The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 8

Date Due: April 4, 2017, 10:00pm

Total Marks: 75

1 Submission Instructions

- Assignments must be submitted using Moodle.
- Responses to written (non-programming) questions must be submitted in a PDF file, plain text file (`.txt`), Rich Text file (`.rtf`), or MS Word's `.doc` or `.docx` files. Digital images of handwritten pages are also acceptable, provided that they are **clearly** legible.
- Programs must be written in Java.
- If you are using IntelliJ (or similar development environment), do not submit the Module (project). Hand in only those files identified in Section 5. Export your `.java` source files from the workspace and submit only the `.java` files. **Compressed archives are not acceptable.**
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

Nice and short this time!

2.1 The Knapsack Problem

The knapsack problem, like the Travelling Salesperson Problem, cannot be solved by an algorithm on a silicon computer in less than exponential time (unless $P = NP$). The problem definition is as follows.

You have a set of n items and each item has a *weight* and a *value*. The i-th item has a weight w_i and a value v_i . You also have a knapsack (bag) that can carry a total amount of weight equal to W. The problem is to find the most valuable set of items that you can fit into the knapsack. In other words, you need to find the most valuable subset of items whose total weight is less than W.

For the more mathematically inclined, this can be formally stated as:

$$\text{Maximize } \sum_{i=1}^{n} x_i v_i \text{ subject to } \sum_{i=1}^{n} x_i w_i < W.$$

where x_i is either 0 or 1 (indicating whether or not the i-th item is in the knapsack).

We will investigate backtracking and greedy solutions to the knapsack problem in Questions 2 and 3.

3 Your Tasks

Question 1 (30 points):

For this problem you will write a method (or methods) to sort an array of strings using the MSD Radix Sort. For purposes of this assignment, you may assume that strings contain only the uppercase letters A through Z.

You have been provided with an IntelliJ module `RadixSortMSD-Template` which includes a short main program that will load a data file containing strings to be sorted. There are several files provided named `words-XXXXXX.txt` where "`XXXXXX`" denotes the number of words in the file. The file format starts with the number of words in the file, followed by one word per line. There is also a file `words-basictest.txt` which is a good set of words to use to determine whether your sort is running correctly.

The pseudocode for MSD Radix Sort from the notes is duplicated on the next page for your convenience. **Note that we are removing the optimization of sorting short lists with insertion sort, as indicated by the strikethrough text.** You may just always recursively radix sort any list with more than one element on it.[1]

Complete the following tasks:

1. Write your sort method(s) within the `RadixSortMSD` class. It should accept an array of strings as input, and return nothing. When the method returns, the array that was passed in should be in lexicographic order (i.e. dictionary order).
2. Call your sort at the spot indicated in the main() function.
3. Record in a file called `a8q1.txt/doc/pdf` the time in milliseconds it takes to sort 50, 100, 500, 1000, 10000, 50000, and 235884 items (there are input files with each of these numbers of words provided). Include this file in your assignment submission.
4. When you hand in `RadixSortMSD.java`, leave the input file set at `words-basictest.txt` so that it is easy for the markers to run your program on this input file to see that it works.

Assessment

There will be marks allotted to the following aspects of your solution:

Correctness. As always, the solution has to work!

Design and speed of your implementation. Design and speed will be considered together because they influence each other — a poor design choice may result in a slower runtime. Design-wise, any reasonable design will be accepted; marks will be deducted for especially poor choices. Speed-wise, the bar to get full marks here will be fairly low, but if your sort is egregiously slow we will deduct some points.

Javadoc and inline commenting. As usual, include a javadoc comment with each header, and document meaningful blocks of code with inline comments to enhance understanding of the code.

[1] You can, however, use the insertion-sort optimization if you want to, but you will probably have to implement your own insertion sort. If you choose to attempt this optional optimization, you are permitted to use resources from the internet to implement insertion sort, but **only** for the insertion sort, and **only** if you properly attribute any code you use to its original author or website.

Implementation Hints

- One of the most important decisions you have to make in your implementation is the choice of data structure to represent the array of lists used in the sortByDigit() helper method. Choose carefully! Your choice **will** have an impact on the speed of your sort, and the ease of implementing it. When considering which data structure to use, you may select from containers in either lib280 or the standard Java API. Case in point: on my first attempt, I made a bad decision that caused the sort of 10000 words to take several minutes. Now it takes 24ms (on my computer).
- Although runtimes will vary from machine to machine, on a decent machine you should be able to sort even the 235884-word input file in less than one second (1000ms). Even on slower machines if it is taking more than a few seconds, then you've done something particularly inefficient.
- Don't take the pseudocode below too literally. This is very high-level pseudocode which is intended to describe the operation of the algorithm, but intentionally glosses over a lot of details that become important at implementation-time (that's what pseudocode is **for**!). You need to fill in those details as you go. Don't be afraid to do what you need to do to get the job done, but that said, you should not need to write hundreds of lines of code (if you are, you should seek help and advice from Mark or a TA).

```
Algoirthm MsdRadixSort(keys, R)
keys - keys to be sorted
R - the radix

sortByDigit(keys, R, 0)


Algorithm sortByDigit(keys, R, i)
keys - keys to be sorted
R - the radix
i - digit on which to partition -- i = 0 is the left-most digit

    for k = 0 to R-1
        list[k] = new list  // Make a new list for each digit

    for each key
        if the i-th digit of the key has value k add the key to list k

    for k = 0 to R-1
        if there is another digit to consider
            ~~if list[k] is small~~
                ~~use an insertion sort to sort the items in list[k]~~
            ~~else~~
                sortByDigit(list[k], i+1)

    keys = new list // empty the input list

    For k = 0 to R-1
        keys = keys append list[k]
```

Question 2 (20 points):

This problem deals with backtracking solutions to NP-Complete and NP-hard problems. These are problems that cannot be solved by a silicon computer in less than exponential time, that is, $O(k^n)$ for some k. The travelling salesperson problem (TSP) is an example of such a problem.

You have been provided with an IntelliJ project, `TSP` which contains two backtracking solutions to TSP. The "Naive" solution simply traverses the entire search space and tries **every** possible solution The other version prunes the search space by detecting non-feasible partial solutions. If a partial solution has a cost that exceeds the cost of the best solution found so far, the search immediately backtracks. This greatly decreases the number of path extensions that are tried. You can see the large difference when you run the program because both algorithms will count the number of extensions of partial solutions that are tried and report them. It is possible to do even better than this, if you include heuristics.

The project contains several files of the from `graphXX.dat` where "XX" denotes the number of nodes in the graph. Try changing the input file in the main program to graphs of different size to see the effect of early detection of non-feasible partial solutions. The number of path extensions will be much lower compared with trying every solution, and that there is a corresponding increase in efficiency.

However, you will also note that at a certain point, the graph gets too large for **either** version to solve the problem in a reasonable amount of time because the savings gained from detecting non-feasible partial solutions is **rapidly** outpaced by number of possible solutions. Why? Because the number of possible solutions increases exponentially with the size of the graph, but the time savings gained by detection of non-feasible partial solutions increases only sub-exponentially. Try different input files and try to determine the graph size at which the runtime exceeds 10 or 15 minutes. It will probably be smaller than you think!

The knapsack is another NP-complete problem that was described in Section 2. You have been provided with a project called `Knapsack-template` which contains a few things to get you started. In `Knapsack.java` you will find definitions of:

`Item` **class:** `Item` is a non-public class that stores the properties of an item (i.e. it's weight and value).

`KnapsackInstance` **class:** The non-public `KnapsackInstance` class contains all of the information about an instance of the knapsack problem to be solved, including the capacity of the knapsack, the number of items, and the items themselves (stored as an array of `Item` objects).

`Knapsack` **class:** The `Knapsack` class is where you will write code to solve the knapsack problem. Here you will find an already completed method called `readKnapsackInstance` which can read a data file describing an instance of the knapsack problem and return an instance of `KnapsackInstance`.

Also in the project, you will find several input files named `knapsack-XXXX.dat` where the "`XXXX`" denotes the number of items in the problem instance. Each contains the data for a different instance of the knapsack problem. The first line of each file contains the total weight capacity of the knapsack as a floating-point number. The second line of each file contains the number of items n (an integer). The subsequent n lines each contain two floating-point numbers which describe a single item: the value of the item, and the weight the item. There is also a small `knapsack-basictest.dat` file which you should use first to ensure your solutions work.

Now, here are your tasks:

1. Study the backtracking solutions to TSP to gain an understanding of how they work.
2. Write a backtracking solution to the Knapsack problem in the `Knapsack` class. The approach will be similar to that of the TSP. You need to systematically try every possible solution (i.e. every subset of the set of items) while using optimizations such as detection of non-feasible solutions and heuristics where possible to prune the search space. The solution should take the form of a method that returns the total value of the most valuable set of items that can fit in the

knapsack. You may, of course, write other helper methods if you deem it appropriate. Since you are implementing a backtracking solution it is expected that you should be able to obtain the optimum solution. The optimum solution for `knapsack-basictest.dat` is a set of items worth 540.0. Note: you do not have to determine the actual set of items, just the value of the most valuable set of items that will fit in the knapsack.

Hints

Note that the quality of your solution, how you prune the search space, and choice of heuristics will determine which input files you can process in a few minutes and which you cannot. For example, on my computer, my solution (which is not especially clever) starts choking when the problem instance gets up over 500 items. Depending on your solution and your computer, you may do better or worse.

Assessment

This question will be assessed on correctness of your implementation. While we will not be assessing your commenting, you should still practice good commenting style to aid the markers in understanding how you designed and implemented your solution. A happy marker is a generous marker!

Question 3 (25 points):

The following is a greedy algorithm for solving the knapsack problem:

```
Given: a set of items with known weight and value,
       and a knapsack with known capacity.

Start with an empty knapsack.

while there is at least one item not in the knapsack that will fit
   add to knapsack the most valuable item that can still fit
```

The idea here is that we repeatedly look at how much weight capacity remains in the knapsack, and then add the most valuable item that has weight less than or equal to the remaining capacity. This algorithm will terminate when the remaining capacity in the bag is smaller than the weights of any remaining items not in the bag. It should be clear that this algorithm can be implemented to run in polynomial time. However, note that this algorithm is **not** guaranteed to produce the optimal solution.

Here are the tasks for this question:

1. Implement the greedy algorithm for the knapsack problem given above. Write your solution in `Knapsack.java` (yes, `Knapsack.java` should contain your code for both questions 2 and 3!). Use comments to make sure that your solution to questions 2 and 3 are **clearly labeled** so that the marker can identify them easily. The solution should be in the form of a method that returns the total value of the items that were packed in the knapsack. Again, you do not have to return the actual set of items in the knapsack, just the sum of the value of the items in the knapsack. By luck, the greedy algorithm manages to find the optimum set of items worth 540.0 when given `knapsack-basictest.dat` as input. This will not be the case for many of the other given problem instances.
2. For each input file given, compare the result of the greedy algorithm to the result for the backtracking algorithm. In a file called `a8q3.txt/doc/pdf`:
   (a) For each input file, record the total value of the items in the knapsack for each algorithm, let's call these $V_{backtrack}$ and V_{greedy}.
   (b) For each input file, record the ratio $\frac{V_{greedy}}{V_{backtrack}}$. This ratio represents the percentage of the true optimum value that the greedy algorithm was able to achieve.
   (c) Extrapolate or hypothesize, to the extent possible, on the relative performance of the greedy algorithm vs the backtracking algorithm. Do you think that the greedy algorithm can provide any guaranteed minimum performance relative to the optimum? Will the greedy algorithm always choose items with value at least 30% of the optimum? 50%? 90%? Based on your observations, what do you think?

Hints

If you want to generate more problem instances to test your hypothesis for part 2(c), you can use the `RandomKnapsackInstance` class that is provided. At the top of the main file you can change the capacity of the knapsack, the number of items, the maximum weight, and value of an item. Run the program and it will generate a random datafile that describes an instance of the knapsack problem that can be read in by the `readKnapsackInstance` method of the `Knapsack` class.

Assessment

This question will be assessed on correctness of your implementation (tasks 1 and 2(a)), and the observations that you make for task 2(c).

4 Files Provided

lib280-asn8: A copy of lib280 which includes:

- solutions to assignment 7;

RadixSortMSD-Template: The project template for question 1.

TSP: Examples of backtracking solutions to the TSP.

Knapsack-Template: The project template for question 2.

5 What to Hand In

RadixSortMSD.java: Your completed radix sort from question 1.

a8a1.txt/doc/pdf: Your timing observations from your sort in question 1.

Knapsack.java: Your backtracking and greedy solutions to the knapsack problem of questions 2 and 3.

a8q3.txt/doc/pdf: Your comparison of the greedy and backtracking algorithms and discussion of your observations from Question 3, task 2.