

Note	3
Part 1. Foundations	4
4. Divide-and-Conquer	4
Exercise 4.1-5	4
Problem 4-4	4
Problem 4-5	6
Problem 4-6	6
Part 2. Sorting and Order Statistics	7
6. Heapsort	7
Exercise 6.5-7	7
Exercise 6.5-9	8
Problem 6-3	8
7. Quicksort	8
Problem 7-6	8
8. Sorting in Linear Time	9
Exercise 8.2-4	9
Exercise 8.3-4	9
Exercise 8.4-4	9
Exercise 8.4-5	9
Problem 8-5	10
9. Medians and Order Statistics	10
Exercise 9.1-1	10
Exercise 9.3-5	10
Exercise 9.3-6	10
Exercise 9.3-7	10
Exercise 9.3-8	10
Exercise 9.3-9	10
Problem 9-2	11
Part 3. Data Structures	11
10. Elementary Data Structures	11
Exercise 10.1-2	11
Exercise 10.1-5	11
Exercise 10.1-6	11
Exercise 10.1-7	12
Exercise 10.4-2	12
Exercise 10.4-3	12
11. Hash Tables	12
Exercise 11.1-2	12
Exercise 11.1-3	12
Exercise 11.1-4	12
Exercise 11.2-5	13
Exercise 11.2-6	13
12. Binary Search Trees	13
Exercise 12.1-2	13
Exercise 12.1-3	13
Exercise 12.1-4	14
Exercise 12.2-2	14
Exercise 12.2-3	15
14. Augmenting Data Structures	15
Exercise 14.1-5	15
Exercise 14.1-7	15
Exercise 14.1-8	15
Exercise 14.3-3	15

Exercise 14.3-4	16
Exercise 14.3-6	16
Exercise 14.3-7	16
Problem 14-1	16
Problem 14-2	16
Part 4. Advanced Design and Analysis Techniques	17
16. Greedy Algorithms	17
Exercise 16.1-4	17
Exercise 16.1-5	17
Exercise 16.2-2	17
Exercise 16.2-3	17
Exercise 16.2-4	18
Exercise 16.2-5	18
Exercise 16.2-6	18
Exercise 16.2-7	18
Exercise 16.3-3	18
Exercise 16.3-3	18
Exercise 16.3-3	18
Problem 14-1	18
Problem 16-2	19
Part 5. Advanced Data Structures	19
21. Data Structures for Disjoint Sets	19
Exercise 21.1-3	19
Exercise 16.1-5	20
Problem 21-1	20
Problem 21-2	20
Problem 21-3	21
Part 6. Graph Algorithms	21
22. Elementary Graph Algorithms	21
Exercise 22.1-1	21
Exercise 22.1-3	21
Exercise 22.1-3	21
Exercise 22.1-5	21
Exercise 22.1-6	22
Exercise 22.1-6	22
Exercise 22.2-3	22
Exercise 22.2-7	22
Exercise 22.2-8	22
Exercise 22.2-9	22
Exercise 22.3-13	23
Exercise 22.4-2	23
Exercise 22.4-3	23
Exercise 22.5-3	23
Exercise 22.5-5	23
Exercise 22.5-6	23
Exercise 22.5-7	23
Problem 22-2	24
Problem 22-4	25
23. Minimum Spanning trees	25
Exercise 23.1-1	25
Exercise 22.1-3	25
Exercise 23.1-11	25
Exercise 23.2-2	25

Exercise 23.2-4	26
Exercise 23.2-5	26
Exercise 23.2-7	26
Problem 23-1	26
24. Single-Source Shortest Paths	26
Exercise 24.1-3	26
Exercise 24.1-5	27
Exercise 24.1-6	27
Exercise 24.2-4	27
Exercise 24.3-4	27
Exercise 24.3-6	27
Exercise 24.3-7	28
Exercise 24.3-8	28
Exercise 24.4-3	28
Exercise 24.4-4	28
Exercise 24.4-6	28
Exercise 24.4-7	28
Problem 24-2	28
Problem 24-3	29
Problem 24-3	29
25. All-Pairs Shortest Paths	29
Exercise 25.1-6	29
Exercise 25.1-9	29
Exercise 25.1-10	29
Exercise 25.2-6	29
Exercise 25.2-8	29
Problem 24-2	29

Contents

Note

I have variously stolen, plagiarized, copied, etc. from many places. Rarely I cite. This is out of pure laziness on my part. For the sake of intellectual honesty consider that absolutely none of this is my own work (it's simply a collection). Maybe eventually I'll go back and cite but probably not.

Everything is 1 indexed, despite using vaguely Pythonic syntax. This means $A[\text{len}(A)] = A[-1]$. Slicing is $A[a : b] = [A_a, A_{a+1}, \dots, A_{b-1}]$.

Where bounds checking is obviously necessary it is omitted. I assume a different memory model from Python: each entry of $B = [[]]$ is an independent list.

Ranges are represented using MATLAB notation $1 : n$.

In certain places I play fast and loose with what a dictionary is keyed on and whether a label is just a label or a pointer (in particular in the Graph Algorithms section). Also I iterate over a dictionary, which is possible with python's `dict.items()`.

The layout has large gaps intentionally. This is so pictures and diagrams follow their introduction-s/allusions/references in the text. That means if a picture/diagram is introduced and isn't on the page then it leads on the following page.

Part 1. Foundations

4. DIVIDE-AND-CONQUER

Exercise (4.1-5). Given $A = [a_1, \dots, a_n]$, how to find the subarray with the maximum positive sum? Write a linear-time, nonrecursive algorithm for the maximum-subarray problem.

Kidane's algorithm: 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 $\Theta(n)$.

```

1 Kidane-Max-Subarray(A)
2   # m_ is max
3   m_here = m_all = A[1]
4   for i = 2 : len(A):
5       m_here = max(0, m_here + A[i])
6       m_all = max(m_all, m_here)
7   return m_all

```

Note that if at $j - 1$ the subarray was empty, and hence $m_{\text{here}} = 0$ then at j it's the case that $m_{\text{here}} = A[j]$. In order to recover the actual subarray you need to keep track of whether counting is reset or subarray is extended. Easiest way to do this is using Python tricks. In general this is calling keeping "back-pointers" and works in all such cases for reconstructing the solution (forthwith omitted).

```

1 Kidane-Max-Subarray-Mod(A)
2   m_here = m_all = [0, A[1]]
3   for i = 2 : len(A):
4       # take max wrt. first entry of arguments, i.e. max(0, m_here + A[i])
5       m_here = max([0, [ ]], [m_here + A[i], m_here], key=itemgetter(1))
6       m_all = max(m_all, m_here, key=itemgetter(1))
7   return m_all

```

Problem (4-4). Fibonacci numbers. Given the generating function for Fibonacci numbers

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$$

where F_i is the i th Fibonacci number

(a) Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

Solution. Let $\mathcal{F}(z) = (0, 1, 1, 2, 3, 5, 8, 13, \dots)$ the coefficients of the terms. Then multiplication by z

$$z\mathcal{F}(z) = (0, 0, 1, 1, 2, 3, 5, 8, \dots)$$

and

$$z^2\mathcal{F}(z) = (0, 0, 0, 1, 1, 2, 3, 5, \dots)$$

Hence

$$\begin{aligned}
 z &= (0, 1, 0, 0, 0, 0, 0, \dots) \\
 z\mathcal{F}(z) &= (0, 0, 1, 1, 2, 3, 5, \dots) \\
 + z^2\mathcal{F}(z) &= (0, 0, 0, 1, 1, 2, 3, 5, \dots) \\
 \hline
 \mathcal{F}(z) &= (0, 1, 1, 2, 3, 5, 8, 13, \dots)
 \end{aligned}$$

(b) Show that

$$\mathcal{F}(z) = \frac{1}{\sqrt{5}} \left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z} \right)$$

where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Solution. Since

$$F(z) = z + zF(z) + z^2F(z)$$

we have that

$$F(z)(1-z-z^2) = z$$

or

$$F(z) = \frac{z}{1-z-z^2}$$

Factoring the denominator

$$\begin{aligned} F(z) &= \frac{z}{-\left(z + \frac{(1-\sqrt{5})}{2}\right)\left(z + \frac{1+\sqrt{5}}{2}\right)} \\ &= \frac{z}{\left(1-z\left(\frac{1+\sqrt{5}}{2}\right)\right)\left(1-z\left(\frac{1-\sqrt{5}}{2}\right)\right)} \\ &= \frac{z}{(1-\phi z)(1-\hat{\phi}z)} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1}{(1-\phi z)} - \frac{1}{(1-\hat{\phi}z)} \right) \end{aligned}$$

(c) Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} \left(\phi^i - (\hat{\phi})^i \right) z^i$$

Solution. Using the Taylor series

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n$$

we have by above

$$\begin{aligned} F(z) &= \frac{1}{\sqrt{5}} \left(\frac{1}{(1-\phi z)} - \frac{1}{(1-\hat{\phi}z)} \right) \\ &= \frac{1}{\sqrt{5}} \left(\sum_{n=0}^{\infty} (\phi z)^n - \sum_{n=0}^{\infty} (\hat{\phi}z)^n \right) \\ &= \sum_{n=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n) z^n \end{aligned}$$

(d) Use part (c) to prove that $\{F_i\} = \phi^i/\sqrt{5}$, where $\{\}$ is rounding to the nearest integer.

Solution. By comparing coefficients in the the original generating function and the re-expression

$$F(z) = \sum_{n=0}^{\infty} F_n z^n = \sum_{n=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n) z^n$$

we see that

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$$

Since $|\hat{\phi}| < 1$ it's the case that $|\hat{\phi}^n| < 1$ and hence is fractional.

Problem (4-5). Chip testing.

- (a) Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Solution. Let g be the number of good chips and $n - g \geq g$ be the number of bad chips. Then there exists a set of good chips G and a set of bad chips B such that $|G| = |B|$. The bad chips can conspire to fool the professor in the following way: they call themselves good and the actually good chips bad. The good chips of course report exactly antisymmetrically that they're good and the bad chips are bad. Therefore these two sets of chips are indistinguishable.

- (b) Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

Solution. Note that if a test is (good, good) then either both chips are bad or both good. Otherwise at least one is bad. Here's the Divide-and-Conquer algorithm:

- (1) If there's only one chip, then it must be good.
- (2) Split the chips into two-chip pairs. If the number of chips is odd let c denote the odd one out.
- (3) Test each pair. If the result is (good, good), then throw one away, otherwise throw away both.
- (4) Repeat.

The algorithm performs $\lfloor n/2 \rfloor$ pairwise tests, and keeps at most $\lceil n/2 \rceil$ chips. Now to show that at least half of the remaining chips are good each time: at a particular iteration, assume x pairs consist of two good chips, y pairs are mixed, z pairs consist of bad chips. Then there are possibilities:

- If n is even, then $g = 2x + y \geq y + 2z = b$ and $x \geq y$ implies more at least as many good chips as bad chips remain.
- If n is odd, and c is bad then $g = 2x + y \geq y + 2z + 1 = b$ and $x \geq z + 1$ (since x, z are integers). Since in fact x good chips and $z + 1$ bad chips remain, it is the case that more good chips than bad chips remain.
- If n is odd, and c is good then $g = 2x + y + 1 \geq y + 2z = b$ and $x + 1 \geq z$ (since x, z are integers). Since in fact $x + 1$ good chips and z bad chips remain, it is the case that more good chips than bad chips remain.

Therefore more good chips than bad chips remain always.

- (c) Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Solution. Use the result of (b) to find a good chip in $\Theta(\lg n)$ time and then use it to perform the other $n - 1$ comparisons.

Problem (4-6). Monge arrays.

- (a) Prove that an array is Monge iff for all $i = 1, \dots, m - 1$ and $j = 1, \dots, n - 1$ we have that

$$A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$$

Solution. If an array is Monge then it the property holds by definition. Conversely suppose an $m \times n$ array has the property. We prove that

$$A[i, j] + A[i + x, j + y] \leq A[i, j + y] + A[i + x, j]$$

for all x, y such that $1 \leq x \leq m-1$ and $1 \leq y \leq n-j$, i.e. the array is Monge. For $x = y = 1$ the property holds by assumption. First suppose $x' < m$ and $y' \leq n$ and the property holds for all x, y such that $1 \leq x \leq x'$ and $1 \leq y \leq y'$. Then it holds for $x = x' + 1$ and $y = y'$: consider $i < m - x'$ and $j \leq n - y'$. By assumption we have that

$$A[i, j] + A[i + x', j + y'] \leq A[i, j + y'] + A[i + x', j]$$

and

$$A[i + x', j] + A[i + x' + 1, j + y'] \leq A[i + x', j + y'] + A[i + x' + 1, j]$$

Summing these two implies

$$A[i, j] + A[i + x' + 1, j + y'] \leq A[i, j + y'] + A[i + x' + 1, j]$$

Similarly we can argue the case for $x' \leq m$ and $y' < n$ and thus it holds for $x = x'$ and $y = y' + 1$.

- (c) Let $f(i)$ be the index of the column containing the leftmost minimum element of row i . Prove that $f(1) \leq f(2) \leq \dots \leq f(m)$ for any $m \times n$ Monge array.

Solution. By contradiction: assume the inequality is false. Then there is some i such that $f(i) > f(i+1)$ such that $A[i, f(i+1)] > A[i, f(i)]$. Then

$$A[i, f(i+1)] + A[i+1, f(i)] > A[i, f(i)] + A[i+1, f(i+1)]$$

a contradiction.

- (d) Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A : Construct a submatrix A' of A consisting of the even-numbered rows of A . Recursively determine the leftmost minimum for each row of A' . Then compute the leftmost minimum in the odd-numbered rows of A .

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum, and its index, of the even-numbered rows is known) in $O(m+n)$ time.

Solution. Using part (c), if we know the minimum elements $f(i)$ for the even rows then for each odd $2i+1$ row we only need to check columns between $f(2i)$ and $f(2i+2)$. Hence we can compute the minima of the odd rows in time

$$\sum_{i=1}^{m/2} f(2i+2) - f(2i) + 1 = O(m+n)$$

- (e) Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \lg m)$.

Solution. The recurrence is

$$\begin{aligned} T(m) &= T(m/2) + O(m+n) \\ &= O\left(\sum_{k=0}^{\lg m} \left(\frac{m}{2^k} + n\right)\right) \\ &= O(m + n \lg m) \end{aligned}$$

Part 2. Sorting and Order Statistics

6. HEAPSORT

Exercise (6.5-7). Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.

Solution. Run a timer. To construct a FIFO make the priority key the insertion time. To construct a LIFO make the priority key $1/\text{insertion time}$.

Exercise (6.5-9). Give an $O(n \lg k)$ algorithm for constructing a sorted array from k already sorted arrays (where the total number of elements is n).

Solution. Use a MinHeap with the extract min property: construct a MinHeap from first elements in each array. Pop the the minimum element and add to a surrogate array. Replace with the next element of the array that that one came from. This way the smallest element of each of the k arrays is always in direct competition. Constructing the initial array is $O(k)$ and then each of the Extract-min operations costs $\lg k$, hence $O(n \lg k)$.

Problem (6-3). Young tableaux.

- (c) Give an algorithm to implement **Extract-Min** on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time.

Solution. $Y[1,1]$ is clearly the minimum. Pop it and replace it with the bottom right element, then “percolate down”.

- (d) Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

Solution. Insert at the bottom right, then “percolate up”.

- (e) Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

Solution. Repeatedly **Extract-Min**.

- (f) Give an $O(m + n)$ -time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Solution. Start at the top right, then you know everything below you is greater and everything to the left is smaller. If the number you’re looking for is smaller than the current entry then go left, and if the number is greater than the current entry then go down.

7. QUICKSORT

Problem (7-6). Fuzzy sorting of intervals.

- (a) Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.).

Solution. The key is that two intervals intersect (overlap) then they don’t need to be sorted. That’s where the speedup comes from. To that end here’s code to compute the intersection (if any exists) of a set $I = ((a_1, b_1), \dots, (a_n, b_n))$ of intervals

```

1  Intersection(I)
2      i = random()
3      I[-1], I[i] = I[i], I[-1]
4      a, b = I[-1][1], I[-1][2]
5      for i = 1 : len(I) - 1:
6          if a ≤ I[i][1] ≤ b or a ≤ I[i][2] ≤ b:
7              if a < I[i][1]:
8                  a = I[i][1]
9              if I[i][2] < b:
10                 b = I[i][2]
11      return a, b

```


This computes the intersection of all intervals if one exists; it does not find one! Note that $a \leq I[i][1] \leq b$ or $a \leq I[i][2] \leq b$ can be simplified down to $I[i][1] \leq b \wedge I[i][1] \geq a$, since $a_i \leq b_i$. Running time is clearly $\Theta(n)$.

Now using the model of **Quicksort** we can build a **Fuzzy-sort**: partition the input array into “left”, “middle”, and “right” subarrays, where the “middle” subarray contains intervals that overlap the intersection of all of them [the intervals] and don’t need to be sorted any further. Running time is $O(n \lg n)$ in general but if all of the intervals overlap then the recursion returns without executing anything and so only the **filters** run (which are $O(n)$).

```

1 Fuzzy-Sort(I)
2   if len(I) ≤ 1:
3     return I
4   else:
5     a, b = Intersection(A, B)
6     # first partition for similar reasons to Quicksort,
7     # in order to actually sort, i.e. everything in Iright
8     # follows everything in Ileft in the final ordering
9     # but use a as the pivot in order for the second
10    # partition to be effective
11    Ileft = filter(I, λi: i[1] ≤ a)
12    Iright = filter(I, λi: i[1] > a)
13    # find all the intervals in Ileft that overlap [a, b], but
14    # since [a, b] is an intersection it should be
15    # contained in these intervals
16    # therefore everything in Imiddle is such that [a, b] ⊆ [ai, bi]
17    Imiddle = filter(I, λi: b ≤ i[2])
18    # and Ileft-left is everything else.
19    Ileft-left = filter(I, λi: i[2] < b)
20    return Fuzzy-Sort(Ileft-left) + Imiddle + Fuzzy-Sort(Iright)

```

8. SORTING IN LINEAR TIME

Exercise (8.2-4). Describe an algorithm that, given n integers in the range 0 to k , preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \dots b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Solution. Suppose the the cumulates array constructed by **Counting-Sort** is C . Then $C[b] - C[a]$ is the answer to the query.

Exercise (8.3-4). Show how to sort n integers in the range 0 to n^3-1 in $O(n)$ time.

Solution. n base n the number $n^3 - 1$ are two digits numbers e.g. $1000_n = 1 \times n^3 + 0n^2 + 0n + 0 \times 1$. So we make 4 passes using radix sort

$$\Theta(4(n + n)) = O(n)$$

Exercise (8.4-4). We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, \dots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin.

Solution. A differential ring of area on the unit circle is $dA = 2\pi r dr$ so using bucket sort we can divide up the buckets according this scaling.

Exercise (8.4-5). Suppose that we draw a list of n random variables X_1, \dots, X_n from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Solution. $Y = P(X_i)$ is uniformly distributed.

Problem (8-5). Jugs.

- (a) Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

Solution. Test every blue jug against every red jug.

- (b) Skip.

- (c) Show how to match the jugs in $O(n \lg n)$ time.

Solution. You could match up the jugs by sorting each set and lining them up. Too bad you can't compare red jugs against red jugs right? But you can just use the **Quicksort** model and with blue jugs being pivots for red jugs and red jugs being pivots for blue jugs.

9. MEDIAN AND ORDER STATISTICS

Exercise (9.1-1). Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Solution. Tournament style to determine minimum: comparing all pairs costs $n/2$, compare all winners of the first round costs $n/4$, etc. In total this is $n - 1$ comparisons. The only way in which the second smallest element is not in the final round is it was eliminated in an earlier round. Therefore keep track of all of the elements that the smallest element “played” against, which is $\lceil \lg n \rceil$, and find the smallest of them. This costs $\lceil \lg n \rceil - 1$ comparisons.

Exercise (9.3-5). Suppose that you have a “black-box” worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Solution. Suppose the rank you're looking for is r and the number of elements is n . Use binary search: find the median, then if the order statistic is higher than the median find the $r - \lfloor n/2 \rfloor$ order statistic of the elements larger than the median, and if the order statistic is lower then find the rank r statistic of the elements smaller than the median, and so on.

Exercise (9.3-6). The k th quantiles of an n -element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$ -time algorithm to list the k th quantiles of a set.

Solution. If k is even then there are $k - 1$ (an odd number) of “pivots” and one of them is the median. Find the median, partition, then solve the subproblems. If k is odd do the same thing but be more careful.

Exercise (9.3-7). Describe an $O(n)$ -time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S .

Solution. Find the median, then subtract the median from every element, then find the k th order statistic (and in doing so partition).

Exercise (9.3-8). Let $X[1 \dots n]$ and $Y[1 \dots n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$ -time algorithm to find the median of all $2n$ elements in arrays X and Y .

Solution. The median all of $2n$ elements is always in between the median of each array (by value). Compute the medians in $O(1)$ time. If they're equal return them. Otherwise recurse to either the leftside or rightside of each array depending on which median is larger than which.

Exercise (9.3-9). Given the x - and y -coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Solution. The median is the element that minimizes the L_1 norm, i.e. the sum of distances.

Problem (9-2). Weighted median.

- (a) Argue that the median of x_1, \dots, x_n is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, \dots, n$.

Solution. This is trivially true (algebraically).

- (b) Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

Solution. Sort then sum weights, in order of increasing elements, until you exceed $1/2$.

- (c) Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

Solution. Call $t = 1/2$ the target. Find the median (and in doing so partition around it) and compute the sum of the weights in the “lower” half. If they sum to t then return the median. If the exceed then compute the median in the “lower” half. If the sum is less than t then compute the median in the “top half” but with target being t minus the sum you just computed.

- (d) Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

Solution. This is true for the same reason the median minimizes the L_1 norm.

- (e) Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the Manhattan distance given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Solution. Since the components of the distance “vector” are decoupled you can just do median in each coordinate, i.e. take the median of all of the x_i and the median of all of the y_i .

Part 3. Data Structures

10. ELEMENTARY DATA STRUCTURES

Exercise (10.1-2). Explain how to implement two stacks in one array $A[1 \dots n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n . The PUSH and POP operations should run in $O(1)$ time.

Solution. Have one stack grow from the left side of the array and the other stack from the right side of the array (store their ends). When they collide then they’re both “full”.

Exercise (10.1-5). Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a deque (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$ -time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

Solution. Either use a circular list with the head and tail linked or an array using mod to update indices that keep track of the back and front and checking for collision of the indices.

Exercise (10.1-6). Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Solution. Call the two stacks “inbox” and “outbox”. Push to one stack and pop from the other. If the “outbox” stack is empty then pop everything from the inbox stack and push to outbox stack. They’ll be pushed in reverse order and pops will produce the correct behavior. Amortized time is $O(n)$.

Exercise (10.1-7). Show how to implement a stack using two queues. Analyze the running time of the stack operations.

Solution. Exactly like 10.1-6.

Exercise (10.4-2). Write an $O(n)$ -time recursive procedure that, given an n -node binary tree, prints out the key of each node in the tree.

Solution. DFS (or BFS).

Exercise (10.4-3). Write an $O(n)$ -time nonrecursive procedure that, given an n -node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

Solution. This is of course DFS but written iteratively. Assume the tree is represented by a `dict()` with 'leftchild' and 'rightchild' keys (whose corresponding values are the nodes). Since we're traversing a tree we don't need to check for backpointers (i.e. don't need to mark visited).

```

1 DFS(T)
2   stk = [T]
3   while len(stk) > 0:
4       next = stk.pop()
5       if next is not None:
6           print next
7           stk.append(next['leftchild'])
8           stk.append(next['rightchild'])

```

11. HASH TABLES

Exercise (11.1-2). A **bit vector** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

Solution. Assign an index to each element. Use the bit vector to indicate membership by settings bits by initializing all bits to 0 and then setting to 1 when inserting (deletion is resetting to 0).

Exercise (11.1-3). Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.

Solution. Use chaining. For DELETE set the pointer to a NULL.

Exercise (11.1-4). We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations INSERT, DELETE, and SEARCH should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. Hint: Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.

Solution. Use the stack to store the actual values. Let *huge* be the array and *stack* be the stack. To insert an element with key x append x to the stack and store in *huge*[x] the length of the stack (i.e. the index of x in the stack). To search for an element y (i.e. verify membership) check that *huge*[y] \leq len(*stack*) and that *stack*[*huge*[y]] = y . To delete an element x swap the top of the stack with the element to be deleted and update relevant entries in *huge*:

$$\begin{aligned}
 \text{stack}[\text{huge}[x]] &= \text{stack}[-1] \\
 \text{huge}[\text{stack}[-1]] &= \text{huge}[x] \\
 \text{huge}[x] &= \text{None}
 \end{aligned}$$

and pop the stack.

Exercise (11.2-5). Suppose that we are storing a set of n keys into a hash table of size m . Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

Solution. Pigeonhole principle.

Exercise (11.2-6). Suppose we have stored n keys in a hash table of size m , with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$, where $\alpha = n/m$.

Solution. The keyword being randomly (not just any). Pick a random bucket, which will have k elements, then pick an index i from $1, \dots, L$. Reject if $i > k$ and draw i again. This is essentially rejection sampling the array, i.e. how to uniformly random pick an element from $1, \dots, k$ if you can only generate random numbers from 1 to L . The expected number of elements k is equal to the load and so

$$P(i \leq k) = \frac{(n/m)}{L}$$

and hence expected number of times before success is

$$\frac{1}{\frac{(n/m)}{L}} = \frac{L \cdot m}{n}$$

(since “success” is a geometric random variable with probability of success being $P(i \leq k)$). Picking the initial bucket doesn’t “cost” anything, hence combined with time L to traverse we get

$$O\left(L + L \frac{m}{n}\right) = O\left(L \cdot \left(1 + \frac{1}{\alpha}\right)\right)$$

expected running time.

12. BINARY SEARCH TREES

Exercise (12.1-2). What is the difference between the binary-search-tree property and the min-heap property?

Solution. The min-heap property says parents should be larger than both of its children, while the binary-search property says greater than left-child but not right-child.

Exercise (12.1-3). Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.)

Solution. Both implementations are annoying as hell (and disagree that the stack-less form is the least bit elegant).

For the implementation with the stack the thing to remember is that only none-None nodes should get pushed to the stack and a pointer should be used to keep track of which node we’re actually on. Also since we’re using both the stack and the pointer to keep track of things, the loop invariant is that either one (not necessarily both) are not None.

```

1 In-Order-Stack(T)
2   stk = [ ]
3   crnt = T
4   while crnt or len(stk) > 0:
5       if crnt:
6           # push the last node that's
7           # legit
8           stk.append(crnt)

```

```

9      crnt = crnt['lchild']
10     # crnt == None so the last node
11     # on the stack has no left child
12     else: # len(stk) > 0
13         p = stk.pop()
14         print p.val
15         crnt = p['rchild']

```

Fof the implementation without the stack you need parent pointers in order (get it...) to be able to discover which child you're in. Also don't forget the first thing you need to check is if you're coming back up from the right child.

```

1  In-Order-No-Stack(T)
2      prev = crnt = T
3      while True:
4          prev = crnt
5          if prev == crnt['rchild']:
6              # come back up from the
7              # right child
8              if crnt['prnt'] != None:
9                  crnt = crnt['prnt']
10             else:
11                 break
12         elif crnt['lchild'] == None or prev == crnt['lchild']:
13             print crnt.val
14             if crnt['rchild'] != None:
15                 crnt = crnt['rchild']
16             elif crnt['prnt'] != None:
17                 crnt = crnt['prnt']
18             else:
19                 break
20         else:
21             crnt = crnt['lchild']

```

Exercise (12.1-4). Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

Solution. Recursive implementations are easy; iterative implementations are hard. I'll just comment on how these differ from **In-Order-Stack** (for stack-less it's obvious): to perform an Pre-Order traversal move the print statement into the first branch of the conditional (before reassigning *crnt*). Printing out a Post-Order traversal is actually easy: just use **DFS** from 10.4-3 and reverse the order.

Exercise (12.2-2). Write recursive versions of **TREE-MINIMUM** and **TREE-MAXIMUM**.

Solution. I have no idea why you would insist on a recursive implementation of both of these since they're easily implemented iteratively.

```

1  Tree-Min(T)
2      ptr = T
3      while ptr['lchild']:
4          ptr = ptr['lchild']
5      return ptr

```

```

1  Tree-Max(T)
2      ptr = T
3      while ptr['rchild']:
4          ptr = ptr['rchild']

```

```
5 | return ptr
```

Exercise (12.2-3). Write the TREE-PREDECESSOR procedure.

Solution. If a node's left-subtree exists then the node's predecessor is the maximum of that subtree. If the node's subtree doesn't exist then the node's predecessor is "on top" of it: it's the "lowest" ancestor of the node whose right child is also an ancestor of the node. Basically go up and cut the first left that you can.

```
1 Predecessor(x)
2   # assume x is a pointer to the node
3   # (which could be found by a search)
4   if x['lchild']:
5       return Tree-Max(x['lchild'])
6   else:
7       ptr = x
8       prnt = x['prnt']
9       # while you haven't cut a left
10      while ptr and ptr == prnt['lchild']:
11          ptr = prnt
12          prnt = ptr['prnt']
13      return prnt
```

Just for completeness note that successor is symmetric: either minimum of the right-subtree or go up and cut a right.

14. AUGMENTING DATA STRUCTURES

Exercise (14.1-5). Given an element x in an n -node order-statistic tree and a natural number i , how can we determine the i th successor of x in the linear order of the tree in $O(\lg n)$ time?

Solution. Find the rank r of x , and then find the element whose rank is $r + i$.

Exercise (14.1-7). Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

Solution. Recall that how "far" an element is ahead of its rank (in the array) is how many inversions there are that include that element. If an element is "behind" its rank then it will appear as an inversion for other elements. So find an element's rank and how much its position differs from its rank. If it differs by a negative amount then don't count it. Otherwise it's $i - r(i)$, where i is the index of the element and $r(i)$ is its rank.

Exercise (14.1-8). Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$ -time algorithm to determine the number of pairs of chords that intersect inside the circle.

Solution. Go around the circle counter-clockwise. every time you reach an endpoint "label" it and push it to an array. Also label it's polar opposite something else (in order to distinguish the line as already being encountered). When you run out of unlabeled points go around counter-clockwise, starting from the same point as where you started prior, but now push only the endpoints that have been labeled opposites to a different array. Now treat one array as the correct order and the other one as having inversions (count the number of inversions using the algorithm in 14.1-7).

Exercise (14.3-3). Describe an efficient algorithm that, given an interval i , returns an interval overlapping i that has the minimum low endpoint, or **None** if no such interval exists.

Solution. Given **Interval-Search** we can find if there's any overlapping interval, then repeatedly **Interval-Search** in the left-subtree.

Exercise (14.3-4). Given an interval tree T and an interval i , describe how to list all intervals in T that overlap i in $O(\min\{n, k \lg n\})$ -time, where k is the number of intervals in the output list.

Solution. Use a depth-first search while check the same condition that **Interval-Search** checks, i.e. that $x.\text{left.max} \geq i.\text{low}$ (and only exploring the left child is this is the case).

Exercise (14.3-6). Show how to maintain a dynamic set Q of numbers that supports the operation **MIN-GAP**, which gives the magnitude of the difference of the two closest numbers in Q . Make the operations **INSERT**, **DELETE**, **SEARCH**, and **MIN-GAP** as efficient as possible, and analyze their running times.

Solution. When you insert into a binary search tree you can keep track of predecessor and successor at no extra cost: the predecessor is the last right to left turn you made during insertion and the successor is the last left to right turn you made during insertion. The narrowest gap between an element and any other element is the minimum of the gap between elements and their predecessor or successor. So you can insert into a binary search tree and keep track of all the minimum gaps (between each element and either its predecessor or successor). Deletion simply recomputes the predecessor and successor of affected elements.

Exercise (14.3-7). VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x - and y -axes), so that we represent a rectangle by its minimum and maximum x - and y -coordinates. Give an $O(n \lg n)$ -time algorithm to decide whether or not a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect.

Solution. Use an Interval Tree and sweep a line in both directions adding and removing intervals as they appear. On adding each interval check if there's a collision using **Interval-Search**.

Problem (14-1). Point of maximum overlap.

- (a) Show that there will always be a point of maximum overlap that is an endpoint of one of the segments.

Solution. Skip.

- (b) Design a data structure that efficiently supports the operations **INTERVAL-INSERT**, **INTERVAL-DELETE**, and **FIND-POM**, which returns a point of maximum overlap. (Hint: Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

Solution. Keep track of the sum of negatives and positives on the left side of a node and the right side of a node. A surplus of negatives means the endpoint corresponding to the node you're at is inside that many intervals (on the left). A surplus of positives on the right means the same thing. Keep a pointer to the current maximum overlap by simply checking against the current maximum stored somewhere every time a new interval is inserted (can't figure out how to compute it from the tree without traversing entire tree). It should be simple to update augmentations during insertion and deletion.

Problem (14-2). Josephus permutation.

- (a) Suppose that m is a constant. Describe an $O(n)$ -time algorithm that, given an integer n , outputs the (n, m) -Josephus permutation.

Solution. Use a circularly linked-list and just delete nodes while skipping forward m times. Running time is $O(mn)$ but m is a constant so $O(n)$. Alternatively use the recurrence relation

$$Josephus(n, k) = [(Josephus(n-1, k) + k - 1) \bmod n] + 1$$

The reasoning here is that once the first element of n elements is deleted there are $n-1$ elements remaining and we need the last survivor of those $n-1$ elements. The $k-1$ is

because we need to count from where the first element is deleted, i.e. the $(k - 1)$ th position. The $\bmod n$ is to wrap around and $+1$ is shifting to 1 counting because using mod makes it as if we're counting from 0.

- (b) Suppose that m is not a constant. Describe an $O(n \lg n)$ -time algorithm that, given integers n and m , outputs the (n, m) -Josephus permutation.

Solution. Use an order statistic tree and delete the k th ranked element, then delete the $[(2k) \bmod n] + 1$ ranked element, and so on.

Part 4. Advanced Design and Analysis Techniques

16. GREEDY ALGORITHMS

Exercise (16.1-4). Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the **interval-graph coloring problem**. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

Solution. Sort by start time and then allocate to the first available lecture hall. If no lecture hall is available then open a new one. What does this have to do with graph coloring?

Exercise (16.1-5). Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i . The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Solution. Pick that activities that are max over value of compatible jobs. This problem has the same Bellman equation as for resource allocation. Either the next job contributes to an optimal solution or it doesn't. If it does then the sub problem is the maximum value job allocation strategy over jobs compatible with it. If it doesn't then just use prior value.

Exercise (16.2-2). Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Solution. The Bellman equation is

$$k(w, j) = \max \{k(w - w_j, j - 1) + v_j, k(w, j - 1)\}$$

i.e. consider a subset $1, \dots, j$ of the n items and either include the j th item or not. If you include then the value of the knapsack is the value of the optimal knapsack lighter by w_j plus the value of the j th item v_j . If you don't include it then the value of the knapsack is just the value of the knapsack of weight w but comprised of only a subset of the items $1, \dots, j - 1$.

Exercise (16.2-3). Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

Solution. Take the lightest items of course.

Exercise (16.2-4). Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations. The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

Solution. Go to the last stop before m miles, refill, repeat.

Exercise (16.2-5). Describe an efficient algorithm that, given a set $\{x_1, \dots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Solution. Sort the points. Use the first point as the left endpoint of the first unit-length interval. For the next point ask the question: is it covered? If not repeat.

Exercise (16.2-6). Show how to solve the fractional knapsack problem in $O(n)$ time.

Solution. Take items according to value density: value/weight.

Exercise (16.2-7). Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Solution. Pair the largest number in A with the largest numbers in B .

Exercise (16.3-3). What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

Solution. Because of the recurrence relation the Huffman tree skews to the right.

Exercise (16.3-6). Suppose we have an optimal prefix code on a set $C = \{1, \dots, n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n-1 + n \lceil \lg n \rceil$ bits. (Hint: Use $2n-11$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

Solution. A full binary tree is uniquely defined by its preorder traversal. Then represent each of the numbers using $\lg n$ bits in the order they're seen by the preorder traversal.

Exercise (16.3-7). Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

Solution. Just create tri-valent nodes and then pop 3 at a time from the priority-queue.

Problem (16-1). Coin changing.

- (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

Solution. Exchange for as many of the decreasingly smaller denomination as possible.

- (b) Suppose that the available coins are in the denominations that are powers of c , i.e., the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

Solution. This is just base c representation of the amount of money (and hence obviously optimal).

- (c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n .

Solution. $(4, 3, 1)$ fails on 6.

- (d) Give an $O(nk)$ -time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Solution. Suppose the coins come sorted in decreasing order so $c_1 > c_2 > \dots > c_k = 1$. Let $C(i, j)$ be the optimal number of coins used to make change for $i\text{¢}$ using only coins j, \dots, 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 $(i - c_j)\text{¢}$ needs to be changed, potentially using the coin j again.

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

Problem (16-2). Suppose you are given a set $S = \{a_1, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the completion time of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n) \sum_{i=1}^n c_i$.

- (a) Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Solution. Schedule them in order of shortest running time.

- (b) Suppose now that the tasks are not all available at once. That is, each task cannot start until its release time r_i . Suppose also that we allow preemption, so that a task can be suspended and restarted at a later time. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Solution. At every release it's like you've got some number of jobs competing (where the job running currently counts as a job that has some amount of time left, and similarly all the other paused jobs). Just run the one that ends soonest and reassess every release time. You can use a min-heap. Each job could be pushed and popped a total of k times, where k is the number of distinct release times so total running time is $O(k^2 \log n)$.

Part 5. Advanced Data Structures

21. DATA STRUCTURES FOR DISJOINT SETS

Exercise (21.1-3). During the execution of **CONNECTED-COMPONENTS** on an undirected graph $G = (V, E)$ with k connected components, how many times is **FIND-SET** called? How many times is **UNION** called? Express your answers in terms of $|V|, |E|, k$.

Solution. Worst case both linear (the rub is that **FIND-SET** is linear time too without path compression), and union is bad too.

Exercise (21.1-4). Suppose that we wish to add the operation $\text{PRINT-SET}(x)$, which is given a node x and prints all the members of x 's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that $\text{PRINT-SET}(x)$ takes time linear in the number of members of x 's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

Solution. Connect all the nodes up like a linked list. Representative node points to first child. Then when joining the other set is "spliced in": call the pointer *next*. When splicing in the next disjoint set take *next* of the joined-to set and point it at the *next* of the representative of the joined in set, then put the *next* of the joined-in set to point to the original *next* of the joined-to set.

Problem (21-1). Off-line minimum.

We want to determine the m th minimum that would be extract from a set of numbers with maximum n , given a particular set of insertion. Partition the sequence of actions into I_j and E_m , where I_j is all of the insertions between extractions E_{j-1} and E_j . I_j might be empty (consecutive extractions). For each I_j insert corresponding keys into a set K_j . Then the algorithm is

```

1  Offline-Min( $m, n$ )
2    for  $i = 1 : n$ :
3      determine  $j$  s.t.  $i \in K_j$ 
4      if  $j \neq m + 1$ :
5         $extracted[j] = i$ 
6         $\ell = \min_{j' > j} \{K_{j'} \text{ exists}\}$ 
7         $K_\ell = K_j \cup K_\ell$ 
8      del  $K_j$ 
9  return  $extracted$ 

```

- (a) Argue that the array *extracted* returned by **Offline-Min** is correct.

Solution. The algorithm works in a kind of inverted sense. Instead of finding the minimum of the set at particular extraction it finds the extraction at which a number is the minimum (that's why it counts up from 1 to n). Then in order to keep the sets of inserted numbers in contention (but prevent the same sequence "moment" from being counted twice) it unions the sets with the next possible "moment". Think about what would happen if lines 7,8 weren't executed: the same K_j might get contain two consecutive minima (for example for $\{3, 4\} \subset K_j$) and $extracted[j]$ would be overwritten.

- (b) Describe how to implement **Offline-Min** efficiently with a disjoint- set data structure.

Solution. Python code appears in `disjoint_forests.py` in the `code/ch21` directory.

Problem (21-2). Depth determination.

- (a) Skip.
 (b) Skip.
 (c) Show how to modify **FIND-SET** to implement **FIND-DEPTH**. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

Solution. Let the pseudo-distance be the distance from each node to its parent. Along with path compression update the pseudo-distance of a node to be the sum of the pseudo-distances of all of its ancestors'. This could be implemented by using the path-compression trick (unwinding the stack) but return a tuple that includes the pseudo-distance and using it to sum (something like a `fold`). For example suppose $x.d[v_i]$ to be the pseudo-distance of node v_i . Then the return value of **Find-Set** could be modified to be $(x.p, x.d[v_i])$ and path-compression line could be modified to be

$$\begin{aligned}
 x.p, d &= \text{Find-Set}(x.p) \\
 x.d[v_i] &= x.d[v_i] + d
 \end{aligned}$$

- (d) Show how to implement **Graft**(r, v), which combines the sets containing r and v , by modifying the **UNION** and **LINK** procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i .

Solution. The only thing to be careful about is how to update the pseudo-distance. Let the path from v to the root be $(v = v_0), v_1, v_2, \dots, (v_k = w)$, where w is the root. If $\text{rank}(r) < \text{rank}(w)$ then **UNION** r 's set to w 's set (i.e. make r 's parent pointer point to w) and update the pseudo-distance as described in part (c), not forgetting to include $d[v]$. If $\text{rank}(r) \geq \text{rank}(w)$ then make w 's parent pointer point to r , and update r 's pseudo-distance skipping $d[v]$. Furthermore update w 's pseudo-distance to be $d[w] - d[r]$. What's happening here? For the purposes of maintaining time bounds we, of course, want to union the smaller set to the larger set (so that fewer parent pointers end up being updated). But then the pseudo-distance becomes wrong for the root (which becomes a set-child of r 's set). We compensate by biasing w 's pseudo-distance (note it becomes negative). v 's set doesn't need to be updated since it already has w as its root.

Problem (21-3). Tarjan's Algorithm.

Solution. Write up appears in `tarjan1ca.pdf` in the `code/ch21` directory.

Part 6. Graph Algorithms

22. ELEMENTARY GRAPH ALGORITHMS

Exercise (22.1-1). Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Solution. Out-degree is easy. Constant time if you for example store the length of the adjacency list. In-degree is much harder $|V|^2$ for each vertex, so $|V|^3$. To solve the problem I would just construct the adjacency-matrix representation first pass (easy: just traverse the adjacency-list and mark entries in the matrix). Then for each following query it would be just a sum across the v th column.

Exercise (22.1-3). The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V \mid (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G , for both the adjacency-list and adjacency-matrix representations of G . Analyze the running times of your algorithms.

Solution. For the adjacency-matrix representation the construction is constant time essentially (just traverse the matrix in the "opposite" direction). For the adjacency-list representation you could create a hash-table of vertices then traverse all of the adjacency-lists and hash on entries (vertices) then push to linked lists of those vertices the owner of that linked-list.

Exercise (22.1-4). Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$ -time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

Solution. Hash on (u, v) . If there's a collision then don't insert in the new adjacency list.

Exercise (22.1-5). The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v . Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G . Analyze the running times of your algorithms.

Solution. G^2 for adjacency matrix is just the normed square of the matrix because taking the dot product "turns on" connections to the 2 hop nodes. For adjacency-list I guess you have to just traverse each list. If you treat each adjacency list as a set you can do set union but those are still linear ($n^2 \equiv O(|V| |E|)$).

Exercise (22.1-6). Most graph algorithms that take an adjacency-matrix representation as input require time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a universal sink - a vertex with in-degree $|V| - 1$ and out-degree 0 - in time $O(V)$, given an adjacency matrix for G .

Solution. Start at top left of the adjacency-matrix. Go right until you hit a 1. If you never hit one then there's no universal sink since that vertex is disconnected from all other vertices. When you hit a one continue down until you hit a zero, then turn right and search for a 1 again. If you reach the bottom row then the column where you hit the bottom is the only universal sink candidate. Why only? Because if there were another vertex with all 1s in its columns it would have two ones in its row). All that remains is to check that entire column for 1s. If you hit the right wall then there's no universal sink. There are two edge cases: if you hit the right wall on the first row the first vertex might be a universal sink. Finally you also need to check that the row of the universal sink vertex has only one 1 (otherwise there might be two "universal sinks").

Exercise (22.1-7). The incidence matrix of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix B such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \\ 1 & \text{if edge } j \text{ enters vertex } i \\ 0 & \text{otherwise} \end{cases}$$

Describe what the entries of BB^T represent.

Solution. The only non-zero entries are on the diagonal and they're the total degree (in degree - out degree). Nope that's wrong. Off diagonal entries exist, because nodes connected to each other have opposite sign entries in b_{ij} . So off diagonal entries is number of edges connecting vertex i to j .

Exercise (22.2-3). Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if lines 5 and 14 were removed.

Solution. This works if you simply only push vertices that haven't been visited to the queue.

Exercise (22.2-7). There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$ -time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Solution. 2 color the graph: set the vertices in each round of the BFS to be the opposite color of the parent. If node is already colored same color then no such coloring is possible.

Exercise (22.2-8). The **diameter** of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Solution. Start at any node and do a BFS. Pick the node u that's the farthest, then do another BFS from that node picking the node v that's the farthest from u .

Exercise (22.2-9). Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$ -time algorithm to compute a path in G that traverses each edge in E exactly once in each direction.

Solution. Use the same walk as in tarjan's algorithm:

```

1 Walk(u)
2   print u
3   for v ∈ u['children']:
4       Walk(v)
5   print u

```

Exercise (22.3-13). A directed graph is **singly connected** if $u \rightsquigarrow v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

Solution. Run DFS from each vertex. If you discover any back edges then there's a cycle and therefore two paths. Running time is $O(VE)$.

Exercise (22.4-2). Give a linear-time algorithm that takes as input a directed acyclic graph and two vertices s and t , and returns the number of simple paths from s to t in G .

Solution. Dynamic programming. The number of paths from s to t is the number of paths from s to each of t 's parents, summed across the parents, and on. Do a topological sort first in order to figure out which vertices are potentially t 's parents. The number of paths from s to s is 0. The number of paths from s to the next vertex in the topological sort is 1 if there's an edge from s to that vertex, etc.

Exercise (22.4-3). Give a linear-time algorithm that takes as input a directed acyclic graph and two vertices s and t , and returns the number of simple paths from s to t in G .

Solution. Do a DFS search but quit as soon as you hit a back edge. If there's a cycle then there will be greater than $|V|$ edges and so the DFS will # stop after $|V|$ edges. If there's no cycle then the DFS will stop naturally after $|V| - 1$ edges.

Exercise (22.5-3). Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?

Solution. Because who says there's no edge out of the first closed vertex to another component?

Exercise (22.5-5). Give an $O(V + E)$ -time algorithm to compute the component graph of a directed graph. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

Solution. Compute the strongly connected component graph, then add edges as such: add edge (u', v') if $u \in C_{u'}, v \in C_{v'}$ and there is an edge $(u, v) \in E$.

Exercise (22.5-6). Given a directed graph G , explain how to create another graph $G' = (V, E')$ such that (a) G' has the same strongly connected components as G , (b) G' has the same component graph as G , and (c) E' is as small as possible. Describe a fast algorithm to compute G' .

Solution. Compute strongly connected components then use a DFS to explore the connected components, and return $E' \subset E$ leaving out edges that aren't in the DFS.

Exercise (22.5-7). A directed graph G is **semiconnected** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

Solution. Something something transpose? Run DFS on G then DFS on G' . If both DFSs return the same list of visited vertices then G is semiconnected. I guess you could do that but a better way is apparently to compute the SCC graph (which is a dag) then topologically sort the SCC, then check if there's an edge between each consecutive pair of vertices (u, v) (pairs of SCC components) in the topological sort. If one is missing then (since topological sort and dag) there will be no connections from the v .

Problem (22-2). Articulation points, bridges, and biconnected components.

Let G be a connected, undirected graph. An **articulation point** of G is a vertex whose removal disconnects G . A **bridge** of G is an edge whose removal disconnects G . A **biconnected component** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G .

- (a) Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G .

Solution. Well obviously if it's an articulation point it has to have at least two out edges. If it has two children in the DFS tree that means its second child wasn't reached by path from its first child (other wise that second child would have been someone else's child) and therefore the root is an articulation point (because removing it would cut off contact between the first child and the second).

- (b) Let v be a *nonroot* vertex of G . Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v .

Solution. If every child has such a back edge (or a descendent of s has such a back edge) to an ancestor of v then clearly we can reach any child and any descendant by going through the ancestor instead of v .

- (c) Let

$$\begin{aligned} v.low &= \min \begin{cases} v.d \\ w.d & (u, w) \text{ is a back edge for some descendant } u \text{ of } v \end{cases} \\ &= \min \begin{cases} v.d \\ \min_{y:(v,y) \in E} y.low \\ w.d & (v, w) \text{ is a back-edge} \end{cases} \end{aligned}$$

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ time.

Solution. $v.low$ is something like the earliest time a vertex v could've been discovered during some DFS (not necessarily the particular DFS we ran). For leaves of G_π there are no descendants so $v.low$ is straightforward (either G_π contains a backedge (v, w) or not). Note that we include v as a descendant of v in order to consider backedges from v itself. For vertices v in the tree, if u is a descendant of v in G_π , we know that $u.d > v.d$. Therefore if there's some backedge (u, w) and $w.d < v.d$ we also have that $w.d < u.d$, and hence we will have set $u.low = w.d$. Therefore starting at the leaves of G_π working up (towards the root) everything will have been computed in time to compute $v.low$.

- (d) Show how to compute all articulation points in $O(E)$ time.

Solution. For a vertex v if there is a subtree rooted at a child of v which **does not** have a back edge connecting to vertex with a smaller discovery time than $v.d$, then v is an articulation point. I.e., is there a child y of v that cannot reach a vertex visited before v ? If yes, then removing v will disconnect y .

Run a DFS to compute G_π and compute $v.low$ for all vertices using the algorithm in part (c). By part (a) we can test whether the root of the G_π is an articulation point. By part (b), any non-root vertex v is an articulation point iff it has at least one child s such that neither s , nor any of its descendants, has a backedge to a proper ancestor of v . By part (c) this is true iff $s.low \geq v.d$. We can check this in time proportional to the number of children of v in G_π , so over all non-root vertices, this takes $O(V)$ time. Thus, the total time to find all the articulation points is $O(E)$.

- (e) Skip.

- (f) Show how to compute all the bridges of G in $O(E)$ time.

Solution. Use DFS to find simple cycles. Any edges on a simple cycle can be removed.

Problem (22-4). Reachability.

Let G be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, \dots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V \mid u \rightsquigarrow v\}$ be the set of vertices that are reachable from u . Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimal. Give an $O(V + E)$ -time algorithm that computes $\min(u)$ for all vertices $u \in V$.

Solution. One solution is to compute the strongly connected components of the graph and erase all but the smallest label vertex in each component C ; let this vertex be denoted $w(C)$. For every edge (u, v) with u not in C and v in C add an edge (u, w) . For every edge (v, u) with v in C and u not in C add an edge (w, u) (this process is called contracting C to a single vertex w). The resulting graph is a DAG. This DAG can be computed in $O(V + E)$ time (since strongly connected components can be computed in $O(V + E)$ time). So we reduced the problem to the same problem on a DAG. Now it is simple: traverse the graph in reverse topological order. Initially every vertex has $\min(u) = u$. For every vertex u look at its outgoing edges (u, v) and update

$$\min(u) = \min \{ \min(v) \mid (u, v) \}$$

Since we traverse vertices in reverse topological order all outgoing vertices (u, v) of u will have already found their final label $\min(v)$.

A simpler way (if you know which vertices have which number) simply set all of the vertices with incoming edges into a vertex to have the number equal to L of the vertex coming into, in order of increasing $L(u)$, i.e. **in order of increasing $L(u)$** compute for each v such that $(v, u) \in E$

$$\min(v) = \min_{u:(v,u) \in E} L(u)$$

In order of increasing $L(u)$ puts a sort of topological ordering on the computation of minimum, enabling a DP algorithm. That is, first perform BFS(1) in G^T , where “1” is the vertex u with $L(u) = 1$; this will visit all vertices reachable from “1” in G^T (that is, which can reach 1 in G) and set their $\min(u) = 1$. Then find the next smallest vertex that has not been reached in the previous BFS and start BFS from it, and so on.

23. MINIMUM SPANNING TREES

Exercise (23.1-1). Let (u, v) be a minimum-weight edge in a connected graph G . Show that (u, v) belongs to some minimum spanning tree of G .

Solution. Assume that it doesn't. Then for no minimum spanning tree is (u, v) in E . So take an arbitrary such tree. Adding (u, v) creates a cycle from which we can remove any edge and preserve connectivity of the graph. therefore remove any edge other than (u, v) . This produces a potentially different minimum spanning tree. Contradiction.

Exercise (23.1-3). Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Solution. Remove the edge and you disconnect the graph. So it's clearly a safe edge. If it weren't light then we would have picked a lighter edge to be in the minimum spanning tree and the original one wouldn't have been a minimum spanning tree.

Exercise (23.1-11). Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of one of the edges not in T . Give an algorithm for finding the minimum spanning tree in the modified graph.

Solution. Add the edge to create a cycle then DFS to find the edge on that cycle of highest weight.

Exercise (23.2-2). Suppose that we represent the graph G as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

Solution. Just keep finding the minimum weight edge.

Exercise (23.2-4). Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W ?

Solution. Use radix sort to sort edges by weight.

Exercise (23.2-5). Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W ?

Solution. Use radix sort to sort edges by weight.

Exercise (23.2-7). Suppose that a graph G has a minimum spanning tree already computed. How quickly can we update the minimum spanning tree if we add a new vertex and incident edges to G ?

Solution. Throw in all the edges adjacent on the new vertex and run either Kruskal's or Prim's on the old MST + new graph.

Problem (23-1). Second-best minimum spanning tree.

Let G be an undirected, connected graph whose weight function is $w : E \rightarrow \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct. We define a second-best minimum spanning tree as follows. Let \mathcal{T} be the set of all spanning trees of G , and let T' be a minimum spanning tree of G . Then a **second-best minimum spanning tree** is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$

- (a) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique. .

Solution. Something something unique weights.

- (b) Let T be the minimum spanning tree of G . Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G .

Solution. Isn't this like literally the definition of second-best minimum spanning tree?

- (c) Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T . Describe an $O(V^2)$ -time algorithm that, given T , computes $\max[u, v]$ for all $u, v \in V$.

Solution. Just do DFS on the spanning tree. Since there are no cycles and only $|V| - 1$ edges each run takes $O(|V|)$ time. Run it for all vertices (and even with repeats) you get $O(V^2)$ running time.

- (d) Give an efficient algorithm to compute the second-best minimum spanning tree of G .

Solution. By part (b) you only need to flip one edge to get from MST to second best MST. First compute MST in $O(E \lg V)$ time. Use part (c) to compute all maximum weight edges. Then for all the edges in E but not in T find the one such that that edge's weight minus $\max[u, v]$ is smallest. Then adding that edge and removing the $\max[u, v]$ edge adds the smallest difference to the minimum spanning tree.

24. SINGLE-SOURCE SHORTEST PATHS

Exercise (24.1-3). Given a weighted, directed graph G with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v . Here, the shortest path is by weight, not the number of edges. Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Solution. Keep tabs on whether any relaxations happened in the last "sweep". If none, then stop.

Exercise (24.1-5). Let G be a weighted, directed graph with weight function $w : E \rightarrow \mathbb{R}$. Give an $O(VE)$ -time algorithm to find, for each vertex $v \in V$, the value $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

Solution. There are two solutions:

1. Create a universal source (with zero weight edges to all vertices), and run Bellman-Ford from the universal source. Shortest negative weight paths will be those chosen by this Bellman-Ford run. If there are no vertices which are "negatively" close then just compute G^T and find outgoing edge with smallest weight.
2. We can initialize the distance of all vertices to be its shortest distance to some other vertex (or ∞ if it's disconnected). Then relaxing $|V| - 1$ times (i.e. running Bellman-Ford) will set the $v.d = \delta^*(v)$ for each $v \in V$.

Exercise (24.1-6). Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Solution. Run Bellman-Ford. If the second for loop returns false record which edge (u, v) is relaxed. To find the shortest cycle run Bellman-Ford with new source v to find path to u .

Exercise (24.2-4). Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

Solution. Basically the DP algorithm in 22.4-2. Set the "distance" function for each vertex to 1 and then do a topological sort and then in topologically sorted order compute "distance" to be the sum of "distances" of edge going into the vertex.

Exercise (24.3-4). Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$ -time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

Solution. Key is to check whether any δ s can be further relaxed. The rest is just a consistency check: check that the tree is actually a tree, check distance from s to s is 0, and check that distance to a vertex is equal to the distance to the parent plus edge weight from parent to vertex.

Exercise (24.3-6). We are given a directed graph G on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $[0, 1]$ that represents the reliability of a communication channel from vertex u to vertex v . We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution. There are two ways to do this.

1. Define $f(u, v) = 1 - r(u, v)$ as the probability of failure. Then we want to minimize $\prod f(u, v)$ over paths p . Taking log we get $\min_p \left(\sum_{(u,v) \in p} \log(f(u, v)) \right)$ which is akin shortest path. There are some details about negativity to iron out but otherwise use Dijkstra's.
2. Alternatively you can modify dijkstra to look for max-ish paths by using max-heap and **Extract-Max** and relaxing if $v.d < u.d \cdot r(u, v)$. Although I don't exactly understand why works since the network isn't necessarily a DAG. If the graph were a DAG then to find the longest path do topological sort/shortest paths on $-G$. This picks out the most negative edges, ie the longest path in G .

Note that to find shortest cycle including s on a positive graph (all edges have positive weight) using Dijkstra: perform Dijkstra and then use distances computed to find shortest cycle (look through edges of all vertices to see if they connect back to s).

Exercise (24.3-7). Let G be a weighted, directed graph with positive weight function $w : E \rightarrow \{1, 2, \dots, W\}$ for some positive integer W , and assume that no two vertices have the same shortest-path weights from source vertex s . Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G' . Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G .

Solution. This is exactly the idea you have when you meet bfs: if the edges were integers then just adding "virtual nodes" enable you to solve shortest paths using BFS.

Exercise (24.3-8). Let G be a weighted, directed graph with positive weight function $w : E \rightarrow \{1, 2, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given vertex s in $O(WV + E)$ time.

Solution. Use radix sort or something, i.e. "bucketize" vertices according to their distance from s . Since max weight wedge is W there are a max of $(V - 1)W$ buckets. Examine the buckets one by one in increasing order of distances in bucket (moving vertices from bucket to bucket to when updated "keys").

Exercise (24.4-3). Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive?

Solution. The objective is sum of weights over all paths.

Exercise (24.4-4). Express the single-pair shortest-path problem as a linear program.

Solution. Nope, in fact the path weight can't be greater or equal to zero.

Exercise (24.4-6). Suppose that in addition to a system of difference constraints, we want to handle equality constraints of the form $x_i = x_j + b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

Solution. Rewrite the equality constraints as two inequality constraints.

Exercise (24.4-7). Show how to solve a system of difference constraints by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0 .

Solution. Probably the same as how you run DFS on all nodes on an unconnected graph: by running the main loop in a **for** loop over all of the vertices.

Problem (24-2). Nesting boxes.

- (a) Argue that the nesting relation is transitive.

Solution. Well duh. If box A can contain box B and box B can contain box C then obviously A can contain C .

- (b) Describe an efficient method to determine whether or not one d -dimensional box nests inside another.

Solution. Sort both sets of dimensions and if one set is entry-wise prior then it fits.

- (c) Give an efficient algorithm to find the longest sequence $\{B_1, B_2, \dots, B_n\}$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, \dots, k - 1$.

Solution. Do the n^2 comparisons to figure out all (B_i, B_j) , where B_i can contain B_j . Then do a topological sort on the graph because it's a DAG (because what would it mean for a cycle of box containments). Then find the longest path using DP:

Algorithm 1 Longest Path

```

1 Longest-Path( $G$ )
2    $T = \text{Top-Sort}(G)$ 
3   for  $v \in T$ 
4        $v.d = \max_{(u,v) \in E} \{u.d + 1\}$ 
5   return  $\max_{v \in V} \{v.d\}$ 

```

Problem (24-3). Arbitrage.

Solution. Similar to Exercise 24.3-6.

Problem (24-6). Bitonic shortest paths.

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. Suppose that we are given a directed graph G with weight function $w : E \rightarrow \mathbb{R}$, where all edge weights are **unique**, and we wish to find single-source shortest paths from a source vertex s . We are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from s to form a bitonic sequence. Give the most efficient algorithm you can to solve this problem, and analyze its running time.

Solution. Sort the edges by weight. Then relax increasing order, decreasing order, then increasing again and you're done.

25. ALL-PAIRS SHORTEST PATHS

Exercise (25.1-6). Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $O(n^3)$ time.

Solution. Loop over everything and check against edge weights to see where shortest path between two vertices is edge weight plus shortest path to first vertex.

Exercise (25.1-9). Modify FASTER-ALL-PAIRS-SHORTEST-PATHS so that it can determine whether the graph contains a negative-weight cycle.

Solution. Look for negatives on the diagonal.

Exercise (25.1-10). Give an efficient algorithm to find the length (number of edges) of a minimum length negative-weight cycle in a graph.

Solution. Look for the first m such that L^m has negatives on the diagonal.

Exercise (25.2-6). How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

Solution. Again, it's whether there are negative values on the diagonal.

Exercise (25.2-8). Give an $O(VE)$ -time algorithm for computing the transitive closure of a directed graph G .

Solution. Do a dfs and add all the visited vertices to a set. Do this for each vertex. Running time is $O(V(V + E))$, however $E \leq V^2$ so $O(V(V + E)) = O(VE)$.

Problem (25-1). Transitive closure of a dynamic graph.

- (a) Show how to update the transitive closure $G^* = (V, E^*)$ of a graph G in $O(V^2)$ time when a new edge is added to G .

Solution. The obvious solution: for new edge (u, v) any vertex already connected to u now becomes connected to every vertex that v connects to.

- (b) Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph.

Solution. Something something $|E| \leq |V|$?