Programming Assignment 2

Deadline: Sun Nov 29, 11 :59PM

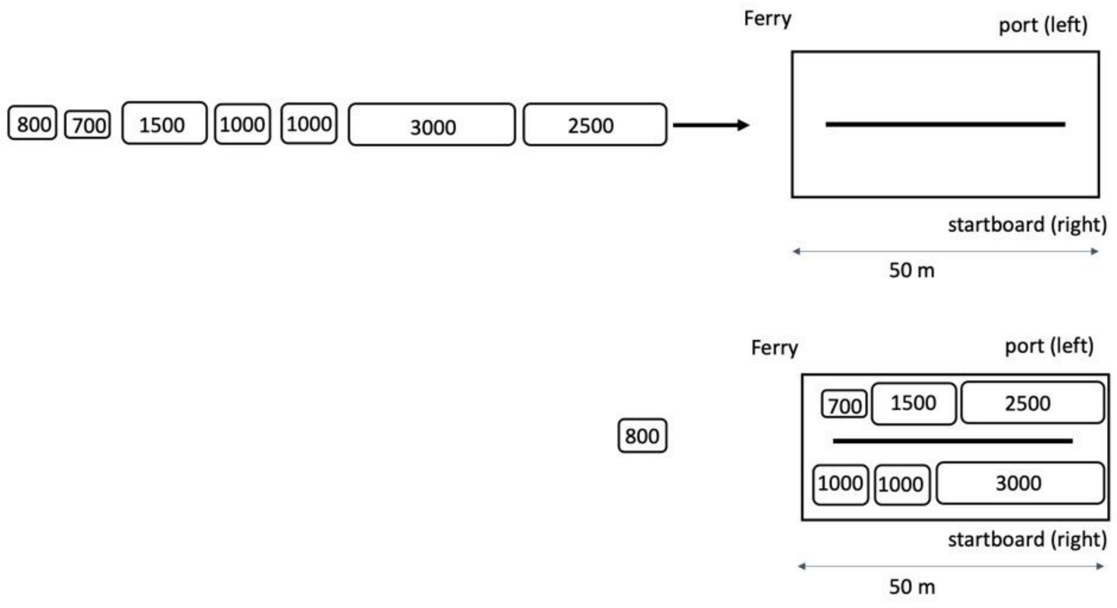
The Ferry Loading Problem

Introduction

In this assignment, you will solve the Ferry Loading problem, a problem used for practice for ACM programming competitions. The focus here is for you to solve the problem on your own, and not inspect any previous solutions to this problem. Indeed, we will be asking you to use a specific algorithm, which is not the most common method used to solve this problem. For this assignment, you will be submitting your code to an online judge which tests your code and gives a verdict. To receive the "accepted" verdict, your code needs to correctly solve their battery of tests within the allotted maximum time.

Problem Description

See the Ferry Loading problem description the next page. Here is a drawing representing the example given in the problem description:



Ferry Loading problem description PC/UVa IDs: 111106/10261 *obtained at* https://onlinejudge.org/

Before bridges were common, ferries were used to transport cars across rivers. River ferries, unlike their larger cousins, run on a guide line and are powered by the river's current. Two lanes of cars drive onto the ferry from one end, the ferry crosses the river, and the cars exit from the other end of the ferry.

The cars waiting to board the ferry form a single queue, and the operator directs each car in turn to drive onto the port (left) or starboard (right) lane of the ferry so as to balance the load. Each car in the queue has a different length, which the operator estimates by inspecting the queue. Based on this inspection, the operator decides which side of the ferry each car should board, and boards as many cars as possible from the queue, subject to the length limit of the ferry. Your job is to write a program that will tell the operator which car to load on which side so as to maximize the number of cars loaded.

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The first line of input contains a single integer between 1 and 100: the length of the ferry (in metres). For each car in the queue there is an additional line of input specifying the length of the car (in cm, an integer between 100 and 3000 inclusive). A final line of input contains the integer 0. The cars must be loaded in order, subject to the constraint that the total length of cars on either side does not exceed the length of the ferry. Subject to this constraint as many cars should be loaded as possible, starting with the first car in the queue and loading cars in order until no more can be loaded.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

The first line of outuput should give the number of cars that can be loaded onto the ferry. For each car that can be loaded onto the ferry, in the order the cars appear in the input, output a line containing '`port`' if the car is to be directed to the port side and '`starboard`' if the car is to be directed to the starboard side. If several arrangements of the cars meet the criteria above, any one will do.

Sample Input

```
1

50
2500
3000
1000
1000
1500
700
800
0
```

Sample Output

```
6
port
starboard
starboard
starboard
port
port
```

<u>Algorithm to be used</u>

You must solve the problem using **backtracking** and **memoization**, as described in article by Bento et al. 2015.[1] Memoization will use either a normal table or a Hash table; you will implement both methods.

The algorithm must decide what is the largest number M of cars that can be loaded as well as give the corresponding loading arrangement. In the previous example, we can represent the solution to the problem as: M=6 and x^*=(1,0,0,0,1,1) where x^*_i= 1 if car i went to left/port and x^*_i= 0 if car i went to right/starboard.

Backtracking is a method of trial-and-error which instead of looking over all possible solutions (there would be $\sum_{k=0}^{n} 2^k = 2^{n+1}$ possible solutions), it navigates the space of relevant solutions while avoiding the examination of many solutions that are equivalent to the ones inspected.

At each stage we build a partial solution where k cars are loaded. For the given example, we may consider k=3 and partial solution x=(0,1,0) and note that the relevant information depends on the cumulative length of the currently boarded cars (the first k) and the space available in the left/port lane which we call s, in this case s=2000cm. In our example with k=3, we have a total length of 2500+3000+1000=6500cm. Since the ferry lenth is L=50m, the space occupied on the left side is (L - s) =50m-2000cm=3000cm. We therefore have 6500-3000=3500cm occupied space on the right or 1500cm available space on the right. So we represent the state of the problem when we build x=(0,1,0) as (k=3, s=2000), and we note that I only need to consider one solution or partial solution corresponding to each state, since other solutions with the same state are equivalent. They are equivalent because they are just different arrangements of the same k cars occupying the same total amount of space in each lane.

The backtracking algorithm goes through solutions recursively, making sure states are not visited twice. From a partial solution with k cars we can build at most two other possible solutions with k+1 cars (by placing the next car on the left or right side of the ferry), and the corresponding states can be updated accordingly. In the example above, from partial solution x=(0,1,0) we can build x_left = (0,1,0,1) and x_right=(0,1,0,0) with states (k=4, s=1000) and (k=4, s=2000), respectively. In some cases, x_left or x_right may not be feasible (due to a car not fitting on the left or on the right lane) and so they are not explored (i.e "backtrack" on the process of building a solution for them). See Figure 1 in page 4 where a pseudocode of the backtracking algorithm to solve this problem is provided.

The **memorization** part, which is the recording of the visited states (see statements marked //**// in the recursive procedure), will be done in two alternative ways:

1. **Big Table:** Given a problem with n cars and Ferry length L in cm, this uses an (n+1)x(L+1) boolean array `visited` where `visited[k,s]` is true if and only if state `(k,s)` has been visited.
2. **Hash Table:** you will use a Hash function and a Hash Table **much smaller than the big table** described above to store as keys the states `(k,s)` that have been visited and permit

a later search for states in the Hash Table when needing to verify if a state has been visited.

The implementation with **Big Table** is going to be faster than the one with **Hash Table**, since indexing `visited` directly is much faster than computing a hash function plus searching on the Hash Table with possibly colliding keys. The purpose of the Hash Table is to save space since the number of visited states is supposed to be much smaller than (n+1)x(L+1).

Choosing a **Hash function** and a **Hash Table size** will be an important part of your design which should consider a tradeoff between two conflicting objectives: the one of saving memory and the one of saving time. You should experiment with various options and describe your findings on a report.

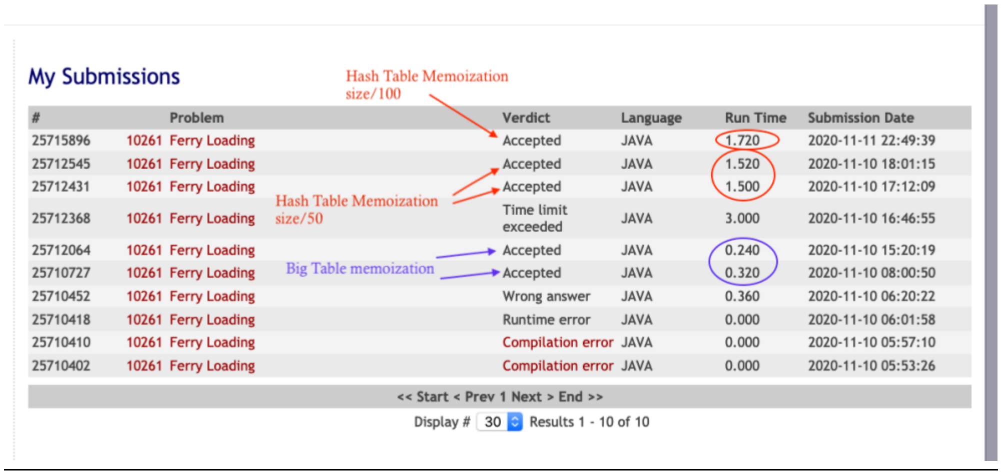
For you to have a baseline for comparison, we show the submission results of our current basic implementation (without much experimentation with hashing options or optimizations) in Figure 2.

The results show higher time spent in the Hash Table version with size reduction by factors of 50, 100.

Figure 1: Pseudocode using backtracking method

```
Global variables to the procedure Bakctrack:
integers bestK; arrays currX[0..n], bestX[0..n].
Main Method:
   Read data and initialize variables: n, length[0..n-1], L; bestK=-1
   BacktrackSolve(0,L);
   Write solution stored in bestX[0..bestK-1]
Recursive Method:
BacktrackSolve(int currK, currS)
  // currK cars have been added; currS space remains at the left side
    if currK>bestK then update bestK,bestX with currK, currX
    if currK < n then // there are cars left to consider
       if "possible to add next car to left" and "(currK+1,currS-
length[currK])was
           not visited" //**// then
                  currX[currk]=1; newS=currS-length[currK];
                  BacktrackSolve(currK+1,newS)
                  Mark (currK+1,newS) as visited. //**//
       if "possible to add next car to right" and "(currK+1,currS) was
           not visited"//**// then
                  currX[currk]=0;
                  BacktrackSolve(currK+1,currS)
                  Mark state(currK+1,currS)as visited //**//
```

Figure 2: Sample submission results for memorization using BigTable and HashTable


Hash Table Memoization size/100
Hash Table Memoization size/50
Big Table memoization

My Submissions

#	Problem	Verdict	Language	Run Time	Submission Date
25715896	10261 Ferry Loading	Accepted	JAVA	1.720	2020-11-11 22:49:39
25712545	10261 Ferry Loading	Accepted	JAVA	1.520	2020-11-10 18:01:15
25712431	10261 Ferry Loading	Accepted	JAVA	1.500	2020-11-10 17:12:09
25712368	10261 Ferry Loading	Time limit exceeded	JAVA	3.000	2020-11-10 16:46:55
25712064	10261 Ferry Loading	Accepted	JAVA	0.240	2020-11-10 15:20:19
25710727	10261 Ferry Loading	Accepted	JAVA	0.320	2020-11-10 08:00:50
25710452	10261 Ferry Loading	Wrong answer	JAVA	0.360	2020-11-10 06:20:22
25710418	10261 Ferry Loading	Runtime error	JAVA	0.000	2020-11-10 06:01:58
25710410	10261 Ferry Loading	Compilation error	JAVA	0.000	2020-11-10 05:57:10
25710402	10261 Ferry Loading	Compilation error	JAVA	0.000	2020-11-10 05:53:26

<< Start < Prev 1 Next > End >>

Display # 30 Results 1 - 10 of 10

Your tasks:

Part 1: Backtracking using Memoization using Big Table

- Implement this version of the code as efficiently as possible (inefficient solutions may not pass the online judge).
- Debug and test using the test cases provided with this assignment.
- Test using the online judge and record the results by taking screen shots of the veridic for your submissions. NOTE: If this faster solution does not pass the time limit of the online judge then there is no chance that the solution with hashing will; so you must try to improve this part in such a way that you can successfully pass the online judge test with the smallest time you can.

Part 2: Backtracking using Memoization using Hash Table

- Design a Hash function and a method for deciding your Hash Table size to be used in your Hashing implementation. Use your ingenuity and creativity; there isn't a right answer but you will be judged by your success in both reducing table size and being able to have your solution accepted by the online judge (correct and running all their tests within the maximum time limit of 3.000 seconds). For the **implementation of the Hash Table**, you are free to use HashMap from the Java standard library, use your own Hash Table implementation, or adapt the Hash Table implementation provided as solution in your lab. For Java STL HashMap choose `initialCapacity` large enough to avoid triggering the expensive re-hashing; see: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

- Implement this version of the code as efficiently as possible (inefficient solutions may not pass the online judge).
- Debug and test using the testing provided with this assignment.
- Test using the online judge and record the results by taking screen shots of the veridic for your submissions.
- If needed iterate previous steps to obtain improvements; record this process to add to the report.

Part 3: Report

Write a short report about your findings, including the following sections and information:

A. Status of Part 1: Summary of code correctness indicating if it passed all our tests and if it was accepted by the online judge; provide online judge screen shots and report on running time for the online judge on this test.
B. Status of Part 2: Summary of code correctness indicating if it passed all our tests and if it was accepted by the online judge; provide online judge screen shots and report on running time for the online judge on this test. Report on your approach and reduction on table size.

```
Details on design of Part 2: Specify your hash function, your
choice for hash table size as a function of the input, your
choice of hash map code/implantation. Give any details used in
your design and describe any improvements in performance you
obtained in this process.
```

Requirements

- You must follow the submission specs for the online judge and the specifications of the problem:
  - https://onlinejudge.org/index.php?option=com_content&task=view&id=14&Itemid=29
  - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1202
- Review thoroughly the tabs: Additional information>Submission Specification and look at Java Specifications in the online judge site, and check their Java sample code.
- Since the input file is required to be given as the standard input stream (System.in) and the output file is required to be sent to the standard output stream (System.out), use the error stream (System.err) for any debugging printouts or any statistics printouts required in this assignment. Note that when you submit to the online judge, these extra printouts to System.err must be commented out as they unnecessarily increase your running time possibly leading to exceeding the maximum time of 3.000 secs total allowed.

- Your submission must be in a zipped folder called p2<student number>. In the main directory of this folder we must have:
  - subdirectory BigTable containing: Main.java for your version of the problem that uses BigTable memoization, results of your run of this program for given tests in the format output1.txt, output2.txt, etc corresponding to input1.txt, input2.txt, etc.
  - subdirectory HashTable containing Main.java for your version of the problem that uses BigTable memoization, results of your run on this program for given tests in the format output1.txt, output2.txt, etc corresponding to input1.txt, input2.txt, etc.
  - A file called report.pdf or report.doc containing your complete report.

```
Optional readme.txt file explaining the status of your code,
known bugs, success in various tests, or anything your wish to
tell the marker, etc.
```

Marking Scheme (60 marks, 100%)

- Part 1: Code design and quality for backtracking with Big Table memorization (12 marks, 20%)
  Correctness for basic testing (6 marks, 10%)
  Online judge acceptance (6 marks, 10%)
- Part 2: Code design and quality of Hash Table design for memorization. (12 marks, 20%)
  Correctness basic testing (6 marks, 10%)
  Online judge acceptance (6 marks, 10%)
- Part 3 Report parts A & B (3 marks, 5%)
  Report part C: explanation and analysis of hash table design and results (9 marks, 15%)