1. Minimum-Sum Descent

(a) Give C++ pseudocode for a dynamic programming algorithm to solve this problem.

```
int n;
int graph = [n][n];

graph [0][0] = 2;
graph [1][0] = 5;
graph [1][1] = 4;
graph [2][0] = 1;
graph [2][1] = 4;
graph [2][2] = 7;
graph [3][0] = 8;
graph [3][1] = 6;
graph [3][2] = 9;
graph [3][3] = 6;

/*
n = 4;

0       2
1     5 4
2    1 4 7
3   8 6 9 6
*/

// Assuming graph has been initialized, and n was appropriately set

for (int i = n-2; i>=0; i--) {
        for (int j = 0; j<=i; j++) {
                if (graph[i+1][j] < graph[i+1][j+1]) {
                        graph[i][j] += graph[i+1][j];
                } else {
                        graph[i][j] += graph[i+1][j+1];
                }
        }
}

print("Minimum-Sum Descent is : %d\n", graph[0][0]);
```

(b) Apply your algorithm to the triangle in Figure 1.

Applying my algorithm to the provided example results in a minimum sum descent of 14, on the path (from top to bottom) of elements [0, 0, 0, 1]

(c) What is the time and space efficient of your algorithm?
This can be solved in quadratic time.
My algorithm results in a time cost of: n(n-2) from both loops
My algorithm results in a space efficiency cost of: 1. The algorithm itself has constant memory, as it is an in-place algorithm.

2. World Series Odds
Consider two teams, A and B, playing a series of games until one of the teams wins n games. Assume that the probability of A winning a game is the same for each game and equal to p and the probability of A losing a game is q = 1 – p. (Hence, there are no ties.) Let P(i, j) be the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series.

(a) Set up a recurrence relation for P(i, j) that can be used by a dynamic programming algorithm. Hint: In the situation where teams A and B need i and j games, respectively, to win the series, consider the result of team A winning the game and the result of team A losing the game.

P(i, j): P(0, j)=1 for $j > 0$, P(i, 0) = 0 for $i > 0$:
If A wins (a chance of probability p), A will need i – 1 more wins to win the world series.
B will need j wins.
If A looses (a chance of probability q = 1 – p), A will need i wins.
B will need j – 1 wins to win the world series.

This results in a recurrence of: $P(i, j) = pP(i - 1, j) + qP(i, j - 1)$, where $i \&\& j > 0$ and natural numbers.

(b) Find the probability of team A winning a seven-game series if the probability of it winning a game is 0.4. Hint: Set up a table with five rows ($0 \le i \le 4$) and five columns ($0 \le j \le 4$) and fill it by using the recurrence derived in part (a).

i / j	0	1	2	3	4
0	0	1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.6	0.18	0.32	0.46
4	0	0.03	0.9	0.18	0.29

(c) Write C/C++ pseudocode for a dynamic programming algorithm to solve this problem and determine its time and space efficiencies. Hint: Your pseudocode should be guided by the recurrence you set up in part (a).

```
winProbability(int n, int p) {
        int q = 1 - p;
```

```
        for (int j=1; j<n; j++)
                P[0, j] = 1;
        for (int i=1; i<n; i++)
                P[i, 0] = 0
                for (int j=1; j<n; j++)
                P[i, j] = (p * P[i-1, j]) + (q * P[i, j-1])
                        return P[n, n]
}
```

3. Knapsack Problem

Given <items> items of known weights $w_1,...,w_n$ and values $v_1,...,v_n$ and a knapsack of capacity <capacity>, the knapsack problem is to find the most valuable subset of the items that fit into the knapsack.

(a) Write C/C++ pseudocode for a bottom-up dynamic programming algorithm for the knapsack problem.

```
#include    <stdio.h>
#include    <iostream>

using namespace std;

int max (int a, int b);
int knapsack(int items, int capacity, int weights[], int values[]);

int sack[255][255] = {0}; // Picked elements (our final knapsack)

int sackPicked[255][255] = {0}; // Whether or not an element was picked (1 or -1
        respectively)

int main(){
  // This can be manually entered, obviously, but for demonstration it is initialized
        statically
  int items = 5;

  int capacity; // knapsack capacity
  cin >> capacity;

  int weights[5] = {3,2,1,4,5};

  int values[5] = {25,20,15,40,50};

  printf("Max knapsack value: %d\n", knapsack(items,capacity,weights,values));

  return 0;
}

int knapsack(int items, int capacity, int weights[], int values[]){
```

```
    for (int i=1; i <= items; i++){
        for (int j=0; j <= capacity; j++){
            if (weights[i-1] <= j){
                sack[i][j] = max(sack[i-1][j], (values[i-1]+sack[i-1][j-weights[i-1]]));
                if ((values[i-1] + sack[i-1][j-weights[i-1]]) > sack[i-1][j]) {
                    sackPicked[i][j] = 1;
                }
                else {
                    sackPicked[i][j] = -1;
                }
            }
            else {
                sackPicked[i][j] = -1;
                sack[i][j] = sack[i-1][j];
            }
        }
    }

    return sack[items][capacity];
}

// Utility method to return max of two integers
int max (int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}
```

(b) Apply your algorithm in part (a) to the following instance of the knapsack problem with a knapsack of capacity W = 6:
[Item, Weight, $Value]: (1, 3, $25), (2, 2, $20), (3, 1, $15) (4, 4, $40), (5, 5, $50)
Done, code shown in subproblem (a). After applying my algorithm to the supplied knapsack instance, I get a knapsack value of $65.

(c) How many different optimal subsets does the instance of part (a) have?
Three (3) optimal subsets were found in instance of part (a).

(d) In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for an instance of the knapsack problem?
By looking at our generated table, we can see that each row has subsets of the elements available to put in our knapsack, incrementing the number of items available to pick. The columns display combinations for sized knapsacks.
This can be observed to decide whether or not a subset is optimal.

4. The Bridge Crossing Problem

There are n > 1 people who want to cross a rickety bridge; they all begin on the same side. It is night, and they have one flashlight among them. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. The crossing times in minutes of the n people are $t_1, t_2, \ldots, t_n$, respectively. A pair must walk together at the pace of the slower person. The problem is to minimize the bridge crossing time.

(a) Give C/C++ pseudocode for a greedy algorithm to get all n people to cross the bridge, and determine how long it will take to cross the bridge by using your algorithm.

```
void bridgeCrossing() {
  // Bridge torch problem (cross two fastest)
  int n;
  cin >> n;
  int peopleHereTimes[n]; // array of people on starting side of bridge (value in array is
       their times to cross)
  int peopleThereTimes[n]; // array of people on finished side of bridge (value in array is
       their times to cross)

  int timer;

  for (int i=0; i<n; i++) { // For each person
       cin >> peopleHereTimes[i]; // Read in their time <tn>
  }

  // Sort the people by speed
  sort(peopleHereTimes);

  // Cross two fastest, take back our fastest, bring over the slowest, bring back out fastest
       of those, take over slowest, bring back fastest of those, etc..
  cross(peopleTimes[0], peopleTimes[1]); // Cross two fastest people, , they get removed
       from peopleHereTimes[] and put into peopleThereTimes[]
  timer+=peopleHereTimes[1];  // Record time of slowest crosser

  while (peopleHereTimes.length != 0) { // For the rest of the people, bring back fastest,
       bring two slowest
       bringBackFastest(peopleThereTimes); // bring back fastest person, they get
       removed from peopleThereTimes[] and put into peopleHereTimes[]
```

```
        cross(peopleHereTimes[peopleHereTimes.length],
        peopleHereTimes[peopleHereTimes.length-1]); // Cross two slowest people, they
        get removed from peopleHereTimes[] and put into peopleThereTimes[]
        timer+=peopleHereTimes[peopleHereTimes.length]; // Record time of slowest
        crosser
    }

}
```

(b) Apply your algorithm to an instance of the problem with n = 4 people, with crossing times of $t_1 = 1$ minute, $t_2 = 2$ minutes, $t_3 = 5$ minutes, and $t_4 = 10$ minutes, respectively.

Okay. Done. The runtime is 17 elapsed minutes. Below is the sequence

Elapsed Time	PeopleHereTimes[]	Description	PeopleThereTimes[]
0	1, 2, 5, 10	Starting	
2	5, 10	Cross two fastest	1, 2
3	1, 5, 10	Bring back fastest	2
13	1,	Cross two slowest	2, 5, 10
15	1, 2	Bring back fastest	5, 10
17		Cross two slowest	1, 2, 5, 10

(c) Does your algorithm yield a minimum crossing time for every instance of the problem? If it does then prove it; if it does not, find a small counterexample.
No, my algorithm does not give the minimum time in every instance. If there are repeated sequences of large numbers, it is better to use our fastest Person as the *carrier*.

For instance: PeopleHereTimes[] = {1, 5, 5, 5}

My algorithm moves the fastest two PeopleHereTimes[], bring back the fastest PeopleThereTimes[], then repeatedly brings the two slowest PeopleHereTimes[] and brings back the fastest PeopeThereTimes[].

This results in:

Elapsed Time	PeopleHereTimes[]	Description	PeopleThereTimes[]
0	1, 5, 5, 5	Starting	
5	5, 5	Cross two fastest	1, 5
6	1, 5, 5	Bring back fastest	5
11	1,	Cross two slowest	5, 5, 5
16	1, 5	Bring back fastest	5, 5
21		Cross two slowest	1, 5, 5, 5

The optimal algorithm would actually use our fastest People as the *carrier*, in this particular instance. It would result in:

Elapsed Time	PeopleHereTimes[]	Description	PeopleThereTimes[]
0	1, 5, 5, 5	Starting	
5	5, 5	Cross two fastest	1, 5
6	1, 5, 5	Bring back fastest	5
11	5,	Cross two fastest	1, 5, 5
12	1, 5	Bring back fastest	5, 5
17		Cross two fastest	1, 5, 5, 5

5. Scheduling Video Streams

Suppose n video streams need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits to be sent, at a constant rate, over a period of t_i seconds. Two streams cannot be sent at the same time; instead, a schedule must be determined, i.e., an order in which to send them. No matter the order, there cannot be any delays between the end of one stream and the start of the next. Assume the schedule starts at time zero (and ends at time $\sum_{i=1}^{n} t_i$, no matter the schedule), and that all the values b_i and t_i are positive integers. In addition, the link imposes the following constraint, using a fixed parameter r: For each natural number t > 0, the total number of bits sent over the time interval from 0 to t cannot exceed rt. A schedule is valid if it satisfies the constraint imposed by the link.

Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, the problem is to determine whether a valid schedule exists.

Example: Suppose there are n = 3 streams with (b_1, t_1) = (2000, 1), (b_2, t_2) = (6000, 2), and (b_3, t_3) = (2000, 1), and the link parameter is r = 5000. Then the schedule that runs the streams in the order 1, 2, 3 is valid because the constraint is satisfied: At t = 1: The whole first stream is sent; 2000 < 5000·1. At t=2: Half of the second stream i s sent; 2000 + 3000 < 5000·2. Similarly for t=3 and t=4.

(a) Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$. Prove this claim true or false.

This claim is false. This can be proven by example. With n=2 steams, (2, 1) and (1, 1000), if r = 1, then the first stream does not satisfy the claim. A valid schedule exists by ordering $stream_2$ before $stream_1$.

(b) Give C/C++ pseudocode for a greedy algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether a valid schedule exists.

```
int[] steamBits
int[] streamTime
int[] sbst        // bi/ti
int n, r;

void initializeSBST (streamBits, streamTime) {
   for (int i = 0; i < steamBits.length; i++) {
         int elementSBST = streamBits[i] / steamTime[i];
         sbst[i] = elementSBST;
   }
}

void orderStreams (streamBits, streamTime) {
   for (int i = 0; i < sbst.length; i++) {
   if (sbst[i] > sbst[i+1]) {
         //swap i  with i+1 (inplace with temp)
   }
}

void provideValidity(sbst) {
   if ( // conditions met) {
         return sbst;
   }
   else {
    // swap out of order streams
   }
}
```

/* Order streams in bi/ti order. Check validity of conditions as stated by problem. If two streams are out of order, greedily swap and recheck. If conditions are in order, our schedule order is found. */

(c) Clearly explain what makes your algorithm greedy.

A greedy algorithm is one which makes shortsighted decisions, based on conditionals without respect to the rest of the input or sequence. In this particular algorithm, the *greediness* comes from the fact that the streams are ordered by b_i/t_i, and checked to the specified conditions. When an out of order stream is found, swap them. This swapping is what makes the greediness.

(d) What is the worst case running time of your algorithm?

The worst case running time of the algorithm is polynomial in n.