University of Nevada, Las Vegas Computer Science 477/677 Fall 2021

Answers to Assignment 7: Due Monday November 8, 2021 11:59 pm

In Problem 1 below, we assume that G is a weighted directed graph with vertices numbered 0 through n-1. We let $E[i,j]$ be the weight of the arc from vertex i to vertex j. If no such arc exists, $E[i,j] = \infty$ by default.

1. Consider the single source shortest path problem on a weighted digraph G. (digraph = directed graph) Fill in each blank with **one** word, or formula. Assume G has n vertices and m arcs (directed edges.) Assume that every answer (shortest path from the source to a vertex) has no more than p edges.

   (a) The simple dynamic programming algorithm requires that G be **acyclic**. In that case, the time complexity of the algorithm is $O(m)$

   (b) The Bellman-Ford algorithm requires that G have no **negative cycle**. In that case, the time complexity of the algorithm is $O(mn)$.

   (c) Dijkstra's algorithm requires that G have no **negative weight**. In that case, the time complexity of the algorithm is $O(m \log n)$

   (d) In order to work the simple dynamic programming algorithm for the single source shortest path problem, we must visit the vertices of G in **topological** order.

2. (a) Select a set of numbers from the list 2,6,3,1,8 which has the maximum total, but with the rule that you may not select any two consecutive numbers. (This is an instance of problem (b) below.)

   6,8

   (b) Write, in pseudocode, an algorithm which writes a maximum total subsequence of a sequence of numbers, with the rule that no two consecutive members of the sequence may both be selected.

   I actually intended to say that the numbers are non-negative, but I forgot. So, we'll do the problem with arbitrary numbers.

   Let $x[1], x[2], \ldots x[n]$ be the sequence.

   If all we want is the maximum sum of such a subsequence, we can find it easily with the following dynamic program, where $S[i]$ is the sum of the best subsequence of $x[1], \ldots x[i]$.

```
S[0] = 0;
S[1] = max(0,x[1])
for(int i = 2; i <= n; i++)
  S[i] = max(S[i-1],S[i-2]+x[i]);
cout << S[n] << endl;
```

   If we want to actually print out the subset, and in the original correct order, it's a little trickier. Let $S[i]$ be the sum of the best subsequence of $x[i] \ldots x[n]$.

```
S[n+2] = 0;
S[n+1] = 0;
for(i = n; i > 0; i = i-1)
 S[i] = max(S[i+1],S[i+2]+x[i]);
// Note that S[i] >= S[i+1].
// Now write the best subsequence.

i = 1;
while(i <= n)
 {
  if(S[i] == S[i+1]) i++; // the best subseqence of x[i] ... x[n]
                          // does not contain x[i]
  else // S[i] > S[i+1] the best subsequence of x[i] ... x[n] contains x[i]
   {
    cout << x[i] << " ";
    i = i+2;
   }
 }
cout << endl;
```

3. Explain how to implement a sparse array using a search structure. (This **exact** problem will be on the next examination on November 15.) Hint: You must explicitly describe how to implement the operators **fetch** and **store**.

   Let A be the sparse array. Our search structure $\mathcal{S}$ contains ordered pairs of the form $(i, A[i])$. The key of the search structure is i, that is the first element of each pair. If there is no pair for some i, then $A[i] = 0$.

   Here is how we implement the sparse array. First, initialize $\mathcal{S}$ to be empty.

   We execute **fetch(i)** by searching $\mathcal{S}$ for a pair (i, x). If such a pair is found, fetch returns the value x; otherwise it returns 0.

   We execute **store(i,x)** by searching $\mathcal{S}$ for a pair (i, y) for any y. If such a pair is found, it is replaced by the pair (i, x). If it is not found, the pair (i, x) is inserted into $\mathcal{S}$.

4.
   ```
   int f(int n)
   {
    if(n<7) return 1;
    else return f(n/2)+f(n/2+1)+f(n/2+2)+f(n/2+3);
   }
   ```

   The function f(n) can be computed by recursion, as given in the C++ code above. However, we could also compute f(n) using dynamic programming or memoization.

   (a) What is the asymptotic time complexity of the recursive computation of f(n)? (You should be able to solve this problem using one of the theorems we've covered, but if you can't, try programming it for various values of n.)

   Let $T(n)$ be the time complexity of computing $f(n)$. We have the asymptotic recurrence: $T(n) = 4T(n/2) + 1$ By the master theorem, $T(n) = \Theta(n^2)$.

(b) What is the asymptotic time complexity of the dynamic programming computation of f (n)? (There is no excuse for not being able to figure this out without writing a program.)

Declare an array $f[i]$ for $0 \leq i \leq n$. The DP is

```
for(i = 1; i <= n; i++)
 if(i < 7) f[i] = 7;
 else f[i] = f[n/2]+f[n/2+1]+f[n/2+2]+f[n/2+3];
cout << f[n] << endl;
```

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

(c) What is the asymptotic time complexity of the computation of f (n) using memoization? (This is harder than the others. If you write a program and try various values of n, you need a range of large values, like 1024 and up, to get the picture. Remember: memoization uses a sparse array structure.)

The answer is $\Theta(\log n)$.

Let G be the directed graph whose vertices are the integers from 0 to n, Each integer m represents the obligation to compute f(m). There is an arc (i,j) if f(i) must be computed before f(j). Because of the non-recursive branch of the code, a number less than 7 has no in-neighbors. If m ¿ 7, m has four in-neighbors, namely m/2, m/2+1, m/2+2, m/2+3. The number of memos that must be stored is equal to 1 plus the number of predecessors of n. Here is an example. Let n = 1785. The memo (i,f(i)) must be computed and stored for each i which is either 1785 or a predecessor of 1785. We list those values of i.

1785, 892, 893, 894, 895, 446, 447, 448, 449, 450, 223, 224, 225, 226, 227, 228, 111, 112, 113, 114, 115, 116, 117, 55, 56, 57, 58, 59, 60, 61, 27, 28, 29, 30, 31, 32, 33, 13, 14, 15, 16, 17, 18, 19, 6, 7, 8, 9, 10, 11, 12, 4, 5.

Note that, except for a few numbers at the end of the list, the predecessors fall into "blocks" of size at most 7, where the first number in each block is n divided by a power of 2, that is, 1785, 892, 446, 223, etc. The number of blocks is approximately $\log_2$n, hence the number of memos needed is approximately 7$\log_2$n. $= \Theta(\log n)$.

Suppose we count the time required to search the set of memos. If, for example, we use a balanced binary tree, our search time is $O(\log n \log\log n)$, which dominates the time complexity of the algorithm.

5. Read the handout johnson.pdf, which describes Johnson's algorithm. Work the exercise given on the last page.

	A	B	C	D	E	F	G
A	0	3	5	1	0	1	0
		A	E	B	D	E	F
B	1	0	2	−2	−3	−1	1
	C		E	B	D	E	F
C	−1	2	0	0	−1	1	3
	C	*A*		*B*	*D*	*E*	*F*
D	3	6	4	0	−1	1	3
	C	*A*	*E*		*D*	*E*	*F*
E	4	7	5	5	0	2	4
	C	*A*	*E*	*B*		*E*	*G*
F	5	8	6	6	1	0	2
	C	*A*	*E*	*B*	*G*		*F*
G	3	6	4	4	−1	1	0
	C	*A*	*E*	*B*	*G*	*E*	