Chapter 1

Efficient Multiplication, I

1.1. Introduction

Part of the purpose of an introductory course in applied mathematics is to change our vantage on how we view math. For this reason, we'll revisit some things that you've hopefully seen in previous courses, but our goal will be different. In many pure math courses, the focus is on finding a solution. This is of course a good goal – it's nice to be able to solve the problem! Unfortunately, for many real world applications it's not enough to know that we can do something; we need to be able to do it efficiently.

A great example is factorization. Any integer $N \geq 2$ can be written uniquely as a product of prime powers $p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$, where $1 < p_1 < p_2 < \cdots < p_k$ (we need to impose this ordering of the primes to get uniqueness, as we don't want to consider $2^2 \cdot 3$, $2 \cdot 3 \cdot 2$ and $3 \cdot 2^2$ as different factorizations of 12). The proof that such a factorization exists and is unique is fairly straightforward; we leave finding it and analyzing run-times to the reader in Exercises 1.7.1 and 1.7.2. Thus while it's easy to give a procedure to factor a number, *finding* that factorization quickly is a different story. Many modern encryption systems are in fact built around this perceived difficulty (remember that just because no one has published a fast way to factor doesn't mean no fast way exists).

So, being able to do something is not enough; we need to be able to do it well. This means we must be able to solve the problem in a reasonable amount of time with a reasonable amount of resources; it's unlikely to expect to have every subatomic particle in the universe replaced with a supercomputer devoted to performing our computations throughout all eternity (which is less than what we would need to make some of the brute force approaches work for numbers used in practice).

While we can sometimes find better algorithms, often there's a trade-off between efficiency and exactness. Fortunately, for many problems we do not need an exact answer, and frequently a close approximation which can be found quickly

suffices. This is terrific, as unlike factorization, some problems are not only computationally difficult but deny a closed form solution. For an advanced example, we know from Galois Theory that there's no closed form expression for the roots of polynomials of degree greater than or equal to 5; however, there are iterative methods (Newton's method, discussed in §13.5, is a great one) that can yield solutions accurate to, say, 32 decimal places, within moments. For many problems, 32 digits of accuracy is more than enough (and if we needed more we could run our algorithm longer).

Of course, there are also problems where we can neither find an exact solution nor a good approximation quickly; not surprisingly, we won't focus too much on those in this book. We'll concentrate on problems where we can either quickly find the answer or at least a very good approximation. As a warm-up, we explore fast ways to multiply numbers and the consequences of these improvements over brute force. In addition to having tremendous applications, these problems provide a nice tour to the ideas and approaches we'll see later.

We begin our tour with a very familiar operation: multiplication. This is a natural choice, as it allows us to start with something you've seen numerous times but with the opportunity of highlighting aspects and applications you've probably never seen.

1.2. Babylonian Multiplication

We start our tour of multiplication with the Babylonians. Much of their mathematics is still with us today. Some is apparent in our everyday life and is due to their working in base 60 (most people use base 10, the **decimal** system, though base 2, **binary**, is very useful in computer science). 60 is a great choice, as it is evenly divisible by many different numbers, which is advantageous when you have to divide quantities into groups. To this day, we can see their influence in 60 seconds in a minute, 60 minutes in an hour, 360 degrees in a circle, and so on.

Other aspects, however, are not as well known. An important one is a result of the difficulty of doing mathematics in base 60. In elementary school we learn the multiplication table; if we can just learn the products $d_1 d_2$ with each $d_i \in \{0, 1, \ldots, 9\}$ we can perform any multiplication. In base 60, however, this would require a mastery of $60 \cdot 60 = 3600$ products!

Of course, it isn't quite as bad as that, as multiplication is commutative and this almost cuts in half the products we must memorize: $\frac{1}{2}60 \cdot 59 + 60 = 1830$; the first factor is the number of products (where order does not matter) of two distinct numbers, and then of course we must include the squares. While this almost cuts our work in half, it's too much to expect the ancient Babylonian children and merchants to memorize almost 2000 products!

Fortunately, there is a better way. They noticed that

$$a \cdot b \;=\; \frac{(a+b)^2 - (a-b)^2}{4}.$$

At first this looks like a horrible trade, as we're replacing one multiplication with two products, a subtraction and a division! The gain becomes apparent, however, when we notice that the products are both squares. Thus if we just learn our

squares, subtraction, and division by 4, we can do any multiplication! Note that there are only 120 squares to learn (remember $a + b$ can get as large as 120). This is tremendous progress – we've gone from needing to learn 1830 products to just 120, a savings of more than an order of magnitude.

The great idea lurking in the background here is that of a **look-up table**. If there is a computation you will need again and again, do it once, store the value, and then recall it as needed. Additionally, we can store a few key values and use those to interpolate the function at other points (see Chapter 18). These savings are essential for many modern applications. For example, cell phones and other similar devices have had a tremendous infiltration in our society; we seem to be constantly online and connected. While the computing power of these devices has steadily increased, we are putting smaller and smaller devices online, and thus efficiencies in computing are not only welcomed but needed. A terrific example is streaming video; we don't want to have to send every pixel every moment!

1.3. Horner's Algorithm

Our Babylonian example introduced one of the great concepts in modern computing: the look-up table. If there is a problem you need to do again and again, it's often a great strategy to do it once, save the result, and recall as needed. What if, however, we have an entirely new computation (or if memory is expensive)? For a nice example, let's consider a polynomial

$$f(x) \;=\; a_n x^n + \cdots + a_1 x + a_0.$$

You should know an easy way to find its value for any given x, say $x = 2$; all we need to do is plug in 2 for x above and evaluate. Thus if

$$f(x) \;=\; 3x^4 - 11x^3 + 5x - 6$$

we find

$$f(2) \;=\; 3 \cdot 2^4 - 11 \cdot 2^3 + 5 \cdot 2 - 6 \;=\; -36.$$

In general, how 'expensive' is this procedure? How many operations does it take? While we'll count both additions and multiplications, for large numbers addition is essentially free relative to the cost of multiplication (think of how many digit operations are required to add two 100 digits numbers versus multiplying them). To evaluate our general f at $x = 2$ we would do

$$a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2 + a_0.$$

Notice the k^{th} term requires k multiplications ($k - 1$ to compute 2^k and then one more to multiply that by a_k). So the total number of multiplications needed, if we use this brute force approach, is

$$n + (n-1) + (n-2) + \cdots + 1 + 0 \;=\; \frac{n(n+1)}{2}$$

(see Exercise 1.7.13). Thus the total cost, in multiplications, of evaluating the polynomial at $x = 2$ is on the order of n^2 (or if you prefer, approximately $n^2/2$). See Exercise 1.7.17 for ways to estimate the sum of the first n integers.

Amazingly, we can do significantly better.

Horner's Algorithm: By grouping the terms and rewriting the polynomial $f(x)$ as

$$f(x) \;=\; (\cdots(((a_n x + a_{n-1})\,x + a_{n-2})\,x + a_{n-3})\cdots + a_1)\,x + a_0,$$

it costs n multiplications and n additions to evaluate $f(x)$.

Example 1.3.1. *If*

$$f(x) \;=\; 2x^4 + 5x^3 + 7x^2 - 2x + 8,$$

then we re-write it as

$$f(x) \;=\; (((2x+5)x + 7)\,x - 2)\,x + 8$$

and see there are 4 multiplications and 4 additions, a savings over the 10 multiplications and 4 additions needed for the standard approach.

Thus, Horner's Algorithm reduces the multiplications cost of polynomial evaluation to n, while maintaining the additions cost at n; compared to the brute force approach, there is a square-root savings in the number of multiplications required. To give a sense of how significant this improvement is, for a polynomial of order $n = 1000$, this means we need only do 1000 multiplications instead of a massive $1,000,000$. Efficient algorithms like this one have many useful applications; for instance, Horner's Algorithm is very helpful in the study of fractals, which require thousands of iterations of polynomials. Further, this example illustrates a very important principle: *Just because we have done something one way for a long time does not mean this is the best way to do it.*

It seems like we can't do much better than Horner's Algorithm: given that a polynomial of order n has n terms, it seems that we need at least n multiplications to evaluate the expression. While this is true for a general polynomial, perhaps there are faster methods for special polynomials. *You should get in the habit of always asking questions such as this:* ***If we make additional assumptions, are there improvements to be had?*** We'll see in the next section that the answer is most definitely 'yes', and the resulting fast multiplication has tremendous applications (such as RSA encryption).

1.4. Fast Multiplication

In the last section we found that polynomial evaluation, which at first appeared to require on the order of n^2 multiplications (if it is of degree n), could be done significantly faster with just n products by cleverly grouping. Unfortunately, for a general polynomial we cannot do better, but perhaps we can for special polynomials. When looking for special cases to investigate, start simple. The simplest degree n polynomial is $f(x) = x^n$. We can write this as

$$f(x) \;=\; (\cdots(((x \cdot x) \cdot x) \cdot x) \cdot x)\cdots x$$

and see that in this case brute force and Horner's Algorithm give the same answer: $n - 1$ multiplications.

Interestingly, we can do significantly better by again cleverly grouping the products. We use the binary expansion of n to save on the required number of

multiplications. We'll first illustrate the method with an example. Note that a subscript of 2 after a number whose digits are just 0's and 1's means that it's written in binary; thus 1101_2 is $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, or 13.

Example 1.4.1. *Let $f(x) = x^{100}$. We write 100 in binary:*

$$100 \;=\; 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 0 \cdot 1 \;=\; 1100100_2.$$

We now repeatedly square x, saving the results, and then multiply the powers which correspond to a non-zero digit in the binary expansion:

$$\begin{aligned} x \cdot x &= x^2 \\ x^2 \cdot x^2 &= x^4 \\ x^4 \cdot x^4 &= x^8 \\ x^8 \cdot x^8 &= x^{16} \\ x^{16} \cdot x^{16} &= x^{32} \\ x^{32} \cdot x^{32} &= x^{64} \end{aligned} \qquad\qquad x^{100} \;=\; x^{64} \cdot x^{32} \cdot x^4.$$

There is of course nothing special about 100.

Fast Multiplication: To compute x^n, write n in binary. Repeatedly square $(x, x^2, x^4, x^8, \dots)$, save the results, and then multiply the terms corresponding to powers present in the binary expansion of n.

Is this better than brute force? Yes, as far as the number of required multiplications is concerned. For our example with $n = 100$ we now only need to perform 8 multiplications, as opposed to 99 for brute force; in Exercise 1.7.26 you'll show that you need at most $2\log_2(n)$ products.

Unfortunately, this method does require something that brute force does not: we need to store the products from the repeated squaring. Fortunately, it's easy to save on the storage requirements as well. Let's revisit our example from before. Start with the pair $(1, x)$, and square the second element in the pair. Whenever this yields a power of x that is in the binary expansion of x^{100}, multiply the first element by this power of x. Repeating this process goes as follows:

$$\begin{aligned} (1, x) \to (1, x^2) \to (1, x^4) \to (x^4, x^4) \to (x^4, x^8) \to (x^4, x^{16}) \\ \to (x^4, x^{32}) \to (x^{36}, x^{32}) \to (x^{36}, x^{64}) \to (x^{100}, x^{64}). \end{aligned}$$

We still get x^{100} in only 8 multiplications (if you don't count the $1 \cdot x^4$ that we used to update our first component), but now we only require storing 1 additional value compared to brute force and the binary expansion of n.

This problem illustrates many of the issues that arise in implementing mathematics: efficiency, storage capacity, and even overflow (see Exercise 1.7.23). Fast multiplication is one of the key ingredients in many modern encryption schemes; without fast and secure methods to encode and decode information, e-commerce as we know it could not exist. We describe one of the most used systems, RSA, in a series of problems beginning with Exercise 2.6.81.

1.5. Strassen's Algorithm

The repeated squaring algorithm from the last section is amazing and probably a bit surprising the first time you see it. It allows us to replace an order N process with one that is order $\log N$ (see Exercise 1.7.29). The real lesson here is that while you may know a way to do something, it can be well worth the time to investigate and see if there is a better way.

A terrific example, similar in spirit to what we've done, is in matrix multiplication. You should hopefully have seen the importance of matrix multiplication (if not, we provide one of many examples in §1.6). How expensive is the brute force approach we've been taught?

While for many applications we're often interested in high powers of a fixed matrix, let's start by looking a bit more generally (similar to Horner's method). Given matrices $\mathbf{A}$ and $\mathbf{B}$, the entry in the i^{th} row and j^{th} column of $\mathbf{AB}$ is the dot product of the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$. For two 2×2 matrices we have

$$\mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{21} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

Notice this requires 8 multiplications and 4 additions. More generally,

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2j} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nj} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & \cdots & c_{1n} \\ c_{21} & \ddots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \mathbf{c_{ij}} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & \cdots & c_{nn} \end{pmatrix}$$

with

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}.$$

Each c_{ij} costs n multiplications and $n-1$ additions; this means the cost of finding C is n^3 multiplications and $n^3 - n^2$ additions.

Thus, if we use brute force, matrix multiplication (of two $n \times n$ matrices) is an order n^3 process; as we can have matrices with more than 10,000 rows, *any* savings on the exponent 3 in the n^3 is highly desirable. The difficulty is that we are finding n^2 dot products, each costing n multiplications. If we wanted a specific c_{ij} we would be out of luck; however, we want *all* of them, and this suggests a way out. Rather than computing each c_{ij} one at a time, perhaps there are other helpful quantities we can compute first, and then use to efficiently find the c_{ij}. This turns out to be possible; the first such approach is due to Strassen, which allows us to do matrix multiplication in order $n^{\log_2 7}$ steps.

Essentially what happens is we save one multiplication. As $\log_2 8 = 3$ and $\log_2 7 \approx 2.8073$, for a matrix with 10,000 rows the brute force would require $1.0 \cdot 10^{12}$ multiplications, while the Strassen algorithm needs about $1.7 \cdot 10^{11}$. Thus we save almost a factor of 6, and the savings only get larger as we increase n (we save a little over a factor of 200 for matrices with a trillion rows)!

Strassen's Algorithm (for $n = 2$): Given 2×2 matrices $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$, compute the following seven quantities:

$$\begin{aligned} m_1 &= (a_{11} + a_{12})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}). \end{aligned}$$

Then $\mathbf{C} = \mathbf{AB}$ can be computed by

$$\begin{aligned} c_{11} &= m_1 + m_4 - m_5 + m_7 \\ c_{12} &= m_3 + m_5 \\ c_{21} &= m_2 + m_4 \\ c_{22} &= m_1 - m_2 + m_5 + m_6, \end{aligned}$$

which costs 7 multiplications and 18 additions (as opposed to the naive algorithm's 8 multiplications and 4 additions).

While we only discussed the case of two 2×2 matrices, Strassen's algorithm can be extended to arbitrary square matrices. Not surprisingly, it's easier to discuss when n is a power of 2. We split 4×4 blocks into 2×2 blocks, 8×8 blocks into 4×4 blocks, and so on. In these cases, the cost in multiplications of finding the product of two matrices is taken from $n^3 = n^{\log_2 8}$ to $n^{\log_2 7}$, while increasing the required number of additions. While this improvement may seem underwhelming, this was in fact one of the biggest advances in Linear Algebra of the 20th century: when n becomes very large, the savings on computational efficiency are enormous (see Exercise 1.7.31).

Finally, we should note that this is a problem for which it's easy to check that a potential solution works, but very hard to find the solution (the correct set of seven multiplications) in the first place. Sadly, this type of problem is common throughout mathematics.

1.6. Eigenvalues, Eigenvectors and the Fibonacci Numbers

We end our tour of multiplication with a nice example of why we care so much about it. One of the most studied sequences in mathematics is the **Fibonacci numbers** $\{F_n\}$, where $F_0 = 0$, $F_1 = 1$, and for $n > 0$ we set $F_{n+1} = F_n + F_{n-1}$. If this were theoretical mathematics, we might not be very concerned with any difficulties in calculating the values of F_n for given n, for the recursion formula gives us a simple procedure to march down the line and compute any desired term. In practice, however, this can be quite difficult to implement. For example, if we want the trillionth Fibonacci number we would need to compute all the previous

ones. Fortunately, it turns out that we can use matrices to jump to the desired term.

Let $\vec{v}_n = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$, so $\vec{v}_{n+1} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$. We can represent this using a matrix equation:

$$\vec{v}_{n+1} \;=\; \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \vec{v}_n.$$

Indeed, we have

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \vec{v}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \vec{v}_{n+1}.$$

Therefore, if we now let $\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, we find

$$\vec{v}_{n+1} \;=\; \mathbf{A}\vec{v}_n \;=\; \mathbf{A}^2\vec{v}_{n-1} \;=\; \cdots \;=\; \mathbf{A}^n\vec{v}_1.$$

Thus, if we can compute high powers of a matrix quickly, we can jump ahead to whatever Fibonacci number we want. See Exercise 1.7.41 for another example.

Remark 1.6.1. *Whenever you see a problem in mathematics, you should look for generalizations or special cases where you have intuition. Here we are taking large powers of a matrix; this should recall our earlier investigations into large powers of a number. We saw an enormous advantage in writing the binary expansion of the exponent; we can do that here as well! Thus to compute A^{100} we would do $\boldsymbol{A}^{64}\boldsymbol{A}^{32}\boldsymbol{A}^4$.*

We conclude this chapter with one final remark: often one does not know how a math result will be useful, but only that there is a chance it may be so. Just because we can multiply matrices relatively quickly now does not mean we should proceed this way; perhaps there is another approach using more advanced ideas.

Given a matrix $\mathbf{A}$, $\mathbf{A}\vec{v}$ is typically a new vector with a different length and direction than $\vec{v}$; if, however, $\mathbf{A}\vec{v}$ is in the same direction as $\vec{v}$ (and is not the zero vector), we say $\vec{v}$ is an **eigenvector** of $\mathbf{A}$. We can thus write $\mathbf{A}\vec{v} = \lambda\vec{v}$ for some $\lambda \in \mathbb{C}$, and call λ the **eigenvalue** associated to $\vec{v}$; note λ could be zero, but $\vec{v} \neq \vec{0}$. The advantage of eigenvectors is that matrix multiplication just rescales their length, and we can thus compute high powers quickly.

For example, the Spectral Theorem asserts that if $\mathbf{A}$ is a real-symmetric matrix (the result holds more generally), then there is an orthonormal basis of eigenvectors. This means that we have eigenvectors $\vec{v}_i$ with eigenvalues λ_i such that given any $\vec{v}$ we have

$$\vec{v} \;=\; c_1\vec{v}_1 + \cdots + c_n\vec{v_n},$$

and thus

$$\mathbf{A}^m\vec{v} \;=\; c_1\lambda_1^m\vec{v}_1 + \cdots + c_n\lambda_n^m\vec{v_n}.$$

The difficulty, of course, is in finding the eigenvalues and eigenvectors of the matrix. We know from Linear Algebra that we can find the eigenvalues of an $n \times n$

matrix $\mathbf{A}$ by finding the solutions to the equation

$$\det(\mathbf{A} - \lambda \mathbf{I}_n) \;=\; 0,$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix. As an aside, if you've never been warned about trying to use this approach for large matrices, consider yourself warned! There are stability issues, and once $n \geq 5$ we no longer have closed form expressions for the roots.

Fortunately, there are many situations in which eigenvalues and eigenvectors are easy to find. For instance, let's revisit the Fibonacci sequence. By finding the eigenvalues, eigenvectors, and matching the initial conditions, in Exercise 1.7.38 you will prove **Binet's formula** for the n^{th} Fibonacci number:

$$F_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

This expression is quite surprising: even though it's filled with fractions and irrational numbers, it always yields an integer result given a natural number n. Further, notice how instead of finding a large power of a 2×2 matrix we need only compute high powers of two real numbers, another great savings.

1.7. Exercises

Introduction:

Exercise 1.7.1. *Find an algorithm to factor an integer $N \geq 2$ that requires on the order of N divisions. Can you do $N/2$? What about $N/3$? What about N^δ for some $\delta < 1$?*

Exercise 1.7.2. *In the previous problem you hopefully came up with several approaches to factor a number. It's important to be able to compare them and see how much of an improvement the observations make. First assume that any division costs the same to perform as any other; how many divisions do each of your algorithms require? Now assume that the cost of a division depends on the number of digit multiplications (to keep things simple we'll assume there is zero cost for addition and carrying, though you are welcome to estimate the effects of these). Thus as 41446301 divided by 1701 is 24601, the cost of the division would be 20 digit multiplications. Now how expensive are the methods?*

Exercise 1.7.3. *In fourth grade my son had a homework assignment requiring him to count how often each letter of the alphabet appears in three sentences. While we did this with a computer program, imagine you have to do it by hand. Which is a better method:*

- *read letter by letter, and as you read increase the running tally for that letter, or*
- *read the entire sentence from start to finish and count the number of* `a`*'s, then read from start to finish to count the number of* `b`*'s, and so on*

(or will the two methods take the same amount of time)? Try this for several sentences, and if the two do not take the same amount of time see which factors affect which wins. Are there other approaches you could do by hand?

Babylonian Multiplication:

Exercise 1.7.4. *Another way to do multiplication is to note*

$$a \cdot b \;=\; \frac{(a+b)^2 - a^2 - b^2}{2};$$

how does this compare to the Babylonian method?

Exercise 1.7.5. *If you were a Babylonian, how would you efficiently compute $a \cdot b \cdot c$ and $a \cdot b \cdot c \cdot d$?*

There are many ways to create and use a look-up table. If we only have a small number of possibilities, as we did with multiplying base 60, we can just do an exhaustive enumeration. Most of the time, however, it's not possible to precompute every possibility, and we must somehow interpolate between entries. The next few problems explore some methods to do just that. An important part of the problems is to determine how to measure errors; for example, an error of size .01 is much worse if the values are around .2 than if they are around 20.

Exercise 1.7.6. *If we know the values of either* $\sin(x)$ *or* $\cos(x)$ *for* $0 \le x \le \pi/4$ *(or from 0 to 45 degrees if you prefer not to work in radians), then we can find the values of all trig functions using basic relations (such as* $\sin(x + \pi/2) = \cos(x)$*). Create a look-up table of* $\sin(x)$ *by finding its values for* $x = \frac{k}{45}\frac{\pi}{4}$ *with* $k \in \{0, 1, \ldots, 45\}$*. Come up with at least two different ways to interpolate values of* $\sin(x)$ *for* x *not on your list, and compare their accuracies. For which values of* x *are your interpolations most accurate?*

A **continuous probability distribution** (or **density**) f is a non-negative function that integrates to 1; the area under f from a to b is the probability that we observe a value between a and b.

Exercise 1.7.7. *One of the most important densities in probability theory is the standard normal:*

$$f(x) \;=\; \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

The corresponding cumulative distribution function, $F(x)$*, gives the probability of obtaining a value at most* x*:*

$$F(x) \;=\; \int_{-\infty}^{x} f(t)dt;$$

note $F' = f$*. Unfortunately there is no simple closed form expression for the antiderivative. We could try to find a series expansion for it by expanding the exponential; unfortunately, as one of the bounds is negative infinity we cannot integrate term by term. We are saved by noting that* $F(0) = 1/2$ *and thus*

$$F(x) \;=\; \frac{1}{2} + \int_{0}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt;$$

the convention is to define the ***error function*** *(or* ***erf****) by*

$$\mathrm{erf}(x) \;=\; \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du \;=\; 2 \int_0^{x\sqrt{2}} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt.$$

Create a look-up table for $F(x)$ for $x \in [0,5]$ (say in steps of size .1). Come up with at least two different ways to interpolate values of $\operatorname{erf}(x)$ for x not on your list, and compare their accuracies. For which values of x are your interpolations most accurate?

Exercise 1.7.8. *Imagine you want to create a look-up table to find square-roots of numbers from 0 to 100. If you could only have 10 entries in your table, what would you choose and why? What if you could have 20 values?*

You should have a great familiarity with **logarithms**, but as their uses extend to (and far beyond) this course, as well as provide an excellent example of look-up tables in action, we'll quickly go over their basics. We say $\log_b x = y$ if and only if $x = b^y$; note log is always the logarithm base e. We have the following, familiar **laws of logarithms**:

(1) $\log_b x_1 + \log_b x_2 = \log_b(x_1 x_2)$,

(2) $\log_b(x^r) = r \log_b x$,

(3) $b^{\log_b x} = x$,

(4) $\log_b x_1 - \log_b x_2 = \log_b(\frac{x_1}{x_2})$,

(5) $\log_b x = \frac{\log_c x}{\log_c b}$, the oft-forgotten change of basis formula!

Exercise 1.7.9. *Prove the five log laws. For example, for the first we have $\log_b x_i = y_i$, so $x_i = b^{y_i}$. Thus $x_1 x_2 = b^{y_1} b^{y_2} = b^{y_1+y_2}$. By definition, we now get $\log_b(x_1 x_2) = y_1 + y_2$, which finally yields $\log_b(x_1 x_2) = \log_b x_1 + \log_b x_2$.*

The change of basis formula is extremely useful because it allows us to compute logarithms in any base so long as we know how to compute them in a single base. Before we had calculators, the ability to do this was crucial since all values of functions had to be tabulated manually: **look-up tables**, which store pre-computed values, were vital in increasing efficiency. The change of basis formula meant that we only needed a single look-up table, for a single basis value, in order to compute logarithms of any kind, which allowed for immense savings in terms of storage, time, and efficiency.

Exercise 1.7.10. *Prove $\log(e^x) = x$ and $e^{\log x} = x$; thus the exponential and logarithm functions are inverses. This allows us to pass from the derivative of one to the derivative of the other via the Chain Rule: if $f(g(x)) = x$, then $f'(g(x))g'(x) = 1$. Using the derivative of e^x is e^x find the derivative of $\log x$, as well as the derivatives of b^x and $\log_b x$ for a fixed $b > 1$.*

Horner's Algorithm:

Exercise 1.7.11. *Consider a polynomial $f(x) = a_n x^n + \cdots + a_1 x + a_0$, and assume $a_i = 0$ whenever i is even. How can you modify Horner's Algorithm to efficiently compute $f(x)$?*

Exercise 1.7.12. *How would you generalize Horner's Algorithm to a polynomial in two variables? In d variables? For example, how would you efficiently compute*

$$f(x,y) \;=\; \sum_{m=0}^{M} \sum_{n=0}^{n} a_{mn} x^m y^n$$

at $(x,y) = (a,b)$?

Exercise 1.7.13. *Show that*

$$0 + 1 + 2 + \cdots + (n-1) + n \;=\; \frac{n(n+1)}{2}.$$

Hint: one way is to proceed by induction; another way is to write the numbers in reverse order, add, and then divide by 2.

Exercise 1.7.14. *Generalize the previous problem and find a formula for the sum of the squares of integers up to n. Find a formula for the sum of cubes.*

Exercise 1.7.15. *In the last two problems you should have noticed that your answer is always a polynomial, and perhaps you also noticed something interesting about the exponent of the highest degree term as well as its coefficient. Make some conjectures, test with other power sums, and try to prove. For more on this topic look up Bernoulli polynomials.*

Exercise 1.7.16. *Here is another way to get the result from Exercise* 1.7.13*. Let $S_k(n) = 1^k + 2^k + \cdots + n^k$. Then*

$$(n+1)^2 \;=\; S_2(n+1) - S_2(n) \;=\; \sum_{m=0}^{n} \left((m+1)^2 - m^2 \right) \;=\; \sum_{m=0}^{n} (2m+1).$$

Complete the argument. Can you generalize this approach and get a formula for $S_3(n)$ or $S_4(n)$?

Exercise 1.7.17. *In Exercise* 1.7.13 *you proved that the sum of the first n integers is $n(n+1)/2$, which is of approximately $n^2/2$. For many problems the actual value doesn't matter; all we need is a good approximation. The following provides a way to sketch that, in some sense, this sum grows at a similar rate as cn^2 for some constant c:*

- *Show that the sum is at most n^2.*
- *Show that the sum is at least $(n/2) \cdot (n/2)$ if n is even (if n is odd show it's at least $\frac{n-1}{2}\frac{n-1}{2}$).*

The previous problem shows that $1 + 2 + \cdots + n$ is at least $n^2/4$ and at most n^2; the actual answer is about $n^2/2$. Interestingly one obtains the correct approximation by taking the **geometric mean** of the two numbers, where the geometric mean of k numbers is $(a_1 a_2 \cdots a_k)^{1/k}$; this is different than the **arithmetic mean**, which would be $\frac{1}{k}(a_1 + a_2 + \cdots + a_k)$.

Exercise 1.7.18. *Let a_1 and a_2 be two positive numbers. Prove that their arithmetic mean is never smaller than their geometric mean. When are the two equal?*

Exercise 1.7.19. *More generally, given $a_1, \dots, a_k > 0$ prove that their arithmetic mean is never smaller than their geometric mean. When are the two equal?*

Exercise 1.7.20. *Given $a_1, \dots, a_n > 0$, define the k^{th} symmetric mean $M_k(a_1, \dots, a_n)$ by*

$$M_k(a_1, \dots, a_n) \;=\; \left(\sum_{1 \le i_1 < i_2 < \cdots < i_k \le n} \frac{a_{i_1} a_{i_2} \cdots a_{i_k}}{\binom{n}{k}} \right)^{1/k};$$

note $M_1(a_1, \dots, a_n)$ is the arithmetic mean and $M_n(a_1, \dots, a_n)$ is the geometric mean. Prove ***Maclaurin's inequality****:*

$$M_1(a_1, \dots, a_n) \;\ge\; M_2(a_1, \dots, a_n) \;\ge\; \cdots \;\ge\; M_n(a_1, \dots, a_n)$$

(see [**B-AC**] *if you are stuck).*

The next problem explores how we can take our initial crude bound and refine it. Of course, while for this problem we can easily determine the exact answer, our goal is to introduce a new method that is useful for a variety of other problems.

Exercise 1.7.21. *Let's revisit our estimation for $1+2+\cdots+n$; in Exercise* 1.7.17 *we saw it's between $n^2/4$ and n^2. We highlight a powerful technique, called* ***dyadic decomposition****, which allows us to improve these bounds and close their gap. For simplicity you may assume $n = 2^k$ so we can divide n evenly by any power of 2.*

We first split $\{1, 2, \dots, n/2, n/2+1, \dots, n\}$ into two sets: $S_{11} = \{1, 2, \dots, n/2\}$ and $S_{12} = \{n/2+1, n/2+2, \dots, n\}$ and apply our bounds to the sum from each, remembering that $n/2$ is playing the role of n from before. Thus the sum of the terms in S_{11} is at least $(n/2)^2/4$ and at most $(n/2)^2$. For S_{12}, notice that each term is $n/2$ more than the corresponding term in S_{11}, and therefore its sum is $(n/2)\cdot(n/2) = n^2/4$ more than the sum from S_{11}. Combining we find

$$\frac{(n/2)^2}{4} + \frac{n^2}{4} + \frac{(n/2)^2}{4} \;\le\; 1+2+\cdots+n \;\le\; \left(\frac{n}{2}\right)^2 + \frac{n^2}{4} + \left(\frac{n}{2}\right)^2$$

or

$$\frac{3}{8}n^2 \;\le\; 1+2+\cdots+n \;\le\; \frac{6}{8}n^2.$$

Notice that the bounds are a lot closer and that the lower bound is less than 1/2 and the upper bound exceeds 1/2 (so our answers are reasonable).

Iterate this procedure one or two more times. Do the upper and lower bounds appear to converge to 1/2? Can you prove that rigorously?

Exercise 1.7.22. *The* ***factorial*** *of a non-negative integer n, written $n!$, is the product of all integers at most n (with $0!$ set to be 1); note we can interpret this as the number of ways to order n objects when order matters. Find easy upper and lower bounds for $n!$, and then use the method of dyadic decompositions from the previous problem to improve your bounds and narrow the gap between them. For n large,* ***Stirling's formula*** *states that $n! \approx (n/e)^n \sqrt{2\pi n}$.*

Fast Multiplication:

Many encryption algorithms, such as RSA, use **modular (or clock) arithmetic**. We say $x \equiv y \bmod n$ (read x is equivalent or congruent to y **modulo** n) if $x - y$ is a multiple of n; thus $5 \equiv 17 \bmod 12$ (our choice of 12 for this example is to highlight the connection with a clock). For more on RSA see the problems beginning with Exercise 2.6.81.

Exercise 1.7.23. *Estimate how large 25^{100} is. Assume we only need to know $25^{100} \bmod 47$; use fast multiplication and congruences to compute this.*

Exercise 1.7.24. *Instead of using binary we could use ternary (base 3). How many operations would it take to compute x^{100} using the ternary expansion of 100? Is it ever more efficient to use ternary over binary? If yes, try to quantify how often ternary is better and how often binary is better. Note there are many ways you can compare the two, from average number of operations to worst-case scenarios.*

Exercise 1.7.25. *Show that any non-negative integer n has an unique binary expansion $n = \sum_{i=0}^{k} d_i 2^i$ with each $d_i \in \{0, 1\}$.*

Exercise 1.7.26. *Prove that the number of multiplications required by fast multiplication to compute x^n is at most $2\log_2(n)$.*

Exercise 1.7.27. *On average how many multiplications are needed to compute x^n for $n \in \{1, \dots, N\}$? When investigating questions such as this it's often easier if we assume N is of a special form; in this case, perhaps it would be helpful to assume $N = 2^k$.*

Exercise 1.7.28. *Let $N = 2^k$. Which $n \in \{1, \dots, N\}$ require the most multiplications to compute?*

Exercise 1.7.29. *Consider four algorithms: the first runs in e^N steps, the second in N^2 steps, the third in $N^{1/2}$, and the fourth in $\log N$ steps. Percentagewise how much of an increase is there in run-time going from an input of N to $2N$ (i.e., doubling the input)? Express your answer as a function of N, but for definiteness also do it for $N \in \{100, 10^{10}, 10^{100}\}$.*

Strassen's Algorithm:

Exercise 1.7.30. *Prove that Strassen's algorithm works as claimed (i.e., that we do recover the matrix product).*

Exercise 1.7.31. *Let $n \in \{10^4, 10^9, 10^{100}\}$. Compare how many multiplications are needed to compute the product of two $n \times n$ matrices using the brute force approach and using Strassen's algorithm.*

Exercise 1.7.32. *Implement Strassen's algorithm for 4×4 matrices.*

Exercise 1.7.33. *Discuss how to implement Strassen's algorithm if n is not a power of 2. What can you do? Which $n \in \{2^k, \dots, 2^{k+1} - 1\}$ will lead to the least savings?*

Exercise 1.7.34. *Is there a fast way to multiply three matrices? Four? If yes how much better can you do than brute force?*

Exercise 1.7.35. *Let $n = 2$ and $\boldsymbol{A}$ be a 2×2 matrix. What is the most efficient way to compute $\boldsymbol{A}^m$ for $m = 2^\ell$, and how many steps does it take?*

Exercise 1.7.36. *Consider two 2×2 matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ such that the lower left entry of $\boldsymbol{A}$ and the lower right entry of $\boldsymbol{B}$ are zero. Can you improve on Strassen's algorithm? What if now these are $2^k \times 2^k$ matrices and the lower left $2^{k-1} \times 2^{k-1}$ block of $\boldsymbol{A}$ and the lower right $2^{k-1} \times 2^{k-1}$ block of $\boldsymbol{B}$ are all zeros?*

Exercise 1.7.37. *If you know the sum of the first powers up to n, one can find the the sum of the first n cubes by noting*

$$\left(\sum_{k=1}^{n} k\right)^2 \;=\; \sum_{i=1}^{n} i^2 + 2\sum_{j=2}^{n}\sum_{\ell=1}^{j-1} j\ell.$$

Prove this identity and deduce

$$\sum_{m=1}^{n} m^3 \;=\; \frac{n^2(n+1)^2}{4}.$$

Eigenvalues, Eigenvectors and the Fibonacci Numbers:

Exercise 1.7.38. *Find the eigenvalues and eigenvectors for the Fibonacci matrix*

$$\boldsymbol{A} \;=\; \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

and prove Binet's formula.

Exercise 1.7.39. *Prove that n is a Fibonacci number if and only if either $5n^2 + 4$ or $5n^2 - 4$ is a square. This problem was posed by Ira Gessel, Problem H-187 in the Fibonacci Quarterly **10** (1972), no. 4, page 417. Notice this provides a very fast test of whether or not an integer is a Fibonacci number, though it doesn't tell us which Fibonacci number it is; for more on fast tests see the Carmichael number exercises from §7.7.*

Exercise 1.7.40. *Generalize the previous problem to other recurrences; for example, is there a nice relation for the Tribonacci numbers, given by $T_{n+1} = T_n + T_{n-1} + T_{n-2}$ for appropriate choices of T_0, T_1 and T_2? For what a and b is there a nice recurrence relation describing numbers n such that n is in the sequence if and only if $an^2 \pm b$ is a square?*

Exercise 1.7.41. *Consider the following simplified model for the number of pairs of whales alive at a given moment in time. We make the following semi-reasonable assumptions:*

(1) Time moves in discrete steps of 1 year.

(2) The number of whale pairs that are 0, 1, 2, and 3 years old in year n are denoted by a_n, b_n, c_n, and d_n respectively; all whales die when they turn 4 (and at no other time).

(3) If a whale pair is 1 year old it gives birth to two new pairs of whales, if a whale pair is 2 years old it gives birth to one new pair of whales, and no other pair of whales give birth.

Let

$$v_n = \begin{pmatrix} a_n \\ b_n \\ c_n \\ d_n \end{pmatrix};$$

find a matrix $\boldsymbol{A}$ *such that*

$$v_{n+1} = \boldsymbol{A} v_n$$

(these are called ***Leslie matrices****, and are very important in mathematical biology). In Star Trek IV the crew of the Enterprise goes back in time and rescues a pair of humpback whales (George and Gracie), needed to repopulate the species and save the planet from an alien probe. How many whales will there be, according to this model, in 100 years? In 1000 years? In a million years?*

Exercise 1.7.42. *Compare how many multiplications are needed to find* F_n *using the Strassen algorithm versus Binet's formula.*

Exercise 1.7.43. *Show that for* n *large we do not need the second term in Binet's formula and can deduce* F_n *from an examination of just the first.*

The next few problems concern real matrices; they can be generalized to complex matrices.

Exercise 1.7.44 (The Triangularization Lemma)**.** *Given any* $n \times n$ *matrix* $\boldsymbol{A}$ *there exists an invertible matrix* $\boldsymbol{S}$ *such that* $\boldsymbol{S}^{-1}\boldsymbol{A}\boldsymbol{S}$ *is an upper triangular matrix* $\boldsymbol{T}$ *(i.e., the entries of* $\boldsymbol{T}$ *below the main diagonal are all zero).*

Exercise 1.7.45. *Find the eigenvalues of an upper triangular matrix* $\boldsymbol{T}$*.*

Exercise 1.7.46. *Prove that if* λ *is an eigenvalue of* $\boldsymbol{A}$*, then there is a corresponding eigenvector. Is it possible that an eigenvvalue could have multiplicity* r *but not have* r *linearly independent eigenvectors? If yes give an example.*

Exercise 1.7.47. *Prove if* $\boldsymbol{A}$ *is a real symmetric matrix (so* $\boldsymbol{A}^T = \boldsymbol{A}$*), then* A *has an orthonormal basis of eigenvectors.*

Exercise 1.7.48. *A (real)* $n \times n$ *matrix* $\boldsymbol{Q}$ *is orthogonal if* $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}_n$*. Prove that the eigenvalues of* $\boldsymbol{Q}$ *have absolute value 1 and that if* $\boldsymbol{A}$ *is a real-symmetric matrix, then we can find an orthogonal* $\boldsymbol{Q}$ *such that* $\boldsymbol{Q}^T A \boldsymbol{Q} = \Lambda$*, where* $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ *is a diagonal matrix of eigenvalues of* $\boldsymbol{A}$*. How many choices of* Λ *are there?*

Exercise 1.7.49. *We can use the Spectral Theorem to find eigenvalues, rather than the determinant formula. Assume for simplicity that* A *is an* $n \times n$ *real symmetric matrix, so it has an orthonormal basis of eigenvectors* $\vec{v}_i$ *with eigenvalues* λ_i*, ordered so that* $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$*. Consider a random vector* $\vec{v}$*; as the eigenvectors are a basis we have* c_i *such that*

$$\vec{v} = c_1\vec{v}_1 + \cdots + c_n\vec{v}_n.$$

If $|\lambda_1| > |\lambda_2|$ (and thus non-zero), then if we are fortunate enough to have $c_1 \neq 0$, then

$$A^m \overrightarrow{v} \;=\; c_1 \lambda_1^m \overrightarrow{v}_1 + \cdots + c_n \lambda_n^m \overrightarrow{v}_n \;\approx\; c_1 \lambda_1^m \overrightarrow{v}_1.$$

Thus if we take the logarithms of the lengths of both sides,

$$\log ||A^m \overrightarrow{v}|| \;\approx\; m \log \lambda_1 + \log c_1 || \overrightarrow{v}_1 ||,$$

and sending m to infinity gives

$$\lambda_1 \;=\; \lim_{m \to \infty} \frac{\log ||A^m \overrightarrow{v}||}{m}.$$

Justify the approximations above and discuss how we should send m to infinity and how we should compute the corresponding A^m. Also, the above assume $c_1 \neq 0$, but we don't know when we choose a vector whether or not the corresponding linear combination will or will not have $c_1 \neq 0$; what should we do to deal with this issue? What if $|\lambda_1| = |\lambda_2|$?

Chapter 10

Multi-Objective and Quadratic Programming

There are many ways to complicate the canonical linear programming problem. Our first example is **multi-objective linear programming**, where we have multiple items we wish to simultaneously maximize. After that we turn to **quadratic programming**.

While we have seen that some non-linearities can be brought in to linear programming, others cannot and require new methods. The bad news is that there is no method that works as well as the Simplex Method for general problems. The good news is that in many real-world situations small errors would not cause any great harm. There are several reasons for this. First, we often have to estimate parameters for our model, and thus as the model is only an estimate, perhaps it's not too bad that we cannot exactly solve an approximation to reality. Second, we can often get *very* close to the true optimal answer in a reasonable amount of time. For example, if your company measures revenues in the billions and you can quickly find a decision that is within $50 of the optimal, for all practical purposes you have found the optimal decision.

10.1. Multi-Objective Linear Programming

We now turn to multi-objective linear programming. One of the first problems we examined was the Diet Problem, where we tried to find the cheapest possible diet that satisfied the single objective of staying alive. In practice, however, there are many other objectives that could and should be involved. We might wish for the food to be palatable, for example. Or perhaps we want variety, or we need to minimize preparation time, or we care whether or not the food is organic or locally grown.

Importantly, the only objective that is not a luxury is that of staying alive. All the others are secondary. This means that we have to assign weights to the

objectives, and prioritize. The advantage of having everything be a number is that it facilitates comparisons; we're comparing apples and apples, not apples and oranges.

The need to make such hard choices is not limited to the diet problem. Perhaps we're shopping for a new car and have to weigh how much price and miles per gallon are worth to us; of course, we might also care about items such as where the car is made, how it looks, ... For one last example, imagine you own a major sports franchise and are faced with allocating your salaries among players with different skills (in baseball maybe balancing pitching versus hitting versus fielding needs).

The general framework in all of these problems is the following, where we start with a canonical Linear Programming problem but modify the objective function.

Multi-Objective Linear Programming: Consider a constraint matrix $\mathbf{A}$, leading to $\mathbf{A}\vec{x} = \vec{b}$ with $\vec{x} \geq \vec{0}$. Choose non-negative weight $w_1, \ldots, w_n$ such that $w_1 + \cdots + w_n = 1$ (which immediately implies $0 \leq w_i \leq 1$) and objective vectors $\vec{c}_1, \ldots, \vec{c}_n$. The goal is to minimize

$$w_1 \vec{c}_1{}^T \vec{x} + \cdots + w_n \vec{c}_n{}^T \vec{x}.$$

In some sense, our phrasing above is misleading as this is, in fact, *a canonical Linear Programming problem!* If we let

$$\vec{c}_w \;=\; w_1\vec{c}_1 + \cdots + w_n\vec{c}_n,$$

then our objective is to minimize $\vec{c}_w^T\vec{x}$, exactly as it should be. The challenge in multi-objective linear programming is determining the relative weights. Returning to the diet example: how much more do we care about cost than the taste of food? For purchasing a car: how much is fuel efficiency worth to us compared to convenience? We need to be able to make these relative evaluations.

Further, we need to keep everything linear. While linearity is reasonable in certain ranges, at some point such an assumption probably breaks down. For example, before streaming audio and video people used to listen to CDs, cassette tapes, records, ... (take your pick depending on your age). One could record at various quality levels, but at some point the human ear is unable to detect the improvements, and thus it would make no sense to continue giving credit for further gains.

10.2. Quadratic Programming

As we just saw, we can handle multi-objective linear programming with linear weights; this means we have a set of functions to weigh and minimize the weighted sum. There are, however, situations where we don't want linear weights; the statistical Method of Least Squares is a major example of this, which plays a large role in problems of regression in statistics.

Thus, we may not always want linear terms. We examine the simplest such case, in which the terms are quadratic.

Recall the canonical form for linear programming problems:

- $\mathbf{A}\vec{x} = \vec{b}$,
- $\vec{x} \geq \vec{0}$,
- minimize $\vec{c}^{\,T}\vec{x}$.

How could we insert quadratic elements into this canonical form? We would need to have certain elements depend on $\vec{x}$. There are only two places for this dependence.

(1) We could have the constraint matrix depend on $\vec{x}$: $\mathbf{A} = \mathbf{A}(\vec{x})$ or
(2) we could have the objective function depend on $\vec{x}$: $\vec{c} = \vec{c}(\vec{x})$.

Which of these two options seems more sensible? While both are easily realized in a variety of problems (as shown in the next section), the first is significantly harder to solve. In Exercise 10.6.5 you will prove that if we could incorporate and handle quadratic constraints, then we could solve *any* binary integer linear programming problem!

However, it turns out that we can handle quadratic elements in the objective function. Consider the following generalization of the canonical Linear Programming problem:

- $\mathbf{A}\vec{x} = \vec{b}$,
- $\vec{x} \geq \vec{0}$,
- minimize $(\vec{c}^{\,T}\vec{x} + \vec{x}^T\mathbf{C}\vec{x})$ where $\mathbf{C}$ is a **positive definite matrix** (see Exercise 10.6.6).

The Simplex Method can be generalized in this case. Recall that when we discussed the Simplex Method, we never used the linearity of the cost function; we simply needed it to be decreasing. (We also never proved that it runs as fast as we claimed it does, so perhaps the linearity is used there.) It turns out that the Simplex Method can be generalized to the case where the matrix $\mathbf{C}$ is positive definite, which means that $\vec{x}^T\mathbf{C}\vec{x} \geq \vec{0}$, with $\vec{x}^T\mathbf{C}\vec{x} = \vec{0}$ if and only if $\vec{x} = \vec{0}$. While a proof that the Simplex Method generalizes is beyond the scope of this book, the knowledge that it could should serve to give a glimpse of the broader applications of the Simplex Method.

10.3. Example: Quadratic Objective Function

In this section we introduce how quadratic and higher order functions could be introduced in our programming problem. In the next section we attempt to solve this problem using linear programming, with the hope that it will demonstrate how we could generalize the process to a large class of quadratic programming problems.

Consider the following example. We own a movie theater, and our goal is to schedule movies appropriately (and of course to make money). Assume the time slots are in 10 minute increments (labeled $0, 1, \ldots, T$), there are M movies (labeled $1, \ldots, M$), and there are S screens (labeled $1, \ldots, S$). We have the following decision variables:

$$y_{tm} = \begin{cases} 1 & \text{if movie } m \text{ is playing at time } t, \\ 0 & \text{otherwise} \end{cases}$$

and

$$d_{tm} \;=\; \text{demand for movie } m \text{ at time } t.$$

Our objective function would be to maximize the revenue of the theater. To do that, we need to know d_{tm}. However, often d_{tm} is not fixed but rather depends on the other movies that are playing. For example, it's reasonable to think that there are moviegoers who do not decide on what they want to see beforehand, and they may simply be in the mood of seeing an action movie. Assume that in our movie set we have two action blockbusters that are played at the same time. Since the two movies are competing for customers' attention, the total demand for these two movies would be smaller than the sum of the individual demand for each had they been played at different time slot.

This problem calls for some correcting factor. Here's an attempt at one. For a fixed time t, we consider the polynomial:

$$\begin{aligned} p(x) &= \prod_{m=1}^{m_1} y_{tm} \prod_{m=m_1+1}^{m_1+m_2} (1 - y_{tm}) \\ &= \begin{cases} 1 & \text{if at time } t \text{ movies 1 through } m_1 \text{ are being shown} \\ & \text{and movies } m_1+1 \text{ through } m_1+m_2 \text{ are not being shown;} \\ 0 & \text{otherwise,} \end{cases} \end{aligned}$$

in which $m = m_1 + m_2$. Here we are dividing our set of movies into two subsets of size m_1 and m_2 respectively. By considering all possible polynomials of this form, we can handle any m_1-tuple of movies playing and m_2-tuple of movies not playing.

This correction would bring polynomial non-linearities in the objective function. In the next section we will investigate on how to linearize them. Although we are concentrating on removing non-linearities in the objective function (the schedule to maximize revenue), the method is identical for removing such non-linearities from the constraints.

Note that every variable occurs to either the zeroth or first power: as $y_{tm} \in \{0, 1\}$, $y_{tm}^n = y_{tm}$ for any integer $n \geq 1$. ***This is an extremely useful consequence of having binary variables!***

10.4. Removing Quadratic (and Higher Order) Terms in Constraints

As mentioned in §10.2, there are some generalizations to linear programming methods such as the Simplex Method which work on quadratic constraints and objective functions. Describing these techniques and proving their correctness, however, is too much of a detour. We thus take a different approach and show how we can linearize these constraints, taking the problem from the above section as an example. Of course, just because we can linearize these quantities does not mean this will be the faster way to solve them; perhaps there are special techniques which could exploit the polynomial structure.

Our goal is to replace terms in the objective function of the form

$$-\text{Const} \cdot p(x)$$

with linear terms, possibly at the cost of additional variables and constraints. In our example, these $m_1 + m_2$ movies compete with each other for demand, and we must adjust the demand of each movie based on the competition concurrently screened.

How could we do that? One approach would be to look at the simplest case and try to generalize. Let $m_1 = 1, m_2 = 1$. We need to generalize

$$p_t = y_{t1}(1 - y_{t2}).$$

Notice that $p_t = 1$ if and only if $y_{t1} = 1$ and $y_{t2} = 0$, or $1 - y_{t2} = 1$. Thus we could say $p_t = y_{t1}$ **AND** $(1 - y_{t2})$. This condition could be expressed through the two constraints:

(1) $y_{t1} + (1 - y_{t2}) - 2p_t \geq 0$,

(2) $y_{t1} + (1 - y_{t2}) - p_t \leq 1$.

If $y_{t1} = 1$ and $y_{t2} = 0$, then the second constraint forces $p_t = 1$ while the first constraint does not impose any limitation. Otherwise, the second condition does not constraint p_t, but the first forces $p_t = 0$.

Now we generalize the above approach for the general case

$$p_t = \prod_{m=1}^{m_1} y_{tm} \prod_{m=m_1+1}^{m_1+m_2} (1 - y_{tm}).$$

Using similar reasoning, we replace the product with the following two constraints:

(1) $\sum_{m=1}^{m_1} y_{tm} + \sum_{m=m_1+1}^{m_1+m_2} (1 - y_{tm}) - (m_1 + m_2)p_t \geq 0$,

(2) $\sum_{m=1}^{m_1} y_{tm} + \sum_{m=m_1+1}^{m_1+m_2} (1 - y_{tm}) - p_t \leq m_1 + m_2 - 1$.

The detailed reasoning on why these constraints are equivalent to our product is left for the reader as a small practice.

10.5. Summary

As remarked, we have only touched the beginning of a very important generalization of linear programming. It is important to analyze the *cost* of linearizing our problem specifically for real world problems. Can the linearized problems be solved (or approximately solved) in a reasonable amount of time?

We are reminded again of the quote of Abraham Maslow, who remarked that if all one has is a hammer, pretty soon all problems look like nails. Once we know how to do and solve Linear Programming problems, it's tempting to convert other problems to Linear Programming problems. While this yields a reasonable solution in many situations, there are additional techniques that are better able to handle many of these problems.

10.6. Exercises

Multi-Objective Linear Programming

Exercise 10.6.1. *Revisit the diet problem. Add at least one additional objective to the objective function, choose some reasonable numbers, and determine.*

Exercise 10.6.2. *Consider the Diet Problem from §3.5. Assume we are also concerned with our fat intake and wish to minimize that. If one unit of the first food has 3 grams of fat and the second has 2, we now want a cheap diet with little fat. Solve this problem as a function of the weights $w_1, w_2 = 1 - w_1$.*

Exercise 10.6.3. *Consider a multi-objective Linear Programming problem. Must the solution vary continuously with the weights?*

Exercise 10.6.4. *Consider a multi-objective Linear Programming problem. Must the solution vary differentiably with the weights?*

Quadratic Programming

Exercise 10.6.5. *Show that* if *we had the ability to incorporate quadratic constraints and solve the resulting problem, then we would immediately be able to solve any binary integer Linear Programming problem (or, more generally, any integer Linear Programming problem!).*

The next few problems involve positive definite matrices. Recall a square matrix $\mathbf{C}$ is **positive definite** if for any non-zero vector $\vec{x}$ we have $\vec{x}^T\mathbf{C}\vec{x} > 0$.

Exercise 10.6.6. *Prove that if $\boldsymbol{C}$ is a real-symmetric matrix whose eigenvalues are all positive, then $\boldsymbol{C}$ is positive definite.*

Exercise 10.6.7. *Give an example of a 10×10 positive definite matrix such that every element of the matrix is non-zero.*

Exercise 10.6.8. *If a matrix has all of its eigenvalues distinct and positive, is it positive definite?*

Exercise 10.6.9. *One place where positive definite matrices arise is in the multivariable generalization of the second derivative test. Consider a twice continuously differentiable function $f(x_1, \ldots, x_n)$ with a critical point at $(a_1, \ldots, a_n)$ (so $(\nabla f)(a_1, \ldots, a_n) = \vec{0}$). The Hessian matrix $\boldsymbol{H}_f(a_1, \ldots, a_n)$ has a $(i,j)^{th}$ entry of $\frac{\partial^2 f}{\partial x_i \partial x_j}(a_1, \ldots, a_n)$. Prove the Hessian matrix is real-symmetric and that if all of its eigenvalues are positive, then f has a minimum at $(a_1, \ldots, a_n)$.*

Note: if you have never seen this language before, the above problem hopefully explains the involved second derivative tests from multivariable calculus, where we have a minimum or a maximum depending on values of partial derivatives and their products.

Removing Quadratic (and Higher Order) Terms

Exercise 10.6.10. *In the chapter we looked at terms arising from m_1 movies playing and m_2 movies not. What if now all we care about is that at least n_1 of the first m_1 movies are playing and at least n_2 of the next m_2 are not. Is it still possible to introduce binary indicator variables and constraints to encode this linearly?*

Exercise 10.6.11. *Now we return to the safe pawns problem we introduced in Exercise* 5.4.6. *Here are some suggestions on how to formalize the problem as a linear programming problem.*

Let x_{ij} represent whether position (i, j) is occupied by any queen and let y_{ij} represent whether position (i, j) is safe (i.e, not occupied by any queen and not attacked by any queen).

What is our objective function? How could we write y_{ij} as a function of x_{ij}? Could we replace the function with some linear constraints? Formalize the problem with linear constraints and objective function.