Design and Analysis of Algorithms

2018A7PS0193P

January 27, 2021

1 Fundamentals

Definition 1.1. *An algorithm is a well defined computational procedure. It takes an input, does some computation and terminates with output*

To check the correctness of an algorithm, we must check the following characteristics:

- Initialization: The algorithm is correct at the beginning
- Maintenance : The algorithm remains correct as it runs
- Termination : The algorithm terminates in finite time, correctly

For this entire course, we must always prove these characteristics when defining any algorithm.

Algorithms are generally defined by a complexity - the time taken to complete the computation on a given input size. There are three ways we could consider this - best case, worst case, or average case.

Complexity is discussed a lot in DSA, so I'm not going to rewrite it here. A quick roundup is:

- $O(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$
- $\Omega(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) : \text{there exists } c, n_0 : 0 \leq c_2 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$

To find complexities in the case of recurrences, we use the **master method**. Let the recurrence be given by:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Here, $a, b \geq 1$. Let ϵ be a constant. Then:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = O(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = O(n^{\log_b a+\epsilon})$ then $T(n) = \Theta(f(n))$ provided if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Here, we redo DSA despite it being a prerequisite of the course. This recap has lasted 3 lectures (so far). You should probably just read CLRS, this is a waste. The topics covered are:

- Quicksort (and it's average case analysis)
- The $\Omega(n \log n)$ lower bound of comparison sorting
- Non-comparison sorting like counting sort, radix sort, etc.
- Average case analysis of bucket sort

2 Matrix Multiplication

Naive Matrix multiplication is $\Theta(N^3)$, since we can express the result $A \cdot B = C$ as:

$$C_{ij} = \sum_{k=1}^{r} A_{ik} \times B_{kj}$$

We can improve this using a divide-and-conquer approach with **Strassen's Multiplication**. It has four steps:

1. Divide the input matrices A and B and the output matrix C into four $n/2 \times n/2$ submatrices. This takes $\Theta(1)$ time.
2. Create 10 matrices $S_1, S_2, ...S_10$, each of which is of size $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1.
3. Using these submatrices, we can recursively compute seven matrix products $P_1, P_2, ...P_7$, each of which is $n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ by adding and subtracting various combinations of the P_i matrices. We can compute all four in $\Theta(N^2)$ time.

The details of this can be seen on page 80 of CLRS, but the running time recurrence will be given by:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

By master method, this is $T(n) = \Theta(n^{\log 7})$

3 Polynomial Multiplication

Polynomials are functions of the form:

$$f(x) = a_0 + a_1 x + a_x^2 + ...a_{n-1}x^{n-1}$$

One way to express this is as a vector of coefficients - this is called the **Coefficient form**. This form also allows us to evaluate $f(x)$ in $O(n)$ using Horner's rule, where we express the polynomial as:

$$a_0 + x(a_1 + x(a_2 + ...x(a_n - 1)...))$$

Another way to express this is using the **point value form** , where we express it as n point of the form $(x_i, f(x_i))$. This point value form uniquely identifies a polynomial. Generally this would take $\Theta(N^2)$ time , but with FFT we can do it in $O(N \log N)$.

The process of getting the coefficient form from the point value form is known as **interpolation**. We can do this in $O(n^3)$ using Gaussian Elimination, or in $O(n^2)$ with Lagrange Interpolation.

Generally, the multiplication of polynomials takes $\Theta(N^2)$. However, we can do this much faster using **Fast Fourier Transform**.

3.1 Fourier Transform

3.1.1 Discrete Fourier Transform

From now on, w_n^k will denote the k^{th} solution of $x^n = 1$, i.e. the n^{th} root of unity. w_n will denote the principal n^{th} root of unity. Remember the following properties:

1. $w_n^k = e^{\frac{2k\pi i}{n}}$
2. $w_{dn}^{dk} = w_n^k$

3. $w_n^{n/2} = w_2 = -1$

4. If $n > 0$ is even, then the squares of the n complex n^{th} roots of unity are the $n/2$ complex $n/2$th roots of unity. (Halving Lemma)

5. $\sum_{j=0}^{n-1} (w_n^k)^j = 0$ (Summation Property)

We call the vector $y = (y_0, y_1, ...y_{n-1})$ the **discrete Fourier Transform** of the polynomial A if $y_k = A(w_n^k)$.

3.1.2 Fast Fourier Transform

FFT consists of three parts:

1. Evaluation, where we find the DFT in $O(n \log n)$

2. Pointwise Multiplication of the two DFTs

3. Interpolation or the inverse FFT, where we find the coefficient form in $O(n \log n)$.

Consider the following polynomials:

$$A(x) = a_0x^0 + a_1x^1 + ... + a_{n-1}x^{n-1}$$
$$A_0(x) = a_0x^0 + a_2x^1 + ... + a_{n-2}x^{n/2-1}$$
$$A_1(x) = a_1x^0 + a_3x^1 + ... + a_{n-1}x^{n/2-1}$$

It is easy to see that:

$$A(x) = A_0(x^2) + xA_1(x^2)$$

These polynomials have only half as many coefficients as the polynomial A. So, if we can compute $DFT(A)$ from $DFT(A_1)$ and $DFT(A_0)$ in linear time, we would be able to do this in $O(n \log n)$ (direct from master method).

We find that do this with the equations:

$$y_k = y_k^0 + w_n^k y_k^1$$
$$y_{k+n/2} = y_k^0 - w_n^k y_k^1$$

Here, $k \in [0, n/2 - 1]$. The proof of this can be seen on cp-algorithms.

As such, we have found the DFT of the polynomial in $O(n \log n)$ time.

After performing the pointwise multiplication of the DFT of our polynomials A and B, we have to interpolate to find the coefficient form of our resulting polynomial C.

TODO: Add interpolation with Vandermonde matrix, can be seen on cp-algorithms