University of Nevada, Las Vegas Las Vegas Computer Science 477/677 Fall 2019

Answers to Assignment 7 Due Monday November 4, 2019

Name:____

1. Design a dynamic programming algorithm that inputs a sequence of integers, and finds that subsequence which has the largest total, subject to the condition that no two consecutive terms of the original sequence are in the subsequence. Identify the subproblems.

   For example, if the input sequence is 3,1,4,1,5,9,2,6,5,3,5, your algorithm will find the subsequence 3,4,9,6,5, whose terms add up to 27.

   We say that a subsequence is *legal* if no two terms of the subsequence are consecutive terms of the input sequence.

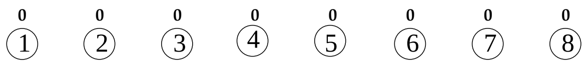
   Let $x[1 \ldots n]$ be the input sequence. There is one subproblem for each $i \leq n$, namely to find the maximum weight legal subsequenc which ends at $x[i]$. Let $s[i]$ be the weight of that subsequence, and let $b[i]$ be the index of the next-to-the-last term in that subsequence, the backpointer. If the first term of the subsequence is $x[i]$, we let $b[i] = 0$. The table shows the values computed by the algorithm for the example input sequence.

i	1	2	3	4	5	6	7	8	9	10	11
$x[i]$	3	1	4	1	5	9	2	6	5	3	5
$s[i]$	3	1	7	4	12	16	14	22	21	25	27
$b[i]$	0	0	1	1	3	3	5	6	6	8	8

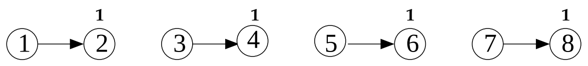
```
s[1] = x[1];
b[1] = 0;
s[2] = x[2];
b[2] = 0;
s[3] = s[1]+x[3];
b[3] = 1;
for(int i = 4, i <= n, i++)
 if(s[i-2]+x[i] > s[i-3]+x[i])
  {
   s[i] = s[i-2]+x[i];
   b[i] = i-2;
  }
 else
  {
   s[i] = s[i-3]+x[i];
   b[i] = i-3;
  }
// now write the subsequence
int j = n;
while(j > 0)
 {
  write(x[j]);
  j = b[j];
 }
```

2. Work problem 5.11 on page 149 of your textbook. Your answer should consist of a sequence of pictures. The actions are:

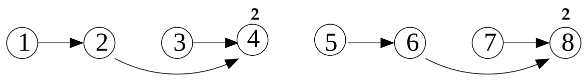
   `union(1,2) union(3,4) union(5,6) union(7,8) union(1,4)`
   `union(6,7) union(7,8) union(1,4) union(4,5) find(1)`



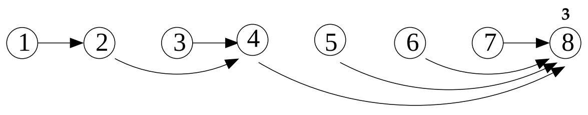
Initially, each node is in a separate tree.



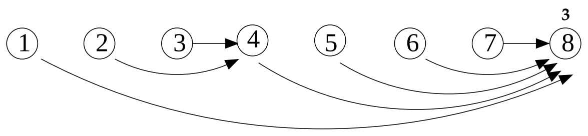
Execute the first four actions.



Execute the next two actions.



When `union(4,5)` is executed, the parent of 5 changes to 8 by path compression.



When `find(1)` is executed, the parent of 1 changes to 8 by path compression.

3. Work problem 6.1 on page 177 of your textbook.

The book says to find a linear time algorithm. However, that is tricky. There is a simple $O(n^3)$-time algorithm, there is a not quite so simple $O(n^2)$-time algorithm, a much more sophisticated $O(n \log n)$-time algorithm, and finally a really clever $O(n)$-time dynamic programming algorithm. Do the best you can.

Let $x[0], x[1], \ldots x[n-1]$ be the original sequence, and let $S[i,j] = \sum_{k=i}^{j} x[k]$, the sum of the i^{th} through j^{th} terms.

The "dumb" algorithm is to spend (n) time to compute each $S[i,j]$, and then to spend $O(n^2)$ time to find the largest of these. The time complexity of the dumb algorithm is $\Theta(n^3)$.

```
int maxint(int a, int b)
 {
  if (a < b) return b;
  else return a;
 }
int S(int i, int j)
 {
  assert(i <= j);
  int s = 0;
  for(int k = 1; k <= j; k++)
   s = s+x[k];
  return s;
 }
int rslt = 0;  // the empty subsequence
               // has total 0
for(int i = 0; i < n; i++)
 for(int j = i; j < n; j++)
  rslt = maxint(rslt,S(i,j));
cout << rslt << endl;
```

A slightly smarter algorithm makes use of the fact that $S[i,j+1] = S[i,j] + x[j]$. It is not actually necessary to save the values of $S[i,j]$; instead, when each is computed, we compare it to the maximum we have found so far. The time complexity is $\Theta(n^2)$.

```
int maxint(int a, int b)
 {
  if (a < b) return b;
  else return a;
 }
int rslt = 0;
for(int i = 0; i < n; i++)
 {int s = 0;
  for(int j = i; j < n; j++)
   {
    s = s + x[j];
    rslt = maxint(rslt,s);
   }
  }
cout << rslt << endl;
```

For the linear time dynamic programming algorithm, there are $2n$ subproblems. Let $A[k] = \max\{S[i,j] : i \leq j \leq k\}$ and $B[k] = \max\{S[i,k] : i \leq k\}$. Then $A[0, n-1]$ is the result. We have:

$B[0] = x[0]$
$A[0] = \max\{x[0], 0\}$
$B[k] = x[k] + \max\{0, B[k-1]\}$
$A[k] = \max\{A[k-1], B[k]\}$

Compute $B[k]$, then $A[k]$ for all k.

The time complexity is $\Theta(n)$.

```
int A[n];
int B[n];
B[0] = x[0];
A[0] = maxint(0,x[0]);
for(int k = 1; k < n; k++)
 {
  B[k] = x[k] + maxint(0,B[k-1]);
  A[k] = maxint(A[k-1],B[k]);
 }
cout << A[n-1] << endl;
```

4. Work problem 6.4 on page 178 of the textbook.

We say a substring is *good* if it can be subdivided into English words. Our algorithm has three phases. During the first phase, the Boolean array `good` is computed using dynamic programming. `good[i]` means that $s[1 \ldots i]$ is good. The computation of `good[i]` takes $O(i)$ time. During the first phase, the integer array `back` is also computed. `back[i]` is defined if and only if `good[i]`; `back[i] = j` means that $s[1 \ldots j]$ is obtained by deleting the last substring which is an English word from $s[1 \ldots i]$. The first phase takes $\Theta(n^2)$ time. During the second phase, the Boolean array `break` is computed. starting with `back[n]`, following the chain of backpointers, `break[i]` if and only if i is a backpointer. This phase takes $O(n)$ time. During the third phase, the string of words is written, provided `good[i]`. This phase takes $\Theta(n)$ time. The time complexity of our algorithm is $\Theta(n^2)$. The worst case time complexity of any algorithm for this problem is $\Omega(n^2)$ since, in the worst case, `dict` must be called for each of the $\binom{n}{2} = \Theta(n^2)$ substrings. Thus, our algorithm is optimal. We show the arrays constructed with the 25 symbol string **attackingothersisterrible**. (I found 17 English word substrings. Not all of them play a role when the algorithm is executed.) The solution found by the algorithm is **attack in got hers is terrible**.

		a	t	t	a	c	k	i	n	g	o	t	h	e	r	s	i	s	t	e	r	r	i	b	l	e
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
good	T	T	T	F	F	F	T	F	T	T	F	T	F	T	T	T	F	T	F	F	T	F	F	T	F	T
back		0	0				2		6	2	8	8		10	11	11		15			14			20		17
break	T	F	T	F	F	F	T	F	T	F	F	T	F	F	F	T	F	T	F	F	F	F	F	F	F	F

```
bool good[n+1]{true};
for(int i = 1; j <= n; i++)
 {
  good[i] = false;
  for(int j = 0; j < i; j++)
  if(good[j] and dict[j+1,i])
   {
    good[i] = true;
    back[i] = j;
   }
 }
if(good[n])
 {
  bool break[n+1]{false};
  int k = n;
  while (k > 0)
   {
    k = back[k];
    break[k] = true;
   }
  for(int i = 1; i <= n; i++)
   {
    cout << s[i];
    if(break[i]) cout << " ";
   }
 }
else cout << "No solution";
cout << endl;
```

5. Work problem 6.11 on page 180 of your textbook. This is the LCS (longest common subsequence) problem. To find the longest common subsequence of string $x[1..m]$ and $y[1..n]$, let $z[i,j]$ be the length of the longest common subsequence of the substrings $x[1..i]$ and $y[1..j]$. We initialize $z[i,0] = z[0,j]$ for all i and j. For $i > 0$ and $j > 0$, we have two cases. If $x[i] \neq y[j]$, let $z[i,j] = \max\{z[i-1,j], z[i,j-1]\}$. If $x[i] = y[j]$, let $z[i,j] = 1 + z[i-1,j-1]$. We now find LCS(`abrahamlincoln,georgewashington`). To avoid losing track, we first create an array which indicates those (i,j) for which $x[i] = y[j]$. We then compute the array z. The longest common substring is `rahion`.

	g	e	o	r	g	e	w	a	s	h	i	n	g	t	o	n
a								x								
b																
r				x												
a								x								
h										x						
a								x								
m																
l																
i											x					
n												x				x
c																
o			x												x	
l																
n												x				x

	g	e	o	r	g	e	w	a	s	h	i	n	g	t	o	n
a	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
b	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
r	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
a	0	0	0	1	1	1	1	2	2	2	2	2	2	2	2	2
h	0	0	0	1	1	1	1	2	2	3	3	3	3	3	3	3
a	0	0	0	1	1	1	1	2	2	3	3	3	3	3	3	3
m	0	0	0	1	1	1	1	2	2	3	3	3	3	3	3	3
l	0	0	0	1	1	1	1	2	2	3	3	3	3	3	3	3
i	0	0	0	1	1	1	1	2	2	3	4	4	4	4	4	4
n	0	0	0	1	1	1	1	2	2	3	4	4	4	4	4	4
c	0	0	0	1	1	1	1	2	2	3	4	4	4	4	4	4
o	0	0	1	1	1	1	1	2	2	3	4	4	4	4	5	5
l	0	0	1	1	1	1	1	2	2	3	4	4	4	4	5	5
n	0	0	1	1	1	1	1	2	2	3	4	5	5	5	5	6

Who is the only person known to have personally met both George Washinton and Abraham Lincoln?

6. Work problem 6.20 on page 182 of your textbook. This is a well-known problem with an interesting history. The "obvious" (but still tricky) dynamic programming algorithm, which involves finding optimal binary search trees for each contiguous subsequence of the list, takes $O(n^3)$ time. In 1971, Donald Knuth found an $O(n^2)$-time algorithm for the problem. The algorithm was correct, but Knuth's proof of correctness was flawed.

   "Experts" believe there should be a faster algorithm, but no one has found one.

   Given an alphabetic list of n items, $a_1 < a_2 < \ldots < a_n$ where a_k has *weight* w_k, the weighted path length of a binary search tree on those items is defined to be $\sum_{i=k}^{n} p_k$, where p_k is 1 if a_k is at the root of the tree, 2 if a_k is a child of the root, and so forth.

   items is optimal if its weighted path length is as small as possible. Every subtree of an optimal binary search tree is an optimal binary search tree. By bottom-up dynamic programming, we can compute the minimum WPL of the subtree on each of the $\binom{n+1}{n}$ contiguous sublists. The WPL of a subtree of one item is 0. If $1 \leq i \leq j \leq n$, we define $W[i,j] = \sum_{k=1}^{j} w_k$, the weight of the sublist $a_i \ldots a_j$. We also define $W[i,i-1] = 0$.

   Subtrees. Any subtree of an optimal binary search tree is an optimal binary search tree. Let $T[i,j]$ be the optimal binary search tree on the items $a_i \ldots a_j$. We let $T[i,i-1]$ be the empty tree. and let WPL$[i,j]$ be the WPL of $T[i,j]$, hence WPL$[i,i-1] = 0$.

   Dynamic Programming. There is one subproblem for each sublist, a total of $\frac{n(n+1)}{2}$ subproblems. Trivially, WPL$[i,i] = w_i$. The subproblems are worked in bottom-up order, that is, in order of the length of the sublist. If $i \leq j$, let a_r be the item at the root of $T[i,j]$. Then WPL$[i,j] =$ WPL$[i,r-1] +$ WPL$[r+1,j] + W[i,j]$. Thus, computation of WPL$[i,j]$ depends on choosing the correct r. Since we don't know r in advance, we try all of them. Our program looks like this:

```
WPL$[i,i-1] = 0$ for all $i$.
WPL$[i,i] = w_i$ for all $i$.
For i from n-1 downto 1
  For j from i+1 to n
    WPL$[i,j] = W[i,j] + \min\{\text{WPL}[i,r-1] + \text{WPL}[r+1,j] \ : \ i \leq r \leq j\}$
    Let $r[i,j]$ be the value of $r$ which is chosen.
```

   Then WPL$[1,n]$ is the weighted path length of $T[1,n]$. The shape of $T[1,n]$ can be recovered by keeping track of the $r[i,j]$.

   Knuth's Speedup. In 1971, Knuth noticed that $r[i,j-1] \leq r[i,j] \leq r[i+1,j]$ for all $i < j$, thus, instead of spending linear time to search for $r[i,j]$, we only need to search for values in the range $r[i,j-1] \ldots r[i+1,j]$. This property is called *monotonicity*. The amortized time for each such search is $O(1)$, and thus the dynamic program can be executed in $O(n^2)$ time.[1]

[1]Knuth's proof of monotonicity was several pages long, and so complex I could not understand it. I found out later that it was wrong, anyway! However, now there is a relatively simple, and correct, proof of monotonicity.