CS 677: Assignment 5

Terence Henriod

April 14, 2014

Abstract

In this assignment, various dynamic programming problems and related ones are presented.

1. Consider that you are the manager of a consulting team of expert computer programmers, and each week you have to select a job for them to work on. The jobs are of two categories: either *low-stress* (e.g. setting up a Web site for a small fundraising event), or *high-stress* (e.g., protecting a companys valuable patents). Each week, the main question with what type of job to take on: low-stress or high-stress.

   For a particular week i, choosing a low-stress job will earn you a revenue of $l_i > 0$ dollars, while for a high-stress job, you get a revenue of $h_i > 0$ dollars (high-stress jobs typically pay more). However, if the team works on a high-stress job in week i, they cannot do any job (of either type) in the previous week $i-1$ (they need that previous week to prepare for the high stress level). If they work on a low-stress job in week i, they can work on any job (of either type) in the previous week $i-1$.

   A *plan* for the team, is specified as a choice of *low-stress*, *high-stress* or *none*, for a sequence of n given weeks (with the constraint that if *high-stress* is selected for week $i > 1$, then *none* must be chosen for week $i-1$. (It is permitted to choose a high-stress job in week 1.) The revenue of the plan is computed as follows: for each week i, add l_i to the total if choosing *low-stress* in week i, and add h_i to the total if choosing *high-stress* in week i (add 0 if choosing *none* in week i.)

   The goal of the problem is that given a set of values $l_1, l_2, \ldots, l_n$ and $h_1, h_2, \ldots, h_n$ to find a plan of maximum value. Develop a dynamic programming algorithm that finds the value of an optimal plan using the steps outlined above.

   (a) Determine and **prove** the optimal substructure of the problem and write a recursive formula of an optimal solution (i.e., define the variable that you wish to optimize and explain how a solution to computing it can be obtained from solutions to subproblems). **Submit**: the recursive formula, along with definitions and explanations on what is computed.

   Solution: In this problem we wish to solve for the maximum revenue of a plan for the team for a given number of weeks i, call it $P[i]$. If $P[i]$ is the maximum revenue that is attainable for a planning period of i weeks, then $P[i-1]$ would be the optimal revenue that could be obtained for all weeks leading up to week i. Clearly, since some plan $P[i-1]$'s is optimal, its revenue can only be improved by taking a job with revenue, that is, not resting, in week i. However, the choice of job allowed in week i depends on the choice in week $i-1$, that is, a high-stress can only be chosen in week i if rest was chosen in week $i-1$, which of course is not an optimal solution. This is why we must also consider the optimal solution that involves resting in week $i-1$, or $P[i-2]+0$ so that we can include high-stress jobs in our consideration. By considering both ways to increase revenue in week i it is possible to optimize the revenue earned for a plan of i weeks. For completeness, we can consider all weeks previous to the first one as "rest" weeks where no revenue was earned, so the job with the highest revenue should be initially chosen. Thus, for $i = 1, 2$, $P[i-1]$ and $P[i-2]$ have trivial solutions.

   If it was not clear yet, the previous logic has led us to the "optimal sub-structure" of the problem. Assuming that optimal revenue plans have been found for $P[i-1]$ and $P[i-2]$, the optimal solution $P[i]$ will build on them. If there were better solutions that produced larger revenue than $P[i-1]$ and $P[i-2]$, then they would be better than the optimal solutions, which of course is not possible (if only because we would select those solutions instead.
   Formally, the optimal solution can be characterized as:

$$P[i] = \begin{cases} max[l_1, h_1] & \text{if } i = 1 \\ max[P[i-1] + l_i, P[i-2] + h_i] & \text{if } i \geq 2 \end{cases}$$

(b) Write an algorithm that computes an optimal solution to this problem, based on the recurrence above. Implement your algorithm in C/C++ and run it on the following values:

	Week 1	Week 2	Week 3	Week 4
l	10	1	10	10
h	5	50	5	1

Submit:
- A printed version of the algorithm
- A printout of the table that contains the solutions to the subproblems, run on the values given above (print the entire table!)

Solution:
Code:

```
enum JobType
{
  LOW_STRESS = 0,
  HIGH_STRESS,
  REST
};

int main( int argc, char** argv )
{
  // variables
  int job_payouts[2][5] = {{0, 10, 1, 10, 10}, {0, 5, 50, 5, 1}};
  int solutions[3][5];  // is save results of all 3 choices instead of
  int choices[3][5];    // P[i-1] and P[i-2] so I can reference Rest(P[i-1])

/* I know that this is rather inflexible, but if I had my way, I would do
   it all with vectors or structs that might increase difficulty of grading,
   use stacks instead of recursion, etc. Instead I settled for hacking it
   together to make things fit the assignment problems better.
 */

  // test the solution
  std::cout << "Solution: " << std::endl;
  findOptimalRevenue(job_payouts, solutions, choices );
  std::cout << std::endl << std::endl;

  // end the program
  return 0;
}


int findOptimalRevenue( int job_payouts[2][5], int solutions[3][5] )
{
/* NOTE: even though the recursive formula uses the optimal choice from two
```

```
       weeks previous, I will just keep a running "rest" choice row in the
       table which will be used to denote the best optimum solution up to the
       two weeks previous.
*/

 // variables
 std::pair<int, int> previous_result;
 std::pair<int, int> low_stress_option;
 std::pair<int, int> high_stress_option;
 int i = 0;

 // setup the "zero" week of choices
 for( i = 0; i <= REST; ++i )
 {
   // set the revenue earned so far to zero and the choice to rest
   solutions[i][0] = 0;
 }

 // go through the choices for each week considering optimum solutions to
 // previous weeks
 for( i = 1; i < 5; ++i )
 {
   // find the optimum result of choosing a low stress job
   previous_result = findOptimumChoice( solutions, (i - 1) );
   solutions[LOW_STRESS][i] = previous_result.first +
                              job_payouts[LOW_STRESS][i];

   // fill in the rest/do nothing portion of the table(s)
   solutions[REST][i] = previous_result.first; // + 0

   // find the optimum result of choosing a high stress job
   solutions[HIGH_STRESS][i] = solutions[REST][i - 1] +
                               job_payouts[HIGH_STRESS][i];
 }

 // get the information associated with the optimum revenue for the whole set
 previous_result = findOptimumChoice( solutions, i - 1 );

 // print the resulting optimum choice value
 printf( "(Part B) The highest revenue plan will generate an "
         "income of: %d\r\n", previous_result.first );  // Part B

 // print the results table that displays the values used in finding
 // the optimal solution
 printf( "(Part B) The revenue outcomes for each choice of each week:\r\n" );
 printRevenuesTable( solutions ); // Part B
 printf( "\r\n" );

 // return the optimal plan value
 return previous_result.first;
```

```
}

std::pair<int, int> findOptimumChoice( int weekly_revenues[3][5], int week_i )
{
  //variables
  std::pair<int, int> optimum_choice( NO_VALUE, REST );
  int job_choice = 0;

  // initialize the pair
  optimum_choice.first = NO_VALUE;
  optimum_choice.second = REST;

  // check every option to see which one has the highest value
  for( job_choice = 0; job_choice <= REST; ++job_choice )
  {
    // case: we have found a better solution, use it and keep track of
    //       which decision resulted in this one
    if( optimum_choice.first < weekly_revenues[job_choice][week_i] )
    {
      // update the optimum choice
      optimum_choice.first = weekly_revenues[job_choice][week_i];
      optimum_choice.second = job_choice;
    }
  }

  // return the resulting pair
  return optimum_choice;
}


void printRevenuesTable( int solutions[3][5] )
{
  // variables
  int i = 0;

  // print the top of the table
  printf( "%11s  ", "Week" );
  for( i = 0; i < 4; ++i )
  {
    printf( "%2d  ", i );
  }
  printf( "\r\n" );

  // print a divider
  for( i = 0; i < 28; ++i )
  {
    printf( "-" );
  }
  printf( "\r\n" );
```

```
  // print the low stress row of the table
  printf( "%11s |", "Low-stress" );
  for( i = 1; i < 5; ++i )
  {
    printf( "%2d  ", solutions[LOW_STRESS][i] );
  }
  printf( "\r\n" );

  // print the high stress row of the table
  printf( "%11s |", "High-stress" );
  for( i = 1; i < 5; ++i )
  {
    printf( "%2d  ", solutions[HIGH_STRESS][i] );
  }
  printf( "\r\n" );

  // print the rest stress row of the table
  printf( "%11s |", "Rest" );
  for( i = 1; i < 5; ++i )
  {
    printf( "%2d  ", solutions[REST][i] );
  }
  printf( "\r\n" );

  // no return - void
}
```

Output:

```
Solution:
(Part B) The highest revenue plan will generate an income of: 70
(Part B) The revenue outcomes for each choice of each week
(assuming the optimal choice was made in the week prior):
       Week   1   2   3   4
---------------------------
 Low-stress |10  11  60  70
High-stress | 5  50  15  51
       Rest | 0  10  50  60
```

(c) Update the algorithm you developed at point (b) to enable the reconstruction of the optimal solution, i.e., which jobs were selected in an optimal solution for the sequence of 4 weeks. (Hint: use an auxiliary table like we did in the examples in class.) Include these updates in your algorithm implementation from point (b).

Submit: - A printed version of the algorithm
- A printout of the values that you obtain in the table containing the additional information needed to reconstruct the optimal solution, run on the values given above (print the entire table!)

Solution:
Code:

```
int findOptimalRevenue( int job_payouts[2][5], int solutions[3][5],
                        int choices[3][5] )
{
/* NOTE: even though the recursive formula uses the optimal choice from two
         weeks previous, I will just keep a running "rest" choice row in the
         table which will be used to denote the best optimum solution up to the
         two weeks previous.
 */

  // variables
  std::pair<int, int> previous_result;
  std::pair<int, int> low_stress_option;
  std::pair<int, int> high_stress_option;
  int i = 0;

  // setup the "zero" week of choices
  for( i = 0; i <= REST; ++i )
  {
    // set the revenue earned so far to zero and the choice to rest
    solutions[i][0] = 0;
    choices[i][0] = REST;  // part C
  }

  // go through the choices for each week considering optimum solutions to
  // previous weeks
  for( i = 1; i < 5; ++i )
  {
    // find the optimum result of choosing a low stress job
    previous_result = findOptimumChoice( solutions, (i - 1) );
    solutions[LOW_STRESS][i] = previous_result.first +
                               job_payouts[LOW_STRESS][i];
    choices[LOW_STRESS][i] = previous_result.second;  // part C

    // fill in the rest/do nothing portion of the table(s)
    solutions[REST][i] = previous_result.first; // + 0
    choices[REST][i] = previous_result.second;  // part C

    // find the optimum result of choosing a high stress job
    solutions[HIGH_STRESS][i] = solutions[REST][i - 1] +
                               job_payouts[HIGH_STRESS][i];
    choices[HIGH_STRESS][i] = REST;  // part C
  }

  // get the information associated with the optimum revenue for the whole set
  previous_result = findOptimumChoice( solutions, i - 1 );
```

```
  // print the resulting optimum choice value
  printf( "(Part B) The highest revenue plan will generate an "
          "income of: %d\r\n", previous_result.first );  // Part B

  // print the results table that displays the values used in finding
  // the optimal solution
  printf( "(Part B) The revenue outcomes for each choice of each week:\r\n" );
  printRevenuesTable( solutions ); // Part B
  printf( "\r\n" );

  // print the choices prior to each week that led to the optimal solution
  printf( "(Part C) The choices made prior to each week:\r\n" );
  printPreviousChoicesTable( choices );  // Part C
  printf( "\r\n" );

  // return the optimal plan value
  return previous_result.first;
}


void printPreviousChoicesTable( int choices[3][5] )
{
  // variables
  int i = 0;

  // print an explanation of the table
  printf( "  Values in cells indicate which job was chosen the week "
          "prior to week i.\r\n" );
  printf( "  Row labels to the right indicate the type of job choice "
          "for the week i.\r\n" );

  // print the top of the table
  printf( "%11s  ", "Week" );
  for( i = 0; i < 4; ++i )
  {
    printf( "%11d  ", i );
  }
  printf( "\r\n" );

  // print a divider
  for( i = 0; i < 70; ++i )
  {
    printf( "-" );
  }
  printf( "\r\n" );

  // print the low stress row of the table
  printf( "%11s |", "Low-stress" );
  for( i = 1; i < 5; ++i )
  {
```

```
    printf( " " );
    printJobType( choices[LOW_STRESS][i] );
    printf( " " );
  }
  printf( "\r\n" );

  // print the high stress row of the table
  printf( "%11s |", "High-stress" );
  for( i = 1; i < 5; ++i )
  {
    printf( " " );
    printJobType( choices[HIGH_STRESS][i] );
    printf( " " );
  }
  printf( "\r\n" );

  // print the rest stress row of the table
  printf( "%11s |", "Rest" );
  for( i = 1; i < 5; ++i )
  {
    printf( " " );
    printJobType( choices[REST][i] );
    printf( " " );
  }
  printf( "\r\n" );

  // no return - void
}


void printJobType( int job_type_code )
{
  // case: the choice was a low-stress job
  if( job_type_code == LOW_STRESS )
  {
    printf( "%11s", "Low-Stress" );
  }
  // case: the choice was a high-stress job
  else if( job_type_code == HIGH_STRESS )
  {
    printf( "%11s", "High-Stress" );
  }
  // case: the choice was to rest
  else
  {
    printf( "%11s", "Rest" );
  }

  // no return - void
}
```

Output:

```
Solution:
(Part B) The highest revenue plan will generate an income of: 70
(Part B) The revenue outcomes for each choice of each week
(assuming the optimal choice was made in the week prior):
       Week   1   2   3   4
---------------------------
 Low-stress |10  11  60  70
High-stress | 5  50  15  51
       Rest | 0  10  50  60

(Part C) The choices made prior to each week:
  Values in cells indicate which job was chosen the week prior to week i.
  Row labels to the right indicate the type of job choice for the week i.
       Week            1            2            3            4
-------------------------------------------------------------------------
 Low-stress |  Low-Stress   Low-Stress  High-Stress   Low-Stress
High-stress |        Rest         Rest         Rest         Rest
       Rest |  Low-Stress   Low-Stress  High-Stress   Low-Stress
```

(d) Using the additional information computed at point (c), write an algorithm that outputs which jobs have been selected in every week. Implement this algorithm in C/C+.
Submit: - A printed version of the algorithm - A printout of the solution to the problem given by the numerical values in point (b).

Solution:
Code:

```
int findOptimalRevenue( int job_payouts[2][5], int solutions[3][5],
                        int choices[3][5] )
{
/* NOTE: even though the recursive formula uses the optimal choice from two
         weeks previous, I will just keep a running "rest" choice row in the
         table which will be used to denote the best optimum solution up to the
         two weeks previous.
 */

  // variables
  std::pair<int, int> previous_result;
  std::pair<int, int> low_stress_option;
  std::pair<int, int> high_stress_option;
  int i = 0;

  // setup the "zero" week of choices
  for( i = 0; i <= REST; ++i )
  {
```

```
  // set the revenue earned so far to zero and the choice to rest
  solutions[i][0] = 0;
  choices[i][0] = REST;  // part C
}

// go through the choices for each week considering optimum solutions to
// previous weeks
for( i = 1; i < 5; ++i )
{
  // find the optimum result of choosing a low stress job
  previous_result = findOptimumChoice( solutions, (i - 1) );
  solutions[LOW_STRESS][i] = previous_result.first +
                             job_payouts[LOW_STRESS][i];
  choices[LOW_STRESS][i] = previous_result.second;  // part C

  // fill in the rest/do nothing portion of the table(s)
  solutions[REST][i] = previous_result.first; // + 0
  choices[REST][i] = previous_result.second;  // part C

  // find the optimum result of choosing a high stress job
  solutions[HIGH_STRESS][i] = solutions[REST][i - 1] +
                              job_payouts[HIGH_STRESS][i];
  choices[HIGH_STRESS][i] = REST;  // part C
}

// get the information associated with the optimum revenue for the whole set
previous_result = findOptimumChoice( solutions, i - 1 );

// print the resulting optimum choice value
printf( "(Part B) The highest revenue plan will generate an "
        "income of: %d\r\n", previous_result.first );  // Part B

// print the results table that displays the values used in finding
// the optimal solution
printf( "(Part B) The revenue outcomes for each choice of each week:\r\n" );
printRevenuesTable( solutions ); // Part B
printf( "\r\n" );

// print the choices prior to each week that led to the optimal solution
printf( "(Part C) The choices made prior to each week:\r\n" );
printPreviousChoicesTable( choices );  // Part C
printf( "\r\n" );

// print the choices made to reach the solution
printf( "(Part D) The choices that lead to maximum revenue at week 4:\r\n" );
printJobSelections( choices, 4, previous_result.second ); // Part D
printf( "\r\n" );

// return the optimal plan value
return previous_result.first;
```

```
}


void printJobSelections( int choices[3][5], int week_number, int final_choice )
{
  // variables

  //case: we still have not reached the beginning of the list
  if( week_number > 1 )
  {
    // get the selection for the previous week
    printJobSelections( choices, (week_number - 1),
                        choices[final_choice][week_number] );

    // print this week's selection
    printf( " -> Week %d: ", week_number );

    // print the type of job chosen
    printJobType( final_choice );
  }
  // case: we are at the first week
  else if( week_number == 1 )
  {
    // print this week's selection
    printf( "Week %d:", week_number );

    // print the type of job chosen
    printJobType( final_choice );
  }
}
```

Output:

```
Solution:
(Part B) The highest revenue plan will generate an income of: 70
(Part B) The revenue outcomes for each choice of each week
(assuming the optimal choice was made in the week prior):
       Week   1   2   3   4
---------------------------
 Low-stress |10  11  60  70
High-stress | 5  50  15  51
       Rest | 0  10  50  60

(Part C) The choices made prior to each week:
  Values in cells indicate which job was chosen the week prior to week i.
  Row labels to the right indicate the type of job choice for the week i.
       Week            1            2            3            4
-------------------------------------------------------------------------
 Low-stress |  Low-Stress   Low-Stress  High-Stress   Low-Stress
```

```
High-stress |        Rest         Rest         Rest         Rest
       Rest |  Low-Stress   Low-Stress  High-Stress   Low-Stress
```

(Part D) The choices that lead to maximum revenue at week 4:
Week 1: Rest -> Week 2: High-Stress -> Week 3: Low-Stress -> Week 4: Low-Stress

2. Show how the algorithm MATRIX-CHAIN-ORDER discussed in class computes the number of scalar multiplications for the product of the following three matrices (i.e., give the values in table m as computed by the algorithm):
   A: size 4x3
   B: size 3x5
   C: size 5x2

   Solution: If one was to just give the table (with some of the work shown, of course), it would appear as:

	A	B	C
C	$(4 * 3 * 2) + 30 = 54$	$(3 * 5 * 2) + 0 = 30$	0
B	$(4 * 3 * 5) + 0 = 60$	0	XXX
A	0	XXX	XXX

   The final result is: $(A * (B * C))$.

   A second table could be used to indicate where parenthesizations should occur via inidices or some other mechanism. In this case, that table would indicate that A should be parenthesized separately from the result of $(B * C)$. This is why there is a +30 term in the AC cell - it indicates the cost of multiplying $B * C$. This also means that the $(4 * 3 * 2)$ part represents the cost of multiplying $A * BC$.

3. Indicate whether the following statements are true or false and justify your answers.

   (a) If X and Y are sequences that both begin with the character A, every longest common subsequence of X and Y begins with A.

   Solution: True. If X and Y both start with the character A, then the character A can be used as a perfix to the longest common subsequence of the sequences X' and Y' (the original sequences minus their first character) without detracting from any possible longest common subsequence of X and Y.

   (b) If X and Y are sequences that both end with the character A, some longest common subsequence of X and Y ends with A.

   Solution: True. By a similar argument as in part (a), adding the character A to the LCS of X" and Y" (sequences lacking the last character, A) will produce an LCS solution for X and Y.