1. Suppose you manage a small grocery store with a single checkout line, and you want to determine the expected time your customers will have to wait in line before being checked out. Assume that a new customer arrives in line every 1 to 3 minutes with equal probability, and the clerk checks out his customers in 1 to 8 minutes with the following probabilities:

<table>
<thead>
<tr>
<th>Minutes</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.10</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>0.20</td>
</tr>
<tr>
<td>5</td>
<td>0.20</td>
</tr>
<tr>
<td>6</td>
<td>0.15</td>
</tr>
<tr>
<td>7</td>
<td>0.10</td>
</tr>
<tr>
<td>8</td>
<td>0.05</td>
</tr>
</tbody>
</table>

When a customer arrives, he or she gets in a single file line behind the other customers that have arrived previously. When the checkout clerk becomes free, the customer in front of the line goes to the clerk to checkout. A customer’s waiting time is defined to be the difference between the time he or she leaves the line to begin checkout and the time he or she arrived in line in the first place.

We can use this information to simulate our grocery store and arrive at an answer to our question. We will think of customer arrivals and departures as events in this simulation. Each event will have a time associated with it, and all pending events are stored in an event queue. A virtual clock ticks once each minute. With each tick of the clock, we look in the event queue for events that should happen at the current time. The simulation handles each such event before ticking the clock again.

Notice that this event queue really should be a minimum priority queue, ordered by event times! With each clock tick, we want to check if the event in the queue with minimum time has the current time. If it does, we extract it (take the person out of line) and service it (check the customer out). Then we check to see if the new minimum event also has the current time, continuing until the minimum event has a time greater than the current time. When this happens, we tick the clock and repeat the process with the new time. Equivalently, since nothing happens at times for which there is no event, we can simply move the clock ahead to the time of the next event each time an event is extracted from the queue.

Your min-priority queue ADT will need to include 3 operations — \texttt{MINIMUM}, \texttt{EXTRACTMIN}, and \texttt{INSERT}. Implement this ADT as a template class in C++ using a binary heap. Here is the definition:

```cpp
template <class T, class K>
class MinPriorityQueue
{
    public:
        MinPriorityQueue(); // constructor -- init a MinPriorityQueue
        ~MinPriorityQueue(); // destructor -- destroy a MinPriorityQueue
        K Minimum(); // return the key value of the minimum element
        T* ExtractMin(); // delete the minimum element and return its data
        void Insert(T* item, K key); // insert a new element
        void Print(); // print the contents of the priority queue
    private:
        int Left(int i) { return 2*i; } // return index of left child of i
        int Right(int i) { return 2*i+1; } // return index of right child of i
        int Parent(int i) { return (i/2); } // return index of parent of i
        void Swap(int i, int j); // swap two elements in the array
        void DecreaseKey(int i, K key); // decrease the key value of an element
        void Heapify(int i); // heapify the heap
        K *keys; // array of heap keys
        T **data; // array of pointers to the satellite data
        int heap_size; // current size of the heap

    // NOTE: function parameters pretend the elements are in 1..heap_size (like the book)
};
```
Notes:

- The generic class $T$ is the type of the (satellite) data contained in the priority queue and the generic class $K$ is the type of the key values. Notice that the priority queue contains a pair of “parallel” arrays, one that holds key values and one that holds the corresponding data.

- Notice that the $\text{Minimum}$ function returns a key value while $\text{ExtractMin}$ returns the data associated with the minimum element. I have done it this way for convenience so that these functions do not need to return 2 things (key and satellite data). These return types are typically the most useful for each of these functions. (You can do it another way if you wish.)

- The public function $\text{Print}()$ should print out the contents of the array in which your priority queue keys are stored. This function is not formally a part of the ADT; it is just used for testing.

- The private functions are used as auxiliary functions by the public functions.

Here is the general idea of the simulation in C++ pseudocode:

\begin{verbatim}
Simulation()
{
  clock = 0;
  insert customer arrival events into a min-priority queue eventQ;
  nextTime = eventQ.Minimum();
  while (nextTime >= 0)
  {
    clock = nextTime;
    event = eventQ.ExtractMin();
    if (event.type == arrival)
      {
        if (clerk not busy)
          service customer and insert departure event into eventQ;
        else
          insert customer into another priority queue representing the waiting line;
      }
    else if (event.type == departure)
      {
        delete event.customer;
        if (a customer is waiting in line)
          extract this customer, service it, and insert departure event into eventQ;
        else
          set clerk to not busy;
      }
    delete event;
    nextTime = eventQ.Minimum();
  }
}
\end{verbatim}

Notes:

- Your simulation will need to keep track of the total time customers wait in line so that the average wait time can be given at the end.

- Your program should accept 2 command-line parameters: a random seed and the number of customers. For example, 

  \texttt{simulation 123 100}

  should run your simulation with random seed 123 and 100 customers.

- Your simulation will actually use two independent priority queues. One is the event queue and the other is the virtual line in which customers wait.

- You will need to implement two small auxiliary classes to represent the data (events and customers) in the two priority queues.

- Print meaningful output when each event occurs in your simulation so that you can see what is happening.
Use your simulation to decide what the average wait time will be for 10, 100, and 1000 customers.

**EXTRA CREDIT:** Enhance your simulation so that it simulates a grocery store with multiple checkout clerks that service customers waiting in a single line. Add a third command line parameter for the number of clerks. How many clerks are needed so that 10, 100, and 1000 customers will never have to wait in line?

2. Add an implementation of quicksort to your collection of sort implementations. Plot its running time for a variety of input sizes and compare this to the running time of the other sorting algorithms you have written.

3. As mentioned in class, the **PARTITION** procedure used in the text is not the original partition algorithm used in quicksort. Here is the original quicksort, invented by C. A. R. Hoare in 1961:

   **HOARE-PARTITION**(A, p, r)
   
   \[x \leftarrow A[p]\]
   \[i \leftarrow p - 1\]
   \[j \leftarrow r + 1\]
   
   while true do
   
   \[do j \leftarrow j - 1\]
   
   \[while A[j] > x\]
   
   \[do i \leftarrow i + 1\]
   
   \[while A[i] < x\]
   
   if i < j then
   
   \[exchange(A[i], A[j])\]
   
   else
   
   return \(j\)
   
   **HOARE-QUICKSORT**(A, p, r)
   
   if \(p < r\) then
   
   \[q = \text{HOARE-PARTITION}(A, p, r)\]
   
   **HOARE-QUICKSORT**(A, p, q)
   
   **HOARE-QUICKSORT**(A, q + 1, r)

   Notice that the pivot value is now contained in the first subarray.

   (a) Prove that **HOARE-PARTITION** is correct by giving a careful proof of each of the following statements:
   
   i. The indices \(i\) and \(j\) are such that we never access an element of \(A\) outside the subarray \([p..r]\).
   
   ii. When **HOARE-PARTITION** terminates, it returns a value \(j\) such that \(p \leq j < r\).
   
   iii. Every element of \([p..j]\) is less than or equal to every element of \([j + 1..r]\) when **HOARE-PARTITION** terminates. (This is the crucial statement proving that the algorithm is correct.)

   (b) Implement this version of **QUICKSORT** and compare it to your other sorting algorithms. Explain why its running time differs from the Quicksort in your textbook.