27 Multithreaded Algorithms

27.1 The basics of dynamic multithreading

27.1-1

> Suppose that we spawn $\text{P-FIB}(n-2)$ in line 4 of P-FIB, rather than calling it as is done in the code. What is the impact on the asymptotic work, span, and parallelism?

(Removed)

27.1-2

> Draw the computation dag that results from executing $\text{P-FIB}(5)$. Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the dag on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

- Work: $T_1 = 29$.
- Span: $T_\infty = 10$.
- Parallelism: $T_1/T_\infty \approx 2.9$.

27.1-3

> Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proven in Theorem 27.1:
>
> $$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty . \tag{27.5}$$

Suppose that there are x incomplete steps in a run of the program. Since each of these steps causes at least one unit of work to be done, we have that there is at most $(T_1 - x)$ units of work done in the complete steps. Then, we suppose by contradiction that the number of complete steps is strictly greater than $\lfloor (T_1 - x)/P \rfloor$. Then, we have that the total amount of work done during the complete steps is

$$P \cdot (\lfloor (T_1 - x)/P \rfloor + 1) = P\lfloor (T_1 - x)/P = (T_1 - x) - ((T_1 - x) \mod P) + P > T_1 - x.$$

This is a contradiction because there are only $(T_1 - x)$ units of work done during complete steps, which is less than the amount we would be doing. Notice that since T_∞ is abound on the total number of both kinds of steps, it is a bound on the number of incomplete steps, x, so,

$$T_P \le \lfloor (T_1 - x)/P \rfloor + x \le \lfloor (T_1 - T_\infty)/P \rfloor + T_\infty .$$

Where the second inequality comes by noting that the middle expression, as a function of x is monotonically increasing, and so is bounded by the largest value of x that is possible, namely T_∞.

27.1-4

> Construct a computation dag for which one execution of a greedy scheduler can take nearly twice the time of another execution of a greedy scheduler on the same number of processors. Describe how the two executions would proceed.

The computation is given in the image below. Let vertex u have degree k, and assume that there are m vertices in each vertical chain. Assume that this is executed on k processors. In one execution, each strand from among the k on the left is executed concurrently, and then the m strands on the right are executed one at a time. If each strand takes unit time to execute, then the total computation takes $2m$ time. On the other hand, suppose that on each time step of the computation, $k-1$ strands from the left (descendants of u) are executed, and one from the right (a descendant of v), is executed. If each strand take unit time to executed, the total computation takes $m + m/k$. Thus, the ratio of times is $2m/(m + m/k) = 2/(1 + 1/k)$. As k gets large, this approaches 2 as desired.

27.1-5

> Professor Karan measures her deterministic multithreaded algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the three runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds. Argue that the professor is either lying or incompetent. (Hint: Use the work law (27.2), the span law (27.3), and inequality (27.5) from Exercise 27.1-3.)

(Removed)

27.1-6

> Give a multithreaded algorithm to multiply an $n \times n$ matrix by an n-vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

(Removed)

27.1-7

> Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:
>
> ```
> P-TRANSPOSE(A)
>     n = A.rows
>     parallel for j = 2 to n
>         parallel for i = 1 to j - 1
>             exchange a[i, j] with a[j, i]
> ```
>
> Analyze the work, span, and parallelism of this algorithm.

(Removed)

27.1-8

> Suppose that we replace the **parallel for** loop in line 3 of P-TRANSPOSE (see Exercise 27.1-7) with an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

(Removed)

27.1-9

> For how many processors do the two versions of the chess programs run equally fast, assuming that $T_P = T_1/P + T_\infty$?

(Removed)

27.2 Multithreaded matrix multiplication

27.2-1

> Draw the computation dag for computing $\text{P-SQUARE-MATRIX-MULTIPLY}$ on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Use the convention that spawn and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, analyze the work, span, and parallelism of this computation.

(Omit!)

27.2-2

> Repeat Exercise 27.2-1 for $\text{P-MATRIX-MULTIPLY-RECURSIVE}$.

(Omit!)

27.2-3

> Give pseudocode for a multithreaded algorithm that multiplies two $n \times n$ matrices with work $\Theta(n^3)$ but span only $\Theta(\lg n)$. Analyze your algorithm.

(Removed)

27.2-4

> Give pseudocode for an efficient multithreaded algorithm that multiplies a $p \times q$ matrix by a $q \times r$ matrix. Your algorithm should be highly parallel even if any of p, q, and r are 1. Analyze your algorithm.

(Removed)

27.2-5

> Give pseudocode for an efficient multithreaded algorithm that transposes an $n \times n$ matrix in place by using divide-and-conquer to divide the matrix recursively into four $n/2 \times n/2$ submatrices. Analyze your algorithm.

```
P-MATRIX-TRANSPOSE(A)
    n = A.rows
    if n == 1
        return
    partition A into n / 2 × n / 2 submatrices A11, A12, A21, A22
    spawn P-MATRIX-TRANSPOSE(A11)
    spawn P-MATRIX-TRANSPOSE(A12)
    spawn P-MATRIX-TRANSPOSE(A21)
    P-MATRIX-TRANSPOSE(A22)
    sync
    // exchange A12 with A21
    parallel for i = 1 to n / 2
        parallel for j = 1 + n / 2 to n
            exchange A[i, j] with A[i + n / 2, j - n / 2]
```

- span: $T(n) = T(n/2) + O(\lg n) = O(\lg^2 n)$.
- work: $T(n) = 4T(n/2) + O(n^2) = O(n^2 \lg n)$.

27.2-6

> Give pseudocode for an efficient multithreaded implementation of the Floyd-Warshall algorithm (see Section 25.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

(Removed)

27.3 Multithreaded merge sort

27.3-1

> Explain how to coarsen the base case of P-MERGE.

Replace the condition on line 2 with a check that $n < k$ for some base case size k. And instead of just copying over the particular element of A to the right spot in B, you would call a serial sort on the remaining segment of A and copy the result of that over into the right spots in B.

27.3-2

> Instead of finding a median element in the larger subarray, as P-MERGE does, consider a variant that finds a median element of all the elements in the two sorted subarrays using the result of Exercise 9.3-8. Give pseudocode for an efficient multithreaded merging procedure that uses this median-finding procedure. Analyze your algorithm.

By a slight modification of exercise 9.3-8 we can find we can find the median of all elements in two sorted arrays of total length n in $O(\lg n)$ time. We'll modify P-MERGE to use this fact. Let $\text{MEDIAN}(T, p_1, r_1, p_2, r_2)$ be the function which returns a pair, q, where $q.\text{pos}$ is the position of the median of all the elements T which lie between positions p_1 and r_1, and between positions p_2 and r_2, and $q.\text{arr}$ is 1 if the position is between p_1 and r_1, and 2 otherwise.

```
P-MEDIAN-MERGE(T, p[1], r[1], p[2], r[2], A, p[3])
    n[1] = r[1] - p[1] + 1
    n[2] = r[2] - p[2] + 1
    if n[1] < n[2]              // ensure that n[1] ≥ n[2]
        exchange p[1] with p[2]
        exchange r[1] with r[2]
        exchange n[1] with n[2]
    if n[1] == 0                // both empty?
        return
    q = MEDIAN(T, p[1], r[1], p[2], r[2])
    if q.arr == 1
        q[2] = BINARY-SEARCH(T[q.pos], T, p[2], r[2])
        q[3] = p[3] + q.pos - p[1] + q[2] - p[2]
        A[q[3]] = T[q.pos]
        spawn P-MEDIAN-MERGE(T, p[1], q.pos - 1, p[2], q[2] - 1, A, p[3])
        P-MEDIAN-MERGE(T, q.pos + 1, r[1], q[2] + 1, r[2], A, p[3])
        sync
    else
        q[2] = BINARY-SEARCH(T[q.pos], T, p[1], r[1])
        q[3] = p[3] + q.pos - p[2] + q[2] - p[1]
        A[q[3]] = T[q.pos]
        spawn P-MEDIAN-MERGE(T, p[1], q[2] - 1, p[2], q.pos - 1, A, p[3])
        P-MEDIAN-MERGE(T, q[2] + 1, r[1], q.pos + 1, r[2], A, p[3])
        sync
```

The work is characterized by the recurrence $T_1(n) = O(\lg n) + 2T_1(n/2)$, whose solution tells us that $T_1(n) = O(n)$. The work is at least $\Omega(n)$ since we need to examine each element, so the work is $\Theta(n)$. The span satisfies the recurrence

$$\begin{aligned} T_\infty(n) &= O(\lg n) + O(\lg n/2) + T_\infty(n/2) \\ &= O(\lg n) + T_\infty(n/2) \\ &= \Theta(\lg^2 n), \end{aligned}$$

by exercise 4.6-2.

27.3-3

> Give an efficient multithreaded algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 171. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (Hint: You may need an auxiliary array and may need to make more than one pass over the input elements.)

Suppose that there are c different processors, and the array has length n and you are going to use its last element as a pivot. Then, look at each chunk of size $\lceil \frac{n}{c} \rceil$ of entries before the last element, give one to each processor. Then, each counts the number of elements that are less than the pivot. Then, we compute all the running sums of these values that are returned. This can be done easily by considering all of the subarrays placed along the leaves of a binary tree, and then summing up adjacent pairs. This computation can be done in time $\lg(\min\{c, n\})$ since it's the log of the number of leaves. From there, we can compute all the running sums for each of the subarrays also in logarithmic time. This is by keeping track of the sum of all more left cousins of each internal node, which is found by adding the left sibling's sum vale to the left cousin value of the parent, with the root's left cousin value initiated to 0. This also just takes time the depth of the tree, so is $\lg(\min\{c, n\})$. Once all of these values are computed at the root, it is the index that the subarray's elements less than the pivot should be put. To find the position where the subarray's elements larger than the root should be put, just put it at twice the sum value of the root minus the left cousin value for that subarray. Then, the time taken is just $O(\frac{n}{c})$. By doing this procedure, the total work is just $O(n)$, and the span is $O(\lg n)$, and so has parallelization of $O(\frac{n}{\lg n})$. This whole process is split across the several algoithms appearing here.

27.3-4

> Give a multithreaded version of RECURSIVE-FFT on page 911. Make your implementation as parallel as possible. Analyze your algorithm.

```
P-RECURSIVE-FFT(a)
    n = a.length
    if n == 1
        return a
    w[n] = e^{2 * π * i / n}
    w = 1
    a(0) = [a[0], a[2]..a[n - 2]]
    a(1) = [a[1], a[3]..a[n - 1]]
    y(0) = spawn P-RECURSIVE-FFT(a[0])
    y(1) = P-RECURSIVE-FFT(a[1])
    sync
    parallel for k = 0 to n / 2 - 1
        y[k] = y[k](0) + w * y[k](1)
        y[k + n / 2] = y[k](0) - w * y[k](1)
        w = w * w[n]
    return y
```

P-RECURSIVE-FFT parallelized over the two recursive calls, having a parallel for works because each of the iterations of the **for** loop touch independent sets of variables. The span of the procedure is only $\Theta(\lg n)$ giving

it a parallelization of $\Theta(n)$.

27.3-5 ⋆

> Give a multithreaded version of RANDOMIZED-SELECT on page 216. Make your implementation as parallel as possible. Analyze your algorithm. (Hint: Use the partitioning algorithm from Exercise 27.3-3.)

Randomly pick a pivot element, swap it with the last element, so that it is in the correct format for running the procedure described in 27.3-3. Run partition from problem 27.3-3. As an intermediate step, in that procedure, we compute the number of elements less than the pivot ($T.root.sum$), so keep track of that value after the end of partition. Then, if we have that it is less than k, recurse on the subarray that was greater than or equal to the pivot, decreasing the order statistic of the element to be selected by $T.root.sum$. If it is larger than the order statistic of the element to be selected, then leave it unchanged and recurse on the subarray that was formed to be less than the pivot. A lot of the analysis in section 9.2 still applies, except replacing the timer needed for partitioning with the runtime of the algorithm in problem 27.3-3. The work is unchanged from the serial case because when $c = 1$, the algorithm reduces to the serial algorithm for partitioning. For span, the $O(n)$ term in the equation half way down page 218 can be replaced with an $O(\lg n)$ term. It can be seen with the substitution method that the solution to this is logarithmic

$$E[T(n)] \le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} C\lg k + O(\lg n) \le O(\lg n).$$

So, the total span of this algorithm will still just be $O(\lg n)$.

27.3-6 ⋆

> Show how to multithread SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

Let $\text{MEDIAN}(A)$ denote a brute force method which returns the median element of the array A. We will only use this to find the median of small arrays, in particular, those of size at most 5, so it will always run in constant time. We also let $A[i..j]$ denote the array whose elements are $A[i], A[i+1], \ldots, A[j]$. The function $\text{P-PARTITION}(A, x)$ is a multithreaded function which partitions A around the input element x and returns the number of elements in A which are less than or equal to x. Using a parallel **for** loop, its span is logarithmic in the number of elements in A. The work is the same as the serialization, which is $\Theta(n)$ according to section 9.3. The span satisfies the recurrence

$$\begin{aligned} T_\infty(n) &= \Theta(\lg n/5) + T_\infty(n/5) + \Theta(\lg n) + T_\infty(7n/10 + 6) \\ &\le \Theta(\lg n) + T_\infty(n/5) + T_\infty(7n/10 + 6). \end{aligned}$$

Using the substitution method we can show that $T_\infty(n) = O(n^\epsilon)$ for some $\epsilon < 1$. In particular, $\epsilon = 0.9$ works. This gives a parallelization of $\Omega(n^{0.1})$.

```
P-SELECT(A, i)
    if n == 1
        return A[1]
    let T[1..floor(n / 5)] be a new array
    parallel for i = 0 to floor(n / 5) - 1
        T[i + 1] = MEDIAN(A[i * floor(n / 5)..i * floor(n / 5) + 4])
    if n / 5 is not an integer
        T[floor(n / 5)] = MEDIAN(A[5 * floor(n / 5)..n])
    x = P-SELECT(T, ceil(n / 5))
    k = P-PARTITION(A, x)
    if k == i
        return x
    else if i < k
        P-SELECT(A[1..k - 1], i)
    else
        P-SELECT(A[k + 1..n], i - k)
```

Problem 27-1 Implementing parallel loops using nested parallelism

> Consider the following multithreaded algorithm for performing pairwise addition on n-element arrays $A[1..n]$ and $B[1..n]$, storing the sums in $C[1..n]$:
>
> ```
> SUM-ARRAYS(A, B, C)
>     parallel for i = 1 to A.length
>         C[i] = A[i] + B[i]
> ```
>
> **a.** Rewrite the parallel loop in SUM-ARRAYS using nested parallelism (**spawn** and **sync**) in the manner of MAT-VEC-MAIN-LOOP. Analyze the parallelism of your implementation.
>
> Consider the following alternative implementation of the parallel loop, which contains a value grain-size to be specified:
>
> ```
> SUM-ARRAYS'(A, B, C)
>     n = A.length
>     grain-size = ?      // to be determined
>     r = ceil(n / grain-size)
>     for k = 0 to r - 1
>         spawn ADD-SUBARRAY(A, B, C, k * grain-size + 1, min((k + 1) *
> grain-size, n))
>     sync
>
> ADD-SUBARRAY(A, B, C, i, j)
>     for k = i to j
>         C[k] = A[k] + B[k]
> ```
>
> **b.** Suppose that we set grain-size $= 1$. What is the parallelism of this implementation?
>
> **c.** Give a formula for the span of SUM-ARRAYS′ in terms of n and grain-size. Derive the best value for grain-size to maximize parallelism.

a. See the algorithm $\text{SUM-ARRAYS}(A, B, C)$. The parallelism is $O(n)$ since it's work is $n \lg n$ and the span is $\lg n$.

b. If grainsize is 1, this means that each call of ADD-SUBARRAY just sums a single pair of numbers. This means that since the for loop on line 4 will run n times, both the span and work will be $O(n)$. So, the parallelism is just $O(1)$.

```
SUM-ARRAYS(A, B, C)
    n = floor(A.length / 2)
    if n == 0
        C[1] = A[1] + B[1]
    else
        spawn SUM-ARRAYS(A[1..n], B[1..n], C[1..n])
        SUM-ARRAYS(A[n + 1..A.length], B[n + 1..A..length], C[n +
1..A.length])
```

c. Let g be the grainsize. The runtime of the function that spawns all the other functions is $\lceil \frac{n}{g} \rceil$. The runtime of any particular spawned task is g. So, we want to minimize

$$\frac{n}{g} + g.$$

To do this we pull out our freshman calculus hat and take a derivative, we have

$$0 = 1 - \frac{n}{g^2}.$$

To solve this, we set $g = \sqrt{n}$. This minimizes the quantity and makes the span $O(n/g + g) = O(\sqrt{n})$. Resulting in a parallelism of $O(\sqrt{n})$.

Problem 27-2 Saving temporary space in matrix multiplication

> The P-MATRIX-MULTIPLY-RECURSIVE procedure has the disadvantage that it must allocate a temporary matrix T of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation.
>
> The P-MATRIX-MULTIPLY-RECURSIVE procedure does have high parallelism, however. For example, ignoring the constants in the Θ-notation, the parallelism for multiplying 1000×1000 matrices comes to approximately $1000^3/10^2 = 10^7$, since $\lg 1000 \approx 10$. Most parallel computers have far fewer than 10 million processors.
>
> **a.** Describe a recursive multithreaded algorithm that eliminates the need for the temporary matrix T at the cost of increasing the span to $\Theta(n)$. (Hint: Compute $C = C + AB$ following the general strategy of P-MATRIX-MULTIPLY-RECURSIVE, but initialize C in parallel and insert a sync in a judiciously chosen location.)
>
> **b.** Give and solve recurrences for the work and span of your implementation.
>
> **c.** Analyze the parallelism of your implementation. Ignoring the constants in the Θ-notation, estimate the parallelism on 1000×1000 matrices. Compare with the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

(Removed)

Problem 27-3 Multithreaded matrix algorithms

> **a.** Parallelize the LU-DECOMPOSITION procedure on page 821 by giving pseudocode for a multithreaded version of this algorithm. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.
>
> **b.** Do the same for LUP-DECOMPOSITION on page 824.
>
> **c.** Do the same for LUP-SOLVE on page 817.
>
> **d.** Do the same for a multithreaded algorithm based on equation (28.13) for inverting a symmetric positive-definite matrix.

a. For the algorithm $\text{LU-DECOMPOSITION}(A)$ on page 821, the inner **for** loops can be parallelized, since they never update values that are read on later runs of those loops. However, the outermost **for** loop cannot be parallelized because across iterations of it the changes to the matrices from previous runs are used to affect the next. This means that the span will be $\Theta(n \lg n)$, work will still be $\Theta(n^3)$ and, so, the parallelization will be $\Theta(\frac{n^3}{n \lg n}) = \Theta(\frac{n^2}{\lg n})$.

b. The **for** loop on lines 7-10 is taking the max of a set of things, while recording the index that that max occurs. This **for** loop can therefor be replaced with a $\lg n$ span parallelized procedure in which we arrange the n elements into the leaves of an almost balanced binary tree, and we let each internal node be the max of its two children. Then, the span will just be the depth of this tree. This procedure can gracefully scale with the number of processors to make the span be linear, though even if it is $\Theta(n \lg n)$ it will be less than the $\Theta(n^2)$ work later. The **for** loop on line 14-15 and the implicit **for** loop on line 15 have no concurrent editing, and so, can be made parallel to have a span of $\lg n$. While the **for** loop on lines 18-19 can be made parallel, the one containing it cannot without creating data races. Therefore, the total span of the naive parallelized algorithm will be $\Theta(n^2 \lg n)$, with a work of $\Theta(n^3)$. So, the parallelization will be $\Theta(\frac{n}{\lg n})$. Not as parallized as part (a), but still a significant improvement.

c. We can parallelize the computing of the sums on lines 4 and 6, but cannot also parallize the **for** loops containing them without creating an issue of concurrently modifying data that we are reading. This means that the span will be $\Theta(n \lg n)$, work will still be $\Theta(n^2)$, and so the parallelization will be $\Theta(\frac{n}{\lg n})$.

d. The recurrence governing the amount of work of implementing this procedure is given by

$$I(n) \le 2I(n/2) + 4M(n/2) + O(n^2).$$

However, the two inversions that we need to do are independent, and the span of parallelized matrix multiply is just $O(\lg n)$. Also, the n^2 work of having to take a transpose and subtract and add matrices has a span of only $O(\lg n)$. Therefore, the span satisfies the recurrence

$$I_\infty(n) \le I_\infty(n/2) + O(\lg n).$$

This recurrence has the solution $I_\infty(n) \in \Theta(\lg^2 n)$ by exercise 4.6-2. Therefore, the span of the inversion algorithm obtained by looking at the procedure detailed on page 830. This makes the parallelization of it equal to $\Theta(M(n)/\lg^2 n)$ where $M(n)$ is the time to compute matrix products.

Problem 27-4 Multithreading reductions and prefix computations

> A $\otimes$-***reduction*** of an array $x[1 \ldots n]$, where $\otimes$ is an associative operator, is the value
>
> $$y = x[1] \otimes x[2] \otimes \cdots \otimes x[n].$$
>
> The following procedure computes the $\otimes$-reduction of a subarray $x[i \ldots j]$ serially.
>
> ```
> REDUCE(x, i, j)
>     y = x[i]
>     for k = i + 1 to j
>         y = y ⊗ x[k]
>     return y
> ```
>
> **a.** Use nested parallelism to implement a multithreaded algorithm P-REDUCE, which performs the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span. Analyze your algorithm.
>
> A related problem is that of computing a $\otimes$-***prefix computation***, sometimes called a $\otimes$-***scan***, on an array $x[1 \ldots n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan produces the array $y[1 \ldots n]$ given by
>
> $$\begin{aligned} y[1] &= x[1], \\ y[2] &= x[1] \otimes x[2], \\ y[3] &= x[1] \otimes x[2] \otimes x[3], \\ &\vdots \\ y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n], \end{aligned}$$
>
> that is, all prefixes of the array x "summed" using $\otimes$ operator. The following serial procedure SCAN performs a $\otimes$-prefix computation:
>
> ```
> SCAN(x)
>     n = x.length
>     let y[1..n] be a new array
>     y[1] = x[1]
>     for i = 2 to n
>         y[i] = y[i - 1] ⊗ x[i]
>     return y
> ```
>
> Unfortunately, multithreading SCAN is not straightforward. For example, changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The following procedure P-SCAN-1 performs the $\otimes$-prefix computation in parallel, albeit inefficiently.
>
> ```
> P-SCAN-1(x)
>     n = x.length
>     let y[1..n] be a new array
>     P-SCAN-1-AUX(x, y, 1, n)
>     return y
> ```
>
> ```
> P-SCAN-1-AUX(x, y, i, j)
>     parallel for l = i to j
>         y[l] = P-REDUCE(x, 1, l)
> ```
>
> **b.** Analyze the work, span, and parallelism of P-SCAN-1.
>
> By using nested parallelism, we can obtain a more efficient $\otimes$-prefix computation:
>
> ```
> P-SCAN-2(x)
>     n = x.length
>     let y[1..n] be a new array
>     P-SCAN-2-AUX(x, y, 1, n)
>     return y
> ```

```
P-SCAN-2-AUX(x, y, i, j)
    if i == j
        y[i] = x[i]
    else k = floor((i + j) / 2)
        spawn P-SCAN-2-AUX(x, y, i, k)
        P-SCAN-2-AUX(x, y, k + 1, j)
        sync
        parallel for l = k + 1 to j
            y[l] = y[k] ⊗ y[l]
```

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

We can improve on both P-SCAN-1 and P-SCAN-2 by performing the $\otimes$-prefix computation in two distinct passes over the data. On the first pass, we gather the terms for various contiguous subarrays of x into a temporary array t, and on the second pass we use the terms in t to compute the final result y. The following pseudocode implements this strategy, but certain expressions have been omitted:

```
P-SCAN-3(x)
    n = x.length
    let y[1..n] and t[1..n] be new arrays
    y[1] = x[1]
    if n > 1
        P-SCAN-UP(x, t, 2, n)
        P-SCAN-DOWN(x[1], x, t, y, 2, n)
    return y
```

```
P-SCAN-UP(x, t, i, j)
    if i == j
        return x[i]
    else
        k = floor((i + j) / 2)
        t[k] = spawn P-SCAN-UP(x, t, i, k)
        right = P-SCAN-UP(x, t, k + 1, j)
        sync
        return ____            // fill in the blank
```

```
P-SCAN-DOWN(v, x, t, y, i, j)
    if i == j
        y[i] = v ⊗ x[i]
    else
        k = floor((i + j) / 2)
        spawn P-SCAN-DOWN(____, x, t, y, i, k)  // fill in the blank
        P-SCAN-DOWN(____, x, t, y, k + 1, j)    // fill in the blank
        sync
```

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with expressions you supplied, P-SCAN-3 is correct. (Hint: Prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

e. Analyze the work, span, and parallelism of P-SCAN-3.

(Removed)

Problem 27-5 Multithreading a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a ***stencil***. For example, Section 15.4 presents a stencil algorithm to compute a longest common subsequence, where the value in entry $c[i, j]$ depends only on the values in $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, as well as the elements x_i and y_j within the two sequences given as inputs. The input sequences are fixed, but the algorithm fills in the two-dimensional array c so that it computes entry $c[i, j]$ after computing all three entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$.

In this problem, we examine how to use nested parallelism to multithread a simple stencil calculation on an $n \times n$ array A in which, of the values in A, the value placed into entry $A[i, j]$ depends only on values in $A[i', j']$, where $i' \le i$ and $j' \le j$ (and of course, $i' \ne i$ or $j' \ne j$). In other words, the value in an entry depends only on values in entries that are above it and/or to its left, along with static information outside of the array. Furthermore, we assume throughout this problem that once we have filled in the entries upon which $A[i, j]$ depends, we can fill in $A[i, j]$ in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 15.4).

We can partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \tag{27.11}$$

Observe now that we can fill in subarray A_{11} recursively, since it does not depend on the entries of the other three subarrays. Once A_{11} is complete, we can continue to fill in A_{12} and A_{21} recursively in parallel, because although they both depend on A_{11}, they do not depend on each other. Finally, we can fill in A_{22} recursively.

a. Give multithreaded pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (27.11) and the discussion above. (Don't worry about the details of the base case, which depends on the specific stencil.) Give and solve recurrences for the work and span of this algorithm in terms of n. What is the parallelism?

b. Modify your solution to part (a) to divide an $n \times n$ array into nine $n/3 \times n/3$ subarrays, again recursing with as much parallelism as possible. Analyze this algorithm. How much more or less parallelism does this algorithm have compared with the algorithm from part (a)?

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \ge 2$. Divide an $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible. In terms of n and b, what are the work, span, and parallelism of your algorithm? Argue that, using this approach, the parallelism must be $o(n)$ for any choice of $b \ge 2$. (Hint: For this last argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \ge 2$.)

d. Give pseudocode for a multithreaded algorithm for this simple stencil calculation that achieves $\Theta(n \lg n)$ parallelism. Argue using notions of work and span that the problem, in fact, has $\Theta(n)$ inherent parallelism. As it turns out, the divide-and-conquer nature of our multithreaded pseudocode does not let us achieve this maximal parallelism.

(Removed)

Problem 27-6 Randomized multithreaded algorithms

Just as with ordinary serial algorithms, we sometimes want to implement randomized multithreaded algorithms. This problem explores how to adapt the various performance measures in order to handle the expected behavior of such algorithms. It also asks you to design and analyze a multithreaded algorithm for randomized quicksort.

a. Explain how to modify the work law (27.2), span law (27.3), and greedy scheduler bound (27.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

b. Consider a randomized multithreaded algorithm for which 1 of the time we have $T_1 = 10^4$ and $T_{10,000} = 1$, but for 99 of the time we have $T_1 = T_{10,000} = 10^9$. Argue that the ***speedup*** of a randomized multithreaded algorithm should be defined as $E[T_1]/E[T_P]$, rather than $E[T_1/T_P]$.

c. Argue that the ***parallelism*** of a randomized multithreaded algorithm should be defined as the ratio $E[T_1]/E[T_\infty]$.

d. Multithread the RANDOMIZED-QUICKSORT algorithm on page 179 by using nested parallelism. (Do not parallelize RANDOMIZED-PARTITION.) Give the pseudocode for your P-RANDOMIZED-QUICKSORT algorithm.

e. Analyze your multithreaded algorithm for randomized quicksort. (Hint: Review the analysis of RANDOMIZED-SELECT on page 216.)

a.

$$E[T_P] \ge E[T_1]/P$$
$$E[T_P] \ge E[T_\infty]$$
$$E[T_P] \le E[T_1]/P + E[T_\infty].$$

b.

$$E[T_1] \approx E[T_{10,000}] \approx 9.9 \times 10^8, E[T_1]/E[T_P] = 1.$$
$$E[T_1/T_{10,000}] = 10^4 * 0.01 + 0.99 = 100.99.$$

c. Same as the above.

d.

```
RANDOMIZED-QUICKSORT(A, p, r)
    if p < r
        q = RANDOM-PARTITION(A, p, r)
    spawn RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
    sync
```

e.

$$E[T_1] = O(n \lg n)$$
$$E[T_\infty] = O(\lg n)$$
$$E[T_1]/E[T_\infty] = O(n).$$

28 Matrix Operations

28.1 Solving systems of linear equations

28.1-1

Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

by using forward substitution.

$$\begin{pmatrix} 3 \\ 14 - 4 \cdot 3 \\ -7 - 5 \cdot (14 - 4 \cdot 3) - (-6) \cdot 3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}.$$

28.1-2

Find an LU decomposition of the matrix

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & -5 & 6 \\ 0 & 4 & -5 \\ 0 & 0 & 4 \end{pmatrix}.$$

28.1-3

Solve the equation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

by using forward substitution.

We have

$$A = \begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix},$$
$$b = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix},$$

and we with to solve for the unknown x. The LUP decomposition is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -\frac{3.2}{3.4} & 1 \end{pmatrix},$$
$$U = \begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 2.2 + \frac{11.52}{3.4} \end{pmatrix},$$
$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Using forward substitution, we solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -\frac{3.2}{3.4} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 12 \\ 9 \end{pmatrix},$$

obtaining

$$y = \begin{pmatrix} 5 \\ 11 \\ 7 + \frac{35.2}{3.4} \end{pmatrix}$$

by computing first y_1, then y_2, and finally y_3. Using back substitution, we solve $Ux = y$ for x:

$$\begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 2.2 + \frac{11.52}{3.4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 7 + \frac{35.2}{3.4} \end{pmatrix},$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} -\frac{3}{19} \\ -\frac{1}{19} \\ \frac{49}{19} \end{pmatrix}$$

by computing first x_3, then x_2, and finally x_1.

28.1-4

> Describe the LUP decomposition of a diagonal matrix.

The LUP decomposition of a diagonal matrix D is $D = IDI$ where I is the identity matrix.

28.1-5

> Describe the LUP decomposition of a permutation matrix A, and prove that it is unique.

(Omit!)

28.1-6

> Show that for all $n \geq 1$, there exists a singular $n \times n$ matrix that has an LU decomposition.

The zero matrix always has an LU decomposition by taking L to be any unit lower-triangular matrix and U to be the zero matrix, which is upper triangular.

28.1-7

> In LU-DECOMPOSITION, is it necessary to perform the outermost **for** loop iteration when $k = n$? How about in LUP-DECOMPOSITION?

For LU-DECOMPOSITION, it is indeed necessary. If we didn't run the line 6 of the outermost **for** loop, u_{nn} would be left its initial value of 0 instead of being set equal to a_{nn}. This can clearly produce incorrect results, because the LU-DECOMPOSITION of any non-singular matrix must have both L and U having positive determinant. However, if $u_{nn} = 0$, the determinant of U will be 0 by problem D.2-2.

For LUP-DECOMPOSITION, the iteration of the outermost **for** loop that occurs with $k = n$ will not change the final answer. Since π would have to be a permutation on a single element, it cannot be non-trivial. and the **for** loop on line 16 will not run at all.

28.2 Inverting matrices

28.2-1

> Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $S(n)$ denote the time required to square an $n \times n$ matrix. Show that multiplying and squaring matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time squaring algorithm, and an $S(n)$-time squaring algorithm implies an $O(S(n))$-time matrix-multiplication algorithm.

Showing that being able to multiply matrices in time $M(n)$ implies being able to square matrices in time $M(n)$ is trivial because squaring a matrix is just multiplying it by itself.

The more tricky direction is showing that being able to square matrices in time $S(n)$ implies being able to multiply matrices in time $O(S(n))$.

As we do this, we apply the same regularity condition that $S(2n) \in O(S(n))$. Suppose that we are trying to multiply the matrices, A and B, that is, find AB. Then, define the matrix

$$C = \begin{pmatrix} I & A \\ 0 & B \end{pmatrix}$$

Then, we can find C^2 in time $S(2n) \in O(S(n))$. Since

$$C^2 = \begin{pmatrix} I & A + AB \\ 0 & B \end{pmatrix}$$

Then we can just take the upper right quarter of C^2 and subtract A from it to obtain the desired result. Apart from the squaring, we've only done work that is $O(n^2)$. Since $S(n)$ is $\Omega(n^2)$ anyways, we have that the total amount of work we've done is $O(n^2)$.

28.2-2

> Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $L(n)$ be the time to compute the LUP decomposition of an $n \times n$ matrix. Show that multiplying matrices and computing LUP decompositions of matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm, and an $L(n)$-time LUP-decomposition algorithm implies an $O(L(n))$-time matrix-multiplication algorithm.

Let A be an $n \times n$ matrix. Without loss of generality we'll assume $n = 2^k$, and impose the regularity condition that $L(n/2) \leq cL(n)$ where $c < 1/2$ and $L(n)$ is the time it takes to find an LUP decomposition of an $n \times n$ matrix. First, decompose A as

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$$

where A_1 is $n/2$ by n. Let $A_1 = L_1 U_1 P_1$ be an LUP decomposition of A_1, where L_1 is $/2$ by $n/2$, U_1 is $n/2$ by n, and P_1 is n by n. Perform a block decomposition of U_1 and $A_2 P_1^{-1}$ as $U_1 = [\overline{U_1}|B]$ and $A_2 P_1^{-1} = [C|D]$ where $\overline{U_1}$ and C are $n/2$ by $n/2$ matrices. Since we assume that A is nonsingular, $\overline{U_1}$ must also be nonsingular.

Set $F = D - C\overline{U_1}^{-1} B$. Then we have

$$A = \begin{pmatrix} L_1 & 0 \\ C\overline{U_1}^{-1} & I_{n/2} \end{pmatrix} \begin{pmatrix} \overline{U_1} & B \\ 0 & F \end{pmatrix} P_1.$$

Now let $F = L_2 U_2 P_2$ be an LUP decomposition of F, and let $\overline{P} = \begin{pmatrix} I_{n/2} & 0 \\ 0 & P_2 \end{pmatrix}$. Then we may write

$$A = \begin{pmatrix} L_1 & 0 \\ C\overline{U_1}^{-1} & L_2 \end{pmatrix} \begin{pmatrix} \overline{U_1} & BP_2^{-1} \\ 0 & U_2 \end{pmatrix} \overline{P} P_1.$$

This is an LUP decomposition of A. To achieve it, we computed two LUP decompositions of half size, a constant number of matrix multiplications, and a constant number of matrix inversions. Since matrix inversion and multiplication are computationally equivalent, we conclude that the runtime is $O(M(n))$.

28.2-3

> Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $D(n)$ denote the time required to find the determinant of an $n \times n$ matrix. Show that multiplying matrices and computing the determinant have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time determinant algorithm, and a $D(n)$-time determinant algorithm implies an $O(D(n))$-time matrix-multiplication algorithm.

(Omit!)

28.2-4

> Let $M(n)$ be the time to multiply two $n \times n$ boolean matrices, and let $T(n)$ be the time to find the transitive closure of an $n \times n$ boolean matrix. (See Section 25.2.) Show that an $M(n)$-time boolean matrix-multiplication algorithm implies an $O(M(n) \lg n)$-time transitive-closure algorithm, and a $T(n)$-time transitive-closure algorithm implies an $O(T(n))$-time boolean matrix-multiplication algorithm.

Suppose we can multiply boolean matrices in $M(n)$ time, where we assume this means that if we're multiplying boolean matrices A and B, then $(AB)_{ij} = (a_{i1} \wedge b_{1j}) \vee \cdots \vee (a_{in} \wedge b_{nj})$. To find the transitive closure of a boolean matrix A we just need to find the n^{th} power of A. We can do this by computing A^2, then $(A^2)^2$, then $((A^2)^2)^2$ and so on. This requires only $\lg n$ multiplications, so the transitive closure can be computed in $O(M(n) \lg n)$.

For the other direction, first view A and B as adjacency matrices, and impose the regularity condition $T(3n) = O(T(n))$, where $T(n)$ is the time to compute the transitive closure of a graph on n vertices. We will define a new graph whose transitive closure matrix contains the boolean product of A and B. Start by placing $3n$ vertices down, labeling them $1, 2, \ldots, n, 1', 2', \ldots, n', 1'', 2'', \ldots, n''$.

Connect vertex i to vertex j' if and only if $A_{ij} = 1$. Connect vertex j' to vertex k'' if and only if $B_{jk} = 1$. In the resulting graph, the only way to get from the first set of n vertices to the third set is to first take an edge which "looks like" an edge in A, then take an edge which "looks like" an edge in B. In particular, the transitive closure of this graph is:

$$\begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}.$$

Since the graph is only of size $3n$, computing its transitive closure can be done in $O(T(3n)) = O(T(n))$ by the regularity condition. Therefore multiplying matrices and finding transitive closure are are equally hard.

28.2-5

> Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix elements are drawn from the field of integers modulo 2? Explain.

It does not work necessarily over the field of two elements. The problem comes in in applying theorem D.6 to conclude that $A^T A$ is positive definite. In the proof of that theorem they obtain that $||Ax||^2 \geq 0$ and only zero if every entry of Ax is zero. This second part is not true over the field with two elements, all that would be required is that there is an even number of ones in Ax. This means that we can only say that $A^T A$ is positive semi-definite instead of the positive definiteness that the algorithm requires.

28.2-6 ⋆

> Generalize the matrix-inversion algorithm of Theorem 28.2 to handle matrices of complex numbers, and prove that your generalization works correctly. (Hint: Instead of the transpose of A, use the ***conjugate transpose*** A^*, which you obtain from the transpose of A by replacing every entry with its complex conjugate. Instead of symmetric matrices, consider ***Hermitian*** matrices, which are matrices A such that $A = A^*$.)

We may again assume that our matrix is a power of 2, this time with complex entries. For the moment we assume our matrix A is Hermitian and positivedefinite. The proof goes through exactly as before, with matrix transposes replaced by conjugate transposes, and using the fact that Hermitian positivedefinite matrices are invertible. Finally, we need to justify that we can obtain the same asymptotic running time for matrix multiplication as for matrix inversion when A is invertible, but not Hermitian positive-definite.

For any nonsingular matrix A, the matrix A^*A is Hermitian and positive definite, since for any x we have $x^*A^*Ax = \langle Ax, Ax \rangle > 0$ by the definition of inner product. To invert A, we first compute $(A^*A)^{-1} = A^{-1}(A^*)^{-1}$. Then we need only multiply this result on the right by A^*. Each of these steps takes $O(M(n))$ time, so we can invert any nonsingluar matrix with complex entries in $O(M(n))$ time.

28.3 Symmetric positive-definite matrices and least-squares approximation

28.3-1

> Prove that every diagonal element of a symmetric positive-definite matrix is positive.

To see this, let e_i be the vector that is 0s except for a 1 in the ith position. Then, we consider the quantity $e_i^T A e_i$ for every i. Ae_i takes each row of A and pulls out the ith column of it, and puts those values into a column vector. Then, we multiply that on the left by e_i^T, pulls out the ith row of this quantity, which means that the quantity $e_i^T A e_i$ exactly the value of $A_{i,i}$.

So, we have that by positive definiteness, since e_i is nonzero, that quantity must be positive. Since we do this for every i, we have that every entry along the diagonal must be positive.

28.3-2

> Let
> $$A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$$
> be a 2×2 symmetrix positive-definite matrix. Prove that its determinant $ac - b^2$ is positive by "completing the square" in a manner similar to that used in the proof of Lemma 28.5.

Let $x = -by/a$. Since A is positive-definite, we have

$$\begin{aligned} 0 &< \begin{pmatrix} x & y \end{pmatrix}^T A \begin{pmatrix} x \\ y \end{pmatrix} \\ &= \begin{pmatrix} x & y \end{pmatrix}^T \begin{pmatrix} ax + by \\ bx + cy \end{pmatrix} \\ &= ax^2 + 2bxy + cy^2 \\ &= cy^2 - \frac{b^2 y^2}{a} \\ &= (c - b^2/a)y^2. \end{aligned}$$

Thus, $c - b^2/a > 0$, which implies $ac - b^2 > 0$, since $a > 0$.

28.3-3

> Prove that the maximum element in a symmetric positive-definite matrix lies on the diagonal.

Suppose to a contradiction that there were some element a_{ij} with $i \neq j$ so that a_{ij} were a largest element. We will use e_i to denote the vector that is all zeroes except for having a 1 at position i. Then, we consider the value $(e_i - e_j)^T A (e_i - e_j)$. When we compute $A(e_i - e_j)$ this will return a vector which is column i minus column j. Then, when we do the last multiplication, we will get the quantity which is the ith row minus the jth row. So,

$$\begin{aligned} (e_i - e_j)^T A (e_i - e_j) &= a_{ii} - a_{ij} - a_{ji} + a_{jj} \\ &= a_{ii} + a_{jj} - 2a_{ij} \leq 0 \end{aligned}$$

Where we used symmetry to get that $a_{ij} = a_{ji}$. This result contradicts the fact that A was positive definite. So, our assumption that there was a element tied for largest off the diagonal must of been false.

28.3-4

Prove that the determinant of each leading submatrix of a symmetrix positive-definite matrix is positive.

The claim clearly holds for matrices of size 1 because the single entry in the matrix is positive the only leading submatrix is the matrix itself. Now suppose the claim holds for matrices of size n, and let A be an $(n+1) \times (n+1)$ symmetric positive-definite matrix. We can write A as

$$A = \left[\begin{array}{c|c} A' & w \\ \hline v & c \end{array}\right].$$

Then A' is clearly symmetric, and for any x we have $x^T A' x = \begin{pmatrix} x & 0 \end{pmatrix} A \begin{pmatrix} x \\ 0 \end{pmatrix} > 0$, so A' is positive-definite. By our induction hypothesis, every leading submatrix of A' has positive determinant, so we are left only to show that A has positive determinant. By Theorem D.4, the determinant of A is equal to the determinant of the matrix

$$B = \left[\begin{array}{c|c} c & v \\ \hline w & A' \end{array}\right].$$

Theorem D.4 also tells us that the determinant is unchanged if we add a multiple of one column of a matrix to another. Since $0 < e_{n+1}^T A e_{n+1} = c$, we can use multiples of the first column to zero out every entry in the first row other than c. Specifically, the determinant of B is the same as the determinant of the matrix obtained in this way, which looks like

$$C = \left[\begin{array}{c|c} c & 0 \\ \hline w & A'' \end{array}\right].$$

By definition, $\det(A) = c \det(A'')$. By our induction hypothesis, $\det(A'') > 0$. Since $c > 0$ as well, we conclude that $\det(A) > 0$, which completes the proof.

28.3-5

Let A_k denote the kth leading submatrix of a symmetric positive-definite matrix A. Prove that $\det(A_k)/\det(A_{k-1})$ is the kth pivot during LU decomposition, where, by convention, $\det(A_0) = 1$.

When we do an LU decomposition of a positive definite symmetric matrix, we never need to permute the rows. This means that the pivot value being used from the first operation is the entry in the upper left corner. This gets us that for the case $k = 1$, it holds because we were told to define $\det(A_0) = 1$, getting us, $a_{11} = \det(A_1)/\det(A_0)$. When Diagonalizing a matrix, the product of the pivot values used gives the determinant of the matrix. So, we have that the determinant of A_k is a product of the kth pivot value with all the previous values. By induction, the product of all the previous values is $\det(A_{k-1})$. So, we have that if x is the kth pivot value, $\det(A_k) = x \det(A_{k-1})$, giving us the desired result that the kth pivot value is $\det(A_k)/\det(A_{k-1})$.

28.3-6

Find the function of the form

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

that is the best least-squares fit to the data points

$$(1,1), (2,1), (3,3), (4,8).$$

First we form the A matrix

$$A = \begin{pmatrix} 1 & 0 & e \\ 1 & 2 & e^2 \\ 1 & 3\lg 3 & e^3 \\ 1 & 8 & e^4 \end{pmatrix}.$$

We compute the pseudoinverse, then multiply it by y, to obtain the coefficient vector

$$c = \begin{pmatrix} 0.411741 \\ -0.20487 \\ 0.16546 \end{pmatrix}.$$

28.3-7

Show that the pseudoinverse A^+ satisfies the following four equations:

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^T &= AA^+, \\ (A^+A)^T &= A^+A. \end{aligned}$$

$$\begin{aligned} AA^+A &= A((A^TA)^{-1}A^T)A \\ &= A(A^TA)^{-1}(A^TA) \\ &= A, \end{aligned}$$

$$\begin{aligned} A^+AA^+ &= ((A^TA)^{-1}A^T)A((A^TA)^{-1}A^T) \\ &= (A^TA)^{-1}(A^TA)(A^TA)^{-1}A^T \\ &= (A^TA)^{-1}A^T \\ &= A^+, \end{aligned}$$

$$\begin{aligned} (AA^+)^T &= (A(A^TA)^{-1}A^T)^T \\ &= A((A^TA)^{-1})^TA^T \\ &= A((A^TA)^T)^{-1}A^T \\ &= A(A^TA)^{-1}A^T \\ &= AA^+, \end{aligned}$$

$$\begin{aligned} (A^+A)^T &= ((A^TA)^{-1}A^TA)^T \\ &= ((A^TA)^{-1}(A^TA))^T \\ &= I^T \\ &= I \\ &= (A^TA)^{-1}(A^TA) \\ &= A^+A. \end{aligned}$$

Problem 28-1 Tridiagonal systems of linear equations

Consider the tridiagonal matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Find an LU decomposition of A.

b. Solve the equation $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^T$ by using forward and back substitution.

c. Find the inverse of A.

d. Show how, for any $n \times n$ symmetric positive-definite, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LU decomposition. Argue that any method based on forming A^{-1} is asymptotically more expensive in the worst case.

e. Show how, for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LUP decomposition.

a.

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

b. We first do back substitution to obtain that

$$Ux = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}.$$

By forward substitution, we have that

$$x = \begin{pmatrix} 5 \\ 9 \\ 12 \\ 14 \\ 15 \end{pmatrix}.$$

c. We will set $Ax = e_i$ for each i, where e_i is the vector that is all zeroes except for a one in the ith position. Then, we will just concatenate all of these solutions together to get the desired inverse.

equation	solution
$Ax_1 = e_1$	$x_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$
$Ax_2 = e_2$	$x_2 = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$
$Ax_3 = e_3$	$x_3 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 3 \\ 3 \end{pmatrix}$
$Ax_4 = e_4$	$x_4 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 4 \end{pmatrix}$
$Ax_5 = e_5$	$x_5 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$

Thus,

$$A^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 \\ 1 & 2 & 3 & 3 & 3 \\ 1 & 2 & 3 & 4 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

d. When performing the LU decomposition, we only need to take the max over at most two different rows, so the loop on line 7 of LUP-DECOMPOSITION drops to $O(1)$. There are only some constant number of nonzero entries in each row, so the loop on line 14 can also be reduced to being $O(1)$. Lastly, there are only some constant number of nonzero entries of the form a_{ik} and a_{kj}. Since the square of a constant is also a constant, this means that the nested **for** loops on lines 16-19 also only take time $O(1)$ to run. Since the **for** loops on lines 3 and 5 both run $O(n)$ times and take $O(1)$ time each to run (provided we are smart to not consider a bunch of zero entries in the matrix), the total runtime can be brought down to $O(n)$.

Since for a Tridiagonal matrix, it will only ever have finitely many nonzero entries in any row, we can do both the forward and back substitution each in time only $O(n)$.

Since the asymptotics of performing the LU decomposition on a positive definite tridiagonal matrix is $O(n)$, and this decomposition can be used to solve the equation in time $O(n)$, the total time for solving it with this method is $O(n)$. However, to simply record the inverse of the tridiagonal matrix would take time $O(n^2)$ since there are that many entries, so, any method based on computing the inverse of the matrix would take time $\Omega(n^2)$ which is clearly slower than the previous method.

e. The runtime of our LUP decomposition algorithm drops to being $O(n)$ because we know there are only ever a constant number of nonzero entries in each row and column, as before. Once we have an LUP decomposition, we also know that that decomposition have both L and U having only a constant number of non-zero entries in each row and column. This means that when we perform the forward and backward substitution, we only spend a constant amount of time per entry in x, and so, only takes $O(n)$ time.

Problem 28-2 Splines

A pratical method for interpolating a set of points with a curve is to use **cubic splines**. We are given a set $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ of $n+1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. We wish to fit a piecewise-cubic curve (spline) $f(x)$ to the points. That is, the curve $f(x)$ is made up of n cubic polynomials $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ for $i = 0, 1, \ldots, n-1$, where if x falls in the range $x_i \le x \le x_{i+1}$, then the value of the curve is given by $f(x) = f_i(x - x_i)$. The points x_i at which the cubic polynomials are "pasted" together are called **knots**. For simplicity, we shall assume that $x_i = i$ for $i = 0, 1, \ldots, n$.

To ensure continuity of $f(x)$, we require that

$$\begin{aligned} f(x_i) &= f_i(0) = y_i, \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. To ensure that $f(x)$ is sufficiently smooth, we also insist that the first derivative be continuous at each knot:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

for $i = 0, 1, \ldots, n-2$.

a. Suppose that for $i = 0, 1, \ldots, n$, we are given not only the point-value pairs $\{(x_i, y_i)\}$ but also the first derivatives $D_i = f'(x_i)$ at each knot. Express each coefficient a_i, b_i, c_i and d_i in terms of the values y_i, y_{i+1}, D_i, and D_{i+1}. (Remember that $x_i = i$.) How quickly can we compute the $4n$ coefficients from the point-value pairs and first derivatives?

The question remains of how to choose the first derivatives of $f(x)$ at the knots. One method is to require the second derivatives to be continuous at the knots:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

for $i = 0, 1, \ldots, n-2$. At the first and last knots, we assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_{n-1}''(1) = 0$; these assumptions make $f(x)$ a ***natural*** cubic spline.

b. Use the continuity constraints on the second derivative to show that for $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}). \tag{23.21}$$

c. Show that

$$2D_0 + D_1 = 3(y_1 - y_0), \tag{28.22}$$
$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \tag{28.23}$$

d. Rewrite equations (28.21)–(28.23) as a matrix equation involving the vector $D = \langle D_0, D_1, \ldots, D_n \rangle$ or unknowns. What attributes does the matrix in your equation have?

e. Argue that a natural cubic spline can interpolate a set of $n+1$ point-value pairs in $O(n)$ time (see Problem 28-1).

f. Show how to determine a natural cubic spline that interpolates a set of $n+1$ points (x_i, y_i) satisfying $x_0 < x_1 < \cdots < x_n$, even when x_i is not necessarily equal to i. What matrix equation must your method solve, and how quickly does your algorithm run?

a. We have $a_i = f_i(0) = y_i$ and $b_i = f_i'(0) = f'(x_i) = D_i$. Since $f_i(1) = a_i + b_i + c_i + d_i$ and $f_i'(1) = b_i + 2c_i + 3d_i$, we have $d_i = D_{i+1} - 2y_{i+1} + 2y_i + D_i$ which implies $c_i = 3y_{i+1} - 3y_i - D_{i+1} - 2D_i$. Since each coefficient can be computed in constant time from the known values, we can compute the $4n$ coefficients in linear time.

b. By the continuity constraints, we have $f_i''(1) = f_{i+1}''(0)$ which implies that $2c_i + 6d_i = 2c_{i+1}$, or $c_i + 3d_i = c_{i+1}$. Using our equations from above, this is equivalent to

$$D_i + 2D_{i+1} + 3y_i - 3y_{i+1} = 3y_{i+2} - 3y_{i+1} - D_{i+2} - 2D_{i+1}.$$

Rearranging gives the desired equation

$$D_i + 4D_{i+1} + D_{i+2} = 3(y_{i+2} - y_i).$$

c. The condition on the left endpoint tells us that $f_0''(0) = 0$, which implies $2c_0 = 0$. By part (a), this means $3(y_1 - y_0) = 2D_0 + D_1$. The condition on the right endpoint tells us that $f_{n-1}''(1) = 0$, which implies $c_{n-1} + 3d_{n-1} = 0$. By part (a), this means $3(y_n - y_{n-1}) = D_{n-1} + 2D_n$.

d. The matrix equation has the form $AD = Y$, where A is symmetric and tridiagonal. It looks like this:

$$\begin{pmatrix} 2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 4 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 4 & 1 & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & 4 & 1 & 0 \\ 0 & \cdots & 0 & 1 & 4 & 1 \\ 0 & \cdots & 0 & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix}$$

$$= \begin{pmatrix} 3(y_1 - y_0) \\ 3(y_2 - y_0) \\ 3(y_3 - y_1) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_n - y_{n-1}) \end{pmatrix}$$

.

e. Since the matrix is symmetric and tridiagonal, Problem 28-1 (e) tells us that we can solve the equation in $O(n)$ time by performing an LUP decomposition. By part (a), once we know each D_i we can compute each f_i in $O(n)$ time.

f. For the general case of solving the nonuniform natural cubic spline problem, we require that $f(x_{i+1}) = f_i(x_{i+1} - x_i) = y_{i+1}$, $f'(x_{i+1}) = f_i'(x_{i+1} - x_i) = f_{i+1}'(0)$ and $f''(x_{i+1}) = f_i''(x_{i+1} - x_i) = f_{i+1}''(0)$. We can still solve for each of a_i, b_i, c_i and d_i in terms of y_i, y_{i+1}, D_i and D_{i+1}, so we still get a tridiagonal matrix equation. The solution will be slightly messier, but ultimately it is solved just like the simpler case, in $O(n)$ time.

29 Linear Programming

29.1 Standard and slack forms

29.1-1

If we express the linear program in (29.24)–(29.28) in the compact notation of (29.19)–(29.21), what are n, m, A, b, and c?

$$n = m = 3,$$
$$A = \begin{pmatrix} 1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -2 & 2 \end{pmatrix},$$
$$b = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix},$$
$$c = \begin{pmatrix} 2 \\ -3 \\ 3 \end{pmatrix}.$$

29.1-2

Give three feasible solutions to the linear program in (29.24)–(29.28). What is the objective value of each one?

1. $(x_1, x_2, x_3) = (6, 1, 0)$ with objective value 9.
2. $(x_1, x_2, x_3) = (5, 2, 0)$ with objective value 4.
3. $(x_1, x_2, x_3) = (4, 3, 0)$ with objective value -1.

29.1-3

For the slack form in (29.38)–(29.41), what are N, B, A, b, c, and v?

$$N = \{1, 2, 3\},$$
$$B = \{4, 5, 6\},$$
$$A = \begin{pmatrix} 1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -2 & 2 \end{pmatrix},$$
$$b = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix},$$
$$c = \begin{pmatrix} 2 \\ -3 \\ 3 \end{pmatrix},$$
$$v = 0.$$

29.1-4

Convert the following linear program into standard form:

$$\begin{aligned} \text{minimize} \quad & 2x_1 + 7x_2 + x_3 \\ \text{subject to} \quad & \\ & x_1 - x_3 = 7 \\ & 3x_1 + x_2 \geq 24 \\ & x_2 \geq 0 \\ & x_3 \leq 0 . \end{aligned}$$

$$\begin{aligned} \text{maximize} \quad & -2x_1 - 2x_2 - 7x_3 + x_4 \\ \text{subject to} \quad & \\ & -x_1 + x_2 - x_4 \leq -7 \\ & x_1 - x_2 + x_4 \leq 7 \\ & -3x_1 + 3x_2 - x_3 \leq -24 \\ & x_1, x_2, x_3, x_4 \leq 0 . \end{aligned}$$

29.1-5

Convert the following linear program into slack form:

$$\begin{aligned} \text{maximize} \quad & 2x_1 - 6x_3 \\ \text{subject to} \quad & \\ & x_1 + x_2 - x_3 \leq 7 \\ & 3x_1 - x_2 \geq 8 \\ & -x_1 + 2x_2 + 2x_3 \geq 0 \\ & x_1, x_2, x_3 \geq 0 . \end{aligned}$$

What are the basic and nonbasic variables?

First, we will multiply the second and third inequalities by minus one to make it so that they are all $\leq$ inequalities. We will introduce the three new variables x_4, x_5, x_6, and perform the usual procedure for rewriting in slack form

$$\begin{aligned} x_4 &= 7 - x_1 - x_2 + x_3 \\ x_5 &= -8 + 3x_1 - x_2 \\ x_6 &= - x_1 + 2x_2 + 2x_3 \\ x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0 , \end{aligned}$$

where we are still trying to maximize $2x_1 - 6x_3$. The basic variables are x_4, x_5, x_6 and the nonbasic variables are x_1, x_2, x_3.

29.1-6

Show that the following linear program is infeasible:

$$\begin{aligned} \text{minimize} \quad & 3x_1 - 2x_2 \\ \text{subject to} \quad & \\ & x_1 + x_2 \leq 2 \\ & -2x_1 - 2x_2 \leq -10 \\ & x_1, x_2 \geq 0 . \end{aligned}$$

By dividing the second constraint by 2 and adding to the first, we have $0 \leq -3$, which is impossible. Therefore there linear program is unfeasible.

29.1-7

Show that the following linear program is unbounded:

$$\begin{aligned} \text{minimize} \quad & x_1 - x_2 \\ \text{subject to} \quad & \\ & -2x_1 + x_2 \leq -1 \\ & -x_1 - 2x_2 \leq -2 \\ & x_1, x_2 \geq 0 . \end{aligned}$$

For any number $r > 1$, we can set $x_1 = 2r$ and $x_2 = r$. Then, the restaints become

$$\begin{aligned} -2(2r) + r = -3r &\leq -1 \\ -2r - 2r = -4r &\leq -2 \\ 2r, r &\geq 0 . \end{aligned}$$

All of these inequalities are clearly satisfied because of our initial restriction in selecting r. Now, we look to the objective function, it is $2r - r = r$. So, since we can select r to be arbitrarily large, and still satisfy all of the constraints, we can achieve an arbitrarily large value of the objective function.

29.1-8

Suppose that we have a general linear program with n variables and m constraints, and suppose that we convert it into standard form. Give an upper bound on the number of variables and constraints in the resulting linear program.

In the worst case, we have to introduce 2 variables for every variable to ensure that we have nonnegativity constraints, so the resulting program will have $2n$ variables. If each constraint is an equality, we would have to double the number of constraints to create inequalities, resulting in $2m$ constraints. Changing minimization to maximization and greater-than signs to less-than signs don't affect the number of variables or constraints, so the upper bound is $2n$ on variables and $2m$ on constraints.

29.1-9

Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

Consider the linear program where we want to maximize $x_1 - x_2$ subject to the constraints $x_1 - x_2 \leq 1$ and $x_1, x_2 \geq 0$. clearly the objective value can never be greater than one, and it is easy to achieve the optimal value of 1, by setting

$x_1 = 1$ and $x_2 = 0$. Then, this feasible region is unbounded because for any number r, we could set $x_1 = x_2 = r$, and that would be feasible because the difference of the two would be zero which is ≤ 1.

If we further wanted it so that there was a single solution that achieved the finite optimal value, we could add the requirements that $x_1 \leq 1$.

29.2 Formulating problems as linear programs

29.2-1

Put the single-pair shortest-path linear program from (29.44)–(29.46) into standard form.

The objective is already in normal form. However, some of the constraints are equality constraints instead of $\leq$ constraints. This means that we need to rewrite them as a pair of inequality constraints, the overlap of whose solutions is just the case where we have equality. we also need to deal with the fact that most of the variables can be negative. To do that, we will introduce variables for the negative part and positive part, each of which need be positive, and we'll just be sure to subtract the negative part. d_s need not be changed in this way since it can never be negative since we are not assuming the existence of negative weight cycles.

$$d_v^+ - d_v^- - d_u^+ + d_u^- \leq w(u, v) \text{ for every edge } (u, v)$$
$$d_s \leq 0$$

29.2-2

Write out explicitly the linear program corresponding to finding the shortest path from node s to node y in Figure 24.2(a).

$$\begin{aligned} \text{minimize} \quad & d_y \\ \text{subject to} \quad & \\ & d_t \leq d_s + 3 \\ & d_x \leq d_t + 6 \\ & d_y \leq d_s + 5 \\ & d_y \leq d_t + 2 \\ & d_z \leq d_x + 2 \\ & d_t \leq d_y + 1 \\ & d_x \leq d_y + 4 \\ & d_z \leq d_y + 1 \\ & d_s \leq d_z + 1 \\ & d_x \leq d_z + 7 \\ & d_z = 0. \end{aligned}$$

29.2-3

In the single-source shortest-paths problem, we want to find the shortest-path weights from a source vertex s to all vertices $v \in V$. Given a graph G, write a linear program for which the solution has the property that d_v is the shortest-path weight from s to v for each vertex $v \in V$.

We will follow a similar idea to the way to when we were finding the shortest path between two particular vertices.

$$\begin{aligned}\text{maximize} \quad & \sum_{v \in V} d_v \\ \text{subject to} \quad & \\ & d_v \le d_u + w(u,v) \text{ for each edge } (u,v) \\ & d_s = 0.\end{aligned}$$

The first type of constraint makes sure that we never say that a vertex is further away than it would be if we just took the edge corresponding to that constraint. Also, since we are trying to maximize all of the variables, we will make it so that there is no slack anywhere, and so all the dv values will correspond to lengths of shortest paths to v. This is because the only thing holding back the variables is the information about relaxing along the edges, which is what determines shortest paths.

29.2-4

> Write out explicitly the linear program corresponding to finding the maximum flow in Figure 26.1(a).

$$\begin{aligned}\text{maximize} \quad & f_{sv_1} + f_{sv_2} \\ \text{subject to} \quad & \\ & f_{sv_1} \le 16 \\ & f_{sv_2} \le 14 \\ & f_{v_1v_3} \le 12 \\ & f_{v_2v_1} \le 4 \\ & f_{v_2v_4} \le 14 \\ & f_{v_3v_2} \le 9 \\ & f_{v_3t} \le 20 \\ & f_{v_4v_3} \le 7 \\ & f_{v_4t} \le 4 \\ & f_{sv_1} + f_{v_2v_1} = f_{v_1v_3} \\ & f_{sv_2} + f_{v_3v_2} = f_{v_2v_1} + f_{v_2v_4} \\ & f_{v_1v_3} + f_{v_4v_3} = f_{v_3v_2} + f_{v_3t} \\ & f_{v_2v_4} = f_{v_4v_3} + f_{v_4t} \\ & f_{uv} \ge 0 \text{ for } u, v \in \{s, v_1, v_2, v_3, v_4, t\}.\end{aligned}$$

29.2-5

> Rewrite the linear program for maximum flow (29.47)–(29.50) so that it uses only $O(V + E)$ constraints.

All we need to do to bring the number of constraints down from $O(V^2)$ to $O(V + E)$ is to replace the way we index the flows. Instead of indexing it by a pair of vertices, we will index it by an edge. This won't change anything about the analysis because between pairs of vertices that don't have an edge between them, there definitely won't be any flow. Also, it brings the number of constraints of the first and third time down to $O(E)$ and the number of constraints of the second kind stays at $O(V)$.

$$\begin{aligned}\text{maximize} \quad & \sum_{\text{edges e leaving s}} f_e - \sum_{\text{edges e entering s}} f_s \\ \text{subject to} \quad & \\ & f_{(u,v)} \le c(u,v) \text{ for each edge } (u,v) \\ & \sum_{\text{edges e leaving u}} f_e - \sum_{\text{edges e entering u}} f_e \text{ for each edge } u \in V - \{s,t\} \\ & f_e \ge 0 \text{ for each edge e.}\end{aligned}$$

29.2-6

> Write a linear program that, given a bipartite graph $G = (V, E)$ solves the maximum-bipartite-matching problem.

Recall from section 26.3 that we can solve the maximum-bipartite-matching problem by viewing it as a network flow problem, where we append a source s and sink t, each connected to every vertex is L and R respectively by an edge with capacity 1, and we give every edge already in the bipartite graph capacity 1. The integral maximum flows are in correspondence with maximum bipartite matchings. In this setup, the linear programming problem to solve is as follows:

$$\begin{aligned}\text{maximize} \quad & \sum_{v \in L} f_{sv} \\ \text{subject to} \quad & \\ & f_{(u,v)} \le 1 \text{ for each } u, v \in \{s\} \cup L \cup R \cup \{t\} = V \\ & \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \text{ for each } u \in L \cup R \\ & f_{uv} \ge 0 \text{ for each } u, v \in V\end{aligned}$$

29.2-7

> In the ***minimum-cost multicommodity-flow problem***, we are given directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$ and a cost $a(u, v)$. As in the multicommodity-flow problem, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. We define the flow f_i for commodity i and the aggregate flow f_{uv} on edge (u, v) as in the multicommodity-flow problem. A feasible flow is one in which the aggregate flow on each edge (u, v) is no more than the capacity of edge (u, v). The cost of a flow is $\sum_{u,v \in V} a(u, v) f_{uv}$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

As in the minimum cost flow problem, we have constraints for the edge capacities, for the conservation of flow, and nonegativity. The difference is that the restraint that before we required exactly d units to flow, now, we require that for each commodity, the right amount of that commodity flows. the conservation equalities will be applied to each different type of commodity independently. If we super script f that will denote the type of commodity the flow is describing, if we do not superscript it, it will denote the aggregate flow

We want to minimize

$$\sum_{u,v \in V} a(u, v) f_{uv}.$$

The capacity constraints are that

$$\sum_{i \in [k]} \sum_{u,v \in V} f^i_{uv} \le c(u, v).$$

The conservation constraints are that for every $i \in [k]$, for every $u \in V \setminus \{s_i, t_i\}$.

$$\sum_{v \in V} f^i_{uv} = \sum_{v \in V} f^i_{vu}.$$

Now, the constraints that correspond to requiring a certain amount of flow are that for every $i \in [k]$.

$$\sum_{v \in V} f^i_{s_i,v} - \sum_{v \in V} f^i_{v,s_i} = d.$$

Now, we put in the constraint that makes sure what we called the aggregate flow is actually the aggregate flow, so, for every $u, v \in V$,

$$f_{uv} = \sum_{i \in [k]} f^i_{uv}.$$

Finally, we get to the fact that all flows are nonnegative, for every $u, v \in V$,

$$f_{uv} \ge 0.$$

29.3 The simplex algorithm

29.3-1

> Complete the proof of Lemma 29.4 by showing that it must be the case that $c = c'$ and $v = v'$.

We subtract equation (29.81) from equation (29.79).

$$z = v + \sum_{j \in N} c_j x_j, \tag{29.79}$$

$$z = v' + \sum_{j \in N} c'_j x_j. \tag{29.81}$$

Thus we have,

$$0 = v - v' + \sum_{j \in N} (c_j - c'_j) x_j.$$

$$\sum_{j \in N} c'_j x_j = v - v' + \sum_{j \in N} c_j x_j.$$

By Lemma 29.3, we have $c_j = c'_j$ for every j and $v = v'$ since $v - v' = 0$.

29.3-2

> Show that the call to PIVOT in line 12 of SIMPLEX never decreases the value of v.

The only time v is updated in PIVOT is line 14, so it will suffice to show that $c_e \hat{b_e} \ge 0$. Prior to making the call to PIVOT, we choose an index e such that $c_e > 0$, and this is unchanged in PIVOT. We set $\hat{b_e}$ in line 3 to be b_l / a_{le}.

The loop invariant proved in Lemma 29.2 tells us that $b_l \ge 0$. The if-condition of line 6 of SIMPLEX tells us that only the noninfinite δ_i must have $a_{ie} > 0$, and we choose l to minimize δ_l, so we must have $a_{le} > 0$. Thus, $c_e \hat{b_e} \ge 0$, which implies v can never decrease.

29.3-3

> Prove that the slack form given to the PIVOT procedure and the slack form that the procedure returns are equivalent.

To show that the two slack forms are equivalent, we will show both that they have equal objective functions, and their sets of feasible solutions are equal.

First, we'll check that their sets of feasible solutions are equal. Basically all we do to the constraints when we pivot is take the non-basic variable, e, and solve the equation corresponding to the basic variable l for e. We are then taking that expression and replacing e in all the constraints with this expression we got by solving the equation corresponding to l. Since each of these algebraic operations are valid, the result of the sequence of them is also algebraically equivalent to the original.

Next, we'll see that the objective functions are equal. We decrease each c_j by $c_e \hat{a_{ej}}$, which is to say that we replace the non-basic variable we are making basic with the expression we got it was equal to once we made it basic.

Since the slack form returned by PIVOT, has the same feasible region and an equal objective function, it is equivalent to the original slack form passed in.

29.3-4

> Suppose we convert a linear program (A, b, c) in standard form to slack form. Show that the basic solution is feasible if and only if $b_i \ge 0$ for $i = 1, 2, \ldots, m$.

First suppose that the basic solution is feasible. We set each $x_i = 0$ for $1 \le i \le n$, so we have $x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j = b_i$ as a satisfied constraint. Since we also require $x_{n+i} \ge 0$ for all $1 \le i \le m$, this implies $b_i \ge 0$.

Now suppose $b_i \ge 0$ for all i. In the basic solution we set $x_i = 0$ for $1 \le i \le n$ which satisfies the nonnegativity constraints. We set $x_{n+i} = b_i$ for $1 \le i \le m$ which satisfies the other constraint equations, and also the nonnegativity constraints on the basic variables since $b_i \ge 0$. Thus, every constraint is satisfied, so the basic solution is feasible.

29.3-5

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned}\text{maximize} \quad & 18x_1 + 12.5x_2 \\ \text{subject to} \quad & \\ & x_1 + x_2 \le 20 \\ & x_1 \le 12 \\ & x_2 \le 16 \\ & x_1, x_2 \ge 0.\end{aligned}$$

First, we rewrite the linear program into it's slack form

$$\begin{aligned}\text{maximize} \quad & 18x_1 + 12.5x_2 \\ \text{subject to} \quad & \\ & x_3 = 20 - x_1 - x_2 \\ & x_4 = 12 - x_1 \\ & x_5 = 16 - x_2 \\ & x_1, x_2, x_3, x_4, x_5 \ge 0.\end{aligned}$$

We now stop since no more non-basic variables appear in the objective with a positive coefficient. Our solution is $(12, 8, 0, 0, 8)$ and has a value of 316. Going back to the standard form we started with, we just disregard the values of x_3 through x_5 and have the solution that $x_1 = 12$ and $x_2 = 8$. We can check that this is both feasible and has the objective achieve 316.

29.3-6

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned}\text{maximize} \quad & 5x_1 - 3x_2 \\ \text{subject to} \quad & \\ & x_1 - x_2 \le 1 \\ & 2x_1 + x_2 \le 2 \\ & x_1, x_2 \ge 0.\end{aligned}$$

First, we convert the linear program into it's slack form

$$\begin{aligned} z &= 5x_1 - 3x_2 \\ x_3 &= 1 - x_1 + x_2 \\ x_4 &= 2 - 2x_1 - x_2. \end{aligned}$$

The nonbasic variables are x_1 and x_2. Of these, only x_1 has a positive coefficient in the objective function, so we must choose $x_e = x_1$. Both equations limit x_1 by 1, so we'll choose the first one to rewrite x_1 with. Using $x_1 = 1 - x_3 + x_2$ we obtain the new system

$$\begin{aligned} z &= 5 - 5x_3 + 2x_2 \\ x_1 &= 1 - x_3 + x_2 \\ x_4 &= 2x_3 - 3x_2. \end{aligned}$$

Now x_2 is the only nonbasic variable with positive coefficient in the objective function, so we set $x_e = x_2$. The last equation limits x_2 by 0 which is most restrictive, so we set $x_2 = \frac{2}{3}x_3 - \frac{1}{3}x_4$. Rewriting, our new system becomes

$$\begin{aligned} z &= 5 - \tfrac{11}{3}x_3 - \tfrac{2}{3}x_4 \\ x_1 &= 1 - \tfrac{1}{3}x_3 - \tfrac{1}{3}x_4 \\ x_2 &= \tfrac{2}{3}x_3 - \tfrac{1}{3}x_4. \end{aligned}$$

Every nonbasic variable now has negative coefficient in the objective function, so we take the basic solution $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$. The objective value this achieves is 5.

29.3-7

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned}\text{minimize} \quad & x_1 + x_2 + x_3 \\ \text{subject to} \quad & \\ & 2x_1 + 7.5x_2 + 3x_3 \ge 10000 \\ & 20x_1 + 5x_2 + 10x_3 \ge 30000 \\ & x_1, x_2, x_3 \ge 0.\end{aligned}$$

First, we convert this equation to the slack form. Doing so doesn't change the objective, but the constraints become

$$\begin{aligned} z &= -x_1 - x_2 - x_3 \\ x_4 &= -10000 + 2x_1 + 7.5x_2 + 3x_3 \\ x_5 &= -30000 + 20x_1 + 5x_2 + 10x_3 \\ x_1, x_2, x_3, x_4, x_5 &\ge 0. \end{aligned}$$

Also, since the objective is to minimize a given function, we'll change it over to maximizing the negative of that function. In particular maximize $-x_1 - x_2 - x_3$. Now, we note that the initial basic solution is not feasible, because it would leave x_4 and x_5 being negative. This means that finding an initial solution requires using the method of section 29.5. The auxiliary linear program in slack form is

$$\begin{aligned} z &= -x_0 \\ x_4 &= -10000 + x_0 + 2x_1 + 7.5x_2 + 3x_3 \\ x_5 &= -30000 + x_0 + 20x_1 + 5x_2 + 10x_3 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\ge 0. \end{aligned}$$

We choose x_0 as the entering variable and x_5 as the leaving variable, since it is the basic variable whose value in the basic solution is most negative. After pivoting, we have the slack form

$$\begin{aligned} z &= -30000 + 20x_1 + 5x_2 + 10x_3 - x_5 \\ x_0 &= 30000 - 20x_1 - 5x_2 - 10x_3 + x_5 \\ x_4 &= 20000 - 18x_1 + 2.5x_2 - 7x_3 + x_5 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\ge 0. \end{aligned}$$

The associated basic solution is feasible, so now we just need to repeatedly call PIVOT until we obtain an optimal solution to L_{aux}. We'll choose x_2 as our entering variable. This gives

$$\begin{aligned} z &= -x_0 \\ x_2 &= 6000 - 0.2x_0 - 4x_1 - 2x_3 + 0.2x_5 \\ x_4 &= 35000 - 0.5x_0 - 28x_1 - 12x_3 + 1.5x_5 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\ge 0. \end{aligned}$$

This slack form is the final solution to the auxiliary problem. Since this solution has $x_0 = 0$, we know that our initial problem was feasible. Furthermore, since $x_0 = 0$, we can just remove it from the set of constraints. We then restore the original objective function, with appropriate substitutions made to include only the nonbasic variables. This yields

$$\begin{aligned} z &= -6000 + 3x_1 + x_3 - 0.2x_5 \\ x_2 &= 6000 - 4x_1 - 2x_3 + 0.2x_5 \\ x_4 &= 35000 - 28x_1 - 12x_3 + 1.5x_5 \\ x_1, x_2, x_3, x_4, x_5 &\ge 0. \end{aligned}$$

This slack form has a feasible basic solution, and we can return it to SIMPLEX. We choose x_1 as our entering variable. This gives

$$\begin{aligned} z &= -2250 - \tfrac{2}{7}x_3 - \tfrac{3}{28}x_4 - \tfrac{11}{280}x_5 \\ x_1 &= 1250 - \tfrac{3}{7}x_3 - \tfrac{1}{28}x_4 + \tfrac{15}{280}x_5 \\ x_2 &= 1000 - \tfrac{2}{7}x_3 + \tfrac{4}{28}x_4 - \tfrac{4}{280}x_5 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

At this point, all coefficients in the objective function are negative, so the basic solution is an optimal solution. This solution is $(x_1, x_2, x_3) = (1250, 1000, 0)$.

29.3-8

> In the proof of Lemma 29.5, we argued that there are at most $\binom{m+n}{n}$ ways to choose a set B of basic variables. Give an example of a linear program in which there are strictly fewer than $\binom{m+n}{n}$ ways to choose the set B.

Consider the simple program

$$\begin{aligned} z &= -x_1 \\ x_2 &= 1 - x_1 . \end{aligned}$$

In this case, we have $m = n = 1$, so $\binom{m+n}{n} = \binom{2}{1} = 2$, however, since the only coefficients of the objective function are negative, we can't make any other choices for basic variable. We must immediately terminate with the basic solution $(x_1, x_2) = (0, 1)$, which is optimal.

29.4 Duality

29.4-1

> Formulate the dual of the linear program given in Exercise 29.3-5.

By just transposing A, swapping b and c, and switching the maximization to a minimization, we want to minimize $20y_1 + 12y_2 + 16y_3$ subject to the constrain

$$\begin{aligned} y_1 + y_2 &\geq 18 \\ y_1 + y_3 &\geq 12.5 \\ y_1, y_2, y_3 &\geq 0 \end{aligned}$$

29.4-2

> Suppose that we have a linear program that is not in standard form. We could produce the dual by first converting it to standard form, and then taking the dual. It would be more convenient, however, to be able to produce the dual directly. Explain how we can directly take the dual of an arbitrary linear program.

By working through each aspect of putting a general linear program into standard form, as outlined on page 852, we can show how to deal with transforming each into the dual individually. If the problem is a minimization instead of a maximization, replace c_j by $-c_j$ in (29.84). If there is a lack of nonnegativity constraint on x_i we duplicate the jth column of A, which corresponds to duplicating the jth row of A^T. If there is an equality constraint for b_i, we convert it to two inequalities by duplicating then negating the ith column of A^T, duplicating then negating the ith entry of b, and adding an extra y_i variable. We handle the greater-than-or-equal-to sign $\sum_{i=1}^{n} a_{ij}x_j \geq b_i$ by negating ith column of A^T and negating b_i. Then we solve the dual problem of minimizing $b^T y$ subject to $A^T y$ and $y \geq 0$.

29.4-3

> Write down the dual of the maximum-flow linear program, as given in lines (29.47)–(29.50) on page 860. Explain how to interpret this formulation as a minimum-cut problem.

First, we'll convert the linear program for maximum flow described in equation (29.47)–(29.50) into standard form. The objective function says that c is a vector indexed by a pair of vertices, and it is positive one if s is the first index and negative one if s is the second index (zero if it is both). Next, we'll modify the constraints by switching the equalities over into inequalities to get

$$\begin{aligned} f_{uv} &\leq c(u, v) && \text{for each } u, v \in V \\ \sum_{u \in V} f_{vu} &\leq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\ \sum_{u \in V} f_{vu} &\geq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\ f_{uv} &\geq 0 && \text{for each } u, v \in V \end{aligned}$$

Then, we'll convert all but the last set of the inequalities to be $\leq$ by multiplying the third line by -1.

$$\begin{aligned} f_{uv} &\leq c(u, v) && \text{for each } u, v \in V \\ \sum_{u \in V} f_{vu} &\leq \sum_{u \in V} f_{uv} && \text{for each } v \in V - \{s, t\} \\ \sum_{u \in V} -f_{vu} &\leq \sum_{u \in V} -f_{uv} && \text{for each } v \in V - \{s, t\} \\ f_{uv} &\geq 0 && \text{for each } u, v \in V \end{aligned}$$

Finally, we'll bring all the variables over to the left to get

$$\begin{aligned} f_{uv} &\leq c(u, v) && \text{for each } u, v \in V \\ \sum_{u \in V} f_{vu} - \sum_{u \in V} f_{uv} &\leq 0 && \text{for each } v \in V - \{s, t\} \\ \sum_{u \in V} -f_{vu} - \sum_{u \in V} -f_{uv} &\leq 0 && \text{for each } v \in V - \{s, t\} \\ f_{uv} &\geq 0 && \text{for each } u, v \in V \end{aligned}$$

Now, we can finally write down our A and b. A will be a $|V|^2 \times |V|^2 + 2|V| - 4$ matrix built from smaller matrices A_1 and A_2 which correspond to the three types of constraints that we have (of course, not counting the non-negativity constraints). We will let $g(u, v)$ be any bijective mapping from $V \times V$ to $[|V|^2]$. We'll also let h be any bijection from $V - \{s, t\}$ to $[|V| - 2]$.

$$A = \begin{pmatrix} A_1 \\ A_2 \\ -A_2 \end{pmatrix},$$

where A_1 is defined as having its row $g(u, v)$ be all zeroes except for having the value 1 at at the $g(u, v)$th entry. We define A_2 to have it's row $h(u)$ be equal to 1 at all columns j for which $j = g(v, u)$ for some v and equal to -1 at all columns j for which $j = g(u, v)$ for some v. Lastly, we mention that b is defined as having it's jth entry be equal to $c(u, v)$ if $j = g(u, v)$ and zero if $j > |V|^2$.

Now that we have placed the linear program in standard form, we can take its dual. We want to minimize $\sum_{i=1}^{|V|^2+2|V|-2} b_i y_i$ given the constraints that all the y values are non-negative, and $A^T y \geq c$.

29.4-4

> Write down the dual of the minimum-cost-flow linear program, as given in lines (29.51)–(29.52) on page 862. Explain how to interpret this problem in terms of graphs and flows.

First we need to put the linear programming problem into standard form, as follows:

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_{(u,v) \in E} -a(u, v) f_{uv} \\ \text{subject to} \quad & \\ f_{uv} &\leq c(u, v) && \text{for each } u, v \in V \\ \textstyle\sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} &\leq 0 && \text{for each } u \in V - \{s, t\} \\ \textstyle\sum_{v \in V} f_{uv} - \sum_{v \in V} f_{vu} &\leq 0 && \text{for each } u \in V - \{s, t\} \\ \textstyle\sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} &\leq d \\ \textstyle\sum_{v \in V} f_{vs} - \sum_{v \in V} f_{sv} &\leq -d \\ f_{uv} &\geq 0 . \end{aligned}$$

We now formulate the dual problem. Let the vertices be denoted $v_1, v_2, \ldots, v_n, s, t$ and the edges be $e_1, e_2, \ldots, e_k$. Then we have $b_i = c(e_i)$ for $1 \leq i \leq k$, $b_i = 0$ for $k + 1 \leq i \leq k + 2n$, $b_{k+2n+1} = d$, and $b_{k+2n+2} = -d$. We also have $c_i = -a(e_i)$ for $1 \leq i \leq k$. For notation, let $j.\mathrm{left}$ denote the tail of edge e_j and $j.\mathrm{right}$ denote the head. Let $\chi_a(e_j) = 1$ if e_j enters s, set it equal to -1 if e_j leaves s, and set it equal to 0 if e_j is not incident with s. The dual problem is:

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_{i=1}^{k} c(e_j) y_i + d y_{k+2n+1} - d y_{k+2n+2} \\ \text{subject to} \quad & \\ & y_j + y_{k+e_j.\mathrm{right}} - y_{k+j.\mathrm{left}} - y_{k+n+e_j.\mathrm{right}} + y_{k+n+e_j.\mathrm{left}} - \chi_a(e_j) y_{k+2n+1} + \chi_a(e_j) y_{k+2n+2} \geq -a(e_j), \end{aligned}$$

where j runs between 1 and k. There is one constraint equation for each edge e_j.

29.4-5

> Show that the dual of the dual of a linear program is the primal linear program.

Suppose that our original linear program is in standard form for some A, b, c. Then, the dual of this is to minimize $\sum_{i=1}^{m} b_i y_i$ subject to $A^T y \geq c$ This can be rewritten as wanting to maximize $\sum_{i=1}^{m} (-b_i) y_i$ subject to $(-A)^T y \leq -c$. Since this is a standard form, we can take its dual easily, it is minimize $\sum_{j=1}^{n} (-c_j) x_j$ subject to $(-A)x \geq -b$. This is the same as minimizing $\sum_{j=1}^{n} c_j x_j$ subject to $Ax \leq b$, which was the original linear program.

29.4-6

> Which result from Chapter 26 can be interpreted as weak duality for the maximum-flow problem?

Corollary 26.5 from Chapter 26 can be interpreted as weak duality.

29.5 The initial basic feasible solution

29.5-1

> Give detailed pseudocode to implement lines 5 and 14 of INITIALIZE-SIMPLEX.

For line 5, first let (N, B, A, b, c, v) be the result of calling PIVOT on L_{aux} using x_0 as the entering variable. Then repeatedly call PIVOT until an optimal solution to L_{aux} is obtained, and return this to (N, B, A, b, c, v). To remove x_0 from the constraints, set $a_{i,0} = 0$ for all $i \in B$, and set $N = N \setminus \{0\}$. To restore the original objective function of L, for each $j \in N$ and each $i \in B$, set $c_j = c_j - c_i a_{ij}$.

29.5-2

> Show that when the main loop of SIMPLEX is run by INITIALIZE-SIMPLEX, it can never return "unbounded."

In order to enter line 10 of INITIALIZE-SIMPLEX and begin iterating the main loop of SIMPLEX, we must have recovered a basic solution which is feasible for L_{aux}. Since $x_0 \geq 0$ and the objective function is $-x_0$, the objective value associated to this solution (or any solution) must be negative. Since the goal is to aximize, we have an upper bound of 0 on the objective value. By Lemma 29.2, SIMPLEX correctly determines whether or not the input linear program is unbounded. Since L_{aux} is not unbounded, this can never be returned by SIMPLEX.

29.5-3

> Suppose that we are given a linear program L in standard form, and suppose that for both L and the dual of L, the basic solutions associated with the initial slack forms are feasible. Show that the optimal objective value of L is 0.

Since it is in standard form, the objective function has no constant term, it is entirely given by $\sum_{i=1}^{n} c_i x_i$, which is going to be zero for any basic solution. The same thing goes for its dual. Since there is some solution which has the objective function achieve the same value both for the dual and the primal, by the corollary to the weak duality theorem, that common value must be the optimal value of the objective function.

29.5-4

> Suppose that we allow strict inequalities in a linear program. Show that in this case, the fundamental theorem of linear programming does not hold.

Consider the linear program in which we wish to maximize x_1 subject to the constraint $x_1 < 1$ and $x_1 \geq 0$. This has no optimal solution, but it is clearly bounded and has feasible solutions. Thus, the Fundamental theorem of linear programming does not hold in the case of strict inequalities.

29.5-5

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned} \text{maxmize} \quad & x_1 + 3x_2 \\ \text{subject to} \quad & \\ & x_1 - x_2 \leq 8 \\ & -x_1 - x_2 \leq -3 \\ & -x_1 + 4x_2 \leq 2 \\ & x_1, x_2 \geq 0 . \end{aligned}$$

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program,

$$\begin{aligned} \text{maxmize} \quad & -x_0 \\ \text{subject to} \quad & \\ & -x_0 + x_1 - x_2 \leq 8 \\ & -x_0 - x_1 - x_2 \leq -3 \\ & -x_0 - x_1 + 4x_2 \leq 2 \\ & x_0, x_1, x_2 \geq 0 . \end{aligned}$$

Writing this linear program in slack form,

$$\begin{aligned} z &= -x_0 \\ x_3 &= 8 + x_0 - x_1 + x_2 \\ x_4 &= -3 + x_0 + x_1 + x_2 \\ x_5 &= 2 + x_0 + x_1 - 4x_2 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable.

$$\begin{aligned} z &= -3 + x_1 + x_2 - x_4 \\ x_0 &= 3 - x_1 - x_2 + x_4 \\ x_3 &= 11 - 2x_1 + x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

The basic solution is feasible, so we repeatedly call PIVOT to get the optimal solution to L_{aux}. We'll choose x_1 to be our entering variable and x_0 to be the leaving variable. This gives

$$\begin{aligned} z &= -x_0 \\ x_1 &= 3 - x_0 - x_2 + x_4 \\ x_3 &= 5 + 2x_0 + 2x_2 - x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

The basic solution is now optimal for L_{aux}, so we return this slack form to SIMPLEX, set $x_0 = 0$, and update the objective function which yields

$$\begin{aligned} z &= 3 + 2x_2 + x_4 \\ x_1 &= 3 - x_2 + x_4 \\ x_3 &= 5 + 2x_2 - x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

We'll choose x_2 as our entering variable, which makes x_5 our leaving variable. PIVOT then gives,

$$\begin{aligned} z &= 5 + (7/5)x_4 - (2/5)x_5 \\ x_1 &= 2 + (1/5)x_4 + (1/5)x_5 \\ x_2 &= 1 + (4/5)x_4 - (1/5)x_5 \\ x_3 &= 7 - (3/5)x_4 - (2/5)x_5 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

We'll choose x_4 as our entering variable, which makes x_3 our leaving variable. PIVOT then gives,

$$\begin{aligned} z &= (64/3) - (7/3)x_3 - (4/3)x_5 \\ x_1 &= (34/3) - (4/3)x_3 - (1/3)x_5 \\ x_2 &= (10/3) - (1/3)x_3 - (1/3)x_5 \\ x_4 &= (35/3) - (5/3)x_3 - (2/3)x_5 \\ x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Now all coefficients in the objective function are negative, so the basic solution is the optimal solution. It is $(x_1, x_2) = (34/3, 10/3)$.

29.5-6

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned} \text{maxmize} \quad & x_1 - 2x_2 \\ \text{subject to} \quad & \\ & x_1 + 2x_2 \leq 4 \\ & -2x_1 - 6x_2 \leq -12 \\ & x_2 \leq 1 \\ & x_1, x_2 \geq 0 . \end{aligned}$$

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program,

$$\begin{aligned} \text{maxmize} \quad & -x_0 \\ \text{subject to} \quad & \\ & -x_0 + x_1 + 2x_2 \leq 4 \\ & -x_0 - 2x_1 - 6x_2 \leq -12 \\ & -x_0 + x_2 \leq 1 \\ & x_0, x_1, x_2 \geq 0 . \end{aligned}$$

Writing this linear program in slack form,

$$\begin{aligned} z &= -x_0 \\ x_3 &= 4 + x_0 - x_1 - 2x_2 \\ x_4 &= -12 + x_0 + 2x_1 + 6x_2 \\ x_5 &= 1 + x_0 - x_2 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable.

$$\begin{aligned} z &= -12 + 2x_1 + 6x_2 - x_4 \\ x_0 &= 12 - 2x_1 - 6x_2 + x_4 \\ x_3 &= 16 - 3x_1 - 8x_2 + x_4 \\ x_5 &= 13 - 2x_1 - 8x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

The basic solution is $(x_0, x_1, x_2, x_3, x_4, x_5) = (12, 0, 0, 16, 0, 13)$ which is feasible for the auxiliary program. Now we need to run SIMPLEX to find the optimal objective value to L_{aux}. Let x_1 be our next entering variable. It is most

constrained by x_3, which will be our leaving variable. After PIVOT, the new linear program is

$$\begin{aligned} z &= -(4/3) + (2/3)x_2 - (2/3)x_3 + (1/3)x_4 \\ x_0 &= (4/3) - (2/3)x_2 + (2/3)x_3 + (1/3)x_4 \\ x_1 &= (16/3) - (8/3)x_2 - (1/3)x_3 + (1/3)x_4 \\ x_5 &= (7/3) - (8/3)x_2 + (2/3)x_3 + (1/3)x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Every coefficient in the objective function is negative, so we take the basic solution $(x_0, x_1, x_2, x_3, x_4, x_5) = (4/3, 16/3, 0, 0, 0, 7/3)$ which is also optimal. Since $x_0 \neq 0$, the original linear program must be unfeasible.

29.5-7

> Solve the following linear program using SIMPLEX:
>
> $$\begin{aligned} \text{maxmize} \quad & x_1 + 3x_2 \\ \text{subject to} \quad & \\ & -x_1 + x_2 \leq -1 \\ & -x_1 - x_2 \leq -3 \\ & -x_1 + 4x_2 \leq 2 \\ & x_1, x_2 \geq 0 . \end{aligned}$$

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program,

$$\begin{aligned} \text{maxmize} \quad & -x_0 \\ \text{subject to} \quad & \\ & -x_0 - x_1 + x_2 \leq -1 \\ & -x_0 - x_1 - x_2 \leq -3 \\ & -x_0 - x_1 + 4x_2 \leq 2 \\ & x_0, x_1, x_2 \geq 0 . \end{aligned}$$

Writing this linear program in slack form,

$$\begin{aligned} z &= -x_0 \\ x_3 &= -1 + x_0 + x_1 - x_2 \\ x_4 &= -3 + x_0 + x_1 + x_2 \\ x_5 &= 2 + x_0 + x_1 - 4x_2 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Next we make one call to PIVOT where x_0 is the entering variable and x_4 is the leaving variable.

$$\begin{aligned} z &= -3 + x_1 + x_2 - x_4 \\ x_0 &= 3 - x_1 - x_2 + x_4 \\ x_3 &= 2 - 2x_2 + x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Let x_1 be our entering variable. Then x_0 is our leaving variable, and we have

$$\begin{aligned} z &= -x_0 \\ x_1 &= 3 - x_0 - x_2 + x_4 \\ x_3 &= 2 - 2x_2 + x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

The basic solution is feasible, and optimal for L_{aux}, so we return this and run SIMPLEX. Updating the objective function and setting $x_0 = 0$ gives

$$\begin{aligned} z &= 3 + 2x_2 + x_4 \\ x_1 &= 3 - x_2 + x_4 \\ x_3 &= 2 - 2x_2 + x_4 \\ x_5 &= 5 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

We'll choose x_2 as our entering variable, which makes x_3 our leaving variable. This gives

$$\begin{aligned} z &= 5 - x_3 + 2x_4 \\ x_1 &= 2 + (1/2)x_3 + (1/2)x_4 \\ x_2 &= 1 - (1/2)x_3 + (1/2)x_4 \\ x_5 &= (5/2)x_3 - (3/2)x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Next we use x_4 as our entering variable, which makes x_5 our leaving variable. This gives

$$\begin{aligned} z &= 5 + (7/3)x_3 - (4/3)x_5 \\ x_1 &= 2 + (4/3)x_3 - (1/3)x_5 \\ x_2 &= 1 + (1/3)x_3 - (1/3)x_5 \\ x_4 &= (5/3)x_3 - (2/3)x_5 \\ x_0, x_1, x_2, x_3, x_4, x_5 &\geq 0 . \end{aligned}$$

Finally, we would like to choose x_3 as our entering variable, but every coefficient on x_3 is positive, so SIMPLEX returns that the linear program is unbounded.

29.5-8

> Solve the linear program given in (29.6)–(29.10).

We first put the linear program in standard form,

$$\begin{aligned} \text{maxmize} \quad & x_1 + x_2 + x_3 + x_4 \\ \text{subject to} \quad & \\ & 2x_1 - 8x_2 - 10x_4 \leq -50 \\ & -5x_1 - 2x_2 \leq -100 \\ & -3x_1 + 5x_2 - 10x_3 + 2x_4 \leq -25 \\ & x_1, x_2, x_3, x_4 \geq 0 . \end{aligned}$$

The initial basic solution isn't feasible, so we will need to form the auxiliary linear program.

$$\begin{aligned} z &= -x_0 \\ x_5 &= -50 + x_0 - 2x_1 + 8x_2 + 10x_4 \\ x_6 &= -100 + x_0 + 5x_1 + 2x_2 \\ x_7 &= -25 + x_0 + 3x_1 - 5x_2 + 10x_3 - 2x_4 \\ x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 &\geq 0 . \end{aligned}$$

The index of the minimum b_i is 2, so we take x_0 to be our entering variable and x_6 to be our leaving variable. The call to PIVOT on line 8 yields

$$\begin{aligned} z &= -100 + 5x_1 + 2x_2 - x_6 \\ x_0 &= 100 - 5x_1 - 2x_2 + x_6 \\ x_5 &= 50 - 7x_1 + 8x_2 + 10x_4 + x_6 \\ x_7 &= 75 - 2x_1 - 7x_2 + 10x_3 - 2x_4 + x_6 \\ x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 &\geq 0 . \end{aligned}$$

Next we'll take x_2 to be our entering variable and x_5 to be our leaving variable. The call to PIVOT yields

$$\begin{aligned} z &= -225/2 + (27/4)x_1 - (10/4)x_4 + (1/4)x_5 - (5/4)x_6 \\ x_0 &= 225/2 - (27/4)x_1 + (10/4)x_4 - (1/4)x_5 + (5/4)x_6 \\ x_2 &= -50/8 + (7/8)x_1 - (10/8)x_4 + (1/8)x_5 - (1/8)x_6 \\ x_7 &= 475/4 - (65/8)x_1 + 10x_3 + (54/8)x_4 - (7/8)x_5 + (15/8)x_6 \\ x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 &\geq 0 . \end{aligned}$$

The work gets rather messy, but INITIALIZE-SIMPLEX does eventually give a feasible solution to the linear program, and after running the simplex method we find that $(x_1, x_2, x_3, x_4) = (175/11, 225/22, 125/44, 0)$ is an optimal solution to the original linear programming problem.

29.5-9

> Consider the following 1-variable linear program, which we call P:
>
> $$\begin{aligned} \text{maximize} \quad & tx \\ \text{subject to} \quad & rx \leq s \\ & x \geq 0 , \end{aligned}$$
>
> where r, s, and t are arbitrary real numbers. Let D be the dual of P.
>
> State for which values of r, s, and t you can assert that
>
> 1. Both P and D have optimal solutions with finite objective values.
> 2. P is feasible, but D is infeasible.
> 3. D is feasible, but P is infeasible.
> 4. Neither P nor D is feasible.

1. One option is that $r = 0$, $s \geq 0$ and $t \leq 0$. Suppose that $r > 0$, then, if we have that s is non-negative and t is non-positive, it will be as we want.
2. We will split into two cases based on r. If $r = 0$, then this is exactly when t is non-positive and s is non-negative. The other possible case is that r is negative, and t is positive. In which case, because r is negative, we can always get rx as small as we want so s doesn't matter, however, we can never make rx positive so it can never be $\geq t$.
3. Again, we split into two possible cases for r. If $r = 0$, then it is when t is nonnegative and s is non-positive. The other possible case is that r is positive, and s is negative. Since r is positive, rx will always be non-negative, so it cannot be $\leq s$. But since r is positive, we have that we can always make rx as big as we want, in particular, greater than t.
4. If we have that $r = 0$ and t is positive and s is negative. If r is nonzero, then we can always either make rx really big or really small depending on the sign of r, meaning that either the primal or the dual would be feasable.

Problem 29-1 Linear-inequality feasibility

> Given a set of m linear inequalities on n variables $x_1, x_2, \ldots, x_n$, the ***linear-inequality feasibility problem*** asks whether there is a setting of the variables that simultaneously satisfies each of the inequalities.
>
> **a.** Show that if we have an algorithm for linear programming, we can use it to solve a linear-inequality feasibility problem. The number of variables and constraints that you use in the linear-programming problem should be polynomial in n and m.
>
> **b.** Show that if we have an algorithm for the linear-inequality feasibility problem, we can use it to solve a linear-programming problem. The number of variables and linear inequalities that you use in the linear-inequality feasibility problem should be polynomial in n and m, the number of variables and constraints in the linear program.

a. We just let the linear inequalities that we need to satisfy be our set of constraints in the linear program. We let our function to maximize just be a constant. The solver for linear programs would fail to detect any feasible solution if the linear constraints were not feasible. If the linear programming solver returns any solution at all, we know that the linear constraints are feasible.

b. Suppose that we are trying to solve the linear program in standard form with some particular A, b, c. That is, we want to maximize $\sum_{j=1}^{n} c_j x_j$ subject to $Ax \leq b$ and all entries of the x vector are non-negative. Now, consider the dual program, that is, we want to minimize $\sum_{i=1}^{m} b_i y_i$ subject to $A^T y \geq c$ and all the entries in the y vector are nonzero. We know by Corollary 29.9, if x and y are feasible solutions to their respective problems, then, if we have that their objective functions are equal, then, they are both optimal solutions.

We can force their objective functions to be equal. To do this, let c_k be some nonzero entry in the c vector. If there are no nonzero entries, then the function we are trying to optimize is just the zero function, and it is exactly a feasibility question, so we we would be done. Then, we add two linear inequalities to require $x_k = \frac{1}{c_k}\left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j\right)$. This will require that whatever values the variables take, their objective functions will be equal. Lastly, we just throw these in with the inequalities we already had. So, the constraints will be:

$$\begin{aligned} Ax &\leq b \\ A^T y &\geq c \\ x_k &\leq \frac{1}{c_k}\left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j\right) \\ x_k &\geq \frac{1}{c_k}\left(\sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j\right) \\ x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m &\geq 0. \end{aligned}$$

We have a number of variables equal to $n + m$ and a number of constraints equal to $2 + 2n + 2m$, so both are polynomial in n and m. Also, any assignment of variables which satisfy all of these constraints will be a feasible solution to both the problem and its dual that cause the respective objective functions to take the same value, and so, must be an optimal solution to both the original problem and its dual. This of course assumes that the linear inequality feasibility solver doesn't merely say that the inequalities are satisfiable, but actually returns a satisfying assignment.

Lastly, it is necessary to note that if there is some optimal solution x, then, we can obtain an optimal solution for the dual that makes the objective functions equal by theorem 29.10. This ensures that the two constraints we added to force the objectives of the primal and the dual to be equal don't cause us to change the optimal solution to the linear program.

Problem 29-2 Complementary slackness

> ***Complementary slackness*** describes a relationship between the values of primal variables and dual constraints and between the values of dual variables and primal constraints. Let $\bar{x}$ be a feasible solution to the primal linear program given in (29.16)–(29.18), and let $\bar{y}$ be a feasible solution to the dual linear program given in (29.83)–(29.85). Complementary slackness states that the following conditions are necessary and sufficient for $\bar{x}$ and $\bar{y}$ to be optimal:
>
> $$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ or } \bar{x}_j = 0 \text{ for } j = 1, 2, \ldots, n$$
>
> and
>
> $$\sum_{j=1}^{m} a_{ij}\bar{x}_j = b_i \text{ or } \bar{y}_i = 0 \text{ for } j = 1, 2, \ldots, m.$$
>
> **a.** Verify that complementary slackness holds for the linear program in lines (29.53)–(29.57).
>
> **b.** Prove that complementary slackness holds for any primal linear program and its corresponding dual.
>
> **c.** Prove that a feasible solution $\bar{x}$ to a primal linear program given in lines (29.16)–(29.18) is optimal if and only if there exist values $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ such that
>
> 1. $\bar{y}$ is a feasible solution to the dual linear program given in (29.83)–(29.85),
> 2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ for all j such that $\bar{x}_j > 0$, and
> 3. $\bar{y}_i = 0$ for all i such that $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

a. An optimal solution to the LP program given in (29.53)-(29.57) is $(x_1, x_2, x_3) = (8, 4, 0)$. An optimal solution to the dual is $(y_1, y_2, y_3) = (0, 1/6, 2/3)$. It is then straightforward to verify that the equations hold.

b. First suppose that complementary slackness holds. Then the optimal objective value of the primal problem is, if it exists,

$$\begin{aligned} \sum_{k=1} c_k x_k &= \sum_{k=1}^{n}\sum_{i=1}^{m} a_{ik} y_i x_k \\ &= \sum_{i=1}^{m}\sum_{k=1}^{n} a_{ik} x_k y_i \\ &= \sum_{i=1}^{m} b_i y_i, \end{aligned}$$

which is precisely the optimal objective value of the dual problem. If any x_j is 0, then those terms drop out of them sum, so we can safely replace c_k by whatever we like in those terms. Since the objective values are equal, they must be optimal. An identical argument shows that if an optimal solution exists for the dual problem then any feasible solution for the primal problem which satisfies the second equality of complementary slackness must also be optimal.

Now suppose that x and y are optimal solutions, but that complementary slackness fails. In other words, there exists some j such that $x_j \neq 0$ but $\sum_{i=1}^{m} a_{ij} y_i > c_j$, or there exists some i such that $y_i \neq 0$ but $\sum_{j=1}^{n} a_{ij} x_j < b_i$. In the first case we have

$$\begin{aligned} \sum_{k=1} c_k x_k &< \sum_{k=1}^{n}\sum_{i=1}^{m} a_{ik} y_i x_k \\ &= \sum_{i=1}^{m}\sum_{k=1}^{n} a_{ik} x_k y_i \\ &= \sum_{i=1}^{m} b_i y_i. \end{aligned}$$

This implies that the optimal objective value of the primal solution is strictly less than the optimal value of the dual solution, a contradiction. The argument for the second case is identical. Thus, x and y are optimal solutions if and only if complementary slackness holds.

c. This follows immediately from part (b). If x is feasible and y satisfies conditions 1, 2, and 3, then complementary slackness holds, so x and y are optimal. On the other hand, if x is optimal, then the dual linear program must have an optimal solution y as well, according to Theorem 29.10. Optimal solutions are feasible, and by part (b), x and y satisfy complementary slackness. Thus, conditions 1, 2, and 3 hold.

Problem 29-3 Integer linear programming

> An ***integer linear-programming problem*** is a linear-programming problem with the additional constraint that the variables x must take on integral values. Exercise 34.5-3 shows that just determining whether an integer linear program has a feasible solution is NP-hard, which means that there is no known polynomial-time algorithm for this problem.
>
> **a.** Show that weak duality (Lemma 29.8) holds for an integer linear program.
>
> **b.** Show that duality (Theorem 29.10) does not always hold for an integer linear program.
>
> **c.** Given a primal linear program in standard form, let us define P to be the optimal objective value for the primal linear program, D to be the optimal objective value for its dual, IP to be the optimal objective value for the integer version of the primal (that is, the primal with the added constraint that the variables take on integer values), and

ID to be the optimal objective value for the integer version of the dual. Assuming that both the primal integer program and the dual integer program are feasible and bounded, show that

$$\mathrm{IP} \le \mathrm{P} = \mathrm{D} \le \mathrm{ID}.$$

a. The proof for weak duality goes through identically. Nowhere in it does it use the integrality of the solutions.

b. Consider the linear program given in standard form by $A = (1)$, $b = (\frac{1}{2})$ and $c = (2)$. The highest we can get this is 0 since that's that only value that x can be. Now, consider the dual to this, that is, we are trying to minimize $\frac{x}{2}$ subject to the constraint that $x \ge 2$. This will be minimized when $x = 2$, so, the smallest solution we can get is 1.

Since we have just exhibited an example of a linear program that has a different optimal solution as it's dual, the duality theorem does not hold for integer linear program.

c. The first inequality comes from looking at the fact that by adding the restriction that the solution must be integer valued, we obtain a set of feasible solutions that is a subset of the feasible solutions of the original primal linear program. Since, to get IP, we are taking the max over a subset of the things we are taking a max over to get P, we must get a number that is no larger. The third inequality is similar, except since we are taking min over a subset, the inequality goes the other way. The middle equality is given by Theorem 29.10.

Problem 29-4 Farkas'ss lemma

Let A be an $m \times n$ matrix and c be an n-vector. Then Farkas's lemma states that exactly one of the systems

$$\begin{aligned} Ax &\le 0, \\ c^{\mathrm{T}}x &> 0 \end{aligned}$$

and

$$\begin{aligned} A^{\mathrm{T}}y &= c, \\ y &\le 0 \end{aligned}$$

is solvable, where x is an n-vector and y is an m-vector. Prove Farkas's lemma.

Suppose that both systems are solvable, let x denote a solution to the first system, and y denote a solution to the second. Taking transposes we have $x^{\mathrm{T}}A^{\mathrm{T}} \le 0^{\mathrm{T}}$. Right multiplying by y gives $x^{\mathrm{T}}c = x^{\mathrm{T}}A^{\mathrm{t}}y \le 0^{\mathrm{T}}$, which is a contradiction to the fact that $c^{\mathrm{T}}x > 0$. Thus, both systems cannot be simultaneously solved. Now suppose that the second system fails. Consider the following linear program:

$$\text{maximize } 0x \text{ subject to } A^{\mathrm{T}}y = c \text{ and } y \ge 0,$$

and its corresponding dual program

$$\text{minimize } -c^{\mathrm{T}}x \text{ subject to } Ax \le 0.$$

Since the second system fails, the primal is infeasible. However, the dual is always feasible by taking $x = 0$. If there were a finite solution to the dual, then duality says there would also be a finite solution to the primal. Thus, the dual must be unbounded. Thus, there must exist a vector x which makes $-c^{\mathrm{T}}x$ arbitrarily small, implying that there exist vectors x for which $c^{\mathrm{T}}x$ is strictly greater than 0. Thus, there is always at least one solution.

Problem 29-5 Minimum-cost circulation

In this problem, we consider a variant of the minimum-cost-flow problem from Section 29.2 in which we are not given a demand, a source, or a sink. Instead, we are given, as before, a flow network and edge costs $a(u, v)$ flow is feasible if it satisfies the capacity constraint on every edge and flow conservation at *every* vertex. The goal is to find, among all feasible flows, the one of minimum cost. We call this problem the ***minimum-cost-circulation problem***.

a. Formulate the minimum-cost-circulation problem as a linear program.

b. Suppose that for all edges $(u, v) \in E$, we have $a(u, v) > 0$. Characterize an optimal solution to the minimum-cost-circulation problem.

c. Formulate the maximum-flow problem as a minimum-cost-circulation problem linear program. That is given a maximum-flow problem instance $G = (V, E)$ with source s, sink t and edge capacities c, create a minimum-cost-circulation problem by giving a (possibly different) network $G' = (V', E')$ with edge capacities c' and edge costs a' such that you can discern a solution to the maximum-flow problem from a solution to the minimum-cost-circulation problem.

d. Formulate the single-source shortest-path problem as a minimum-cost-circulation problem linear program.

a. This is exactly the linear program given in equations (29.51)-(29.52) except that the equation on the third line of the constraints should be removed, and for the equation on the second line of the constraints, u should be selected from all of V instead of from $V \setminus \{s, t\}$.

b. If $a(u, v) > 0$ for every pair of vertices, then, there is no point in sending any flow at all. So, an optimal solution is just to have no flow. This obviously satisfies the capacity constraints, it also satisfies the conservation constraints because the flow into and out of each vertex is zero.

c. We assume that the edge (t, s) is not in E because that would be a silly edge to have for a maximum flow from s to t. If it is there, remove it and it won't decrease the maximum flow. Let $V' = V$ and $E' = E \cup \{(t, s)\}$. For the edges of E' that are in E, let the capacity be as it is in E and let the cost be 0. For the other edge, we set $c(t, s) = \infty$ and $a(t, s) = -1$. Then, if we have any circulation in G', it will be trying to get as much flow to go across the edge (t, s) in order to drive the objective function lower, the other flows will have no affect on the objective function. Then, by Kirchhoff's current law (a.k.a. common sense), the amount going across the edge (t, s) is the same as the total flow in the rest of the network from s to t. This means that maximizing the flow across the edge (t, s) is also maximizing the flow from s to t. So, all we need to do to recover the maximum flow for the original network is to keep the same flow values, but throw away the edge (t, s).

d. Suppose that s is the vertex that we are computing shortest distance from. Then, we make the circulation network by first starting with the original graph, giving each edge a cost of whatever it was before and infinite capacity. Then, we place an edge going from every vertex that isn't s to s that has a capacity of 1 and a cost of $-|E|$ times the largest cost appearing among all the edge costs already in the graph. Giving it such a negative cost ensures that placing other flow through the network in order to get a unit of flow across it will cause the total cost to decrease. Then, to recover the shortest path for any vertex, start at that vertex and then go to any vertex that is sending a unit of flow to it. Repeat this until you've reached s.

30 Polynomials and the FFT

30.1 Representing polynomials

30.1-1

Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using equations (30.1) and (30.2).

$$\begin{aligned} & 56x^6 - 8x^5 + (8 - 42)x^4 + (-80 + 6 + 21)x^3 + (-3 - 6)x^2 + (60 + 3)x - 30 \\ = \; & 56x^6 - 8x^5 - 34x^4 - 53x^3 - 9x^2 + 63x - 30. \end{aligned}$$

30.1-2

Another way to evaluate a polynomial $A(x)$ of degree-bound n at a given point x_0 is to divide $A(x)$ by the polynomial $(x - x_0)$, obtaining a quotient polynomial $q(x)$ of degree-bound $n - 1$ and a remainder r, such that

$$A(x) = q(x)(x - x_0) + r.$$

Clearly, $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ in time $\Theta(n)$ from x_0 and the coefficients of A.

Let A be the matrix with 1's on the diagonal, $-x_0$'s on the super diagonal, and 0's everywhere else. Let q be the vector $(r, q_0, q_1, \ldots, x_{n-2})$. If $a = (a_0, a_1, \ldots, a_{n-1})$ then we need to solve the matrix equation $Aq = a$ to compute the remainder and coefficients. Since A is tridiagonal, Problem 28-1 (e) tells us how to solve this equation in linear time.

30.1-3

Derive a point-value representation for $A^{\mathrm{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_j x^j$, assuming that none of the points is 0.

For each pair of points, $(p, A(p))$, we can compute the pair $(\frac{1}{p}, A^{\mathrm{rev}}(\frac{1}{p}))$. To do this, we note that

$$\begin{aligned} A^{\mathrm{rev}}(\frac{1}{p}) &= \sum_{j=0}^{n-1} a_{n-1-j} (\frac{1}{p})^j \\ &= \sum_{j=0}^{n-1} a_j (\frac{1}{p})^{n-1-j} \\ &= p^{1-n} \sum_{j=0}^{n-1} a_j p^j \\ &= p^{1-n} A(p), \end{aligned}$$

since we know what $A(p)$ is, we can compute $A^{\mathrm{rev}}(\frac{1}{p})$ of course, we are using the fact that $p \ne 0$ because we are dividing by it. Also, we know that each of these points are distinct, because $\frac{1}{p} = \frac{1}{p'}$ implies that $p = p'$ by cross multiplication. So, since all the x values were distinct in the point value representation of A, they will be distinct in this point value representation of A^{rev} that we have made.

30.1-4

Prove that n distinct point-value pairs are necessary to uniquely specify a polynomial of degree-bound n, that is, if fewer than n distinct point-value pairs are given, they fail to specify a unique polynomial of degree-bound n. (Hint: Using Theorem 30.1, what can you say about a set of $n - 1$ point-value pairs to which you add one more arbitrarily chosen point-value pair?)

Suppose that just $n - 1$ point-value pairs uniquely determine a polynomial P which satisfies them. Append the point value pair (x_{n-1}, y_{n-1}) to them, and let P' be the unique polynomial which agrees with the n pairs, given by Theorem 30.1. Now append instead (x_{n-1}, y'_{n-1}) where $y_{n-1} \ne y'_{n-1}$, and let P'' be the polynomial obtained from these points via Theorem 30.1. Since polynomials coming from n pairs are unique, $P' \ne P''$. However, P' and P'' agree on the original $n - 1$ point-value pairs, contradicting the fact that P was determined uniquely.

30.1-5

Show how to use equation (30.5) to interpolate in time $\Theta(n^2)$. (Hint: First compute the coefficient representation of the polynomial $\prod_j (x - x_j)$ and then divide by $(x - x_k)$ as necessary for the numerator of each term; see Exercise 30.1-2. You can compute each of the n denominators in time $O(n)$.)

First, we show that we can compute the coefficient representation of $\prod_j (x - x_j)$ in time $\Theta(n^2)$. We will do it by recursion, showing that multiplying $\prod_{j<k} (x - x_j)$ by $(x - x_k)$ only takes time $O(n)$, since this only needs to be done n times, this gets is total runtime of $O(n)$. Suppose that $\sum_{i=0}^{k-1} k_i x^i$ is a coefficient representation of $\prod_{j<k} (x - x_j)$. To multiply this by $(x - x_k)$, we just set $(k + 1)_i = k_{i-1} - x_k k_i$ for $i = 1, \ldots, k$ and $(k + 1)_0 = -x_k \cdot k_0$. Each of these coefficients can be computed in constant time, since there are only linearly many coefficients, then, the time to compute the next partial product is just $O(n)$.

Now that we have a coefficient representation of $\prod_j (x - x_j)$, we need to compute, for each k $\prod_{j-k} (x - x_j)$, each of which can be computed in time $\Theta(n)$ by problem 30.1-2. Since the polynomial is defined as a product of things containing the thing we are dividing by, we have that the remainder in each case is equal to 0. Lets call these polynomials f_k. Then, we need only compute the sum $\sum_k y_k \frac{f_i(x)}{f_i(x_k)}$. That is, we compute $f(x_k)$ each in time $\Theta(n)$, so all told, only $\Theta(n^2)$ time is spent computing all the $f(x_k)$ values. For each of the terms in the sum, dividing the polynomial $f_k(x)$ by the number $f_k(x_k)$ and multiplying by y_k only takes time $\Theta(n)$, so total it takes time $\Theta(n^2)$. Lastly, we are adding up n polynomials, each of degree bound $n - 1$, so the total time taken there is $\Theta(n^2)$.

30.1-6

Explain what is wrong with the "obvious" approach to polynomial division using a point-value representation, i.e., dividing the corresponding y values. Discuss separately the case in which the division comes out exactly and the case in which it doesn't.

If we wish to compute P/Q but Q takes on the value zero at some of these points, then we can't carry out the "obvious" method. However, as long as all point value pairs (x_i, y_i) we choose for Q are such that $y_i \ne 0$, then the approach comes out exactly as we would like.

30.1-7

Consider two sets A and B, each having n integers in the range from 0 to $10n$. We wish to compute the ***Cartesian sum*** of A and B, defined by

$$C = \{x + y : x \in A \text{ and } y \in B\}.$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Show how to solve the problem in $O(n \lg n)$ time. (Hint: Represent A and B as polynomials of degree at most $10n$.)

For the set A, we define the polynomial f_A to have a coefficient representation that has a_i equal zero if $i \notin A$ and equal to 1 if $i \in A$. Similarly define f_B. Then, we claim that looking at $f_C := f_A \cdot f_B$ in coefficient form, we have that the ith coefficient, c_i is exactly equal to the number of times that i is realized as a sum of elements from A and B. Since we can perform the polynomial multiplication in time $O(n \lg n)$ by the methods of this chapter, we can get the final answer in time $O(n \lg n)$. To see that f_C has the nice property described, we'll look at the ways that we could end up having a term of x^i appear. Each contribution to that coefficient must come from there being some k so that $a_k \ne 0$ and $b_{i-k} \ne 0$, because the powers of x attached to each are additive when we multiply. Since each of these contributions is only ever 1, the final coefficient is counting the total number of such contributions, therefore counting the number of $k \in A$ such that $-k \in B$, which is exactly what we claimed f_C was counting.

30.2 The DFT and FFT

30.2-1

Prove Corollary 30.4.

(Omit!)

30.2-2

Compute the DFT of the vector $(0, 1, 2, 3)$.

(Omit!)

30.2-3

Do Exercise 30.1-1 by using the $\Theta(n \lg n)$-time scheme.

(Omit!)

30.2-4

Write pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time.

(Omit!)

30.2-5

Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time, and solve the recurrence.

(Omit!)

30.2-6 ⋆

Suppose that instead of performing an n-element FFT over the field of complex numbers (where n is even), we use the ring $\mathbb{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. Use $\omega = 2^t$ instead of ω_n as a principal nth root of unity, modulo m. Prove that the DFT and the inverse DFT are well defined in this system.

(Omit!)

30.2-7

Given a list of values $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree-bound $n + 1$ that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions). Your procedure should run in time $O(n \lg^2 n)$. (Hint: The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

(Omit!)

30.2-8 ⋆

The ***chirp transform*** of a vector $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ and z is any complex number. The DFT is therefore a special case of the chirp

transform, obtained by taking $z = \omega_n$. Show how to evaluate the chirp transform in time $O(n \lg n)$ for any complex number z. (Hint: Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right)\left(z^{-(k-j)^2/2}\right)$$

to view the chirp transform as a convolution.)

(Omit!)

30.3 Efficient FFT implementations

30.3-1

Show how ITERATIVE-FFT computes the DFT of the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

(Omit!)

30.3-2

Show how to implement an FFT algorithm with the bit-reversal permutation occurring at the end, rather than at the beginning, of the computation. (Hint: Consider the inverse DFT.)

(Omit!)

30.3-3

How many times does ITERATIVE-FFT compute twiddle factors in each stage? Rewrite ITERATIVE-FFT to compute twiddle factors only 2^{s-1} times in stage s.

(Omit!)

30.3-4 ⋆

Suppose that the adders within the butterfly operations of the FFT circuit sometimes fail in such a manner that they always produce a zero output, independent of their inputs. Suppose that exactly one adder has failed, but that you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. How efficient is your method?

(Omit!)

Problem 30-1 Divide-and-conquer multiplication

a. Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (Hint: One of the multiplications is $(a + b) \cdot (c + d)$.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n in $\Theta(n^{\lg 3})$ time. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

c. Show how to multiply two n-bit integers in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

(Omit!)

Problem 30-2 Toeplitz matrices

A ***Toeplitz matrix*** is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b. Describe how to represent a Toeplitz matrix so that you can add two $n \times n$ Toeplitz matrices in $O(n)$ time.

c. Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).

d. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

(Omit!)

Problem 30-3 Multidimensional fast Fourier transform

We can generalize the 1-dimensional discrete Fourier transform defined by equation (30.8) to d dimensions. The input is a d-dimensional array $A = (a_{j_1, j_2, \ldots, j_d})$ whose dimensions are $n_1, n_2, \ldots, n_d$, where $n_1 n_2 \cdots n_d = n$. We define the d-dimensional discrete Fourier transform by the equation

$$y_{k_1,k_2,\ldots,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,\ldots,j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

for $0 \le k_1 < n_1, 0 \le k_2 < n_2, \ldots, 0 \le k_d < n_d$.

a. Show that we can compute a d-dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, we first compute n/n_1 separate 1-dimensional DFTs along dimension 1. Then, using the result of the DFTs along dimension 1 as the input, we compute n/n_2 separate 1-dimensional DFTs along dimension 2. Using this result as the input, we compute n/n_3 separate 1-dimensional DFTs along dimension 3, and so on, through dimension d.

b. Show that the ordering of dimensions does not matter, so that we can compute a d-dimensional DFT by computing the 1-dimensional DFTs in any order of the d dimensions.

c. Show that if we compute each 1-dimensional DFT by computing the fast Fourier transform, the total time to compute a d-dimensional DFT is $O(n \lg n)$, independent of d.

(Omit!)

Problem 30-4 Evaluating all derivatives of a polynomial at a point

Given a polynomial $A(x)$ of degree-bound n, we define its tth derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0, \\ \frac{d}{dx} A^{(t-1)}(x) & \text{if } 1 \le t \le n-1, \\ 0 & \text{if } t \ge n. \end{cases}$$

From the coefficient representation $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$ and a given point x_0, we wish to determine $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$.

a. Given coefficients $b_0, b_1, \ldots, b_{n-1}$ such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

show how to compute $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$, in $O(n)$ time.

b. Explain how to find $b_0, b_1, \ldots, b_{n-1}$ in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$.

c. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

where $f(j) = a_j \cdot j!$ and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \le l \le 0, \\ 0 & \text{if } 1 \le l \le n-1. \end{cases}$$

d. Explain how to evaluate $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$ in $O(n \lg n)$ time. Conclude that we can evaluate all nontrivial derivatives of $A(x)$ at x_0 in $O(n \lg n)$ time.

(Omit!)

Problem 30-5 Polynomial evaluation at multiple points

We have seen how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. We have also discovered how to evaluate such a polynomial at all n complex roots of unity in $O(n \lg n)$ time using the FFT. We shall now show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(n \lg^2 n)$ time.

To do so, we shall assume that we can compute the polynomial remainder when one such polynomial is divided by another in $O(n \lg n)$ time, a result that we state without proof. For example, the remainder of $3x^3 + x^2 - 3x + 1$ when divided by $x^2 + x + 2$ is

$$(3x^3 + x^2 - 3x + 1) \mod (x^2 + x + 2) = -7x + 5.$$

Given the coefficient representation of a polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and n points $x_0, x_1, \ldots, x_{n-1}$, we wish to compute the n values $A(x_0), A(x_1), \ldots, A(x_{n-1})$. For $0 \le i \le j \le n-1$, define the polynomials $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ and $Q_{ij}(x) = A(x) \mod P_{ij}(x)$. Note that $Q_{ij}(x)$ has degree at most $j - i$.

a. Prove that $A(x) \mod (x - z) = A(z)$ for any point z.

b. Prove that $Q_{kk}(x) = A(x_k)$ and that $Q_{0,n-1}(x) = A(x)$.

c. Prove that for $i \le k \le j$, we have $Q_{ik}(x) = Q_{ij}(x) \mod P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x) \mod P_{kj}(x)$.

d. Give an $O(n \lg^2 n)$-time algorithm to evaluate $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

(Omit!)

Problem 30-6 FFT using modular arithmetic

As defined, the discrete Fourier transform requires us to compute with complex numbers, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and by using a variant of the FFT based on modular arithmetic, we can guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 gives one approach, using a modulus of length $\Omega(n)$ bits to handle a DFT on n points. This problem gives another approach, which uses a modulus of the more reasonable length $O(\lg n)$; it requires that you understand the material of Chapter 31. Let n be a power of 2.

a. Suppose that we search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why we might expect k to be approximately $\ln n$. (The value of k might be much larger or smaller, but we can reasonably expect to examine $O(\lg n)$ candidate values of k on average.) How does the expected length of p compare to the length of n?

Let g be a generator of $\mathbb{Z}_p^*$, and let $w = g^k \mod p$.

b. Argue that the DFT and the inverse DFT are well-defined inverse operations modulo p, where w is used as a principal nth root of unity.

c. Show how to make the FFT and its inverse work modulo p in time $O(n \lg n)$, where operations on words of $O(\lg n)$ bits take unit time. Assume that the algorithm is given p and w.

d. Compute the DFT modulo $p = 17$ of the vector $(0, 5, 3, 7, 7, 2, 1, 6)$. Note that $g = 3$ is a generator of $\mathbb{Z}_{17}^*$.

(Omit!)

31 Number-Theoretic Algorithms

31.1 Elementary number-theoretic notions

31.1-1

Prove that if $a > b > 0$ and $c = a + b$, then $c \mod a = b$.

$$\begin{aligned} c \mod a &= (a + b) \mod a \\ &= (a \mod a) + (b \mod a) \\ &= 0 + b \\ &= b. \end{aligned}$$

31.1-2

Prove that there are infinitely many primes.

$$\begin{aligned} ((p_1 p_2 \cdots p_k) + 1) \mod p_i &= (p_1 p_2 \cdots p_k) \mod p_i + (1 \mod p_i) \\ &= 0 + 1 \\ &= 1. \end{aligned}$$

if $p_1 p_2 \cdots p_k$ is all prime numbers, then $(p_1 p_2 \cdots p_k) + 1$ is a new prime number.

31.1-3

Prove that if $a \nmid b$ and $b \nmid c$, then $a \nmid c$.

- If $a \mid b$, then $b = a \cdot k_1$.
- If $b \mid c$, then $c = b \cdot k_2 = a \cdot (k_1 \cdot k_2) = a \cdot k_3$, then $a \mid c$.

31.1-4

Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

- If $k \neq 1$, then $k \nmid p$.
- If $k = 1$, then the divisor is 1.

31.1-5

Prove Corollary 31.5.

For all positive integers n, a, and b, if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

Assume for the purpose of contradiction that $n \mid ab$ and $\gcd(a, n) = 1$, but $n \nmid b$, so $\gcd(n, b) = 1$, for theorem 31.6, $\gcd(n, ab) = 1$ which is contradicting our assumption.

31.1-6

> Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a and b and all primes p,
>
> $$(a+b)^p \equiv a^p + b^p \pmod{p}.$$

$$\begin{aligned}(a+b)^p &\equiv a^p + \binom{p}{1}a^{p-1}b^1 + \cdots + \binom{p}{p-1}a^1b^{p-1} + b^p && \pmod{p}\\ &\equiv a^p + 0 + \cdots + 0 + b^p && \pmod{p}\\ &\equiv a^p + b^p && \pmod{p}\end{aligned}$$

31.1-7

> Prove that if a and b are any positive integers such that $a \mid b$, then
>
> $$(x \bmod b) \bmod a = x \bmod a$$
>
> for any x. Prove, under the same assumptions, that
>
> $$x \equiv y \pmod{b} \text{ implies } x \equiv y \pmod{a}$$
>
> for any integers x and y.

Suppose $x = kb + c$, we have

$$(x \bmod b) \bmod a = c \bmod a,$$

and

$$x \bmod a = (kb + c) \bmod a = (kb \bmod a) + (c \bmod a) = c \bmod a.$$

31.1-8

> For any integer $k > 0$, an integer n is a k***th power*** if there exists an integer a such that $a^k = n$. Furthermore, $n > 1$ is a ***nontrivial power*** if it is a kth power for some integer $k > 1$. Show how to determine whether a given β-bit integer n is a nontrivial power in time polynomial in β.

Because $2^\beta > n$, we only need to test values of k that satisfy $2 \le k < \beta$, therefore the testing procedure remains $O(\beta)$.

For any nontrivial power k, where $2 \le k < \beta$, do a binary search on a that costs

$$O(\log \sqrt[k]{n}) = O(\log \sqrt[k]{2^\beta}) = O(\frac{1}{2}\log 2^\beta) = O(\beta).$$

Thus, the total time complexity is

$$O(\beta) \times O(\beta) = O(\beta^2).$$

31.1-9

> Prove equations (31.6)–(31.10).

(Omit!)

31.1-10

> Show that the gcd operator is associative. That is, prove that for all integers a, b, and c,
>
> $$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c).$$

[The following proof is provided by my friend, Tony Xiao.]

Let $d = \gcd(a, b, c)$, $a = dp$, $b = dq$ and $c = dr$.

Claim $\gcd(a, \gcd(b, c)) = d$.

Let $e = \gcd(b, c)$, thus

$$\begin{aligned}b &= es,\\ c &= et.\end{aligned}$$

Since $d \mid b$ and $d \mid c$, thus $d \mid e$.

Let $e = dm$, thus

$$\begin{aligned}b &= (dm)s = dq,\\ c &= (dm)t = dr.\end{aligned}$$

Suppose $k = \gcd(p, m)$,

$$\begin{aligned}&k \mid p, k \mid m,\\ \Rightarrow &dk \mid dp, dk \mid dm,\\ \Rightarrow &dk \mid dp, dk \mid (dm)s, dk \mid (dm)t,\\ \Rightarrow &dk \mid a, dk \mid b, dk \mid c.\end{aligned}$$

Since $d = \gcd(a, b, c)$, thus $k = 1$.

$$\begin{aligned}\gcd(a, \gcd(b, c)) &= \gcd(a, e)\\ &= \gcd(dp, dm)\\ &= d \cdot \gcd(p, m)\\ &= d \cdot k\\ &= d.\end{aligned}$$

By the claim,

$$\gcd(a, \gcd(b, c)) = d = \gcd(\gcd(a, b), c).$$

31.1-11 ⋆

> Prove Theorem 31.8.

(Omit!)

31.1-12

> Give efficient algorithms for the operations of dividing a β-bit integer by a shorter integer and of taking the remainder of a β-bit integer when divided by a shorter integer. Your algorithms should run in time $\Theta(\beta^2)$.

Shift left until the two numbers have the same length, then repeatedly subtract with proper multiplier and shift right.

31.1-13

> Give an efficient algorithm to convert a given β-bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes time $M(\beta)$, then we can convert binary to decimal in time $\Theta(M(\beta)\lg\beta)$.

(Omit!)

31.2 Greatest common divisor

31.2-1

> Prove that equations (31.11) and (31.12) imply equation (31.13).

(Omit!)

31.2-2

> Compute the values (d, x, y) that the call EXTENDED-EUCLID$(899, 493)$ returns.

$(29, -6, 11)$.

31.2-3

> Prove that for all integers a, k, and n,
>
> $$\gcd(a, n) = \gcd(a + kn, n).$$

- $\gcd(a, n) \mid \gcd(a + kn, n)$.

  Let $d = \gcd(a, n)$, then $d \mid a$ and $d \mid n$.

  Since

  $$(a + kn) \bmod d = a \bmod d + k \cdot (n \bmod d) = 0$$

  and $d \mid n$, we have

  $$d \mid \gcd(a + kn, n)$$

  and

  $$\gcd(a, n) \mid \gcd(a + kn, n).$$

- $\gcd(a + kn, n) \mid \gcd(a, n)$.

  Suppose $d = \gcd(a + kn, n)$, we have $d \mid n$ and $d \mid (a + kn)$.

  Since

  $$(a + kn) \bmod d = a \bmod d + k \cdot (n \bmod d) = a \bmod d = 0,$$

  we have $d \mid a$.

  Since $d \mid a$ and $d \mid n$, we have

  $$d \mid \gcd(a, n)$$

  and

  $$\gcd(a + kn, n) \mid \gcd(a, n).$$

  Since

  $$\gcd(a, n) \mid \gcd(a + kn, n)$$

  and

  $$\gcd(a + kn, n) \mid \gcd(a, n),$$

  we have

  $$\gcd(a, n) = \gcd(a + kn, n).$$

31.2-4

> Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

```
EUCLID(a, b)
    while b != 0
        t = a
        a = b
        b = t % b
    return a
```

31.2-5

> If $a > b \ge 0$, show that the call EUCLID(a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b/\gcd(a, b))$.

$$b \ge F_{k+1} \approx \phi^{k+1}/\sqrt{5}$$

$$k + 1 < \log_\phi \sqrt{5} + \log_\phi b \approx 1.67 + \log_\phi b$$

$$k < 0.67 + \log_\phi b < 1 + \log_\phi b.$$

Since $d \cdot a \bmod d \cdot b = d \cdot (a \bmod b)$, $\gcd(d \cdot a, d \cdot b)$ has the same number of recursive calls with $\gcd(a, b)$, therefore we could let $b' = b/\gcd(a, b)$, the inequality $k < 1 + \log_\phi(b') = 1 + \log_\phi(b/\gcd(a, b))$. will holds.

31.2-6

> What does EXTENDED-EUCLID(F_{k+1}, F_k) return? Prove your answer correct.

- If k is odd, then $(1, -F_{k-2}, F_{k-1})$.
- If k is even, then $(1, F_{k-2}, -F_{k-1})$.

31.2-7

> Define the gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, \cdots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \cdots, a_n))$. Show that the gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers $x_0, x_1, \cdots, x_n$ such that $\gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, \cdots, a_n\}))$.

Suppose

$$\gcd(a_0, \gcd(a_1, a_2, \cdots, a_n)) = a_0 \cdot x + \gcd(a_1, a_2, \cdots, a_n) \cdot y$$

and

$$\gcd(a_1, \gcd(a_2, a_3, \cdots, a_n)) = a_1 \cdot x' + \gcd(a_2, a_3, \cdots, a_n) \cdot y',$$

then the coefficient of a_1 is $y \cdot x'$.

```
EXTENDED-EUCLID(a, b)
    if b == 0
        return (a, 1, 0)
    (d, x, y) = EXTENDED-EUCLID(b, a % b)
    return (d, y, x - (a / b) * y)
```

```
EXTENDED-EUCLID-MULTIPLE(a)
    if a.length == 1
        return (a[0], 1)
    g = a[a.length - 1]
    xs = [1] * a.length
    ys = [0] * a.length
    for i = a.length - 2 downto 0
        (g, xs[i], ys[i + 1]) = EXTENDED-EUCLID(a[i], g)
    m = 1
    for i = 1 to a.length
        m *= ys[i]
        xs[i] *= m
    return (g, xs)
```

31.2-8

> Define $\text{lcm}(a_1, a_2, \ldots, a_n)$ to be the ***least common multiple*** of the n integers $a_1, a_2, \ldots, a_n$, that is, the smallest nonnegative integer that is a multiple of each a_i. Show how to compute $\text{lcm}(a_1, a_2, \ldots, a_n)$ efficiently using the (two-argument) gcd operation as a subroutine.

```
GCD(a, b)
    if b == 0
        return a
    return GCD(b, a % b)
```

```
LCM(a, b)
    return a / GCD(a, b) * b
```

```
LCM-MULTIPLE(a)
    l = a[0]
    for i = 1 to a.length
        l = LCM(l, a[i])
    return l
```

31.2-9

> Prove that n_1, n_2, n_3, and n_4 are pairwise relatively prime if and only if
>
> $$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1.$$

> More generally, show that $n_1, n_2, \ldots, n_k$ are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

Suppose $n_1 n_2 x + n_3 n_4 y = 1$, then $n_1(n_2 x) + n_3(n_4 y) = 1$, thus n_1 and n_3 are relatively prime, n_1 and n_4, n_2 and n_3, n_2 and n_4 are the all relatively prime. And since $\gcd(n_1 n_3, n_2 n_4) = 1$, all the pairs are relatively prime.

General: in the first round, divide the elements into two sets with equal number of elements, calculate the products of the two set separately, if the two products are relatively prime, then the element in one set is pairwise relatively prime with the element in the other set. In the next iterations, for each set, we divide the elements into two subsets, suppose we have subsets $\{(s_1, s_2), (s_3, s_4), \ldots\}$, then we calculate the products of $\{s_1, s_3, \ldots\}$ and $\{s_2, s_4, \ldots\}$, if the two products are relatively prime, then all the pairs of subset are pairwise relatively prime similar to the first round. In each iteration, the number of elements in a subset is half of the original set, thus there are $\lceil \lg k \rceil$ pairs of products.

To choose the subsets efficiently, in the kth iteration, we could divide the numbers based on the value of the index's kth bit.

31.3 Modular arithmetic

31.3-1

> Draw the group operation tables for the groups $(\mathbb{Z}_4, +_4)$ and $(\mathbb{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence α between their elements such that $a + b \equiv c \pmod 4$ if and only if $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod 5$.

- $(\mathbb{Z}_4, +_4)$: $\{0, 1, 2, 3\}$.
- $(\mathbb{Z}_5^*, \cdot_5)$: $\{1, 2, 3, 4\}$.

$\alpha(x) = 2^{x-1}$.

31.3-2

> List all subgroups of $\mathbb{Z}_9$ and of $\mathbb{Z}_{13}^*$.

- $\mathbb{Z}_9$:
  - $\langle 0 \rangle = \{0\}$,
  - $\langle 1 \rangle = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$,
  - $\langle 2 \rangle = \{0, 2, 4, 6, 8\}$.
- $\mathbb{Z}_{13}^*$:
  - $\langle 1 \rangle = \{1\}$,
  - $\langle 2 \rangle = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.

31.3-3

> Prove Theorem 31.14.

A nonempty closed subset of a finite group is a subgroup.

- Closure: the subset is closed.
- Identity: suppose $a \in S'$, then $a^{(k)} \in S'$. Since the subset is finite, there must be a period such that $a^{(m)} = a^{(n)}$, hence $a^{(m)} a^{(-n)} = a^{(m-n)} = 1$, therefore the subset must contain the identity.
- Associativity: inherit from the origin group.
- Inverses: suppose $a^{(k)} = 1$, the inverse of element a is $a^{(k-1)}$ since $a a^{(k-1)} = a^{(k)} = 1$.

31.3-4

> Show that if p is prime and e is a positive integer, then
>
> $\phi(p^e) = p^{e-1}(p-1)$.

$\phi(p^e) = p^e \cdot \left(1 - \frac{1}{p}\right) = p^{e-1}(p-1)$.

31.3-5

> Show that for any integer $n > 1$ and for any $a \in \mathbb{Z}_n^*$, the function $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ defined by $f_a(x) = ax \mod n$ is a permutation of $\mathbb{Z}_n^*$.

To prove it is a permutation, we need to prove that

- for each element $x \in \mathbb{Z}_n^*$, $f_a(x) \in \mathbb{Z}_n^*$,
- the numbers generated by f_a are distinct.

Since $a \in \mathbb{Z}_n^*$ and $x \in \mathbb{Z}_n^*$, then $f_a(x) = ax \mod n \in \mathbb{Z}_n^*$ by the closure property.

Suppose there are two distinct numbers $x \in \mathbb{Z}_n^*$ and $y \in \mathbb{Z}_n^*$ that $f_a(x) = f_a(y)$,

$$\begin{aligned} f_a(x) &= f_a(y) \\ ax &\mod n = ay \mod n \\ (a \mod n)(x &\mod n) = (a \mod n)(y \mod n) \\ x &\mod n = y \mod n \\ x &\equiv y \mod n, \end{aligned}$$

which contradicts the assumption, therefore the numbers generated by f_a are distinct.

31.4 Solving modular linear equations

31.4-1

> Find all solutions to the equation $35x \equiv 10 \pmod{50}$.

$\{6, 16, 26, 36, 46\}$.

31.4-2

> Prove that the equation $ax \equiv ay \pmod n$ implies $x \equiv y \pmod n$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

$$d = \gcd(ax, n) = \gcd(x, n)$$

Since $ax \cdot x' + n \cdot y' = d$,

we have

$$x \cdot (ax') + n \cdot y' = d.$$

$$\begin{aligned} x_0 &= x'(ay/d), \\ x_0' &= (ax')(y/d) = x'(ay/d) = x_0. \end{aligned}$$

31.4-3

> Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:
>
> ```
> 3  x0 = x'(b / d) mod (n / d)
> ```
>
> Will this work? Explain why or why not.

Assume that $x_0 \geq n/d$, then the largest solution is $x_0 + (d-1) \cdot (n/d) \geq d \cdot n/d \geq n$, which is impossible, therefore $x_0 < n/d$.

31.4-4 ⋆

> Let p be prime and $f(x) \equiv f_0 + f_1 x + \cdots + f_t x^t \pmod p$ be a polynomial of degree t, with coefficients f_i drawn from $\mathbb{Z}_p$. We say that $a \in \mathbb{Z}_p$ is a ***zero*** of f if $f(a) \equiv 0 \pmod p$. Prove that if a is a zero of f, then $f(x) \equiv (x - a)g(x) \pmod p$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that if p is prime, then a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo p.

(Omit!)

31.5 The Chinese remainder theorem

31.5-1

> Find all solutions to the equations $x \equiv 4 \pmod 5$ and $x \equiv 5 \pmod{11}$.

$$\begin{aligned} m_1 &= 11, m_2 = 5. \\ m_1^{-1} &= 1, m_2^{-1} = 9. \\ c_1 &= 11, c_2 = 45. \\ a &= (c_1 \cdot a_1 + c_2 \cdot a_2) \mod (n_1 \cdot n_2) \\ &= (11 \cdot 4 + 45 \cdot 5) \mod 55 = 49. \end{aligned}$$

31.5-2

> Find all integers x that leave remainders $1, 2, 3$ when divided by $9, 8, 7$ respectively.

$10 + 504i, i \in \mathbb{Z}$.

31.5-3

> Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then
>
> $$(a^{-1} \mod n) \leftrightarrow ((a_1^{-1} \mod n_1), (a_2^{-1} \mod n_2), \ldots, (a_k^{-1} \mod n_k)).$$
>
> $$\gcd(a, n) = 1 \to \gcd(a, n_i) = 1.$$

31.5-4

> Under the definitions of Theorem 31.27, prove that for any polynomial f, the number of roots of the equation $f(x) \equiv 0 \pmod n$ equals the product of the number of roots of each of the equations
>
> $$f(x) \equiv 0 \pmod{n_1}, f(x) \equiv 0 \pmod{n_2}, \ldots, f(x) \equiv 0 \pmod{n_k}.$$

Based on 31.28–31.30.

31.6 Powers of an element

31.6-1

> Draw a table showing the order of every element in $\mathbb{Z}_{11}^*$. Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbb{Z}_{11}^*$.

$g = 2$, $\{1, 2, 4, 8, 5, 10, 9, 7, 3, 6\}$.

31.6-2

> Give a modular exponentiation algorithm that examines the bits of b from right to left instead of left to right.

```
MODULAR-EXPONENTIATION(a, b, n)
    i = 0
    d = 1
    while (1 << i) ≤ b
        if (b & (1 << i)) > 0
            d = (d * a) % n
        a = (a * a) % n
        i = i + 1
    return d
```

31.6-3

> Assuming that you know $\phi(n)$, explain how to compute $a^{-1} \mod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

$$\begin{aligned} a^{\phi(n)} &\equiv 1 \pmod n, \\ a \cdot a^{\phi(n)-1} &\equiv 1 \pmod n, \\ a^{-1} &\equiv a^{\phi(n)-1} \pmod n. \end{aligned}$$

31.7 The RSA public-key cryptosystem

31.7-1

> Consider an RSA key set with $p = 11, q = 29, n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

$\phi(n) = (p-1) \cdot (q-1) = 280$.

$d = e^{-1} \mod \phi(n) = 187$.

$P(M) = M^e \mod n = 254$.

$S(C) = C^d \mod n = 254^{187} \mod n = 100$.

31.7-2

> Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d, where $0 < d < \phi(n)$, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n. (Although you are not asked to prove it, you may be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [255].)

$$ed \equiv 1 \mod \phi(n)$$

$$ed - 1 = 3d - 1 = k\phi(n)$$

If $p, q < n/4$, then

$$\phi(n) = n - (p+q) + 1 > n - n/2 + 1 = n/2 + 1 > n/2.$$

$kn/2 < 3d - 1 < 3d < 3n$, then $k < 6$, then we can solve $3d - 1 = n - p - n/p + 1$.

31.7-3 ⋆

> Prove that RSA is multiplicative in the sense that
>
> $P_A(M_1)P_A(M_2) \equiv P_A(M_1, M_2) \pmod n$.
>
> Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1 percent of messages from $\mathbb{Z}_n$ encrypted with P_A, then he could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

Multiplicative: e is relatively prime to n.

Decrypt: In each iteration randomly choose a prime number m that m is relatively prime to n, if we can decrypt $m \cdot M$, then we can return $m^{-1} M$ since $m^{-1} = m^{n-2}$.

31.8 Primality testing

31.8-1

> Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1 modulo n.

(Omit!)

31.8-2 ⋆

> It is possible to strengthen Euler's theorem slightly to the form
>
> $a^{\lambda(n)} \equiv 1 \pmod n$ for all $a \in \mathbb{Z}_n^*$,
>
> where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by
>
> $$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})). \tag{31.42}$$
>
> Prove that $\lambda(n) \mid \phi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n - 1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$; here, $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. (For this reason, they are not very common.)

1. Prove that $\lambda(n) \mid \phi(n)$.

   We have

   $$\begin{aligned} n &= p_1^{e_1} \cdots p_r^{e_r} \\ \phi(n) &= \phi(p_1^{e_1}) \times \cdots \times \phi(p_r^{e_r}). \end{aligned}$$

   Thus,

$$\lambda(n) \mid \phi(n)$$
$$\Rightarrow \operatorname{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) \mid (\phi(p_1^{e_1}) \times \cdots \times \phi(p_r^{e_r}))$$

2. Prove that Carmichael numbers must be "square-free" (not divisible by the square of any prime).

   Assume $n = p^\alpha m$ is a Carmichael number, where $\alpha \geq 2$ and $p \nmid m$.

   By the Chinese Remainder Theorem, the system of congruences

   $$\begin{aligned} x &\equiv 1 + p \pmod{p^\alpha}, \\ x &\equiv 1 \pmod{m} \end{aligned}$$

   has a solution a. Note that $\gcd(a, n) = 1$.

   Since n is a Carmichael number, we have $a^{n-1} \equiv 1 \pmod{n}$. In particular, $a^{n-1} \equiv 1 \pmod{p^\alpha}$; therefore, $a^n \equiv a \pmod{p^2}$.

   So, $(1+p)^n \equiv 1 + p \pmod{p^2}$. Expand $(1+p)^n$ modulo p^2 using the binomial theorem. We have

   $$(1+p)^n \equiv 1 \pmod{p^2},$$

   since the first two terms of the expansion are 1 and np, and the rest of the terms are divisible by p^2.

   Thus, $1 \equiv 1 + p \pmod{p^2}$. This is impossible.

   Stack Exchange Reference

3. Prove that Carmichael numbers must be the product of at least three primes.

   Assume that $n = pq$ is a Carmichael number, where p and q are distant primes and $p < q$.

   Then we have

   $$\begin{aligned} &q \equiv 1 \pmod{q-1} \\ \Rightarrow &n \equiv pq \equiv p \pmod{q-1} \\ \Rightarrow &n - 1 \equiv p - 1 \pmod{q-1}, \end{aligned}$$

   Here, $0 < p - 1 < q - 1$, so $n - 1$ is not divisible by $q - 1$.

   Stack Exchange Reference

31.8-3

> Prove that if x is a nontrivial square root of 1, modulo n, then $\gcd(x-1, n)$ and $\gcd(x+1, n)$ are both nontrivial divisors of n.

$$\begin{aligned} x^2 &\equiv 1 \pmod{n}, \\ x^2 - 1 &\equiv 0 \pmod{n}, \\ (x+1)(x-1) &\equiv 0 \pmod{n}. \end{aligned}$$

$n \mid (x+1)(x-1)$, suppose $\gcd(x-1, n) = 1$, then $n \mid (x+1)$, then $x \equiv -1 \pmod{n}$ which is trivial, it contradicts the fact that x is nontrivial, therefore $\gcd(x-1, n) \neq 1$, $\gcd(x+1, n) \neq 1$.

31.9 Integer factorization

31.9-1

> Referring to the execution history shown in Figure 31.7(a), when does \text{POLLARDRHO} print the factor 73 of 1387?

$x = 84, y = 814$.

31.9-2

> Suppose that we are given a function $f : \mathbb{Z}_n \to \mathbb{Z}_n$ and an initial value $x_0 \in \mathbb{Z}_n$. Define $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots$. Let t and $u > 0$ be the smallest values such that $x_{t+i} = x_{t+u+i}$ for $i = 0, 1, \ldots$. In the terminology of Pollard's rho algorithm, t is the length of the tail and u is the length of the cycle of the rho. Give an efficient algorithm to determine t and u exactly, and analyze its running time.

(Omit!)

31.9-3

> How many steps would you expect POLLARD-RHO to require to discover a factor of the form p^e, where p is prime and $e > 1$?

$\Theta(\sqrt{p})$.

31.9-4 ⋆

> One disadvantage of POLLARD-RHO as written is that it requires one gcd computation for each step of the recurrence. Instead, we could batch the gcd computations by accumulating the product of several x_i values in a row and then using this product instead of x_i in the gcd computation. Describe carefully how you would implement this idea, why it works, and what batch size you would pick as the most effective when working on a β-bit number n.

(Omit!)

Problem 31-1 Binary gcd algorithm

> Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the ***binary gcd algorithm***, which avoids the remainder computations used in Euclid's algorithm.
>
> **a.** Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.
>
> **b.** Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.
>
> **c.** Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a-b)/2, b)$.
>
> **d.** Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time.

(Omit!)

d.

```
BINARY-GCD(a, b)
    if a < b
        return BINARY-GCD(b, a)
    if b == 0
        return a
    if (a & 1 == 1) and (b & 1 == 1)
        return BINARY-GCD((a - b) >> 1, b)
    if (a & 1 == 0) and (b & 1 == 0)
        return BINARY-GCD(a >> 1, b >> 1) << 1
    if a & 1 == 1
        return BINARY-GCD(a, b >> 1)
    return BINARY-GCD(a >> 1, b)
```

Problem 31-2 Analysis of bit operations in Euclid's algorithm

> **a.** Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b, which yields a quotient q and remainder r. Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.
>
> **b.** Define $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \mod b)$ is at most $c(\mu(a, b) - \mu(b, a \mod b))$ for some sufficiently large constant $c > 0$.
>
> **c.** Show that $\text{EUCLID}(a, b)$ requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β-bit inputs.

a.

- Number of comparisons and subtractions: $\lceil \lg a \rceil - \lceil \lg b \rceil + 1 = \lceil \lg q \rceil$.
- Length of subtraction: $\lceil \lg b \rceil$.
- Total: $O((1 + \lg q) \lg b)$.

b.

$$\begin{aligned} & \mu(a, b) - \mu(b, a \mod b) \\ = \; & \mu(a, b) - \mu(b, r) \\ = \; & (1 + \lg a)(1 + \lg b) - (1 + \lg b)(1 + \lg r) \\ = \; & (1 + \lg b)(\lg a - \lg r) && (\lg r \leq \lg b) \\ \geq \; & (1 + \lg b)(\lg a - \lg b) \\ = \; & (1 + \lg b)(\lg q + 1) \\ \geq \; & (1 + \lg q) \lg b \end{aligned}$$

c. $\mu(a, b) = (1 + \lg a)(1 + \lg b) \approx \beta^2$

Problem 31-3 Three algorithms for Fibonacci numbers

> This problem compares the efficiency of three methods for computing the nth Fibonacci number F_n, given n. Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.
>
> **a.** Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.22) is exponential in n. (See, for example, the FIB procedure on page 775.)
>
> **b.** Show how to compute F_n in $O(n)$ time using memoization.
>
> **c.** Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (Hint: Consider the matrix
>
> $$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$
>
> and its powers.)
>
> **d.** Assume now that adding two β-bit numbers takes $\Theta(\beta)$ time and that multiplying two β-bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

a. In order to solve $\text{FIB}(n)$, we need to compute $\text{FIB}(n-1)$ and $\text{FIB}(n-1)$. Therefore we have the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1).$$

We can get the upper bound of Fibonacci as $O(2^n)$, but this is not the tight upper bound.

The Fibonacci recurrence is defined as

$$F(n) = F(n-1) + F(n-2).$$

The characteristic equation for this function will be

$$\begin{aligned} x^2 &= x + 1 \\ x^2 - x - 1 &= 0. \end{aligned}$$

We can get the roots by quadratic formula: $x = \frac{1 \pm \sqrt{5}}{2}$.

We know the solution of a linear recursive function is given as

$$\begin{aligned} F(n) &= \alpha_1^n + \alpha_2^n \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n, \end{aligned}$$

where α_1 and α_2 are the roots of the characteristic equation.

Since both $T(n)$ and $F(n)$ are representing the same thing, they are asymptotically the same.

Hence,

$$\begin{aligned} T(n) &= \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^n \\ &\approx O(1.618)^n. \end{aligned}$$

b. This is same as 15.1-5.

c. Assume that all integer multiplications and additions can be done in $O(1)$. First, we want to show that

$$ \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^k

$$\begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix}$$

. $$

By induction,

$$\begin{aligned} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{k+1} &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^k \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix}^k \\ &= \begin{pmatrix} F_k & F_{k+1} \\ F_{k-1} + F_k & F_k + F_{k+1} \end{pmatrix} \\ &= \begin{pmatrix} F_k & F_{k+1} \\ F_{k+1} & F_{k+2} \end{pmatrix}. \end{aligned}$$

We show that we can compute the given matrix to the power $n-1$ in time $O(\lg n)$, and the bottom right entry is F_n.

We should note that by 8 multiplications and 4 additions, we can multiply any two 2 by 2 matrices, that means matrix multiplications can be done in constant time. Thus we only need to bound the number of those in our algorithm.

It takes $O(\lg n)$ to run the algorithm $\text{MATRIX-POW}(A, n-1)$ becasue we half the value of n in each step, and within each step, we perform a constant amount of calculation.

The recurrence is

$$T(n) = T(n/2) + \Theta(1).$$

```
MATRIX-POW(A, n)
    if n % 2 == 1
        return A * MATRIX-POW(A^2, (n - 1) / 2)
    return MATRIX-POW(A^2, n / 2)
```

d. First, we should note that $\beta = O(\log n)$.

- For part (a),

  We naively add a β-bit number which is growing exponentially each time, so the recurrence becomes

  $$\begin{aligned} T(n) &= T(n-1) + T(n-2) + \Theta(\beta) \\ &= T(n-1) + T(n-2) + \Theta(\log n), \end{aligned}$$

  which has the same solution $T(n) = O\left(\frac{1+\sqrt{5}}{2}\right)^n$ since $\Theta(\log n)$ can be absorbed by exponential time.

- For part (b),

  The recurrence of the memoized verstion becomes

  $$M(n) = M(n-1) + \Theta(\beta).$$

  This has a solution of $\sum_{i=2}^{n} \beta = \Theta(n\beta) = \Theta(2^\beta \cdot \beta)$ or $\Theta(n \log n)$.

- For part (c),

  We perform a constant number of both additions and multiplications. The recurrence becomes

  $$P(n) = P(n/2) + \Theta(\beta^2),$$

  which has a solution of $\Theta(\log n \cdot \beta^2) = \Theta(\beta^3)$ or $\Theta(\log^3 n)$.

Problem 31-4 Quadratic residues

Let p be an odd prime. A number $a \in Z_p^*$ is a ***quadratic residue*** if the equation $x^2 = a \pmod p$ has a solution for the unknown x.

a. Show that there are exactly $(p-1)/2$ quadratic residues, modulo p.

b. If p is prime, we define the ***Legendre symbol*** $\left(\frac{a}{p}\right)$, for $a \in \mathbb{Z}_p^*$, to be 1 if a is a quadratic residue modulo p and -1 otherwise. Prove that if $a \in \mathbb{Z}_p^*$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p.$$

Give an efficient algorithm that determines whether a given number a is a quadratic residue modulo p. Analyze the efficiency of your algorithm.

c. Prove that if p is a prime of the form $4k+3$ and a is a quadratic residue in $\mathbb{Z}_p^*$, then $a^{k+1} \mod p$ is a square root of a, modulo p. How much time is required to find the square root of a quadratic residue a modulo p?

d. Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p, that is, a member of $\mathbb{Z}_p^*$ that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

(Omit!)

32 String Matching

32.1 The naive string-matching algorithm

32.1-1

Show the comparisons the naive string matcher makes for the pattern $P = 0001$ in the text $T = 000010001010001$.

```
STRING-MATCHER(P, T, i)
    for j = i to i + P.length
        if P[j - i + 1] != T[j]
            return false
    return true
```

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T.

Suppose $T[i] \neq P[j]$, then for $k \in [1, j)$, $T[i-k] = P[j-k] \neq P[0]$, the $[i-k, i)$ are all invalid shifts which could be skipped, therefore we can compare $T[i]$ with $P[0]$ in the next iteration.

32.1-3

Suppose that pattern P and text T are randomly chosen strings of length m and n, respectively, from the d-ary alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where $d \geq 2$. Show that the expected number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

Suppose for each shift, the number of compared characters is L, then:

$$\begin{aligned} E[L] &= 1 \cdot \frac{d-1}{d} + 2 \cdot (\frac{1}{d})^1 \frac{d-1}{d} + \cdots + m \cdot (\frac{1}{d})^{m-1} \frac{d-1}{d} + m \cdot (\frac{1}{d})^m \\ &= (1 + 2 \cdot (\frac{1}{d})^1 + \cdots + m \cdot (\frac{1}{d})^m) \frac{d-1}{d} + m \cdot (\frac{1}{d})^m. \end{aligned}$$

$$\begin{aligned} S &= 1 + 2 \cdot (\frac{1}{d})^1 + \cdots + m \cdot (\frac{1}{d})^{m-1} \\ \frac{1}{d} S &= 1 \cdot (\frac{1}{d})^1 + \cdots + (m-1) \cdot (\frac{1}{d})^{m-1} + m \cdot (\frac{1}{d})^m \\ \frac{d-1}{d} S &= 1 + (\frac{1}{d})^1 + \cdots + (\frac{1}{d})^{m-1} - m \cdot (\frac{1}{d})^m \\ \frac{d-1}{d} S &= \frac{1-d^{-m}}{1-d^{-1}} - m \cdot (\frac{1}{d})^m. \end{aligned}$$

$$\begin{aligned} E[L] &= (1 + 2 \cdot (\frac{1}{d})^1 + \cdots + m \cdot (\frac{1}{d})^m) \frac{d-1}{d} + m \cdot (\frac{1}{d})^m \\ &= \frac{1-d^{-m}}{1-d^{-1}} - m \cdot (\frac{1}{d})^m + m \cdot (\frac{1}{d})^m \\ &= \frac{1-d^{-m}}{1-d^{-1}}. \end{aligned}$$

There are $n-m+1$ shifts, therefore the expected number of comparisons is:

$$(n-m+1) \cdot E[L] = (n-m+1)\frac{1-d^{-m}}{1-d^{-1}}$$

Since $d \geq 2$, $1 - d^{-1} \geq 0.5$, $1 - d^{-m} < 1$, and $\frac{1-d^{-m}}{1-d^{-1}} \leq 2$, therefore

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1).$$

32.1-4

Suppose we allow the pattern P to contain occurrences of a ***gap character*** $\diamond$ that can match an *arbitrary* string of characters (even one of zero length). For example, the pattern $ab \diamond ba \diamond c$ occurs in the text $cabccbacbacab$ as

$$\underbrace{cab}_{ab \diamond}\underbrace{ccbacba}_{ba \diamond}\underbrace{c}_{c}ab$$

and as

$$\underbrace{cabccbacba}_{ab \diamond ba \diamond}\underbrace{c}_{c}ab$$

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

By using dynamic programming, the time complexity is $O(mn)$ where m is the length of the text T and n is the length of the pattern P; the space complexity is $O(mn)$, too.

This problem is similar to LeetCode 44. WildCard Matching, except that it has no question mark (`?`) requirement. You can see my naive DP implementation here.

32.2 The Rabin-Karp algorithm

32.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

$|\{15, 59, 92\}| = 3$.

32.2-2

How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

Truncation.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

Calculate the hashes in each column just like the Rabin-Karp in one-dimension, then treat the hashes in each row as the characters and hashing again.

32.2-4

Alice has a copy of a long n-bit file $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, and Bob similarly has an n-bit file $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, \ldots, q-1\}$. Then, Alice evaluates

$$A(x) = (\sum_{i=0}^{n-1} a_i x^i) \mod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (Hint: See Exercise 31.4-4.)

(Omit!)

32.3 String matching with finite automata

32.3-1

Construct the string-matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaabaababaab$.

$$0 \to 1 \to 2 \to 2 \to 3 \to 4 \to 5 \to 1 \to 2 \to 3 \to 4 \to 2 \to 3 \to 4 \to 5 \to 1 \to 2 \to 3.$$

32.3-2

Draw a state-transition diagram for a string-matching automaton for the pattern $ababbabbababbababbabb$ over the alphabet $\sigma = \{a, b\}$.

0	1	0
1	1	2
2	3	0
3	1	4
4	3	5
5	6	0
6	1	7
7	3	8
8	9	0
9	1	10
10	11	0
11	1	12
12	3	13
13	14	0
14	1	15
15	16	8
16	1	17
17	3	18
18	19	0
19	1	20
20	3	21
21	9	0

32.3-3

We call a pattern P ***nonoverlappable*** if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

$\delta(q, a) \in \{0, 1, q+1\}$.

32.3-4 ⋆

Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of either pattern. Try to minimize the number of states in your automaton.

Combine the common prefix and suffix.

32.3-5

Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

Split the string with the gap characters, build finite automatons for each substring. When a substring is matched, moved to the next finite automaton.

32.4 The Knuth-Morris-Pratt algorithm

32.4-1

Compute the prefix function π for the pattern $ababbabbabbababbabb$.

$$\pi = \{0, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 4, 5, 6, 7, 8\}.$$

32.4-2

Give an upper bound on the size of $\pi^*[q]$ as a function of q. Give an example to show that your bound is tight.

$|\pi^*[q]| < q$.

32.4-3

Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT (the string of length $m+n$ that is the concatenation of P and T).

$\{q \mid \pi[q] = m \text{ and } q \geq 2m\}$.

32.4-4

Use an aggregate analysis to show that the running time of KMP-MATCHER is Θ.

The number of $q = q + 1$ is at most n.

32.4-5

Use a potential function to show that the running time of KMP-MATCHER is $\Theta(n)$.

$\Phi = p$.

32.4-6

Show how to improve KMP-MATCHER by replacing the occurrence of π in line 7 (but not line 12) by π', where π' is defined recursively for $q = 1, 2, \ldots, m-1$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] = P[q+1] \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this change constitutes an improvement.

If $P[q+1] \neq T[i]$, then if $P[\pi[q]+q] = P[q+1] \neq T[i]$, there is no need to compare $P[\pi[q]+q]$ with $T[i]$.

32.4-7

Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, arc and car are cyclic rotations of each other.

Find T' in TT.

32.4-8 ⋆

Give an $O(m|\Sigma|)$-time algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P. (Hint: Prove that $\delta(q, a) = \delta(\pi[q], a)$ if $q = m$ or $P[q+1] \neq a$.)

Compute the prefix function m times.

Problem 32-1 String matching based on repetition factors

Let y^i denote the concatenation of string y with itself i times. For example, $(ab)^3 = ababab$. We say that a string $x \in \Sigma^*$ has ***repetition factor*** r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $\rho(x)$ denote the largest r such that x has repetition factor r.

a. Give an efficient algorithm that takes as input a pattern $P[1 \ldots m]$ and computes the value $\rho(P_i)$ for $i = 1, 2, \ldots, m$. What is the running time of your algorithm?

b. For any pattern $P[1 \ldots m]$, let $\rho^*(P)$ be defined as $\max_{1 \le i \le m} \rho(P_i)$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ is $O(1)$.

c. Argue that the following string-matching algorithm correctly finds all occurrences of pattern P in a text $T[1 \ldots n]$ in time $O(\rho^*(P)n + m)$:

```
REPETITION_MATCHER(P, T)
    m = P.length
    n = T.length
    k = 1 + ρ*(P)
    q = 0
    s = 0
    while s ≤ n − m
        if T[s + q + 1] == P[q + 1]
            q = q + 1
            if q == m
                print "Pattern occurs with shift" s
        if q == m or T[s + q + 1] != P[q + 1]
            s = s + max(1, ceil(q / k))
            q = 0
```

This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtained a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T.

a. Compute π, let $l = m - \pi[m]$, if $m \bmod l = 0$ and for all $p = m - i \cdot l > 0$, $p - \pi[p] = l$, then $\rho(P_i) = m/l$, otherwise $\rho(P_i) = 1$. The running time is $\Theta(n)$.

b.

$$\begin{aligned} P(\rho^*(P) \ge 2) &= \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \cdots \approx \frac{2}{3} \\ P(\rho^*(P) \ge 3) &= \frac{1}{4} + \frac{1}{32} + \frac{1}{256} + \cdots \approx \frac{2}{7} \\ P(\rho^*(P) \ge 4) &= \frac{1}{8} + \frac{1}{128} + \frac{1}{2048} + \cdots \approx \frac{2}{15} \\ P(\rho^*(P) = 1) &= \frac{1}{3} \\ P(\rho^*(P) = 2) &= \frac{8}{21} \\ P(\rho^*(P) = 3) &= \frac{16}{105} \\ E[\rho^*(P)] &= 1 \cdot \frac{1}{3} + 2 \cdot \frac{8}{21} + 3 \cdot \frac{16}{105} + \ldots \approx 2.21 \end{aligned}$$

c.

(Omit!)

33 Computational Geometry

33.1 Line-segment properties

33.1-1

Prove that if $p_1 \times p_2$ is positive, then vector p_1 is clockwise from vector p_2 with respect to the origin $(0, 0)$ and that if this cross product is negative, then p_1 is counterclockwise from p_2.

(Omit!)

33.1-2

Professor van Pelt proposes that only the x-dimension needs to be tested in line 1 of ON-SEGMENT. Show why the professor is wrong.

$(0, 0), (5, 5), (10, 0)$.

33.1-3

The ***polar angle*** of a point p_1 with respect to an origin point p_0 is the angle of the vector $p_1 - p_0$ in the usual polar coordinate system. For example, the polar angle of $(3, 5)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 45 degrees or $\pi/4$ radians. The polar angle of $(3, 3)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 315 degrees or $7\pi/4$ radians. Write pseudocode to sort a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ of n points according to their polar angles with respect to a given origin point p_0. Your procedure should take $O(n \lg n)$ time and use cross products to compare angles.

(Omit!)

33.1-4

Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are colinear.

Based on exercise 33.1-3, for each point p_i, let p_i be p_0 and sort other points according to their polar angles mod π. Then scan linearly to see whether two points have the same polar angle. $O(n \cdot n \lg n) = O(n^2 \lg n)$.

33.1-5

A ***polygon*** is a piecewise-linear, closed curve in the plane. That is, it is a curve ending on itself that is formed by a sequence of straight-line segments, called the ***sides*** of the polygon. A point joining two consecutive sides is a ***vertex*** of the polygon. If the polygon is ***simple***, as we shall generally assume, it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the ***interior*** of the polygon, the set of points on the polygon itself forms its ***boundary***, and the set of points surrounding the polygon forms its ***exterior***. A simple polygon is convex if, given any two points on its

boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. A vertex of a convex polygon cannot be expressed as a convex combination of any two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a sequence $\langle p_1, p_2, \ldots, p_{n-1} \rangle$ of n points forms the consecutive vertices of a convex polygon. Output "yes" if the set $\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1$, where subscript addition is performed modulo n, does not contain both left turns and right turns; otherwise, output "no." Show that although this method runs in linear time, it does not always produce the correct answer. Modify the professor's method so that it always produces the correct answer in linear time.

A line.

33.1-6

Given a point $p_0 = (x_0, y_0)$, the ***right horizontal ray*** from p_0 is the set of points $p_i = (x_i, y_i) : x_i \ge x_0$ and $y_i = y_0$, that is, it is the set of points due right of p_0 along with p_0 itself. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{p_1 p_2}$ in $O(1)$ time by reducing the problem to that of determining whether two line segments intersect.

$p_1.y = p_2.y = 0$ and $\max(p_1.x, p_2.x) \ge 0$.

or

sign($p_1.y$) ≠ sign($p_2.y$) and $\displaystyle p_1.y \cdot \frac{p_1.x - p_2.x}{p_1.y - p_2.y} \ge 0{equation}

33.1-7

One way to determine whether a point p_0 is in the interior of a simple, but not necessarily convex, polygon P is to look at any ray from p_0 and check that the ray intersects the boundary of P an odd number of times but that p_0 itself is not on the boundary of P. Show how to compute in $\Theta(n)$ time whether a point p_0 is in the interior of an n-vertex polygon P. (Hint: Use Exercise 33.1-6. Make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.)

Based on exercise 33.1-6, use $p_i - p_0$ as p_i.

33.1-8

Show how to compute the area of an n-vertex simple, but not necessarily convex, polygon in $\Theta(n)$ time. (See Exercise 33.1-5 for definitions pertaining to polygons.)

Half of the sum of the cross products of $\overline{p_1 p_i}, \overline{p_1 p_{i+1}} \mid i \in [2, n-1]$.

33.2 Determining whether any pair of segments intersects

33.2-1

Show that a set of n line segments may contain $\Theta(n^2)$ intersections.

Star.

33.2-2

Given two segments a and b that are comparable at x, show how to determine in $O(1)$ time which of $a \succeq_x b$ or $b \succeq_x a$ holds. Assume that neither segment is vertical.

(Omit!)

33.2-3

Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that instead of returning upon finding an intersection, it prints the segments that intersect and continues on to the next iteration of the **for** loop. The professor calls the resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints all intersections, from left to right, as they occur in the set of line segments. Professor Dixon disagrees, claiming that Professor Mason's idea is incorrect. Which professor is right? Will PRINT-INTERSECTING-SEGMENTS always find the leftmost intersection first? Will it always find all the intersections?

No.

33.2-4

Give an $O(n \lg n)$-time algorithm to determine whether an n-vertex polygon is simple.

Same as ANY-SEGMENTS-INTERSECT.

33.2-5

Give an $O(n \lg n)$-time algorithm to determine whether two simple polygons with a total of n vertices intersect.

Same as ANY-SEGMENTS-INTERSECT.

33.2-6

A ***disk*** consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have any point in common. Give an $O(n \lg n)$- time algorithm to determine whether any two disks in a set of n intersect.

Same as ANY-SEGMENTS-INTERSECT.

33.2-7

Given a set of n line segments containing a total of k intersections, show how to output all k intersections in $O((n + k) \lg)$ time.

Treat the intersection points as event points.

33.2-8

Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more segments intersect at the same point.

(Omit!)

33.2-9

Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 33.2-2 change if we allow vertical segments?

(Omit!)

33.3 Finding the convex hull

33.3-1

Prove that in the procedure GRAHAM-SCAN, points p_1 and p_m must be vertices of $CH(Q)$.

To see this, note that p_1 and p_m are the points with the lowest and highest polar angle with respect to p_0. By symmetry, we may just show it for p_1 and we would also have it for p_m just by reflecting the set of points across a vertical line.

To see a contradiction, suppose that we have the convex hull doesn't contain p_1. Then, let p be the point in the convex hull that has the lowest polar angle with respect to p_0. If p is on the line from p_0 to p_1, we could replace it with p_1 and have a convex hull, meaning we didn't start with a convex hull.

If we have that it is not on that line, then there is no way that the convex hull given contains p_1, also contradicting the fact that we had selected a convex hull

33.3-2

Consider a model of computation that supports addition, comparison, and multiplication and for which there is a lower bound of $\Omega(n \lg n)$ to sort n numbers. Prove that $\Omega(n \lg n)$ is a lower bound for computing, in order, the vertices of the convex hull of a set of n points in such a model.

Let our n numbers be $a_1, a_2, \ldots, a_n$ and f be a strictly convex function, such as e^x. Let $p_i = (a_i, f(a_i))$. Compute the convex hull of $p_1, p_2, \ldots, p_n$. Then every point is in the convex hull. We can recover the numbers themselves by looking at the x-coordinates of the points in the order returned by the convex-hull algorithm,

which will necessarily be a cyclic shift of the numbers in increasing order, so we can recover the proper order in linear time.

In an algorithm such as GRAHAM-SCAN which starts with the point with minimum y-coordinate, the order returned actually gives the numbers in increasing order.

33.3-3

> Given a set of points Q, prove that the pair of points farthest from each other must be vertices of $CH(Q)$.

Suppose that p and q are the two furthest apart points. Also, to a contradiction, suppose, without loss of generality that p is on the interior of the convex hull. Then, construct the circle whose center is q and which has p on the circle. Then, if we have that there are any vertices of the convex hull that are outside this circle, we could pick that vertex and q, they would have a higher distance than between p and q. So, we know that all of the vertices of the convex hull lie inside the circle. This means that the sides of the convex hull consist of line segments that are contained within the circle. So, the only way that they could contain p, a point on the circle is if it was a vertex, but we suppsed that p wasn't a vertex of the convex hull, giving us our contradiction.

33.3-4

> For a given polygon P and a point q on its boundary, the ***shadow*** of q is the set of points r such that the segment $\overline{qr}$ is entirely on the boundary or in the interior of P. As Figure 33.10 illustrates, a polygon P is ***star-shaped*** if there exists a point p in the interior of P that is in the shadow of every point on the boundary of P. The set of all such points p is called the ***kernel*** of P. Given an n-vertex, star-shaped polygon P specified by its vertices in counterclockwise order, show how to compute $CH(P)$ in $O(n)$ time.

We simply run GRAHAM-SCAN but without sorting the points, so the runtime becomes $O(n)$. To prove this, we'll prove the following loop invariant: At the start of each iteration of the for loop of lines 7-10, stack S consists of, from bottom to top, exactly the vertices of $CH(Q_{i-1})$. The proof is quite similar to the proof of correctness. The invariant holds the first time we execute line 7 for the same reasons outline in the section. At the start of the ith iteration, S contains $CH(Q_{i-1})$. Let p_j be the top point on S after executing the while loop of lines 8-9, but before p_i is pushed, and let pk be the point just below p_j on S. At this point, S contains $CH(Q_j)$ in counterclockwise order from bottom to top. Thus, when we push p_i, S contains exactly the vertices of $CH(Q_j \cup \{p_i\})$.

We now show that this is the same set of points as $CH(Q_i)$. Let p_t be any point that was popped from S during iteration i and p_r be the point just below p_t on stack S at the time p_t was popped. Let p be a point in the kernel of P. Since the angle $\angle p_r p_t p_i$ makes a nonelft turn and P is star shaped, p_t must be in the interior or on the boundary of the triangle formed by p_r, p_i, and p. Thus, p_t is not in the convex hull of Q_i, so we have $CH(Q_i - \{p_t\}) = CH(Q_i)$. Applying this equality repeatedly for each point removed from S in the while loop of lines 8-9, we have $CH(Q_j \cup \{p_i\}) = CH(Q_i)$.

When the loop terminates, the loop invariant implies that S consists of exactly the vertices of $CH(Q_m)$ in counterclockwise order, proving correctness.

33.3-5

> In the ***on-line convex-hull problem***, we are given the set Q of n points one point at a time. After receiving each point, we compute the convex hull of the points seen so far. Obviously, we could run Graham's scan once for each point, with a total running time of $O(n^2 \lg n)$. Show how to solve the on-line convex-hull problem in a total of $O(n^2)$ time.

Suppose that we have a convex hull computed from the previous stage $\{q_0, q_1, \ldots, q_m\}$, and we want to add a new vertex, p in and keep track of how we should change the convex hull.

First, process the vertices in a clockwise manner, and look for the first time that we would have to make a non-left to get to p. This tells us where to start cutting vertices out of the convex hull. To find out the upper bound on the vertices that we need to cut out, turn around, start processing vertices in a clockwise manner and see the first time that we would need to make a non-right.

Then, we just remove the vertices that are in this set of vertices and replace the with p. There is one last case to consider, which is when we end up passing ourselves when we do our clockwise sweep.

Then we just remove no vertices and add p in in between the two vertices that we had found in the two sweeps. Since for each vertex we add we are only considering each point in the previous step's convex hull twice, the runtime is $O(nh) = O(n^2)$ where h is the number of points in the convex hull.

33.3-6 ⋆

> Show how to implement the incremental method for computing the convex hull of n points so that it runs in $O(n \lg n)$ time.

(Omit!)

33.4 Finding the closest pair of points

33.4-1

> Professor Williams comes up with a scheme that allows the closest-pair algorithm to check only 5 points following each point in array Y'. The idea is always to place points on line l into set P_L. Then, there cannot be pairs of coincident points on line l with one point in P_L and one in P_R. Thus, at most 6 points can reside in the $\delta \times 2\delta$ rectangle. What is the flaw in the professor's scheme?

In particular, when we select line l, we may be unable perform an even split of the vertices. So, we don't neccesarily have that both the left set of points and right set of points have fallen to roughly half. For example, suppose that the points are all arranged on a vertical line, then, when we recurse on the the left set of points, we haven't reduced the problem size at all, let alone by a factor of two. There is also the issue in this setup that you may end up asking about a set of size less than two when looking at the right set of points.

33.4-2

> Show that it actually suffices to check only the points in the 5 array positions following each point in the array Y'.

Since we only care about the shortest distance, the distance δ' must be strictly less than δ. The picture in Figure 33.11(b) only illustrates the case of a nonstrict inequality. If we exclude the possibility of points whose x coordinate differs by exactly δ from l, then it is only possible to place at most 6 points in the $\delta \times 2\delta$ rectangle, so it suffices to check on the points in the 5 array positions following each point in the array Y'.

33.4-3

> We can define the distance between two points in ways other than euclidean. In the plane, the L_m-***distance*** between points p_1 and p_2 is given by the expression $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Euclidean distance, therefore, is L_2-distance. Modify the closest-pair algorithm to use the L_1-distance, which is also known as the ***Manhattan distance***.

In the analysis of the algorithm, most of it goes through just based on the triangle inequality. The only main point of difference is in looking at the number of points that can be fit into a $\delta \times 2\delta$ rectangle. In particular, we can cram in two more points than the eight shown into the rectangle by placing points at the centers of the two squares that the rectangle breaks into. This means that we need to consider points up to 9 away in Y' instead of 7 away. This has no impact on the asymptotics of the algorithm and it is the only correction to the algorithm that is needed if we switch from L_2 to L_1.

33.4-4

> Given two points p_1 and p_2 in the plane, the L_∞-distance between them is given by $\max(|x_1 - x_2|, |y_1 - y_2|)$. Modify the closest-pair algorithm to use the L_∞-distance.

We can simply run the divide and conquer algorithm described in the section, modifying the brute force search for $|P| \leq 3$ and the check against the next 7 points in Y' to use the L_∞ distance. Since the L_∞ distance between two points is always less than the euclidean distance, there can be at most 8 points in the $\delta \times 2\delta$ rectangle which we need to examine in order to determine whether the closest pair is in that box. Thus, the modified algorithm is still correct and has the same runtime.

33.4-5

> Suppose that $\Omega(n)$ of the points given to the closest-pair algorithm are covertical. Show how to determine the sets P_L and P_R and how to determine whether each point of Y is in P_L or P_R so that the running time for the closest-pair algorithm remains $O(n \lg n)$.

We select the line l so that it is roughly equal, and then, we won't run into any issue if we just pick an arbitrary subset of the vertices that are on the line to go to one side or the other.

Since the analysis of the algorithm allowed for both elements from P_L and P_R to be on the line, we still have correctness if we do this. To determine what values of Y belong to which of the set can be made easier if we select our set going to P_L to be the lowest however many points are needed, and the P_R to be the higher points. Then, just knowing the index of Y that we are looking at, we know whether that point belonged to P_L or to P_R.

33.4-6

> Suggest a change to the closest-pair algorithm that avoids presorting the Y array but leaves the running time as $O(n \lg n)$. (Hint: Merge sorted arrays Y_L and Y_R to form the sorted array Y.)

In addition to returning the distance of the closest pair, the modify the algorithm to also return the points passed to it, sorted by y-coordinate, as Y. To do this, merge Y_L and Y_R returned by each of its recursive calls. If we are at the base case, when $n \leq 3$, simply use insertion sort to sort the elements by y-coordinate directly. Since each merge takes linear time, this doesn't affect the recursive equation for the runtime.

Problem 33-1 Convex layers

> Given a set Q of points in the plane, we define the ***convex layers*** of Q inductively. The first convex layer of Q consists of those points in Q that are vertices of $CH(Q)$. For $i > 1$, define Q_i to consist of the points of Q with all points in convex layers $1, 2, \ldots, i-1$ removed. Then, the ith convex layer of Q is $CH(Q_i)$ if $Q_i \neq \emptyset$ and is undefined otherwise.
>
> **a.** Give an $O(n^2)$- time algorithm to find the convex layers of a set of n points.
>
> **b.** Prove that $\Omega(n \lg n)$ time is required to compute the convex layers of a set of n points with any model of computation that requires $\Omega(n \lg n)$ time to sort n real numbers.

(Omit!)

Problem 33-2 Maximal layers

> Let Q be a set of n points in the plane. We say that point (x, y) ***dominates*** point (x', y') if $x \geq x'$ and $y \geq y'$. A point in Q that is dominated by no other points in Q is said to be ***maximal***. Note that Q may contain many maximal points, which can be organized into ***maximal layers*** as follows. The first maximal layer L_1 is the set of maximal points of Q. For $i > 1$, the ith maximal layer L_i is the set of maximal points in $Q - \cup_{j=1}^{i-1} L_j$.
>
> Suppose that Q has k nonempty maximal layers, and let y_i be the y-coordinate of the leftmost point in L_i for $i = 1, 2, \ldots, k$. For now, assume that no two points in Q have the same x- or y-coordinate.
>
> **a.** Show that $y_1 > y_2 > \cdots > y_k$.
>
> Consider a point (x, y) that is to the left of any point in Q and for which y is distinct from the y-coordinate of any point in Q. Let $Q' = Q \cup \{(w, y)\}$.
>
> **b.** Let j be the minimum index such that $y_j < y$, unless $y < y_k$, in which case we let $j = k + 1$. Show that the maximal layers of Q' are as follows:
>
> - If $j \leq k$, then the maximal layers of Q' are the same as the maximal layers of Q, except that L_j also includes (x, y) as its new leftmost point.
> - If $j = k + 1$, then the first k maximal layers of Q' are the same as for Q, but in addition, Q' has a nonempty $(k + 1)$st maximal layer: $L_{k+1} = \{(x, y)\}$.
>
> **c.** Describe an $O(n \lg n)$-time algorithm to compute the maximal layers of a set Q of n points. (Hint: Move a sweep line from right to left.)
>
> **d.** Do any difficulties arise if we now allow input points to have the same x- or y-coordinate? Suggest a way to resolve such problems.

(Omit!)

Problem 33-3 Ghostbusters and ghosts

> A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is very dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross.
>
> Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are colinear.
>
> **a.** Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \lg n)$ time.
>
> **b.** Give an $O(n^2 \lg n)$-time algorithm to pair Ghostbusters with ghosts in such a way that no streams cross.

(Omit!)

Problem 33-4 Picking up sticks

> Professor Charon has a set of n sticks, which are piled up in some configuration. Each stick is specified by its endpoints, and each endpoint is an ordered triple giving its (x, y, z) coordinates. No stick is vertical. He wishes to pick up all the sticks, one at a time, subject to the condition that he may pick up a stick only if there is no other stick on top of it.
>
> **a.** Give a procedure that takes two sticks a and b and reports whether a is above, below, or unrelated to b.
>
> **b.** Describe an efficient algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order in which to pick them up.

(Omit!)

Problem 33-5 Sparse-hulled distributions

> Consider the problem of computing the convex hull of a set of points in the plane that have been drawn according to some known random distribution. Sometimes, the number of points, or size, of the convex hull of n points drawn from such a distribution has expectation $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$. We call such a distribution ***sparse-hulled***. Sparse-hulled distributions include the following:
>
> - Points drawn uniformly from a unit-radius disk. The convex hull has expected size $\Theta(n^{1/3})$.
> - Points drawn uniformly from the interior of a convex polygon with k sides, for any constant k. The convex hull has expected size $\Theta(\lg n)$.
> - Points drawn according to a two-dimensional normal distribution. The convex p hull has expected size $\Theta(\sqrt{\lg n})$.
>
> **a.** Given two convex polygons with n_1 and n_2 vertices respectively, show how to compute the convex hull of all $n_1 + n_2$ points in $O(n_1 + n_2)$ time. (The polygons may overlap.)
>
> **b.** Show how to compute the convex hull of a set of n points drawn independently according to a sparse-hulled distribution in $O(n)$ average-case time. (Hint: Recursively find the convex hulls of the first $n/2$ points and the second $n/2$ points, and then combine the results.)

(Omit!)

34 NP-Completeness

34.1 Polynomial time

34.1-1

> Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph, $u, v \in V, k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges $\}$. Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in P$.

Showing that LONGEST-PATH-LENGTH being polynomial implies that LONGEST-PATH is polynomial is trivial, because we can just compute the length of the longest path and reject the instance of LONGEST-PATH if and only if k is larger than the number we computed as the length of the longest path.

Since we know that the number of edges in the longest path length is between 0 and $|E|$, we can perform a binary search for it's length. That is, we construct an instance of LONGEST-PATH with the given parameters along with $k = \frac{|E|}{2}$. If we hear yes, we know that the length of the longest path is somewhere above the halfway point. If we hear no, we know it is somewhere below. Since each time we are halving the possible range, we have that the procedure can require $O(\lg |E|)$ many steps. However, running a polynomial time subroutine $\lg n$ many times still gets us a polynomial time procedure, since we know that with this procedure we will never be feeding output of one call of LONGEST-PATH into the next.

34.1-2

> Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.

The problem LONGST-SIMPLE-CYCLE is the relation that associates each instance of a graph with the longest simple cycle contained in that graph. The decision problem is, given k, to determine whether or not the instance graph has a simple cycle of length at least k. If yes, output 1. Otherwise output 0. The language corresponding to the decision problem is the set of all $\langle G, k\rangle$ such that $G = (V, E)$ is an undirected graph, $k \geq 0$ is an integer, and there exists a simple cycle in G consisting of at least k edges.

34.1-3

> Give a formal encoding of directed graphs as binary strings using an adjacencymatrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

(Omit!)

34.1-4

> Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

This isn't a polynomial-time algorithm. Recall that the algorithm from Exercise 16.2-2 had running time $\Theta(nW)$ where W was the maximum weight supported by the knapsack. Consider an encoding of the problem. There is a polynomial encoding of each item by giving the binary representation of its index, worth, and weight, represented as some binary string of length $a = \Omega(n)$. We then encode W, in polynomial time. This will have length $\Theta(\lg W) = b$. The solution to this problem of length $a + b$ is found in time $\Theta(nW) = \Theta(a \cdot 2^b)$. Thus, the algorithm is actually exponential.

34.1-5

> Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

(Omit!)

34.1-6

> Show that the class P, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in P$, then $L_1 \cup L_2 \in P$, $L_1 \cap L_2 \in P$, $L_1L_2 \in P$, $\overline{L_1} \in P$, and $L_1^* \in P$.

(Omit!)

34.2 Polynomial-time verification

34.2-1

> Consider the language GRAPH-ISOMORPHISM $= \{\langle G_1, G_2\rangle : G_1$ and G_2 are isomorphic graphs$\}$. Prove that GRAPH-ISOMORPHISM $\in$ NP by describing a polynomial-time algorithm to verify the language.

(Omit!)

34.2-2

> Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian.

(Omit!)

34.2-3

> Show that if HAM-CYCLE $\in P$, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

(Omit!)

34.2-4

> Prove that the class NP of languages is closed under union, intersection, concatenation, and Kleene star. Discuss the closure of NP under complement.

(Omit!)

34.2-5

> Show that any language in NP can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k.

(Omit!)

34.2-6

> A ***hamiltonian path*** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH $= \{\langle G, u, v\rangle :$ there is a hamiltonian path from u to v in graph $G\}$ belongs to NP.

(Omit!)

34.2-7

> Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.

(Omit!)

34.2-8

> Let ϕ be a boolean formula constructed from the boolean input variables $x_1, x_2, \ldots, x_k$, negations ($\neg$), ANDs ($\wedge$), ORs ($\vee$), and parentheses. The formula ϕ is a ***tautology*** if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in$ co-NP.

(Omit!)

34.2-9

> Prove that P $\subseteq$ co-NP.

(Omit!)

34.2-10

> Prove that if NP $\neq$ co-NP, then P $\neq$ NP.

(Omit!)

34.2-11

> Let G be a connected, undirected graph with at least 3 vertices, and let G^3 be the graph obtained by connecting all pairs of vertices that are connected by a path in G of length at most 3. Prove that G^3 is hamiltonian. (Hint: Construct a spanning tree for G, and use an inductive argument.)

(Omit!)

34.3 NP-completeness and reducibility

34.3-1

> Verify that the circuit in Figure 34.8(b) is unsatisfiable.

(Omit!)

34.3-2

> Show that the $\leq_P$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

(Omit!)

34.3-3

> Prove that $L \leq_P \overline{L}$ if and only if $\overline{L} \leq_P L$.

(Omit!)

34.3-4

> Show that we could have used a satisfying assignment as a certificate in an alternative proof of Lemma 34.5. Which certificate makes for an easier proof?

(Omit!)

34.3-5

> The proof of Lemma 34.6 assumes that the working storage for algorithm A occupies a contiguous region of polynomial size. Where in the proof do we exploit this assumption? Argue that this assumption does not involve any loss of generality.

(Omit!)

34.3-6

> A language L is ***complete*** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_P L$ for all $L' \in C$. Show that $\emptyset$ and $\{0, 1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

(Omit!)

34.3-7

> Show that, with respect to polynomial-time reductions (see Exercise 34.3-6), L is complete for NP if and only if $\overline{L}$ is complete for \text{co-NP}$.

(Omit!)

34.3-8

> The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit $C = f(x)$ based on knowledge of x, A, and k. Professor Sartre observes that the string x is input to F, but only the existence of A, k, and the constant factor implicit in the $O(n^k)$ running time is known to F (since the language L belongs to NP), not their actual values. Thus, the professor concludes that F can't possibly construct the circuit C and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

(Omit!)

34.4 NP-completeness proofs

34.4-1

> Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size n that, when converted to a formula by this method, yields a formula whose size is exponential in n.

(Omit!)

34.4-2

> Show the 3-CNF formula that results when we use the method of Theorem 34.10 on the formula (34.3).

(Omit!)

34.4-3

> Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

(Omit!)

34.4-4

> Show that the problem of determining whether a boolean formula is a tautology is complete for co-NP. (Hint: See Exercise 34.3-7.)

(Omit!)

34.4-5

> Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

(Omit!)

34.4-6

> Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

(Omit!)

34.4-7

> Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT $\in P$. Make your algorithm as efficient as possible. (Hint: Observe that $x \vee y$ is equivalent to $\neg x \rightarrow y$. Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

(Omit!)

34.5 NP-complete problems

34.5-1

> The ***subgraph-isomorphism problem*** takes two undirected graphs G_1 and G_2, and it asks whether G_1 is isomorphic to a subgraph of G_2. Show that the subgraphisomorphism problem is NP-complete.

(Omit!)

34.5-2

Given an integer $m \times n$ matrix A and an integer m-vector b, the ***0-1 integerprogramming problem*** asks whether there exists an integer n-vector x with elements in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (Hint: Reduce from 3-CNF-SAT.)

(Omit!)

34.5-3

The integer ***linear-programming problem*** is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector x may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

(Omit!)

34.5-4

Show how to solve the subset-sum problem in polynomial time if the target value t is expressed in unary.

(Omit!)

34.5-5

The ***set-partition problem*** takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\bar{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$. Show that the set-partition problem is NP-complete.

(Omit!)

34.5-6

Show that the hamiltonian-path problem is NP-complete.

(Omit!)

34.5-7

The ***longest-simple-cycle problem*** is the problem of determining a simple cycle (no repeated vertices) of maximum length in a graph. Formulate a related decision problem, and show that the decision problem is NP-complete.

(Omit!)

34.5-8

In the ***half 3-CNF satisfiability*** problem, we are given a 3-CNF formula ϕ with n variables and m clauses, where m is even. We wish to determine whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.

(Omit!)

Problem 34-1 Independent set

An ***independent set*** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The ***independent-set problem*** is to find a maximum-size independent set in G.

a. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete. (Hint: Reduce from the clique problem.)

b. Suppose that you are given a "black-box" subroutine to solve the decision problem you defined in part (a). Give an algorithm to find an independent set of maximum size. The running time of your algorithm should be polynomial in $|V|$ and $|E|$, counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special cases are polynomial-time solvable.

c. Give an efficient algorithm to solve the independent-set problem when each vertex in G has degree 2. Analyze the running time, and prove that your algorithm works correctly.

d. Give an efficient algorithm to solve the independent-set problem when G is bipartite. Analyze the running time, and prove that your algorithm works correctly. (Hint: Use the results of Section 26.3.)

(Omit!)

Problem 34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only 2 different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a nonnegative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

(Omit!)

Problem 34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph $G = (V, E)$ in which each vertex represents a country and vertices whose respective countries share a border are adjacent. Then, a ***k-coloring*** is a function $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$. In other words, the numbers $1, 2, \ldots, k$ represent the k colors, and adjacent vertices must have different colors. The ***graph-coloring problem*** is to determine the minimum number of colors needed to color a given graph.

a. Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT. Given a formula ϕ of m clauses on n variables $x_1, x_2, \ldots, x_n$, we construct a graph $G = (V, E)$ as follows. The set V consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause, and 3 special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on x_i, $\neg x_i$, and RED for $i = 1, 2, \ldots, n$.

d. Argue that in any 3-coloring c of a graph containing the literal edges, exactly one of a variable and its negation is colored $c(\text{TRUE})$ and the other is colored $c(\text{FALSE})$. Argue that for any truth assignment for ϕ, there exists a 3-coloring of the graph containing just the literal edges.

The widget shown in Figure 34.20 helps to enforce the condition corresponding to a clause $(x \vee y \vee z)$. Each clause requires a unique copy of the 5 vertices that are heavily shaded in the figure; they connect as shown to the literals of the clause and the special vertex TRUE.

e. Argue that if each of x, y, and z is colored $c(\text{TRUE})$ or $c(\text{FALSE})$, then the widget is 3-colorable if and only if at least one of x, y, or z is colored $c(\text{TRUE})$.

f. Complete the proof that 3-COLOR is NP-complete.

(Omit!)

Problem 34-4 Scheduling with profits and deadlines

Suppose that we have one machine and a set of n tasks $a_1, a_2, \ldots, a_n$, each of which requires time on the machine. Each task a_j requires t_j time units on the machine (its processing time), yields a profit of p_j, and has a deadline d_j. The machine can process only one task at a time, and task a_j must run without interruption for t_j consecutive time units. If we complete task a_j by its deadline d_j, we receive a profit p_j, but if we complete it after its deadline, we receive no profit. As an optimization problem, we are given the processing times, profits, and deadlines for a set of n tasks, and we wish to find a schedule that completes all the tasks and returns the greatest amount of profit. The processing times, profits, and deadlines are all nonnegative numbers.

a. State this problem as a decision problem.

b. Show that the decision problem is NP-complete.

c. Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to n. (Hint: Use dynamic programming.)

d. Give a polynomial-time algorithm for the optimization problem, assuming that all processing times are integers from 1 to n.

(Omit!)

35 Approximation Algorithms

35.1 The vertex-cover problem

35.1-1

Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.

(Omit!)

35.1-2

Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph G.

(Omit!)

35.1-3 *

Professor Bündchen proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. (Hint: Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

Consider a bibartite graph with left part L and right part R such that L has 5 vertices of degrees $(5, 5, 5, 5, 5)$ and R has 11 vertices of degrees $(5, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1)$ (the graph is easy to draw and the figure is omitted here).

Clearly there exists a vertex-cover of size 5 (the left vertices). The idea is to show that the proposed algorithm chooses all the vertices on the right part, resulting in the approximation ratio of $11/5 > 2$.

1. After choosing the first vertex in R, the degrees on L decrease to $(4, 4, 4, 4, 4)$.
2. After choosing the second vertex in R, the degrees on L decrease to $(4, 3, 3, 3, 3)$.
3. After choosing the third vertex in R, the degrees on L decrease to $(3, 3, 2, 2, 2)$.
4. After choosing the fourth vertex in R, the degrees on L decrease to $(2, 2, 2, 2, 1)$.
5. After choosing the fifth vertex in R, the degrees on L decrease to $(2, 2, 1, 1, 1)$.
6. After choosing the sixth vertex in R, the degrees on L decrease to $(1, 1, 1, 1, 1)$.

Now the algorithm still has to choose 5 more vertices.

35.1-4

Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

(Omit!)

35.1-5

From the proof of Theorem 34.12, we know that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

(Omit!)

35.2 The traveling-salesman problem

35.2-1

Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

Let $u, v \in V$. Let w be any other vertex in V. Then by the triangle inequality we have

$$c(w, u) + c(u, v) \geq c(w, v)$$
$$c(w, v) + c(v, u) \geq c(w, u).$$

Adding the above inequalities together gives

$$\cancel{c(w, u)} + \cancel{c(w, v)} + c(u, v) + c(v, u) \geq \cancel{c(w, v)} + \cancel{c(w, u)}.$$

Since G is undirected we have $c(u, v) = c(v, u)$, and so the above inequality gives $2c(u, v) \geq 0$ from which the result follows.

35.2-2

Show how in polynomial time we can transform one instance of the traveling-salesman problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict Theorem 35.3, assuming that $P \neq NP$.

(Omit!)

35.2-3

Consider the following ***closest-point heuristic*** for building an approximate traveling-salesman tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v. Extend the cycle to include u by inserting u just after v. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

(Omit!)

35.2-4

In the ***bottleneck traveling-salesman problem***, we wish to find the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (Hint: Show recursively that we can visit all the nodes in a bottleneck spanning tree, as discussed in Problem 23-3, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost that is at most the cost of the costliest edge in a bottleneck hamiltonian cycle.)

(Omit!)

35.2-5

Suppose that the vertices for an instance of the traveling-salesman problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points u and v. Show that an optimal tour never crosses itself.

(Omit!)

35.3 The set-covering problem

35.3-1

Consider each of the following words as a set of letters: {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}. Show which set cover GREEDY-SET-COVER produces when we break ties in favor of the word that appears first in the dictionary.

(Omit!)

35.3-2

Show that the decision version of the set-covering problem is NP-complete by reducing it from the vertex-cover problem.

(Omit!)

35.3-3

Show how to implement GREEDY-SET-COVER in such a way that it runs in time $O\left(\sum_{S \in \mathcal{F}} |S|\right)$.

(Omit!)

35.3-4

Show that the following weaker form of Theorem 35.4 is trivially true:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\}.$$

(Omit!)

35.3-5

GREEDY-SET-COVER can return a number of different solutions, depending on how we break ties in line 4. Give a procedure BAD-SET-COVER-INSTANCE(n) that returns an n-element instance of the set-covering problem for which, depending on how we break ties in line 4, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

(Omit!)

35.4 Randomization and linear programming

35.4-1

Show that even if we allow a clause to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

(Omit!)

35.4-2

The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly 3 literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

(Omit!)

35.4-3

In the MAX-CUT problem, we are given an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 23 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that for each vertex v, we randomly and independently place v in S with probability $1/2$ and in $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

$\gdef\Ex{\mathbf{E}}$ $\gdef\Prob{\mathbf{P}}$ $\gdef\opt{\mathtt{opt}}$

We first rewrite the algorithm for clarity.

```
APPROX-MAX-CUT(G)
    for each v in V
        flip a fair coin
        if heads
            add v to S
        else
            add v to V - S
```

This algorithm clearly runs in linear time. For each edge $(u, v) \in E$, define the event A_{uv} to be the event where edge (i, j) crosses the cut $(S, V - S)$, and let $1_{A_{uv}}$ be the indicator random variable for A_{uv}.

The event A_{ij} occurs if and only if the vertices u and v are placed in different sets during the main loop in APPROX-MAX-CUT. Hence,

$$\begin{aligned}\Prob\{A_{uv}\} &= \Prob\{u \in S \wedge v \in V - S\} + P\{u \in V - S \wedge v \in S\} \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \\ &= \frac{1}{2}.\end{aligned}$$

Let $\opt$ denote the cost of a maximum cut in G, and let $c = |(S, V - S)|$, that is, the size of the cut produced by APPROX-MAX-CUT. Clearly $c = \sum_{(u,v) \in E} 1_{A_{uv}}$. Also, note that $\opt \le |E|$ (this is tight iff G is bipartite). Hence,

$$\begin{aligned}\Ex[c] &= \Ex\left[\sum_{(u,v) \in E} 1_{A_{uv}}\right] \\ &= \sum_{(u,v) \in E} \Ex[1_{A_{uv}}] \\ &= \sum_{(u,v) \in E} \Prob\{A_{uv}\} \\ &= \frac{1}{2}|E| \\ &\ge \frac{1}{2}\opt\end{aligned}$$

Hence, $\Ex\left[\frac{\opt}{c}\right] \le \frac{|E|}{1/2|E|} = 2$, and so APPROX-MAX-CUT is a randomized 2-approximation algorithm.

35.4-4

Show that the constraints in line (35.19) are redundant in the sense that if we remove them from the linear program in lines (35.17)–(35.20), any optimal solution to the resulting linear program must satisfy $x(v) \le 1$ for each $v \in V$.

(Omit!)

35.5 The subset-sum problem

35.5-1

Prove equation (35.23). Then show that after executing line 5 of EXACT-SUBSET-SUM, L_i is a sorted list containing every element of P_i whose value is not more than t.

(Omit!)

35.5-2

Using induction on i, prove inequality (35.26).

(Omit!)

35.5-3

Prove inequality (35.29).

(Omit!)

35.5-4

How would you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

(Omit!)

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

(Omit!)

Problem 35-1 Bin packing

Suppose that we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is NP-hard. (Hint: Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^{n} s_i$.

b. Argue that the optimal number of bins required is at least $\lceil S \rceil$.

c. Argue that the first-fit heuristic leaves at most one bin less than half full.

d. Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.

e. Prove an approximation ratio of 2 for the first-fit heuristic.

f. Give an efficient implementation of the first-fit heuristic, and analyze its running time.

(Omit!)

Problem 35-2 Approximating the size of a maximum clique

Let $G = (V, E)$ be an undirected graph. For any $k \ge 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.

a. Prove that the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G.

b. Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

(Omit!)

Problem 35-3 Weighted set-covering problem

Suppose that we generalize the set-covering problem so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i and the weight of a cover $\mathcal{F}$ is $\sum_{S_i \in \mathcal{F}} w_i$. We wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Show that your heuristic has an approximation ratio of $H(d)$, where d is the maximum size of any set S_i.

(Omit!)

Problem 35-4 Maximum matching

Recall that for an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. In Section 26.3, we saw how to find a maximum matching in a bipartite graph. In this problem, we will look at matchings in undirected graphs in general (i.e., the graphs are not required to be bipartite).

a. A ***maximal matching*** is a matching that is not a proper subset of any other matching. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. (Hint: You can find such a graph with only four vertices.)

b. Consider an undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

In this problem, we shall concentrate on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

c. Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.

d. Consider a maximal matching M in $G = (V, E)$. Let

$$T = \{v \in V : \text{ some edge in } M \text{ is incident on } v\}.$$

What can you say about the subgraph of G induced by the vertices of G that are not in T?

e. Conclude from part (d) that $2|M|$ is the size of a vertex cover for G.

f. Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

(Omit!)

Problem 35-5 Parallel machine scheduling

In the ***parallel-machine-scheduling problem***, we are given n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k. We are also given m identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job J_k, the machine on which it runs and the time period during which it runs. Each job J_k must run on some machine M_i for p_k consecutive time units, and during that time period no other job may run on M_i. Let C_k denote the ***completion time*** of job J_k, that is, the time at which job J_k completes processing. Given a schedule, we define $C_{max} = \max_{1 \le k \le n} C_k$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, suppose that we have two machines M_1 and M_2 and that we have four jobs J_1, J_2, J_3, J_4, with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine M_1, job J_1 followed by job J_2, and on machine M_2, it runs job J_4 followed by job J_3. For this schedule, $C_1 = 2$,

$C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{max} = 14$. An optimal schedule runs J_2 on machine M_1, and it runs jobs J_1, J_3, and J_4 on machine M_2. For this schedule, $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$, and $C_{max} = 12$.

Given a parallel-machine-scheduling problem, we let C^*_{max} denote the makespan of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{max} \geq \max_{1 \leq k \leq n} p_k .$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k .$$

Suppose that we use the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

$$C_{max} \leq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k .$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

(Omit!)

Problem 35-6 Approximating a maximum spanning tree

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, let $\max(v) = \max_{(u,v) \in E} \{w(u, v)\}$ be the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let T_G be the maximum-weight spanning tree of G, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

a. Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

b. Give an example of a graph with at least 4 vertices for which $S_G \neq T_G$.

c. Prove that $S_G \subseteq T_G$ for any graph G.

d. Prove that $w(T_G) \geq w(S_G)/2$ for any graph G.

e. Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

(Omit!)

Problem 35-7 An approximation algorithm for the 0-1 knapsack problem

Recall the knapsack problem from Section 16.2. There are n items, where the ith item is worth v_i dollars and weighs w_i pounds. We are also given a knapsack that can hold at most W pounds. Here, we add the further assumptions that each weight w_i is at most W and that the items are indexed in monotonically decreasing order of their values: $v_1 \geq v_2 \geq \cdots \geq v_n$.

In the 0-1 knapsack problem, we wish to find a subset of the items whose total weight is at most W and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that we are allowed to take a fraction of each item, rather than being restricted to taking either all or none of each item. If we take a fraction x_i of item i, where $0 \leq x_i \leq 1$, we contribute $x_i w_i$ to the weight of the knapsack and receive value $x_i v_i$. Our goal is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, we consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, we form restricted instances I_j, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j - 1$ and requiring the solution to include item j (all of item j in both the fractional and 0-1 knapsack problems). No items are removed in instance I_1. For instance I_j, let P_j denote an optimal solution to the 0-1 problem and Q_j denote an optimal solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

b. Prove that we can find an optimal solution Q_j to the fractional problem for instance I_j by including item j and then using the greedy algorithm in which at each step, we take as much as possible of the unchosen item in the set $\{j + 1, j + 2, \ldots, n\}$ with maximum value per pound v_i/w_i.

c. Prove that we can always construct an optimal solution Q_j to the fractional problem for instance I_j that includes at most one item fractionally. That is, for all items except possibly one, we either include all of the item or none of the item in the knapsack.

d. Given an optimal solution Q_j to the fractional problem for instance I_j, form solution R_j from Q_j by deleting any fractional items from Q_j. Let $v(S)$ denote the total value of items taken in a solution S. Prove that $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

e. Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

(Omit!)