4   start with any T/F assignment A
5   repeat t times:                    // t to be chosen
6     if A satisfies E:
7       return YES
8     pick an unsatisfied clause C
9     randomly pick a literal a in C
10    flip a's value
11  return NO
```

Note that YES is always correct and NO might be an error: a satisfiable formula but we failed to find an assignment A that works. What is this error rate?
Let A^* be a valid (i.e. makes the formula true) assignment that satisfies E. Let i denote the # of variables with the same value in current A and A^*. If i reaches n then $A = A^*$ and algorithm outputs YES.
How does i change? In each iteration, $i = i + 1$ or $i = i - 1$ (it either becomes the right assignment or the wrong assignment).
We analyze the probability of either case for 2-SAT and 3-SAT separately. We require the notion of **random walks**. As a Markov chain: suppose we are currently at state i and we can move to either $i - 1$ and $i + 1$. The probability of moving to $i + 1$ and $i - 1$ is $\frac{1}{2}$ each, and at state 0 the probability of moving to state 1 is 1 (wall at 0).
What is the expected # of steps to n? Let t_i denote the expected number of steps to get from i to n. We can derive recurrence relations in terms of t_i by conditioning on the first step (first step analysis):

$$\begin{aligned} t_n &= 0 \\ t_0 &= 1 + t_1 \\ t_i &= 1 + \frac{1}{2} t_{i-1} + \frac{1}{2} t_{i+1} \end{aligned}$$

Note that from the recurrence we have

$$\begin{aligned} t_i - t_{i+1} &= 2 + t_{i-1} - t_i \\ \Rightarrow d_i &= 2 + d_{i-1} \end{aligned}$$

where $d_i = t_i - t_{i+1}$, and thus $d_0 = t_0 - t_1 = 1$. Solving the recurrent we get $d_i = 1 + 2i$ so plugging this back into $t_i = d_i + t_{i+1}$ and $t_n = 0$ we get

$$\begin{aligned} t_i &= \sum_{j=i}^{n-1} d_j \\ &= \sum_{j=i}^{n-1} (1 + 2j) \\ &= (n - i) + n(n - 1) - i(i - 1) \\ &= n^2 - i^2 \end{aligned}$$

where max of t_i is n^2.
Back to 2-SAT where we have $(x_i \vee x_j)$ clauses. What is the probability our i # of matches increases $i = i + 11$ or decreases $i = i - 1$? Note the algorithm picks an unsatisfied clause $C = (\alpha \vee \beta)$ where in A (our current assignment) $\alpha = F$ and $\beta = F$. In A^*, at least one is T. Suppose $\alpha = T$. We pick to flip α, β to true with probability $\frac{1}{2}$ each. If we pick α then $i = i + 1$ and if we pick β then i goes up or down.
Applying the Markov chain results $E(\text{number of steps to reach } i = n) \leq n^2$.
Note the algorithm might succeed earlier with $A \neq A^*$ but A might still satisfy E. Furthermore, the probability of the $i = i + 1$ case is higher than $\frac{1}{2}$ since the choice of flipping β might improve our i. Therefore our analysis is not tight (upper bound).
What value of t should we pick such that $P(\text{not reaching } i = n \text{after } t \text{ steps}) \leq \frac{1}{2}$.
By Markov's inequality we have $P(X \geq c\mu) \leq \frac{1}{c}$. In our case we have $\mu = n^2$ (our expected value) so we choose $c = 2$ such that $P(t > 2n^2) \leq \frac{1}{2}$. That is we set $t = n^2$ to have a probability of error $\leq \frac{1}{2}$.
The # of steps we take is $O(n^2)$ thus our runtime is $O(n^2 \text{poly}(n, m))$ where $\text{poly}(n, m)$ is the time for testing A, etc.
Note that this is not a breakthrough for 2-SAT: there are better deterministic algorithms.
For 3-SAT, our unsatisfied clause becomes $C = (\alpha \vee \beta \vee \gamma)$. Since our current A does not satisfy C, we have $\alpha = \beta = \gamma$.
Since A^* does satisfy C, at least one of the variables is T. Suppose $\alpha = T$, then $P(\text{alg. flips } \alpha) = \frac{1}{3}$ thus $P(i \text{ increases}) \geq \frac{1}{3}$.
Thus we have a random walk with $p_{i,i+1} = \frac{1}{3}$ and $p_{i,i-1} = \frac{2}{3}$. The expected # of steps to reach n is approx. 2^n, which is as bad as our brute force algorithm.
A different randomized algorithm (Schoning 1999) introduces two new ideas

1. start with random truth-assignment of A
2. increasing # of trials (t) is not helpful: we're likely to get stuck at 0, so we pick a new random assignment A

The algorithm follows

```
repeat s times:
  pick random A
  repeat t = 3n times:
    if A satisfies E:
      return YES
    pick unsatisfiable clause C
      same as Papadimitriou's
return NO
```

Fact: in the inner loop ($t = 3n$ times), we have $P(\text{error}) \leq 1 - \left(\frac{3}{4}\right)^n$. If we set $s = c(\left(\frac{4}{3}\right)^n)$ (our outer loop iteration count), we get an overall probability of error

$$\begin{aligned} P(\text{error}) &\leq \left(1 - \left(\frac{3}{4}\right)^n\right)^{c(4/3)^n} \\ &\leq \frac{1}{e^c} \end{aligned}$$

where the last line follows since $(1 - \frac{1}{a})^a \leq \frac{1}{e}$, which follows from $\frac{1}{e^x} \geq 1 - x$.
So ultimately we can get a probability of error $\leq \frac{1}{2}$ with # of steps $O(\left(\frac{4}{3}\right)^n \cdot n)$ so the runtime is $\approx O(1.3^n \text{poly}(m, n))$ vs. the best known deterministic algorithm with $O(1.465^n)$ runtime.