The Canadian Airline Problem and the Bitonic Tour: Is This Dynamic Programming?

Article · January 2004

1 author:

Pedro Guerreiro
Universidade do Algarve
49 PUBLICATIONS **242** CITATIONS

SEE PROFILE

The Canadian Airline Problem and the Bitonic Tour: Is This Dynamic Programming?

Pedro Guerreiro

pg@di.fct.unl.pt, http://ctp.di.fct.unl.pt/~pg/

URLs of this document: http://ctp.di.fct.unl.pt/~pg/docs/canadian.pdf, http://ctp.di.fct.unl.pt/~pg/docs/canadian.htm

December 2003

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Table of Contents

The Canadian Airline Problem and the Bitonic Tour: Is This Dynamic Programming? 1
1 Introduction 1
2 Knapsack 2
3 Triangle 14
4 Indiana Jones 19
5 Mars Explorer 24
6 Longest Ascending Subsequence 27
7 Intermezzo: Fibonacci numbers 34
8 Bachet's Game 35
9 Game 31 41
10 Matrix Chain Multiplication 45
11 Longest Common Substring 48
12 Edit Distance 52
13 Euro Change 57
14 Lorries 59
15 Train 64
16 Printing Neatly 67
17 Trail 71
18 Bus stops 76
19 Scheduling 81
20 Pyramids 85
21 Bitonic Tour 91
22 Canadian Airline 100
23 Conclusion 109
Appendix: Utility Functions 110
References 113

The Canadian Airline Problem and the Bitonic Tour: Is This Dynamic Programming?

Pedro João Valente Dias Guerreiro

pg@di.fct.unl.pt http://ctp.di.fct.unl.pt/~pg

1 Introduction

In the 1993 International Olympiad in Informatics, held in Mendoza, Argentina, the forth problem became known as the Canadian Airline Problem: starting from the westernmost city in Canada, Vancouver, and using only the company's flights, travel eastward until you reach the easternmost city, Halifax, and then back to Vancouver, now always going westward, and never visiting a city that was visited on the eastbound leg (except Vancouver, of course). Compute the longest such tour.

When this problem was presented for acceptance to the international jury, someone observed that the problem seemed equivalent to the traveling salesman problem, which is NP-complete. To this the scientific committee in charge of the problems replied that the problem could be solved using dynamic programming in polynomial time. That settled the question and everybody was happy. (I realized later that, at that time, most jury members were not familiar at all with dynamic programming…)

Dynamic programming has since became a very much used technique in programming competitions such as the International Olympiad in Informatics (www.ioinformatics.org) and the ACM Collegiate Programming Contest (http://icpc.baylor.edu/icpc/), to the point that contestants assume that in each contest at least one of the problems will require it. On the other hand, dynamic programming is still considered an advanced topic: Knuth will cover it in Volume 4C of *The Art of Computer Programming*, due in 2007, http://www-cs-faculty.stanford.edu/~knuth/taocp.html [Knuth]. Sedgewick, in the original edition *Algorithms* presents the subject in chapter 37 [Sedgewick]. There are 40 chapters in the book and chapter 37 belongs to the last part, called Advanced Topics. Cormen *et al*., in *Introduction to Algorithms*, second edition, treat the subject earlier, in comparison, but also in a part called Advanced Design and Analysis Techniques [Cormen *et al*.]. Weiss, in his *Data Structures and Algorithm Analysis in C++*, devotes one section of chapter 10, entitled Algorithm Design Techniques, to dynamic programming. The book has 12 chapters [Weiss]. Wirth, in the most classic of all books on algorithms and data structures, *Algorithms + Data Structures = Programs*, does not mention dynamic programming, but uses it in the problem of building optimal search trees [Wirth]. At home, at the Computer Science degree of the New University of Lisbon (http://www.di.fct.unl.pt/lei0304/), dynamic programming is the very last topic in the course Algorithms and Data Structures II (http://ctp.di.fct.unl.pt/lei/aed2/), and is presented in the general section Advanced Techniques for Algorithm Design.

What is dynamic programming? What is so advanced about it?

The first thing to understand is that originally the "programming" in "dynamic programming" does not refer to computer programming, but to it being a tabular method. In fact, the main visible characteristic of dynamic programming is that we keep solutions of smaller problems in table, so as not to compute them again if the need arises. The second characteristic is that it often applies to optimization problems: the solution will give the optimal value of some property. For example, in the Longest Ascending Subsequence problem we are given an array of numbers and want to compute the longest ascending subsequence within the array of numbers. As we shall see in detail shortly, the technique is first to compute the length of the longest ascending sequence, and in the process fill the table, and finally use the table to compute the solution. What dynamic programming does not do is to generate all possible ascending sequences and then pick the longest.

In the Longest Ascending Sequence problem, the table is a vector: in position *x* we store the length of the longest ascending sequence that can be built using the first *x* elements of the given array. In two dimensional problems, the table is a matrix. That is the case of the Indiana Jones problem: Indiana Jones is trapped in the temple of fire and in order to escape he has to cross a rectangular room whose floor is made of square tiles, heated from underneath. Fortunately, he knows the temperature of each tile. Which path must Indiana take in order to get his feet burned as little as possible? When moving towards freedom, he can go one tile forward, or one tile diagonally to the left (provided he does not hit the side wall) or one position diagonally to the right (same observation for the other wall). In this problem, we first compute the minimal sum of temperatures in all possible paths, without generating all those paths. In the process we fill the matrix, storing in position <*x*, *y*> the best that Indiana Jones can do if he landed on that tile with a big leap from the border of the heated floor and then moved towards the exit side according to the above rules. The matrix having been filled, we can compute the optimal path.

In this report we visit some interesting dynamic programming problems, in order to illustrate the technique at work. These problems are taken from programming competitions or from textbooks and some are classics in this field. All of them are a lot of fun to program. All of them are very simple.

All the programs are written in C++. The complete sources can be freely downloaded from my web page: http://ctp.di.fct.unl.pt/~pg/docs/canadian_sources.zip.

We will discuss the following problems: Knapsack, Triangle, Indiana Jones, Mars Explorer, Longest Ascending Subsequence, Bachet's Game, Game 31, Matrix Chain Multiplication, Longest Common Substring, Edit Distance, Euro Change, Lorries, Train, Printing Neatly, Trail, Bus Stops, Scheduling, Pyramids and Bitonic Tour. We wind up with the original pretext for this exercise: the Canadian Airline problem.

2 Knapsack

A thief is robbing a house (or a safe, or a shop, or the police station) where there are plenty of things he can take, some more valuable, some more voluminous. He is dressed in black, carrying a torch and a knapsack on his back. Which items should he pick, if he is a competent thief? Of course he should select items that maximize the value of the loot but that he can carry in the knapsack, which has a finite capacity.

We are given the value and volume of each item and the capacity of the knapsack. There is an unlimited supply of each item. (The black outfit and the torch do not interfere in the solution. They are here just to set the right ambiance.)

Let us suppose there are five types of items with volumes 3, 4, 6, 7 and 9, and values 5, 7, 11, 13 and 16, respectively, and let us furthermore suppose that the capacity of the knapsack is 20. The burglar can fill his knapsack in many ways. For example: 4 items of volume 3 plus 2 items of volume 4, making up a loot of value 34; or 1 item of volume 3, 1 of volume 4, one of volume 6 and one of volume 7, making up a loot of value 36; etc. If he is greedy and chooses the most valuable items first, he will pick two items of value 16, for a total value of 32, but the knapsack would not be full, which shows that being greedy is not always being smart. If he is greedy but careful, he will pick one item of value 16, one item of value 13 and one item of value 7 for a total value of 36, filling up the knapsack. It would be smarter, however, to select two items of volume 7 and one of volume 6, for the value of the loot would be 37. Is there any one combination more valuable than this one?

Of course we could try all combinations with total volume less or equal to 20, but in general this would not be a good idea, because for arbitrary data the number of combinations could be huge.

On the other hand it is an acceptable development technique to follow two different paths, which we can use to assert their mutual correction, even if one of them will be dropped eventually, because the other leads to a much better solution.

Furthermore, when we set up test cases by hand, tailoring them to produce a given result, it may happen that we overlook some aspect of the data, and the solution is not as expected after all. By running the straightforward combinatorial solution we will probably catch those lapses earlier. (We can do this only for small test cases, of course. For large ones, the running time would be prohibitive).

Let us start defining a class Knapsack, with two vector data members, for the volumes of the items and for the values of the items, with an integer for the size of the problem (i.e., the number of types of items), and a function std::list<int> Solution(int x) returning the most valuable combination of items whose total volume is less or equal to x. For example, list <2, 0, 1, 4> means "take two items of type 0, zero of type 1, one of type 2, and four of type 3". The type of each item is represented by its index in the vectors.

```
class Knapsack {
public:
  int size;
  std::vector<int> volumes;
  std::vector<int> values;
public:
  virtual ~Knapsack()
  {
  }

  virtual std::list<int> Solution(int x) const
  {
    //…
  }
};
```

In this style of C++ programming, data members which are inputs or outputs of the algorithms are public. This is done for brevity, for it frees us from having to add access functions to those data.

We define all functions inline, also for brevity. All functions are virtual, because we shall be using inheritance and polymorphism. We include a virtual destructor, because all classes with virtual functions should have one. (These are C++ technical details: they are not essential in the discussion.)

We add a function Read that will read the data from a file (or, more precisely, from an input stream). In that file, the first line states the number of types of items, size, and each of the size following lines contains two numbers, the first indicating the volume of an item and the second its value:

```
class Knapsack {
//…
  virtual void Read(std::istream& input)
  {
    volumes.clear();
    values.clear();
    input >> size;
    volumes.reserve(size);
    values.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int s;
      int v;
      input >> s >> v;
      volumes.push_back(s);
      values.push_back(v);
    }
  }
};
```

Let us add immediately a function to compute the value, or cost, of a given combination, represented by a list:

```
class Knapsack {
//...
  virtual int Cost(const std::list<int>& x) const
  {
    int result = 0;
    for (std::list<int>::const_iterator i = x.begin(); i != x.end(); i++)
      result += values[*i];
    return result;
  }
};
```

We postpone the development of function Solution to digress on the brute force strategy, generating all possible combinations and then picking up the most valuable one, i.e., the one with maximum cost. Given a list of combinations, each represented by a list of integers, computing the most valuable one is rather straightforward:

```
class Knapsack {
//...
  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  // pre !x.empty()
  {
    std::list<std::list<int> >::const_iterator result;
    int max = 0;
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (max < temp)
      {
        max = temp;
        result = i;
      }
    }
    return *result;
  }
};
```

This function can only be called for non empty lists, as the precondition comment states.

We now have to generate all combinations. We shall do that in an inner class Combinations, which has a data member of type std::list<std::list<int> > where those combinations will be stored as they are computed:

```
class Knapsack {
//...
public:
  class Combinations {
  public:
    const Knapsack& k;
    std::list<std::list<int> > combinations;
    int capacity;
  private:
    std::list<int> temp;
    int volume;
  public:
    Combinations(const Knapsack& k, int capacity):
      k(k),
      capacity(capacity)
    {
    }

    virtual ~Combinations()
    {
```

```
    }

    virtual void Compute()
    {
      //…
    }

  private:
    void operator = (const Combinations&){}
  };
};
```

Member variable k is a reference to the knapsack object and the list combinations will contain all the combinations. The constructor merely initializes the reference and sets the capacity of the knapsack for which we are generating the combinations. We provide an empty private assignment operator to avoid a compiler warning “the assignment operator cannot be generated”. (It cannot be generated because the class has a const data member.)

The recursive algorithm is represented by a function Visit (reminiscent of the graph that implicitly underlies this computation). A call to Visit(x) adds x to the current combination, updates its volume (which by construction is not greater than the capacity), appends the current combination to the list of combinations and then recursively visits all item types greater or equal to x, provided one more item of that type still fits in the knapsack. The recursive calculation is started in function Compute:

```
class Knapsack {
//…
  class Combinations {
  //…
  public:
    //…
    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      volume = 0;
      for (int i = 0; i < k.size; i++)
        if (k.volumes[i] <= capacity)
          Visit(i);
    }

  private:
    virtual void Visit(int x)
    {
      temp.push_back(x);
      volume += k.volumes[x];
      combinations.push_back(temp);
      for (int i = x; i < k.size; i++)
        if (volume + k.volumes[i] <= capacity)
          Visit(i);
      volume -= k.volumes[x];
      temp.pop_back();
    }

    //…
  };
};
```

The combinatorial solution, i.e., the solution obtained by generating all combinations and then choosing the most valuable one, is computed by function SolutionCombinatorial in class Knapsack:

```
class Knapsack {
//…
```

```
  virtual std::list<int> SolutionCombinatorial(int x) const
  {
    std::list<int> result;
    Combinations c(*this, x);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }
};
```

Let us experiment this in a situation described above: five items with volumes 3, 4, 6, 7 and 9 and values 5, 7, 11, 13 and 16. This is the corresponding input file:

```
5
3 5
4 7
6 11
7 13
9 17
```

Here is a test function that computes the most valuable combination for all knapsack capacities up to 20 and its cost. It also shows all the combinations and their costs, just for fun:

```
void TestKnapsackCombinatorial(std::istream& input, std::ostream& output)
{
  Knapsack ks;
  ks.Read(input);
  int capacity = 20;
  const std::list<int>& solution= ks.SolutionCombinatorial(capacity);
  output << solution << "/" << ks.Cost(solution) << std::endl;
  output << std::endl;
  Knapsack::Combinations c(ks, capacity);
  c.Compute();
  for (std::list<std::list<int> >::const_iterator i = c.combinations.begin();
        i != c.combinations.end(); i++)
    output << *i << "/" << ks.Cost(*i) << std::endl;
}
```

We get the following output:

```
1 3 4/37

0/5
0 0/10
0 0 0/15
0 0 0 0/20
0 0 0 0 0/25
0 0 0 0 0 0/30
0 0 0 0 0 1/32
0 0 0 0 1/27
0 0 0 0 1 1/34
0 0 0 0 2/31
0 0 0 0 3/33
0 0 0 1/22
0 0 0 1 1/29
0 0 0 1 2/33
0 0 0 1 3/35
0 0 0 2/26
0 0 0 3/28
0 0 0 4/32
0 0 1/17
0 0 1 1/24
0 0 1 1 1/31
0 0 1 1 2/35
```

0 0 1 2/28
0 0 1 3/30
0 0 1 4/34
0 0 2/21
0 0 2 2/32
0 0 2 3/34
0 0 3/23
0 0 3 3/36
0 0 4/27
0 1/12
0 1 1/19
0 1 1 1/26
0 1 1 1 1/33
0 1 1 2/30
0 1 1 3/32
0 1 1 4/36
0 1 2/23
0 1 2 2/34
0 1 2 3/36
0 1 3/25
0 1 4/29
0 2/16
0 2 2/27
0 2 3/29
0 2 4/33
0 3/18
0 3 3/31
0 3 4/35
0 4/22
1/7
1 1/14
1 1 1/21
1 1 1 1/28
1 1 1 1 1/35
1 1 1 2/32
1 1 1 3/34
1 1 2/25
1 1 2 2/36
1 1 3/27
1 1 4/31
1 2/18
1 2 2/29
1 2 3/31
1 2 4/35
1 3/20
1 3 3/33
1 3 4/37
1 4/24
2/11
2 2/22
2 2 2/33
2 2 3/35
2 3/24
2 3 3/37
2 4/28
3/13
3 3/26
3 4/30
4/17
4 4/34

That's 84 combinations.

In order to write a list using operator <<, we must define this operator for generic lists. This is taken care of in our utilities library, which is presented in the Appendix.

So much for the combinatorial approach.

Let us now handle the problem in another fashion, through a new function int Optimal(int x, int y) that computes the maximum possible value of the loot if the thief uses a knapsack of capacity x and decides to take only items of the y first types:

```
class Knapsack {
//…
  virtual int Optimal(int x, int y) const
  {
    //…
  }
};
```

A moment of reflection (or two…) will tell us that perhaps the y-1-th item is not used in the combination that yields the optimal value, in which case Optimal(x, y) is the same as Optimal(x, y-1). (We are indexing the items from zero, so the first item is zero, and the last item, when there are y items, is y-1.) On the other hand, if one (or more) of the y-1-th items is used in the combination that yields the optimal value and we remove one such y-1-th element from the combination, the combination we obtain yields the optimal value for a knapsack of capacity x-sizes[y-1]. In this case, Optimal(x, y) is Optimal(x-volumes[y-1], y) + values[y-1]. Reversing this analysis and admitting that the values of Optimal(x, y-1) and Optimal(x-volumes[y-1], y) have already been computed, we conclude that the value of Optimal(x, y) is the maximum of these two values: Optimal(x, y-1) and Optimal(x-volumes[y-1], y) + values[y-1]. The second expression can only be used if x >= volumes[y-1]. The case where y is zero is trivial: whatever the capacity of the knapsack, the cost is zero. This leads us to the following rather compact recursive definition:

```
class Knapsack {
//…
  virtual int Optimal(int x, int y) const
  {
    int result = 0;
    if (y > 0)
      result = mas::Max(Optimal(x, y-1),
                        x >= volumes[y-1] ? values[y-1] + Optimal(x-volumes[y-1], y): 0);
    return result;
  }
};
```

The recursion is sound because the pair of arguments in the recursive calls is lexicographically less than the pair of arguments in the function call. (We are assuming that the sizes are all positive.)

Functions from the namespace mas, such as mas::Max above, are defined in the library presented in the Appendix.

We can reason in a slightly different way. We want to compute Optimal(x, y). In the loot that yields this value there can be zero items of type y-1, or one item, or two, etc. The maximum number of y-1 items is given by the quotient of capacity by volumes[y-1]. Suppose the right number of items of type y-1 is i. Then the value of the loot is Optimal(x – i * volumes[y-1], y-1) + i * values[y-1]. Note that Optimal(x - i * volumes[y-1], y-1) is the value of the optimal loot when the capacity of the knapsack is x - i * volumes[y-1] and only items of the y-1 types are used. We conclude that Optimal(x, y) is the maximum of Optimal(x – i * volumes[y-1], y-1) + i * values[y-1] for all possible values of i. This yields the following alternative function, with function Optimal2:

```
class Knapsack {
//…
```

```
  virtual int Optimal2(int x, int y) const
  {
    int result = 0;
    if (y > 0)
    {
      result = Optimal2(x, y-1);
      for (int i = 1; x >= i * volumes[y-1]; i++)
        result = mas::Max(result, i * values[y-1] + Optimal2(x - i * volumes[y-1], y-1));
    }
    return result;
  }
};
```

Now that we have a function (actually we have two) that gives the optimal cost for any knapsack and any number of items to use, how can we find the solution, i.e., the combination that yields that value? It's simple: we "undo" the computation of Optimal(x, y). If Optimal(x, y) is equal to Optimal(x, y-1) then the y-1-th is not used; otherwise, the solution will contain a y-1-th element, plus all the items in the solution for Optimal(x-size[y-1], y). Although this formulation is recursive, we can program it directly using a repetitive statement:

```
class Knapsack {
//…
  virtual std::list<int> Solution(int x) const
  {
    std::list<int> result;
    int i = x;
    int j = size;
    int z = Optimal(i, j);  // value of the loot.
    while (z > 0)
      if (z == Optimal(i, j-1))
        j--;  // do not use any more j items
      else
      {
        // use another j item
        result.push_front(j-1);
        i -= volumes[j-1];
        z -= values[j-1];
      }
    return result;
  }
};
```

Function Solution computes the optimal solution for a knapsack of capacity x using all the available items. Is this equivalent to the function SolutionCombinatorial presented before? Yes and no. Both functions calculate optimal solutions i.e., combinations of items that yield the maximum value. Function SolutionCombinatorial, however, is wired to compute the lexicographically first such combination, whereas function Solution constructs the lexicographically last. Therefore, if there is more than one optimal combination, the functions give different results.

Which of them is more efficient? Well, both functions are not very efficient. In both cases the number of recursive call is exponential on the size of the knapsack. Anyway, they are easy to understand, they match closely our intuition of the problem, and they can be readily experimented with simple examples. As an additional bonus, and not forgetting the programming adage "premature optimization is the root of all evil", function Optimal can be optimized by storing previously computed results in a table. If we had a table optimal such that optimal[x][y] contained the previously computed value of Optimal(x, y), then the right hand side of the assignment in the else branch in the definition of Optimal could be written as follows:

```
Max(optimal[x][y-1], x >= volumes[y-1] ? values[y-1] + optimal[x-volumes[y-1]][y]: 0)
```

Let's call this new function OptimalTabular, and let us define it in a new class KnapsackDynamic, derived from Knapsack:

```
class KnapsackDynamic: public Knapsack {
private:
  std::vector<std::vector<int> > optimal;
private:
  virtual int OptimalTabular(int x, int y) const
  {
    int result = 0;
    if (y > 0)
      result = mas::Max(optimal[x][y-1],
                        x >= volumes[y-1] ? values[y-1] + optimal[x-volumes[y-1]][y]: 0);
    return result;
  }
};
```

Here we mimicked the computation of Optimal. We could have done the same based on Optimal2:

```
class KnapsackDynamic: public Knapsack {
private:
  //…
  virtual int OptimalTabular2(int x, int y) const
  {
    int result = 0;
    if (y > 0)
    {
      result = optimal[x][y-1];
      for (int i = 1; x >= volumes[y-1]; i++)
        result = mas::Max(result, i * values[y-1] + optimal[x -= volumes[y-1]][y-1]);
    }
    return result;
  }
 };
```

Function OptimalTabular(x, y) can only be used provided the values optimal[x][y-1] and optimal[x-volumes[y-1]][y] are already available (and likewise for OptimalTabular2). We can guarantee that by filling the table upwards from the lower indices. Let us do that in a function void Compute(int x), whose argument represents capacity of the knapsack:

```
class KnapsackDynamic: public Knapsack {
//…
public:
  virtual void Compute(int x)
  {
    optimal.clear();
    optimal.resize(x+1);
    for (int i = 0; i <= x; i++)
    {
      optimal[i].resize(size + 1, 0);
      for (int j = 0; j < size + 1; j++)
        optimal[i][j] = OptimalTabular(i, j);
    }
  }
};
```

Function Optimal is now efficiently redefined using the table:

```
class KnapsackDynamic: public Knapsack {
//…
public:
  virtual int Optimal(int x, int y) const
  {
    return optimal[x][y];
  }
};
```

This is dynamic programming: using a table to store previously computed results of recursive functions so that those values do not have to be recursively computed again.

Here is a test function, for illustration:

```
void TestKnapsackDynamic(std::istream& input, std::ostream& output)
{
 int capacity = 20;
 KnapsackDynamic ksd;
 ksd.Read(input);
 mas::WriteLine(ksd.volumes, " ", output);
 mas::WriteLine(ksd.values, " ", output);
 output << std::endl;
 ksd.Compute(capacity);
 mas::WriteV(ksd.optimal, "\n", 3, output);
 output << std::endl;
 for (int i = 0; i <= capacity; i++)
 {
  std::list<int> s = ksd.Solution(i);
  output << i << ": " << s << "/" << ksd.Cost(s) << std::endl;
 }
}
```

We do not have to redefine function Solution in the derived class: it is available via inheritance.

Here's the output file. The first two lines merely echo the input values. Then we observe the optimal matrix. Finally, the optimal loots with the capacity varying from 0 to 20.

```
3 4 6 7 9
5 7 11 13 17

  0  0  0  0  0  0
  0  0  0  0  0  0
  0  0  0  0  0  0
  0  5  5  5  5  5
  0  5  7  7  7  7
  0  5  7  7  7  7
  0 10 10 11 11 11
  0 10 12 12 13 13
  0 10 14 14 14 14
  0 15 15 16 16 17
  0 15 17 18 18 18
  0 15 19 19 20 20
  0 20 21 22 22 22
  0 20 22 23 24 24
  0 20 24 25 26 26
  0 25 26 27 27 28
  0 25 28 29 29 30
  0 25 29 30 31 31
  0 30 31 33 33 34
  0 30 33 34 35 35
  0 30 35 36 37 37

0: /0
1: /0
2: /0
3: 0/5
4: 1/7
5: 1/7
6: 2/11
7: 3/13
8: 1 1/14
9: 4/17
10: 1 2/18
11: 1 3/20
12: 2 2/22
13: 2 3/24
14: 3 3/26
```

```
15: 2 4/28
16: 3 4/30
17: 1 2 3/31
18: 4 4/34
19: 2 2 3/35
20: 2 3 3/37
```

This is fine, but having to fill the table explicitly by calling function Compute is a bit annoying. As a matter of fact, we can use the table in a slightly different way, which dispenses its explicit initialization: initially we place –1 (or another "impossible" value) in all positions. When we compute Optimal(x, y) for the first time for a given pair of values x and y, we compute recursively using the inherited function, and in the end we store the result in the table. Thereafter, we use the stored value. This technique is called *memoization*, because the table is used as a memo. We implement it in a new class KnapsackMemo, also derived from Knapsack:

```
class KnapsackMemo: public Knapsack {
public:
  mutable std::vector<std::vector<int> > optimal;
public:
  virtual int Optimal(int x, int y) const
  {
    if (x >= static_cast<int>(optimal.size()))
      Grow(x);
    if (optimal[x][y] == -1)
      optimal[x][y] = Knapsack::Optimal(x, y);
    return optimal[x][y];
  }

private:
  virtual void Grow(int x) const
  {
    int z = static_cast<int>(optimal.size());
    optimal.resize(x+1);
    for (int i = z; i < static_cast<int>(optimal.size()); i++)
      optimal[i].resize(sizes.size() + 1, -1);
  }
};
```

The memo is implemented as a mutable matrix. It is mutable so that it can be modified in function Optimal, which is a const function. Function Grow resizes the matrix as required, filling the new positions with –1.

Classes KnapsackDynamic and KnapsackMemo use a matrix to store the values of function Optimal. Now, observe closely function OptimalTabular from class KnapsackDynamic. The right hand side of the assignment in the else branch uses only the current column of the matrix at an upper row, and the previous column at the same row. This means in each case we can live with only one column, provided we fill it from top to bottom, and then from left to right. From left to right is a matter of speaking, since we have only one column, but it means that this works as if we were filling the matrix column by column, from left to right. Let us implement this idea in a new class KnapsackDynamic1:

```
class KnapsackDynamic1: public Knapsack {
public:
  std::vector<int> optimal;
public:
  virtual int OptimalTabular(int x, int y) const
  {
    int result = 0;
    if (y > 0)
      result = mas::Max(optimal[x],
                        x >= volumes[y-1] ? values[y-1] + optimal[x-volumes[y-1]]: 0);
    return result;
  }
```

```
public:
  virtual void Compute(int x)
  {
    optimal.clear();
    optimal.resize(x+1);
    for (int j = 1; j < static_cast<int>(sizes.size()) + 1; j++)
      for (int i = 0; i <= x; i++)
        optimal[i] = OptimalTabular(i, j);
    }
  }
};
```

Note that OptimalTabular(x, y) will actually be the same as Optimal(x, y) only if the vector optimal has been filled y times, for successive values of y, i.e., for zero items used, for one item used, …, for y-1 items used. The final values in the array represent the values of Optimal when all the items are used. These will be available through function Optimal1:

```
class KnapsackDynamic1: public Knapsack {
//…
  virtual int Optimal1(int x) const
  {
    return optimal[x];
  }
};
```

This is a big improvement over the classes KnapsackDynamic and KnapsackMemo, since it uses much less memory: a vector instead of a matrix. As it stands, however, it does not redefine function Optimal, and, therefore it does not improve by inheritance function Solution. We must redefine this function using another technique. The idea is to record the item that was used to complete the optimal combination for each capacity. Using that information we can reconstruct the optimal combination: we include the item i that completes the optimal combination for capacity x, and then, recursively, all the items in the optimal combination for x-size[i]. For this, we add a vector completing in class KnapsackDynamic1, modify function Compute so that it initializes that vector, and redefine function Solution so that it takes advantage of it:

```
class KnapsackDynamic1: public Knapsack {
//…
public:

  virtual void Compute(int x)
  {
    optimal.clear();
    optimal.resize(x+1);
    completing.clear();
    completing.resize(x+1);
    for (int j = 1; j < size + 1; j++)
      for (int i = 0; i <= x; i++)
        if (optimal[i] < OptimalTabular(i, j))
        {
          optimal[i] = OptimalTabular(i, j);
          completing[i] = j;
        }
  }

  virtual int Optimal1(int x) const
  {
    return optimal[x];
  }

  virtual std::list<int> Solution(int x) const
  {
    std::list<int> result;
```

```
    int z = Optimal1(x);  // value of the loot.
    while (z > 0)       // stop when the sum of values of selected items is the value of the loot.
    {
      int j = completing[x];
      result.push_front(j-1);
      x -= volumes[j-1];
      z -= values[j-1];
    }
    return result;
  }
};
```

This concludes our development of the knapsack problem. In summary, we started with the combinatorial approach, as a warm up. Then we tried a recursive version, knowing it could not be very efficient. Next we introduced a table to record previously computed values, and filled it systematically. This was the dynamic programming solution. Alternatively, we used dynamic programming with a memo, i.e., with a table that was computed as necessary (and not filled initially from the bottom up as before). This is the general pattern we shall use in the problems in this report, even if in many cases we omit one or two aspects, for brevity. In the knapsack problem, we went one step further by modifying the dynamic solution in order to use a vector of previously computed values, instead of a matrix.

3 Triangle

We are given a triangle of numbers, all of them non negative integers, and we want to find the path from the top element to the bottom row that gives the highest sum. When going from one row to the next we can pick the number at the left or the number at the right. Here is an example:

```
    8
   4 9
  9 2 1
 3 8 5 5
5 6 3 7 6
```

This problem was used in the International Olympiad for Informatics held in Sweden, in 1994.

A greedy algorithm would pick the largest possible number on the way down, giving the sequence of numbers <8, 9, 2, 8, 6> for a total of 33. A combinatorial algorithm would try all paths from the top item downwards and choose the "most valuable" one. Let us experiment with these two approaches first, knowing that we will drop them afterwards. For the greedy solution we use a function SolutionGreedy. For the combinatorial solution we use the same technique as in the knapsack problem: an inner class for the recursively generating all combinations, a function for choosing the most valuable, and a function SolutionCombinatorial for actually doing the computations. Both functions, SolutionGreedy and SolutionCombinatorial have two arguments, x and y, indicating that the starting point for the problem is in line x column y. In this way, we can solve the problem for any subtriangle of the given triangle. Both functions return a list of integers, representing the column numbers of the solution on the way down from the starting point. Here is class Triangle:

```
class Triangle {
public:
  std::vector<std::vector<int> > numbers;
  int size;
public:
  virtual void Read(std::istream& input = std::cin)
  {
    input >> size;
    numbers.clear();
    numbers.resize(size);
    for (int i = 0; i < size; i++)
    {
```

```
      numbers[i].reserve(i+1);
      for (int j = 0; j <= i; j++)
      {
        int x;
        input >> x;
        numbers[i].push_back(x);
      }
    }
  }

  virtual int Cost(const std::list<int>& x) const
  {
    int result = 0;
    int z = size - 1;
    for (std::list<int>::const_reverse_iterator i = x.rbegin(); i != x.rend(); i++)
      result += numbers[z--][*i];
    return result;
  }

  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  // pre !x.empty()
  {
    std::list<std::list<int> >::const_iterator result;
    int max = 0;
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (max < temp)
      {
        max = temp;
        result = i;
      }
    }
    return *result;
  }

  virtual std::list<int> SolutionCombinatorial(int x, int y) const
  {
    std::list<int> result;
    Combinations c(*this);
    c.Compute(x, y);
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

  virtual std::list<int> SolutionGreedy(int x, int y) const
  {
    std::list<int> result;
    result.push_back(x);
    for (int i = x+1, j = y; i < static_cast<int>(numbers.size()); i++)
    {
      j += numbers[i][j] < numbers[i][j+1];
      result.push_back(j);
    }
    return result;
  }

public:
  class Combinations {
  public:
    const Triangle& t;
    std::list<std::list<int> > combinations;
  private:
```

```
    std::list<int> temp;
  public:
    Combinations(const Triangle& t):
      t(t)
    {
    }

    virtual ~Combinations()
    {
    }

    virtual void Visit(int x, int y)
    {
      temp.push_back(y);
      if (x == t.size - 1)
        combinations.push_back(temp);
      else
      {
        Visit(x+1, y);
        Visit(x+1, y+1);
      }
      temp.pop_back();
    }

    virtual void Compute(int x, int y)
    {
      temp.clear();
      combinations.clear();
      Visit(x, y);
    }

  private:
    void operator = (const Combinations&) {}
  };
};
```

Here is the test function:

```
void TestTriangle(std::istream& input, std::ostream& output)
{
  Triangle t;
  t.Read(input);
  mas::WriteV(t.numbers, "\n", " ", output);
  output << std::endl;
  std::list<int> g = t.SolutionGreedy(0, 0);
  output << t.Cost(g) << "/" << g << std::endl;
  std::list<int> c = t.SolutionCombinatorial(0, 0);
  output << t.Cost(c) << "/" << c << std::endl;
}
```

For an input file describing the triangle shown in the beginning, we obtain the following output:

```
8
4 9
9 2 1
3 8 5 5
5 6 3 7 6

33/0 1 1 1 1
35/0 0 0 1 1
```

It shows that the best path is <8 4 9 8 6>, summing 35. The value of the greedy solution is 33.

Let us now try the recursive solution. Like in the knapsack problem, we first compute the optimal sum, i.e., the maximum possible sum that we can obtain from row x, column y downward. How can we express that optimal sum? Well, the optimal sum is the number at position <x, y>

plus the maximum of the optimal sums for the positions just below. That's exactly what the following recursive function Optimal expresses:

```
class Triangle {
//…
  virtual int Optimal(int x, int y) const
  {
    return numbers[x][y]
      + (x < size - 1 ? mas::Max(Optimal(x+1, y), Optimal(x+1, y+1)) : 0);
  }
};
```

The conditional expression x < size – 1 ? … : … is used to stop the recursion at the bottom row.

This function is essentially equivalent to the combinatorial algorithm, but it can be the basis for our dynamic programming solution.

Using this function, we can compute the solution, "undoing" the computation, just like in the knapsack problem:

```
class Triangle {
//…
  virtual std::list<int> Solution(int x, int y) const
  {
    std::list<int> result;
    result.push_back(y);
    for (int i = x+1, j = y; i < size; i++)
    {
      j += Optimal(i, j) < Optimal(i, j+1);
      result.push_back(j);
    }
    return result;
  }
};
```

We know that these functions, Optimal and Solution, are awful in what regards efficiency, but they are nice, simple and very readable. We shall now improve the efficiency using two derived classes, one for the dynamic programming solution, TriangleDynamic, and one for the memoized solution, TriangleMemo. Just like in the knapsack problem, both classes have an additional data member for the optimal table and both redefine function Optimal. Let us deal with class TriangleDynamic first:

```
class TriangleDynamic: public Triangle {
private:
  std::vector<std::vector<int> > optimal;
public:
  virtual int Optimal(int x, int y) const
  {
    return optimal[x][y];
  }
};
```

Before function Optimal can be used, the table must be initialized. We do as in the knapsack problem, by means of a function Compute, with the help of another function OptimalTabular:

```
class TriangleDynamic: public Triangle {
  //…
  virtual int OptimalTabular(int x, int y) const
  {
    return numbers[x][y]
      + (x < size - 1 ? mas::Max(optimal[x+1][y], optimal[x+1][y+1]) : 0);
  }

  virtual void Compute()
  {
```

```
    optimal.clear();
    optimal.resize(size);
    for (int i = size - 1; i >= 0; i--)
    {
      optimal[i].resize(i+1);
      for (int j = 0; j <= i; j++)
        optimal[i][j] += OptimalTabular(i, j);
    }
  }
};
```

There is a slight difference to the knapsack version. In that case, the table was filled upwards, here it is filled downwards. This is not essential, because we could have chosen the indices for the triangle in a different way, zero being the bottom row. (In that case, however, the functions would be programmed differently, with `x-1` instead of `x+1`, etc.)

Here is a test function for the dynamic programming solution, which merely computes and then prints the `optimal` table:

```
void TestTriangleDynamic(std::istream& input, std::ostream& output)
{
  TriangleDynamic td;
  td.Read(input);
  mas::WriteV(td.numbers, "\n", " ", output);
  output << std::endl;
  td.Compute();
  mas::WriteV(td.optimal, "\n", 3, output);
}
```

Function `WriteV` is a utility function that writes vectors of vectors. The second argument represents the separator between vectors and the third represents either the string to be inserted between adjacent items of a vector or the integer field width with which each item is written. (The test function illustrates both cases.)

The test function gives the following output:

```
8
4 9
9 2 1
3 8 5 5
5 6 3 7 6

 35
 27 25
 23 16 13
  9 14 12 12
  5  6  3  7  6
```

The top element, which corresponds to `Optimal(0, 0)` is 35, as expected.

In our dynamic programming solution, we decided to use a table to record the function results. From a strictly algorithmic point of view, if all we want is the sum along the optimal path (and not the path itself) we could reuse the number triangle, thus saving memory. But from a software engineering point of view, that would be a bad idea, for it makes little sense having the sum of a triangle that no longer exists. Still, we could use a vector instead of the matrix and in this case we would use much less memory and not destroy the triangle. That's better, but it would not allow us to reconstruct the path in a systematic way.

Consider now the memoized version, in class `TriangleMemo`:

```
class TriangleMemo: public Triangle {
public:
  mutable std::vector<std::vector<int> > optimal;

  virtual int Optimal(int x, int y) const
  {
```

```
    if (x >= static_cast<int>(optimal.size()))
      Grow(x);
    if (optimal[x][y] == -1)
      optimal[x][y] = Triangle::Optimal(x, y);
    return optimal[x][y];
  }

private:
  virtual void Grow(int x) const
  {
    int z = static_cast<int>(optimal.size());
    optimal.resize(x+1);
    for (int i = z; i < static_cast<int>(optimal.size()); i++)
      optimal[i].resize(i+1, -1);
  }
};
```

This is simpler than the dynamic programming version, and is almost identical to the knapsack case. The only difference is in the resizing of each row. Here we take advantage of knowing that the matrix is triangular, whereas in the previous case the number of columns was the same for all rows.

We finish this example showing a test function for class TriangleMemo:

```
void TestTriangleMemo(std::istream& input, std::ostream& output)
{
  TriangleMemo tm;
  tm.Read(input);
  mas::WriteV(tm.numbers, "\n", " ", output);
  output << std::endl;
  tm.Optimal(0, 0);  // this effectively builds the table
  mas::WriteV(tm.optimal, "\n", 3, output);
}
```

The output is identical to the output of the previous test function.

4 Indiana Jones

Our friend Indiana Jones got trapped in the temple of fire and his life is in danger. In order to escape he has to cross a rectangular room whose floor is heated from underneath. The floor is made up of large square tiles and all of them are very hot. Indiana Jones has a high-tech remote thermometer with which he can measure the temperature of the tiles from a distance. He wants to escape from temple, but he also wants to get his boots burned as little as possible. When racing through the room, he can jump from one tile to another tile in the next row, either in front of the current tile, or one position to the left, or one position to the right. He can only go to the left or to the right provided he does not hit the side wall, of course. He can enter the room at any tile of the first row and he can leave the room at any tile in the last row. Which path should he take?

He'd better not be greedy, for the greedy solution is not always the best one. Consider the following small temple, with six tiles:

```
50 55 60
50 40 25
```

Indiana Jones is at the top and freedom is at the bottom. If he greedily chooses the less hot tile from the first row, the minimal possible accumulated temperature will be 90, whereas if he enters at the top right, where the hottest tile is, he will be able to escape with a total of 85.

Anyway, let us set up the class for this problem with the help of the greedy approach. There will be a data member, temperatures, for the temperatures (this is a rectangular matrix of integers), a function SolutionGreedy (x, y) to compute the greedy path starting in row x, column y, a function SolutionGreedy(x), with one argument, to compute the greedy path starting in row x

and a function Cost that computes the cost, i.e., the accumulated temperature along a given path. Functions SolutionGreedy return lists of integers, and the argument of Cost is one such list:

```
class IndianaJones {
public:
  std::vector<std::vector<int> > temperatures;
  int rows;
  int columns;
public:
  virtual void Read(std::istream& input = std::cin)
  {
    input >> rows >> columns;
    temperatures.clear();
    temperatures.resize(rows);
    for (int i = 0; i < rows; i++)
    {
      temperatures[i].reserve(columns);
      for (int j = 0; j < columns; j++)
      {
        int x;
        input >> x;
        temperatures[i].push_back(x);
      }
    }
  }

  virtual std::list<int> SolutionGreedy(int x, int y) const
  {
    std::list<int> result;
    result.push_back(y);
    for (int i = x+1, j = y; i < rows; i++)
    {
      //...
    }
    return result;
  }

  virtual std::list<int> SolutionGreedy(int x) const
  {
    int start =
      static_cast<int>(std::min_element(temperatures[x].begin(), temperatures[x].end())
        - temperatures[x].begin());
    return SolutionGreedy(x, start);
  }

  virtual int Cost(const std::list<int>& x) const
  {
    int result = 0;
    int z = rows - 1;
    for (std::list<int>::const_reverse_iterator i = x.rbegin(); i != x.rend(); i++)
      result += temperatures[z--][*i];
    return result;
  }
};
```

Let us now fill the body of the loop in the first function SolutionGreedy. If at some step we are at tile <i, j>, then, at the next step, we will be at the least hot of the three tiles <i+1, j-1>, <i+1, j>, <i+1, j+1>. However, tile <i+1, j-1> does not exist if j is zero, and tile <i+1, j+1> does not exist if j is equal to columns – 1. In order to avoid these particular cases, we introduce three local variables, t1, t2, t3, for the temperatures of those three tiles, with the provision that the values of t1 and of t3 will be "infinite" in case the respective tiles do not exist. By being "infinite" they do not affect the computation of the minimum. In a way, it is as if the walls are extremely hot, and Indiana Jones must never bump into them. Observe:

```
class IndianaJones {
//...
  virtual std::list<int> SolutionGreedy(int x, int y) const
  {
    std::list<int> result;
    result.push_back(y);
    for (int i = x+1, j = y; i < rows; i++)
    {
      int t1 = j == 0 ? std::numeric_limits<int>::max() : temperatures[i][j-1];
      int t2 = temperatures[i][j];
      int t3 = j == columns - 1? std::numeric_limits<int>::max() : temperatures[i][j+1];
      j += mas::Match(mas::Min(t1, t2, t3), t1, t2, t3) - 1;
      result.push_back(j);
    }
    return result;
  }
};
```

A word about utility function mas::Match. It is a function with a variable number of arguments, and it returns the order of the first argument in the list of arguments, not counting the first, whose value is equal to the value of the first argument:

```
template <class T>
int Match(T x, ...)
{
  va_list p;
  va_start(p, x);
  int result = 0;
  while (static_cast<T>(va_arg(p, T)) != x)
    result++;
  va_end(p);
  return result;
}
```

For example, the value of mas::Match(5, 2, 9, 2, 5, -1, 8) is 3 and the value of mas::Match(4, 4) is zero. The function must not be called if no other argument is equal to the first.

This is fine, but we do not want the greedy path, we want the optimal path. As usual, we will need two functions: function Optimal(x, y), returning the cost of the optimal path that starts at the tile in row x, column y, and function Solution(x, y), computing that path. For function Optimal(int x, int y) we use a straightforward recursive definition, repeating the trick of considering the walls very hot:

```
class IndianaJones {
//...
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == rows - 1)
      result = temperatures[x][y];
    else
    {
      int t1 = y == 0 ? std::numeric_limits<int>::max() : Optimal(x+1, y-1);
      int t2 = Optimal(x+1, y);
      int t3 = y == columns - 1? std::numeric_limits<int>::max() : Optimal(x+1, y+1);
      result = temperatures[x][y] + mas::Min(t1, t2, t3);
    }
    return result;
  }
};
```

We can program function Solution along the lines of function SolutionGreedy, now extending the path to the next optimal tile, instead of extending to the next least hot tile:

```
class IndianaJones {
```

```
//...
  virtual std::list<int> Solution(int x, int y) const
  {
    std::list<int> result;
    result.push_back(y);
    for (int i = x+1, j = y; i < rows; i++)
    {
      int t1 = j == 0 ? std::numeric_limits<int>::max() : Optimal(i, j-1);
      int t2 = Optimal(i, j);
      int t3 = j == columns - 1? std::numeric_limits<int>::max() : Optimal(i, j+1);
      j += mas::Match(mas::Min(t1, t2, t3), t1, t2, t3) - 1;
      result.push_back(j);
    }
    return result;
  }
};
```

Function Solution(int x) computes the optimal solution starting at row x.

```
class IndianaJones {
//...
  virtual std::list<int> Solution(int x) const
  {
    int start = 0;
    for (int j = 1; j < columns; j++)
      if (Optimal(x, j) < Optimal(x, start))
        start = j;
    return Solution(x, start);
  }
};
```

Of course we could use local variables to save a few function calls but that will be provided automatically when we derive this class using dynamic programming or memoization.

Let us write a small test function that reads the description of the room from a file, computes the greedy path and the optimal path, and displays them at the console together with their values:

```
void TestIndianaJones(std::istream& input, std::ostream& output)
{
  IndianaJones ij;
  ij.Read(input);
  std::list<int> x = ij.SolutionGreedy(0);
  output << x << "/" <<  ij.Cost(x) << std::endl;
  std::list<int> y = ij.Solution(0);
  output << y << "/" <<  ij.Cost(y) << std::endl;
}
```

Here is an input file:

```
8 5
50 55 50 45 50
60 50 70 60 60
50 70 55 70 70
75 75 50 60 75
45 65 70 65 45
50 55 65 55 50
60 60 70 65 60
65 55 65 55 65
```

The two numbers in the first line represent the number of rows and the number of columns. Recall that Indiana Jones travels top down.

We get the following output:

```
3 3 2 2 1 0 0 1/440
0 1 2 3 4 4 4 3/425
```

The optimal solution is better than the greedy one, as expected.

We could now improve the efficiency of our program using dynamic programming or memoization as in the previous examples. But we will do it differently here. We have observed before, in the knapsack problem and in the triangle problem, that the scheme for memoization is rather systematic. Our goal now is doing the memoization automatically, without having to define a new class for each new problem.

As a matter of fact, it is surprisingly simple. All we need is a class that derives from our problem class redefining function Optimal and storing the computed values in a table for later reuse without recomputation. For cases of functions with two integer variables, the table will be a matrix. The problem class will be represented by a template class argument.

```
template <class T>
class MemoMatrix: public T {
public:
  mutable std::vector<std::vector<int> > optimal;
  int impossible;
public:
  MemoMatrix():
     optimal(),
     impossible(-1)
  {
  }

  virtual int Optimal(int x, int y) const
  {
    if (x >= static_cast<int>(optimal.size()))
      optimal.resize(x+1);
    if (y >= static_cast<int>(optimal[x].size()))
      optimal[x].resize(y+1, impossible);
    if (optimal[x][y] == impossible)
      optimal[x][y] = T::Optimal(x, y);
    return optimal[x][y];
  }
};
```

The table is mutable because it must be modified by function Optimal, which is a const function. Be default we use -1 as the impossible value, and we fill the table elements with it when the table grows.

Here is an example of how to use this generic class:

```
void TestIndianaJonesMemo(std::istream& input, std::ostream& output)
{
  MemoMatrix<IndianaJones> ij;
  ij.Read(input);
  std::list<int> solution = ij.Solution(0);
  mas::WriteV(ij.optimal, "\n", 4, output);
  output << std::endl;
  output << solution << "/" <<  ij.Cost(solution) << std::endl;
}
```

For the same input file, the output is the following:

```
 425 430 425 430 435
 395 375 395 385 400
 335 350 325 340 340
 285 285 280 270 285
 210 230 240 230 210
 165 170 180 170 165
 115 115 125 120 115
  65  55  65  55  65

0 1 2 3 4 4 4 3/425
```

The memoized class is essentially equivalent to the standard dynamic programming class, that would be constructed as we illustrated in the programs of the previous sections (knapsack and triangle), in the sense that function Optimal is actually computed only once for each distinct pair of arguments. Nevertheless, the memoized class uses recursion, and in extreme cases it can cause stack overflow, whereas a pure dynamic programming tabulation avoids that problem.

5 Mars Explorer

A robotic space probe has landed in Mars[1]. Its mission now is to travel from the landing point to the base station, which lies somewhere to the southeast. On the way, the probe will collect samples of rocks, and the more samples it collects, the better. A digitized map of the terrain has been fed into the probe's computers. The map is a grid. In each square of the grid there is a number: zero means that area is flat, can be crossed without problem, but there aren't any interesting rocks there; a positive number *x* means that the area is OK and there are *x* interesting rocks to collect; -1 means that area is dangerous and should be avoided, otherwise the robot vehicle might get stuck there. The vehicle moves from a grid square to another adjacent grid square but always to the east or to the south. Hence, for the purpose of the problem, we can suppose that the vehicle lands at the upper left corner of the map (the northwesternmost square) and that the base station is at the lower right corner (the southeasternmost square).

What path should the probe take, is order to collect as much rocks as possible on its way from the landing point to the base station?

Here is an example of such a grid map:

```
 0  2  0  1 -1  3  0
 1 -1  1  0  0  2  3
 1  0  0  1 -1 -1  0
-1  0  1  0  0  2  0
 2  2  0  0  1  1  0
```

This is a simplification of a problem used at the International Olympiad in Informatics, in 1997, in South Africa.

Actually, this problem is similar to the Indiana Jones problem: it is as if the probe was Indiana Jones, the grid is the room, Indiana enters the room at the upper left corner, leaves the room at the lower right corner and can only go one step to the right or one step down at a time. There would be the additional complication that some tiles are missing, leaving a hole to the basement where a bunch of hungry crocodiles would quickly devour our hero, should he, by mistake, fall through one of those holes on the floor.

Therefore the solution is also similar. As before, it is based on function Optimal. In this case, let Optimal(x, y) be the optimal cost of a path from the landing point to point (x, y). If the point is not in the first column, not in the first line, and there are no dangerous positions, we have the following recursion:

```
Optimal(x, y) = map[y][x] + Optimal(x – 1, y) + Optimal(x, y – 1)
```

Note that we write map[y][x] and not map[x][y]. This is because we are now interpreting x and y as Cartesian coordinates. Coordinate x varies horizontally from left to right and coordinate y varies vertically, from top to bottom. For the map, we want the first index to be the row index (vertical) and the second index to be the column index (horizontal). Note that y grows as we move down, unlike convention in Cartesian geometry.

The case with dangerous spots can be handled with a little more work, if we decide that in this case the optimal value is -1:

```
Optimal(x, y) = map[y][x] == -1 || Optimal(x-1, y) == -1 && Optimal(x, y-1) == -1 ?
```

[1] As I review these pages, on the first days of January 2004, the American rover Spirit has started to send amazing pictures back from Mars, while British lander Beagle 2 still has not contacted earth.

```
                -1 :
                map[y][x] + Optimal(x-1, y) + Optimal(x, y – 1)
```

We must complicate this definition further in order to consider the case where the point is in the first row or in the first column, just as in the Indiana Jones case we had to be careful about the side walls. We will do differently here, to illustrate another simple technique.

The general idea is to surround the map by an extra layer of dangerous squares, like this:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1  0  2  0  1 -1  3  0 -1
-1  1 -1  1  0  0  2  3 -1
-1  1  0  0  1 -1 -1  0 -1
-1 -1  0  1  0  0  2  0 -1
-1  2  2  0  0  1  1  0 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
```

Now the start point is point (1, 1) and the end point is (columns, rows), where columns represents the number of columns and rows represents the number of rows. We do not have to worry any more about recursive calls falling out of the grid, because the robot will never be on the borderline.

We handle the extra columns and rows when initializing the map. As a matter of fact, in this problem, given the shape of the recursion, we do not need the extra column or the right or the extra column at the bottom, so we will omit them. But the general idea is applicable in other situations.

Here's our class MarsExplorer:

```
class MarsExplorer {
public:
  std::vector<std::vector<int> > map;
  int rows;
  int columns;
public:
  virtual ~MarsExplorer()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> rows >> columns;
    if (!input)
      return;
    map.clear();
    map.resize(rows + 1);
    map[0].resize(columns+1, -1);
    for (int i = 1; i <= rows; i++)
    {
      map[i].reserve(columns+1);
      map[i].push_back(-1);
      for (int j = 0; j < columns; j++)
      {
        int x;
        input >> x;
        map[i].push_back(x);
      }
    }
  }
};
```

This function Read, unlike the ones in the previous problems, is generalized in order to be able to deal with a sequence of data sets in the input file. If there is nothing else on the input file, it returns immediately. The calling function can check whether the input file is in error and thus detect the end of file.

Function Optimal(x, y) is rather straightforward, given that we do not have to worry about the border cases:

```
class MarsExplorer {
  //…
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == 1 && y == 1)
      result = map[1][1];
    else if (map[y][x] == -1 || Optimal(x-1, y) == -1 && Optimal(x, y-1) == -1)
      result = -1;
    else
      result = map[y][x] + mas::Max(Optimal(x-1, y), Optimal(x, y-1));
    return result;
  }
};
```

In this problem, the solution will be described by a string of made up of the letters ‘E’ and ‘S’, ‘E’ meaning “go east” and ‘S’ meaning “go south”:

```
class MarsExplorer {
  //…
  virtual std::string Solution(int x, int y) const
  {
    std::string result;
    result.reserve(x + y - 2);
    while (!(x == 1 && y == 1))
      if (Optimal(x-1, y) >= Optimal(x, y-1))
        result += 'E', x--;
      else
        result += 'S', y--;
    std::reverse(result.begin(), result.end());
    return result;
  }
};
```

In case of a tie, the robot goes east rather than south.

Function Cost evaluates the cost of a solution described by a string:

```
class MarsExplorer {
  //…
  int Cost(const std::string& s) const
  {
    int result = 0;
    int x = 1;
    int y = 1;
    for (std::string::const_iterator i = s.begin(); i != s.end(); i++)
      if (*i == 'E')
        result += map[y][x++];
      else
        result += map[y++][x];
    result += map[y][x];
    return result ;
  }
};
```

Here’s a test function that uses automatic memoization. Note that we must reinitialize the impossible value, because the default impossible value, -1, is not impossible in this problem:

```
void TestMarsExplorer(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    MemoMatrix<MarsExplorer> m;
```

```
    m.impossible = -9;
    m.Read(input);
    if (!input)
      return;
    mas::WriteV(m.map, "\n", 3, output);
    output << std::endl;
    output << m.Optimal(m.columns, m.rows) << std::endl;
    if (m.Optimal(m.columns, m.rows) > 0)
    {
      std::string solution = m.Solution(m.columns, m.rows);
      output << solution << "/" << m.Cost(solution) << std::endl;
    }
    mas::WriteV(m.optimal, "\n", 3, output);
    output << std::endl;
  }
}
```

Note that this test function is prepared to handle a sequence of data sets in the input file.

For the input file that corresponds to the grid map show in the beginning of this section, we get the following output:

```
 -1 -1 -1 -1 -1 -1 -1 -1
 -1  0  2  0  1 -1  3  0
 -1  1 -1  1  0  0  2  3
 -1  1  0  0  1 -1 -1  0
 -1 -1  0  1  0  0  2  0
 -1  2  2  0  2  1  1  0

8
SSESSEEEEE/8
 -9 -9 -1 -1 -9 -1
 -9  0  1  2 -1 -1
 -1  2 -1  2  2  4
 -1  2  3  3  4  4
 -1  3  3  4  4  6
 -9 -1  3 -1  4  7
 -1 -1  5 -1  6  8
 -1 -1  8  8  8  8
```

The input file itself is as follows. Note that the first line indicates the number of rows and the number of columns. In general, the input file could have several data sets like this. That's a common situation in programming competitions.

```
5 7
 0  2  0  1 -1  3  0
 1 -1  1  0  0  2  3
 1  0  0  1 -1 -1  0
-1  0  1  0  0  2  0
 2  2  0  2  1  1  0
```

6 Longest Ascending Subsequence

One of the standard programming problems that is solved using dynamic programming is computing the longest ascending subsequence in a sequence of, say, integer numbers. For example consider the sequence <3, 4, 2, 5, 5, 3, 6>. The sequence <4, 2, 5> is a subsequence of that sequence, the sequence <2, 5, 5, 6> is an ascending subsequence, and the sequence <3, 4, 5, 5, 6> is a longest ascending subsequence.

In general, there could be several longest ascending sequences. For example, in the sequence <3, 2, 5, 4> the two subsequences <3, 5> and <2, 4> are both longest, and so are <3, 4> and <2, 5>. We will have functions for the first such subsequence and for the last one. First and last mean first and last in the lexicographical order of the indices of the elements selected for the subsequence. In the sequence <3, 2, 5, 4>, the subsequence <3, 5> is the first longest ascending subsequence and the subsequence <2, 4> is the last.

The combinatorial solution is simple: generate all ascending subsequences and pick the longest. As the algorithm generates subsequences in lexicographical order of the indices, the first longest is generated first (among all the longest) and the last longest in generated last. We will program this, like we did previously, as a warm up, and also to make our class more interesting.

In this problem, we will use a template class LongestAscendingSubsequence<T> so that we can use it for sequences of any type. (More precisely: for sequences of any type with a <= operator, such as int, double and std::string.) The sequence itself is represented by a vector with elements of type T. For generating all ascending subsequences, we use an inner class Combinations, as usual:

```
template <class T>
class LongestAscendingSubsequence {
public:
  std::vector<T> items;
public:
  virtual ~LongestAscendingSubsequence()
  {
  }

  virtual void Read(std::istream& input)
  {
    int n;
    input >> n;
    if (!input)
      return;
    items.clear();
    items.reserve(n);
    for (int i = 0; i < n; i++)
    {
      T x;
      input >> x;
      items.push_back(x);
    }
  }

  virtual const std::list<int>& FirstLongest(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    std::list<int>::size_type max = 0;
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      std::list<int>::size_type temp = i->size();
      if (max < temp)
      {
        max = temp;
        result = i;
      }
    }
    return *result;
  }

  virtual const std::list<int>& LastLongest(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_reverse_iterator result;
    std::list<int>::size_type max = 0;
    for (std::list<std::list<int> >::const_reverse_iterator i = x.rbegin(); i != x.rend(); i++)
    {
      std::list<int>::size_type temp = i->size();
      if (max < temp)
      {
        max = temp;
        result = i;
```

```
    }
   }
   return *result;
  }

  virtual std::list<T> SolutionCombinatorial() const
  {
   std::list<T> result;
   Combinations c(*this);
   c.Compute();
   if (!c.combinations.empty())
    result = FirstLongest(c.combinations);
   return result;
  }

public:
  class Combinations {
  public:
   const LongestAscendingSubsequence& s;
   std::list<std::list<int> > combinations;
  private:
   std::list<int> temp;
  public:
   Combinations(const LongestAscendingSubsequence& s):
    s(s)
   {
   }

   virtual ~Combinations()
   {
   }

   virtual void Compute()
   {
    temp.clear();
    combinations.clear();
    for (int i = 0; i < static_cast<int>(s.items.size()); i++)
     Visit(i);
   }

  private:
   virtual void Visit(int x)
   {
    temp.push_back(s.items[x]);
    combinations.push_back(temp);
    for (int i = x+1; i < static_cast<int>(s.items.size()); i++)
     if (s.items[x] <= s.items[i])
      Visit(i);
    temp.pop_back();
   }
  private:
   void operator = (const Combinations&){}
  };

};
```

Here is a test function that cyclically reads a vectors of numbers, writes it on the output file and then computes and shows their first and last longest ascending subsequences:

```
void TestLongestAscendingSubsequenceCombinatorial(
       std::istream& input, std::ostream& output)
{
 for (;;)
 {
```

```
    LongestAscendingSubsequence<int> las;
    las.Read(input);
    if (!input)
      break;
    mas::WriteLine(las.items, " ", output);
    LongestAscendingSubsequence<int>::Combinations c(las);
    c.Compute();
    mas::WriteLine(las.FirstLongest(c.combinations), " ", output);
    mas::WriteLine(las.LastLongest(c.combinations), " ", output);
    output << std::endl;
  }
}
```

We tried the test function with the following input file (five lines):

```
5 3 6 1 7 2
10 1 2 3 4 1 2 9 10 8 2
20 6 22 3 11 2 66 302 12 34 2 12 18 45 77 2 34 88 12 19 54
30 55 3 4 11 97 45 3 44 2 99 66 44 65 87 2 99 1 55 34 29 25 56 29 31 10 20 37 88 12 54
6 3 2 6 5 9 8
```

(Note that in each line, the first number is the number of numbers that follow.)

This was the result we got on the output file:

```
3 6 1 7 2
3 6 7
3 6 7

1 2 3 4 1 2 9 10 8 2
1 2 3 4 9 10
1 2 3 4 9 10

6 22 3 11 2 66 302 12 34 2 12 18 45 77 2 34 88 12 19 54
6 11 12 12 18 45 77 88
3 11 12 12 18 45 77 88

55 3 4 11 97 45 3 44 2 99 66 44 65 87 2 99 1 55 34 29 25 56 29 31 10 20 37 88 12 54
3 4 11 44 44 65 87 99
3 4 11 25 29 31 37 54

3 2 6 5 9 8
3 6 9
2 5 8
```

So much for the warm up. Let us now tackle the problem using the standard recursive approach, as we did in the previous examples. In this case, the function `Optimal(int x)` will represent the length of the longest ascending subsequence ending in position `x` of the sequence. (The first position is position zero.) Let `k` be an index such that `k < x` and such that `items[k] <= items[x]`. Then there is an ascending subsequence of length `Optimal(k) + 1` ending in `x`. The longest of all such ascending subsequences (for all `k < x` such that `items[k] <= items[x]`) is bound to be the longest ascending subsequence ending in position `x`. This argument explains the following definition:

```
template <class T>
class LongestAscendingSubsequence {
//...
  // Optimal(x) is the length of the longest ascending subsequence ending in position x.
  virtual int Optimal(int x) const
  {
    int result = 0;
    for (int i = 0; i < x; i++)
      if (items[i] <= items[x])
        if (result < Optimal(i))
          result = Optimal(i);
    result++;
    return result;
  }
};
```

Again, we could avoid some function calls using local variables, but we do not bother to do that because we will remove all superfluous calls systematically later.

After this, obtaining the length of the longest ascending subsequence is simple:

```
template <class T>
class LongestAscendingSubsequence {
//…
  virtual int Optimal0() const
  {
    int result = 0;
    for (int i = 0; i < static_cast<int>(items.size()); i++)
      if (result < Optimal(i))
        result = Optimal(i);
    return result;
  }
};
```

Computing the first longest ascending subsequence is more elaborate that merely computing its length. (Remember that by “first” we mean “first in the sense of the lexicographical order of the sequences of indices”, not the lexicographical order of the sequences of items.) Actually, it is more convenient to compute the first longest ascending subsequence ending at a given position. (Later we will provide a function to compute the last position of the first ascending sequence with a given length.) The result, implemented as a list, is built iteratively from the end. At each step we search the end of the longest ascending subsequence of the current length, that sequence being such that its last element is less or equal to the previously selected elements. The current length initially is the length of the longest ascending sequence ending at the given position, and it decreases by one each time we add one element to the result:

```
template <class T>
class LongestAscendingSubsequence {
//…
  std::list<int> FirstSolution(int x) const
  {
    std::list<int> result;
    result.push_front(items[x]);
    for (int z = Optimal(x) - 1; z > 0; z--)
    {
      int i = 0;
      while (!(items[i] <= items[x] && Optimal(i) == z))
        i++;
      result.push_front(items[i]);
      x = i;
    }
    return result;
  }
};
```

As a matter of fact, computing the last ascending sequence ending at a given position is more efficient, because we have to traverse the sequence in one direction only, from the initial position to the beginning of the sequence, always going from right to left:

```
template <class T>
class LongestAscendingSubsequence {
//…
  std::list<int> LastSolution(int x) const
  {
    std::list<int> result;
    result.push_front(items[x]);
    for (int z = Optimal(x) - 1; z > 0; z--)
    {
      int i = x - 1;
      while (!(items[i] <= items[x] && Optimal(i) == z))
        i--;
```

```
    result.push_front(items[i]);
    x = i;
   }
   return result;
  }
};
```

These two functions return the first and the last longest ascending subsequence ending a given position. In order to compute the first or the last longest ascending subsequence with a given length, all we need is functions to compute the first or last positions where a longest ascending subsequence with a given length ends. These are simple linear search functions, which also rely on function Optimal:

```
template <class T>
class LongestAscendingSubsequence {
//…
  virtual int EndingFirst(int x) const
  {
   int result = 0;
   while (Optimal(result) != x)
    result++;
   return result;
  }

  virtual int EndingLast(int x) const
  {
   int result = static_cast<int>(items.size()) - 1;;
   while (Optimal(result) != x)
    result--;
   return result;
  }
 };
```

This ends the first part of the exercise. We now have a recursive solution (other than the combinatorial one) and we know it is not efficient. In order to make it efficient, we may use dynamic programming or memoization. The dynamic programming solution is simpler than the ones we saw in the previous problems, because function Optimal has only one variable, and hence, the table is a vector (and not a matrix, as before):

```
template <class T>
class LongestAscendingSubsequenceDynamic: public LongestAscendingSubsequence<T> {
private:
  std::vector<int> optimal;
private:
 virtual int OptimalTabular(int x) const
  {
   int result = 0;
   for (int i = 0; i < x; i++)
    if (items[i] <= items[x])
     if (result < optimal[i])
      result = optimal[i];
   result++;
   return result;
  }

public:
  virtual void Compute()
  {
   optimal.clear();
   optimal.resize(items.size());
   for (int i = 0; i < static_cast<int>(items.size()); i++)
    optimal[i] = OptimalTabular(i);
  }
```

```
  virtual int Optimal(int x) const
  {
    return optimal[x];
  }
};
```

The memoized version could be built by hand or using an automatic technique, as in the Indiana Jones problem. Instead of a MemoMatrix we now use a MemoVector:

```
template <class T>
class MemoVector: public T {
public:
  mutable std::vector<int> optimal;
  int impossible;
public:
  MemoVector():
    optimal(),
    impossible(-1)
  {
  }

  virtual int Optimal(int x) const
  {
    if (x >= static_cast<int>(optimal.size()))
      optimal.resize(x+1, impossible);
    if (optimal[x] == impossible)
      optimal[x] = T::Optimal(x);
    return optimal[x];
  }
};
```

Here is a test function using automatic memoization:

```
void TestLongestAscendingSubsequence(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    MemoVector<LongestAscendingSubsequence<int> > las;
    las.Read(input);
    if (!input)
      break;
    mas::WriteLine(las.items, " ", output);
    output << las.Optimal0() << std::endl;
    mas::WriteLine(las.optimal, " ", output);
    mas::WriteLine(las.FirstSolution(las.EndingFirst(las.Optimal0())), " ", output);
    mas::WriteLine(las.LastSolution(las.EndingLast(las.Optimal0())), " ", output);
    output << std::endl;
  }
}
```

For the same input file, we now get the following output:

```
3 6 1 7 2
3
1 2 1 3 2
3 6 7
3 6 7

1 2 3 4 1 2 9 10 8 2
6
1 2 3 4 2 3 5 6 5 4
1 2 3 4 9 10
1 2 3 4 9 10

6 22 3 11 2 66 302 12 34 2 12 18 45 77 2 34 88 12 19 54
8
1 2 1 2 1 3 4 3 4 2 4 5 6 7 3 6 8 5 6 7
6 11 12 12 18 45 77 88
3 11 12 12 18 45 77 88
```

```
55 3 4 11 97 45 3 44 2 99 66 44 65 87 2 99 1 55 34 29 25 56 29 31 10 20 37 88 12 54
8
1 1 2 3 4 4 2 4 1 5 5 5 6 7 2 8 1 6 4 4 4 7 5 6 3 4 7 8 4 8
3 4 11 44 44 65 87 99
3 4 11 25 29 31 37 54

3 2 6 5 9 8
3
1 1 2 2 3 3
3 6 9
2 5 8
```

In each group of five lines, the first shows the sequence, the second the length of the longest ascending subsequence, the third the contents of the optimal table from the MemoVector class, the forth the first longest ascending subsequence, and the fifth the last longest ascending subsequence.

7 Intermezzo: Fibonacci numbers

Computing the sequence of Fibonacci numbers, <0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...>, is a standard exercise in elementary programming. It can also be done with classes. Here is a class Fibonacci with a function At, such that for an object f of type Fibonacci, f.At(x) is the x-th number in the Fibonacci sequence. This function can also be used through the () operator:

```
class Fibonacci {
public:
  virtual int At(int x) const
  {
    int result;
    if (x == 0)
      result = 0;
    else if (x == 1)
      result = 1;
    else
      result = At(x-2) + At(x-1);
    return result;
  }

  virtual int operator ()(int x) const
  {
    return At(x);
  }
};
```

It's no surprise that the recursive function At is not very efficient. Can we memoize it using class MemoVector<T>? No, we cannot, because the name of the recursive function is At, not Optimal. Of course we can change the class and rename At as Optimal, or we can create a specialized MemoVector, for recursive functions named At. However, it is more interesting to borrow from Software Engineering the design pattern Adapter, which teaches us how to "convert the interface of a class into another interface clients expect" [Gamma *et al.*, page 139]. In our case, the interface the clients expects exhibits function Optimal; the interface of class Fibonacci offers function At instead. The adapter adapts one to the other:

```
class FibonacciAdapter: public Fibonacci {
public:
  virtual int Optimal(int x) const
  {
    return Fibonacci::At(x);
  }

  virtual int At(int x) const
  {
    return Optimal(x);
  }
```

```
};
```

This adapter has an interesting twist: not only do we have to change Optimal into At from class Fibonacci, which is done by the first function, but also we must make sure the recursive calls of At, for objects of the adapted class, are memoized as well. This is the purpose of the second function.

Here's a small test function that illustrates the use of the adaptor:

```
void TestFibonacci()
{
 MemoVector<FibonacciAdapter> fibm;
 for (;;)
 {
  int x;
  std::cin >> x;
  if (!std::cin)
   break;
  std::cout << fibm(x) << std::endl;
 }
}
```

The standard iterative version which leaves students bewildered the first time they see it is more complicated:

```
class Fibonacci {
//…
 virtual int F(int x) const
 {
  int result;
  if (x == 0)
   result = 0;
  else if (x == 1)
   result = 1;
  else
  {
   result = 0;
   int previous = 1;
   for (int i = 1; i <= x; i++)
   {
    int temp = result;
    result += previous;
    previous = temp;
   }
  }
  return result;
 }
};
```

Indeed, this iterative function uses a simplified kind of dynamic programming: all previous values of the sequence have been computed previously, but only the latest two need to be kept. Needless to say, if the function is computed again, another iteration is carried out, whereas in a truly dynamic programming version, the computed values are there to be reused.

8 Bachet's Game

Bachet's game is a game for two players who make moves alternately. There are *N* stones on a table. One player makes a move by removing 1 or 2 or 3, or … or *K* stones from the table. In each match, the winner is the player who removes the last stone. Apparently this game is named after the French mathematician Claude Gaspar Bachet de Méziriac (1581-1638).

In a more challenging generalization of this game the number of stones that can be removed in each move must be a member of a certain set of numbers (not necessarily all the integers from 1

to *K*). The winner is the player who makes the last move. Unlike in the previous case, if 1 is not in the set of numbers, the match can end even if the table is not empty.

We want to write a program that plays Bachet's game (the generalized version). The program plays optimally, i.e., if it can win, it will make moves that certainly lead to victory, whatever the responses of the opponent are. If the situation is such that victory is not certain, the program plays randomly.

Bachet's game is a popular program in programming competitions.

As we observed, Bachet's game is a game for two players, played with integers. In the next section we will see another game also for two players and also played with integers. What do these games with integers have in common? Well, when players make a move, they change the state of the match. The move is represented by an integer number. Not all integer numbers are legal moves. The state, for Bachet's game is also an integer number (the number of remaining stones), but in general states can be more complicated. The player who wins is the one who makes the last move (by leaving the match in a state for which there are no legal moves). A player is in a winning state if he can make a move leading the match to a state that is not a winning state for the opponent. The initial state can be fixed or not. Each match can start with either player.

Here's an abstract class that captures the nature of these games of integers. The template argument represents the type of the state:

```
template <class T>
class Game {
private:
  bool player; // true -> computer; false -> human.
public:
  virtual  ~Game()
  {
  }

  virtual void Start(const T& x) = 0;
  virtual bool Wins() const = 0;
  virtual int WinningMove() const = 0;  // pre Wins();
  virtual int RandomMove() const = 0;
  virtual void Show() const = 0;
  virtual void Play(int x) = 0;
  virtual void Unplay(int x) = 0;
  virtual int YourMove() const = 0;
  virtual bool ValidMove(int x) const = 0;
  virtual bool MatchOver() const = 0;
};
```

All games are played in the same way. At each step in a match, the program displays the state of the match, using function Show, so that the human player knows what is going on. If it is the computer's turn, it will choose a winning move, with function WinningMove, if it is in a winning state, or a random move, with function RandomMove, otherwise. After having chosen the move, the computer plays using function Play. It finds out whether it is in a winning state, using function Wins. When it's the human player's turn, the computer accepts the human's move with function YourMove. Occasionally, it may be necessary to backtrack a given move. Function Unplay does that. Function ValidMove tests whether the given move is valid, considering the state of the match. Function MatchOver tests whether the match is over (because there are no more legal moves.)

All these functions are pure virtual and they will be defined in the derived class. On the other hand, the functions that represent playing a match from beginning to end, PlayMatchIStart and PlayMatchYouStart, can be programmed in the abstract class in terms of the others:

```
template <class T>
class Game {
```

```
public:
//…

  virtual void PlayMatchIStart()
  {
    //…
  }

  virtual void PlayMatchYouStart()
  {
    //…
  }
};
```

The loop is in function PlayMatchIStart. The other function merely adds a preliminary step:

```
template <class T>
class Game {
public:
//…

  virtual void PlayMatchYouStart()
  {
    Show();
    player = false;
    Play(YourMove());
    PlayMatchIStart();
  }
};
```

Function PlayMatchIStart is more elaborate:

```
template <class T>
class Game {
public:
//…
  virtual void PlayMatchIStart()
  {
    Show();
    while (!MatchOver())
    {
      int myMove;
      if (Wins())
        myMove = WinningMove();
      else
        myMove = RandomMove();
      //std::cout << (Wins() ? "W" : "R") << " ";
      std::cout << "I play " << myMove << std::endl;
      player = true;
      Play(myMove);
      Show();
      if (!MatchOver())
      {
        player = false;
        //if (Wins())
        //  std::cout << "Suggestion: " << WinningMove() << std::endl;
        Play(YourMove());
        Show();
      }
    }
    std::cout << "Match over." << std::endl;
    if (player)
      std::cout << "I win." << std::endl;
    else
      std::cout << "You win." << std::endl;
```

```
  }
};
```

Uncomment the first comment if you want to know if the computer is playing randomly or if it is in a winning state, and playing for victory. Uncomment the second comment for getting advice from the computer on how to play for victory, if victory is possible for the human player.

This was the abstract class. We know what to do in order to get Bachet's game: derive a BachetsGame class from class Game and provide suitable implementations for the abstract functions.

Class BachetsGame has a vector data member for the allowed moves. We keep it sorted, because it simplifies some functions. Function Read reads the allowed moves from a file, or, more generally, from a stream. The remaining of the class consists of the definitions of the inherited pure virtual functions, Start, ValidMove, Show, Play, Unplay, Show, MatchOver, etc:

```
class BachetsGame: public Game<int> {
public:
  std::vector<int> moves;
  int state;
public:
  virtual ~BachetsGame()
  {
  }

  virtual void Start(const int& x)
  {
    state = x;
  }

  virtual void Read(std::istream& input = std::cin)
  {
    int n;
    input >> n;
    moves.reserve(n);
    for (int i = 0; i < n; i++)
    {
      int x;
      input >> x;
      moves.push_back(x);
    }
    std::sort(moves.begin(), moves.end());
  }

  virtual bool ValidMove(int x) const
  {
    return x <= state && std::find(moves.begin(), moves.end(), x) != moves.end();
  }

  virtual void Show() const
  {
    std::cout << state << std::endl;
  }

  virtual void Play(int x)
  {
    state -= x;
  }

  virtual void Unplay(int x)
  {
    state += x;
  }
```

```
  virtual bool MatchOver() const
  {
    return state < moves[0];
  }
};
```

The crucial function is the Boolean function Wins(). We will program it in terms of another Boolean function Wins(int x), in which the argument represents the current state. The value of Wins(x) is true if x is a winning state. We follow the recursive definition: a state is a winning state if there is a move that leads to a non-winning state:

```
class BachetsGame: public Game<int> {
//...
  virtual bool Wins(int x) const
  {
    bool result = false;
    for (std::vector<int>::const_reverse_iterator j = moves.rbegin();
         !result && j  != moves.rend(); j++)
      if (x >= *j && !Wins(x - *j))
        result = true;
    return result;
  }
};
```

The inherited functions Wins() and WinningMove() are defined in terms of this one. Function WinningMove chooses the winning move randomly, for more variety in the game:

```
class BachetsGame: public Game<int> {
//...
  virtual bool Wins() const
  {
    return Wins(state);
  }

  virtual int WinningMove() const
  {
    int i = 1 + ::rand() % static_cast<int>(moves.size());
    while (!ValidMove(moves[i]) || Wins(state - moves[i]))
      ++i %= moves.size();
    return moves[i];
  }
};
```

The remaining functions, RandomMove() and YourMove(), are rather straightforward:

```
class BachetsGame: public Game<int> {
//...
  virtual int RandomMove() const
  {
    int result;
    do
      result = moves[::rand()%moves.size()];
    while (!ValidMove(result));
    return result;
  }

  virtual int YourMove() const
  {
    int x;
    for (;;)
    {
      std::cout << "Your move: ";
      std::cin >> x;
      if (ValidMove(x))
        break;
```

```
    std::cout << "Illegal move. Repeat. " << std::endl;
   }
   return x;
  }
};
```

Here is a test function for our class. The number of stones is chosen randomly between 20 and 99:

```
void TestPlayMatches()
{
  BachetsGame bg;
  std::stringstream ss("3 1 3 8");
  bg.Read(ss);

  for(;;)
  {
    int reply;
    std::cout << "Play? No [0]. Yes, you (computer) start [1]. Yes, I (human) start [2]: ";
    std::cin >> reply;
    if (!std::cin)
      break;
    if (reply == 0)
      break;
    bg.Start(::rand() % 80 + 20);
    if (reply == 1)
      bg.PlayMatchIStart();
    else if (reply == 2)
      bg.PlayMatchYouStart();
    else
      std::cout << "Invalid reply. Repeat, please." << std::endl;
  }
}
```

In the previous test function, the largest possible state is 99, and the recursive solution we have behaves well enough even on a slow computer. Of course we can design a dynamic programming solution straightforwardly, and then be confident that our program holds on even for very large start states. As usual, we design a derived class with a vector for the computed results, a function WinsTabular (instead of OptimalTabular) which computes using the stored values, a function Compute to fill the vector and we redefine function Wins:

```
class BachetsGameDynamic: public BachetsGame {
private:
  std::vector<bool> wins;
private:
  virtual bool WinsTabular(int x) const
  {
    bool result = false;
    for (std::vector<int>::const_reverse_iterator j = moves.rbegin();
         !result && j  != moves.rend(); j++)
      if (x >= *j && !wins[x - *j])
        result = true;
    return result;
  }

public:
  virtual void Compute(int x)
  {
    wins.reserve(x+1);
    for (int i = 0; i <= x; i++)
      wins.push_back(WinsTabular(i));
  }

  virtual bool Wins(int x) const
```

```
  {
    return wins[x];
  }
};
```

This class is used like the other, the sole difference being that function Compute must be called initially for the largest expected start state.

Alternatively, we can use automatic memoization with an adapter, just like we did for the Fibonacci sequence. Here, the recursive function is function Win, and the fact that it is Boolean causes only a minor difficulty:

```
class BachetsGameAdapter: public BachetsGame {
public:
  virtual int Optimal(int x) const
  {
    return BachetsGame::Wins(x);
  }

  virtual bool Wins(int x) const
  {
    return Optimal(x) == 1;
  }
};
```

In order to use this in the test function, it is enough to replace the declaration of variable bg by the following:

```
  MemoVector<BachetsGameAdapter> bg;
```

9 Game 31

The 31 game is another game for two players. Initially there are six piles of four cards on the table. In the first pile all cards are ones, in the second pile all cards are twos, etc., and in the last pile all cards are sixes. Players alternately remove one card from one of the piles. The game ends when removing one more card would certainly make the sum of all removed cards exceed 31. The winner is the player who made the last move (leading to a state in which the sum of cards is still not greater than 31).

The class for this game, class Game31, inherits from class Game. The state is the current configuration of the piles of cards, which can be described by a vector of six numbers, cards. Recall that the template argument is the type of the state data member. We add a data member sum, that keeps the sum of the cards removed so far.

Once again, the crucial functions are Wins() and WinningMove(). All the other functions express directly the rules of the game:

```
class Game31: public Game<std::vector<int> > {
public:
  int sum;
  std::vector<int> cards;
public:

  virtual void Start(const std::vector<int>& x)
  {
    cards = x;
    sum = 0;
  }

  virtual bool ValidMove(int x) const
  {
    return 1 <= x && x <= 6 && cards[x-1] > 0 && x + sum <= 31;
  }
```

```
  virtual void Show() const
  {
    std::cout << sum << " - ";
    for (int i = 0; i < 6; i++)
      for (int j = 0; j < cards[i]; j++)
        std::cout << " " << i + 1;
    std::cout << std::endl;
  }

  virtual int YourMove() const
  {
    int x;
    for (;;)
    {
      std::cout << "Your move: ";
      std::cin >> x;
      if (ValidMove(x))
        break;
      std::cout << "Illegal move. Repeat. ";
    }
    return x;
  }

  virtual void Play(int x)
  {
    cards[x-1]--;
    sum += x;
  }

  virtual void Unplay(int x)
  {
    cards[x-1]++;
    sum -= x;
  }

  virtual bool MatchOver() const
  {
    int i = 0;
    while (cards[i] == 0)
      i++;
    return sum + i + 1 > 31;
  }
};
```

Function Wins is more complicated than in Bachet's game, because the state is a vector, not an integer. In class BachetsGame, function Wins was programmed in terms of another function Win(int x) where x represents the state. We could mimic that, by defining a function Win(const std::vector<int>& x) but we will do otherwise, defining the function directly in terms of the state of the game.

Function Wins(), with no arguments, returns true if the state is a winning state, i.e., if there is a move that leads the game to a non-winning state. Function Wins will call function Play to move to the next state (in order to check whether it is a non-winning state). This introduces a standard C++ hurdle: Wins is a const function, Play is not, hence Wins cannot call Play, in principle. The standard remedy is removing the "constness" of Play inside Wins. This is acceptable because any changes done to the state will have been undone when the function returns. The undoing is accomplished by function Unplay. Observe:

```
class Game31: public Game<std::vector<int> > {
//...
  virtual bool Wins() const
  {
    Game31* that = const_cast<Game31 *>(this);
```

```
    bool result = false;
    for (int i = 1; !result && i <= 6; i++)
      if (cards[i-1] > 0 && i + sum <= 31)
      {
        that->Play(i);
        if (!Wins())
          result = true;
        that->Unplay(i);
      }
    return result;
  }
};
```

Function WinningMove, which selects a random winning move, uses the same technique, but in a slightly different form:

```
class Game31: public Game<std::vector<int> > {
//...
  virtual int WinningMove() const // pre Wins();
  {
    Game31* that = const_cast<Game31*>(this);
    int i = 1 + ::rand() % 6;
    bool found;
    while (!ValidMove(i) || (that->Play(i), found = !Wins(), that->Unplay(i), !found))
      (i %= 6)++;
    return i;
  }
};
```

Function RandomMove chooses a valid move randomly:

```
class Game31: public Game<std::vector<int> > {
//...
  virtual int RandomMove() const
  {
    int x;
    do
      x = ::rand() % 6 + 1;
    while (!ValidMove(x));
    return x;
  }
};
```

Here is a test function similar to the one we used for Bachet's game:

```
void TestPlayMatches()
{
  Game31 g;
  for(;;)
  {
    int reply;
    std::cout << "Play? No [0]. Yes, you (computer) start [1]. Yes, I (human) start [2]: ";
    std::cin >> reply;
    if (!std::cin)
      break;
    if (reply == 0)
      break;
     g.Start(std::vector<int>(6, 4));
    if (reply == 1)
      g.PlayMatchIStart();
    else if (reply == 2)
      g.PlayMatchYouStart();
    else
      std::cout << "Invalid reply. Repeat, please." << std::endl;
  }
}
```

The performance of our class is acceptable, even on a slow computer. However, we could make it much more efficient using dynamic programming as in Bachet's game. There are some minor complications, however. Now, the state is not a number or a pair of numbers, but a vector with six elements. Should we use a table with six dimensions? Or should we number the states systematically (there are $6^4 = 1296$ different states) and represent each state by its number? Both approaches are feasible but both we will pursue a different one, which is much nicer: memoization using maps.

Initially the map is empty. Function results are added to the map as they are computed, the key being the argument of the function. If the argument is not present in the map, then the function has not been computed for that argument yet.

In the case of this game, the situation is slightly different from what we have just described. The function we want to memoize does not have an argument: it performs its computations directly on the state of the object (and not on a data member). Hence, the key to the map must be the state of the object. For that reason, we introduce a function in the class that yields the state:

```
class Game31: public Game<std::vector<int> > {
 //…
 const std::vector<int>& State() const
 {
   return cards;
 }
};
```

Here is the memo map for use with int functions Optimal that operate on the state of the object:

```
template <class T, class K>
class MemoMap: public T {
public:
 mutable std::map<K, int> optimal;
public:
 virtual int Optimal() const
 {
  int result;
  const K& x = State();
  std::map<K, int>::const_iterator i = optimal.find(x);
  if (i != optimal.end())
   result = i->second;
  else
   result = optimal[x] = T::Optimal();
  return result;
 }
};
```

This is not yet ready to use: we need an adapter because function Win is Boolean:

```
class Game31Adapter: public Game31 {
public:
 virtual int Optimal() const
 {
  return Game31::Wins();
 }

 virtual bool Wins() const
 {
  return Optimal() == 1;
 }
};
```

Memoization is now accomplished by declaring the variable representing the game object as follows:

```
 MemoMap<Game31Adapter, std::vector<int> > g;
```

10 Matrix Chain Multiplication

One of the most fascinating dynamic programming problems is that of minimizing the number of multiplications necessary for computing the chain product of several matrices. Recall that using the standard algorithm for matrix multiplication the number of multiplications performed when multiplying a *m*-by-*n* matrix by a *n*-by-*p* matrix is *m***n***p*. The resulting matrix is a *m*-by-*p* matrix. Suppose we have a third matrix, a *p*-by-*q* matrix. If we multiply the previous product by this matrix, we need *m***p***q* multiplications, and obtain a *m*-by-*q* matrix. In total, for the left to right product of the three matrices we shall have made *m***n***p* + *m***p***q* multiplications. Now suppose we compute the product from right to left. We multiply the *n*-by-*p* matrix by the *p*-by-*q* matrix, using *n***p***q* multiplications and obtaining a *n*-by-*q* matrix. Now we multiply the *m*-by-*n* matrix by the previous product, making *m***n***q* multiplications, and obtaining the same matrix as before using a total of *m***n***q* + *n***p***q* multiplications. In general this sum is different from the other. Hence, we should perform the multiplication from left to right if the first sum is less that the second sum, and from right to left, if not. For example if m=2, n=3, p=5 and q=7, the first sum is 100 and the second is 147.

For three matrices, the decision on how to multiply is Boolean. In general, for more than three matrices, there can be many combinations. We want to find out the best, i.e., the one that requires the least number of multiplications.

Suppose we have *N* matrices, *m*[0], *m*[1], …, *m*[N-1], with compatible dimensions (the number of columns of *m*[*i*] is equal to the number of rows of *m*[*i*+1]). Should we multiply *m*[0]*(*m*[1]*…*m*[*N*-1]) or (*m*[0]**m*[1])*(*m*[2]*…**m*[N-1]), or, in general (*m*[0]*…**m*[*i*])*(*m*[*i*+1]*…**m*[*N*-1]), for *i* varying from 0 to *N*-2? In other words, where do we break the list of matrices to insert the multiplication operator? Well, we should break it at the position that minimizes the total number of multiplications. Once we know how to do that, we can apply the same strategy recursively to the separated halves of the list.

We are back to the usual question: what is the optimal number of multiplications when computing the product of those *N* matrices? In general, what is the optimal number of multiplications necessary to compute the product of *m*[*x*], *m*[*x*+1], …, *m*[*y*]? Once we know that, computing the breaking position is simple: just consider all possibilities, 0, 1, 2…, *N*-2, and choose the one that minimizes the number of multiplications.

Note that, in general, if we want to compute the product of matrices *m*[*x*], *m*[*x*+1], …, *m*[*y*], and we break at *s*, the first half will be the product of *m*[*x*], *m*[*x*+1], … *m*[*s*], which is a matrix with the same number of rows as *m*[*x*] and the same number of columns as *m*[*s*]. Likewise, the second half will be the product of *m*[*s*+1], …, *m*[*y*-1], *m*[*y*], which is a matrix with the same number of rows as *m*[*s*+1] and the same number of columns as *m*[*y*]. Hence, multiplying the product of *m*[*x*], *m*[*x*+1], …, *m*[*s*] by the product of *m*[*s*+1], …, *m*[*y*-1], *m*[*y*] requires *rx***cs***cy* multiplications, where *rx* is the number of rows of *m*[*x*], *cs* is the number of columns of *m*[*s*] (which is equal to the number of rows of the next matrix, *m*[*s*+1]), and *cy* is the number of columns of *m*[*y*].

So much for the theory. Let us set up our classes. We are not actually computing any products, we are only counting multiplications. Therefore, we do not care about the matrix coefficients. We care only about the number of rows and the number of columns. Although it is not essential, each matrix will have a name, which is just a string without spaces. We represent matrices with class Matrix:

```
class Matrix {
public:
  std::string name;
  int r;  // number of rows
  int c;  // number of columns
};
```

Class MatrixChainMultiplication is the class for our problem. Its data member is the vector of matrices whose product we want. There's a Read function to read a matrix from a file:

```
class MatrixChainMultiplication {
public:
  std::vector<Matrix> m;
public:
  virtual void Read(std::istream& input)
  {
    m.clear();
    int n;
    input >> n;
    if (!input)
      return;
    m.reserve(n);
    for (int i = 0; i < n; i++)
    {
      Matrix x;
      input >> x.name >> x.r >> x.c;
      m.push_back(x);
    }
  }
};
```

First comes the function that computes the cost of multiplying the product of m[x], m[x+1], …, m[s] by the product of m[s+1], …, m[y-1], m[y], as explained above:

```
class MatrixChainMultiplication {
//…
  virtual int Cost(int x, int s, int y) const
  {
    return m[x].r * m[s].c * m[y].c;
  }
};
```

Let us now define function Optimal(int x, int y), which computes the optimal cost of multiplying matrices m[x], m[x+1], …, m[y]. It relies on function BreakAt(int x, int y) for computing the separation position:

```
class MatrixChainMultiplication {
//…
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == y)
      result = 0;
    else
    {
      int s = BreakAt(x, y);
      result = Optimal(x, s) + Optimal(s+1, y) + Cost(x, s, y);
    }
    return result;
  }
};
```

Function BreakAt merely searches linearly the best separating position between x and y-1:

```
class MatrixChainMultiplication {
//…
  virtual int BreakAt(int x, int y) const // break between result and result+1
  {
    int result = x;
    int min = std::numeric_limits<int>::max();
    for (int i = x; i < y; i++)
    {
```

```
      int temp = Optimal(x, i) + Optimal(i+1, y) + Cost(x, i, y);
      if (min > temp)
      {
        min = temp;
        result = i;
      }
    }
    return result;
  }
};
```

Now we know what the optimal cost is. How would we actually multiply the matrices, if we had to? In previous problems, the solution was given by function Solution, returning a list. In this case a list is not sufficient: we would need a tree. For lack of a readily available class for trees, we will represent the solution as a fully parenthesized string, for example ((A*B)*C), with the usual meaning:

```
class MatrixChainMultiplication {
//...
  virtual std::string Solution(int x, int y) const
  {
    std::string result;
    if (x == y)
      result = m[x].name;
    else
    {
      int s = BreakAt(x, y);
      result = "(" + Solution(x, s) + "*" + Solution(s+1, y)+ ")";
    }
    return result;
  }
};
```

That's it.

The recursive solution performs well only for a small number of matrices. Anyway, we can use automatic memoization directly. Here is a test function that illustrates the technique. It reads some matrices from a file and then computes the optimal cost and the solution for each of them. For fun, and to appreciate the real interest of this method, we also show the number of multiplications which would be necessary to multiply the matrices from left to right and from right to left:

```
void TestMatrixMultiplication(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    //MatrixChainMultiplication m;
    MemoMatrix<MatrixChainMultiplication> x;
    x.Read(input);
    if (!input)
      return;
    int a = x.Optimal(0, static_cast<int>(x.m.size() - 1));
    int b = x.LeftToRight(0, static_cast<int>(x.m.size() - 1));
    int c = x.RightToLeft(0, static_cast<int>(x.m.size() - 1));
    output << a << " " << b << " " << c << std::endl;
    output << x.Solution(0, static_cast<int>(x.m.size() - 1)) << std::endl;
  }
}
```

We tried our program with the following input file:

```
6
A 4 2 B 2 3 C 3 1 D 1 2 E 2 2 F 2 3
6
A 30 35 B 35 15 C 15 5 D 5 10 E 10 20 F 20 25
```

```
3
A 2 3 B 3 5 C 5 7
3
A 7 5 B 5 3 C 3 2
12
A 2 5 B 5 3 C 3 6 D 6 1 E 1 5 F 5 4 G 4 3 H 3 8 I 8 2 J 2 5 K 5 5 L 5 2
```

This is what we got:

```
36 84 69
((A*(B*C))*((D*E)*F))
15125 40500 47500
((A*(B*C))*((D*E)*F))
100 100 147
((A*B)*C)
100 147 100
(A*(B*C))
164 322 322
((A*(B*(C*D)))*(((((((E*F)*G)*H)*I)*J)*K)*L))
215 408 1176
((A*(B*(C*D)))*((((((((((E*F)*G)*H)*I)*J)*K)*L)*M)*N)*O))
```

The two extra functions, LeftToRight and RightToLeft, are simple:

```
class MatrixChainMultiplication {
//...
  virtual int LeftToRight(int x, int y) const
  {
    int result = 0;
    int p = m[x].r;
    for (int i = x+1; i <= y; i++)
      result += p * m[i].r * m[i].c;
    return result;
  }

  virtual int RightToLeft(int x, int y) const
  {
    int result = 0;
    int p = m[y].c;
    for (int i = y-1; i >= x; i--)
      result += p * m[i].r * m[i].c;
    return result;
  }
};
```

A brief analysis of the program shows that the running time for the memoized version is proportional to N^3, N being the number of matrices. The space required is proportional to N^2 (that's the size of the table in class MemoMatrix). Note that each item in this matrix is computed once, and this contributes with a factor of N^2. Each of this N^2 computations requires a loop in function BreakAt, hence N^3.

11 Longest Common Substring

The problem is: given two strings $s1$ and $s2$, compute the longest string which is a substring of both $s1$ and $s2$. In case there are several solutions, any one will do.

We need a clarification here. In this context, we say that a string s is a subtring of a string r if all individual characters of s exist in r in the same order. This contrasts with the meaning of function substr (read "substring") from class std::string, used s.substr(x, y) which computes a string formed by the characters s[x], s[x+1], …, s[x+y-1] in this order.

Given the clue that this is a dynamic programming problem (in fact, it is one of the most popular) we will use the standard strategy: define a function Optimal for the length of the longest

common substring and then a function Solution which works by undoing the computation performed by function Optimal.

Here's the draft of our new class:

```
class LongestCommonSubstring {
public:
  std::string s1;
  std::string s2;
public:

  virtual int Optimal(int x, int y) const
  {
    //…
  }

  std::string Solution(int x, int y) const
  {
    //…
  }
};
```

Function Optimal(int x, int y) will compute the length of the longest common substring of the prefix of s1 having length x and the prefix of s2 having length y. Given a string *s*, the prefix of *s* having length *x* is the string formed by the first *x* characters of *s* (in the same order).

Before we program this function, let us tackle the problem directly, if not very efficiently. This is one of the many occasions in which programming language is clearer than natural language:

```
class LongestCommonSubstring {
//…

  virtual std::string Longest(const std::string& x, const std::string& y) const
  {
    return x.size() >= y.size() ? x : y;
  }

  virtual char Last(const std::string& x) const
  {
    return x[x.size()-1];
  }

  virtual std::string Head(const std::string& x) const
  {
    return x.substr(0, x.size() - 1);
  }

  virtual std::string Direct(const std::string& x, const std::string& y) const
  {
    if (x.empty() || y.empty())
      return std::string();
    else if (Last(x) == Last(y))
      return Direct(Head(x), Head(y)) + Last(x);
    else
      return Longest(Direct(Head(x), y), Direct(x, Head(y)));
  }
};
```

Function Longest returns the longest of the two string arguments, function Last returns the last character of a non-empty string and function Head returns the string that is equal to the argument (which is a non-empty string) minus the last character. Function Direct computes the longest common substring of its two arguments.

As a matter of fact, function Optimal mimics the behavior of function Direct. Observe:

```
class LongestCommonSubstring {
//...
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == 0 || y == 0)
      result = 0;
    else if (s1[x-1] == s2[y-1])
      result = Optimal(x-1, y-1) + 1;
    else
      result = Max(Optimal(x-1, y), Optimal(x, y-1));
    return result;
  }
};
```

How do we undo this in order to compute the solution? Let us try out a few examples, by printing out all values of function Optimal, using the following function:

```
class LongestCommonSubstring {
//...
  void WriteOptimal(std::ostream& output )
  {
    output << "    ";
    for (int j = 0; j < static_cast<int>(s2.size()); j++)
      output << " " << s2[j];
    output << std::endl;
    output << "  ";
    for (int j = 0; j <= static_cast<int>(s2.size()); j++)
      output << " " << Optimal(0, j);
    output << std::endl;
    for (int i = 1; i <= static_cast<int>(s1.size()); i++)
    {
      output << " " << s1[i-1];
      for (int j = 0; j <= static_cast<int>(s2.size()); j++)
        output << " " << Optimal(i, j);
      output << std::endl;
    }
  }
};
```

This is what we get when s1 is "abcz" and s2 is "dacabz" (and the result is "abz"):

```
     d a c a b z
   0 0 0 0 0 0 0
 a 0 0 1 1 1 1 1
 b 0 0 1 1 1 2 2
 c 0 0 1 2 2 2 2
 z 0 0 1 2 2 2 3
```

Look at the lower right corner. There is one 3 and three 2. This occurs because the last letter is the same in both strings. This letter must belong to the solution. After we collect it, we can remove it from the strings and apply the same reasoning to the table after we remove the last row and the last column.

Now consider the case where s1 is "abcdzd" and s2 is "dacz" (and the result is "acz"):

```
     d a c z
   0 0 0 0 0
 a 0 0 1 1 1
 b 0 0 1 1 1
 c 0 0 1 2 2
 d 0 1 1 2 2
 z 0 1 1 2 3
 d 0 1 1 2 3
```

Look at the lower right corner again. Clearly we can ignore the last row, because in the next step we will still be dealing with a substring of length 3. In other words, the depicted situation means

that the last letter of s1 does not belong to the solution. It also means that the last letter of s2 belongs to the solution, but that fact will be dealt with in a subsequent step, when we find a situation similar to the first one (three small numbers and one large number).

The third case is similar. Now, s1 is "abcdz" and s2 is "daczb" (and the result is "acz"):

```
    d a c z b
  0 0 0 0 0 0
a 0 0 1 1 1 1
b 0 0 1 1 1 2
c 0 0 1 2 2 2
d 0 1 1 2 2 2
z 0 1 1 2 3 3
```

By a similar argument we conclude that the last column can be removed.

Now, consider the case where s1 is "abcdad" and s2 is "dacabda":

```
    d a c a b d a
  0 0 0 0 0 0 0 0
a 0 0 1 1 1 1 1 1
b 0 0 1 1 1 2 2 2
c 0 0 1 2 2 2 2 2
d 0 1 1 2 2 2 3 3
a 0 1 2 2 3 3 3 4
d 0 1 2 2 3 3 4 4
```

In this case, we can remove either the last row or the last column, but not both. The resulting strings might not be the same in the end, but the length is 4, certainly. For instance, if in situations like this we decide to remove the last row, we will get the solution "abda". If we decide to remove the last column, we will get "acad". Both are legitimate solutions. We do not have to choose always the last row or always the last column, and if we do not, we may get still other solutions.

Finally consider the case where s1 is "abcdbc" and s2 is "dacabda":

```
    d a c a b d a
  0 0 0 0 0 0 0 0
a 0 0 1 1 1 1 1 1
b 0 0 1 1 1 2 2 2
c 0 0 1 2 2 2 2 2
d 0 1 1 2 2 2 3 3
b 0 1 1 2 2 3 3 3
c 0 1 1 2 2 3 3 3
```

In this case we could remove the last row and the last column. However, we do not need to handle this case separately. We can assimilate it to the previous one.

This detailed analysis translates to the following design:

```
class LongestCommonSubstring {
//…
  std::string Solution(int x, int y) const
  {
    std::string result;
    while (x > 0 && y > 0)
      if (Optimal(x, y) > Max(Optimal(x-1, y), Optimal(x, y-1)))
      {
        result += s1[x-1];
        x--, y--;
      }
      else if (Optimal(x-1, y) >= Optimal(x, y-1))
        x--;
      else
        y--;
    std::reverse(result.begin(), result.end());
    return result;
  }
```

```
};
```

The `>=` in the second `if` statement means that in the situation of the fourth above case we are removing the last row. Note that the string is built in reverse order, hence the final `reverse` operation.

We can use this class directly only for small examples. For large strings, we use the memoized version, of course. Here an example, for interactive use:

```
void TestLongestCommonSubstringInteractive()
{
 for (;;)
 {
  MemoMatrix<LongestCommonSubstring> lcs;
  std::cout << "s1 = ";
  std::getline(std::cin, lcs.s1);
  if (!std::cin)
   break;
  std::cout << "s2 = ";
  std::getline(std::cin, lcs.s2);
  int length = lcs.Optimal(static_cast<int>(lcs.s1.size()), static_cast<int>(lcs.s2.size()));
  std::string solution =
   lcs.Solution(static_cast<int>(lcs.s1.size()), static_cast<int>(lcs.s2.size()));
  std::cout << solution << "/" << length << std::endl;
 }
}
```

The running time of function `Solution` in the memoized version is proportional to $N*M$, N being the size of one string and M the size of the other. This is because it computes function `Optimal` once for each item in the table, at the most. The running time of the loop in function `Solution` modulus the running time of the calls of `Optimal` is proportional to $N+M$. Hence the total running time is dominated by the factor $N*M$.

12 Edit Distance

Another dynamic programming problem with strings is finding the *edit distance*: given two strings, `s1` and `s2`, what is the optimal sequence of edit operations that transform `s1` into `s2`?

Which edit operations are available? The general idea is that we start with `s1` and an empty string, `result`. We move characters from `s1` to `result`, from left to right, in such a way that in the end `result` is equal to `s2`. When moving characters from `s1` to `result`, several things may happen: for example, the character can be replaced by another character (that's operation "replace"); or the character may get lost and not be appended to `result` (that's operation "delete"). In this exercise we will use the set of operations presented in the book *Introduction to Algorithms*, where this problem is given as an exercise [Cormen *et al.*, page 364]. They are as follows:

1. **Copy**: a character from `s1` is copied to `result`.
2. **Replace**: a character from `s1` moves to `result` but is replaced by another character before it arrives.
3. **Delete**: a character from `s1` moves to `result` but does not get there.
4. **Insert**: a character is inserted in `result` that does not come from `s1`.
5. **Twiddle**: two characters move from `s1` to `result` and arrive in reverse order.
6. **Kill**: all the remaining characters from `s1` move to `result` in one operation but disappear before they arrive.

Each of these operations has a cost. The cost of the transformation is the sum of costs of the operations involved. The edit distance is the minimum possible cost.

Let us set up our class:

```
class EditDistance {
public:
 std::string s1;
```

```
  std::string s2;
public:
  virtual int PenaltyDelete(char) const
  {
    return 1;
  }

  virtual int PenaltyCopy(char) const
  {
    return 0;
  }

  virtual int PenaltyReplace(char, char) const
  {
    return 1;
  }

  virtual int PenaltyInsert(char) const
  {
    return 1;
  }

  virtual int PenaltyTwiddle(char, char) const
  {
    return 1;
  }

  virtual int PenaltyKill(int x) const
  {
    return x == 0 ? 0 : 1;
  }
};
```

All these penalty functions except the last return a constant. In general, the result may depend on the characters given as arguments. For example, we may decide that it is more expensive to replace 'p' by 'z' which are far apart in the keyboard, than to replace 't' by 'y', which are side by side. Likewise, for function PenaltyKill, whose argument is an integer, we may decide that it is more expensive to kill two characters than to kill just one.

How do we express the optimal cost of transforming the first x characters of s1 by the first y characters of s2? Let us not consider the kill operation in this phase. Observe. It is longer than usual but very systematic:

```
class EditDistance {
//...
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == 0 && y == 0)
      result = 0;
    else if (x == 0)
      result = Optimal(0, y-1) + PenaltyInsert(s2[y-1]);
    else if (y == 0)
      result = Optimal(x-1, 0) + PenaltyDelete(s1[x-1]);
    else
    {
      int z1 = Optimal(x, y-1) + PenaltyInsert(s2[y-1]);
      int z2 = Optimal(x-1, y) + PenaltyDelete(s1[x-1]);
      int z3 = Optimal(x-1, y-1) + (s1[x-1] == s2[y - 1] ?
                                    PenaltyCopy(s1[x-1]) :
                                    PenaltyReplace(s1[x-1], s2[y-1]));
      int z4 = x > 1 && y > 1 && s1[x-2] == s2[y-1] && s1[x-1] == s2[y-2] ?
                                    Optimal(x-2, y-2) + PenaltyTwiddle(s1[x-2], s2[y-2]) :
                                    std::numeric_limits<int>::max();
```

```
    result = mas::Min(mas::Min(z1, z2), mas::Min(z3, z4));
   }
   return result;
  }
};
```

The kill operation is used at most once, by nature, and in that case it is the last operation. Therefore, it does not need to be handled recursively. We merely consider all possibilities and choose the best:

```
class EditDistance {
//...
  int OptimalKill(int x, int y) const
  {
   int result = Optimal(x, y);
   for (int i = 0; i < x; i++)
    result = mas::Min(result, Optimal(i, y) + PenaltyKill(x - i));
   return result;
  }
};
```

It is this function that computes the edit distance. As usual, we do not stop here. We want to actually transform `s1` into `s2`. The transformation is the sequence of edit operations and we will represent it by a string made up of the letters 'C' for copy, 'D' for delete, 'I' for insert, 'K' for kill, 'R' for replace and 'T' for twiddle. Let us consider first the solution without the kill operation:

```
class EditDistance {
//...
  virtual std::string Solution(int x, int y) const
  {
   std::string result;
   while (x > 0 && y > 0)
    if (Optimal(x, y) == Optimal(x, y - 1) + PenaltyInsert(s2[y - 1]))
    {
     result.append("I");
     y--;
    }
    else if (Optimal(x, y) == Optimal(x - 1, y) + PenaltyDelete(s1[x - 1]))
    {
     result.append("D");
     x--;
    }
    else if (x > 1 && y > 1 && s1[x-2] == s2[y-1] && s1[x-1] == s2[y-2]
            && Optimal(x, y) == Optimal(x-2, y-2) + PenaltyTwiddle(s1[x-2], s2[y-2]))
    {
     result.append("T");
     x -= 2, y -= 2;
    }
    else
    {
     result.append(s1[x-1] == s2[y-1] ? "C" : "R");
     x--, y--;
    }
   while (y > 0)
   {
    result.append("I");
    y--;
   }
   while (x > 0)
   {
    result.append("D");
    x--;
   }
```

```
    std::reverse(result.begin(), result.end());
    return result;
  }
};
```

Now, the complete solution, using the kill operation as well:

```
class EditDistance {
//...
  virtual std::string SolutionKill(int x, int y) const
  {
    int s = OptimalKill(x, y);
    int i = 0;
    while (s != Optimal(i, y) + PenaltyKill(x - i))
      i++;
    std::string result = Solution(i, y);
    if (i != x)
      result += 'K';
    return result;
  }
};
```

The following functions solve the problem, by actually computing the edit distance and the optimal transformation for the given strings, without kill and with kill:

```
class EditDistance {
//...
  virtual int Optimal0() const
  {
    return Optimal(static_cast<int>(s1.size()), static_cast<int>(s2.size()));
  }

  virtual int OptimalKill0() const
  {
    return OptimalKill(static_cast<int>(s1.size()), static_cast<int>(s2.size()));
  }

  virtual std::string Solution() const
  {
    return Solution(static_cast<int>(s1.size()), static_cast<int>(s2.size()));
  }

  virtual std::string SolutionKill() const
  {
    return SolutionKill(static_cast<int>(s1.size()), static_cast<int>(s2.size()));
  }
};
```

In order to appreciate what we have done, we need a function to perform the transformation:

```
class EditDistance {
//...
  virtual void Transform(const std::string& t, std::ostream& output)
  {
    int x = 0;
    int y = 0;
    std::string s;
    for (int i = 0; i < static_cast<int>(t.size()); i++)
    {
      if (t[i] == 'I')
        s += s2[y++];
      else if (t[i] == 'D')
        x++;
      else if (t[i] == 'C' || t[i] == 'R')
      {
        s += s2[y++];
```

```
      x++;
     }
     else if (t[i] == 'T')
     {
      s += s2[y++];
      s += s2[y++];
      x += 2;
     }
     else if (t[i] =='K')
     {
      // nothing to do. 'K' is last character in t.
     }
     output << s1 << " " << x << " " << s2 << " " << y << " " << s << std::endl;
    }
   }
};
```

At each step, the function displays on the output file the two strings, the current positions and the result string that is being built.

We conclude this exercise with an interactive test function that uses class EditDistance with automatic memoization:

```
void TestEditDistanceInteractive()
{
 for (;;)
 {
  MemoMatrix<EditDistance> ed;
  std::cout << "s1 = ";
  std::getline(std::cin, ed.s1);
  if (!std::cin)
   break;
  std::cout << "s2 = ";
  std::getline(std::cin, ed.s2);
  int cost = ed.Optimal0();
  std::string solution = ed.Solution();
  std::cout << solution << "/" << cost << std::endl;
  ed.Transform(solution, std::cout);
  std::cout << std::endl;
 }
}
```

Here's the result of an experiment:

```
s1 = aabbaa
s2 = bababa
RCCTC/2
aabbaa 1 bababa 1 b
aabbaa 2 bababa 2 ba
aabbaa 3 bababa 3 bab
aabbaa 5 bababa 5 babab
aabbaa 6 bababa 6 bababa

s1 = program
s2 = algorithm
RRTCRIIC/6
program 1 algorithm 1 a
program 2 algorithm 2 al
program 4 algorithm 4 algo
program 5 algorithm 5 algor
program 6 algorithm 6 algori
program 6 algorithm 7 algorit
program 6 algorithm 8 algorith
program 7 algorithm 9 algorithm
```

```
s1 = similar
s2 = limit
RCCCIK/3
similar 1 limit 1 l
similar 2 limit 2 li
similar 3 limit 3 lim
similar 4 limit 4 limi
similar 4 limit 5 limit
similar 4 limit 5 limit
```

13 Euro Change

There are eight euro coins: 1 cent, 2 cents, 5 cents, 10 cents, 20 cents, 50 cents, 1 euro and 2 euros. The smallest note is for 5 euros. Therefore any amount up to €4.99 must be made up of coins only. The problem is finding out in how many different ways we can make up a certain amount less than five euros supposing we have an unlimited supply of coins.

This does not look like a dynamic programming problem, since there is no notion of something being optimal. It seems just a matter of counting combinations. So, let's first try the brute force approach and compute the actual combinations:

```
class EuroChange {
private:
   static const int values[];
public:
  virtual ~EuroChange()
  {
  }

  virtual int SolutionCombinatorial(int x, int y) const
  {
    EuroChange::Combinations c(*this, x, y);
    c.Compute();
    return static_cast<int>(c.combinations.size());
  }

public:
  class Combinations {
  public:
    const EuroChange& ec;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int amount;
    int limit;
    int value;
  public:
    Combinations(const EuroChange& ec, int amount, int limit):
      ec(ec),
      amount(amount),
      limit(limit)
    {
    }

    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      value = 0;
      for (int i = 1; i <= limit; i++)
        if (values[i-1] <= amount)
          Visit(i);
    }
```

```
    void operator = (const Combinations&){}

  private:
    virtual void Visit(int x)
    {
      temp.push_back(x);
      value += values[x-1];
      if (value == amount)
        combinations.push_back(temp);
      else
        for (int i = x; i <= limit; i++)
          if (value + values[i-1] <= amount)
            Visit(i);
      value -= values[x-1];
      temp.pop_back();
    }
  };
};

const int EuroChange::values[] = {1, 2, 5, 10, 20, 50, 100, 200};
```

The other technique for counting the combinations is finding a formula for the number of combinations of value x (in cents), using only the y first coins. Let's call this function Optimal, so that we can use automatic memoization with it, even if there is nothing optimal about it:

```
class EuroChange {
//...
  // number of combinations to make up x cents using the y lowest coins
  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == 0 || y == 1)
      result = 1;
    else
      result = Optimal(x, y-1) + (x >= values[y-1] ? Optimal(x - values[y-1], y): 0);
    return result;
  }
};
```

Actually, this is quite similar to the Optimal function for the knapsack problem.

Here is a test function, with automatic memoization. If the amount is not greater than 20 cents, it computes and displays the combinations.

```
void TestEuroChange(std::istream& input, std::ostream& output)
{
  MemoMatrix<EuroChange> ec;
  for (;;)
  {
    int amount;
    input >> amount;
    if (!input)
      break;
    int x = ec.Optimal(amount, 8);
    output << x << std::endl;
    if (amount <= 20)
    {
      EuroChange::Combinations c(ec, amount, 8);
      c.Compute();
      mas::WriteLine(c.combinations, "\n", output);
      int y = ec.SolutionCombinatorial(amount, 8);
      output << y << std::endl;
    }
    output << std::endl;
  }
```

```
}
```

Here is the result of an experiment:

```
100
4563

499
6224452

4
3
1 1 1 1
1 1 2
2 2
3

12
15
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 3
1 1 1 1 1 1 2 2 2
1 1 1 1 1 2 3
1 1 1 1 2 2 2 2
1 1 1 2 2 3
1 1 2 2 2 2 2
1 1 3 3
1 1 4
1 2 2 2 3
2 2 2 2 2 2
2 3 3
2 4
15
```

For example, the combination <1 2 2 2 3> represents one one cent coin, three two cents coins and one five cents coin.

14 Lorries

There's this big ancient monument that has to be relocated because of some environmental requirement. As the building is dismantled, the stone blocks are numbered from zero (the chief engineer has a strong C programming background) and loaded onto lorries that take them to the new site. All lorries are equal and they can carry blocks up to a certain load. The blocks are of various weights and must be loaded sequentially onto the lorries, i.e., blocks 0, …, *N*, go on the first lorry, blocks *N*+1, …, *M* go on the second lorry, etc., to ensure that they are not lost. Not all lorries will be loaded to their full capacity, of course, but it is guaranteed that no block is too heavy for the lorries. We do not want to load the lorries greedily (load as many blocks as you can on *this* lorry and when the next block does not fit, start a new lorry) because we may end up with a rather unbalanced convoy. Instead, we want to minimize the load factor of the convoy, the load factor being, by definition, the sum of squares of the unused capacity of each lorry.

Given the capacity of the lorries and sequence of weights of the blocks, how many blocks should each lorry carry, and how many lorries are required?

This is a variation of a problem used in a local programming competition at my university, suggested by my colleague Margarida Mamede, with inspiration from the Printing Neatly problem, which will appear a little later in this report.

The brute force approach is to try out all combinations and pick the best. Most likely this will not be feasible unless the number of blocks is small, but the exercise can be enlightening. Let's do it, as a warm up.

Each combination is a list of integers. We need a function Cost for computing the cost of a list, i.e., the load factor of the convoy. Function Cost uses function Penalty(int x, int y) which computes the partial cost of carrying stones, x, x+1, …, y in a lorry. If total weight of the blocks is too much for the lorry, the penalty is infinite.

Here's the beginning of our class, with the data members, the destructor, function Read, function Cost, function Penalty and also function MostValuable, which finds the most valuable combination in a list of combinations. In this problem, the most valuable combination is the one with the minimum cost. All these functions are quite straightforward:

```
class Lorries {
public:
  int capacity;  // capacity of each lorry
  std::vector<int> blocks;
  int size;

public:
  virtual ~Lorries()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> capacity >> size;
    if (!input)
      return;
    blocks.clear();
    blocks.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
      input >> x;
      blocks.push_back(x);
    }
  }

  // Cost of carrying blocks x, x+1, ..., y in a lorry. First block is block 0.
  // If the block overflow, the cost is infinite.
  virtual int Penalty(int x, int y) const
  {
    int sum = std::accumulate(blocks.begin() + x, blocks.begin() + y + 1, 0);
    int result = sum <= capacity ? mas::Square(capacity - sum) :
                                   std::numeric_limits<int>::max();
    return result;
  }

  virtual int Cost(const std::list<int>& x) const  // cost of a given combination
  {
    int result = 0;
    int j = 0;
    for (std::list<int>::const_iterator i = x.begin();
         i != x.end() && result != std::numeric_limits<int>::max(); i++)
    {
      int temp = Penalty(j, j + *i - 1);
      result = temp == std::numeric_limits<int>::max() ? temp : result + temp;
      j += *i;
    }
    return result;
  }
```

```
  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    int min = std::numeric_limits<int>::max();
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (min > temp)
      {
        min = temp;
        result = i;
      }
    }
    return *result;
  }
};
```

As usual, we will now define a inner class Combinations with a function Compute to actually generate all combinations. Function SolutionCombinatiorial which yields the combinatorial solution uses that function to generate all the combinations and then function MostValuable to pick the best. The generation pattern for the combinations is simple: after inserting the *n*-th block, we recursively insert the *n*+1-th block in the same lorry, if it still fits there, and in the next lorry (where it certainly fits, because it is the first block in that lorry):

```
class Lorries {
//...
  virtual std::list<int> SolutionCombinatorial() const
  {
    std::list<int> result;
    Combinations c(*this);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

public:
  class Combinations {
  public:
    const Lorries& r;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int count;
  public:
    Combinations(const Lorries& r):
      r(r)
    {
    }

    virtual ~Combinations()
    {
    }

    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      count = 0;
      Visit(0, false);
    }
```

```
    void operator = (const Combinations&){}

  private:
    virtual void Visit(int x, bool next)
    {
      if (next)
      {
        temp.push_back(count);
        count = 1;
      }
      else
        count++;
      if (x == r.size - 1)
      {
        temp.push_back(count);
        combinations.push_back(temp);
        temp.pop_back();
      }
      else
      {
        int sum =
            std::accumulate(r.blocks.begin() + x + 1 - count, r.blocks.begin() + x + 1, 0);
        Visit(x+1, true);
        if (sum + r.blocks[x+1] <= r.capacity)
          Visit(x+1, false);
      }
      if (next)
      {
        count = temp.back();
        temp.pop_back();
      }
      else
        count--;
    }
  };
};
```

The pattern is clearly exponential.

Let us experiment with a small example. Here's the test function:

```
void TestLorriesCombinatorial(std::istream& input, std::ostream& output)
{
  Lorries r;
  r.Read(input);
  Lorries::Combinations lc(r);
  mas::WriteLine(r.blocks, " ", output);
  lc.Compute();
  mas::WriteLine(lc.combinations, "\n", output);
  output << std::endl;
  const std::list<int>& solution = r.MostValuable(lc.combinations);
  output << solution << "/" << r.Cost(solution) << std::endl;
}
```

Here's the input file:

```
6  5    1 2 3 1 1
```

(The capacity of the lorries is 6 and there are 5 blocks with weights 1, 2, 3, 1 and 1.)

And here's what we get:

```
1 1 1 1 1
1 1 1 2
1 1 2 1
1 1 3
```

```
1 2 1 1
1 2 2
1 3 1
2 1 1 1
2 1 2
2 2 1
2 3
3 1 1
3 2

2 3/10
```

The last line shows the best combination and its cost. The others list all the combinations.

Let us now use the dynamic programming strategy. We are not loading the lorries greedily, and so we must not ask in which lorry should the x-th block be loaded, after the preceding x-1 blocks have been loaded. Instead, we should ask what is the optimal cost for loading x blocks.

In the optimal solution with x blocks, the last lorry will carry either one block (the x-th block) or 2 blocks (the x-th block and the x-1-block), or 3 blocks, etc., provided those blocks together do not exceed the capacity of the lorry. If the last lorry carries i blocks, and that is the optimal solution, the remaining lorries carry x-i blocks, of course, and they should carry those blocks optimally. Therefore, we should choose the value of i such that Optimal(x-i) + Penalty(x-i, x-1) is minimal. By choosing the appropriate i, we get the value of Optimal(x):

```
class Lorries {
//
  virtual int Optimal(int x) const
  {
    int result = 0;
    if (x > 0)
    {
      result = std::numeric_limits<int>::max();
      for (int i = 1; i <= x && Penalty(x-i, x-1) != std::numeric_limits<int>::max(); i++)
        result = mas::Min(result, Optimal(x-i) + Penalty(x-i, x-1));
    }
    return result;
  }
};
```

This function is optimized to exit the loop as soon as the penalty becomes infinite, because this is a sign that we cannot load any more blocks on the last lorry.

We can now compute the solution from the last lorry backwards:

```
class Lorries {
//
  virtual std::list<int> Solution(int x) const // optimal solution for carrying x blocks.
  {
    std::list<int> result;
    while (x > 0)
    {
      int i = 1;
      while (Optimal(x-i) + Penalty(x-i, x-1) > Optimal(x))
        i++;
      result.push_front(i);
      x -= i;
    }
    return result;
  }
};
```

We acknowledge that these functions make heavy use of recursive computations. We already know that that is not a problem, because we can either redefine function Optimal in a dynamic programming derived class or use automatic memoization.

15 Train

Let us consider a small variant of problem Lorries. Now, instead of a fleet of lorries, we have a train with a fixed number of cars. Does the problem change at all?

Yes, it does. Take two blocks of weight 1 and lorries of capacity 2. A single lorry is enough, and the cost is zero. Now, take a train with two cars, each car with capacity 2, just like the lorries. Loading the cars as if they were lorries would leave the second car empty, for a cost of 4; however, if we put one block in each car, the cost is 2.

In this problem we are given the capacity of the car, the number of cars, the number of blocks and the weight of each block. We can suppose, without loss of generality, that the number of blocks is greater or equal to the number of cars (otherwise, the extra cars will be empty), but we have to consider the possibility that the train is too short and cannot load all the blocks. In that case, there is no solution. Before, the number of lorries was not limited, and therefore, there was always a solution. Note that, like before, we suppose that the blocks are not too heavy, i.e., that their weight is not greater than the capacity of the cars.

Here is our class Train:

```
class Train {
public:
  int capacity;  // capacity of each car
  int cars; // number of cars
  std::vector<int> blocks;
  int size;

public:
  virtual ~Train()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> capacity >> cars >> size;
    if (!input)
      return;
    blocks.clear();
    blocks.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
      input >> x;
      blocks.push_back(x);
    }
  }

  // Cost of carrying blocks x, x+1, ..., y in a car. First block is block 0.
  // If the blocks overflow, the cost is infinite.
  virtual int Penalty(int x, int y) const
  {
    int sum = std::accumulate(blocks.begin() + x, blocks.begin() + y + 1, 0);
    int result = sum <= capacity ? mas::Square(capacity - sum)
                                 : std::numeric_limits<int>::max();
    return result;
  }

  // Optimal cost of carrying x blocks using y cars. First block is block 0.
  // Note that Optimal(x, y) can be infinite.
  virtual int Optimal(int x, int y) const
  {
    //...
  }
```

```
  virtual std::list<int> Solution(int x, int y) const
  // optimal solution for carrying x blocks using y cars.
  {
    //…
  }

  virtual int Cost(const std::list<int>& x) const  // cost of a given combination
  {
    int result = 0;
    int j = 0;
    for (std::list<int>::const_iterator i = x.begin();
          i != x.end() && result != std::numeric_limits<int>::max(); i++)
    {
      int temp = Penalty(j, j + *i - 1);
      result = temp == std::numeric_limits<int>::max() ? temp : result + temp;
      j += *i;
    }
    return result;
  }

  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    int min = std::numeric_limits<int>::max();
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (min > temp)
      {
        min = temp;
        result = i;
      }
    }
    return *result;
  }

  virtual std::list<int> SolutionCombinatorial() const
  {
    std::list<int> result;
    Combinations c(*this);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

public:
  class Combinations {
  public:
    const Train& t;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int countBlocks;
    int countCars;
  public:
    Combinations(const Train& t):
      t(t)
    {
    }

    virtual ~Combinations()
    {
```

```
  }

  virtual void Compute()
  {
   temp.clear();
   combinations.clear();
   countBlocks = 0;
   countCars = 0;
   Visit(0, false);
  }

  void operator = (const Combinations&){}

 private:
  virtual void Visit(int x, bool next)
  {
   if (next)
   {
    temp.push_back(countBlocks);
    countCars++;
    countBlocks = 1;
   }
   else
    countBlocks++;
   if (x == t.size - 1)
   {
    temp.push_back(countBlocks);
    countCars++;
    if (countCars == t.cars)
     combinations.push_back(temp);
    countCars--;
    temp.pop_back();
   }
   else
   {
    int sum =
     std::accumulate(t.blocks.begin() + x + 1 - countBlocks, t.blocks.begin() + x + 1, 0);
    Visit(x+1, true);
    if (sum + t.blocks[x+1] <= t.capacity)
     Visit(x+1, false);
   }
   if (next)
   {
    countBlocks = temp.back();
    countCars--;
    temp.pop_back();
   }
   else
    countBlocks--;
  }
 };
};
```

This is very similar to the previous class Lorries. There are some slight modifications in function Visit, because now we are only interested in combinations with a certain length. Function Read now also inputs the new data member. The interesting differences lie in function Optimal, now with two variables, and in the corresponding function Solution:

```
class Train {
//...
 // Optimal cost of carrying x blocks using y cars. First block is block 0.
 virtual int Optimal(int x, int y) const
 {
  int result;
```

```
    if (y == 0)
      result = x == 0 ? 0 : std::numeric_limits<int>::max();
    else
    {
      result = std::numeric_limits<int>::max();
      for (int i = 1; i <= x && Penalty(x-i, x-1) != std::numeric_limits<int>::max(); i++)
        if (Optimal(x-i, y-1) != std::numeric_limits<int>::max())
          result = mas::Min(result, Optimal(x-i, y-1) + Penalty(x-i, x-1));
    }
    return result;
  }

  virtual std::list<int> Solution(int x, int y) const
  // optimal solution for carrying x blocks using y cars.
  {
    std::list<int> result;
    while (x > 0)
    {
      int i = 1;
      while (Optimal(x-i, y-1) == std::numeric_limits<int>::max()
             || Optimal(x-i, y-1) + Penalty(x-i, x-1) > Optimal(x, y))
        i++;
      result.push_front(i);
      x -= i;
      y--;
    }
    return result;
  }
};
```

The complexity of this solution is N^2*M, where N represents the number of blocks and M the number of cars: each item of the memo matrix is computed once, and each computation involves a for loop whose control variable varies from 1 to N. For Lorries, the complexity is $N*M$, with a similar argument, noting that the memo is a vector. (Actually, this is not what happens in the programs as they stand, because function Penalty also involves a loop. However, we could have precomputed this function at a cost N^2. This actually dominates the complexity of Lorries, in general.)

16 Printing Neatly

Is your favorite word processor greedy? When you type a word that does not fit in the current line, does it simply move it to the next line, or does it also reformat the preceding lines in that paragraph so as to redistribute spaces at the end of each line as evenly as possible, in order to try to avoid that some lines have a lot of spaces at the end and others not? Consider the following simple example with a fixed size font and 20 characters per line:

```
aaa bbbb cccccc d ee
fffffff ggggg hh ii
jjj kkkkk l
mmmmmmmmmmmm nnnnnn
ooooo ppp
```

The ‘m’ word is quite long and had to be placed on the forth line, leaving a lot of spaces on the third. Were the word processor not greedy, it would reformat the first three lines in the following way, which looks much better:

```
aaa bbbb cccccc d
ee fffffff ggggg
hh ii jjj kkkkk l
mmmmmmmmmmmm nnnnnn
ooooo ppp
```

What's the criterion for deciding that one paragraph looks better that the other? We add the cubes of the number of spaces at the end of each line in the paragraph, except the last. The lower the sum, the neater the paragraph is. In the first example, the neatness is $0^3+1^3+9^3+2^3=738$. In the second example, it is $3^3+4^3+3^3+2^3=126$. Got it?

The "printing neatly" problem is the following: given a list of words in a paragraph, print them as neatly as possible, in a printer whose line capacity is known.

This a standard dynamic programming problem from text books. It is presented as an exercise in *Introduction to Algorithms* [Cormen *et al.*, page 364].

Before tackling the problem with words, let's solve it with numbers. Each number represents the length of a word and we want to find out how many words go to each line.

But this is almost exactly like the lorries problem! Instead of lorries, we have lines; instead of blocks, we have words.

The difference lies only in the Penalty function: we now use cubes instead of squares, and, subtly, the last line does not count. In other words, the penalty for the last line is always zero.

So, let's copy everything from the class Lorries to the new class PrintingNeatly, changing some identifiers, and omitting function Penalty, for the moment:

```
class PrintingNeatly {
public:
  std::vector<int> words;  // word lengths
  int capacity;                 // line capacity
  int size;
public:
  virtual ~PrintingNeatly()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> capacity >> size;
    if (!input)
      return;
    words.clear();
    words.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
      input >> x;
      words.push_back(x);
    }
  }

  virtual int Penalty(int x, int y) const
  {
    //...
  }

  virtual int Optimal(int x) const
  {
    int result = 0;
    if (x > 0)
    {
      result = std::numeric_limits<int>::max();
      for (int i = 1; Penalty(x-i, x-1) < std::numeric_limits<int>::max() && i <= x; i++)
        result = mas::Min(result, Optimal(x-i) + Penalty(x-i, x-1));
    }
    return result;
  }
```

```
  virtual std::list<int> Solution(int x) const
  {
    std::list<int> result;
    while (x > 0)
    {
      int i = 1;
      while (Optimal(x-i) + Penalty(x-i, x-1) > Optimal(x))
        i++;
      result.push_front(i);
      x -= i;
    }
    return result;
  }

  virtual int Cost(const std::list<int>& x) const
  {
    int result = 0;
    int j = 0;
    for (std::list<int>::const_iterator i = x.begin();
          i != x.end() && result != std::numeric_limits<int>::max(); i++)
    {
      int temp = Penalty(j, j + *i - 1);
      result = temp == std::numeric_limits<int>::max() ? temp : result + temp;
      j += *i;
    }
    return result;
  }
};
```

As a matter of fact, functions Cost, Solution and Optimal do not address the data members directly, and therefore they are exactly like those from class Lorries.

Let us see now function Penalty. Recall that Penalty(x, y) is the cost of having words x, x+1, …, y on the same line. This is just like having blocks x, x+1, …, y on the same lorry, except for the last line. There are three cases now: if the line is overflowing (too many words on the line) the cost is infinite; else if the line is not the last line the penalty is the cube of the number of spaces on the right; else this is a non overflowing last line, and the penalty is zero. The line is the last line if y is the last word:

```
class PrintingNeatly {
//
  virtual int Penalty(int x, int y) const
  {
    int result;
    int sum = std::accumulate(words.begin() + x, words.begin() + y + 1, 0) + y - x;
    if (sum > capacity)
      result = std::numeric_limits<int>::max();
    else if (y < size - 1)
      result = mas::Cube(capacity - sum);
    else
      result = 0;
    return result;
  }
};
```

Note that the value of sum includes one space character between each pair of adjacent words in a line.

How can we use this to really print some paragraphs?

We develop a new class PrintingNeatlyText, derived from PrintingNeatly, or, better, from the memoized version of PrintingNeatly, for the actual words. After reading the text into a list of

words we initialize the vector words in the base class. We then write on the output file as many words in each line as dictated by the solution list:

```
class PrintingNeatlyText: public MemoVector<PrintingNeatly> {
public:
  std::list<std::string> text;
public:
  PrintingNeatlyText(int capacity)
  {
    this->capacity = capacity;
  }

  virtual ~PrintingNeatlyText()
  {
  }

  virtual void Read(std::istream& input)
  {
    text.clear();
    std::string s;
    while (input >> s)
      text.push_back(s);
    size = static_cast<int>(text.size());
    words.clear();
    words.reserve(size);
    for (std::list<std::string>::const_iterator i = text.begin(); i != text.end(); i++)
      words.push_back(static_cast<int>(i->size()));
  }

  virtual void Write(std::ostream& output) const
  {
    Write(Solution(size), output);
  }

  virtual void Write(const std::list<int>& solution, std::ostream& output) const
  {
    std::list<std::string>::const_iterator j = text.begin();
    for (std::list<int>::const_iterator i = solution.begin(); i != solution.end(); i++)
    {
      for (int k = 0; k < *i ; k++, j++)
        output << (k == 0 ? "" : " ") << *j;
      output << std::endl;
    }
  }
};
```

Just for fun, let's add a function for writing greedily. First, we need a function SolutionGreedy in the base class:

```
class PrintingNeatly {
// …
  virtual std::list<int> SolutionGreedy() const
  {
    std::list<int> result;
    if (words.empty())
      return result;
    std::vector<int>::const_iterator i = words.begin();
    int count = 1; // words in line
    int chars = static_cast<int>(*i); // chars in each line
    for (i++; i != words.end(); i++)
    {
      if (chars + 1 + static_cast<int>(*i) > capacity)
      {
        result.push_back(count);
```

```
      count = 0;
      chars = 0;
     }
     else
      chars++;
     count++;
     chars += static_cast<int>(*i);
    }
    result.push_back(count);
    return result;
  }
};
```

For writing greedily, it suffices to use this function instead of the other:

```
class PrintingNeatlyText: public MemoVector<PrintingNeatly> {
//…
  virtual void WriteGreedily(std::ostream& output) const
  {
    Write(SolutionGreedy(), output);
  }
};
```

Here's a test function that reads a paragraph from a file, prints it neatly and then prints is greedily in lines of capacity 20:

```
void TestPrintingNeatlyText(std::istream& input, std::ostream& output)
{
  PrintingNeatlyText t(20);
  t.Read(input);
  t.Write(output);
  output << "-------" << std::endl;
  t.WriteGreedily(output);
  output << "-------" << std::endl;
}
```

When the input file is as follows:

```
aaa bbbb cccccc d ee fffffff ggggg hh ii jjj
kkkkk l mmmmmmmmmmm nnnnnn ooooo ppp
```

we get exactly the texts we used as example in the beginning:

```
aaa bbbb cccccc d
ee fffffff ggggg
hh ii jjj kkkkk l
mmmmmmmmmmm nnnnnn
ooooo ppp
-------
aaa bbbb cccccc d ee
fffffff ggggg hh ii
jjj kkkkk l
mmmmmmmmmmm nnnnnn
ooooo ppp
-------
```

17 Trail

Let us see another problem that is solved using a technique in some ways similar to that of the three previous cases, Lorries, Train and Printing Neatly.

We are going on a walking trip on the mountains. We start at the beginning of a beautiful mountain trail and we want to reach the end within a given number of days. Along the trail there are campsites where we can spend the night. We know the distances between campsites. The trail has no bifurcations. How should we organize our trip so that the maximum amount of walking

in any day is minimum? (This problem was set for the Portuguese Programming Marathon 2003, http://acm.up.pt/miup/, by my colleague Fernando Silva from the University of Porto.)

For example, suppose there are 4 campsites and the distances are 7, 2, 6, 4 and 5. The first number is the distance from the starting point to the first campsite and the last is the distance from the last campsite to the end of the trail. If we want to make the trip in three days, we should walk 9 units in the first day, 6 in the second day and 9 in the third day. Or 7 in the first day, 8 in the second and 9 in the third. In both these cases the maximum distance in a day is 9. Is 9 the minimum value we can get?

To simplify the discussion, we establish a fictitious campsite at the starting point. That's campsite 0. And we establish another fictitious campsite at the end too. A *section* is a portion of the trail between two adjacent campsites.

Here's our class:

```
class Trail {
public:
  int campsites;
  int days;
  std::vector<int> sections;
private:
  int size;
  std::vector<int> distance; // distance[i] = distance from campsite[0] to campsite i

public:
  virtual ~Trail()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> campsites >> days;
    if (!input)
      return;
    size = campsites + 1;
    sections.clear();
    sections.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
      input >> x;
      sections.push_back(x);
    }
    distance.clear();
    distance.reserve(size+2);
    distance.push_back(0);
    for (int i = 0; i < size; i++)
      distance.push_back(sections[i] + distance.back());
  }

  virtual int Distance(int x, int y) const // distance from campsite x to campsite y, x <= y.
  {
    return distance[y] - distance[x];
  }

  virtual int Optimal(int x, int y) const // optimal cost arriving at campsite x in y days, x >= y
  {
    //...
  }

  virtual std::list<int> Solution() const
  {
```

```
  //...
 }
};
```

Note that we compute the distances from the starting point, by accumulating the sections. This will make our life easier later. We introduce a private data member size, for the number of sections, since what the input file gives us is the number of campsites.

Using the combinatorial approach, we would try all combinations of campsites with days+1 elements and such that the first and the last campsites are always present. Here is a function to compute the cost of such a combination:

```
class Trail {
//...
 virtual int Cost(const std::list<int>& x) const
 {
  int result = 0;
  std::list<int>::const_iterator j = x.begin();
  j++;
  for (std::list<int>::const_iterator i = x.begin(); j != x.end(); i++, j++)
   result = mas::Max(result, Distance(*i, *j));
  return result;
 }
};
```

And now the remaining setup for the combinatorial solution:

```
class Trail {
//...
 virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
 {
  std::list<std::list<int> >::const_iterator result;
  int min = std::numeric_limits<int>::max();
  for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
  {
   int temp = Cost(*i);
   if (min > temp)
   {
    min = temp;
    result = i;
   }
  }
  return *result;
 }

 virtual std::list<int> SolutionCombinatorial() const
 {
  std::list<int> result;
  Combinations c(*this);
  c.Compute();
  if (!c.combinations.empty())
   result = MostValuable(c.combinations);
  return result;
 }

public:
 class Combinations {
 public:
  const Trail& t;
  std::list<std::list<int> > combinations;
  int days;
 private:
  std::list<int> temp;
  int size;
 public:
```

```
    Combinations(const Trail& t):
      t(t)
    {
    }

    virtual ~Combinations()
    {
    }

    virtual void Compute()
    {
      temp.clear();
      size = 0;
      combinations.clear();
      Visit(0);
    }
  private:
    virtual void Visit(int x)
    {
      temp.push_back(x);
      size++;
      if (size == t.days)
      {
        temp.push_back(t.campsites + 1);
        combinations.push_back(temp);
        temp.pop_back();
      }
      else
        for (int i = x+1; i <= t.campsites; i++)
          Visit(i);
      size--;
      temp.pop_back();
    }
    void operator = (const Combinations&){}
  };
};
```

Here is a test function for the combinatorial solution:

```
void TestTrailCombinatorial(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    Trail t;
    t.Read(input);
    if (!input)
      break;
    std::list<int> solution = t.SolutionCombinatorial();
    output << solution << "/" << t.Cost(solution) << std::endl;
    Trail::Combinations c(t);
    c.Compute();
    mas::WriteLine(c.combinations, "\n", output);
    output << std::endl;
  }
}
```

Given the following input file:

```
4 3
7 2 6 4 5
```

that corresponds to the example given in the presentation of this problem, we get the following output:

```
0 1 3 5/9
0 1 2 5
```

```
0 1 3 5
0 1 4 5
0 2 3 5
0 2 4 5
0 3 4 5
```

The solution tells to sleep in the first and third campsites. By doing that, we are guaranteed never to walk more than 9 in one day.

Let us now tackle the problem directly, by asking ourselves what is the optimal cost for arriving at campsite x after y days of walking, Optimal(x, y). If y == 1, it's the total distance between campsite zero and campsite x. Otherwise, on the last day (i.e., on day y), we walk either one section or two sections or etc. Suppose we know how to arrive optimally at campsite x-i in y-1 days, for all reasonable values of i., i.e., for i varying from to 1 to x-y+1. Then consider the expressions mas::Max(Optimal(x-i, y-1), Distance(x-i, x))), with i varying. The minimum value of all these expressions is Optimal(x, y). That leads us to the following definition:

```
class Trail {
//…
  virtual int Optimal(int x, int y) const // optimal cost arriving at campsite x in y days, x >= y
  {
    int result = Distance(0, x);
    if (y > 1)
      for (int i = 1; i <= x-y+1 && Distance(x-i, x) < result; i++)
        result = mas::Min(result, mas::Max(Optimal(x-i, y-1), Distance(x-i, x)));
    return result;
  }
};
```

Once we have this function, can we compute the solution greedily? If we do, we may end up with a shorter trip than planned, i.e., we may reach the end of the trail in less than the allocated number of days. For example: suppose the sections are 1, 1, 2 and we want to spend three days in the mountain. The only solution is to walk one section in each day, of course. However, the least maximum walk for three days is 2. Hence, a greedy solution would tell us to cover two sections in the first day, and then the trip would last two days only, not three.

If we want any solution, we may decide to travel greedily in the beginning and then make the last sections one by one, in such a way that the total number of days is as expected. Here's function Solution that does just that:

```
class Trail {
  //…
  virtual std::list<int> Solution(int x, int y) const
  {
    std::list<int> result;
    int cost = Optimal(x, y);
    int remaining = days;
    int i = 0;
    result.push_back(0);
    while (i < size)
    {
      int sum = 0;
      do
        sum += sections[i++];
      while (i <= size - remaining && sum + sections[i] <= cost);
      result.push_back(i);
      remaining--;
    }
    return result;
  }
};
```

The following test function uses a memoized class:

```
void TestTrail(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    MemoMatrix<Trail> t;
    t.Read(input);
    if (!input)
      break;
    output << t.campsites << " " << t.days << std::endl;
    output << t.sections << std::endl;
    output << t.Optimal(t.campsites+1, t.days) << std::endl;
    std::list<int> solution = t.Solution(t.campsites + 1, t.days);
    output << solution << "/" << static_cast<int>(solution.size())
           << "/" << t.Cost(solution) << std::endl;
    output << std::endl;
  }
}
```

The “half-greedy” solution may turn out to be rather unbalanced, with long walks in the first days and short walks towards the end. (That might not be unreasonable, since perhaps we will be more and more tired as the trip unfolds...) If we decide to balance the journey, and still spend that number of days in the mountain, we can use the scheme of problem Train, pretending that sections are blocks and that each day is a wagon whose capacity is the least maximum, previously computed.

18 Bus stops

Meet Linear City. In Linear City there is only one street, appropriately named Unique street. Everybody lives in one of the apartment buildings on Unique street. All buildings are equal except for the color and the height, and they are positioned side by side along the street. There is only one bus line, going from one end of the street to the other. The bus company wants to relocate the bus stops, in order to minimize the distance passengers have to walk from the door of their apartment building to the nearest bus stop. The population of each building is known. It would be anti-economical to have a bus stop in front of each building, although this solution would absolutely minimize the distance. Instead, the number of bus stops to deploy is given.

This problem has some similarities to the previous four, which we will exploit. But, first, let us try the combinatorial approach, to become more acquainted with the problem. Actually, if we have N buildings, numbered from 0 to N-1 and K bus stops, what we need is to compute all the combinations of the integers from 0 to N-1 taken K at a time, and pick the one that gives the minimum cost. Given this observation, we can start to program, following our usual pattern:

```
class BusStops {
public:
  int stops;  // number of bus stops
  int size;
  std::vector<int> buildings; // number of persons living in building.
public:
  virtual ~BusStops()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> stops >> size;
    if (!input)
      return;
    buildings.clear();
    buildings.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
```

```
      input >> x;
      buildings.push_back(x);
    }
  }

  virtual int Cost(const std::list<int>& x) const
  {
    //…
  }

  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    int min = std::numeric_limits<int>::max();
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (min > temp)
      {
        min = temp;
        result = i;
      }
    }
    return *result;
  }

  virtual std::list<int> SolutionCombinatorial() const
  {
    std::list<int> result;
    Combinations c(*this);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

public:
  class Combinations {
  public:
    const BusStops& b;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int count;
  public:
    Combinations(const BusStops& b):
      b(b),
      count(b.stops)
    {
    }

    virtual ~Combinations()
    {
    }

    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      for (int i = 0; i < b.size; i++)
        Visit(i);
    }

  private:
```

```
    virtual void Visit(int x)
    {
      temp.push_back(x);
      if (static_cast<int>(temp.size()) == count)
        combinations.push_back(temp);
      else
        for (int i = x+1; i < b.size; i++)
          Visit(i);
      temp.pop_back();
    }
    void operator = (const Combinations&){}
  };
};
```

In order to define the cost function, we use function Penalty(int x, int y, int z), which computes the cost of having people from buildings x, x+1, …, y, walk to the bus stop placed at z:

We set up our class with functions Read, Cost and Penalty only, for the moment:

```
class BusStops {
//…
  virtual int Penalty(int x, int y, int z) const
  {
    int result = 0;
    for (int i = x; i <= y; i++)
      result += buildings[i] * ::abs(z-i);
    return result;
  }

  virtual int Cost(const std::list<int>& x) const
  {
    int result = 0;
    int a = 0;
    std::list<int>::const_iterator j = x.begin();
    j++;
    for (std::list<int>::const_iterator i = x.begin(); j != x.end(); i++, j++)
    {
      int b = (*i + *j) / 2;
      result += Penalty(a, b, *i);
      a = b+1;
    }
    result += Penalty(a, size - 1, x.back());
    return result;
  }
};
```

Let us experiment this with a small case, with six buildings and three bus stops. Here's the test function:

```
void TestBusStopsCombinatorial(std::istream& input, std::ostream& output)
{
  for (;;)
  {
    BusStops bs;
    bs.Read(input);
    if (!input)
      break;
    BusStops::Combinations c(bs);
    c.Compute();
    mas::WriteLine(c.combinations, "\n", output);
    const std::list<int>& solution = bs.MostValuable(c.combinations);
    output << solution << "/" << bs.Cost(solution) << std::endl;
  }
}
```

This is the input file:

```
3
6
6 1 7 6 2 5
```

We get the following output, showing all the combinations of the integers 0 to 5 taken 3 at a time, and the most valuable one and its cost at the end:

```
0 1 2
0 1 3
0 1 4
0 1 5
0 2 3
0 2 4
0 2 5
0 3 4
0 3 5
0 4 5
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
0 2 5/9
```

Let us now look for the Optimal function. Can we use a function with one argument only, Optimal(int x), which gives the optimal cost of using x bus stops? Or do we need a function with two arguments, Optimal(int x, int y), which computes the optimal cost of using x bus stops, the x-th one being placed at y? Let us try the latter.

Suppose we have already computed Optimal(x-1, i) for all i. In other words, we know the optimal cost of placing x-1 bus stops, the last one being at any of the buildings. Now we want to place the x-th bus stop at y. What will the optimal cost be? The x-th bus stop will be located to the right of the x-1-th, and thus it will steal passengers from that bus stop, but not from any of the others (who are positioned to the left of the x-1-th). If we place the x-th bus stop at y, the x-1-th could be at y-1, y-2, …, x, x-1, x-2, going from right to left (the first building is building zero). Let i be one of those places. If the x-1-th stop is located at i, people living in buildings from (i+y)/2 + 1 to the last building will use the new stop. For those people the cost of walking to the nearest bus stop decreases by an amount we can compute using function Penalty. For the others it remains the same. Hence, the optimal cost of having the x-th stop at y and the x-1-th at i is Optimal(x-1, i) minus the penalty decrease for the inhabitants of buildings from (i+y)/2+1 to the last. Therefore, in order to evaluate Optimal(x, y), we compute the minimum of the optimal cost of having the x-th stop at y and the x-1 stop at i, for i varying from x-2 to y-1. The following function captures this idea:

```
class BusStops {
//
  virtual int Optimal(int x, int y) const // optimal cost using x stops, the last one at y.
  {
    int result = Penalty(0, size - 1, y);
    if (x > 1)
      for (int i = x - 2; i < y; i++)
      {
        int p1 = Penalty((i + y) / 2 + 1, size - 1, i);
        int p2 = Penalty((i + y) / 2 + 1, size - 1, y);
        result = mas::Min(result, Optimal(x-1, i) - p1 + p2);
      }
```

```
    return result;
  }
};
```

The absolute optimal cost for a given number of bus stops is computed by the following function:

```
class BusStops {
//...
  virtual int Optimal1(int x) const
  {
    int result = std::numeric_limits<int>::max();
    for (int i = x - 1; i < size; i++)
      result = mas::Min(result, Optimal(x, i));
    return result;
  }
};
```

We still have to compute the optimal locations for the bus stops. For finding the last one we use linear search:

```
class BusStops {
//
  virtual int LastStop() const
  {
    int x = Optimal1(stops);
    int result = stops - 1;
    while (Optimal(stops, result) != x)
      result++;
    return result;
  }
};
```

For computing the complete solution, we start with the last stop and then proceed backwards:

```
class BusStops {
  //...
  virtual std::list<int> Solution() const
  {
    std::list<int> result;
    int y = LastStop();
    result.push_front(y);
    for (int j = stops - 1; j > 0; j--)
    {
      int i = j - 1;
      while (Optimal(j, i) - Penalty((i + y) / 2 + 1, size - 1, i)
                           + Penalty((i + y) / 2 + 1, size - 1, y)
             != Optimal(j+1, y))
        i++;
      y = i;
      result.push_front(y);
    }
    return result;
  }
};
```

We finalize this exercise with a test function that relies on the memoized version of the class. It handles a sequence of test cases:

```
void TestBusStops(std::istream& input, std::ostream& output)
{
  for(;;)
  {
    MemoMatrix<BusStops> b;
    b.Read(input);
    if (!input)
```

```
    break;
   std::list<int> solution = b.Solution();
   output << solution << "/" << b.Cost(solution) << std::endl;
  }
}
```

19 Scheduling

We run a very successful small shop but, unfortunately, we have one machine only. There are lots of orders for jobs on that machine, more that we can accommodate. Each candidate job has a given processing time on the machine and must be finished before a given deadline. If we are able to meet the deadline, we will get paid a given amount for that job. If not, we are paid nothing, which means that we might as well forget about that job. Which subset of jobs should we run through the machine so that our profit is maximized?

This problem is an exercise from the book *Introduction to Algorithms* [Cormen *et al.*, page 369]. It was used in a slightly different form at 2003 Southwestern European Regional ACM Programming Contest, http://www.polytechnique.edu/icpc2003/.

Here is an example input file:

```
10
4 4 7
3 6 12
4 5 20
3 3 15
3 2 11
4 5 12
7 2 11
6 2 13
3 2 17
3 3 19
```

The first number is the number of jobs. Each following triplet represents a job: duration, profit and deadline.

A combinatorial strategy is in order: sort by deadline, generate all possible schedules, and pick the most profitable one. Here we go:

```
class Job {
public:
  int duration;
  int profit;
  int deadline;

public:
  int Start() const
  {
    return deadline - duration;
  }

  bool operator == (const Job& other) const
  {
    return deadline == other.deadline
      && duration == other.duration
      && profit == other.profit;
  }

  bool operator <= (const Job& other) const
  {
    return deadline < other.deadline
      || deadline == other.deadline
       && (duration < other.duration
         || duration == other.duration && profit <= other.profit);
```

```
  }

  bool operator < (const Job& other) const
  {
    return operator <= (other) && !operator == (other);
  }

  friend std::ostream& operator << (std::ostream& output, const Job& x)
  {
    output << "<" << x.duration << " " << x.profit << " " <<  x.deadline << ">";
    return output;
  }
};
```

Those Boolean functions in class Job will be used by the std::sort function, with which we will sort the jobs by deadline.

In class Scheduling, the data members are a vector of jobs and its size. We provide functions Read, Cost, MostValuable and SolutionCombinatorial and inner class Combinations:

```
class Scheduling {
public:
  std::vector<Job> jobs;
  int size;
public:
  virtual ~Scheduling()
  {
  }

  virtual void Read(std::istream& input = std::cin)
  {
    input >> size;
    if (!input)
      return;
    jobs.clear();
    jobs.reserve(size);
    for (int i = 0; i < size; i++)
    {
      Job x;
      input >> x.duration >> x.profit >> x.deadline;
      jobs.push_back(x);
    }
  }

  int Cost(const std::list<int>& s) const
  {
    int result = 0;
    for (std::list<int>::const_iterator i = s.begin(); i != s.end(); i++)
      result += jobs[*i].profit;
    return result ;
  }

  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    int max = 0;
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
    {
      int temp = Cost(*i);
      if (max < temp)
      {
        max = temp;
        result = i;
      }
```

```
    }
    return *result;
  }

  virtual std::list<int> SolutionCombinatorial() const
  {
    std::list<int> result;
    Combinations c(*this);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

public:
  class Combinations {
  public:
    const Scheduling& s;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int value;
    int size;
    int capacity;
  public:
    Combinations(const Scheduling& s):
      s(s)
    {
    }

    virtual ~Combinations()
    {
    }

    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      for (int i = 0; i < s.size; i++)
        Visit(i);
    }

  private:
    virtual void Visit(int x)
    {
      temp.push_back(x);
      combinations.push_back(temp);
      for (int i = x+1; i < s.size; i++)
        if (s.jobs[x].deadline <= s.jobs[i].Start())
          Visit(i);
      temp.pop_back();
    }
  private:
    void operator = (const Combinations&){}
  };
};
```

Function Visit expresses the idea that each partial combination is extended with each of the jobs that start after the last job in the combination finishes.

We can use this idea to program function Optimal. Let Optimal(x) be the optimal profit we can obtain when the last job that goes through the machine is the x-th job. Suppose we know Optimal(0), Optimal(1), …, Optimal(x-1). The value of Optimal(0) is jobs[0].profit, of course.

Consider the expression Optimal(i) + jobs[x].profit for all i such that job i ends before (or equal) job x starts. The maximum of all those expressions gives us the maximum profit we can get when x is the last job. If no job ends before job x starts, then only job x will run through the machine, for a profit of jobs[x].profit. This explains the following function:

```
class Scheduling {
//...
  virtual int Optimal(int x) const
  {
    int result = jobs[x].profit;
    for (int i = 0; i < x; i++)
      if (jobs[i].deadline <= jobs[x].Start())
        result = mas::Max(result, Optimal(i) + jobs[x].profit);
    return result;
  }
};
```

In order to get the absolute optimal profit, we take the maximum of Optimal(i) for i varying from 0 to size-1:

```
class Scheduling {
//...
  virtual int Optimal0() const
  {
    int result = 0;
    for (int i = 0; i < size; i++)
      result = mas::Max(result, Optimal(i));
    return result;
  }
};
```

Next we have to compute the solution. Once again, we accomplish that by undoing the computation of Optimal and Optimal0:

```
class Scheduling {
//...
  virtual std::list<int> Solution() const
  {
    std::list<int> result;
    int profit = Optimal0();
    int i = size-1;
    while (profit != Optimal(i))
      i--;
    result.push_back(i);
    profit -= jobs[i].profit;
    while (profit > 0)
    {
      while (!(profit == Optimal(i) && jobs[i].deadline <= jobs[result.front()].Start()))
        i--;
      result.push_front(i);
      profit -= jobs[i].profit;
    }
    return result;
  }
};
```

The first while statement searches the last job in the solution. The second searches the remaining jobs, always going from the later jobs to the earlier ones. Note that this solution thus computed is not guaranteed to be the same as the combinatorial one. If several solutions are possible, the combinatorial solution is the lexicographically first (on the indices) and this is the lexicographically last.

Here is a test function that creates a random set of jobs and computes the optimal profit and the two solutions, standard and combinatorial.

```
void TestRandomized(std::ostream& output)
{
  MemoVector<Scheduling> s;
  s.Randomize(8, 16, 4, 50);
  std::sort(s.jobs.begin(), s.jobs.end());
  mas::WriteLine(s.jobs, " ", output);
  output << s.Optimal0() << std::endl;
  std::list<int> solution = s.Solution();
  output << solution << "/" << s.Cost(solution) << std::endl;
  mas::WriteIndirectLine(s.jobs, solution, " ", output);
  std::list<int> solution1 = s.SolutionCombinatorial();
  output << solution1 << "/" << s.Cost(solution1) << std::endl;
  mas::WriteIndirectLine(s.jobs, solution1, " ", output);
}
```

Function Randomize has four arguments: number of jobs, maximum duration, maximum profit and maximum deadline. Observe:

```
class Scheduling {
//...
  virtual void Randomize(int size, int maxDuration, int maxProfit, int maxDeadline)
  {
    this->size = size;
    jobs.clear();
    jobs.reserve(size);
    for (int i = 0; i < size; i++)
    {
      Job x;
      x.duration = ::rand() % maxDuration + 1;
      x.profit = ::rand() % maxProfit + 1;
      x.deadline = ::rand() % maxDeadline + x.duration;
      jobs.push_back(x);
    }
  }
};
```

To simplify the examples, we add the duration to the random deadline, to make sure that all jobs start after time zero. (Thus, the maximum deadline mentioned above can be greater than maxDeadline.)

Here is the output of a randomly created example that illustrates the case where the two solutions (standard and combinatorial) are different:

```
<2 4 7> <8 2 22> <11 1 36> <6 1 37> <1 2 42> <5 3 45> <13 3 59> <14 1 59>
13
0 1 3 5 6/13
<2 4 7> <8 2 22> <6 1 37> <5 3 45> <13 3 59>
0 1 2 5 6/13
<2 4 7> <8 2 22> <11 1 36> <5 3 45> <13 3 59>
```

20 Pyramids

This is a solitaire game, played with rectangular pieces of various lengths and widths, all 1 cm thick. The goal of the game is to build a pyramid as tall as possible, by stacking smaller pieces on top of larger ones. Pieces are numbered from zero. A piece can be stacked on top of another piece provided it is strictly smaller, i.e., its shorter side is strictly shorter than the shorter side of the other piece and the same for the longer side. By piling up pieces in this fashion we build something that looks like a pyramid. Which pieces should we choose in order to assemble the tallest pyramid?

The brute force approach would be trying out all permutations of all subsets of the pieces selecting those who make up pyramids, and from these, the tallest. But that would be really brute.

Instead, we may think in terms of graphs. Each piece is a vertex. If a piece *p1* is smaller than a piece *p2*, then there is an edge from *p1* to *p2*. The graph we obtain this way is directed (if there is an edge from *p1* to *p2*, then there is not an edge from *p2* to *p1*) and acyclic (no path in the graph visits the same node twice), of course. We want the longest path in the graph.

Had we developed before a class library for graphs, we would have encountered this problem before, and some function or some class in the library would readily solve it. Let's see how this could have been done. We start with a simple class for graphs, implemented as a vector of adjacency lists:

```
class Graph {
public:
  std::vector<std::list<int> > successors;
public:
  Graph::Graph(int size = 0):
    successors(size)
  {
  }

  virtual ~Graph()
  {
  }

  virtual int Size() const
  {
    return static_cast<int>(successors.size());
  }

  virtual int CountEdges() const
  {
    int result = 0;
    for (std::vector<std::list<int> >::const_iterator i = successors.begin();
          i != successors.end(); i++)
      result += static_cast<int>(i->size());
    return result;
  }

  virtual void Resize(int n)
  {
    successors.resize(n);
  }

  virtual void Clear()
  {
    successors.clear();
  }

  virtual void SwapOut(Graph& other)
  {
    successors.swap(other.successors);
  }

  virtual void Link(int x, int y)
  {
    if (!IsLinked(x, y))
      successors[x].push_back(y);
  }

  virtual bool IsLinked(int x, int y) const
  {
    return std::find(successors[x].begin(), successors[x].end(), y) != successors[x].end();
  }
```

```
  virtual void Transpose()
  {
    Graph temp(static_cast<int>(successors.size()));
    for (unsigned i = 0; i < successors.size(); i++)
      for (std::list<int>::iterator j = successors[i].begin(); j != successors[i].end(); j++)
        temp.Link(*j, i);
    SwapOut(temp);
  }
};
```

In such a general graph, the longest path from a vertex *x* to a vertex *y* must be computed using brute force. Here is a class that, given a graph (at construction time) computes all paths from a vertex to another vertex. The class provides two functions, Longest and Shortest, that return the longest path and the shortest path (or one of them, in each case, if there are more than one). This class uses the familiar Visit pattern, that we have met in many examples before:

```
class Paths {
private:
  const Graph& graph;
  std::list<int> temp;
  std::vector<bool> visited;
public:
  std::list<std::list<int> > paths;
  int source;
  int destination;
public:
  virtual ~Paths()
  {
  }

  Paths(const Graph& graph):
    graph(graph)
  {
  }

  virtual void Compute(int x, int y)
  {
    source = x;
    destination = y;
    paths.clear();
    visited.clear();
    visited.resize(graph.successors.size(), false);
    Visit(x);
  }

  virtual const std::list<int>& Longest() const
  {
    std::list<std::list<int> >::const_iterator result;
    std::list<int>::size_type best = 0;
    for (std::list<std::list<int> >::const_iterator i = paths.begin(); i != paths.end(); i++)
    {
      std::list<int>::size_type temp = i->size();
      if (best < temp)
      {
        best = temp;
        result = i;
      }
    }
    return *result;
  }

  virtual const std::list<int>& Shortest() const
  {
    std::list<std::list<int> >::const_iterator result;
```

```
    std::list<int>::size_type best = std::numeric_limits<int>::max();
    for (std::list<std::list<int> >::const_iterator i = paths.begin(); i != paths.end(); i++)
    {
      std::list<int>::size_type temp = i->size();
      if (best > temp)
      {
        best = temp;
        result = i;
      }
    }
    return *result;
  }

private:
  virtual void Visit(int x)
  {
    temp.push_back(x);
    visited[x] = true;
    if (x == destination)
      paths.push_back(temp);
    else
    {
      const std::list<int>& z = graph.successors[x];
      for (std::list<int>::const_iterator i = z.begin(); i != z.end(); i++)
        if (!visited[*i])
          Visit(*i);
    }
    visited[x] = false;
    temp.pop_back();
  }

private:
  void operator = (const Paths&){};
};
```

Remember that there are more efficient algorithms for computing the shortest path in a graph, for example using breadth-first search, but that is not the topic of this report.

In directed acyclic graphs, we could simplify the algorithm a little, since we would not have to check whether the next vertex is in the path, because it cannot be. That would not be a big improvement, however. Fortunately, for directed acyclic graphs there is a much better solution, using dynamic programming: we first compute the optimal cost of going from x to y, and then reconstruct the path as usual. Actually, this is similar to the problems Indiana Jones and Mars Explorer, only simpler. We gather these operations in class LongestPathDirectedAcyclicGraph:

```
class LongestPathDirectedAcyclicGraph {
private:
  const Graph* graph;
  Graph transposed;
public:
  virtual void SetGraph(const Graph& graph)
  {
    this->graph = &graph;
    transposed = graph;
    transposed.Transpose();
  }

  virtual int Optimal(int x, int y) const
  {
    int result;
    if (x == y)
      result = 1;
    else
    {
```

```
    result = 0;
    const std::list<int>& s = transposed.successors[y];
    for (std::list<int>::const_iterator i = s.begin(); i != s.end(); i++)
      result = mas::Max(result, Optimal(x, *i) + 1);
  }
  return result;
 }

 virtual std::list<int> Solution(int x, int y) const
 {
  std::list<int> result;
  result.push_front(y);
  while (x != y)
  {
    const std::list<int>& s = transposed.successors[y];
    std::list<int>::const_iterator i = s.begin();
    while (Optimal(x, *i) + 1 < Optimal(x, y))
      i++;
    result.push_front(*i);
    y = *i;
  }
  return result;
 }

private:
 void operator = (const LongestPathDirectedAcyclicGraph&){};
};
```

Notice that, when computing the solution, we use the transposed graph in order to traverse the original graph from the end vertex to the start vertex.

In our game, each piece is represented by an object of class Rectangle. This is a very simple class:

```
class Rectangle {
public:
 int length;
 int width;
 // invariant length >= width

 Rectangle(int x, int y):
  length(x),
  width(y)
 {
  if (length < width)
    std::swap(length, width);
 }

 virtual bool SmallerThan(const Rectangle& other) const
 {
  return length < other.length && width < other.width;
 }

 friend std::ostream& operator << (std::ostream& output, const Rectangle& r)
 {
  output << "<" << r.length << " " << r.width << ">";
  return output;
 }
};
```

The constructor swaps the elements if necessary, so that the length is not smaller that the width. This simplifies the definition of function SmallerThan, which computes what its name suggests.

Let us now see our Pyramid class. It has data members for the vector of the rectangular pieces, for the size of the vector, and for the graph that is built from the pieces. Having the graph, both

the combinatorial solution and the standard solution are computed immediately. The combinatorial solution uses class Paths, the standard solution uses class LongestPathDirectedAcyclicGraph with memoization:

```
class Pyramid {
public:
  std::vector<Rectangle> pieces;
  int size;
  Graph graph;

public:
  virtual ~Pyramid()
  {
  }

  virtual void Read(std::istream& input)
  {
    input >> size;
    if (!input)
      return;
    pieces.reserve(size+2);
    pieces.push_back(Rectangle(0, 0));
    for (int i = 0; i < size; i++)
    {
      int x;
      int y;
      input >> x >> y;
      pieces.push_back(Rectangle(x, y));
    }
    pieces.push_back(
       Rectangle(std::numeric_limits<int>::max(), std::numeric_limits<int>::max()));
    MakeGraph();
  }

  virtual void MakeGraph()
  {
    graph.Clear();
    graph.Resize(size + 2);
    for (int i = 0; i < size + 2; i++)
      for (int j = 0; j < size + 2; j++)
        if (pieces[i].SmallerThan(pieces[j]))
          graph.Link(i, j);
  }

  virtual std::list<int> Solution() const
  {
    MemoMatrix<LongestPathDirectedAcyclicGraph> lp;
    lp.SetGraph(graph);
    std::list<int> result = lp.Solution(0, size+1);
    result.pop_front();
    result.pop_back();
    return result;
  }

  virtual std::list<int> SolutionCombinatorial() const
  {
    Paths p(graph);
    p.Compute(0, size +1);
    std::list<int> result = p.Longest();
    result.pop_front();
    result.pop_back();
    return result;
  }
```

```
};
```

Observe that function Read adds two fictitious pieces, one of zero length and width and another of infinite length and width. This simplifies our work since now we only have to look for the longest path from the zero piece to the infinite piece. To get the actual longest path we remove the first and last elements of the solution.

Function Optimal can only be used if there is a path from its first argument to its second argument. Otherwise it will lead to infinite recursion. The trick of the two additional vertices ensures that we are in good shape.

Note that the dynamic programming solution if feasible for directed acyclic graphs but not for graphs in general. This is because the edges in the graph all go in one direction, so to speak, which guarantees that the recursive calls in function Optimal will not be infinite.

Here is a test function, for illustration:

```
void TestPyramid(std::istream& input, std::ostream& output)
{
  Pyramid p;
  p.Read(input);
  const std::list<int>& solution = p.Solution();
  mas::WriteIndirectLine(p.pieces, solution, " ", output);
}
```

Given the following input file:

```
12
3 6  2 8  1 3  2 7  3 6  4 4  7 2  3 8  7 7  1 1  2 2  2 9
```

the result is:

```
<1 1> <2 2> <6 3> <7 7>
```

21 Bitonic Tour

The traveling salesman problem is a classical problem that has to do with complete weighted graphs. Being complete for a graph means that each vertex is connected to all the other vertices. Being weighted means that the each edge in the graph has a certain weight. The actual problem is: compute the simple cycle that visits all vertices and has the least total weight. A cycle in a graph is a sequence of vertices $v[0]$, $v[1]$, …, $v[n]$, such that there is an edge from $v[i]$ to $v[i+1]$, and such that $v[0] = v[n]$. A cycle is simple if no two vertices in it are the same.

This problem is called "the traveling salesman problem[2]" as a metaphor for the situation where a salesman needs to visit a number of cities, with roads linking each city to all other cites (an unlikely situation, we must say…) and wants to decide in which order to make his tour, so as to spend as little fuel in his car as possible, returning in the end to the point of departure, and never visiting the same city twice (except the first one).

This problem is NP-complete. (For a proof of this, see [Cormen *et al.*]) What is the significance of it being NP-complete? The significance is that most likely there isn't a solution to the problem that runs in polynomial time. We say "most likely" because nobody has been able to prove that there isn't but nobody has been able to show that there is.

The Euclidean traveling salesman problem is analogous, but it is expressed without graphs: given a set of points is a plane, compute the shortest cycle that visits all points exactly once.

Since no efficient algorithm is known for the traveling salesman problem, we might settle for some heuristic that, while not solving the problem, at least finds a good solution within reason-

[2] More modernly, it is called "the traveling salesperson problem". It appears that the gender issue does not affect the complexity of the problem…

able computing time. One of such heuristics, suggested by J.L. Bentley, is considering only bitonic tours [Bentley]. A bitonic tour is a cycle that starts at the leftmost point, goes strictly to the right until it visits the rightmost point, and then comes back to the point of departure, now going strictly from right to left and visiting all points that were not visited in the trip from left to right.

Our problem in this section is finding the shortest bitonic tour for a given set of points. In this set of points, no two points have the same *x* coordinate.

As a warm up let us compute the shortest bitonic tour by brute force. And while we are at it, let us also compute the shortest tour of all as well, also by brute force.

The class has a data member for the vector of points, which we leave public, as usual. We must ensure that this vector is sorted by the *x* coordinate, before further processing. Thus, we must sort it when we initialize it by reading. We can also initialize it randomly, and in this case the vector comes out sorted automatically. We precompute the distances once for all and we record the results in a private matrix.

The points are represented by objects of the following class Point:

```
class Point {
public:
  double x;
  double y;
public:
  Point(double x, double y):
    x(x),
    y(y)
  {
  }

  virtual ~Point()
  {
  }

  virtual bool operator < (const Point& other) const
  {
    return x <= other.x && (x != other.x || y < other.y);
  }

  virtual double DistanceTo(const Point& other) const
  {
    return ::sqrt(Square(x-other.x) + Square(y-other.y));
  }

  void Write(std::ostream& output) const
  {
    output << x << " " << y;
  }

  friend std::ostream& operator << (std::ostream& output, const Point& p)
  {
    p.Write(output);
    return output;
  }
};
```

Here is the declaration of our problem class, with a data member for the vector of points and for the size and functions Read, Randomize and ComputeDistances:

```
class BitonicTour {
public:
  std::vector<Point> points;
  int size;
private:
```

```
  std::vector<std::vector<double> > distance;
public:
  virtual void Read(std::istream& input)
  {
    input >> size;
    if (!input)
      return;
    points.clear();
    points.reserve(size);
    for (int i = 0; i < size; i++)
    {
      int x;
      int y;
      input >> x >> y;
      points.push_back(Point(x, y));
    }
    std::sort(points.begin(), points.end());
    ComputeDistances();
  }

  virtual void ComputeDistances()
  {
    distance.clear();
    distance.resize(size);
    for (int i = 0; i < size; i++)
    {
      distance[i].reserve(points.size());
      for (int j = 0; j < size; j++)
        distance[i].push_back(points[i].DistanceTo(points[j]));
    }
  }

  virtual void Randomize(int n)
  {
    points.clear();
    size = n;
    points.reserve(size);
    for (int i = 0; i < size; i++)
      points.push_back(Point(i, ::rand()%size));
    ComputeDistances();
  }
};
```

Let us proceed with the combinatorial solutions. We now have two inner classes, Tours and Bitonic. The former generates all tours starting and ending in the first point. The latter generates all bitonic tours, also starting and ending in the first point:

```
class BitonicTour {
//...
  virtual double Cost(const std::list<int>& x) const
  {
    double result = 0.0;
    std::list<int>::const_iterator i = x.begin();
    std::list<int>::const_iterator j = i;
    for (j++; j != x.end(); i++, j++)
      result += distance[*i][*j];
    return result;
  }

  virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
  {
    std::list<std::list<int> >::const_iterator result;
    double min = std::numeric_limits<double>::infinity();
    for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
```

```
    {
      double temp = Cost(*i);
      if (min > temp)
      {
        min = temp;
        result = i;
      }
    }
    return *result;
  }

  template <class T>
  std::list<int> SolutionCombinatorial() const
  {
    std::list<int> result;
    T c(*this);
    c.Compute();
    if (!c.combinations.empty())
      result = MostValuable(c.combinations);
    return result;
  }

public:
  class Tours {
  public:
    const BitonicTour& t;
    std::list<std::list<int> > combinations;
  private:
    std::list<int> temp;
    int count;
    std::vector<bool> visited;
  public:
    Tours(const BitonicTour& t):
      t(t)
    {
    }

    virtual ~Tours()
    {
    }

    virtual void Compute()
    {
      temp.clear();
      combinations.clear();
      temp.clear();
      visited.clear();
      visited.resize(t.size, false);
      count = 0;
      Visit(0);
    }
    void operator = (const Tours&){}

  private:
    virtual void Visit(int x)
    {
      temp.push_back(x);
      visited[x] = true;
      count++;
      if (count == t.size)
      {
        temp.push_back(0);
        combinations.push_back(temp);
        temp.pop_back();
```

```
    }
    else
      for (int i = 1; i < t.size; i++)
        if (!visited[i])
          Visit(i);
    count--;
    visited[x] = false;
    temp.pop_back();
  }
};

class Bitonic {
public:
  const BitonicTour& t;
  std::list<std::list<int> > combinations;
private:
  int size;
  std::list<int> temp;
  std::vector<bool> visited;

public:
  Bitonic(const BitonicTour& t):
    t(t),
    size(static_cast<int>(t.points.size()))
  {
  }

  virtual ~Bitonic()
  {
  }

  virtual void Compute()
  {
    temp.clear();
    combinations.clear();
    temp.clear();
    visited.clear();
    visited.resize(size, false);
    Visit(0);
  }

  void operator = (const Bitonic&){}

private:
  virtual void Visit(int x)
  {
    temp.push_back(x);
    visited[x] = true;
    if (x == size - 1)
    {
      for (int i = size - 1; i >= 0; i--)
        if (!visited[i])
          temp.push_back(i);
      temp.push_back(0);

      combinations.push_back(temp);

      temp.pop_back();
      for (int i = size - 1; i >= 0; i--)
        if (!visited[i])
          temp.pop_back();
    }
    else
      for (int i = x+1; i < size; i++)
```

```
        if (!visited[i])
          Visit(i);
      visited[x] = false;
      temp.pop_back();
    }
  };
};
```

In both these inner classes, Tours and Bitonic, the running time of function Compute is exponential. Even with fast computers, they are useless in practice with more than a dozen points. So let's turn to dynamic programming for help, and ask the usual question: how can we compute the optimal cost for a bitonic tour that uses the first x points, Optimal(x), given that we have already computed the optimal cost for bitonic tours using 3, 4, …, x-1points? Yes, we need three points for a bitonic tour. Actually, 3 gives us the base case, since Optimal(3) is distance[0][1] + distance[1][2] + distance[0][2].

Let us take an example, with $x = 9$. Now, consider a bitonic tour using the first 6 points, for example. In this bitonic tour, the rightmost point is 5 (and the leftmost is zero, as always). Let us "extend" this tour into a bitonic tour of the first 9 points, by going directly from point 4 to point 8 to the right, and then from point 8 to 7, 7 to 6, 6 to 5 and then take the second half of the 6-point bitonic tour down to point zero.

Note that we extend by going directly from the last but one point in the "small" tour to the new rightmost point. Since we want a bitonic tour, all points strictly between these two points must be visited on the way back.

We can compute the penalty for extending a bitonic tour that uses the first x points to one that uses the first y points, using the technique we described. It's the length of the forward edge, plus the total length of those new edges going to the left, minus the "last" edge in the small tour, which is not present in the new tour:

```
class BitonicTour {
//…
  virtual double Penalty(int x, int y) const
  {
    double result = distance[x-2][y-1];  // "direct" distance <x-2, y-1>
    for (int i = x; i < y; i++)          // distance <x-1, x>, <x, x+1>, ... <y-2, y-1>
      result += distance[i-1][i];
    result -= distance[x-2][x-1];        // edge <x-2, x-1> is not in the tour
    return result;
  }
};
```

Suppose we have already computed Optimal(i) for i = 3, 4, …, x-1, and consider the expression Optimal(i) + Penalty(i, x). It represents the cost of the bitonic tour that uses the first x points obtained by extending the optimal tour that used the first i points with the technique described. Now take the minimum of Optimal(i) + Penalty(i, x), for i varying from 3 to x-1. Is this Optimal(x)?

By hypothesis, we are in a situation where x > 3. Suppose there is a bitonic tour s, using the first x items, such that Cost(s) < Optimal(x) and such that s can be written as <0, *a*, *k*, x-1, x-2, …, *k*+1, *b*, 0>, for some k <= x-2, and where *a* is an ascending sequence of numbers chosen in the interval [1, *k*-1] and *b* is a descending sequence of numbers chosen in the same interval, *a* and *b* are disjoint and their reunion is the interval [1..*k*-1]. Now consider the tour t = <0, *a*, *k*, *b*, 0>. This is a bitonic tour with *k*+1 points, and its cost is Cost(t) = Cost(s) - Penalty(k+1, x). But then Cost(t) < Optimal(k+1) which contradicts our hypothesis that Optimal(k+1) is the optimal cost for a bitonic tour with *k*+1 elements.

The only bitonic tour that cannot be written in the form <0, *a*, *k*, *x*-1, *x*-2, … *k*+1, *b*, 0> is the "basic" bitonic tour, <0, *x*-1, *x*-2, …, 2, 1, 0>. Let's program a function to compute the length of the basic bitonic tour:

```
class BitonicTour {
//…
  virtual double CostBasic(int x) const
  {
    double result = distance[0][x-1];
    for (int i = 1; i < x; i++)
      result += distance[i-1][i];
    return result;
  }
};
```

We conclude that Optimal(x) either is the minimum of CostBasic(x) or the minimum of all Cost(i)+Penalty(i, x) for i from 3 to x-1. This leads us to the following definition:

```
class BitonicTour {
//…
  virtual double Optimal(int x) const
  {
    double result;
    if (x == 3)
      result = distance[0][1] + distance[1][2] + distance[0][2];
    else
    {
      result = CostBasic(x);
      for (int i = 3; i < x; i++)
        result = Min(result, Optimal(i) + Penalty(i, x));
    }
    return result;
  }
};
```

Because that sum in the if branch of the if-else statement is the same as CostBasic(3), we can rephrase the function in a more compact way:

```
class BitonicTour {
//…
  virtual double Optimal(int x) const
  {
    double result = CostBasic(x);
    for (int i = 3; i < x; i++)
      result = Min(result, Optimal(i) + Penalty(i, x));
    return result;
  }
};
```

The idea for computing the solution, now that function Optimal is available, is simple: we proceed backwards, finding out at each step which i was chosen for the minimum. All along we build the list. It starts with x-1 only. Then, if we find out that the optimal cost was computed for i = k1, we extend the solution with an edge from k1-2 to x-1 on one side of x-1 and with edges <x-1, x-2, …, k1-1> on the other side. On the first step, when only x-1 is present in the solution being computed, both sides are equivalent. On the second step we do likewise, now using k1 instead of x, so to speak. Let's suppose Optimal(k1) was computed for i = k2. We have to extend the solution with an edge for k2-2 to k1-1, and this has to go on the side (either going to the right or to the left) where k1-1 is present. The other edges go to the other side.

The operation of extending the partial solution with a new set of edges, when Optimal(y) was computed from Optimal(x) is programmed as follows:

```
class BitonicTour {
//…
  virtual void Extend(std::list<int>& s, int x, int y) const
  {
    if (y - 1 == s.back())
    {
```

```
      s.push_back(x-2);
      while (s.front() > x - 1)
        s.push_front(s.front() - 1);
    }
    else
    {
      s.push_front(x-2);
      while (s.back() > x - 1)
        s.push_back(s.back() - 1);
    }
  }
};
```

We can now program the function that actually computes the solution:

```
class BitonicTour {
//...
  virtual std::list<int> Solution(int x) const
  {
    std::list<int> result;
    result.push_back(x-1);
    while (x > 3)
    {
      int i = x - 1;
      while (i >= 3 && Optimal(x) != Optimal(i) + Penalty(i, x))
        i--;
      Extend(result, i, x);
      x = i;
    }
    //...
    return result;
  }
};
```

Note that if the inner loop exits because `i` has reached 2, then an edge from the first point is added to the solution. In this case, all points will have been added to the solution. However, in order to close the tour, the first point must be added to the other end as well. On the other hand, if the last inner loop exits with `i = 3`, then the first point will be missing from both ends, and we may safely add it. Finally, in order to have our solutions in a canonical form, we decide that the second point must be visited when going to the right. (Although we haven't made that observation yet, it is obvious that bitonic tours come in pairs: once we have one, we can obtain another one by traversing the edges in the opposite direction.)

Function `Complete` does the final processing to ensure that the solution is properly complete:

```
class BitonicTour {
//...
  virtual void Complete(std::list<int>& s) const
  {
    if (s.back() != 0)
      s.push_back(0);
    if (s.front() != 0)
      s.push_front(0);
    std::list<int>::const_iterator x = s.begin();
    x++;
    if (*x != 1)
      std::reverse(s.begin(), s.end());
};
```

Now we can finalize function `Solution`:

```
class BitonicTour {
//...
  virtual std::list<int> Solution(int x) const
  {
```

```
    std::list<int> result;
    result.push_back(x-1);
    while (x > 3)
    {
      int i = x - 1;
      while (i >= 3 && Optimal(x) != Optimal(i) + Penalty(i, x))
        i--;
      Extend(result, i, x);
      x = i;
    }
    Complete(result);
    return result;
  }
};
```

This almost concludes our exercise. Of course we do not have to use the class as it is: we can use it with automatic memoization. However, class MemoVector expects function Optimal to return an int, and ours returns a double. We are at the end of this report, so we do not bother going back and generalize class MemoVector in order to take into account the return type of function Optimal: we simply copy-and-paste it into a new class MemoVectorDouble<T>:

```
template <class T>
class MemoVectorDouble: public T {
public:
  mutable std::vector<double> optimal;
  double impossible;
public:
  MemoVectorDouble():
    optimal(),
    impossible(-1.0)
  {
  }

  virtual double Optimal(int x) const
  {
    if (x >= static_cast<int>(optimal.size()))
      optimal.resize(x+1, impossible);
    if (optimal[x] == impossible)
      optimal[x] = T::Optimal(x);
    return optimal[x];
  }
};
```

Here's a test function that generates a random set of points and then computes the shortest tour using brute force, then the shortest bitonic tour using brute force, and finally the shortest bitonic tour using memoized dynamic programming. In each case it displays the solution and its cost.

```
void TestBitonicTourRandom(std::ostream& output)
{
  MemoVector<BitonicTour, double> t;
  int size = 8;
  t.Randomize(size);
  mas::WriteLine(t.points, "  ", output);
  std::list<int> solution1 = t.SolutionCombinatorial<BitonicTour::Tours>();
  output << t.Cost(solution1) << "/" << solution1 << std::endl;
  std::list<int> solution2 = t.SolutionCombinatorial<BitonicTour::Bitonic>();
  output << t.Cost(solution2) << "/" << solution2 << std::endl;
  std::list<int> solution = t.Solution(size);
  output << t.Cost(solution) << "/" << solution << std::endl;
  output << t.Optimal(size) << std::endl;
}
```

If variable size is initialized to a number greater than 9 or 10, it may be a good idea to comment out the combinatorial computations…

Here's an randomly generated example in which the shortest tour is shorter than the shortest bitonic tour:

```
0 4  1 5  2 0  3 7  4 3  5 5  6 1  7 0
23.0219/0 1 3 5 4 6 7 2 0
24.7766/0 1 3 5 7 6 4 2 0
24.7766/0 1 3 5 7 6 4 2 0
24.7766
```

As a final observation, note that the running time of the memoized version is proportional to the cube of the number of points. Function Optimal is computed once for each number from 3 to the number of points and in each case a single iteration for finding a minimum is used. In each step of that iteration function Penalty is called once. Function Penalty uses a loop to compute the total length of the path <y-1, y-2, …, x, x-1>. Hence the cube. However, the loop can be optimized away from function Penalty by using a vector of accumulated distances:

```
class BitonicTour {
public:
  std::vector<Point> points;
private:
  std::vector<std::vector<double> > distance;
  std::vector<double> accumulated;
public:
  //…

  virtual void ComputeDistances()
  {
    distance.clear();
    distance.resize(size);
    for (int i = 0; i < size; i++)
    {
      distance[i].reserve(points.size());
      for (int j = 0; j < size; j++)
        distance[i].push_back(points[i].DistanceTo(points[j]));
    }
    double temp = 0.0;
    accumulated.reserve(points.size());
    accumulated.push_back(temp);
    for (int i = 1; i < size; i++)
      accumulated.push_back(temp += distance[i-1][i]);
  }

  //…

  virtual double Penalty(int x, int y) const
  // Penalty for extending a tour with x points to a tour with y points
  // by sending an edge from point x-2 to point y-1.
  {
    return distance[x-2][y-1]   // "direct" distance <x-2, y-1>
          + accumulated[y-1] - accumulated[x-1]  // distance <x-1, x, ... , y-2, y-1>
          - distance[x-2][x-1];  // edge <x-2, x-1> is not in the tour
  }

//…
};
```

Now, our program is quadratic.

22 Canadian Airline

Recall the Canadian airline problem from the International Olympiad in Informatics, in 1993, which was the pretext for this report, one hundred pages ago: we are given the domestic routes of a Canadian airline that serves both Vancouver, Canada's westernmost important city, and

Halifax, Canada's easternmost important city. Is it possible to start a trip in Vancouver, and, using only the company's flights, reach Halifax always traveling to the east, and then return to Vancouver, always traveling to the west, and never landing in a city that was visited on the way east? If so, compute one such trip that visits as many cities as possible.

This bears some similarity with the bitonic tour, since it is a question of first going to the east and then to the west. However, neither are we in an Euclidean situation, in which every vertex is connected to all other vertices, nor do we want the shortest tour: we want the tour that visits the most cities.

On the other hand, it is also similar to the longest path in a directed acyclic graph. If we consider only flights from west to east, we do get a directed acyclic graph, and the problem is equivalent to finding two disjoint paths from the first vertex to the last (disjoint in respect to intermediate vertices, since the initial and final vertices are common), such that the sum of their lengths is as large as possible. Note, however, that we do not have a general directed acyclic graph but one where the vertices are ordered, in the sense that vertex x+1 is more to the east than vertex x.

We will attack the problem using this second approach. As a matter of fact, the optimization function, Optimal(int x, int y), will represent the length of the longest "double path" from the first vertex to vertices x and y. A double path is a pair of paths which are disjoint except for the first element, which is the same in both (and counts once for the length). The underlying graph is directed and acyclic and the vertices are ordered. Given the situation that suggested the problem, we will name our class CanadianTour.

```
class CanadianTour {
public:
  Graph graph;
  int size;
  Graph transposed;
public:

  virtual ~CanadianTour()
  {
  }

  virtual void Read(std::istream& input = std::cin)
  {
    graph.Clear();
    transposed.Clear();
    input >> size;
    graph.Resize(size);
    transposed.Resize(size);
    int n;
    input >> n;
    for (int i = 0; i < n; i++)
    {
      int x;
      int y;
      input >> x >> y;
      graph.Link(x, y);
      transposed.Link(y, x);
    }
  }

  virtual int Optimal(int x, int y) const
  {
    int result;
    //...
    return result;
  }
```

```
  virtual std::list<int> Solution(int x)  // optimal solution ending in vertex x
  {
    std::list<int> result;
    //...
    return result;
  }
};
```

Instead of having a single undirected graph, we have two directed graphs, graph and transposed. The second is the transposed of the first, and it simplifies the operations, as we shall see shortly.

In order to discover the definition of Optimal(x, y), let's take a specific case: compute Optimal(7, 4) in a graph with at least 8 vertices. This is the optimal cost for a double path starting at 0 and ending at 7 and 4. Each double path has two parts, the first part, and the second part. In the example, the first part is the one that ends at 7 and the second is the one that ends at 4.

Using our dynamic programming insight, we assume we have already computed Optimal(0, 4), Optimal(1, 4), …, Optimal(6, 4). By the way, Optimal(4, 4) is 0, because a double path with both parts ending in the same vertex cannot exist, unless the last vertex is the initial vertex, which corresponds to Optimal(0, 0), whose value is 1. The last edge on the first part of the double path will be either <0, 7> or <1, 7>, …, or <6, 7>, provided the edge is in the forward graph. When we use edge <2, 7>, for example, the largest length we can obtain for a double path ending in 7 and 4 is 1 + Optimal(2, 4), provided Optimal(2, 4) is not 0 (i.e., provided the double path ending in 2 and 4 exists indeed). Note that edge <2, 7> is on the graph if 2 is a successor of 7 in the transposed graph. Hence, to compute Optimal(7, 4) we compute the maximum of 1 + Optimal(i, 4) for all successors of 7 in the transposed graph for which Optimal(i, 4) is greater than 0:

```
    //...
    result = 0;
    const std::list<int>& s = transposed.successors[x];
    for (std::list<int>::const_iterator i = s.begin(); i != s.end(); i++)
      if (Optimal(*i, y) > 0)
        result = mas::Max(result, 1 + Optimal(*i, y));
    //...
```

Note that the argument does not apply if the "other" part ends in a vertex more to the right. In other words, we cannot compute Optimal(4, 7) from Optimal(0, 7), Optimal(1, 7), Optimal(2, 7) and Optimal(3, 7). For example, suppose there is an edge from 2 to 4. The optimal cost for the double path obtained using that edge is not 1 + Optimal(2, 7), even if Optimal(2, 7) is not zero, because we cannot exclude that 4 is in the optimal path ending in 2 and 7. In the previous situation 7 was certainly not in the optimal path Optimal(2, 4).

On the other hand, Optimal(x, y) is the same as Optimal(y, x), of course.

These observations lead us to the following definition:

```
class CanadianTour {
//...
  virtual int Optimal(int x, int y) const
 // largest number of vertices on an double path 0 -> x, y;
  {
    int result;
    if (x < y)
      result = Optimal(y, x);
    else if (x == 0 && y == 0) // base case
      result = 1;
    else if (x == y)
      result = 0;
    else // assert x > y;
    {
      result = 0;
```

```
    const std::list<int>& s = transposed.successors[x];
    for (std::list<int>::const_iterator i = s.begin(); i != s.end(); i++)
      if (Optimal(*i, y) > 0)
        result = mas::Max(result, 1 + Optimal(*i, y));
  }
  return result;
 }
};
```

The running time of this function, in its memoized version is proportional to the cube of the number of vertices. Function Optimal is called once for each pair of vertices, and in each case it requires a loop that visits some vertices. In the worst case all vertices are visited.

In order to compute the cost of the actual optimal tour, in the sense of the problem, we compute the maximum of all tours in which one of the parts ends in the rightmost vertex and the other ends in a predecessor of that vertex:

```
class CanadianTour {
//…
 virtual int Optimal1(int x)  // optimal cost of double tour using some of the first x vertices
 {
  int result = 0;
  const std::list<int>& s = transposed.successors[x-1];
  for (std::list<int>::const_iterator i = s.begin(); i != s.end(); i++)
    result = mas::Max(result, Optimal(*i, x-1));
  return result;
 }
};
```

Now that we have the costs, we want the tours. Let us start by taking the optimal tour ending in x and y and computing the predecessor of x in that tour. Given the above argument that led to the definition of function Optimal, we can do that by undoing the computation of Optimal(x, y), but only in the case where x > y:

```
class CanadianTour {
//…
 virtual int Predecessor(int x, int y) const // pre x > y;
 {
  int result = -1;
  const std::list<int>& s = transposed.successors[x];
  for (std::list<int>::const_iterator i = s.begin(); i != s.end() && result == -1; i++)
    if (Optimal(x, y) == 1 + Optimal(*i, y))
      result = *i;
  return result;
 }
};
```

If Optimal(x, y) is zero, the result is -1, meaning that there is no predecessor. The case where any of the Optimal(*i, y) is zero is also benign, because, given the precondition, the value of Optimal (x, y) cannot be 1 in this function. (If not we would write the if instruction as if (Optimal(*i, y) > 0 && Optimal(x, y) == 1 + Optimal(*i, y)) …)

Using function Predecessor we can compute the full optimal tour iteratively. We start at the end vertex and its optimal predecessor, i.e., the predecessor that led to the optimal closed tour. This vertex is computed by yet another function:

```
class CanadianTour {
//…
 virtual int Predecessor1(int x)
 {
  int result = -1;
  const std::list<int>& s = transposed.successors[x];
  int z = Optimal1(x+1);
  for (std::list<int>::const_reverse_iterator i = s.rbegin();
```

```
        result == -1 && i != s.rend(); i++)
    if (Optimal(*i, x) == z)
      result = *i;
  return result;
 }
};
```

We visit the predecessors in reverse order, in consideration to the usual situation in which the edges were added to the graph from left to right, although this is not essential.

We represent the tour as list of integers, as in the bitonic tour. The list starts and finishes with zero:

```
class CanadianTour {
//…
 virtual std::list<int> Solution(int x)  // optimal solution ending in vertex x
 {
  std::list<int> result;
  int y = Predecessor1(x);
  result.push_front(x);
  result.push_back(y);
  while(!(x == 0 && y == 0))
   if (x > y)
   {
    x = Predecessor(x, y);
    result.push_front(x);
   }
   else
   {
    y = Predecessor(y, x);
    result.push_back(y);
   }
  return result;
 }
};
```

This solves the problem.

And what about the combinatorial solution? As before it is quite straightforward. We follow the usual scheme:

```
class CanadianTour {
//…
 virtual const std::list<int>& MostValuable(const std::list<std::list<int> >& x) const
 {
  std::list<std::list<int> >::const_iterator result;
  int max = 0;
  for (std::list<std::list<int> >::const_iterator i = x.begin(); i != x.end(); i++)
  {
   int temp = static_cast<int>(i->size());
   if (max < temp)
   {
    max = temp;
    result = i;
   }
  }
  return *result;
 }

 std::list<int> SolutionCombinatorial() const
 {
  std::list<int> result;
  Combinations c(*this);
  c.Compute();
  if (!c.combinations.empty())
```

```
    result = MostValuable(c.combinations);
  return result;
 }

public:
 class Combinations {
 public:
  const CanadianTour& ct;
  std::list<std::list<int> > combinations;
 private:
  std::list<int> temp;
  std::vector<bool> visited;

 public:
  Combinations(const CanadianTour& ct):
   ct(ct)
  {
  }

  virtual ~Combinations()
  {
  }

  virtual void Compute()
  {
   combinations.clear();
   temp.clear();
   visited.clear();
   visited.resize(ct.size, false);
   VisitRight(0);
  }

  void operator = (const Combinations&){}

 private:
  virtual void VisitRight(int x)
  {
   temp.push_back(x);
   visited[x] = true;
   if (x == ct.size - 1)
   {
    temp.pop_back();
    VisitLeft(x);
    temp.push_back(x);
   }
   else
   {
    const std::list<int>& z = ct.graph.successors[x];
    for (std::list<int>::const_iterator i = z.begin(); i != z.end(); i++)
     if (!visited[*i])
      VisitRight(*i);
   }
   visited[x] = false;
   temp.pop_back();
  }

  virtual void VisitLeft(int x)
  {
   temp.push_back(x);
   visited[x] = true;
   if (x == 0)
    combinations.push_back(temp);
   else
   {
```

```
      const std::list<int>& z = ct.transposed.successors[x];
      for (std::list<int>::const_iterator i = z.begin(); i != z.end(); i++)
        if (!visited[*i] || *i == 0)
          VisitLeft(*i);
    }
    visited[x] = false;
    temp.pop_back();
  }
 };
};
```

In order to be able to play with our solution, let us provide a function to create a random problem with a given number of vertices and a given number of edges:

```
class CanadianTour {
//…
 virtual void Randomize(int vertices, int edges)
 {
  graph.Clear();
  transposed.Clear();
  size = vertices;
  graph.Resize(size);
  transposed.Resize(size);
  while (graph.CountEdges() < edges)
  {
   int z = ::rand() % mas::Square(size);
   int x = z / vertices;
   int y = z % vertices;
   if (x != y)
   {
    graph.Link(mas::Min(x, y), mas::Max(x, y));
    transposed.Link(mas::Max(x, y), mas::Min(x, y));
   }
  }
 }
};
```

Here's a test function that uses a memoized class:

```
void TestCanadianTourRandomized(int vertices, int edges, std::ostream& output)
{
 MemoMatrix<CanadianTour> ct;
 ct.Randomize(vertices, edges);
 output << ct.graph.CountEdges() << std::endl;
 mas::WriteLine(ct.graph.successors, "\n", output);
 output << "---------------" << std::endl;
 int x = ct.Optimal1(ct.size);
 mas::WriteV(ct.optimal, "\n", 3, output);
 output << "---------------" << std::endl;
 output << x << std::endl;

 if (x > 0)
 {
  std::list<int> solution = ct.Solution(ct.size - 1);
  output << solution << std::endl;
 }
 output << "---------------" << std::endl;

 CanadianTour::Combinations c(ct);
 c.Compute();
 mas::WriteLine(c.combinations, "\n", output);
 std::list<int> solution = ct.SolutionCombinatorial();
 output << "---------------" << std::endl;
 output << solution << "/" << static_cast<int>(solution.size()) << std::endl;
}
```

Observe the result of a random run with 12 vertices and 28 edges:

```
10 8 3 6
7 6 9 3
6 4 8 7 3 5 10
4
5 9 8
8 10 11 6
8
9 11
10

11

---------------
 1  0  0 -1 -1  4 -1  0
 0  0  0 -1  0  0  0
 0  0  0 -1  0  0  0  0
 2  0  0
 3  0  0 -1  0  0 -1  0
 4  0  0 -1  0  0 -1  0 -1 -1  7  8
-1  0  0 -1 -1  5 -1  0
 0 -1  0 -1  0  0  0  0 -1 -1  0  0
-1 -1 -1 -1 -1  6 -1  0

-1 -1 -1 -1 -1  7 -1  0 -1 -1  0  8
-1 -1 -1 -1 -1  8 -1  0 -1 -1  8
---------------
8
0 3 4 5 11 10 8 6 0
---------------
0 10 11 5 4 3 0
0 8 10 11 5 4 3 0
0 3 4 5 11 10 0
0 3 4 5 11 10 8 0
0 3 4 5 11 10 8 6 0
0 6 8 10 11 5 4 3 0
---------------
0 3 4 5 11 10 8 6 0/9
```

The first group of numbers represents the graph. The second groups shows the optimal matrix from the memo. Note that the matrix has not been computed for all pairs of indices, since some computations simply have not been invoked recursively at any stage. The third group shows the results of functions Optimal and Solution. The fourth group enumerates all combinations. Finally, we see the combinatorial solution. In this case there were not many combinations and, in fact, there is only one with the maximum length.

We still have to solve the actual Canadian Airline problem, as presented in the competition, with the city names and the flights. Here is the main test file that was used in the problem description:

```
8  9
Vancouver
Yellowknife
Edmonton
Calgary
Winnipeg
Toronto
Montreal
Halifax
Vancouver Edmonton
Vancouver Calgary
Calgary Winnipeg
Winnipeg Toronto
```

```
Toronto Halifax
Montreal Halifax
Edmonton Montreal
Edmonton Yellowknife
Edmonton Calgary
```

Our approach is similar to the one we saw in the Printing Neatly problem: use a derived class, redefining function Read:

```
class CanadianAirline: public MemoMatrix<CanadianTour> {
public:
  std::vector<std::string> cities;
public:
  virtual void Read(std::istream& input)
  {
    int n;
    input >> size >> n;
    graph.Resize(size);
    transposed.Resize(size);
    cities.reserve(size);
    for (int i = 0; i < size; i++)
    {
      std::string city;
      input >> city;
      cities.push_back(city);
    }
    for (int i = 0; i < n; i++)
    {
      std::string depart;
      std::string arrival;
      input >> depart >> arrival;
      int x = static_cast<int>(std::find(cities.begin(), cities.end(), depart) - cities.begin());
      int y = static_cast<int>(std::find(cities.begin(), cities.end(), arrival) - cities.begin());
      g.Link(mas::Min(x, y), mas::Max(x, y));
      transposed.Link(mas::Max(x, y), mas::Min(x, y));
    }
  }
};
```

Here is a test function:

```
void TestCanadianAirline(std::istream& input, std::ostream& output)
{
  CanadianAirline c;
  c.Read(input);
  if (c.Optimal1(c.size) == 0)
    output << "No solution." << std::endl;
  else
    mas::WriteIndirectLine(c.cities, c.Solution(c.size-1), "\n", output);
}
```

For the above test file our program prints:

```
Vancouver
Calgary
Winnipeg
Toronto
Halifax
Montreal
Edmonton
Vancouver
```

The second sample file was:

```
5 5
C1
```

```
C2
C3
C4
C5
C5  C4
C2  C3
C3  C1
C4  C1
C5  C2
```

We got:

```
No solution.
```

23 Conclusion

Dynamic programming has always been a somewhat mysterious programming technique. If you have reached the end of this report (without skipping the intervening sections), perhaps it is a little bit less so for you now, as it is for me.

In summary, we have handled all these twenty problems in a similar, rather systematic way. We started either with a simple recursive solution for the function Optimal, which we knew was not very efficient. Using that function, we then computed the actual solution, somehow undoing the computations performed by function Optimal. In the first examples, we showed how to build a table of previously computed function results to be used instead of the recursive function calls. Then, we learned how to automate the construction of that table by using memos. We saw three types of memos: vector, for functions Optimal with one int argument, matrix, for functions Optimal with two int arguments, and map, for functions Optimal with one argument not necessarily of type int.

The main drawback of our memo classes is that the recursive function to be memoized must be called Optimal. We dealt with this issue using the design pattern Adapter, with an adaptation of our own to enforce the correct behavior in the presence of recursion.

In many cases, we complemented the exercise, and we got inspiration for the expression of function Optimal, by considering the combinatorial, brute force, solution. Even though generating all combinations in each case and picking up the best is not computationally attractive, the algorithms themselves are interesting and almost “self evident”, and the combinatorial solution did help us during the development. With the combinatorial approach and with the direct dynamic programming strategy, we had two independent solutions for the same problem. The fact that they gave the same results, even if it does not prove that the solutions are correct, greatly increases our confidence that they are.

From the technical point of view, the ease of generating all the combinations owes a lot to the C++ STL library. Without it, algorithms might have been much more cumbersome.

The examples in this report also show the usefulness of classes in solving competition style problems, which is not a surprise, but also the usefulness of inheritance and generic programming. Inheritance appeared early, when we designed the tabular dynamic programming solutions as derived classes, redefining function Optimal from the base class, where this function Optimal was programmed in a recursive, very inefficient, manner. A little later, inheritance and generic programming became the key ingredients that allowed us to program our memo classes, on which the bulk of the solutions rely. Inheritance alone showed up remarkably in the Canadian Airline and Printing Neatly problems, allowing for a very clear separation of concerns.

Finally, let us observe that the memo technique that was crucial in this exercise was made possible by the template mechanism in C++, which allows for a template class to inherit from its template argument. We have not tried to mimic this in a language without genericity, but in Eiffel, in many aspects a more carefully designed language than C++, the construction is impossible, precisely because in Eiffel a generic class cannot inherit from its generic parameter. This issue is interesting on its own and deserves further investigation.

Appendix: Utility Functions

All along in this report we have been using some utility functions, namely for computing maximums and minimums, for squaring and raising to the cube, and for writing vectors and lists. They are all template functions, and they are gathered (with other that have not appeared here) in file Utilities.h, within the namespace mas. They are all very simple and readable.

Here are the functions that compute the minimum and the maximum of two or three values of any type that provides a <= operator:

```
template <class T>
T Min(T x, T y)
{
 return x <= y ? x : y;
}

template <class T>
T Max(T x, T y)
{
 return x <= y ? y : x;
}

template <class T>
T Min(T x, T y, T z)
{
 return Min(Min(x, y), z);
}

template <class T>
T Max(T x, T y, T z)
{
 return Max(Max(x, y), z);
}
```

Next come functions Square and Cube:

```
template <class T>
T Square(T x)
{
 return x * x;
}

template <class T>
T Cube(T x)
{
 return x * x * x;
}
```

There are several functions for writing vectors and for writing lists. Here a first group of two, for vectors. In the first one we specify the separator that goes between two items; in the second, we specify the field width:

```
template <class T>
void Write(const std::vector<T>& v, const std::string& separator = " ",
           std::ostream& output = std::cout)
{
 for (typename std::vector<T>::const_iterator i = v.begin(); i != v.end(); i++)
   output << (i != v.begin() ? separator : "") << *i;
}

template <class T>
void Write(const std::vector<T>& v, int width, std::ostream& output = std::cout)
{
 for (typename std::vector<T>::const_iterator i = v.begin(); i != v.end(); i++)
```

```
   output << std::setw(width) << *i;
}
```

There are similar functions for lists:

```
template <class T>
void Write(const std::list<T>& v, const std::string& separator = " ",
           std::ostream& output = std::cout)
{
  for (typename std::list<T>::const_iterator i = v.begin(); i != v.end(); i++)
    output << (i != v.begin() ? separator : "") << *i;
}

template <class T>
void Write(const std::list<T>& v, int width, std::ostream& output = std::cout)
{
  for (typename std::list<T>::const_iterator i = v.begin(); i != v.end(); i++)
    output << std::setw(width) << *i;
}
```

The operator << is redefined for vectors and lists:

```
template <class T>
std::ostream& operator << (std::ostream& output, const std::vector<T>& v)
{
  Write(v, " ", output);
  return output;
}

template <class T>
std::ostream& operator << (std::ostream& output, const std::list<T>& v)
{
  Write(v, " ", output);
  return output;
}
```

The following two functions write the elements of the vector in the order of the indices that come in the list, one with the separator, the other with the field width:

```
template <class T>
void WriteIndirect(const std::vector<T>& v, const std::list<int>& x,
                   const std::string& separator = " ", std::ostream& output = std::cout)
{
  for (std::list<int>::const_iterator i = x.begin(); i != x.end(); i++)
    output << (i != x.begin() ? separator : "") << v[*i];
}

template <class T>
void WriteIndirect(const std::vector<T>& v, const std::list<int>& x, int width,
                   std::ostream& output = std::cout)
{
  for (std::list<int>::const_iterator i = x.begin(); i != x.end(); i++)
    output << std::setw(width) << v[*i];
}
```

For each of these functions there is another, with the same name appended by “Line”, which does the same and then adds a newline to the output stream:

```
template <class T>
void WriteLine(const std::vector<T>& v, const std::string& separator = " ",
               std::ostream& output = std::cout)
{
  Write(v, separator, output);
  output << std::endl;
}
```

```
template <class T>
void WriteLine(const std::vector<T>& v, int width, std::ostream& output = std::cout)
{
  Write(v, width, output);
  output << std::endl;
}

template <class T>
void WriteIndirectLine(const std::vector<T>& v, const std::list<int>& x,
                       const std::string& separator = " ", std::ostream& output = std::cout)
{
  WriteIndirect(v, x, separator, output);
  output << std::endl;
}

template <class T>
void WriteIndirectLine(const std::vector<T>& v, const std::list<int>& x, int width,
                       std::ostream& output = std::cout)
{
  WriteIndirect(v, x, width, output);
  output << std::endl;
}

template <class T>
void WriteLine(const std::list<T>& v, const std::string& separator = " ", std::ostream&
output = std::cout)
{
  Write(v, separator, output);
  output << std::endl;
}

template <class T>
void WriteLine(const std::list<T>& v, int width, std::ostream& output = std::cout)
{
  Write(v, width, output);
  output << std::endl;
}
```

We can write vectors of vectors, vectors of lists, lists of vectors and lists of lists using these functions, relying on the operators << that are provided. When finer control of the output is needed, for example when we want to display a rectangular array of numbers, we resort to the following functions:

```
template <class T>
void WriteV(const std::vector<std::vector<T> >& v, const std::string& separator1 = " ",
const std::string& separator2 = " ", std::ostream& output = std::cout)
{
  for (typename std::vector<std::vector<T> >::const_iterator i = v.begin();
       i != v.end(); i++)
  {
    Write(*i, separator2, output);
    output << separator1;
  }
}

template <class T>
void WriteV(const std::vector<std::vector<T> >& v, const std::string& separator1, int
width, std::ostream& output = std::cout)
{
  for (typename std::vector<std::vector<T> >::const_iterator i = v.begin();
       i != v.end(); i++)
  {
    Write(*i, width, output);
    output << separator1;
```

```
  }
}
```

These two functions handle vectors of vectors. Analogous functions exist for vectors of lists, list of vectors and lists of lists but they were not used in this report.

References

[Bentley] Jon Louis Bentley, “Experiments on traveling salesman heuristics”, Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, 1990, ISBN 0-89871-251-3, pages 91-99.

[Cormen *et al*.] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms, Second Edition*, MIT Press, 2001, ISBN 0-262-53196-8.

[Gamma *et al*.] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995, ISBN 0-201.63361-2.

[Knuth] Donald E. Knuth, *The Art of Computer Programming, Third Edition*, Addison-Wesley, 1977, ISBN 0-201-89683-4.

[Sedgewick] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983, ISBN 0-201-06672-6

[Weiss] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*, Second Edition, Addison Wesley, 1999, ISBN 0-201-36122-1.

[Wirth] Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976, ISBN 0-13-022418-9.