1 Find the Universal Hash Families

Let \( U = \{000, 001, 002, \ldots, 999\} \) (aka, all of the numbers between 0 and 999, padded so that they are three digits long) and let \( n = 10 \). For each of the following hash families \( H \) consisting of functions \( h : U \to \{0, \ldots, n-1\} \), decide whether \( H \) is universal or not, and justify your result with a formal proof.

(a) For \( i = 1, 2, 3 \), let \( h_i(x) \) be the \( i \)’th least-significant digit of \( x \). (For example, \( h_2(456) = 5 \)). Define \( H = \{h_1, h_2, h_3\} \). Is \( H \) a universal hash family?

(b) For \( a \in \{1, \ldots, 9\} \), let \( h_a(x) \) be the least-significant digit of \( ax \). (For example, \( h_2(123) \) is the least-significant digit of \( 2 \times 123 = 246 \), which is 6). Define \( H = \{h_i : i = 1, \ldots, 9\} \). Is \( H \) a universal hash family?

1. This is not a universal hash family. To see this, consider \( x = 111 \) and \( y = 112 \). Now, when we choose \( h \in H \) at random, the probability that \( h(x) = h(y) \) is \( 2/3 \), because \( h_2(x) = h_2(y) = 1 \) and \( h_3(x) = h_3(y) = 1 \). But for \( H \) to be a universal hash family I would need \( P(h(x) = h(y)) \leq 1/10 \). Since this is not the case, \( H \) is not a universal hash family.

2. This is not a universal hash family. To see this, consider \( x = 000 \) and \( y = 010 \). For any \( a \in \{1, \ldots, 9\} \), we have \( ax = 0 \) and \( ay = 10 \cdot a \). Thus, the least significant digit of both \( ax \) and \( ay \) is 0. That means that for any \( a \), \( h(x) = h(y) = 0 \), so \( P_{h \in H}(h(x) = h(y)) = 1 \), which is definitely larger than \( 1/10 \).

2 Zero Sum Subarrays

Given an array \( A \) of positive and negative integers, determine if there is a subarray with zero sum. A subarray is a contiguous chunk of the original array and has at least one element.

(a) What is a brute force solution, and what is its runtime?

(b) Design an algorithm that uses hashing to solve this problem in \( O(n) \) time.

Let \( n \) be the length of the array. The brute force solution is to consider all subarrays one by one and check the sum of every subarray. We do this using two loops that decides the start and end indices of the subarray. Time complexity of this method is \( O(n^3) \), space complexity \( O(1) \).

If we pre-compute the running sums for the array, we can find the sum of each subarray in \( O(1) \) giving us a total run time of \( O(n^2) \) and space complexity \( O(n) \).

A better solution that does not use extra space is the following (time complexity \( O(n^2) \)): 

1

```java
for (int i = 0; i < n; i++) {
    int sum=0;
    for (int end = i; end < n; end++) {
        sum+=A[end];
        if (sum == 0)
            return true;
    }
}
return false;
```

Another solution that checks for zero-sum is hashing the running sum $A[0 : i]$ for $0 \leq i \leq n$. If the running sum is zero for any $0 \leq i \leq n$, the subarray $A[0 : i]$ has sum zero. If not, check if the current value of running sum was already stored by our hash function (this check takes $O(1)$ time). If so, there exists an index $j < i$ such that $A[0 : j]$ has the same sum as $A[0 : i]$. This means the subarray $A[j+1 : i]$ has sum 0. In either case we return true. Time and space complexity is $O(n)$.

```
# Initialize a hash-set, say hSet
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += A[i];
    if (hSet.contains(sum) or sum == 0)
        return true;
    hSet.add(sum);
}
return false;
```

## 3 Strongly Connected Components

Ten “friends” decided to make a Valentine’s Day graph in which each person is a node, and an edge points from $v_i$ to $v_j$ if Person $j$ has a crush on Person $i$. Below is their graph:
What are all the strongly connected components of this graph? (i.e., groups of vertices such that there exists a path between any two vertices in the group)

\{E\}, \{A, B, C, D, F, G, H, I, J\}. The only strategy here is to first notice that E is a lonely node, and the rest is just visually verifying that the rest of nodes can all reach each other (i.e. satisfy the definition of an SCC).

4 Bipartite Graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, $U$ and $V$ such that every edge $(u, v)$ either connects a vertex from $U$ to $V$ or a vertex from $V$ to $U$. A bipartite graph is possible if the graph can be colored using two colors such that vertices in the same set are colored with the same color. Design an algorithm using DFS to determine whether or not a graph is bipartite.

The algorithm is essentially the same as that of DFS, except at every node we visit, we either color it if it hasn’t been visited before, or check its color if it has been visited before. The rough algorithm is as follows:

1. Start DFS from any node and color it RED
2. Color the next node BLUE
3. Continue coloring each successive node the opposite color until the end of the tree is reached
4. If at any point a current node is the same color as its parent node, then return false
5. Once every node has been visited, if we haven’t returned false, then the graph is bipartite
5 Russian Boxes

Through hard work and a small stroke of good luck you have glowed up over the past year and are suddenly inundated with admirers for Valentine's Day, who have each sent you a box of chocolates (if you are allergic to chocolates, you may assume they sent you strawberries or something else that is nice and comes in a box). After you finish consuming the spoils of your attractiveness, you are left with \( n \) empty rectangular boxes (you may assume \( n \) VERY large), and you decide to nest some of them within each other for easy storage. The \( i \)-th box has dimensions \( w_i \times h_i \). Box \( i \) can fit inside box \( j \) if and only if \( w_i < w_j \) and \( h_i < h_j \).

A sequence of boxes \( b_1, b_2, ..., b_k \) form a chain if box \( b_i \) fits inside box \( b_{i+1} \) for each \( 1 \leq i < k \). Design an algorithm which takes as input a list of dimensions \( w_i \times h_i \) and returns the length of the longest possible chain of boxes. You must construct a directed graph as part of your solution.

BONUS: Having found the length of the longest possible chain, how can you use your directed graph to return the chain itself? (If there is more than one longest chain, you may return any of them)

Construct a directed graph whose vertices are boxes, and such that there is an edge \((v_i, v_j)\) iff box \( v_i \) fits inside box \( v_j \). Notice that this graph is a DAG (it is impossible for some box \( i \) to fit in box \( j \) and also contain \( j \)). Our goal is now to find the length of the longest path.

We’re going to approach this by trying to compute the length of the longest chain that ends at any node \( v_i \). We think to ourselves, “for any box \( i \), it would be amazing if only we knew the max chain lengths that end at any of the boxes that can fit into box \( i \) - we’d then just pop that chain of boxes into box \( i \) and we would have extended that chain by \( 1^n \).” This is the idea of DYNAMIC PROGRAMMING (DP), which is a concept we’ll cover in the first lecture of week 7. Basically, DP is all about re-using answers to subproblems (‘memoization’, if you’ve heard it from 106B), and the key thing is to figure out what that subproblem structure is. In our case, we’d like to solve for ‘smaller’ boxes first, or really, we want to solve in order of what the topoSort result of our graph would give us (think of it as each node depending only on the boxes that could fit in it). Thus we linearize (i.e., create an ordering \( v_1, \ldots, v_n \) of the nodes via topological sort) the graph, so whenever there is an edge from \( v_i \) to \( v_j \), \( i < j \).

For every node \( v_i \), let \( \ell_i \) be the length of the longest path ending at \( v_i \). We can compute \( \ell_i \) as follows:

\[
\ell_i = 1 + \max_{(v_j, v_i) \in E} \ell_j
\]

Because we have linearized the graph, \( \ell_i \) depends only on \( \ell_j \) for \( j < i \). That is, the length of a chain ending at some box \( i \) only depends on the lengths of the chains ending at all boxes \( j \) that can fit inside \( i \).

Thus we can compute the \( \ell_i \) values in order of “size,” and our longest chain has length \( \ell^* = \max_{i=1}^n \ell_i \). One way to do this is to start at node 1 and add “weights” to each edge of our directed graph, where weights are equivalent to the length of the longest path to the source node (i.e., the weight of edge \((v_i, v_j)\) is \( \ell_i \)).

(This is really a great example of a dynamic programming problem—we will cover more details about dynamic programming later next week, so stay tuned for that, and don’t worry if it’s not an obvious approach to you yet!)

To return the chain itself, we can do a modified version of DFS (or BFS) in which we keep track of the length of our current paths and return once we have found a path of the desired length \( \ell^* \). Here’s one method: construct a starting set \( S \) of all nodes that have no edges in. For each \( v \) in \( S \), begin a recursive depth-first-search from \( v \) which takes two arguments: the current node and the current length of the chain. Rather than having a specific target node to catalyze returning a solution, we return only when the current length is equivalent to our precomputed maximal length.)
Other things to think about on your own: is it better to use DFS or BFS? In what situations would one be faster than the other?