1 Divide and Conquer

Divide and Conquer algorithms work by recursively breaking the problem into smaller pieces and solving the subproblems. To use this problem solving paradigm, you need to think about how to recursively divide a problem into smaller-sized problems, conquer the smaller-sized problems, and then finally combine the solutions obtained from the previous stage into one single solution that solves the original problem.

Examples from Lecture:

- **Mergesort**: We take a list, divide it into two smaller sublists, conquer each sublist by recursively sorting, and then combine the two solutions into a single solution.

- **Karatsuba’s Algorithm for Integer Multiplication**: We split two integers in half each, forming 4 pieces and then performed only 3 multiplications on half-sized numbers.

- **Strassen’s Algorithm**: We perform matrix multiplication on two $n \times n$ matrices by performing 7 multiplications on $n/2 \times n/2$ matrices.

- **Median Finding**

- **Finding the Maximum/Minimum Element in a List**

Exercise 1. Given an array of $n$ elements, you want to determine whether there exists a majority element (that is an element which occurs at least $\lceil \frac{n+1}{2} \rceil$ times) and if so, output this element. Show how to do this in time $O(n \log n)$ using Divide and Conquer.

**Solution**

Split the list into 2 halves and recursively find the majority element of each half. If both sides return “no majority”, then return “no majority”. This is true because if there exists a majority element, then it also has to be the majority on at least one of the two. If both sides return an element, check to see if the elements are the same and if so return it. Otherwise, for each element returned by the recursive call, do a linear time traversal through the entire list to see if it is the majority element. Return it if it is and “no majority” if you don’t detect a majority element this way.

It is not hard to convince yourself that this algorithm is correct, but what is its run-time? We make 2 recursive calls on lists of size $n/2$ and then perform up to 2 linear traversals through the list. Thus:

$$T(n) = 2T(n/2) + O(n)$$

which solves to $O(n \log n)$ just like mergesort.

Exercise 2. (2015 Problem Set 4) You are given an $n$-digit positive integer $x$ written in base-2. Give an efficient algorithm to return its representation in base-10 and analyze its running time. Assume you have black-box access to an integer multiplication algorithm which can multiply two $n$-digit numbers in time $M(n)$ for some $M(n)$ which is $\Omega(n)$ and $O(n^2)$.
Solution
Consider the $n$-digit base-2 integer $x$ as a string of base-2 digits, $x[0..n]$. Assume $n$ is a power of 2. If not, we can pad the left side of the input integer with zeros. This will increase the number of digits by less than a factor of 2, which will not affect the big-O runtime.

We offer a recursive algorithm. In the base case, for an input of size 0, output 0. Divide the integer into two halves, $x[0...\frac{n}{2}]$ and $x[\frac{n}{2}...n]$. Then, recursively convert each half into a base-10 integer. Call these (converted) base-10 integers $y_l$ and $y_r$. Then, the desired integer can be found by computing $y_l \cdot 2^n/2 + y_r$. Note that, $p = 2^{n/2}$ must be in base-10 also. One way we could get the base-10 representation of $p$ is by converting it recursively, too, along with the halves of the input integer. Then we make three recursive calls to our procedure, each with a base-2 integer which is roughly half the size of the problem input. After we obtain these base-10 integers, we have to perform an integer multiplication ($y_l \cdot p$) and an integer addition ($y_l \cdot p + y_r$).

The run-time is:

$$T(n) = 3T(n/2) + M(n/2) + O(n)$$

Suppose $M(n) = O(n^k)$ for some $1 \leq k \leq 2$. Then, $M(n/2) + O(n) = O(M(n/2)) = O(n^k)$. Thus, we can apply the master theorem on what $k$ is:

If $k > \log_2 3$, then $T(n) = O(n^k)$. If $k = \log_2 3$, then $T(n) = O(n^k \log n)$ and if $k < \log_2 3$, then $T(n) = O(n^{\log_2 3})$.

2 Fast Fourier Transform

We can multiply polynomials (more efficiently than the naive method) using the Fast Fourier Transform (FFT).

Suppose we have two polynomials

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_dx^d$$

$$B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + \cdots + b_dx^d$$

The goal is to compute the product

$$C(x) = A(x) \cdot B(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$$

with coefficients $c_k = \sum_{i=0}^{k} a_ib_{k-i}$ (for $i > d$, let $a_i = b_i = 0$).

Finding all $2d+1$ coefficients would require $\Theta(d^2)$ time for degree $d$ polynomials $A(x)$ and $B(x)$. By the interpolation theorem, we know that a degree-$d$ polynomial is uniquely determined by its values at any $d+1$ distinct points. Thus, one way to determine $C(x) = A(x) \cdot B(x)$ is to pick $n \geq 2d+1$ points $x_0, x_1, \ldots, x_{n-1}$. Evaluate the polynomials $A(x)$ and $B(x)$ at these points by computing $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$. Finally, compute $C(x_k) = A(x_k)B(x_k)$ for $k = 0, \ldots, n-1$ and using interpolation, we can recover $C(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$. But this still takes $\Theta(n^2)$ time. However, the method of determining a polynomial via interpolation is the core idea of FFT. We will pick the $n$ points in a specific way. The points are $1, \omega, \omega^2, \ldots, \omega^{n-1}$ where $\omega = e^{2\pi i/n}$ ($i = \sqrt{-1}$) is a “primitive $n$th root of unity”.

2
Then a divide and conquer strategy lets us compute the evaluations $A(1), A(\omega), \ldots, A(\omega^{n-1})$ in $O(n \log n)$ time.

Applications of the FFT

- **Integer Multiplication** Suppose we have two integers $a$ and $b$. We can treat each integer as a polynomial evaluated at $x = 10$. For example, if $a = 6631$ and $b = 2337$, then we can define $A(x) = 1 + 3x + 6x^2 + 6x^3$ and $B(x) = 7 + 3x + 3x^2 + 2x^3$. Then $a \cdot b = c = (A \cdot B)(10)$ (that is, the polynomial $A(x) \cdot B(x)$ evaluation at $x = 10$). Thus, we can use the FFT to compute $a \cdot b$ by computing $A \cdot B$ and evaluating the resulting polynomial at $x = 10$.

- **Pattern Matching** Suppose we have binary strings $P = p_0p_1\cdots p_{m-1}$ and $T = t_0t_1\cdots t_{n-1}$ for $n \geq m$ with $p_i, t_i \in \{0, 1\}$. The goal is to report all indices $i$ such that $P$ starting at position $i$ in $T$ matches the smaller string $P$. Formally, we want to determine $i$ for which $T[i, i+1, \ldots, i+m-1] = P$. The naive approach of trying all possible starting positions $i \in \{1, \ldots, n - m + 1\}$ takes time $\Theta(nm)$. But with the FFT we can solve this problem in $O(n \log n)$ time.

**Exercise 3.** Let $A(x) = 3 + 2x + 3x^2 + 4x^3$.

(a) Write $A(x)$ as $A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$

(b) Compute $A(\omega^0), A(\omega)^0, \ldots, A(\omega^{n-1})$

**Solution**

(a) $A_{\text{even}}(x) = 3 + 3x$

$A_{\text{odd}}(x) = 2 + 4x$

$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2) = (3 + 3x^2) + x(2 + 4x^2)$

(b) $A(\omega^0) = A(1) = 3 + 2 + 3 + 4 = 12$

$A(\omega^1) = A(i) = 3 + 2i - 3 - 4i = -2i$

$A(\omega^2) = A(-1) = 0$

$A(\omega^3) = A(-i) = 2i$

**Exercise 4. Computing FFTs**

(a) What is the FFT of a polynomial with coefficients $(1, 0, 0, 0)$?

(b) What is the FFT of a polynomial with coefficients $(1, 0, 1, -1)$?

**Solution**

Since we are given 4 coefficients (for both parts), $\omega = \cos \frac{2\pi}{4} + i \sin \frac{2\pi}{4} = i$
Exercise 5. $n$th roots of unity

(a) What is the sum of the $n$th roots of unity?

(b) What is the product of the $n$th roots of unity? What if $n$ is odd? Is it different for even $n$?

Solution

By Euler’s formula

$$e^{ix} = \cos(x) + i \sin(x).$$

Then the $n$th roots of unity are $\cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n}$, $k = 0, \ldots, n - 1$.

(a) If $n = 1$, $\omega^0 = \cos(\frac{2\pi \cdot 0}{1}) + i \sin(\frac{2\pi \cdot 0}{1}) = 1$. For $n > 1$,

$$\sum_{k=0}^{n-1} \omega^k = \frac{\omega^n - 1}{\omega - 1}$$

where we have used the geometric series formula.
\( \prod_{k=0}^{n-1} \omega^k = \prod_{k=0}^{n-1} \left( \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n} \right) \)  \( (5) \)

\[
= \cos \left( \sum_{k=0}^{n-1} \frac{2\pi k}{n} \right) + i \sin \left( \sum_{k=0}^{n-1} \frac{2\pi k}{n} \right) \\
= \cos \left( \frac{2\pi \sum_{k=0}^{n-1} k}{n} \right) + i \sin \left( \frac{2\pi \sum_{k=0}^{n-1} k}{n} \right) \\
= \cos \left( \frac{2\pi (n-1)n}{2n} \right) + i \sin \left( \frac{2\pi (n-1)n}{2n} \right) \\
= \cos(\pi(n-1)) + i \sin(\pi(n-1)) \]  \( (6) \)

where we have used that \( \prod_{k=0}^{n-1} e^{2\pi ik/n} = e^{\Sigma_{k=0}^{n-1} 2\pi ik/n} \).

When \( n \) is even \( \prod_{k=0}^{n-1} \omega^k = -1 \) and when \( n \) is odd \( \prod_{k=0}^{n-1} \omega^k = 1 \).