Homework 3, CSE 232

Due September 17 Note: On most of the problem sets through the semester, I'll put a horizontal line with "Optional" under it. Any problems below this section are encouraged - I think they're interesting and will help you learn the subject - but not necessary to complete in order to get credit for the homework.

Problem 1

This week, we're going to go through GCJ 1C 2009 - Bribe the Prisoners.

Here's the problem statement:

> In a kingdom there are prison cells (numbered 1 to P) built to form a straight line segment. Cells number i and i+1 are adjacent, and prisoners in adjacent cells are called "neighbours." A wall with a window separates adjacent cells, and neighbours can communicate through that window.
>
> All prisoners live in peace until a prisoner is released. When that happens, the released prisoner's neighbours find out, and each communicates this to his other neighbour. That prisoner passes it on to his other neighbour, and so on until they reach a prisoner with no other neighbour (because he is in cell 1, or in cell P, or the other adjacent cell is empty). A prisoner who discovers that another prisoner has been released will angrily break everything in his cell, unless he is bribed with a gold coin. So, after releasing a prisoner in cell A, all prisoners housed on either side of cell A - until cell 1, cell P or an empty cell - need to be bribed.
>
> Assume that each prison cell is initially occupied by exactly one prisoner, and that only one prisoner can be released per day. Given the list of Q prisoners to be released in Q days, find the minimum total number of gold coins needed as bribes if the prisoners may be released in any order.
>
> Note that each bribe only has an effect for one day. If a prisoner who was bribed yesterday hears about another released prisoner today, then he needs to be bribed again.

Input

> The first line of input gives the number of cases, N. N test cases follow. Each case consists of 2 lines. The first line is formatted as

```
P Q
```

> where P is the number of prison cells and Q is the number of prisoners to be released. This will be followed by a line with Q distinct cell numbers (of the prisoners to be released), space separated, sorted in ascending order.

Output

> For each test case, output one line in the format

```
Case #X: C
```

> where X is the case number, starting from 1, and C is the minimum number of gold coins needed as bribes.

Limits

> $1 \leq N \leq 100$
>
> $Q \leq P$
>
> Each cell number is between 1 and P, inclusive.

Small dataset

1 ≤ P ≤ 100

1 ≤ Q ≤ 5

Large dataset

1 ≤ P ≤ 10000

1 ≤ Q ≤ 100

a) For the small input limits (less than 100 cells, less than 5 prisoners to be released), what is the total number of possible ways to release the prisoners? Is a complete search solution feasible for the small case? How about for the large case?

There are at most $5 * 4 * 3 * 2 * 1 = 5! = 120$ different orders in which to release the prisoners under the small input limits, making a complete search solution quite reasonable. In the large case, we have instead a maximum of 100! possible ways to release the prisoners, much larger than 10 million, and so we must use an alternative approach.

b) Suppose we choose to release the prisoner in cell q first. What subproblems are we left with to solve in order to complete our solution? Can we treat these subproblems as smaller instances of the same problem? Does this problem exhibit optimal substructure?

Once we've released the prisoner in cell q, we have two completely separate blocks of cells: $[1, q-1]$ and $[q+1, P]$. We can consider each of these blocks of cells independently, as no releases in one can affect the other. That means we can treat each as smaller instances of the first problem, with smaller P and Q covering only those prisoners to be released that reside in each separate block. Because the total bribe required is the sum of P (the initial bribe for releasing prisoner q) and the bribes required for the best way to release prisoners in the first and second remaining blocks, this problem exhibits optimal substructure.

c) What are the parameters of these subproblems that we can use to treat them as instances of the original problem? What parameters should we use for the original problem?

We can use the first and last cells in each block as parameters for our subproblems, in which case the parameters for the original problem are $[1, P]$.

d) Write a recursive solution to this problem using these parameters that performs a complete search of all possible solutions and attach it to your homework. Try it on both the Small and Large inputs. Does it work on both?

```
In [ ]: import sys
        import numpy as np

        def read_input(infile):
            P, Q = np.array(infile.readline().split(), dtype=int)
            released = np.array(infile.readline().split(), dtype=int)
            return P, released

        def solve_case(P, released):

            def find_min_cost(mincell, maxcell):
                cost = 0
                valid = (mincell <= released) & (released <= maxcell)
                for num in released[valid]:
                    tmp = maxcell - mincell + find_min_cost(mincell, num-1) + find_min_cost(num+1, maxce
                    if (not cost) or (tmp < cost):
                        cost = tmp
                return cost

            return find_min_cost(1, P)

        infile = open("%s" % sys.argv[1], 'r')
        outfile = open("%s.out" % sys.argv[1][:-3], 'w')
        cases = int(infile.readline().strip('\n'))
        for i in range(cases):
            case = read_input(infile)
            output = solve_case(*case)
            outfile.write('Case #%i: %s\n' % (i+1, output))
            print 'Case #%i: %s' % (i+1, output)
        infile.close()
        outfile.close()
```

This solution works fine on the small input, but doesn't finish fast enough for the large input.

e) How many different subproblems are there for P cells and Q different prisoners to be released? Is this much larger or smaller than the total number of possible ways to release the prisoners?

Each released prisoner will produce (at most) one block ending in the cell above and one ending in the cell below it. That means there are $Q + 1$ possible upper cells for blocks (one for each released prisoner as well as P itself) and $Q + 1$ possible lower cells for blocks, giving $O(Q^2)$ total possible subproblems. This is much smaller than $O(Q!)$ required for a complete search, and so a dynamic programming solution should run much faster.

f) Using memoization (a dictionary cache to store the results of computation on subproblems), modify your complete search program to produce a dynamic programming solution and attach it to your homework. Try it on both the Small and Large inputs. Does it work on both?

```
In []: import numpy as np
       import sys

       #This is a Python decorator, which you can use to wrap functions - in this case,
       #it provides a cache for an arbitrary function.
       class memoize(object):
          def __init__(self, func):
             self.func = func
             self.cache = {}
          def __call__(self, *args):
             if args in self.cache:
                return self.cache[args]
             else:
                value = self.func(*args)
                self.cache[args] = value
                return value

       def read_input(infile):
           P, Q = np.array(infile.readline().split(), dtype=int)
           released = np.array(infile.readline().split(), dtype=int)
           return P, released
```

```
def solve_case(P, released):

    @memoize
    def find_min_cost(mincell, maxcell):
        cost = 0
        valid = (mincell <= released) & (released <= maxcell)
        for num in released[valid]:
            tmp = maxcell - mincell + find_min_cost(mincell, num-1) + find_min_cost(num+1, maxcel
            if (not cost) or (tmp < cost):
                cost = tmp
        return cost

    return find_min_cost(1, P)

infile = open("%s" % sys.argv[1], 'r')
outfile = open("%s.out" % sys.argv[1][:-3], 'w')
cases = int(infile.readline().strip('\n'))
for i in range(cases):
    case = read_input(infile)
    output = solve_case(*case)
    outfile.write('Case #%i: %s\n' % (i+1, output))
    print 'Case #%i: %s' % (i+1, output)
infile.close()
outfile.close()
```

Optional No optional work this week.