

CSE 317: Design and Analysis of Algorithms

Shahid Hussain

Week 5: September 16, 18: Fall 2024

Divide and Conquer Algorithms

Maximum Subarray Sum

- For a given array $A[1..n]$ of n numbers, the *maximum subarray sum*, denoted as \mathcal{M} is the largest sum of contiguous subarray of $A[i..j]$ where $1 \leq i \leq j \leq n$, i.e.,

$$\mathcal{M}(A) = \sum_{x=i}^j A[x]$$

- For example, for $A = [-2, 3, 4, -1, 5, -3, 2, -1]$, $\mathcal{M}(A) = 11$ for the subarray $[3, 4, -1, 5]$

Maximum Subarray Sum: Trivial Solution

Algorithm: BRUTEFORCE-MAX-SUBARRAY-SUM

Input: An array $A[1..n]$ of n numbers

Output: The maximum subarray sum $\mathcal{M}(A)$

1. $\mathcal{M} = -\infty$
2. **for** $i = 1$ **to** n
3. **for** $j = i$ **to** n
4. $s = 0$
5. **for** $k = i$ **to** j
6. $s = s + A[k]$
7. **if** $s > \mathcal{M}$ **then** $\mathcal{M} = s$
8. **return** \mathcal{M}

The time complexity of above algorithms is $\Theta(n^3)$

Maximum Subarray Sum

- Let us design a divide and conquer algorithm
- We divide the array into two almost equal parts and find the maximum subarray sum in both these parts recursively
- At this point we have three cases:
 1. The maximum subarray sum is in the left part
 2. The maximum subarray sum is in the right part
 3. The maximum subarray sum crosses the middle point
- The algorithm returns the maximum of these three cases
- The crossover sum can be computed in $O(n)$ time by starting from the middle point and going left and right to find the maximum sum
- If $T(n)$ represents the time complexity of the algorithm then we have the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Maximum Subarray Sum: Maximum Crossing Sum

Input: Array $A[l..h]$ of numbers

Output: The max. subarray sum

1. **if** $l = h$ **then return** $A[l]$
2. $m = \lfloor (l + h)/2 \rfloor$
3. $\mathcal{M}_L = \text{DC-MAX-SUBARRAY-SUM}(A[l..m])$
4. $\mathcal{M}_L = \text{DC-MAX-SUBARRAY-SUM}(A[m + 1..h])$
5. $\mathcal{M}_L = \text{MAX-CROSSING-SUM}(A[l..h])$
6. **return** $\max\{\mathcal{M}_L, \mathcal{M}_R, \mathcal{M}_C\}$

Maximum Subarray Sum: Divide and Conquer

Algorithm: MAX-CROSSING-SUM

Input: An array $A[l..h]$ of numbers

Output: The max crossing sum

1. $m = \lfloor (l + h)/2 \rfloor$, $s = 0$, $left_sum = -\infty$
2. **for** $i = m$ **downto** l
3. $s = s + A[i]$
4. **if** $s > left_sum$ **then** $left_sum = s$
5. $s = 0$, $right_sum = -\infty$
6. **for** $i = m + 1$ **to** h
7. $s = s + A[i]$
8. **if** $s > right_sum$ **then** $right_sum = s$
9. **return** $left_sum + right_sum$

Closest-Pair Problem

- Given a set of n points

$$P = \{p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)\}$$

- The *closest-pair* problem asks to find the pair of points p_i and p_j that are closest to each other i.e., $d(p_i, p_j)$ is minimum, where

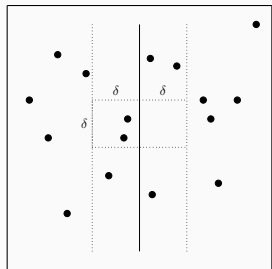
$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- A brute force algorithm will find the distance between all pairs of points and return the minimum distance
- This algorithm has a time complexity of $O(n^2)$ as there are $n(n-1)/2$ different pairs of points
- We will see a divide and conquer algorithm that solves this problem in $O(n \log n)$ time

Closest-Pair Problem: Divide and Conquer

- The first step is to sort the points according to their x -coordinates
- We divide the points into two almost equal parts P_L and P_R and find the closest pair in the left and right parts recursively
- Let l be the vertical line that separates P_L and P_R
- Let δ_L and δ_R be the minimum distance between two points in P_L and P_R , respectively
- Let δ' be the minimum distance between a point in P_L and a point in P_R
- The closest pair is either in P_L or P_R or it crosses the line l , therefore the closest pair is the minimum of δ_L , δ_R and δ'

Closest-Pair Problem: Divide and Conquer



- We set $\delta = \min\{\delta_L, \delta_R\}$
- If there is a pair of points $p_i \in P_L$ and $p_j \in P_R$ such that $d(p_i, p_j) < \delta$ then this is the closest pair
- If the closest pair consists of some point in P_L and P_R then we need to check only those points that are within δ distance from the line l (both sides)
- We can now run a linear scan to find the closest pair in this region from top to bottom (see the figure)

Closest-Pair Problem: Divide and Conquer

- Let $T(n)$ be the time complexity of the algorithm
- We have the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

Fast Fourier Transform

Multiplying polynomials

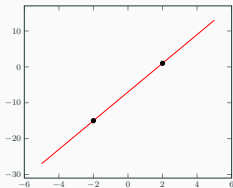
- Let $A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + \cdots + b_{n-1}x^{n-1}$ be two $n - 1$ degree polynomials
- Then the product: $C(x) = A(x) \cdot B(x)$ is a $2n - n$ degree polynomial: defined as

$$C(x) = c_0 + c_1x + \cdots + c_{2n-2}x^{2n-2}$$

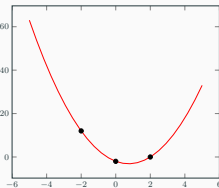
- The polynomial $C(x)$ can be computed in $O(n^2)$ time
- For example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4$$

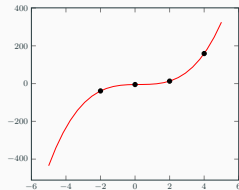
Representating Polynomials



$$y = 4x - 7$$



$$y = 2x^2 - 3x - 2$$



$$y = 3x^3 - 2x^2 + x - 5$$

- We need $n + 1$ distinct points to determine any n degree polynomial
- We can specify any n degree polynomial $A(x) = a_0 + a_1x + \cdots + a_nx^n$ by either one of the following:
 - Its coefficients: a_0, a_1, \dots, a_n
 - The values: $A(x_0), A(x_1), \dots, A(x_n)$

Polynomial Multiplication

- For two n degree polynomials $A(x)$ and $B(x)$ represented using their coefficients a_0, \dots, a_n and b_0, \dots, b_n
- The polynomial $C(x) = A(x) \cdot B(x)$ can easily be computed using $A(x)$ and $B(x)$ as following:

Polynomial Multiplication

- For two n degree polynomials $A(x)$ and $B(x)$ represented using their coefficients a_0, \dots, a_n and b_0, \dots, b_n
- The polynomial $C(x) = A(x) \cdot B(x)$ can easily be computed using $A(x)$ and $B(x)$ as following:

Algorithm: POLYNOMIALMULTIPLICATION (FFT)

Input: Coefficients of $A(x)$ and $B(x)$

Output: The product $C(x) = A(x) \cdot B(x)$

1. Pick some $k \geq 2n + 1$ points x_0, \dots, x_{k-1}
2. Compute $A(x_0), \dots, A(x_{k-1})$ and $B(x_0), \dots, B(x_{k-1})$
3. Compute $C(x_j) = A(x_j) \cdot B(x_j)$ for all $j = 0, \dots, k - 1$
4. Recover $C(x) = c_0 + c_1x + \dots + c_{2n}x^{2n}$

Polynomial Multiplication

- How to pick these points? randomly?
- One idea is to use positive-negative pairs i.e.,

$$\pm x_0, \pm x_1, \dots, \pm x_{n/2-1}$$

This will allow us to reuse some parts of computation e.g., $A(x_i)$ and $A(-x_i)$ will be same for all even powers of x_i

- For example: let $A(x)$ be

$$3+4x+6x^2+2x^3+x^4+10x^5 = (3+6x^2+x^4)+x(4+2x^2+10x^4)$$

- We see that:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

where $A_e(\cdot)$ and $A_o(\cdot)$ are polynomials of degree $\leq n/2 - 1$ of even- and odd-numbered coefficients, respectively

Polynomial Multiplication

- Assuming n is even and the \pm pairs of points $\pm x_i$

$$\begin{aligned}A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2)\end{aligned}$$

- So, evaluating $A(x)$ at n paired points $\pm x_0, \dots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points $x_0^2, \dots, x_{n/2-1}^2$
- Therefore, applying it recursively, the time complexity is now as $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Polynomial Multiplication

- Assuming n is even and the \pm pairs of points $\pm x_i$

$$\begin{aligned}A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2)\end{aligned}$$

- So, evaluating $A(x)$ at n paired points $\pm x_0, \dots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points $x_0^2, \dots, x_{n/2-1}^2$
- Therefore, applying it recursively, the time complexity is now as $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- Unfortunately, this only works for the top-level recursion, we need $n/2$ evaluation points $x_0^2, \dots, x_{n/2-1}^2$ to be themselves plus-minus pairs, which is impossible, unless

Polynomial Multiplication

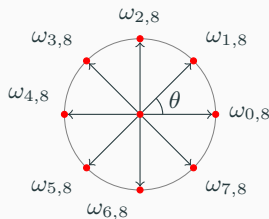
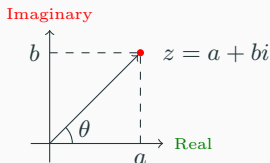
- Assuming n is even and the \pm pairs of points $\pm x_i$

$$\begin{aligned}A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2)\end{aligned}$$

- So, evaluating $A(x)$ at n paired points $\pm x_0, \dots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points $x_0^2, \dots, x_{n/2-1}^2$
- Therefore, applying it recursively, the time complexity is now as $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- Unfortunately, this only works for the top-level recursion, we need $n/2$ evaluation points $x_0^2, \dots, x_{n/2-1}^2$ to be themselves plus-minus pairs, which is impossible, unless
- We use *complex numbers*

Complex Numbers and Roots of Unity

- Let $z = a + bi$ a complex number
- Let $x^8 = 1$ gives us 8 roots of unity, namely $w_{0,1}, \dots, w_{7,8}$ ($w_{0,8} = 1$)



- For $x^n = 1$ we get $w_{0,n} = 1, w_{1,n}, w_{2,n}, \dots, w_{n-1,n}$
- Here $w_{i,n} = e^{2\pi i/n}$ for $i = 0, \dots, n-1$

Fast Fourier Transform

- Let $n = 2^k$ for some $k > 0$
- We represent:

$$\begin{aligned}A(x) &= A_e(x^2) + xA_o(x^2) \\ B(x) &= B_e(x^2) + xB_o(x^2)\end{aligned}$$

- We compute $A_e(x^2)$, $A_o(x^2)$, $B_e(x^2)$ and $B_o(x^2)$ at:

$$(\omega_{j,2n})^2 \quad \text{for } j = 0, \dots, 2n - 1$$

- Note that: $(\omega_{j,2n})^2 = e^{2\pi ji/n} = e^{2\pi jli/n}$ where $l = j \bmod n$.

Fast Fourier Transform

- Let $L(n)$ be the number of operation of multiplication and addition required to evaluate a polynomial A of degree $n - 1$ on $(2n)$ -th roots of unity (Steps 1. and 2. of FFT algorithm)
- Then: $L(n) = 2L(n/2) + 4n = O(n \log n)$ for some $n = 2^k$
- Let us consider the Step 3. of FFT algorithm to recover the coefficients of $C(x)$, we know:

$$C(x) = \sum_{m=0}^{2n-1} c_m x^m$$

- From values:

$$d_0 = C(\omega_{0,2n}), \dots, d_{2n-1} = C(\omega_{2n-1,2n})$$

Fast Fourier Transform

- Let us define a new polynomial $D(x)$ as:

$$D(x) = \sum_{m=0}^{n-1} d_m x^m$$

- The value of $D(x)$ at $x = \omega_{j,2n}$ is:

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{m=0}^{n-1} d_m (\omega_{j,2n})^m = \sum_{m=0}^{n-1} C(\omega_{j,2n}) (\omega_{j,2n})^m \\ &= \sum_{m=0}^{n-1} \left(\sum_{l=0}^{2n-1} c_l (\omega_{m,2n})^l \right) (\omega_{j,2n})^m \\ &= \sum_{l=0}^{2n-1} c_l \left(\sum_{m=0}^{n-1} (\omega_{j,2n})^m (\omega_{m,2n})^l \right) \end{aligned}$$

Fast Fourier Transform

- We know $\omega_{j,2n} = (e^{2\pi ji/2n})^2$
- Therefore,

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{l=0}^{2n-1} c_l \left(\sum_{m=0}^{2n-1} e^{2\pi i(lm+jl)/2n} \right) \\ &= \sum_{l=0}^{2n-1} c_l \left(\sum_{m=0}^{2n-1} (\omega_{l+j,2n})^m \right) \end{aligned}$$

- Let us consider the sum:

$$\sum_{m=0}^{2n-1} (\omega_{l+j,2n})^m = \begin{cases} 2n & \text{if } l+j = 0 \pmod{2n} \\ 0 & \text{otherwise} \end{cases}$$

- Therefore, $D(\omega_{j,2n}) = 2nc_{2n-j}$ and $c_{2n-j} = D(\omega_{j,2n})/2n$

Fast Fourier Transform

- So, if we evaluate D and $(2n)$ -th roots of unity:

$$\omega_{2n,2n} = \omega_0, \omega_{2n-1,2n}, \dots, \omega_{1,2n}$$

- We get: $c_0, c_1, \dots, c_{2n-1}$
- These evaluations can be computed in $O(n \log n)$ operations of multiplications and additions using divide-and-conquer approach developed for the Step 1. of **FFT**
- Thus, **FFT** requires $O(n \log n)$ operations

Fast Fourier Transform

Algorithm: RECURSIVE-FFT

Input: Coefficients of $A(x) = \langle a_0, a_1, \dots, a_{n-1} \rangle$

Output: The discrete FFT of $A(x)$

1. **if** $n = 1$ **then return** a_0
2. $\omega_n = e^{2\pi i/n}$, $\omega = 1$
3. $A_e = \langle a_0, a_2, \dots, a_{n-2} \rangle$, $A_o = \langle a_1, a_3, \dots, a_{n-1} \rangle$
4. $Y_e = \text{RECURSIVE-FFT}(A_e)$, $Y_o = \text{RECURSIVE-FFT}(A_o)$
5. **for** $k = 0$ **to** $n/2 - 1$
6. $y_k = Y_e[k] + \omega Y_o[k]$
7. $y_{k+n/2} = Y_e[k] - \omega Y_o[k]$
8. $\omega = \omega \cdot \omega_n$
9. **return** $Y = \langle y_0, y_1, \dots, y_{n-1} \rangle$

Fast Fourier Transform

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_{1,n} & \omega_{2,n} & \cdots & \omega_{n-1,n} \\ 1 & \omega_{2,n} & \omega_{4,n} & \cdots & \omega_{2(n-1),n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1,n} & \omega_{2(n-1),n} & \cdots & \omega_{(n-1)(n-1),n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$