Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Triple sum

We are given an array $A[0 \dots n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq n$ (the elements are not necessarily distinct). We would like to know if there exist indices $0 \leq i, j, k \leq n-1$ (not necessarily distinct) such that

$$A[i] + A[j] + A[k] = n$$

1. First, consider 2SUM, a simplified version of triple sum where you determine if there exist indices $0 \leq i, j \leq n-1$ (not necessarily distinct) such that

   $$A[i] + A[j] = n$$

   Consider the array $[1, 3, 5]$ and $n = 5$. What are all the possible 2SUMs?

2. Now try encoding the above array into a polynomial to solve 2SUM with one polynomial multiplication. Then, how would you encode an arbitrary array to solve 2SUM?

   Hint: given $p(x) = x^1 + x^3 + x^5$, compute $p(x)^2$. What are the resulting coefficients and exponents in the product? Can they be used to solve 2SUM?

3. Now, design an $\mathcal{O}(n \log n)$ time algorithm for triple sum. Note that you do not need to actually return the indices; just yes or no is enough.

   Food for thought: is it possible to return the number of ways you can add 3 elements from A to equal n?

Solution:

1. (a) $1 + 1 = 2$
   (b) $1 + 3 = 4$

(c) $1 + 5 = 6$
(d) $3 + 3 = 6$
(e) $3 + 5 = 8$
(f) $5 + 5 = 10$

2. $p(x)^2 = x^2 + 2x^4 + 3x^6 + 2x^8 + x^{10}$. In this example, we see that by encoding the array as the exponents of the polynomial, and the square of the polynomial contains terms whose exponents match the two sums. In general, to solve 2 sum, we can encode an array A as $p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}$. Then, we use FFT to compute $p(x)^2$ in $\mathcal{O}(n \log n)$. Then we check if the term x^n has a non-zero coefficient. If it does, then this 2SUM exists, otherwise it doesn't. See part (c) for the full proof and runtime analysis.

3. **Main idea.** Exponentiation converts multiplication to addition.

   For example, observe $x^3 \cdot x^2 = x^{2+3} = x^5$. This gives us the idea to represent the lists as polynomials, with the elements in the lists in the exponents (This is a very common trick that you should remember!). For example, we can represent the array $[1, 3]$ as $x^1 + x^3$. If we multiply this with the polynomial for $[2, 4]$, $x^2 + x^4$, we get the polynomial $x^3 + 2x^5 + x^7$. Notice that $3, 5, 7$ correspond to the possible sums of an element in $[1, 3]$ and an element in $[2, 4]$.

   In general, we 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.

   Proof of Correctness. Observe that

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

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

   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 $0 \ldots n - 1$. 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.

2 Pattern Matching

Consider the following string matching problem:

Input:

- A string g of length n made of 0s and 1s. Let us call g, the "pattern".
- A string s of length m made of 0s and 1s. Let us call s the "sequence".
- Integer k

Goal: Find the (starting) locations of all length n-substrings of s which match g in at least $n-k$ positions.

Example: Using 0-indexing, if $g = 0111$, $s = 01010110111$, and $k = 1$ your algorithm should output 0, 2, 4, and 7.

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

Solution:

For each of $i \in \{0, 1, \ldots, m-n\}$ starting points in s, check if the substring $s[i : i+n-1]$ differs from g in at most k positions. The check takes $O(n)$ time at each of $O(m)$ starting points, so the time complexity is $O(mn)$.

We will now design an $O(m \log m)$ time algorithm for the problem using FFT. Pause a moment here to contemplate how strange this is. *What does matching strings have to do with roots of unity and complex numbers?*

(b) First consider $g = 0110$, $s = 0110$. We know that g and s match at index 0, since all 4 characters of g matches the characters in s.

Now, say we have a function $(x, y) \mapsto f(x, y)$ that returns `len(x)` if the strings x and y match exactly and some number less than `len(x)` otherwise. How can $f(x, y)$ be used to determine the indices where g shows up in s?

Solution: $f(x, y)$ can be used to determine the indices where g and s match by calling

```
f(g, s[i:i+len(g)]) for i in range(0, len(s) - len(g)),
```

and then returning the indices i for which `f(g, s[i:i+len(g)]) == len(g)`.

(c) Now, lets try to create $f(x, y)$ using the dot product of x and y. That is, $f(x, y) = x \cdot y = \sum_i x[i]y[i]$. Give a counter example for when the dot product returns the wrong answer. Why does this happen?

Solution: For the example $g = 0110$ and $s = 0110$, the dot product is 2. Since $0 * 0 = 0$, a simple dot product might not return length(x) even if $x = y$ since it doesn't count all the matching 0.

(d) However, we can modify this dot product to work by first encoding the bits of of x and y. That is, we want to find a bit-mapping function $\Phi : \{0, 1\} \mapsto \mathbb{R}$ such that $f(x, y) = \sum_i \Phi(x[i])\Phi(y[i])$ equals to `len(`x`)` if x and y match exactly and some number less than `len(`x`)` otherwise. Construct a mapping Φ for which this property holds.

Hint: find Φ *such that*

$$\Phi(x[i])\Phi(y[i]) = \begin{cases} 1 & x[i] = y[i] \\ -1 & x[i] \neq y[i] \end{cases}$$

Solution:

$$\Phi(0) = -1$$
$$\Phi(1) = +1$$

(e) Now, devise an FFT based algorithm for the problem that runs in time $O(m \log m)$ using the insights of the previous subparts.

Hint: if $p(x) = p_0 + p_1x + p_2x^2 + ... + p_{n-1}x^{n-1}$ *and* $q(x) = q_0 + q_1x + q_2x^2 + ... + q_{m-1}x^{m-1}$, *then the coefficient of the term* x^b *in* $p(x)q(x)$ *is* $\sum_{i=0}^{b} p_i q_{b-i}$. *Can this be used somehow to compute a dot product?*

Solution:
We want to increase the "similarity" of a slice of s with g whenever their corresponding characters match. Taking the dot product of these 2 bitstrings will reflect how many pairs of bits are both 1. To also capture the case of both bits being 0, we replace each 0 with -1, so when we multiply two of the same bits, we get a result of 1, while if we multiply two different bits, we get -1. The sum of this product over all characters represents how many characters of the slice of s and g match minus the number of characters that don't.

We'd like to efficiently compute these dot products for all slices of s. Encoding the bits of s and the reverse of g as coefficients of polynomials will give us 2 polynomials that, when multiplied, reflect all combinations of slices and starting index for similarity comparisons in its coefficients.

Main Idea: Let g' be g with 0's replaced by -1's. Let

$$p_1(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$$

where $a_d = g'(n-d-1)$ for all $d \in \{0, 1, \ldots, n-1\}$. Similarly, let

$$p_2(x) = b_0 + b_1 x + \cdots + b_{m-1} x^{m-1}$$

where $b_d = s'(d)$ for all $d \in \{0, 1, \ldots, m-1\}$, where again s' is just s 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 g.

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

$$c_{n-1+j} = \sum_{i=0}^{n-1} a_{n-1-i} b_{j+i} = \sum_{i=0}^{n-1} g'(i) s'(j+i)$$

for any $j \in \{0, 1, \ldots, m-n\}$, which is exactly the dot product of the substring in s' starting at index j and the string g'. If these strings differ in at most k positions, then this dot product will be at least $n - 2k$. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m - n$ such that $c_{n-1+j} \geq n - 2k$.

Proof of Correctness: Consider the mapping Φ that maps binary digits as follows:

$$\Phi(0) = -1$$
$$\Phi(1) = +1$$

Going along with the hint, construct the polynomial

$$G(x) = \sum_i G_i x^i$$

from the pattern $g[0 : n-1]$, by setting $G[i] = \Phi(g[n-1-i])$.

Similarly, construct the polynomial

$$S(x) = \sum_i S_i x^i$$

from the pattern $s[0 : m-1]$ by setting $S[i] = \Phi(s[i])$.

Now consider the product of these two polynomials $G(x)$ and $S(x)$,

$$\begin{aligned} G(x) \cdot S(x) &= \left(\sum_{i=0}^{n-1} G[i] x^i \right) \cdot \left(\sum_{j=0}^{m-1} S[j] x^j \right) \\ &= \sum_{r=0}^{m+n-2} x^r \cdot \left(\sum_{\ell=0}^{r} G[\ell] \cdot S[r-\ell] \right) \end{aligned}$$

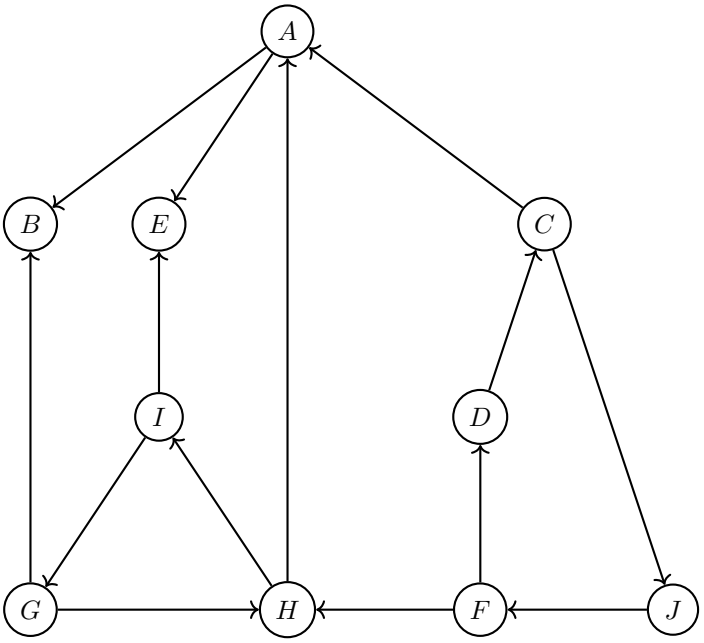
In the last step, we are just writing out the coefficients of polynomial multiplication. Now, let us look at the coefficients a little bit more carefully. Let us start with coefficient of x^{n-1}, i.e., $r = n-1$,

$$\begin{aligned} \text{Coefficient of } x^r &= \sum_{\ell=0}^{n-1} G[\ell] \cdot S[r-\ell] \\ &= G \cdot S[r-n+1 : r]. \end{aligned}$$

This is the dot product of $\Phi(g)$ and $\Phi(s[:n])$ when $r = n-1$. More generally, the coefficient of x^r is the dot product of $\Phi(g)$ and $\Phi(s[:r+1])$. Note that if these strings differ in at most k positions, then this dot product will be at least $n-2k$, since each matching character would add $+1$ at least $n-k$ times, and each differing character would add -1 at most k times. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m-n$ such that $c_{n-1+j} \geq n-2k$.

Runtime Analysis: 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, $O(m \log m)$, 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 $O(m)$ time.

3 Graph Traversal



(a) Recall that given a DFS tree, we can classify edges into one of four types:

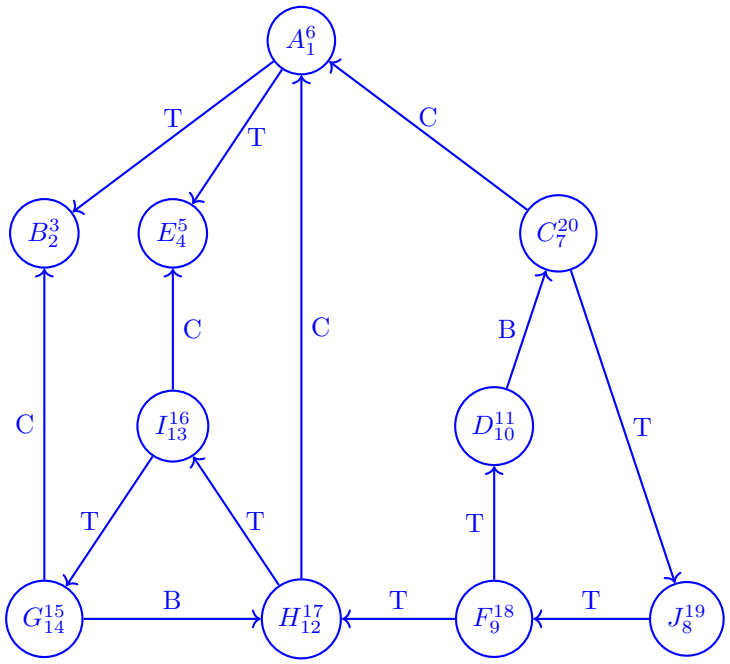
- Tree edges are edges in the DFS tree,
- Back edges are edges (u, v) not in the DFS tree where v is the ancestor of u in the DFS tree
- Forward edges are edges (u, v) not in the DFS tree where u is the ancestor of v in the DFS tree
- Cross edges are edges (u, v) not in the DFS tree where u is not the ancestor of v, nor is v the ancestor of u.

For the directed graph above, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as **T**ree, **B**ack, **F**orward or **C**ross.

(b) A strongly connected component (SCC) is defined as a subset of vertices in which there exists a path from each vertex to another vertex. What are the SCCs of the above graph?

(c) Collapse each SCC you found in part (b) into a meta-node, so that you end up with a graph of the SCC meta-nodes. Draw this graph below, and describe its structure.

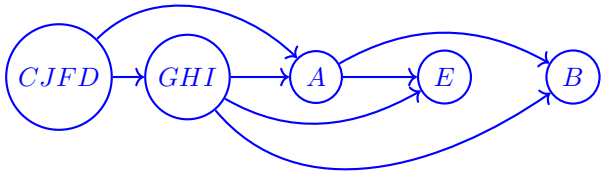
Solution:



(a)

(b)

$$\{A\}, \{B\}, \{E\}, \{G, H, I\}, \{C, J, F, D\}$$



(c)

4 Not So Exciting Scheduling

PNP University requires students to finish all prerequisites for a certain class before taking it; however, they made some mistakes when assigning prerequisites. Thus, some classes at PNP University are NP (Not Possible to take) due to it being impossible to take all its prerequisite classes whilst following the prerequisite rule for them or their prerequisites. Thus, students wish to figure out whether their classes can all be taken or not. Their n classes are labelled with unique identifiers $\{c_1, c_2, \ldots, c_n\}$, and the set of m prerequisites in the form $[c_i, c_j]$ indicate that c_i must be taken before c_j.

Design an algorithm that outputs a potential scheduling of classes (i.e. an order to take all the classes in) if there are no NP classes, return false otherwise.

Solution:

Algorithm: Create a node for each class. For each prerequisite $[c_i, c_j]$, construct a directed edge from c_j to c_i. Run DFS on the resulting graph while keeping track of pre and post numbers. (Cycle detection) If a back edge exists, then output false. Otherwise sort the nodes on post numbers, from highest to lowest (topological sort on DAG) or use a stack to keep track of visited nodes so that you don't need to do extra sorting at the end. And this will create a valid scheduling.

Proof of Correctness: Proof for cycle detection and topological sort will be the same as seen in lecture. A directed graph is a DAG if and only if there is no back edge. And because DAG has no back edge, if we order the vertices from highest post number to lowest post number, it's guaranteed to be topologically sorted by definition.

Runtime Analysis: Constructing the graph will take $O(n + m)$ time since we have n nodes and m directed edges. Running DFS will take again $O(n+m)$. We can do linear scan to find back edge. If we used stack to keep track of visited nodes, then topological sort will simply take linear time because we can just read off the nodes from the back the stack. Therefore, the overall runtime will be $O(n + m)$.