University of Nevada, Las Vegas Computer Science 477/677 Fall 2020

Answers to Assignment 7: Due Monday November 9, 2020

Unless otherwise specified, n refers to the number of vertices and m refers to the number of edges of a graph or directed graph.

1. Give an algorithm for converting the array of outneighbor lists of a weighted directed graph into the array of inneighbor lists of the same graph in $O(n+m)$ time.

   Let the vertices of G be $v_1, v_2, \ldots v_n$. Assume that there is an array Out of length n, where Out$[i]$ is a list of ordered pairs $(j, w_{i,j})$ where v_j is an outneighbor of v and $w_{i,j}$ is the weight of (v_i, v_j).

   for $1 \le j \le n$
   In$[j]$ = empty list
   for $1 \le i \le n$
   for all $(j, w_{i,j})$ in Out$[i]$
   append the ordered pair $(i, w_{i,j})$ to In$[j]$

   In$[j]$ will be the list of ordered pairs $(i, w_{i,j})$ such that $w_{i,j}$ is the weight of the arc (v_i, v_j).

2. 8 items need to be inserted into a cuckoo hash table of size 8. Insert the items in the order listed below. After each item, there are two hash values.

   Cuckoo hasing always has some probability of failure, and it fails in this case. We run the algorithm:

Ann	0	5
Bob	5	2
Cal	2	4
Dan	4	0
Eve	3	6
Fay	1	7
Gus	4	3
Hal	5	4

0	~~Ann~~	Dan	
1	Fay		
2	~~Cal~~	Bob	
3	~~Eve~~	Gus	
4	~~Dan~~	~~Gus~~	Cal
5	~~Bob~~	~~Ann~~	Hal
6	Eve		
7			

0	~~Ann~~	~~Dan~~	~~Ann~~	Dan	
1	Fay				
2	~~Cal~~	~~Bob~~	~~Cal~~	Bob	
3	~~Eve~~	Gus			
4	~~Dan~~	~~Gus~~	~~Cal~~	~~Dan~~	~~Hal~~ Cal
5	~~Bob~~	~~Ann~~	~~Hal~~	~~Bob~~	~~Ann~~ Hal
6	Eve				
7					

   After 14 steps, seven places in the hash table are filled, and Ann needs to be inserted.
   After 10 more steps, the hash table has returned to that configuration, hence the algorithm is in an infinite loop.
   The marriage theorem can be used to prove that no solution is possible. The "boys" are the items to be inserted, and the "girls" are the possible hash values. Let B be the set of "boys" {Ann, Bob, Cal, Dan, Hal}. The set of girls "known" to the boys in B is G $= \{0, 2, 4, 5\}$. Since |B| $= 4$ is greater than |G| $= 4$, There is no way to assign a different hash value to each item.

3. Give code or pseudocode for the longest monotone increasing subsequence problem. Given a sequence of n real numbers A(1) ... A(n), determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence. I have a Youtube video for this problem.

   My algorithm, given below in C++ code, takes $O(n \log n)$ time. There are other algorithms for this problem, some of which could be better. Here are the variables used in the program. We assume that all terms of the given sequence are non-negative integers, since the algorithm for real numbers is the same.

   `A[1] ... A[n]`, the given sequence. We assign `A[0] = -1`, a *sentinel*.
   `back[1] ... back[n]: back[i]` is the index of the next-to-the last term of the longest strictly monotone sub sequence ending at `A[i]`. If that sequence has length 0, then `back[i] = 0`.
   `len`: the length of the longest monotone subsequence found so far.
   `index[0], index[1] ... index[len]: index[ell]` is the index of the last term of the best strictly monontone subsequence of length `ell` found so far.

   Steps of the algorithm.
   Initialization: `len = 0` and `index[0] = 0`.
   Process Indices: When `1 <= k <= n` is processed, `index[ell] = k` for some `ell`, `back[k]` is assigned, and `len` may or may not be incremented.
   There are two cases. If `A[k] > A[index[len]]`, then `A[k]` is appended to the currently longest subsequence: `back[k] = index[len]`, `len++`, then `index[len] = k`.
   Otherwise, use binary search to find the length `ell` such that `A[ell] < A[k] <= A[index[ell+1]]`. `A[k]` will replace `A[index[ell+1]]` as the last term of the best subsequence of length `ell+1`. We execute `back[k] = index[ell]`, then `ell++`, then `index[ell] = k`.
   Finally, the longest strictly monotone increasing subsequence is obtained recursively using the back pointers, starting at `index[len]`.

```
// lincsub04.cpp longest strictly increasing subsequence

  int const N = 1000; // input sequence length cannot exceed 999
  int n; // length of the input sequence
  int A[N+1]; // A[1 .. n] is the input sequence
  int const sentl = -1; // sentinel value, the 0^th term of A
  int back[N+1]; // back pointer
  int len; // length of the longest monotone subsequence so far
  int index[N+1]; // index[ell] = index of last term of longest monotone
                  // increasing subsequence of length ell for 0 <= ell <= len

  void getA() // read the input sequence from cin
   {
    A[0] = sentl;
    cin >> A[1];
    n = 1;
    while(!cin.eof())
     {
      assert(A[n] >= 0); // No negative terms are allowed.
```

```
    n++;
    cin >> A[n];
   }
  n--;
 }

void writesubsequence(int indx)
 { // writes longest monotone increasing subsequence ending at A[indx])
  if(indx > 0)
   {
    writesubsequence(back[indx]);
    cout << " " << A[indx];
   }
 }

void writeall(int k)
 {
  if(k == 1)
   cout << "After processing one term of the sequence:" << endl;
  else
   cout << "After processing " << k << " terms of the sequence:" << endl;
  cout << "A:    ";
  for(int i = 0; i <= k; i++)
   cout << " " << A[i];
  cout << endl;
  cout << "back:    ";
  for(int i = 1; i <= k; i++)
   cout << " " << back[i];
  cout << endl;
  cout << "len = " << len << endl;
  cout << "index: ";
  for(int ell = 0; ell <= len; ell++)
   cout << " " << index[ell];
  cout << endl;
  cout << "subsequence: ";
  writesubsequence(index[len]);
  cout << endl << endl;
 }

void processindex(int k)
 {
  if(A[k] > A[index[len]])
   {
    back[k] = index[len];
    len++;
    index[len] = k;
   }
  else
```

```
   { // binary search for the place to insert new entry A[k]
    int ell = 0;
    int hi = len; // loop invariant: A[index[ell] < A[k] <= A[hi])
     // we must have a STRICTLY monotone increasing subsequence
    while(ell+1 < hi)
     {
      int mid = (ell+hi)/2;
      if(A[index[mid]] < A[k]) ell = mid;
      else hi = mid;
     }
    assert(ell+1 == hi);
    assert(A[k] > A[index[ell]] and A[k] <= A[index[ell+1]]);
    back[k] = index[ell];
    index[ell+1] = k;
   }
  writeall(k); // The current values of the arrays
 }

void finalreport()
 {
  cout << "Finalreport:" << endl;
  cout << "The longest strictly monotone increasing subsequence of" << endl;
  for(int i = 1; i <= n; i++)
   cout << A[i] << " ";
  cout << "is" << endl;
  writesubsequence(index[len]);
  cout << " which has length " << len << endl;
 }

void mainwork()
 {
  getA();
  index[0] = 0;
  len = 0;
  for(int k = 1; k <= n; k++)
   processindex(k);
  cout << endl;
  finalreport();
 }

int main()
 {
  mainwork();
  return 1;
 }
```

Here is my output, where the input file was 4 1 6 8 5 9 2 3 3 6 7 4.

```
After processing one term of the sequence:
A:     -1 4
back:     0
len = 1
index:  0 1
subsequence:  4

After processing 2 terms of the sequence:
A:     -1 4 1
back:     0 0
len = 1
index:  0 2
subsequence:  1

After processing 3 terms of the sequence:
A:     -1 4 1 6
back:     0 0 2
len = 2
index:  0 2 3
subsequence:  1 6

After processing 4 terms of the sequence:
A:     -1 4 1 6 8
back:     0 0 2 3
len = 3
index:  0 2 3 4
subsequence:  1 6 8

After processing 5 terms of the sequence:
A:     -1 4 1 6 8 5
back:     0 0 2 3 2
len = 3
index:  0 2 5 4
subsequence:  1 6 8

After processing 6 terms of the sequence:
A:     -1 4 1 6 8 5 9
back:     0 0 2 3 2 4
len = 4
index:  0 2 5 4 6
subsequence:  1 6 8 9

After processing 7 terms of the sequence:
A:     -1 4 1 6 8 5 9 2
back:     0 0 2 3 2 4 2
len = 4
```

```
index:  0 2 7 4 6
subsequence:  1 6 8 9

After processing 8 terms of the sequence:
A:     -1 4 1 6 8 5 9 2 3
back:     0 0 2 3 2 4 2 7
len = 4
index:  0 2 7 8 6
subsequence:  1 6 8 9

After processing 9 terms of the sequence:
A:     -1 4 1 6 8 5 9 2 3 3
back:     0 0 2 3 2 4 2 7 7
len = 4
index:  0 2 7 9 6
subsequence:  1 6 8 9

After processing 10 terms of the sequence:
A:     -1 4 1 6 8 5 9 2 3 3 6
back:     0 0 2 3 2 4 2 7 7 9
len = 4
index:  0 2 7 9 10
subsequence:  1 2 3 6

After processing 11 terms of the sequence:
A:     -1 4 1 6 8 5 9 2 3 3 6 7
back:     0 0 2 3 2 4 2 7 7 9 10
len = 5
index:  0 2 7 9 10 11
subsequence:  1 2 3 6 7

After processing 12 terms of the sequence:
A:     -1 4 1 6 8 5 9 2 3 3 6 7 4
back:     0 0 2 3 2 4 2 7 7 9 10 9
len = 5
index:  0 2 7 9 12 11
subsequence:  1 2 3 6 7

Finalreport:
The longest strictly monotone increasing subsequence of
4 1 6 8 5 9 2 3 3 6 7 4 is
 1 2 3 6 7 which has length 5
```

4. Give an $O(n+m)$-time algorithm for finding a topological order, if there is one, for a directed graph given as an array of inneighbor lists. If the graph does not have a topological order, your algorithm should report that fact and find a cycle.

   Assume that the arrays In[1..n] and Out[1..n] of lists of neighbors are given. In $O(n)$ time, we can ensure theat each list is sorted.

   For each $1 \leq j \leq n$ compute $Indeg[j]$. Let Z be a stack, which initially contains all vertices of indegree zero.

```
While(Z is not empty)
    j = pop(Z);
    write j onto topological list.
    for all i in Out[j]
        delete j from In[i]
        decrement Indeg[i]
        If Indeg[i] = 0
            push i onto Z
```

   If Z is empty, we are done. Otherwise, there are still vertices with indegree greater than zero. Pick one of those, and start a search using In pointers. When a vertex is repeated, you have a cycle.

5. Suppose that you wish to store a $n \times n \times n$ 3-dimensional array A of integers. You could declare int `A[n][n][n]`, using n^3 space. However, in your application, you only need values of `A[i][j][k]` for i $\geq$ j $\geq$ k. Explain how you would construct a data structure for A which takes $n^3/6 + O(n^2)$ space, and which allows the operators **fetch** and **store** to be executed in $O(1)$ time each.

   We will store all entries of A into a 1-dimensional array `X[N]` where N $= \frac{2n^3+9n^2+4n}{12}$, in lexical order. `A[i,j,k]` will be stored at `X[p]` where p = p(i,j,k) is the number of predecessors of the ordered triple (i,j,k). For convenience, we assume that indices of A are in the range 0 ... n–1, and the indices of X are 0 ... N–1. We use the following formulae:[1]

   $\mathtt{p}(0,0,0) = 0$

   $\mathtt{p}(\mathtt{i},0,0) = \frac{2\mathtt{i}^3+9\mathtt{i}^2+4\mathtt{i}}{12}$

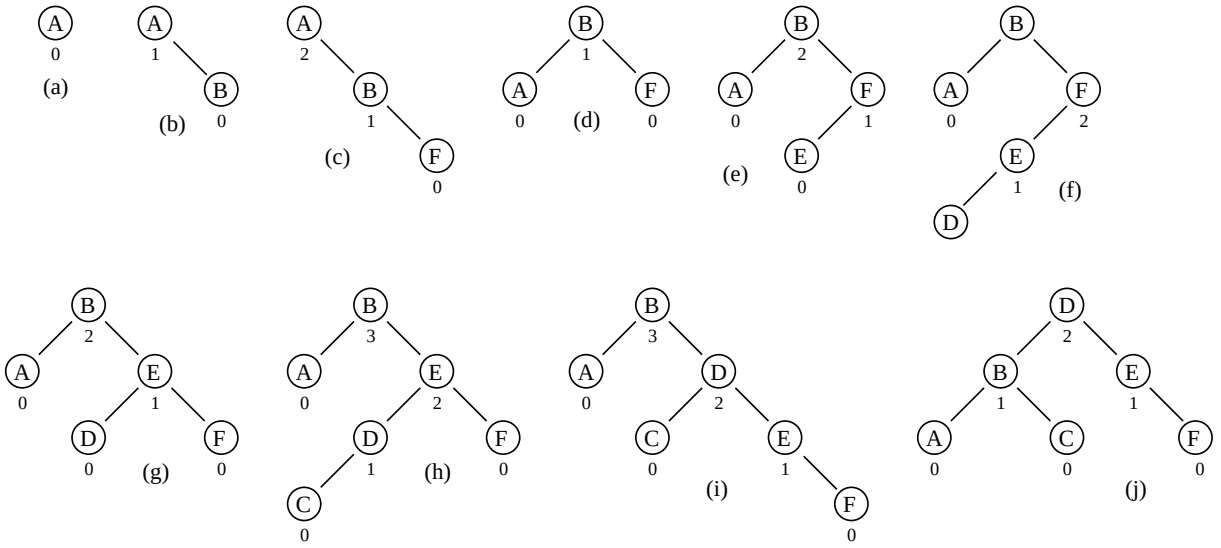
   $\mathtt{p}(\mathtt{i},\mathtt{j},0) = \mathtt{p}(\mathtt{i},0,0) + \frac{\mathtt{j}(\mathtt{j}+1)}{2}$

   $\mathtt{p}(\mathtt{i},\mathtt{j},\mathtt{k}) = \mathtt{p}(\mathtt{i},\mathtt{j},0) + \mathtt{k}$

   Thus, `A[i,j,k]` $= \mathtt{X}\left[\frac{2\mathtt{i}^3+9\mathtt{i}^2+4\mathtt{i}}{12} + \frac{\mathtt{j}(\mathtt{j}+1)}{2} + \mathtt{k}\right]$

6. Starting with an empty AVL tree, insert the items A, B, F, E, D, C in that order. Show each step, including the rotations.

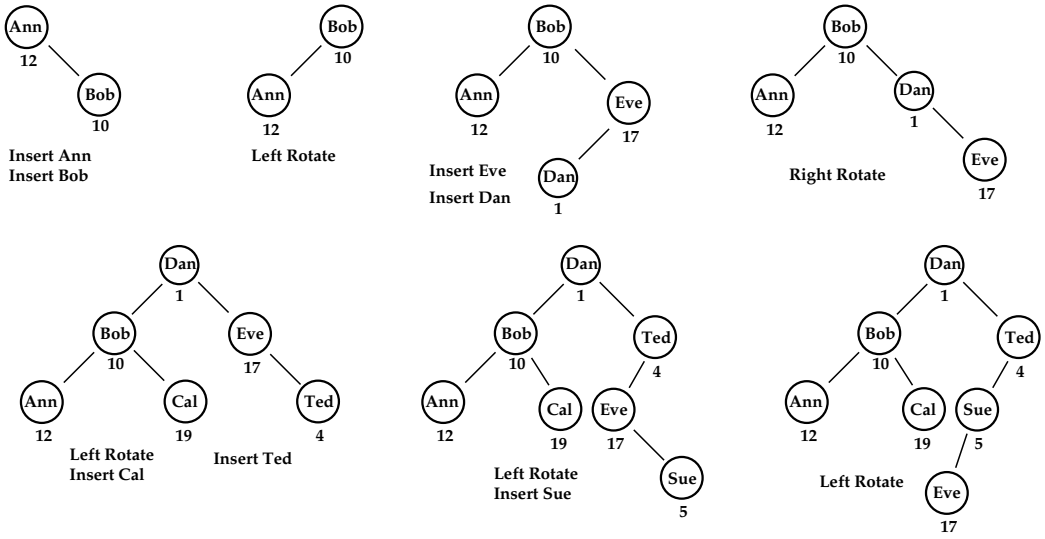
[1] "Sophisticated" plural of formula.



Start with an empty tree, then insert A, then B, as shown in (b).
Insert F, as shown in (c). The tree is unbalanced.
Left rotation at A restores the AVL tree, as shown in (d).
Insert E as shown in (e), then insert (d) as shown in (f). The tree is now unbalanced.
Right rotation at F, shown in (g) restores the AVL property.
Insert C as shown in (h). The tree is now unbalanced.
A left rotation at E fails to rebalance the tree. We need a double rotation. First, a right rotation at D, giving us (i), which is still unbalanced, followed by a left rotation at E, as shown in (j). The AVL property is restored.

7. A treap is a binary tree where the nodes are ordered in two ways. Each item has a *value* (say "Sam") and a randomly chosen *priority*. The nodes are ordered so that the values are alphabetic, as in a normal binary search tree, and the priorities are in heap order, just as in the binary tree implementation of a heap. Create a heap with the following items, inserted in the order given, where the priority of each item is given in the second column. For ease of grading, make it a min-heap. Show the steps, including the rotations.

Ann	12
Bob	10
Eve	17
Dan	1
Cal	19
Ted	4
Sue	5



8. Find a page on the internet which explains the Collatz conjecture. For any positive number n, let f(n) = n/2 if n is even, 3n+1 if n is odd. The conjeccture is that, if you start with any positive integer, you will reach 1 after applying f finitely many times. That number of times is called the *total stopping time* of n.

   For example, f(10) = 5, since the sequence is 10, 5, 16, 8, 4, 2, 1. Here is a recursive function which computes total stopping time:

```
int TST(int n)
  {
   assert(n > 0);
   if (n == 1) return 0;
   else if (n%2) return TST(3*n+1);
   else return TST(n/2);
  }
```

   Describe a program to find the total stopping times of all positive integers from 1 to 100, using memoization. Why is memoization better than either recursion or dynamic programming for this problem? Hint: try computing TNT(27).

   We need a search structure which stores memos. We can assume that each memo stores TST of one number, such as: "TST(8) = 3." Here is our function:

int TST(int n)
    if (TST(n) has been computed) let **result** be the memoized value of TST(n)
  else
    {
     if(n = 1) **result = 0**
     else if(n is even) **result** = TST(n/2)
     else **result** = TST(3*n+1)
     store the memo "TST(n) = **result**"
    }
  return **result**

   Our program is simply:
for (n = 1 to 100) write TST(n)

   We expect recurrence to take roughly 3000 steps, while the memoization program calls TST about 500 times. Ordinary dynamic programming is impractical, since we do not know the subprograms in advance, and even if we did, we would not know the topological order of the subprograms.