CS 170 HW 3

Due on 2019-02-11, at 10:00 pm

1 Study Group

List the names and SIDs of the members in your study group.

2 Maximum Subarray Sum

Given an array of n integers, the maximum subarray is the contiguous subarray (potentially empty) with the largest sum. Design an $\Theta(n \log n)$ algorithm to find the sum of the maximum subarray. For example the maximum subarray of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is $[4, -1, 2, 1]$, whose sum is 6.

(*Extra credit:* Try to design a $\Theta(n)$ solution)

Please give a three-part solution of the following format:

(a) **Clearly** describe your algorithm. You can include the pseudocode optionally.

(b) Write a proof of correctness.

(c) Write a runtime analysis.

Solution:

(a) **Description of the algorithm**

The DQ approach will only give an $O(n \log n)$ algorithm. A linear algorithm can be: scan the array and record the sum. If sum becomes negative then set it to 0. Then the maximum subarray sum is the maximum value that sum has ever been during the scan.

Pseudocode

```
Set $maxSum := 0$, $sum := 0$
for $i$ in 1:$n$ do
    $sum = \max(0, sum + a[i])$
    $maxSum = \max(maxSum, sum)$
return $maxSum$
```

(b) **Proof of correctness.**

If all the numbers in the array are negative, then the algorithm just returns 0.

Otherwise, note that

- Sum is set to 0 right before the scan of the maximum subarray. (Otherwise there is a postive subarray right before the maximum subarray and they can combine together to become a subarray with alarger sum.)
- Sum will not be set to 0 during the scan of the maximum subarray. (Otherwise the maximum subarray starts with a negative subarray.)

Therefore, maximum subrray value is recorded by *maxSum* and algorithm is correcrt.

(c) **Runtime analysis.**
Clearly the algorithm is $O(n)$.

3 Modular Fourier Transform

Fourier transforms (FT) have to deal with computations involving irrational numbers which can be tricky to implement in practice. Motivated by this, in this problem you will demonstrate how to do a Fourier transform in modular arithmetic, using modulo 5 as an example.

(a) There exists $\omega \in \{0, 1, 2, 3, 4\}$ such that ω are 4^{th} roots of unity (modulo 5), i.e., solutions to $z^4 = 1$. When doing the FT in modulo 5, this ω will serve a similar role to the primitive root of unity in our standard FT. Show that $\{1, 2, 3, 4\}$ are the 4^{th} roots of unity (modulo 5). Also show that $1 + \omega + \omega^2 + \omega^3 = 0 \pmod 5$ for $\omega = 2$.

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 0, 2)$ modulo 5; that is, multiply this vector by the matrix $M_4(\omega)$, for the value $\omega = 2$. Be sure to explicitly write out the FT matrix you will be using (with specific values, not just powers of ω). In the matrix multiplication, all calculations should be performed modulo 5.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 5.)

(d) Now show how to multiply the polynomials $2x^2 + 3$ and $-x + 3$ using the FT modulo 5.

Solution:

(a) We can check that $1^4 = 1 \pmod 5$,
$2^4 = 16 = 1 \pmod 5$,
$3^4 = 81 = 1 \pmod 5$,
$4^4 = 256 = 1 \pmod 5$.

Observe that taking $\boxed{\omega = 2}$ produces the following powers: $(\omega, \omega^2, \omega^3) = (2, 4, 3)$. Verify that

$$1 + \omega + \omega^2 + \omega^3 = 1 + 2 + 4 + 3 = 10 = 0 \pmod 5.$$

(b) For $\omega = 2$:

$$M_4(2) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}.$$

Multiplying with the sequence $(0, 1, 0, 2)$ we get the vector $\boxed{(3, 3, 2, 2)}$.

(c) For $\omega = 2$, the inverse matrix of $M_4(2)$ is the matrix

$$4^{-1} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix}$$

Verify that multiplying these two matrices mod 5 equals the identity. Also multiply this matrix with vector $(3, 3, 2, 2)$ to get the original sequence.

(d) For $\omega = 2$: We first express the polynomials as vectors of dimension 4 over the integers mod 5: $a = (3, 0, 2, 0)$, and $b = (3, -1, 0, 0) = (3, 4, 0, 0)$ respectively. We then apply the matrix $M_4(2)$ to both to get the transform of the two sequences. That produces $(0, 1, 0, 1)$ and $(2, 1, 4, 0)$ respectively. Then we just multiply the vectors coordinate-wise to get $(0, 1, 0, 0)$. This is the transform of the product of the two polynomials. Now, all we have to do is multiply by the inverse FT matrix $M_4(2)^{-1}$ to get the final polynomial in the coefficient representation. Recall that whe working with the FT outside of modspace, our inverse matrix of $M_4(\omega)$ would be given by $\frac{1}{4}M_4(\omega^{-1})$. In modspace, we can replace $\frac{1}{4}$ with the multiplicative inverse of 4, and ω^{-1} with the multiplicative inverse of 2. The properties of 2 that we found in the first part allow for this identity to hold. Thus the product is as follows: $\boxed{(4, 2, 1, 3)}$ or $\boxed{3x^3 + x^2 + 2x + 4}$.

4 Polynomial from roots

Given a polynomial with exactly n distinct roots at $r_1, \ldots, r_n$, compute the coefficient representation of this polynomial in time. Your runtime should be $O(n \log^c n)$ for some constant c (you should specify what c is in your runtime analysis). There may be multiple possible answers, but your algorithm should return the polynomial where the coefficient of the highest degree term is 1. You can give only the main idea and runtime analysis, a four part solution is not required.

Note: A root of a polynomial p is a number r such that $p(r) = 0$. The polynomial with roots $r_1, ..., r_k$ can be expressed as $\prod_i (x - r_i)$.

Solution:

Main idea

We are trying to find the coefficients of the polynomial $(x - r_1)(x - r_2) \cdots (x - r_n)$. Split the roots into two (approximately) equal halves: $r_1, r_2, \ldots, r_{\lfloor n/2 \rfloor}$ and $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Recursively find the polynomial whose roots are $r_1, \ldots, r_{\lfloor n/2 \rfloor}$, and the polynomial whose roots are $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Multiply these two polynomials together using FFT, which takes $O(n \log n)$. When the base case is reached with only 1 root r, return $(x - r)$.

Runtime Analysis

The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n \log n)$. Note that the master theorem doesn't apply here, so we'll have to solve this recurrence relation out directly. If we write out the recurrence tree, we can see that on the ith level, we do $2^i * (\frac{n}{2^i}) * log(\frac{n}{2^i})$ work for $\log n$ levels. We can upper bound the work on each level with $n \log n$, to get a total runtime of $O(n \log^2 n)$.

5 Triple sum

Design an efficient algorithm for the following problem. We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq n$. The algorithm must answer the following yes-or-no question: does there exist indices i, j, k such that $A[i] + A[j] + A[k] = n$?

Design an $O(n \lg n)$ time algorithm for this problem. Note that you do not need to actually return the indices; just yes or no is enough.

Hint: define a polynomial of degree $O(n)$ based upon A, then use FFT for fast polynomial multiplication. As an example, $(x + x^2) * (x^{10} + x^{20}) = x^{11} + x^{12} + x^{21} + x^{22}$.

Reminder: don't forget to include explanation, pseudocode, running time analysis, and proof of correctness.

Solution:

Main idea Exponentiation converts multiplication to addition. Observe $x^3 * x^2 = x^{2+3} = x^5$. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^n as a term.

Pseudocode

```
procedure ALGORITHM TRIPLESUM(($A[0 \ldots n-1]$, $t$))
    Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
    Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
    Return whether the coefficient of $x^n$ in $q$ is nonzero.
```

Proof of Correctness Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \leq i < n} x^{A[i]} \right)^3 = \left(\sum_{0 \leq i < n} x^{A[i]} \right) * \left(\sum_{0 \leq j < n} x^{A[j]} \right) * \left(\sum_{0 \leq k < n} x^{A[k]} \right)$$

$$= \sum_{0 \leq i,j,k < n} x^{A[i]} x^{A[j]} x^{A[k]} = \sum_{0 \leq i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^n in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = n$. So the algorithm is correct. (In fact, it does more: the coefficient of x^n tells us *how many* such triples (i, j, k) there are.)

Runtime Analysis Constructing $p(x)$ clearly takes $O(n)$ time. $p(x)$ is a polynomial of degree at most $n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

6 Searching for Viruses

Sherlock Holmes is trying to write a computer antivirus program. He starts by modeling his problem. He thinks of computer RAM as being a binary string s_2 of length m. He thinks of a virus as being a binary string s_1 of length $n < m$. His program needs to find all occurrences of s_1 in s_2 in order to get rid of the virus. Even worse, though, these viruses are still damaging if they differ slightly from s_1. So he wants to find all copies of s_1 in s_2 that differ in at most k locations for arbitrary $k \leq n$.

a) Give a $O(nm)$ time algorithm for this problem.

b) Give a $O(m \log m)$ time algorithm for any k. *(Hint: find a way to use FFT.)*
For this question we will not require a 4-part solution. Instead, please give us a clear description of the algorithm and an analysis of the running time. Give Pseudocode if you feel it will enhance your solution, but it is not required.

Solution:
a) For each of $i \in \{0, 1, \ldots, m-n\}$ starting points in s_2, check if the substring $s_2[i : i+n-1]$ differs from s_1 in at most k positions. The check takes $O(n)$ time at each of $O(m)$ starting points, so the time complexity is $O(mn)$.
b) If we replace the 0's in a bit string with -1's, checking whether two length n strings differ at no more than k bits is equivalent to checking whether the dot product of the two bit strings (i.e. bit-wise multiplication and add up the results) is at least $n - 2k$, as the positions they agree will contribute 1 to the dot product, while positions they disagree will contribute -1 to the dot product.

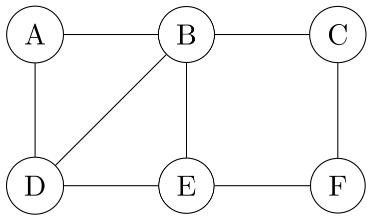
Now all we need are the $m-n+1$ dot products of all the $m-n+1$ length n substrings of s_2 with the bit string s_1. For people familiar with convolution, it should be straightforward to see this is exactly the same computation.

To present it as polynomial multiplication, consider $p_1(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ as a degree $n-1$ polynomial with $a_d = s_1'(n-d-1)$ for all $d \in \{0, 1, \ldots, n-1\}$, where s_1' is just s_1 with 0's replaced by -1's. $p_2(x) = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$ is a degree $m-1$ polynomial with $b_d = s_2'(d)$ for all $d \in \{0, 1, \ldots, m-1\}$, where again s_2' is just s_2 with 0's replaced by -1's. Notice $p_1(x)$ is reversed in the sense that the coefficients are in opposite order of the bits in s_1'

Now consider $p_3(x) = p_1(x) \times p_2(x) = c_0 + c_1 x + \cdots$, the coefficient of x^{n-1+j} in $p_3(x)$ is $c_{n-1+j} = \sum_{i=0}^{n-1} a_{n-1-i} b_{j+i} = \sum_{i=0}^{n-1} s_1'(i) s_2'(j+i)$ for any $j \in \{0, 1, \ldots, m-n\}$, which is exactly the dot product of the substring in s_2' starting at index j and the string s_1'. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m-n$ such that $c_{n-1+j} \geq n - 2k$.

The computation takes a FFT, a point-wise product, an inverse FFT, and a linear scan of the coefficients. The running time of this algorithm, $O(m \log n)$, is dominated by the FFT and inverse FFT steps, which each take $O(m \log m)$ time. The point-wise product and search for $c(i) \geq n - 2k$ each take $O(n)$ time.

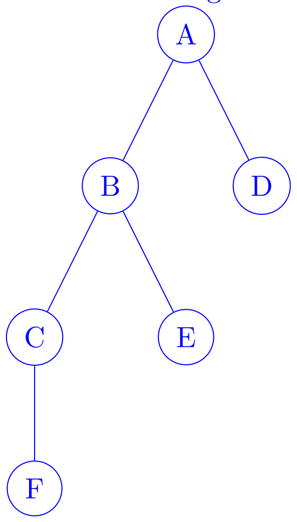
7 Breadth-First Search



Run breadth-first search on the above graph, breaking ties in alphabetical order (so the search starts from node A). At each step, state which node is processed and the resulting state of the queue. Draw the resulting BFS tree.

Solution:

The resulting BFS tree:



The node processed at each step and the resulting queue:

Node Processed	Queue
	A
A	B, D
B	D, C, E
D	C, E
C	E, F
E	F
F	{}

In this implementation of BFS, we ignore the technicality of pushing duplicate vertices into the queue. Note that we can deal with this by checking if the vertex at the head of the queue has been visited already and discarding it if so.

8 True Source

Design an efficient algorithm that given a directed graph G determines whether there is a vertex v from which every other vertex can be reached. (Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.)

Solution:

We provide two solutions below that both run in linear time (there may be many more).

Solution 1: In directed acyclic graphs, this is easy to check. We just need to see if the number of source nodes (zero indegree) is 1 or more than 1. Certainly if it is more than 1, there is no true source, because one cannot reach either source from the other. But if there is only 1, that source can reach every other vertex, because if v is any other vertex, if we keep taking one of the incoming edges, starting at v, we have to either reach the source, or see a repeat vertex. But the fact that the graph is acyclic means that we can't see a repeat vertex, so we have to reach the source. This means that the source can reach any vertex in the graph.

Now for general graphs, we first form the SCCs, and the metagraph. Now if there is only one source SCC, any vertex from it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.

Finding the SCCs/metagraph can be done in $O(|V| + |E|)$ time via DFS as seen in the textbook, and counting the number of sources in the metagraph can also be done in $O(|V|+|E|)$ time by just computing the in-degrees of all vertices using a single scan over the edges.

Solution 2: There is an alternative solution which avoids computing the metagraph altogether. The solution is to run DFS once on G to form a DFS forest. Now, let v be the root of the last tree that this run of DFS visited. Run DFS starting from v to determine if every vertex can be reached from v. If so, output v, if not, output that no true source exists.

It suffices to show no other vertex besides v can be a true source. In this case, if we determine v is not a true source, then saying there is no true source is correct. (Of course, if we find v is a true source, outputting it is also correct)

If there is a true source u, v can't reach u because v is not a true source. So nothing visited after v can be a true source, since v is the last root and thus all vertices visited after v are reachable from v. But every vertex visited before v must not be able to reach v, because otherwise the DFS would have taken a path from one of those vertices to v and thus v would not be a root in the DFS forest. So nothing visited before v can be a true source, since nothing visited before v can reach v. Thus v is the only candidate for a true source.

Since the algorithm just involves running DFS twice, it runs in linear time.