Contents

Part 1. **Foundations**

Insertion Sort. Maintains the invariant that $A[1:j-1]$ is sorted by shifting elements right until $A[j]$ is in the correct position. Insertion sort is *stable*, i.e. two keys already in sorted order remain in the same order at the end. Running time is $O\left(n^2\right)$.

Selection Sort. Maintains the same invariant as Insertion Sort but does so by going forward and *selecting* the smallest element each time and placing it at the head. Running time is $O\left(n^2\right)$.

Bubble Sort. "Bubble up" pair by pair. Stop when no more "bubblings" are possible. Running time is $O\left(n^2\right)$.

Merge Sort. Divide and conquer approach. Divide the array in half, recurse, combine results by merging, i.e. taking the smallest entry from each piece in turn. Base case is just an array with one element. Running time is $O(n \lg n)$.

Binary search. If an array is already sorted then you can find an element in it faster than $O(n)$ time; you can find it in $O(\lg n)$ time. Search in either the left side of the middle entry or the right side.

Horner's Rule. Given $A = [a_1, \ldots, a_n]$ the coefficients of a polynomial and a value x a faster way to calculate $p(x)$ is

$$p(x) = \sum_{k=1}^{n} a_k x^k = a_1 + x\left(a_2 + x\left(a_3 + \cdots + x\left(a_{n-1} + x a_n\right)\right)\right)$$

i.e. $p_0 = 0,\ p_1 = a_n + x p_0,\ p_2 = a_{n-1} + x p_1,\ \ldots$.

Unweighted simple reservoir sampling. Sample k items from n items $A = [a_1, \ldots, a_n]$ fairly, i.e. uniform random, **without replacement**, draws. Put the first k items into a *reservoir* R then for item $i > k$ draw $j \in \{1, \ldots, i\}$ inclusive. If $i \leq k$ the replace ith item. Running time is $\Theta(n)$.

Unweighted complicated reservoir sampling. Use a Max-Queue: generate a uniform random number for first k items and insert, then thereafter generate uniform random and insert if less than max (also pop max). Running time takes $O(n \lg k)$ because of potentially n `Extract-Max` operations on a k length priority queue.

Weighted reservoir sampling. Use the same technique as for unweighted complicated but let the priority be

$$u = (\texttt{random}())^{1/w}$$

where w is the weight of the element. Same running time.

Online Maximum. Fill a reservoir with n/e candidates and pick the maximum from the reservoir. Then pick the next maximum (if one exists) that's higher; this will be the single selection. This can be further simplified by realizing you don't need to keep the entire reservoir and you can return after the first forthcoming maximum (if one exists). Probability of actually picking the max is $1/e$.

Stable Matching. Gale-Shapley: first each man proposes to the woman he prefers best and each woman accepts provisionally, i.e. accepts a proposal but trades up if a more desirable man proposes. Do this for n rounds (or until there are no more unengaged men). Running time is $O\left(n^2\right)$.

Part 2. Sorting and Order Statistics

Heaps. The invariant for a Max heap is $A[i] \leq A[\lfloor i/2 \rfloor]$. Furthermore each entry has "children": $A[2i]$ is the left child and $A[2i+1]$ is the right child of element $A[i]$.

0.0.1. *Max Heapify.* To re-establish the heap property switch the violator with its largest child and then recurse. Running time is $O(\lg n)$.

0.0.2. *Build Max Heap.* To build a heap from an array notice that the deepest children/leaves are already legal heaps so there's no need to `Max-Heapify` them, and the children start at $\lfloor \text{len}(A)/2 \rfloor$. Running time is $O(n)$.

0.0.3. *Extract Min.* $A[1]$ is the maximum element in the heap. Remove $A[1]$ and replace with the last element in the heap and then re-establish the invariant using Max-Heapify from there. Running time is $O(\lg n)$.

0.0.4. *Heap sort.* You can use `Extract-Min` in the obvious way to sort an array. Running time is $O(n \lg n)$.

0.0.5. *Heap increase key.* This just involves re-establish the Max heap invariant by "percolating" the entry up the array. Running time is $O(\lg n)$.

0.0.6. *Heap insert.* Using `Heap-Increase-Key` we can insert into the heap by inserting an $-\infty$ priority element at the end of the heap and then increasing the key to what we want. Running time is $O(\lg n)$.

Quicksort. Quicksort is experimentally the most efficient sorting algorithm. The randomized version runs in $O(n \lg n)$ but is typically faster. Pick a random pivot, swap it to the end, partition the rest of the array (not including the end) on whether elements are $\leq$ or $>$, recurse, and finally insert the pivot in between the two sorted partitions.

Counting Sort. Given keys in the range $1, \ldots, k$ count the number of keys less than or equal to each key a_i and then places a_i in that position (but do it in reverse in order for the sort to be stable). Note that if there are duplicates you need to subtract from cumulates when some a_i is placed. Running time is $O(n+k)$.

```
Counting-Sort(A)
1  k = max(A)
2  C = (k+1) * [0]
3  # count how many of values from 1 to k there is
4  for i = 1 : len(A):
5      C[A[i]] = C[A[i]] + 1
6  # count how entries in A less or equal to i
7  for i = 1 : k:
8      C[i] = C[A[i]] + 1
9  # now place the items in the correct places
10 B = (k+1) * [None]
11 # go in reverse direction in order for sort to be stable
12 for i = len(A) : 1:
13     # a_i has C[a_i] elements to its left in B
14     B[C[A[i]]] = A[i]
15     # if there are multiples of a_i then the next
16     # should be left of in order for stable
17     C[A[i]] = C[A[i]] - 1
```

Radix Sort. Sort stably least significant to most significant digit. For n numbers in base d where each digit ranges from 1 to k the running time is $\Theta(d(n+k))$ if the stable sort runs in $\Theta(n+k)$.

Bucket Sort. Bucket sort depends on values being uniformly distributed $[0, 1]$. It buckets all the entries into $\lfloor n \cdot A[i] \rfloor$ and then subsorts. Expected run time is $\Theta(n)$.

Order statistics.

0.0.7. *Quickselect.* Any sorting algorithm can be used to compute kth order statistics: simply sort and return the kth element. But using the ideas of `Quicksort` you can get down to expected time $O(n)$: only recurse to one side.

0.0.8. *Quickerselect.* Using *median-of-medians* (divide into groups of 5, sort to find median of group, then recurse to median of medians) in order to guarantee good splits. Then recurse either left or right (instead of both partitions). Running time is $O(n)$ deterministically, instead of just expected.

Part 3. Data Structures

Hash Tables. Hash tables are m length arrays keyed on strings instead of numbers.

0.0.9. *Hash function.* A good hash function according to Knuth is

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

where $A \approx \left(\sqrt{5}-1\right)/2$ and $kA \mod 1$ means the fractional part of kA, i.e. $kA - \lfloor kA \rfloor$.

0.0.10. *Hashing with chaining.* Hashing with chaining is basically Bucket Sort, except with the $\lfloor \rfloor$ replaced by a Hash function and retrieval. If n is the total number of items in the hash table and m is the length of the hash table then on average (give uniform hashing) each list has $\alpha = n/m$ items. Therefore insertion is $\Theta(1)$, and retrieval/deletion is $\Theta(1+\alpha)$.

0.0.11. *Hashing with open addressing.* In hashing with open addressing the buckets are "linearized", i.e. just laid out in the table itself: inserts and searches hash and then traverse forward in the table until they find a spot. Deletion is harder so if deletion is necessary then hashing with chaining should be used. Insertion costs at most $1/(1-\alpha)$ and for $\alpha < 1$ retrieval costs

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Integral to these bounds is that α the load factor stay small. In order for the amortized analysis to workout the hash table should be doubled in size (and entries copied) when the table becomes full but halve it only when the load goes down to below 1/4.

Binary Search Tree. A binary tree is a graph where each vertex has at most two children. A binary search tree is a tree with the further constraint that the key of a parent is greater or equal to any of the keys in its left subtree and less than or equal to any of the keys in its right subtree.

0.0.12. *Inserting into a binary search tree.* Start at the root, if the root value is equal to the key you're inserting then go left, otherwise go right. Once you hit a `None` create a new vertex. Running time is $O(\lg n)$ if the tree is balanced.

0.0.13. *Searching a binary search tree.* Start at the root, if the root value is equal to the key you're searching for then you're done, otherwise if the key is less than the value go left, otherwise go right. Running time is $O(\lg n)$ if the tree is balanced.

0.0.14. *Binary search tree min/max.* The minimum of a binary tree is the left-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

The maximum of a binary tree is the right-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

0.0.15. *Binary search tree predecessor/successor.* The predecessor of a vertex the maximum of a vertex's left subtree. Running time is $O(\lg n)$ if the tree is balanced.

The successor of a vertex the minimum of a vertex's right subtree. Running time is $O(\lg n)$ if the tree is balanced.

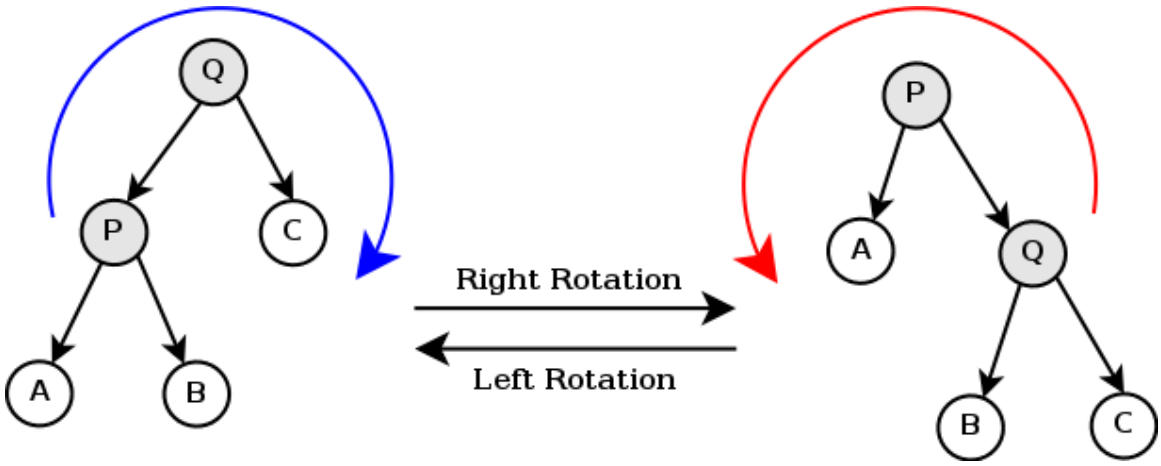
0.0.16. *Deleting from a binary search tree.* Replace with predecessor or successor (be careful about patching up all the pointers and the original site of the replacement). Running time is $O(\lg n)$ if the tree is balanced.

0.0.17. *Pre-order/In-order/Post-order traversal.* Either do the thing before/in between/after recursing into left-child/left-child and right-child/right-child.

Treap. A treap combines the invariants of a binary tree *and* and a heap. There are two sets of attributes: priorities and keys. The priorities obey the heap property (children have smaller priority than their parents) and the keys obey the binary search property. In order to get a balanced binary tree, which is the value of treaps, we randomly generate a priority key.

0.0.18. *Treap search.* Just like for binary search tree and hence omitted.

0.0.19. *Treap insert.* This is easier of the two operations. First we need two auxiliary functions `Left-Rotate` and `Right-Rotate`. The easiest way to remember these is pictures



To insert into a treap, generate a random priority, and insert the key as if it were a binary search tree (i.e. at the bottom), then rotate up until the heap property is restored. Running time is $O(\lg n)$.

0.0.20. *Treap delete.* To delete a vertex rotate it down until it's a leaf node and then delete the leaf node. Rotate down according to which of the vertex's children have a higher priority: if the left child has a higher priority than the right then rotate right (i.e. replace the vertex with its largest child, just like for Heaps), otherwise rotate left.

Cartesian Tree. Given a sequence of **distinct** numbers (or any totally ordered objects), there exists a binary min-heap whose inorder traversal is that sequence. This is known as the Cartesian tree for that sequence. How to construct a Cartesian tree for an arbitrary sequence $A = [a_1, \ldots, a_n]$? Process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value x, start at the node representing the value prior to x in the sequence and follow the path from this node to the root of the tree until finding a value y smaller than x. This node y is the parent of x, and the previous right child of y becomes the new left child of x. Running time is $O(n)$.

Skip Lists. Skip lists are another easy to have $O(\lg n)$ expected time Insert, Search, Delete, and even find by Rank.

The basic idea is to use a **sorted** linked list but to skip ahead (duh) as many entries when performing some operation, e.g. when searching skip ahead 2 entries until you've passed the entry you were looking for. How do implement the ability to skip ahead by 2? Simple: have links not only between consecutive nodes but also every two nodes. The obvious generalization is to also have links every 4 nodes, 8 nodes, etc. Call these nodes which have links to k nodes level-k nodes.

Max level is $\log_2 n$ where n is expected size. Max level is the height of the header.

0.0.21. *Search.* Start at the current highest level of the list at the header. Go right while the next key is greater than the key you're looking for, then go down one level, and go right again. Expected running time is $O(\lg n)$.

0.0.22. *Insertion.* Search for the spot for the entry (all the way from the top this time) while keeping track of the "staircase" you descended down. Generate a random height using $k = 1 - \log_2(\texttt{random}())$ and insert the node, patching up pointers and potentially adjusting the new highest extant level. Expected running time is $O(\lg n)$.

0.0.23. *Deletion.* Deletion works pretty similarly to insert except that when potentially decrementing highest extant level you should start at the current level and decrement until the pointer from the header hits something. Expected running time is $O(\lg n)$.

Interval Trees. An interval tree is built atop your favotire balanced binary tree data structure and stores left endpoints as key. It also keeps track of maximum right endpoint in the subtree rooted at a vertex. It supports interval intersection tests (very useful). Maintaining the max in insertion and deletion is straightforward during rotations.

0.0.24. *Interval search.* Interval search works by being optimistic: $i = [a, b]$ and $j = [x, y]$ two intervals overlap if either $a \leq x \leq b$ or $x \leq a \leq y$. Therefore at each interval we test for overlap and whether $x \leq a \leq y$ where y is the maximum right endpoint for any interval in the left subtree. If so we go left. If in fact $y < a$ then no interval in the left subtree could possibly intersect so we go right. Running time is $O(\lg n)$.

Order Statistics Tree. Order statistics trees are probably the simplest thing to build atop a balanced binary search tree. The only extra extra piece of information each vertex stores is the attribute `size` where $x[\texttt{'size'}] = x[\texttt{'lchild'}][\texttt{'size'}] + x[\texttt{'rchild'}][\texttt{'size'}] + 1$.

0.0.25. *Select.* Finding the ith ordered element in the tree works just like Quickselect. The only trick is that if you recurse into the right child then you're looking for $i - r$ where r is the size of the element where you're currently at (i.e. rank of left child plus 1).

0.0.26. *Rank.* We can find the rank of an element by finding how many elements are to its left. The rank of an element is at least size of its left child plus 1, but the element might be in a right subtree of some other element, therefore we need to count the sizes of the trees to its left. Hence head towards the root: if vertex is right child of parent then there are (rank of parent's left child) + 1 elements to the vertex left, and otherwise (if vertex is left child) then we just keep going.

0.0.27. *Maintenance.* Maintaining `size` is easy: for example in `Left-Rotate` add lines

13 $y\,[\texttt{'size'}] = x\,[\texttt{'size'}]$
14 $x\,[\texttt{'size'}] = x\,[\texttt{'lchild'}]\,[\texttt{'size'}] + x\,[\texttt{'rchild'}]\,[\texttt{'size'}] + 1$

and similarly for `Right-Rotate.`

Union-Find. A union-find data structure is a data structure suited for taking unions and finding members (duh). The particular units of the data structures are sets (not hash table derivatives), each with a representative. The data structure is very thin, basically a wrapper for the primitive data, except for a pointer to the representative of the set and two heuristics that speed up the operations. The path compression heuristic "compresses" the path to representative of the set by setting it to be equal to that representative (which it might not be after a union). The weighted union heuristic makes it so that the smaller of the two sets unioned is the one whose representative pointers need to be updated.

Amortized complexity of n `Make-Set`, `Find-Set, Union,` operations where m are `Make-Set` is $O\left(m\alpha\left(n\right)\right)$ where $\alpha\left(n\right)$ is the Ackermann function and $\alpha\left(n\right) \leq 4$ for any realistic application.

0.0.28. *Make set.* Just wrap the element in a dict.

0.0.29. *Find set.* Recursively look for the set of a set's representative. Also unwinds the stack in order to reset all the representatives in the chain from x to the representative of the set, to the representative of the set.

0.0.30. *Union.* Union by the weighting heuristic: the set with the smaller number of elements should become the representative of the bigger set. If the weights are equal then one of theirs rank should be incremented.

Euler circuit. An Euler circuit visits each vertex in graph twice - once going past it and once coming back across it. How do you print out an Euler circuit of a tree? Use a modified depth first traversal: print before entering the child traversal loop and then print again after recursing into the children.

Tarjan's Least Common Ancestor. The least common ancestor w of two vertices u, v in a tree is the ancestor common to both that's of greatest depth. The algorithm is useful for range-minimum querying. It uses the same traversal as `Euler-Circuit` and the Union-Find data structure augmented with a property `ancestor`. The algorithm proceeds by growing "bottom up" sets corresponding to subtrees whose roots are the least common ancestors of any pair of vertices in the tree **which have been completely traversed by the Euler circuit**. Let P be a global with the set of vertices you're interested in finding least common ancestor of and initialize all vertices to have `color` Blue in order to represent unfinishined (i.e. not completely traversed by the Euler circuit).

```
Least-Common-Ancestor(u)
# u is the root of a tree
u_set = Make-Set(u)
#this is the Euler-Circuit transformation (equivalent of print)
u_set['ancestor'] = u
for v in u['children']:
    Least-Common-Ancestor(v)
    # let's pretend there's a big table where i can fetch v_set from
    Union(u_set, v_set)
    u_set['ancestor'] = u
# u_set['val'] = u
u_set['val']['color'] = Red
for each v such that {u, v} ∈ P:
    if v['color'] == Red:
        print("Least common ancestor of {u, v} is " + v_set['ancestor'])
```

Range Minimum Queries. Given a sequence of distinct values and a subsequence (specified by it's end points) what is the minimum value of the in that subsequences? It's just the least common ancestor of the end points in the cartesian tree representing the sequence.

Part 4. Advanced Design Techniques

Dynamic Programming.

0.0.31. *Fibonacci Sequence.* The simplest dynamic programming algorithm is computing the nth Fibonacci number faster than using the naive recursive definition

$$F_n = F_{n-1} + F_{n-2}$$

Do the computation bottom up by storing F_n, F_{n-1}, F_{n-2}. Running time is $O(n)$.

0.0.32. *Rod Cutting.* Given a rod of length n and a table of price $P = [p_1, \ldots, p_n]$ corresponding to cuts at i units of length what's the maximum value r_n obtained by cutting up the rod? The Bellman equation is (with the $r_0 = 0$)

$$r_i = \max_{j=1,\ldots,i} \{p_j + r_{i-j}\}$$

Running time is $O(n)$.

0.0.33. *Getting to work.* Given a neighborhood of n commuters and n downtown parking lots what is the fastest way for each commuter to get to work given that intersection have delays? The Bellman equation is

$$q(i,j) = \begin{cases} \infty & j < 1 \text{ or } j > n \\ c(i,j) & i = 1 \\ \min\{q(i-1,j-1), q(i-1,j+1)\} + c(i,j) & \text{otherwise} \end{cases}$$

Running time is $O(nk)$.

0.0.34. *Towers of Hanoi.* The solution is purely recursive: let $S(n,h,t)$ be the solution to moving n disks from their "home" rod h to a target rod t. Then

$$S(1,h,t) = \text{just move the disk}$$

and

$$\begin{aligned} S(n,h,t) &= \text{first } S(n-1,h,\text{not}(h,t)) \\ &\quad \text{second } S(1,h,t) \\ &\quad \text{third } S(n-1,\text{not}(h,t),t) \end{aligned}$$

Running time is $O(2^n)$.

0.0.35. *Egg drop.* Suppose you have n eggs, h consecutive floors to be tested, and you drop an egg at floor i in this sequence of h floors. If the egg breaks then the problem reduces to $n-1$ eggs and $i-1$ remaining floors. If the egg doesn't break then the problem reduces to n eggs and $h-i$ remaining floors. The Bellman equation is then

$$W(n,h) = 1 + \min_{i=1,\ldots,h} (\max\{W(n-1,i-1), W(n,h-i)\})$$

If you have only one egg then the minimum number of tests using the best strategy (the one that potentially covers all the floors), if the threshold floor, is the top one is h. So $W(1,h) = h$. If there's only 1 floor we only need 1 egg so $W(n,1) = 1$, and if there are no floors then we need 0 eggs so $W(n,0) = 0$. Running time is $O(nh^2)$ because of the min over $i = 1,\ldots,h$. Since $W(n-1,i-1)$ is increasing in i and $W(n,h-i)$ is decreasing in i a local min of $g(i) = \max\{W(n-1,i-1), W(n,h-i)\}$ is a global min and so you can use binary search so speed the min loop to get a running time of $O(nh\log h)$.

0.0.36. *Maximum Positive Subarray/Kidane's algorithm.* Given $A = [a_1,\ldots,a_n]$, how to find the subarray with the maximum positive sum? Change the problem to look at maximum sum subarray ending at some j. Maximum sum subarray ending at j is either empty, i.e. has negative sum, in which case its sum is 0, or includes $A[j]$. The maximum sum subarray in all of A is the maximum of all subarrays ending at all j. Running time is $\Theta(n)$.

0.0.37. *Longest increasing subsequence.* A subsequence of a sequence $A = [a_1, a_2, \ldots, a_n]$ need not be contiguous. Just like in Kidane's algorithm you should be looking at subsequences ending at some index i: let $L[i]$ be the longest strictly increasing subsequence ending at index i. What's the "optimal" way to obtain $L[i]$? Extend some smaller optimal subsequence ending at index j. But when can you extend some subsequence $L[j]$ ending at position j? Only when $A[j] < A[i]$ since it should be an increasing subsequence! Running time is $O(n^2)$.

0.0.38. *Box stacking.* You have n boxes $B = [b_1, \ldots, b_n]$ with dimensions height h_i, width w_i, and depth d_i. What's the tallest stack of boxes you can make? A box b_i can be stacked atop another box b_j if b_i can oriented such that one of its faces is smaller than the upwarding face of b_j. To simplify the problem simply "replicate" the boxes such that one box with dimensions h_i, w_i, d_i corresponds to 3 boxes

$$\begin{aligned} h_i, w_i, d_i &= h_i, w_i, d_i \\ h_i^{'}, w_i^{'}, d_i^{'} &= w_i, d_i, h_i \\ h_i^{''}, w_i^{''}, d_i^{''} &= d_i, h_i, w_i \end{aligned}$$

where without loss of generality (i.e. fix an orientation of the base $w_i \times d_i$) we require $w_i \leq d_i$. Call $w_i \times d_i$ the base of a box. So box b_i can be stacked atop b_j if the base of box b_i is smaller than the base of box b_j. First sort the boxes (the $3n$ boxes) by decreasing base dimension. Then the rest is just like longest increasing subsequence (except for base comparison). Running time is $O\left(n^2\right)$ just like longest increasing subsequence.

0.0.39. *Bridge crossings.* You have a river crossing a state with n cities on the south bank and n corresponding cities on the north bank (not necessarily in the same order). You want to build as many bridges connecting corresponding cities as possible without building bridges that intersect. Let x_i be the index of the city on the north shore corresponding to the ith city on the south shore. You can figure this out if you're just given the two lists, i.e. integer array $S = [1, 2, \ldots, n]$ to label the southshore cities and integer array $N = [\sigma(1), \sigma(2), \ldots, \sigma(n)]$ for the permutation on the northshore, by sorting the northshore array (while keeping track which index the elements get sorted **from** - think about it and you'll understand). Then you just need to find the longest increasing subsequence of the x_i array. Running time is $O\left(n^2\right)$ just like longest increasing subsequence.

0.0.40. *Integer Knapsack.* You're a thief with a knapsack that has a finite capacity C. You break into a store that has n items with integer sizes s_i and values v_i. Which items should you steal? You can only take whole items and you're allowed duplicates. The subproblems here are filling smaller knapsacks. So let $M(j)$ be the maximum value obtained by filling a knapsack with capacity exactly j. The maximum value j capacity knapsack that can be constructed is either equal to the maximum $j-1$ capacity knapsack that can be constructed or it includes item i and all of the items in the $j - s_i$ capacity knapsack. Therefore the Bellman equation is

$$M(j) = \max\left\{M(j-1), \max_i \{M(i-1, j-s_i) + v_i\}\right\}$$

Running time is $O(nC)$ because you compute C entries but each computation considers n items.

0.0.41. *0/1 Knapsack.* In this instance you can only take whole items (that's the 0/1) and there are no duplicates. The subproblems here are the optimal value for filling a knapsack with capacity exactly j and with some subset of the items $1, \ldots, i$. $M(i, j)$ either includes items i, in which case it includes all of the items of the optimal knapsack over the items $1, \ldots, i-1$, with capacity $j - s_i$, and in which case it has value $M(i-1, j-s_i) + v_i$, or it does not include item i, in which case it has capacity j and has value $M(i-1, j)$. Hence the Bellman equation is

$$M(i, j) = \max\{M(i-1, j), M(i-1, j-s_i) + v_i\}$$

Then the solution to the whole problem is not $M(n, C)$ but $\max_j \{M(n, j)\}$ because you might not need to use the entire capacity. Running time is still $O(nC)$ because there are $n \times C$ subproblems.

0.0.42. *Balanced Partition.* You get n integers $A = [a_1, \ldots, a_n]$, each in the range $0, \ldots, k$, and the goal is to partition A into two sets S_1, S_2 minimizing $|\text{sum}(S_1) - \text{sum}(S_2)|$. Let $P(i,j)$ be a boolean that reports whether some subset of $[a_1, \ldots, a_i]$ sum to j. Then $P(i,j) = 1$ if some subset of $[a_1, \ldots, a_{i-1}]$ sum to j, in which case we don't need to include item i, or if some subset of $[a_1, \ldots, a_{i-1}]$ sums to $j - a_i$, in which case we include item a_i to get a subset that sums to j. Hence the Bellman equation is

$$P(i,j) = \begin{cases} 1 & \text{if } P(i-1,j) = 1 \text{ or } P(i-1, j-a_i) = 1 \\ 0 & \text{otherwise} \end{cases}$$

More succinctly

$$P(i,j) = \max\{P(i-1,j), P(i-1, j-a_i)\}$$

Note this is just a logical or, i.e. $||$. There are n^2k problems because i range from 1 to n but each a_i could have value k so j ranges from 0 to nk. How do you use this to solve the original problem? Let $S = \sum a_i/2$. Then the subset S_j such that

$$\min_{j \le S} \left\{ S - j \middle| P(n,j) = 1 \right\}$$

produces the solution. Running time is the same $O\left(n^2k\right)$.

0.0.43. *Longest common subsequence.* Given two strings $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ what is the longest common subsequence? Let $Z = [z_1, \ldots, z_k]$ be such a longest common subsequence. Working backwards: if $a_n = b_m$ then $z_k = a_n = b_m$ and $[z_1, \ldots, z_{k-1}]$ is a longest common subsequence of $[a_1, \ldots, a_{n-1}]$ and $[b_1, \ldots, b_{m-1}]$. Suppose that the two sequences A and B do not end in the same symbol. Then the longest common subsequence of A and B is the longer of the two sequences $\text{LCS}([a_1, \ldots, a_n], [b_1, \ldots, b_{m-1}])$ and $\text{LCS}([a_1, \ldots, a_{n-1}], [b_1, \ldots, b_m])$. Hence the Bellman equation is

$$\text{LCS}([a_1, \ldots, a_i], [b_1, \ldots, b_j]) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{LCS}([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_{j-1}]) + 1 & \text{if } a_i = b_j \\ \max\{LCS([a_1, \ldots, a_i], [b_1, \ldots, b_{j-1}]), ([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_j])\} & \text{if } a_i \neq b_j \end{cases}$$

Running time is $O(nm)$.

0.0.44. *Edit distance.* Given two strings $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ what is minimum the "cost" of transforming one string into the other, where the costs associated with insertion, deletion, and replacement are C_i, C_d, C_r respectively. The subproblems here are similar to those in longest common subsequence. Let $T(i,j)$ be the minimum cost of transforming $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_j]$. There are 4 ways to transform $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_j]$: either delete a_i and transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_j]$, transform $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_{j-1}]$ then insert b_j, or replace a_i with b_j and then transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_{j-1}]$. Finally if $a_i = b_j$ then just transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_{j-1}]$. Therefore the Bellman equation is

$$T(i,j) = \min\{C_d + T(i-1,j), T(i,j-1) + C_i, T(i-1,j-1) + C_r, T(i-1,j-1) \text{ if } a_i = b_j\}$$

Running time is $O(nm)$.

0.0.45. *Counting Boolean Parenthesizations.* Given a boolean expression with n literals and $n-1$ operators how many different ways are there to parenthesize such that the expression evaluates to true. Let $T(i,j)$ be the number of ways to parenthesize literal i through j such that the subexpression evaluates to true and $F(i,j)$ to be the number of ways such that the subexpression evaluates to false. The base cases $T(i,i), F(i,i)$ are just function of the literals. Note that $i<j$ so we then seek to compute $T(i,i+1), F(i,i+1), T(i,i+2), F(i,i+2)$ for all i. How? Well $T(i,j)$ is always a function of two subexpression and the operand between them: the literals from i to k and from $k+1$ to j. For example if the operand is $\wedge$ then $T(i,j) > T(i,k)\cdot T(k+1,j)$ since the expression including the literals from i to j will be true for any values of the subexpression from i to k which evaluate to true and any values of the subexpression $k+1$ to j which evaluate to true. If the operator were $\vee$ then it would be $T(i,k)\cdot T(k+1,j)+T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j)$. And we need to sum over all possible splits k. So the Bellman equation is

$$T(i,j)=\sum_{i\le k\le j-1}\begin{cases} T(i,k)\cdot T(k+1,j) & \text{if } k\text{th operator is } \wedge \\ T(i,k)\cdot T(k+1,j)+T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j) & \text{if } k\text{th operator is } \vee \\ T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j) & \text{if kth operator is xor}\end{cases}$$

Running time is $O\left(n^3\right)$.

0.0.46. *Coin game.* Given n coins layed out in a row with values $v_1,\ldots,v_n$ you play a game against an opponent where on each turn you pick up one of the two outside coins. The goal is to maximize the sum of the value of the selected coins. Let $V(i,j)$ be the maximum value you can **definitely** win if it's your turn and only the voince $v_i,\ldots,v_j$ remain. The base cases $V(i,i)$ and $V(i,i+1)$ are easily to compute. We seek to compute $V(i,i+2)$ and etc. We need to think two steps ahead to compute arbitrary $V(i,j)$: if we pick v_i then our opponent will either pick the jth coin of the $i+1$th coin. Reasoning conservatively (the opponent will pick the better) we will be presented with the minimum possible scenario of coins $i+1,\ldots,j-1$ and $i+2,\ldots,j$. If we pick v_j then similarly we will be presented with the minimum possible scenario of coins $i,\ldots,j+2$ and $i+1,\ldots,j-1$. Therefore the Bellman equation is

$$V(i,j)=\max\left\{\underbrace{\min\left\{V(i+1,j-1),V(i+2,j)\right\}+v_i}_{\text{pick } v_i},\underbrace{\min\left\{V(i,j+2),V(i+1,j-1)\right\}+v_j}_{\text{pick } v_j}\right\}$$

Greedy Algorithms.

0.0.47. *Activity scheduling.* Suppose you have a set of activities $A=[a_1,\ldots,a_n]$ with sorted start times $S=[s_1,\ldots,s_n]$and sorted finish times $F=[f_1,\ldots,f_n]$. How to schedule the most non-overlapping acitivities? There's an obvious greedy algorithm: always pick the job that doesn't overlap with already picked jobs and ends the soonest.

0.0.48. *Fractional Knapsack.* This is the same as Integer Knapsack but you can take fractions of items (imagine you broke into a spice shop). The greedy strategy that optimally picks items is one that chooses items that give most bang per weight, a kind of value density: pick as much of the item that has the highest v_i/w_i until it's exhausted. Then continue on to the next most value dense item.

0.0.49. *Huffman codes.* What's the most optimal way to encode a message using a $\{0,1\}$ code given the distribution over the input alphabet? Letters that appear most often should have the smallest code words and conversely letters that appear rarely should have the longest code words. Using prefix-free codes (codes such that no codeword is a prefix of any other codeword) we can achieve optimal compression so without loss of generality we can use them, and we will since they make things easiest.

Given the frequency distribution C we can construct a binary tree called a Huffman tree (leaves correspond to terminals of codewords) whose traversal (0 for left and 1 for right) produces the prefix-free codes using a min-queue keyed on the frequency: extract minimums and merge them and reinsert with value being sum of children.

Running time is $O(n \log n)$ due to the min-queue operations. Constructing the codes is done by performing a depth-first traversal of the Huffman tree and keeping track of lefts and rights (zeros and ones).

0.0.50. *Making change.* Consider the problem of making change for n cents using the fewest number of coins $K = [c_1, \ldots, c_k = 1]$. Assume each coin's value is an integer. If the coins are the US quarters, dimes, nickels, and pennies then a greedy algorith is optimal: change as much for quarters as you can, then as much for dimes, etc. The greedy strategy does not always work: suppose the coins are of denomination $4¢, 3¢, 1¢$ to change $6¢$. Let $C(i)$ be the optimal number of coins used to make change for $i¢$ using any of the coins. The minimum number of coins needed to change i is 1 plus $C(i - c_j)$ where c_j is the coin denomination that minimizes $C(i - c_j)$ and $c_j < i$. Therefore the Bellman equation is

$$C(i) = \min_j \left\{ C(i - c_j) \,\middle|\, c_j < i \right\} + 1$$

Running time is $O(nk)$.

0.0.51. *Making change.* Here is another solution. I don't understand why there should be another solution but here it is. Suppose the coins come sorted in decreasing order so $c_1 > c_2 > \cdots > c_k = 1$. Let $C(i, j)$ be the optimal number of coins used to make change for $i¢$ using only coins $j, \ldots, k$. We either use coin c_j or we don't. If we do not then we're solving the problem $C(i, j+1)$. For example we might not use coin c_j if $c_j > i$. If we do use coin c_j then the rest $(i - c_j)¢$ needs to be changed, potentially using the coin j again.

$$C(i,j) = \begin{cases} C(i, j+1) & \text{if } c_j > i \\ \min_j \{C(i, j+1), C(i - c_j, j) + 1\} & \text{if } c_j \leq i \end{cases}$$

Running time is also $O(nk)$.

Part 5. Graph Algorithms

Representations of Graphs. There are two ways to represent a graph $G = (E, V)$: **adjacency matrix** and **adjacency list**.

0.0.52. *Adjaceny matrix.* The former is a table with $n = |V|$ rows and n columns and with an entry in row i column j if there's an edge between vertex i and vertex j. The value of the entry could be anything from simply 1 to indicate an undirected edge, -1 to represent a directed edge, k to represent an edge weight, 0 to represent no edge.

0.0.53. *Adjacency list.* The latter is a list of lists where the i entry in the list is a list containing all j such that edge $(i, j) \in E$. Most algorithms in this section will use the adjacency list representation. Further more we assume that other attributes will be stored in a hash table keyed on the vertex "name", which is a number.

0.0.54. *Transpose.* The transpose graph $G^T = \left(V, E^T\right)$ where $(u, v) \in E^T$ iff $(v, u) \in E^T$, i.e. reverse all the arrows. Computing the transpose graph when a graph is represented by an adjacency matrix amounts to just transposing the matrix. When the original graph is represented by adjacency lists it's a little more complicated but pretty obvious regardless.

Traversals.

0.0.55. *Breadth-first Search.* A bread-first search is exactly what it sounds like: all vertices at a certain breadth (distance) are visited, then the next breadth, then the next breadth, and so on. In order to prevent repeatedly visiting the same vertices we need to keep track of which vertices we've visited. The most elegant way is to "decorate" by constructing tuples $(i, visited)$ and unpacking. An easier way is to just have a hash table that stores that attribute. Running time is $O(V + E)$. In Python you should keep track of whether a vertex is undiscovered, discovered, explored.

0.0.56. *Depth-first Search.* A depth-first search is exactly what it sounds like: go as deep as possible then back up until you can go deep again, and so on. For depth search we also keep track of what are called opening and closing times: open time is when a vertex begins to be explored, and close time is when it's done being explored. Running time is $O(V + E)$. In Python you should keep track of whether a vertex is undiscovered, discovered, explored, and leave vertices on the stack when they're discovered but unexplored (and change their status).

Basic Algorithms.

0.0.57. *Topological Sort.* A topological sort of a directed acyclic graph $G = (V, E)$ is an ordering on V such that if $(u, v) \in E$ then u appears before v in the ordering. Producing a topological sort is easy using `Depth-First-Search`: the `visited` array already returns the topological sort! The vertex at the front of the list is first in topologically sorted order, the second is the second, and so on. If the graph is connected then some vertices might be unvisited after starting from a particular source. In that case you need to run DFS on every vertex (making sure to not to run twice on a vertex that's already been visited). Running time is $O(V + E)$.

0.0.58. *Strongly Connected Components.* A connected component of a graph $G = (V, E)$ is a subset $V' \subset V$ such that for every $u, v \in V'$ there's a path $u \rightsquigarrow v$ and $v \rightsquigarrow u$. How do you compute all of the connected components of a graph? A topological sort and DFS on the transpose graph G^T. First topological-sort all of the vertices in the graph G. Then DFS the transpose graph G^T in the topologically sorted order produced by the topological sort G.

Minimum Spanning Tree.

0.0.59. *Kruskal's algorithm.* Kruskal's algorithm uses the Union-Find data structure to build the minimum spanning tree. Sort the edges by weight the traverse that list unioning sets of vertices (and keeping track of edges that connect them) that haven't been added to the minimum spanning tree yet.

The running time is a function of the running times of the Union-Find data structure operation running times. The second `for` performs $O(E)$ `Find-Set` and `Union` operations. Along with the $O(V)$ `Make-Set` operations in the first `for` the total is $O((V+E)\alpha(V))$, where α is the Ackermann function. Since we assume G is connected (otherwise it could have no spanning tree) it's the case that $E \geq V-1$ and so the Union-Find operations actually take $O(E\alpha(V))$. Then since $\alpha(V) = O(\lg V) = O(\lg E)$ we get that the run time is $O(E \lg E)$. Finally since $|E| < |V|^2$ we have that $\lg(E) = O(\lg V)$ and therefore the running time is $O(E \lg V)$.

0.0.60. *Prim's algorithm.* Prim's algorithm uses a min priority queue to keep the sorted list of "lightest" edges by keeping track of vertices and edges connecting them to the minimum spanning tree. Initially all vertices have $v['\texttt{key}'] = \infty$ and arbitrary vertex is chosen to be the nucleation point of the minimum spanning tree, i.e. $v['\texttt{key}'] = 0$. Insert all of the vertices into the priority queue and keep extracing the minimum and updating neighbors' keys with the weight of the edge connecting the neighbor to that extracted minimum, and setting predecessor pointers. The actual minimum spanning tree is the predecessor tree.

Running time is $O(E \lg E)$ for a standard implementation of a min heap but can be sped up to $O(E + V \lg V)$ using a Fibonacci heap.

Single source Shortest Path. Single source means shortest path from a particular vertex to all other vertices in the graph.

0.0.61. *Bellman-Ford.* Bellman-Ford is kind of stupid simple: just "relax" all of the distance $|V|-1$ times.

`Relax`(u, v, w)

1 `if` $v['\texttt{dist}'] > u['\texttt{dist}'] + w$`:`
2 $\quad v['\texttt{dist}'] = w$
3 $\quad v['\texttt{prnt}'] = u$

Bellman Ford call also check at the end for negative weight cycles: if any distances can be further relaxed then that vertex is on a negative weight cycle (this follows from the fact that any shortest path can undergo at most $|V|-1$ relaxations). Running time is $O(VE)$.

For completeness I'll mention that to actually find a negative weight cycle if one exists run Bellman-Ford twice: the first time finds and edge on a negative weight cycle. The second time run Bellman-Ford with the source vertex being the one that the distance could have been relaxed to and trace the path produced by Bellman-Ford to its parent.

0.0.62. *Shortest Path in a DAG.* In a dag you can speed up Bellman-Ford because you can figure out exactly the order in which to relax the edges: just do a topological sort.

0.0.63. *Dijkstra's.* Dijkstra's shortest path algorithm is very similar to Prim's minimum weight spanning tree algorithm. Just like Prim's it uses a min priority queue to keep track of objects in the graph, except it's nearest vertices according to

$$E\left[(u,v)\right] + u\left['\text{dist}'\right] < v\left['\text{dist}'\right]$$

rather than lightest edges crossing a cut. Running time is $O\left(E + V \lg V\right)$ using a Fibonacci heap implementation of min queue. Note that Dijkstra does not work for graph with negative weight edges.

0.0.64. *Heuristic search.* Suppose you wanted to use Dijkstra's to find the shortest path to a particular vertex g, for goal. Well you could just run it and recover the path to g but Dijkstra will waste a lot of time searching the rest of the graph. You can hack Dijkstra to be a little faster by using a different priority function (one that encodes a heuristic for most expedient direction). This prompts it to explore in a particular direction more often since the vertices prioritized by the heuristic function will be popped first from the min queue. The code is exactly the same except for

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right]$$

which becomes

$$v\left[\texttt{'dist'}\right] = \texttt{heuristic}\left(v, g\right)$$

To be concrete suppose we're trying to find the shortest path to a vertex on a grid. Then $\texttt{heuristic}\left(v, g\right)$ would just be Manhattan distance (closer Manhattan distance means higher priority).

```
Manhattan-Distance(a, b)
1  return |a[x] − b[x]| + |a[y] − b[y]|
```

0.0.65. *A* search.* A star search combines heuristic and Dijkstra's to take into account distance from source and some heuristic for distance to goal. The modification to Dijkstra is

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right]$$

becomes

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right] + \texttt{heuristic}\left(v, g\right)$$

0.0.66. *Difference constraints.* A set of difference constraints is a set $x_j - x_i \leq b_k$ whose solution is $\mathbf{x}$ such that all of the constraints are satisfied. These can be solved by first constructing a constraint graph $G = \left(\{v_0, v_1, \ldots, v_n\}, E\right)$ where

$$E = \left\{ \left(v_i, v_j\right) \middle| x_j - x_i \leq b_k \text{ is a constraint} \right\} \cup \left\{\left(v_0, v_i\right)\right\}$$

where $w\left(\left(v_i, v_j\right)\right) = b_k$ and $w\left(\left(v_0, v_i\right)\right) = 0$. Then using Bellman-Ford to find the shortest path $\delta\left(v_0, v_i\right)$ from v_0 to every other vertex. If there's a negative weight cycle then no solution exists. The

proof that Bellman-Ford produces a solution hinges on the triangle inequality $\delta(a,b) \leq \delta(a,c) + \delta(b,c)$: since the distance to each of the vertices is 0

$$\delta(v_0, v_j) \leq \delta(v_i, v_j) + \delta(v_i, 0)$$

implies

$$\delta(v_0, v_j) - \delta(v_i, v_0) \leq \delta(v_i, v_j) = w((v_i, v_j))$$

A system of m constraints in n unknowns produces a graph with $n+1$ vertices and $n+m$ edges and hence running time is $O\left(n^2 + nm\right)$.

0.0.67. *Transitive Closure.* The transitive closure of a graph $G = (V, E)$ is a graph $G' = (V, E')$ where $(u, v) \in E'$ if there's a path $u \rightsquigarrow v$ in G. This problem has optimal substructure: consider all of the paths from i to j where intermediate vertices (vertices in the path not including i, j) come from vertices $\{1, \ldots, k\} \subset \{1, \ldots, n\}$ and consider a particular path p. Either k is an intermediate vertex of p or not. If k is not an intermediate vertex then all p's intermediate vertices are drawn from $\{1, \ldots, k-1\}$. If k is an intermediate vertex of p then we can further decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ where k is not an intermediate vertex of either p_1 or p_2. Let $t_{ij}^{(k)}$ be 0 or 1 depending on whether i is connected to j in the transitive closure of G or not, then the Bellman equation is

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ or } \left(t_{ik}^{(k-1)} \text{ and } t_{kj}^{(k-1)}\right)$$

with base case

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

All pairs Shortest Paths.

0.0.68. *Shortest paths by exponentiation.* Shortest paths have optimal substructure: if vertices i and j are distinct, then we can decompose the path p from i to j into $i \overset{p'}{\rightsquigarrow} k \to j$ where p' must be the shortest path from i to k. Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to j that contains at most m edges and w_{uv} be the weight of the edge between vertices u and v, then the Bellman equation is

$$\begin{aligned} l_{ij}^{(m)} &= \min\left\{l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right\} \\ &= \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \end{aligned}$$

since $w_{jj} = 0$. Base case is

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

The shortest path weight is then $l_{ij}^{(n-1)}$. Given a matrix L that corresponds to the mth iteration we can compute L' corresponding to the $m+1$th iteration using $W = \{w_{ij}\}$.

The thing to notice is that this is very much like matrix multiplication $L \cdot W$, and hence schematically

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W = W \\ L^{(2)} &= L^{(1)} \cdot W = W^2 \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1} \end{aligned}$$

Therefore we can use exponentiation by repeated squaring to compute $L^{(n-1)}$. In fact it's even simpler because we just need to square and not worry about anything else since $L^{(n+k)} = L^{(n-1)}$ for all k (since shortest paths don't become shorter...). Running time is $\Theta\left(n^3 \lg n\right)$.

0.0.69. *Floyd-Warshall.* Floyd-Warshall is very similar to Transitive closure. Consider a subset $\{1, \ldots, k\}$ of vertices. For any two vertices i, j consider all paths whose intermediate vertices (vertices in the path not including i, j) all come from $\{1, \ldots, k\}$ and let p be the minimal weight path from among them. If k is not an intermediate vertex of p then all intermediate vertices of p come from $\{1, \ldots, k-1\}$. If k is an intermediate vertex of p then we can decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ where k is not an intermediate vertex of neither p_1 nor p_2 where both p_1, p_2 have intermediate vertices coming from $\{1, \ldots, k-1\}$. Furthermore both p_1, p_2 are shortest paths themselves. Therefore the Bellman equation is

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k > 0 \end{cases}$$

Running time is obviously $O\left(n^3\right)$.

0.0.70. *Johnson's Algorithm.* John's algorithm is slightly faster than Floyd-Warshall on sparse graphs. It works by reweighting all of the vertices so that none are negative using Bellman-Ford and then runs Dijkstra from each vertex. It reweights in a way that doesn't change any of the shortest paths: for any $h(u)$ that maps vertices to real numbers

$$\hat{w}((u,v)) = w((u,v)) + h(u) - h(v)$$

does not alter shortest paths. How to pick $h(u)$ so that $\hat{w}((u,v)) > 0$. Make it a distance function: similar to how a super source is used in difference graphs define a new vertex s with 0 weight edges to every other vertex and let $h(u) = \delta(s,v)$. Since $\delta(s,v)$ is a distance function by the triangle inequality

$$h(v) \leq h(u) + w(u,v)$$

and hence

$$\hat{w}((u,v)) = w((u,v)) + h(u) - h(v) \geq 0$$

So just like in difference constraints use Bellman-Ford to compute $\delta(s,v)$ for all $v \in V$. Running time is $O\left(V^2 \lg V + VE\right)$.

Min Cut - Max Flow. A flow network $G = (V, E)$ is a directed graph in which each edge has a capacity $c((u, v)) \geq 0$ and if a forward edge exists then no reverse edge exists. Further there are two distinguished vertex, source s and sink t, such that for every vertex v it's the case that $s \rightsquigarrow v \rightsquigarrow t$. A **flow** on a flow network is a real valued function $f : V \times V \to \mathbb{R}$ that satisfies two properties:

(1) Capacity constraint: For all u, v, it's the case that $0 \leq f(u, v) \leq c((u, v))$
(2) Flow conservation: For all $u \in V - \{s, t\}$, it's the case that that flow in equals flow out, i.e.

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

The **value** of the flow $|f|$ is defined as the total flow out of the source minus the total flow into the source, i.e.

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

Typically, a flow network will not have any edges into the source and so $|f| = \sum f(s, v)$. The **maximum-flow** problem is maximizing $|f|$ on a particular flow network G.

0.0.71. *Ford-Fulkerson/Edmond's-Karp.* Given a flow network G we construct a **residual network** G' which models "room to augment" the flow. Then we augment until there's no more room. Define the **residual capacity** $c_f(u, v)$ to be

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

The first case is clearly "room to grow". The second case is the flow that's currently going across edge (u, v) but in the reverse direction (think of it "room to squelch" the flow). The residual network $G' = (V, E_f)$ where

$$E_f = \left\{ (u, v) \in V \times V \middle| c_f(u, v) > 0 \right\}$$

An **augmentation** of a flow f by f' is a new flow $(f \uparrow f')$ defined to be

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

i.e. add $f'(u, v)$ units of flow along edge (u, v) but squelch by $f'(v, u)$ units (in the residual network there will be an edge (v, u) with $f'(v, u)$ units of flow on it, think of it as back pressure). Turns out that if f' is a flow in G' then $|f \uparrow f'| = |f| + |f'|$.

An **augmenting path** p is a simple path from s to t in G'. The **residual capacity** $c_f(p)$ of p is defined

$$c_f(p) = \min \left\{ c_f(u, v) \,\middle|\, (u, v) \in p \right\}$$

So define a flow $f_p(u, v)$ that flows this minimum capacity along the path, i.e.

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \in p \\ 0 & \text{otherwise} \end{cases}$$

and then f_p is a flow in G' with $|f_p| = c_f(p) > 0$. As a corollary $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Recall cuts $(S, V \backslash S)$ of a graph. The **capacity of a cut** (S, T) where $s \in S$ and $t \in T$ is defined

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

i.e. the sum of the capacities of all of the edges crossing the cut. The **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network (where the source is in one partition and the sink is in the other). Naturally the net flow across any cut is equal to $|f|$. Something stronger is true.

Theorem 1. *Min cut - Max flow*

The following are equivalent

(1) *f is a maximum flow in G.*
(2) *The residual network G' contains no augmenting paths.*
(3) *$|f| = c(S,T)$ for some cut (S,T) of G. In fact the minimum capacity cut.*

After all that the Ford-Fulkerson/Edmond's-Karp algorithm for finding the max flow on a flow network G is simple: construct the residual network G' and keep looking for augmenting paths and use them to augment the flow on G. When there are no more we know that f is a maximum flow and that $|f|$ is equal to the minimum cut. We use breadth-first search to find the augmenting path. First we need a function that constructs the residual network.

To find the min-cut: do a depth-first search in the *residual* network from the source s marking all vertices which can be reached. The cut is all edges in the actual flow network going from marked vertices to unmarked vertices. Those edges are saturated and correspond to the minimum cut.

0.0.72. *Maximum bipartite matching.* A bipartite graph is one for which the vertices can be divided into two sets, with edges only going between the two sets. A **maximum bipartite matching** is one which 1-1 matches the largest number of vertices. The matching problem is finding such a matching. The solution is two use Min cut - Max flow: set up a source that connects to all of the vertices in one partition and a sink that connects to all of the vertices in the other. Then set the capacity of all edges to 1 and run Edmonds-Karp. The minimum cut corresponding to the maximum flow gives you the matching. How do we know that the flow will take on an integer value? I.e. that we won't flow 3/4 and 1/4 down some edges?

Theorem 2. *Integrality theorem*

If the capacity function $c((u,v))$ takes on only integral values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer and furthermore $f(u,v)$ is an integer for all u, v.

Running time is $O(VE)$.

Part 6. Selected Topics

Number Theory.

0.0.73. *Euclidean GCD Algorithm.* $\gcd(a,b)$ is the greatest common divisor of a,b. If $a > b$ then it's the case that $\gcd(a,b) = \gcd(b, a \mod b)$. This leads to a naive algorithm for gcd. Running time is $O(\lg b)$.

```
Naive-GCD(a, b)
# assume a > b
if b == 0:
    return a
else:
    return Naive-GCD(b, a mod b)
```

We can even make the algorithm iterative easily.

```
Naive-GCD-Iter(a, b)
# assume a > b
while b ≠ 0:
    # note the flip so that we preserve a > b
    a, b = b, a mod b
return a
```

Furthermore for any common divisor of a,b we have $d = ax + by$ for some x, y. How do we find x, y for $d = \gcd(a,b)$? In general we need x', y' from the next iteration and a, b from the current iteration and we can use the recursion relation

$$\begin{aligned} x &= y' \\ y &= x' - \left\lfloor \frac{a}{b} \right\rfloor y' \end{aligned}$$

```
Naive-Extended-GCD(a, b)
# assume a > b
if b == 0:
    return a, 1, 0
else:
    d, x, y = Naive-Extended-GCD(b, a mod b)
    x, y = y, x − ⌊a/b⌋ · y
    return d, x, y
```

This version is tougher to make iterative (or at least to prove that the iterative version is correct). Running time is still $O(\lg b)$.

```
Extended-GCD(a, b)
1  # assume a > b
2  x_0, y_0, x_1, y_1 = 1, 0, 0, 1
3  while b ≠ 0:
4      q ⌊a/b⌋
5      a, b = b, a mod b
6      x_0, x_1 = x_1, x_0 − q · x_1
7      y_0, y_1 = y_1, y_0 − q · y_1
8  return a, x_0, y_0
```

0.0.74. *Solving modular linear equations.* A modular linear equation is $ax \equiv b \mod n$ ($ax = kn$ for some k). The task is to solve for x.

Theorem 3. *Let* $\gcd(a, b,) = d = ax' + ny'$ *for integers* x', y'. *If* $d|b$ *then* $ax \equiv_n b$ *has at least one solution*

$$x_0 = x'\left(\frac{b}{d}\right) \mod n$$

Proof. We have

$$\begin{aligned} ax_0 &\equiv_n ax'\left(\frac{b}{d}\right) \\ &\equiv_n d\left(\frac{b}{d}\right) \text{ since } ax' \equiv_n d \\ &\equiv_n b \end{aligned}$$

□

Suppose x_0 *is a solution to the above. Then there are* $d - 1$ *more solutions*

$$x_i = x_0 + i\left(\frac{n}{d}\right)$$

for $i = 1, \ldots, d - 1$.

Proof. Since x_0 is a solution

$$\begin{aligned} ax_i &\equiv_n a\left(x_0 + i\left(\frac{n}{d}\right)\right) \\ &\equiv_n ax_0 + ai\left(\frac{n}{d}\right) \\ &\equiv_n ax_0 \text{ since } d|a \Rightarrow \left(\frac{a}{d}in\right) = kin \\ &\equiv_n b \end{aligned}$$

□

This suggests an algorithm for finding all the solutions to $ax \equiv_n b$.

0.0.75. *Floyd's cycle finding algorithm.* Suppose you have a sequence of values $a_0, a_1, a_2, \ldots$ that loops. How would you detect it without storing all the values? Suppose μ is the first index of the loop and λ is the length of the loop, then for any integers i, k such that $i \geq \mu$ and $k \geq 0$ it's the case that

$$x_i = x_{i+k\lambda}$$

In particular if $i = k\lambda$ we have that $x_i = x_{2i}$. Thus we only need to iterate through the values with two iterators, one that goes twice as fast as the other and wait for equality. Once that occurs the two iterators are a distance $i = k\lambda$ apart, i.e. a multiple of the period of the loop, and the first iterator is $i = k\lambda$ from the beginning. Resetting the second iterator to the beginning and advancing them each one at a time keeps them a fixed distance $i = k\lambda$ apart. Therefore once both of them are on the loop again (i.e. the second iterator returns to the loop) they must agree, i.e. they'll agree for the first time at the first element of the loop μ.

0.0.76. *Exponentiation by squaring.* How does exponentiation by repeated squaring work?

$$a^b = \begin{cases} a\left(a^{\frac{b-1}{2}}\right)^2 & \text{if } b \text{ is odd} \\ \left(a^{\frac{b}{2}}\right)^2 & \text{if } b \text{ is even} \end{cases}$$

The algorithm is to write out the binary representation of the exponent, and start building it using squaring and multiplication by the base.

Another way to look at the recurence relation is

$$a^b = \begin{cases} a\left(a^2\right)^{\frac{b-1}{2}} & \text{if } b \text{ is odd} \\ \left(a^2\right)^{\frac{b}{2}} & \text{if } b \text{ is even} \end{cases}$$

which gives the equivalent algorithm

Exponentiation-Squaring-Rec(a, b)

```
1  if b == 0:
2      return 1
3  if b mod 2 == 0:
4      return Exponentiation-Squaring-Rec(a · a, b/2)
5  else:
6      return a · Exponentiation-Squaring-Rec(a · a, (b−1)/2)
```

This is the one we'll make bottom up.

```
Exponentiation-Squaring-Iter(a, b)
if b == 0:
    return 1
eif b == 1:
    return a
else:
    y = 1
    while b > 1:
        if b mod 2 == 0:
            a, b = a · a, b/2
        else:
            y = y · a
            a, b = a · a, (b − 1) /2
    return a · y
```

Running time for all the algorithms is $O(\lg b)$. Note that all of these can be made to perform modular exponentiation but simply taking mod in the right places.

String Matching.

0.0.77. *Naive string matching.* The obvious algorithm runs in $\Theta((n-m+1)m)$ time.

```
Naive-String-Matching(T, P)
n = len(T)
m = len(P)
for s = 0 : n − m:
    if T[s + 1 : s + m] == P:
        return s
return None
```

0.0.78. *Rabin-Karp.* Rabin-Karp basically uses an incremental hash. Suppose $\Sigma = \{0, \ldots, 9\}$, so that each character is a decimal digit (in general we can use radix-d notation where $d = |\Sigma|$). Given a pattern $P = [p_1, \ldots, p_m]$ we can use horners rule to compute the number represented by the pattern in $\Theta(m)$ time:

$$p = p_m + 10\left(p_{m-1} + 10\left(p_{m-2} + \cdots + 10\left(p_2 + 10p_1\right)\right)\right)$$

Similarly we can compute $t = T[1:m]$ in $\Theta(m)$ time. Once we have p, t we can compare them in a straighforward way, i.e. check if $p - t = 0$. If so then we have a math. If not we need to advance to the next m digits of T. We could recompute $t' = T[2:m+1]$ all over again but we actually do better: the updated t' is related to the first t by

$$t' = 10\left(t - 10^{m-1}t_1\right) + t_{m+1}$$

In general if $t^{(s)}$ represents $T[s+1:s+m]$ then

$$t^{(s+1)} = 10\left(t^{(s)} + 10^{m-1}t_{s+1}\right) + t_{s+m+1}$$

One thing we haven't considered is what happens when p is too large to work with conveniently (larger then a processor word for example). Use mod (this is the sense in which Rabin-Karp is a rolling/incremental hash): pick q to be prime and such that dq fits in a word and then the recurrence update for $t^{(s)}$ becomes

$$t^{(s+1)} = \left[d\left(t^{(s)} - \left(d^{m-1} \mod q\right)t_{s+1}\right) + t_{s+m+1}\right] \mod q$$

One problem is that we might get false positives: two numbers might be equal $\mod q$ but not be equal, so we need to verify. Preprocessing time is $\Theta(m)$ and running time is $O(n+m)$ (given some assumptions about how many valid shifts and spurious hits).

0.0.79. *Finite automata.* A **finite automaton** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where Q is a finite set of **states**, $q_0 \in Q$ is the **start state**, $A \subset Q$ is a "distinguished" set of states called **accepting states**, Σ is a finite **input alphabet**, δ is a function from $Q \times \Sigma \to Q$ called the **transition function**.

The idea is to construct a finite automaton $M(P)$ that ends up in an accepting state if it scans T and there's a match between P and T. In order to do this we focus on the state space Q and the transition function δ. First define the **suffix function**

$$\sigma(x) = \max_k (P[1:k] \sqsupseteq x)$$

i.e. the longest prefix of P that is a suffix of x (think about sliding x from left to right underneath P). Note that $x \sqsupseteq y$ implies $\sigma(x) \leq \sigma(y)$, i.e. x is a suffix of y implies that possibly a longer prefix of P overlaps with a suffix of y, than a suffix of x. Given a pattern P define the string matching automaton to be:

(1) $Q = \{0, 1, \ldots, m\}$, i.e. how many letters of P have been matched. m is the only accept state.
(2) $\delta(q, a) = \sigma(P[1:q]\,a)$, where $P[1:q]\,a$ is $P[1:q]$ concatenated with a.

This definition for δ lets us "recycle" work already done. If we've matched $P[1:q]$ against $T[s+1:s+q]$ and $T[s+q+1] = a$ then

$$T[s+1:s+q]\,a = P[1:q]\,a = P[1:q+1]$$

i.e. a is a match, then we should advance to state $q+1$ (and indeed the longest prefix of P that is a suffix of $P[1:q]\,a$ is $P[1:q+1]$ and therefore $\sigma(P[1:q]\,a) = q+1$). On the other hand if a is not a match then

$$T[s+1:s+q]\,a = P[1:q]\,a \neq P[1:q+1]$$

i.e. we don't necessarily need to start all the way at the beginning, comparing a against $P[1]$, because maybe some trailing part of $T[s+1:s+q]\,a$ matches some prefix P. This is exactly what $\sigma(T[s+1:s+q]\,a) = \sigma(P[1:q]\,a)$ encodes.

Computing δ is straightforward: do exactly what $\sigma(P[1:q]\,a)$ says (look for the longest prefix of P that matches $P[1:q]\,a$). Running time is $O\left(m^3|\Sigma|\right)$ but it can be improved to $O(m|\Sigma|)$.

```
Transition-Function(P, Σ)
m = len(P)
for q = 0 : m:
    for a ∈ Σ:
        # initial optimistic guess, i.e. maybe
        # a is a match and advances us
        k = min(m, q + 1)
        # if we've matched q characters in P
        # and the next character is a
        while P[1 : k] ⋣ P[1 : q] a:
            k = k − 1
        δ(q, a) = k
return δ
```

Then the function that implements M straightforward. Tototal running time then is $O\left(m^3 |\Sigma|\right)$ or $O(m|\Sigma|)$ preprocessing time and $O(n)$ matching time.

```
Finite-Automaton-Matcher(P, T, Σ)
m = len(P)
n = len(T)
δ = Transition-Function(P, Σ)
for i = 1 : n:
    q = δ(q, T[i])
    if q == m:
        # δ only reports m at the end of matching
        # all of P
        return i − m
return None
```

0.0.80. *Knuth-Morris-Pratt.* A clearly useful piece of information to have when matching is: given $P[1:q]$ match against $T[s+1:s+q]$, what is the least shift $s' > s$ such that for $k > q$

$$P[1:k] = T[s'+1:s'+k]$$

where $s' + k = s + q$? Alternatively, given that $P[1:q]$ is a suffix of $T[1:s+q]$ for some s, what is the longest proper prefix $P[1:k]$ of $P[1:q]$ that is also a suffix of $T[1:s+q]$. Knuthi-Morris-Pratt answers this question using a **prefix function** π, which is pre-computed in $\Theta(m)$ time. It does this by comparing P against itself: since $T[s'+1:s'+k]$ is a part of the known portion of the text, it is a suffix of $P[1:q]$. That is to say, we're looking for the greatest $k < q$ such that $P[1:k] \sqsupset P[1:q]$ (longest prefix of P that is a *proper* suffix of $P[1:q]$) and then next potentially valid shift is $s' = s + (q - k)$. Formally

$$\pi[q] = \max_{k<q} \{P[1:k] \sqsupset P[1:q]\}$$

Let's look at the matching algorithm before working out how to compute π. Running time is $O(n)$.

```
Knuth-Morris-Pratt(P, T)
m = len(P)
n = len(T)
π = Prefix-Function(P)
q = 0
# enter the loop with P[0] = ε
# having been matched
for i = 1 : n:
    # at the top of the loop before this next line we have that P[1 : q]
    # characters of P have been matched against some suffix
    # T[s : i − 1] or T[1 : i − 1]. here we test whether the next character
    # of P matches the next character of T
    while q > 0 and P[q + 1] ≠ T[i]:
        # if the match fails then we look for where to "backtrack"
        q = π[q]
    # if we broke out of the loop because the next character after
    # backtracking matches then we've matched one more character
    # otherwise we've matched 0 characters (i.e. not even the first
    # character of the pattern matches T[i])
    if P[q + 1] == T[i]:
        q = q + 1
    if q == m:
        return i − m
return None
```

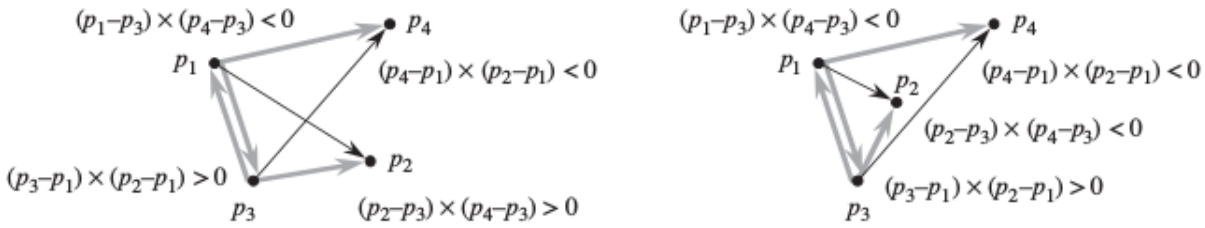
Knuth, Morris, and Pratt (who knows) came up with a clever way to compute π efficiently. I don't understand it exactly but it works in almost the exact same way as the matcher (therefore understanding the matcher, which is sensible, would allow you to reproduce the code for constructing the prefix function).

Running time is $O(m)$.

```
Prefix-Function(P)
1  m = len(P)
2  π = m · [0]
3  k = 0
4  for q = 2 : m:
5      while k > 0 and P[k + 1] ≠ P[q]:
6          k = π[k]
7      if P[k + 1] == P[q]:
8          k = k + 1
9      π[q] = k
10 return π
```

Computational Geometry.

0.0.81. *Segment Intersection.* To figure out whether two line-segments intersect figure out whether their endpoints "straddle" each other. What does that mean? Consider this picture



There's an edge case: if the linge segment intersect at endpoints then some of the cross products will be zero, so we have to check for that. Runtime is constant.

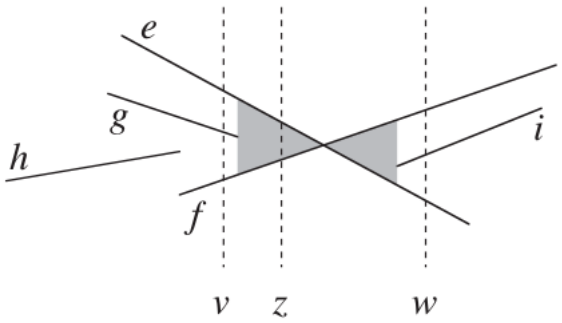
0.0.82. *Segment Pair intersection/Bentley-Ottmann.* Given a set of line segments S figure out whether any pair intersect. The naive algorithm is obviously $O\left(n^2\right)$. The faster algorithm takes $O\left(n \lg n\right)$ and uses a "sweep line": a data structure the keeps track lines intersected as an imaginary line is "swept" across the set of lines.

To line segments s_1, s_2 are **comparable** at coordinate x if the vertical sweep line with x-coordinate x intersects both of them. s_1 is **above** s_2 if s_1, s_2 are comparable and the intersection of s_1 with the sweep line is above the intersection of s_2 with the sweep line. Being above is a total preorder: the relation is transitive and if s_1, s_2 are comparble at x then either s_1 is above s_2, or s_2 is above s_1, or both (if s_1, s_2 intersect at the sweep line with x-coordinate x).

The sweep lines algorithm manages two sets of data: **sweep-line status**, which gives the relationships between objects the sweep line intersects, and **event-point schedule**, which are the x-coordinates of the discrete steps of sweep line. We assume that the sweep-line status is a data structure that supports inserting a line segment s, deleting a line segment s, inspecting any line segments above s, and any line segments below s. We implement all of this with cost $O\left(\lg n\right)$

using a balanced binary search tree keyed on a sort of the line segments using comparison by cross product.

Assume that no three segments intersect at the same point. The key insight of the algorithm is that two line segments that intersect must be *consecutive* in the sweep-line status at some point in the event-point schedule. Consider the following picture



Line segments e, f intersect but they are not consecutive in the sweep-line status until after the end of line segment g. Supposing g were absent they would become consecutive at the beginning of f. So we only need to check at left endpoints whether a line segment intersects with line segments either below and above, or we need to check when removing line segments whether those above and below intersect. Here is the code. Running time is $O(n \lg n)$.

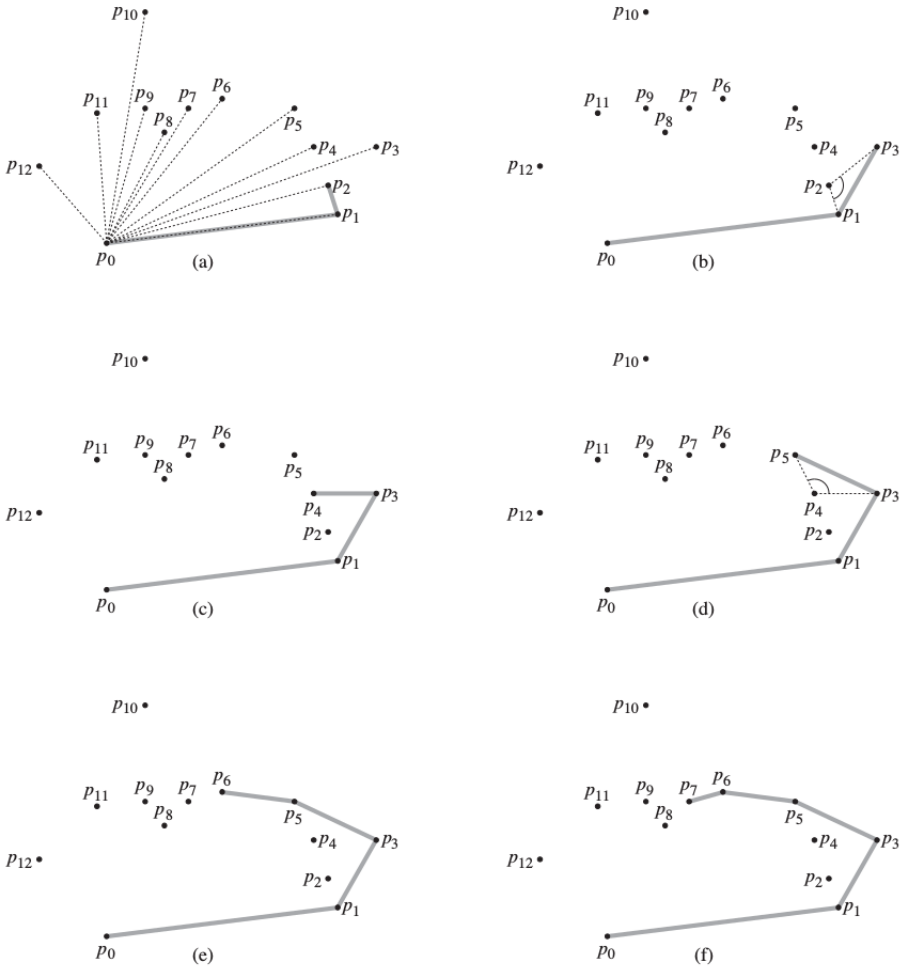
```
Set-Segments-Intersect(S)
# T is a binary balanced search tree with comparison being done
# by relative orientation using cross product
# so that we can fetch line segments by either left
# endpoint or right endpoint
left = {s[1] : s for s ∈ S}
right = {s[2] : s for s ∈ S}
T = tree()
# lexicographically sort all of the endpoints in S
# except left endpoints should precede right endpoints
sched = sorted([p for line ∈ S for p ∈ line]))
for p ∈ sched:
    if p ∈ left:
        if p ∈ right:
            return True
        else:
            s = left[p]
            Insert(T, s)
            if Segment-Intersect(Above(T, s), s) or Segment-Intersect(Below(T, s), s):
                return True
    else: # p is a right endpoint.
        s = right[p]
        # this is the case alluded to above: if a third line segment
        # intervenes between two other line
        if Segment-Intersect(Above(T, s), Below(T, s)):
            return True
        Delete(T, s)
return False
```

Note that the algorithm does not find all intersections (only an intersection). The algorithm that prints *all* of the intersection is called Bentley-Ottman and operates similarly. Instead of simply a sorted list of event points it uses a priority queue of event points. It then does one of 3 things depending on the type of event point

(1) If p is a left endpoint of a line segment s, then insert s into the T, and if s intersects a neighbor then insert their intersection point to the priority queue (computing the intersection point can be done in constant time by solving the system of equations defining the two lines that correspond to the line segments).
(2) If p is right endpoint, then check the intersection of its neighbors, and delete p from T. If the neighbors intersect then add their intersection point to the priority queue.
(3) If p is an intersection point of two lines segments s_1, s_2, then print, and exchange their order in T. If the new neigbhors of s_1, s_2 intersect with either s_1 or s_2 then insert those intersection points.

Running time for k intersections is $O((n+k)\lg n)$.

0.0.83. *Convex hull - Graham's Scan.* Graham's scan solves the problem of finding the convex hull by maintaining a stack of candidate points while traversing a list of all of the points sorted by polar angle in counter-clockwise order. For each new point it figures out whether the new point constitutes a left turn or right turn relative to the existing candidate points. If the point makes a right turn then the point it pivots around isn't in the convex hull, so that point should be removed, and so on until the new point being considered makes a left turn. This picture should illustrate the process



The running time is $O(n \lg n)$.

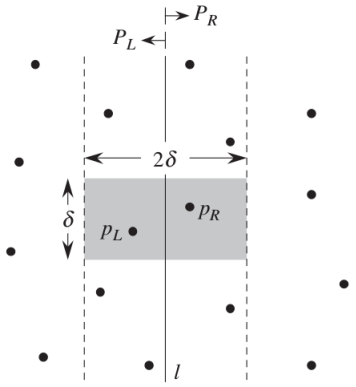
0.0.84. *Jarvis March.* Jarvis march is another convex hull finding algorithm that "gift wraps" the set of points: starting with the most south-eastern point it searchs out the point with the shallowest counter-clockwise angle i.e. "rightest" turn (which is the next point in the convex hull). Running time is $O(nh)$ where h is the number of points in the convex hull.

0.0.85. *Closest pair of points.* Given a set of points $Q = ([x_1, \ldots, x_n], [y_1, \ldots, y_n])$ find the pair of points $p_i, p_j = (x_i, y_i), (x_j, y_j)$ that minimize $\|p_i - p_j\|$. The naive algorithm (try all pairs) obviously runs in $O\left(n^2\right)$. The following divide and conquer algorithm runs faster.

Each recursive call takes a subset $P \subset Q$ and arrays X, Y of presorted points; X is sorted in monotonically increasing x-coordinate and Y in monotonically increasing y-coordinate.

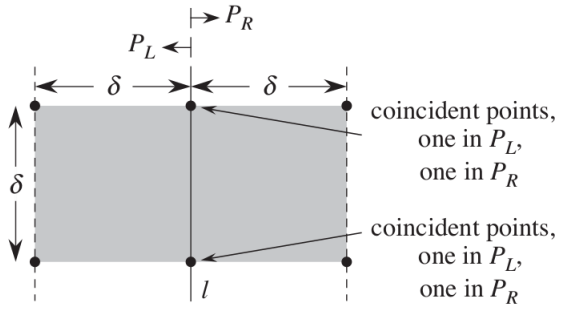
If $|P| \leq 3$ then just brute force it. Otherwise

- **Divide**: split the points according to x-coordinate into two sets $P_L = \lceil |P|/2 \rceil$ and $P_R = \lfloor |P|/2 \rfloor$. More on how to compute the corresponding X_L, Y_L and X_R, Y_R in the code.
- **Conquer**: Recurse into P_L, X_L, Y_L and P_R, X_R, Y_R computing δ_L and δ_R (the minimum distances in the recursions).
- **Combine**: the closest pair with is either $\delta = \min\{\delta_L, \delta_R\}$ or a pair points with one in P_L and one in P_R. Note that for a pair of points to be closer than δ they must each be with δ of the line that was used to create the partition P_L, P_R. The following picture shows the intuition



To find these points

(1) Construct the array Y' with only the points with x-coordinate within δ of the line separating P_L, P_R. Maintain the order of Y.
(2) For each point $p \in Y'$, consider only the following (in the order) 7 points. This is sufficient because at most 7 points can fit in the $2\delta \times \delta$ patch around any $p \in Y'$; consider the picture



The total running time is $O(n \lg n)$.