EECS 336

Litong Wang

Homework 3

April 25, 2016

Problem 1

Describe the generalization of the FFT procedure to the case in which n is a power of 5. Give a recurrence for the running time, and solve the recurrence.

Solution:

Partition $A(x)$ to define five new polynomials $A^{[0]}(x)$, $A^{[1]}(x)$, $A^{[2]}(x)$, $A^{[3]}(x)$ and $A^{[4]}(x)$ of degree-bound $\frac{n}{5}$:

$$\begin{aligned}
A^{[0]}(x) &= a_0 + a_5 x + \cdots + a_{n-5} x^{\frac{n}{5}-1} \\
A^{[1]}(x) &= a_1 + a_6 x + \cdots + a_{n-4} x^{\frac{n}{5}-1} \\
A^{[2]}(x) &= a_2 + a_7 x + \cdots + a_{n-3} x^{\frac{n}{5}-1} \\
A^{[3]}(x) &= a_3 + a_8 x + \cdots + a_{n-2} x^{\frac{n}{5}-1} \\
A^{[4]}(x) &= a_4 + a_9 x + \cdots + a_{n-1} x^{\frac{n}{5}-1} \\
A(x) &= A^{[0]}(x^5) + xA^{[1]}(x^5) + x^2 A^{[2]}(x^5) + x^3 A^{[3]}(x^5) + x^4 A^{[4]}(x^5)
\end{aligned}$$

Recursively evaluate $A^{[0]}(x), A^{[1]}(x), A^{[2]}(x), A^{[3]}(x), A^{[4]}(x)$ at the $\frac{n}{5}$-th roots of unity: $v^0, v^1, \cdots, v^{\frac{n}{5}-1}, v^k = (\omega^5)^k$.

Evaluate A(x) at the n-th roots of unity.

$$\begin{aligned}
&A(\omega^k) = A^{[0]}(v^k) + \omega^k A^{[1]}(v^k) + \omega^{2k} A^{[2]}(v^k) + \omega^{3k} A^{[3]}(v^k) + \omega^{4k} A^{[4]}(v^k) \\
&A(\omega^{k+\frac{n}{5}}) = A^{[0]}(v^k) + \omega^{k+\frac{n}{5}} A^{[1]}(v^k) + \omega^{2(k+\frac{n}{5})} A^{[2]}(v^k) + \omega^{3(k+\frac{n}{5})} A^{[3]}(v^k) + \omega^{4(k+\frac{n}{5})} A^{[4]}(v^k) \\
&A(\omega^{k+\frac{2n}{5}}) = A^{[0]}(v^k) + \omega^{k+\frac{2n}{5}} A^{[1]}(v^k) + \omega^{2(k+\frac{2n}{5})} A^{[2]}(v^k) + \omega^{3(k+\frac{2n}{5})} A^{[3]}(v^k) + \omega^{4(k+\frac{2n}{5})} A^{[4]}(v^k) \\
&A(\omega^{k+\frac{3n}{5}}) = A^{[0]}(v^k) + \omega^{k+\frac{3n}{5}} A^{[1]}(v^k) + \omega^{2(k+\frac{3n}{5})} A^{[2]}(v^k) + \omega^{3(k+\frac{3n}{5})} A^{[3]}(v^k) + \omega^{4(k+\frac{3n}{5})} A^{[4]}(v^k) \\
&A(\omega^{k+\frac{4n}{5}}) = A^{[0]}(v^k) + \omega^{k+\frac{4n}{5}} A^{[1]}(v^k) + \omega^{2(k+\frac{4n}{5})} A^{[2]}(v^k) + \omega^{3(k+\frac{4n}{5})} A^{[3]}(v^k) + \omega^{4(k+\frac{4n}{5})} A^{[4]}(v^k) \\
&0 \le k < \frac{n}{5}
\end{aligned}$$

Runtime complexity:

$$\begin{aligned}
T(n) &= 5T(\frac{n}{5}) + \Theta(n) \\
&= \Theta(n \lg n).
\end{aligned}$$

Problem 2

Consider three sets X, Y and Z, each having n integers in the range from 0 to 10n. We wish to compute the Cartesian sum of X, Y and Z, defined by $C = \{x+y+z : x \in X, y \in Y, z \in Z\}$ Note that the integers in C are in the range from 0 to 30n. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in X, Y and Z. Show how to solve the problem in $\mathcal{O}(n \lg n)$ time. (Hint: Represent X, Y and Z as polynomials of degree at most 10n.)

Solution:

We can define $X(x) = \sum_{i=0}^{n} x^{a_i}, Y(x) = \sum_{i=0}^{n} x^{b_i}$ *and* $Z(x) = \sum_{i=0}^{n} x^{c_i}$ *where* $a_i \in X, b_i \in Y, c_i \in Z$. We can compute $C(x) = X(x) \cdot Y(x) \cdot Z(x)$ using FFT in $\mathcal{O}(n \lg n)$ time.

$$\begin{aligned} C(x) &= X(x) \cdot Y(x) \cdot Z(x) \\ &= \sum_{i=0}^{n} d_i x^{a_i+b_i+c_i} \end{aligned}$$

We can see that $C_i = (a_i + b_i + c_i) \in C, C = \{x + y + z : x \in X, y \in Y, z \in Z\}$ and d_i is the number of times of C_i appearing in C.

Problem 3

Derive a point-value representation for $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_j x^j$ assuming that none of the points is 0.

Solution:

$$\begin{aligned} \frac{A(x)}{x^{n-1}} &= \sum_{j=0}^{n-1} a_j \frac{x^j}{x^{n-1}} \\ &= \sum_{j=0}^{n-1} a_j \frac{1}{x^{n-1-j}} \\ &= \sum_{i=0}^{n-1} a_{n-1-i} \frac{1}{x^i} \\ &= A^{rev}(\frac{1}{x}) \end{aligned}$$

$\therefore$ we can get a point-value representation for $A^{rev}(x)$ $\{x'_i, y'_i\}$ from a point-value representation for $A(x)$ such that $x'_i = x_i, y'_i = \frac{y_i}{x_i^{n-1}}$

Problem 4

Show that the solution to $T(n) = 2T(\lfloor \frac{n}{2} \rfloor + \lceil \log n \rceil) + n$ is $\Theta(n \lg n)$

Solution:

$$
\begin{aligned}
&g(n) = \lfloor \frac{n}{2} \rfloor \\
&\frac{n}{2} - 1 < g(n) \leq \frac{n}{2} \\
&\therefore |g(n)| \in \mathcal{O}(n^c) \\
&h(n) = \lceil \log n \rceil \\
&\log n \leq h(n) < \log n + 1 \\
&\because \lim_{n\to\infty} \frac{\lg(\log n)^3}{\lg n} = \lim_{n\to\infty} \frac{\log n}{\frac{n}{(\log n)^2}} = \lim_{n\to\infty} \frac{3 \lg \log n}{\lg n} = 0 \\
&\therefore h(n) \in |\frac{n}{(\log n)^2}| \\
&\because 0 < a, 0 < b < 1 \\
&We\ can\ apply\ Akra - Bazzi\ Master\ Theorem \\
&\because P = 1, a(b)^P = 1 \\
&T(n) = \Theta(n(1 + \int_1^n \frac{u}{u^2} du)) \\
&T(n) = \Theta(n(1 + \int_1^n d \log u)) \\
&T(n) = \Theta(n \log n)
\end{aligned}
$$

Problem 5

What is the largest k such that if you can multiply 5×5 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

Solution:

We can multiply $n \times n$ matrices by multiplying $\frac{n}{5} \times \frac{n}{5}$ matrices recursively. Hence, we have: $T(n) = kT(\frac{n}{5}) + \Theta(n^2)$.

Apply master method:

$$
\begin{aligned}
& a = k, b = \frac{1}{5}, \alpha = 2, \beta = 0 \\
& a(b)^x = 1 \\
& if\ x < \alpha, case\ 1: T(n) = \Theta(n^2) \\
& T(n) = o(n^{\lg 7}) \\
& \therefore a \cdot \frac{1}{25} < 1 \\
& \therefore k = a < 25 \\
& if\ x = \alpha, case\ 2: T(n) = \Theta(n^2(\log n)) \\
& T(n) = o(n^{\lg 7}) \\
& \therefore a \cdot \frac{1}{25} = 1 \\
& \therefore k = a = 25 \\
& if\ x > \alpha, case\ 3: T(n) = \Theta(n^x) \\
& \because T(n) = o(n^{\lg 7}) \\
& \therefore x < \lg 7 \\
& \because a = \frac{1}{b^x} \\
& \therefore a \leq 5^{\lg 7}
\end{aligned}
$$

The largest integer such that smaller than $5^{\lg 7}$ is 91
The running time is $\Theta(n^{\log_5 91})$

Problem 6

Assume you have an array X[1..n] of n elements. A majority element of X is defined to be an element occurring in more than $\frac{n}{2}$ positions (e.g., if $n = 6$ or $n = 7$, a majority element will occur in at least 4 positions). Assume that elements cannot be ordered or sorted, but can be compared for equality. (You might think of the elements as chips, and there is a tester that can be used to determine whether or not two chips are identical.)

a) Design an algorithm to find a majority element in X or determine that no majority element exists. The time complexity of your algorithm should be $\mathcal{O}(n \log n)$.

b) Design an algorithm to find a majority element in X or determine that no majority element

exists. The time complexity of your algorithm should be $\mathcal{O}(n)$.
Prove the correctness and time complexity of each of your algorithms.
Solution:

a) Split the array A into two arrays $A_1, A_,2$ with half size of A, we look for the majority elements in both arrays. If the two majority elements in two subarrays are same, then we can just return this element. If they are not, we need to compute the frequency of the majority elements. This operation takes $\mathcal{O}(n)$ time. Then we can decide which is the majority element or the majority element doesn't exist.

Algorithm: Majority element1

Data: $A[1, \cdots, n]$
Result: The majority element
if $n = 1$ **then**
  $returnA[1]$;
end
$mid = \frac{n}{2}$;
$left = Majority\ element1(A[1, mid])$;
$right = Majority\ element1(A[mid + 1, n])$;
if $left = right$ **then**
  $return\ left$;
else
  $f_{left} = getFrequency(left, A[1, n])$;
  $f_{right} = getFrequency(right, A[1, n])$;
end
if $f_{left} > \frac{n}{2}$ **then**
  $return\ left$;
end
if $f_{right} > \frac{n}{2}$ **then**
  $return\ right$;
end
$return\ None$;

Correctness:

Proof.
1) If A has a majority element e, then e must be the majority element of A_1 *or* A_2 *or both.*
2) Base case: $n = 1$ return A[1], the algorithm is correct.
3) Induction hypothesis:
In step k, assume that for input size is n. If A has a majority element e, then e must be the

majority element of A_1 *or* A_2 *or both*.
In next step, the input size is 2n, we have three scenarios:
Majority element A_1 of A and majority element B_1 of B.
a) $A_1 = B_1$, $frequency(A_1) > \frac{n}{2}$ *and* $frequency(B_1) > \frac{n}{2}$ then $frequency(A_1)+frequency(B_1) > n$, we can get the majority element.
b) $A_1 \neq B_1$, then according to the algorithm, we should check the $frequency(A_1)$ *and* $frequency(B_1)$. If frequency of each element is greater than n, then we find the majority element or there is no majority element.
c) A_1 *and* B_1 don't exist, then there is no majority element. 4) We reduce the search range by half in each iteration, then the algorthm will always terminate.
Time complexity:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \mathcal{O}(n) \\ &= \mathcal{O}(n \log n) \end{aligned}$$

Solution:

b) We use a hashtable to store pair $(element, frequency)$. We go through the whole array, if element is already in hashtable, we update the frequency($frequency++$) or we insert a new pair $(element, frequency)$, $frequency = 1$ into the hashtable. We also create a variable count to store and update the highest frequency. If after looking through the whole array, $count \leq \frac{n}{2}$, then there is no majority element.
Algorithm: Majority element2

Data: $A[1, \cdots, n]$
Result: The majority element
if $n = 1$ **then**
    $returnA[1]$;
end
Create a hashtable: hash ;
count $= 0$;
while $i \leq n$ **do**
    if $hash.contains(A[i])$ **then**
        $hash.put(A[i], hash.get(A[i] + 1))$;
        $count = max(count, hash.get(A[i] + 1))$;
    else
        $hash.put(A[i], 1)$;
    end
    $i + +$;
    if $count > \frac{n}{2}$ **then**
        $return\ A[i]$;
    end
end
$return\ None$;

Correctness:

We keep track of the frequency of each element. If the majority element exist, by definition, its $frequency > \frac{n}{2}$. If there is no element which has frequency higher than $\frac{n}{2}$, then the majority element doesn't exist.

Time complexity:

Because the runtime of update the hashtable is $\mathcal{O}(1)$ and we only need to do n iteration to search through the whole array. So the time complexity of this algorithm is $\mathcal{O}(n)$.