Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2025

Problem Set 6

Group Formation: 4:00 PM, Friday, April 25, 2025
Due: 4:00 PM, Monday, May 5, 2025 (firm, no late submissions)

You may work alone or with a partner or two on this assignment. However, in order to make sure you learn the material and are well-prepared for the exams, you should work through the problems on your own before discussing them with your partner, should you choose to work with someone. In particular, the "you do these and I'll do these" approach is sure to leave you unprepared for the exams.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 4:00 PM, Friday, April 25, 2025. This applies to those who choose to work alone as well!

There is a significant amount of work to be done here, and you are sure to have questions. It will be difficult if not impossible to complete the assignment if you wait until the last minute. A slow and steady approach will be much more effective.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `ps6-yourgitname`, for this lab. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

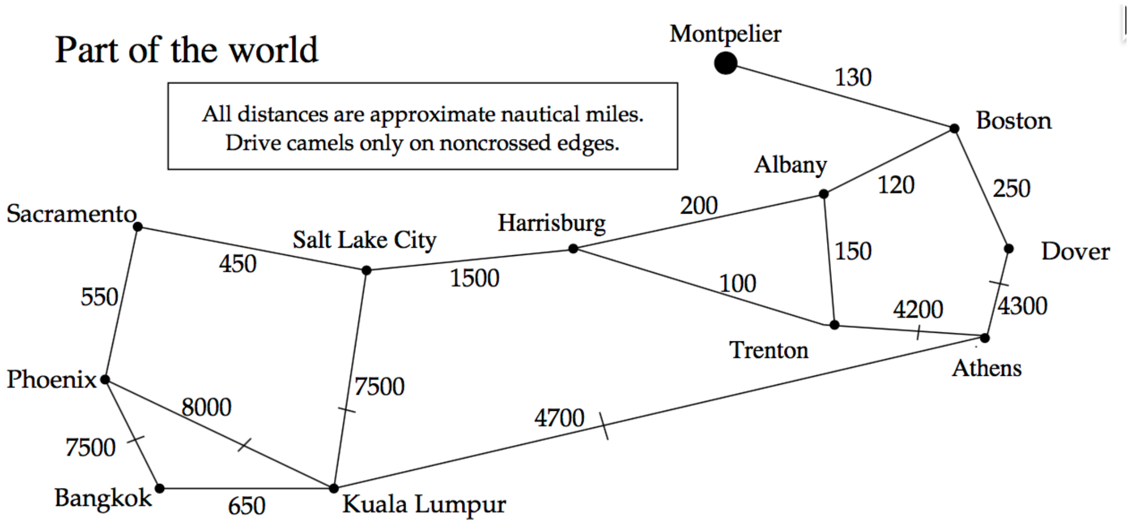
Submitting

Please submit a hard copy (typeset preferred, handwritten OK but must be legible) for all written questions. Only one submission per group is needed.

Your submission requires that all required code deliverables are committed and pushed to the main branch for your repository's origin on GitHub. If you see everything you intend to submit when you visit your repository's page on GitHub, you're set.

Greedy Algorithms

For these problems, you may wish to print the last few pages and write in your answers on those tables and graphs or make your own tables where you can record similar information. You will be working with the graph below, (credit: Bailey Problem 16.7, p. 436) for all of the questions in this section. The last page of this assignment also contains two additional copies of the graph convenient for printing and drawing.



Question 1: Use Dijkstra's algorithm to compute the shortest distance from Dover to all other cities by filling in the appropriate tables, using the algorithm and notation as shown in the example from class. Please read the remaining instructions below the graph before proceeding. (10 points)

The table you will be completing is a map (*i.e.*, a lookup table which stores the values associated with a set of keys) where the cities are the keys and the values are pairs that group the shortest distance from Dover to the that city and last edge traversed on that shortest route.

It is easiest to specify edges by the labels of their endpoints rather than the edge labels, as those are not necessarily unique.

Also, keep track of your priority queue. Remember, don't erase entries when you remove them from the queue, just cross them out and mark them with a number in the "Seq" column of the table entry to indicate the sequence in which the values were removed from the queue. Put a check in the "Added?" column if the edge resulted in a new entry being added to the map.

Question 2: Working your way back up the table until you get to Dover, what is the computed shortest path from Phoenix? (2 points)

Question 3: Now apply Prim's Algorithm, again starting in "Dover". Please bold the edges in the resulting spanning tree on the copy of the graph below, and use the table in the work packet to keep track of your priority queue. Do not erase entries when you remove them from the queue, just cross them out and mark them with a number in the "Seq" column of the table entry to indicate the sequence in which the values were removed from the queue. Place a check in the "Added?" column if the edge was added to the spanning tree. (5 points)

Question 4: Apply Kruskal's Algorithm to the graph we have been working with. Please bold the edges in the resulting spanning tree on the copy of the graph below, and use the table in the work packet to keep track of your priority queue. Do not erase entries when you remove them from the queue, just cross them out and mark them with a number in the "Seq" column of the table entry to indicate the sequence in which the values were removed from the queue. Place a check in the "Added?" column for edges that are added to the spanning tree. (5 points)

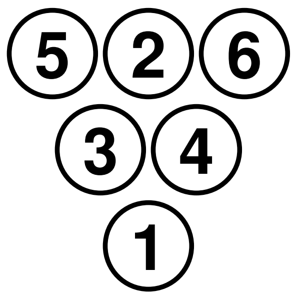
Backtracking Practice

Question 5: Read Levitin p. 427–428, which describes the subset-sum problem. For each non-solution leaf in Figure 12.4 on p. 428 of Levitin, explain where the numbers in the condition come from and why that means the search can be cut off at that point. (5 points)

Question 6: Draw a complete state-space tree for a backtracking approach to the subset-sum problem as applied to the example in Levitin Exercise 12.1.8 (a), p. 431. (5 points)

More Backtracking: The Billiard Ball Problem

The *billiard ball problem* consists of arranging a triangle of numbered billiard balls such that the resulting layout has the following arithmetic property: every ball below the top level is the absolute value of the difference between two balls immediately above it. For instance, the following is a solution to the problem when the top row consists of three balls (which requires a total of six balls).



Note that the balls are numbered from 1 to 6 in this case.

For the problems with four and five balls in the top row, there are a total of 10 and 15 balls, respectively.

You can read more about the problem in Martin Gardner's *Penrose Tiles to Trapdoor Ciphers: And the Return of Dr Matrix* (Cambridge Press 1997) on pages 119-120. (`http://books.google.com/books?id=8-FlYl6-ML8C&lpg=PP1&pg=PA119#v=onepage&q&f=false`)

As far as I can tell, there are no known solutions for problems larger than five balls in the top row.

A Backtracking Approach

A backtracking algorithm, very similar to that for n-Queens problem, is one approach to solving this problem.

The idea is that we attempt to build up a candidate solution by adding balls to the top row. Each time a ball is added, we make sure we have not broken any of the rules (in this case, there is just one: there are no repeated numbers in the triangle generated by that top row, including the top row itself). If we have not yet broken a rule, we either have found a solution (if the top row now contains the desired number of balls) or we have a partial candidate solution and we should add another ball. Any time we generate a candidate solution that does violate the rule, we have hit a dead end, so we undo the most recent addition and try the next option. If we ever backtrack all the way to the beginning and have run out of options for our first move, we know no solution exists.

For example, consider a backtracking solution to the problem where there are 2 balls in the top row, and we choose numbers from the largest to the smallest each time we reach a decision point. (Note that this is a good strategy to get a solution more quickly, as larger numbers will tend to be in the top row.)

We start with an empty solution, and we are ready to add the first ball to the top row. Since we are trying numbers from the largest to smallest, we start with 3:

```
(3)
```

This is a legal configuration: there are no repeated digits when we expand this out (in fact, there is no expansion needed for a single ball). So we accept this as a partial solution and move on, trying to add a ball to the second position. The first ball we attempt to place at this position is the highest numbered, 3:

`(3 3)` which expands to

```
(3 3)
 (0)
```

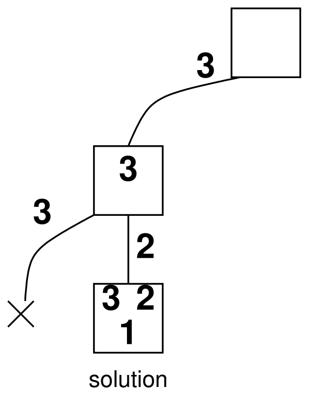
This is not a legal configuration: it includes two 3's. So we backtrack and erase our last move, and instead try the next option, which is to use the 2 ball:

`(3 2)` which expands to

```
(3 2)
 (1)
```

This is a legal solution: no repeats. Plus, we now have filled the top row, so our solution is complete.

We can display this in the format of a state space search diagram, like the text does for n-Queens in Figure 12.2 on p. 426.



Question 7: Show the state space search diagram for a backtracking solution to the problem with two balls in the top row if we instead chose balls for each position in increasing instead of decreasing numerical order. (4 points)

Now, let's consider the start of the procedure for the much more interesting (and much longer) backtracking computation of a solution to the problem with three balls in the top row. Note that here, we have a total of six balls.

Our first move is to place the largest number into the top row.

```
(6)
```

This is again legal, so we continue by adding a second number.

```
(6 6)
```

This contains a duplicate, so we backtrack and try a 5 in the second position.

```
(6 5)
 (1)
```

This is legal, so we accept the 5 for now, and start working on the third ball. We begin, as before, with the highest numbered ball and work our way down if we encounter illegal moves.

```
(6 5 6)
 (1 1)
  (0)
```

This has duplicates, so it is not legal. In fact, all of our choices for the third ball will result in illegal configurations here:

```
(6 5 6)    (6 5 5)    (6 5 4)    (6 5 3)    (6 5 2)    (6 5 1)
 (1 1)      (1 0)      (1 1)      (1 2)      (1 3)      (1 4)
  (0)        (1)        (0)        (1)        (2)        (3)
```

So this means 5 in the second position of the top row was a dead end. And we backtrack, and try a 4 there instead:

```
(6 4)
 (2)
```

So far so good here, so we move on trying each ball in the 3rd position of the top row...

Question 8: Draw the state space search diagram for the 6-ball solution that will result from the above procedure. Note that this will not necessarily lead to the sample solution pictured earlier in this document. (8 points)

Question 9: Write a program to find solutions to the billiard ball problem. It should be very similar to the n-Queens solution (use that as a guide or as a starting point). Include your code in your GitHub repository. Be sure it is well documented and that class, variable, and method names are appropriate for this problem. (15 points)

Question 10: Using your program, show the solutions computed or indicate that no solution is found for 1 through 9 balls in the top row. Give the elapsed time for each and the number of nontrivial recursive calls made in your repository's `README.md` file. (4 points)

Question 11: In the worst case, what is the Big O growth rate of this problem in terms of the number of balls in the top row? (2 points)

Finishing the Sorting Empirical Study

Earlier in the semester, you completed an empirical study of the naive sorting algorithms we had studied up to that point.

There is too much going on to have everyone add the advanced sorting algorithms to your code, run that code to generate timing and operation count data, and analyze that data. So the first two parts are done for you. On the Canvas page for this assignment, there is a link to a repository that contains Java code that implements most of the sorting algorithms we have studied this semester, all instrumented to count the number of comparisons and the number of swaps/assignments performed to accomplish the sorting, as well as the elapsed time taken to do it. The repository also includes a plain-text CSV file with these results for all of the implemented sorting algorithms, and an Excel spreadsheet with that same data formatted a bit and with charts inserted to visualize the trends as the problem size changes. Implementations and data are provided for bubble sort, selection sort, insertion sort, merge sort, quicksort (standard pivot selection, random pivot selection, and median of three pivot selection), and heapsort, and each has been run on arrays that were initially random, nearly sorted, sorted, and reverse sorted.

Question 12: For each sorting algorithm (all 8, including the ones you studied in the previous assignment and all 3 variants of quicksort), briefly discuss the timings and operation counts for each input type and relate those to the theoretical expectations. (20 points)

Question 13: Considering only the implementations of merge sort, quicksort with standard pivot selection, and heapsort, and considering only the random input arrays, rank them in terms of memory overhead, running time, number of comparisons, and number of swaps. (5 points)

Grading

This assignment will be graded out of 90 points.

Feature	Value	Score
Question 1	10	
Question 2	2	
Question 3	5	
Question 4	5	
Question 5	5	
Question 6	5	
Question 7	4	
Question 8	8	
Question 9	15	
Question 10	4	
Question 11	2	
Question 12	20	
Question 13	5	
Total	90	

City	(distance,last-edge)
Dover	(0, `null`)

Table 1: Map of Dijkstra's algorithm results for last edge traversed to find each city for the first time.

(distance,last-edge)	Seq	Added?

Table 2: Table showing the progress of the priority queue in Dijkstra's algorithm.

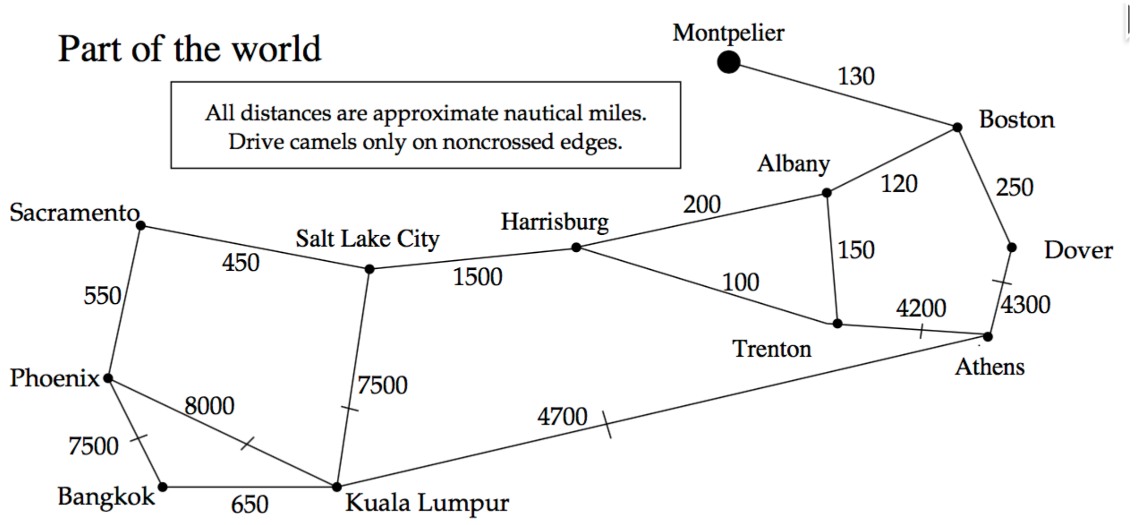
Candidate Edge	Seq	Added?

Table 3: Table showing the progress of the priority queue in Prim's algorithm.

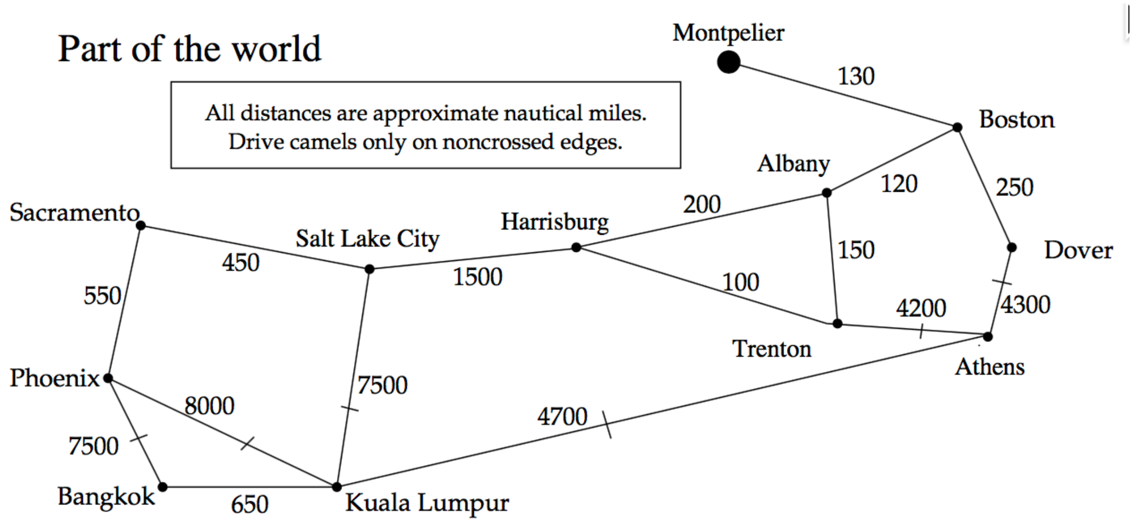
Candidate Edge	Seq	Added?

Table 4: Table showing the progress of the priority queue in Kruskal's algorithm.

Copy of the graph for drawing the Prim's Algorithm spanning tree.



Copy of the graph for drawing the Kruskal's Algorithm spanning tree.


Part of the world
All distances are approximate nautical miles.
Drive camels only on noncrossed edges.
Montpelier
130
Boston
Albany
120
250
Dover
Sacramento
Salt Lake City
Harrisburg
200
450
1500
150
100
550
4200
4300
Trenton
Athens
Phoenix
8000
7500
4700
7500
Bangkok
650
Kuala Lumpur