Intensive Algorithms Lecture 11

DFT and DP

Lecturer: Daniel A. Spielman March 1, 2022

11.1 Introduction

The purpose of this lecture is to learn how to use the Discrete Fourier Transform to save space in Dynamic Programming. I am teaching this because it introduces a completely new way to think about dynamic programming. If there is time at the end, I will talk a little about the use of polynomials for error-correction in digital communication.

The main result comes from the paper "Saving Space by Algebraization" by Lokshtanof and Nederlof [LN10]. I do not recommend that you try to read this paper because it solves a more abstract problem. The results in that paper have been improved. For example, see [JVW21].

11.2 Space and Subset Sum

The problem that we will use to demonstrate their technique is called *subset sum*. It is a special case of the knapsack problem. The input to the subset sum problem is a list of positive integers $a_1, \ldots, a_n$ along with a target integer T, and the answer is "yes" if there is an $S \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in S} a_i = T.$$

The problem only asks for the one bit answer: "yes" or "no." We may assume without loss of generality that all $a_i \leq T$, as we can safely discard any a_i that are larger.

We can solve this problem by Dynamic Programming in time $O(nT)$ by building a table of achievable sums. For example, we could set $\mathsf{opt}(j, t)$ to be true if there is an $S \subseteq \{1, \ldots, j\}$ for which $\sum_{i \in S} a_i = t$. We can then compute all the entires in this table by the recurrence

$$\mathsf{opt}(j, t) = \mathsf{opt}(j-1, t) \quad \text{or} \quad \mathsf{opt}(j-1, t-a_j).$$

In fact, we can make this table a little smaller by dropping the first coordinate. Consider the following routine:

1. For t in 1 to n, set $\mathsf{opt}(t) = \texttt{false}$. Set $\mathsf{opt}(0) = \texttt{true}$.
2. For i from 1 to n
    a. For t from T down to 1, if $a_i \leq t$ set $\mathsf{opt}(t) = \mathsf{opt}(t)$ or $\mathsf{opt}(t - a_i)$.

You can show that after the ith iteration, $\texttt{opt}(t) = \texttt{opt}(i,t)$ for all t. This version uses space $O(T)$ and takes time $O(nT)$.

We could even use a loop like this to count the number of solutions. Consider instead

1. For t in 1 to n, set $c(t) = 0$. Set $c(0) = \texttt{true}$.
2. For i from 1 to n
    a. For t from T down to 1, if $a_i \leq t$ set $c(t) = c(t) + c(t - a_i)$.

One can prove by induction that after the ith iteration, $c(t)$ equals the number of subsets of $\{1, \ldots, i\}$ that give the sum t. After observing that each $c(t)$ is always at most 2^n, we see that this version requires at most $O(nT)$ bits.

We would now like to see how to solve this problem by using less space. The important property of space / memory that we are going to exploit is that it can be reused.

To get us used to this idea, I will show that we can solve this problem using $O(n + \log T)$ bits, but $O(2^n n \log T)$ time. This is the space and time required to go through every possible subset (there are 2^n), compute the sum for that subset (in time $O(n \log T)$), and check if it equals T. We need n bits to keep track of the current subset, and $O(\log T)$ bits to write down the sum for that subset. It might seem that we could need as many as $\log(nT)$ bits, but we can avoid this by stopping if the sum exceeds T.

This algorithm is reasonable if $T \geq 2^n$, but is very slow if $T << 2^n$. We see an algorithm that uses space essentially $O(n + \log(T))$ and time almost $O(n^3 T)$.

11.3 Subset Sum by Polynomials

Consider the polynomial

$$p(x) = \prod_{i=1}^{n} (1 + x^{a_i}).$$

The coefficient of x^t in this polynomial equals $c(t)$. Our plan is to use the DFT to compute the coefficient of x^T in low space. In addition to determining if there is a subset that achieves sum T, we will find out how many there are that do.

We will, of course, do this using a Discrete Fourier Transform.

11.4 The DFT and inverse DFT

We now recalling the Discrete Fourier Transform from last lecture, and its inverse. Let $\omega = e^{2\pi i/n}$. The DFT is a linear transformation that maps a vector $(a_0, a_1, \ldots, a_{n-1})$ to a vector $(y_0, \ldots, y_{n-1})$

by setting

$$y_k = \sum_{j=0}^{n-1} a_j \left(\omega^k\right)^j = \sum_{j=0}^{n-1} a_j \omega^{jk}.$$

That is, it treats its input vector as the coefficients of a polynomial

$$p(x) = \sum_{j=0}^{n-1} a_j x^j,$$

and then sets

$$y_k = p(\omega^k)$$

for all k.

It is sometimes useful to expres the DFT as multiplication by a matrix. We may evaluate the polynomial p at a point x_0 by forming the vector of powers of x_0, and then taking its inner product with the vector of coefficients of p:

$$p(x_0) = \left(1, x_0, x_0^2, \ldots, x_0^{n-1}\right) \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

If we assemble all of those row vectors into a matrix, we obtain

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2n-2} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3n-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \omega^{3n-3} & \cdots & \omega \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Last lecture, we learned an algorithm called the FFT for computing the DFT in time $O(n \log n)$ when n is a power of 2. Today's lecture will not actually require the FFT. But, we will need the inverse of the DFT, which is what we use to interpolate the coefficients of a polynomial from its values.

We saw that we can compute the coefficients of p from its evaluations at powers of ω by the formula

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-jk} y_k.$$

There are two things that you should take away from this. The first is that we can obtain the coefficient a_j by computing the inner product of the DFT of $p(x)$ and the vector with entries ω^{-jk}. The second is that the computation that inverts the DFT is almost identical to the one that computes it.

11.5 Repeated Squaring

For an integer a, we can compute x^a while using only $O(\log a)$ arithmetic operations: we can compute x^{2^i} for various i by repeated squaring, and then multiplying together the appropriate terms. For example,

$$x^{11} = x^1 x^2 x^8.$$

In general, we expand a in binary as $b_0 + 2b_1 + 4b_2 + \cdots + 2^k b_k$, then compute

$$x^a = \prod_{j:b_j=1} x^{2^{b_j}}$$

by the loop

1. $z = x$. $y = 1$.
2. For j in 1 to k
   a. If $b_j = 1$, $y = y * z$.
   b. $z = z^2$.
3. Return y.

11.6 Numerics

Because our algorithm is going to be computing potentially large numbers, say up to 2^n or nT, and is going to be taking powers of ω of this order, we need to worry a little about numerical precision. So, I'll give a crash course on floating point computation. My favorite reference for this material is the book by Higham [Hig02]. This is the sort of thing you can learn about in a course on Numerical Analysis. But we will take a looser approach.

Floating point numbers come in the form

$$\pm m 2^e,$$

where $0 \leq m < 2^B$ is an integer, called the significand, and e is an exponent between e_{min} and e_{max}. For convenience, we will assume $e_{min} = -e_{max}$. In the standard double-precision float, $e_{max} \approx 1023$ and $B = 53$. For any number x, we let $fl(x)$ denote the closest floating point number. For every x between $2^{e_{min}}$ and $2^{t+e_{max}}$,

$$|fl(x) - x| < u\,|x|\,,$$

where

$$u = 2^{-B}.$$

Note that some numbers you might expected to represent, like $1/10$, can not be exactly represented in this way. If you play with numerical computation, you will often encounter numbers around 10^{-16} that are indistinguishable from zero. The reason is that $2^{-53} \approx 1.1 \cdot 10^{-16}$.

We will now give a loose approximation of the standard model of floating point computation. We will make the non-standard assumption that we can choose t and e_{max} for our number system, and that we will use $O(t+e_{max})$ bits to represent each number. With this many bits, we can implement approximations of the basic operations of $+$, $-$, $*$, and $/$, by operations we call $\oplus$, $\ominus$, $\otimes$, $\oslash$ with the properties that

$$fl(x \oplus y) = (x+y)(1+\delta) \quad \text{for some } |\delta| \leq u,$$

provided that all of x, y, and $x+y$ are 0 or between $2^{e_{min}}$ and $2^{t+e_{max}}$.

This does *not* mean that if $x+y+z=0$, then $fl(x \oplus y \oplus z) = 0$. The reason is that $fl(x \oplus y)$ will probably not equal z. Another problem is that you probably won't be dealing with x, y, and z, but rather approximations of them. For example, if I try to compute $0.3 - 0.1 - 0.2$ on my laptop, I get $-2.7 \cdot 10^{-17}$.

One issue here is the difference between absolute and relative error. We say that $\hat{x}$ is an ϵ additive approximation of x if $|x - \hat{x}| \leq \epsilon$. But, floating point is geared towards maintaing relative approximations. We say that $\hat{x}$ is an an ϵ relative approximation of x if $|x - \hat{x}| \leq \epsilon |x|$. Note that we should only consider $\epsilon < 1/2$. We will use the fact that an ϵ relative approximation of x is an $\epsilon |x|$ absolute approximation.

We will represent complex numbers z by the real numbers x and y such that $z = x + iy$. Adding complex numbers just requires adding the real and imaginary parts. But, multiplying them requires 4 real multiplications and 2 real additions.

To write the running time of these operations, we say that $f(n) \leq \tilde{O}(g(n))$ if there is some constant c for which

$$f(n) \leq O(g(n) \log^c g(n)).$$

This enables us skip writing low order logarithmic terms. All of the elementary floating point operations, $\oplus$, $\ominus$, $\otimes$, $\oslash$, can be performed in time $\tilde{O}(e_{max} + B)$.

We can compute complex roots of unity by computing their real and imaginary parts separately by the Taylor series for sine and cosine. This is not the most efficient way to do it. But, it will suffice for our crude purposes. I assert that to compute $e^{2\pi x}$ to ϵ additive accuracy, we need to compute at most the first $O(\log(1/\epsilon))$ terms in their Taylor series. This would take time $Otilde(log(1/\epsilon)(e_{max} + B))$.

Our approach to numerical analysis will be very crude, and would make most Numerical Analysts scream. But, it will be sufficient for our purposes as getting numerics right is not the point of this course.

11.7 The Algorithm for Subset Sum

For each $1 \leq i \leq n$, let $p_i(x) = 1 + x^{a_i}$. Recall that we can evaluate $p_i(x)$ in at most $O(\log a_i)$ operations: $O(\log a_i)$ multiplications to compute x^{a_i}, and we then add 1.

The following is a description of the algorithm for computing $c(T)$. We will write it in a way that makes it easy to reason about its space usage.

1. Set $N = nT + 1$, and $\omega = e^{2\pi i/N}$.
2. Set $c = 0$.
3. For j in 0 to $N - 1$,
   a. Set $z = 1$ and $x = \omega^j$.
   b. For k in 1 to n, $z = z * p_k(x)$.
   c. Set $c = c + z * \omega^{-jT}$
4. Return c/N, rounded to the nearest integer.

The number c returned at the end of the algorithm equals

$$c(T) = \frac{1}{N} \sum_{j=0}^{N-1} \omega^{-jT} p(\omega^j),$$

and thus is the coefficient of x^T in the polynomial p, provided we have carried out our computations to sufficient accuracy.

We will see that it suffices to perform all the calculations with $B = \tilde{O}(n + \log T)$ bits of accuracy. The algorithm keeps all of its storage in 4 variables: j, k, z and c. So, its total space usage is $\tilde{O}(n + \log T)$.

We now work backwards to figure out how much accuracy we need. We need an additive approximation of c to less than $1/2$ absolute accuracy in order to round it to the correct integer. c is the sum of nT numbers. So, if we approximate each of those to absolute accuracy $1/4nT$, then we will approximate c to within $nt/4nT + nTu$ absolute accuracy. Thus, we want $u < 1/4nT$.

But, each term z that goes into the sum to construct c can be big. It is a product of n roots of unity plus 1. Each of these is at most 2, so each term z is at most 2^n. To ensure that we obtain $1/4nT$ absolute accuracy in each of those, we will compute each term in the product to relative accuracy $2^{-n}/8n^2T$. For this purposes, it suffices to have $u < 2^{-n}/8n^2T$ and thus $B \leq \tilde{O}(n + \log(T))$.

Finally, we consider how we compute $p_k(\omega^j)$. As suggested before, we could use the Taylor series to approximate the root of unity ω^{ja_k}. To do this to absolute accuracy $2^{-n}/8n^2T$ will take $\tilde{O}(n+\log T)$ arithmetic operations on numbers with $B \leq \tilde{O}(n+\log(T))$. To bound the total running time of the algorithm, we observe that the outer loop has nT iterations, we compute n terms $p_k(x)$, and each of these computations requires time $\tilde{O}(n + \log(T))$. Thus, the total running time of the algorithm is at most $\tilde{O}(nTn(n + \log T)) \leq \tilde{O}(n^3T)$.

11.8 Modulo a prime

Instead of worrying about numerics, we could perform these computations modulo a prime. The most natural way of doing this would first involve discovering a prime P just a little bit larger than

$nT + 1$. Using randomized algorithms, we can do this with high probability in time polynomial in $\log(nT)$.

We then need to find a *generator* of multiplicative group modulo P to use as ω. This is an element for which $\omega^{P-1} = 1$, but $\omega^k \neq 1$ for $0 < k < P - 1$. This implies that ω^j varies over all non-zero numbers modulo P. Again, there are algorithms for doing this quickly (assuming some standard number theoretic conjectures are true). If we now go through the computation, we will compute $c(T)$ modulo P. This, however, does not necessarily allow us to determine if $c(T)$ is zero or not: it could be divisible by P.

Sun-tzu's Remainder Theorem, formerly known as the Chinese Remainder Theorem, allows us to solve this problem: we just need to repeat the process with primes $P_1, \ldots, P_q$ whose product exceeds $c(T)$.

References

[Hig02] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[JVW21] Ce Jin, Nikhil Vyas, and Ryan Williams. Fast low-space algorithms for subset sum. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1776. SIAM, 2021.

[LN10] Daniel Lokshtanov and Jesper Nederlof. Saving space by algebraization. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 321–330. ACM, 2010.