Summary

We are studying the hypothetical "Big Long River," a 225 mile wilderness river accessible only at two points, called "First Launch" and "Final Exit." Each year, adventurers travel down the river on six to eighteen night trips, stopping at nights to camp. They can either travel by motorized boats, at 8 mph, or by oar-powered boats, at 4 mph. The problem is how to schedule an optimal number of trips down the river, while using the campsites most efficiently, since two boats cannot stop at the same campsite on the same night.

To clarify and simplify the problem, we made several assumptions. We know the season is six months, so we assumed that in the most likely case, the schedule would run from April 1st to September 30th, giving us a total of 183 days to schedule (or 182 nights). These months are generally warm enough in most regions for boating, and assuming them gave us a definite length for our schedule. We also assumed that campsites are equally distributed. In other words, if there are Y campsites, the distance between two campsites is $\frac{225}{Y+1}$ miles. Another assumption is that there are at least 18 campsites. More importantly, having very few campsites makes it difficult for six-night trips on oar boats to finish in time - if one particular campsite is already in use, they may not be able to travel as far as necessary to stay on schedule. We assume that boats do not travel "backwards" upstream, since that would simply not make sense, and we worked that in as a rule into our scheduling algorithm. Finally we assume that all trip requests are submitted well in advance of the trip's departure date. This is both reasonable and necessary for optimal scheduling, and it allows us to essentially view trip requests as an infinite queue, of which the algorithm satisfies as many requests as possible.

We considered several factors that go into an optimal schedule. The first is simply how many trips make it down the river by the end of the season - the more, the better. The second is the number of times boats pass each other. Since the trips are supposed to be wilderness experiences, we would like to minimize the number of these "crossovers." Finally, it is possible that while building a schedule, we determine that a boat will not be able to arrive at the end of the river as soon as scheduled, but instead is scheduled to arrive "late." If the trip was specifically scheduled for a customer with real time constraints, they may not accept such a change in plans, and the schedule will have to be reworked. Such reworking has a cost - either the schedule will have to undergo major rollback changes, or there will be a significant loss in efficiency.

The algorithm we developed for designing a schedule is priority-based. By associating a priority with each boat, the algorithm ensures that high priority boats choose campsites first so that they have every chance to meet their deadline, while low priority boats basically get whatever sites are left. The way we used to determine priority is to count how many days the boat has to finish its trip, and give priority to the boats with the least time left in their trip. If there is a tie in this scenario, oar boats have priority over motor boats. We also studied how far a boat should travel each day, and determined that we had the highest efficiency and least rescheduling when boats "paced themselves," rather than traveling as far as possible the first few days.

We tested our algorithm under many conditions, varying parameters including the number of campsites, the distribution of trip lengths and boat types, and the time spent traveling each day. We also determined how to calculate the theoretically optimal efficiency as a function of the number of sites: the maximum possible number of trips, M, can be found by $M = \frac{182Y}{\overline{L}}$, where Y is the number of sites, and $\overline{L}$ is the average number of nights per trip. We calculated this number for our best runs and compared our results against it. We found that our algorithm generally had very good efficiency, as high as 98.8% under ideal circumstances, and remained efficient for nearly all likely conditions. We found that our optimality peaked when we had certain "magic numbers" in our parameters. Specifically, the parameters that had magic numbers were the ones that went into the maximum distance equations: $D_O(Y) = \lfloor\frac{4h(Y+1)}{225}\rfloor$ and $D_M = \lfloor\frac{8h(Y+1)}{225}\rfloor$, where h is the maximum hours spent traveling per day, and Y is the number of sites. The "magic" combinations of h and Y occur when $D_O(Y) \neq D_O(Y-1)$ and $D_M(Y) = 2D_O(Y)$ When deciding what final parameters to use when making a schedule, these **must** be taken into account.

Our model is fairly insensitive to small changes in our assumptions. When we increased or decreased the number of days in the calendar, the number of trips increased linearly, which means that the efficiency remained constant. Also, changing the ratio of oar boats to motor boats only affected our efficiency by about 2.5%, which is relatively insignificant. Assuming either a uniform distribution or a Gaussian distribution in the desired length of trips did not affect our efficiency in any significant way either. In general, our model behaved well under many different parameter combinations.

The strengths of our model are that it is highly efficient and minimizes rescheduling of trips. Another strength is that our algorithm inherently reduces crossovers. This is true because **no boat will cross another that has fewer days remaining than itself**. One weakness in our model is that the boats are put in the water in an essentially random order (first come, first served) - we attempted to find a better way of ordering the boats, but all our attempts gave worse results than a random ordering. If we were to study this problem further, we might refine the model and determine why some orderings work better than others. The issue of rescheduling trips could also be improved; ideally our model could be optimized so that there would be no trips that needed to be rescheduled.

Memo

To whom it may concern,

We would like to inform you of the findings that we obtained from our research into the problem of creating a good schedule and determining the carrying capacity of the Big Long River. We have formulated an algorithm for you that will create a schedule for a given group of reservations and will allow for a maximum amount of trips to occur during the season. This algorithm will require that you have the reservations well in advance of when the trip will be scheduled to leave, in order for the algorithm to provide the best possible schedule that it can. This algorithm also provides the absolute minimum amount of interaction between the trips and provides for a true wilderness experience.

In regards to the problem of determining the carrying capacity of the river, we found that our algorithm almost always filled every campsite that was available and so the capacity ended up being Y, the number of campsites on the river. While we were working on the scheduling problem we found what we have come to refer to as magic numbers. These numbers are certain values of h, the maximum number of hours spent traveling each day, and Y, the total number of campsites, that will maximize the amount of trips in a season. When $\lfloor \frac{8h(Y+1)}{225} \rfloor \neq \lfloor \frac{8hY}{225} \rfloor$ and $\lfloor \frac{8h(Y+1)}{225} \rfloor = 2\lfloor \frac{4h(Y+1)}{225} \rfloor$ is true, then the values for h and Y are the magic values for maximizing trips. So to optimize your trips, use the formula and plug in your value of Y to find a reasonable value for h that is a magic number and run the algorithm using that value.

Getting our Priorities Straight

For the Purpose of Maximizing Efficiency on the Big Long River

Introduction to Problem

The problem that we chose was to create a schedule that would increase the number of trips down a river. Each trip consists of taking either an oar- powered rubber raft or a motorized boat down the 225 mile long river for 6-18 nights. Each of the two types of boats travels at a different speed; the oar rafts at 4 mph and the motor boats at 8 mph. We were also given that there are currently X trips and Y camp sites and that these trips are scheduled for a 6 month period. We were tasked with finding a way to maximize the amount of trips while still giving every boat a place to camp for the night, and at the same time minimizing the amount of contact that the boats have with one another.

Assumptions and Parameters

Assumptions and Justifications

In the problem we made some assumptions; they are as follows:

1. The length of the river is 225 miles. This was given to us as part of the core problem.

2. The first trip starts on April 1st and ends September 30th. This means that the length of the schedule is 183 days, or 182 nights. This is a reasonable schedule and it is consistent with a six month season. (One particular river with camping is open from May 1st to September 30th for a shorter five-month season).[1]

3. The campsites are equally distanced at every $\frac{225}{Y+1}$ miles. The problem breaks into a discrete algorithm that is much nicer when this is assumed, and if the number of sites is high enough, this approximation becomes very good.

4. $Y \geq 18$. Having less than eighteen sites on the river is unrealistic for a 225 mile river - it makes it very difficult for six-night oar boat trips to finish on time, and it also means that every eighteen-night trip will spend multiple nights at least one of the sites, which is certain to cause problems in any scheduling algorithm.

5. Boats cannot go backwards. Since the trips involve white-water rafting, it would be impractical to schedule any trip which would involve traveling upstream.

6. Requests are submitted well in advance of a departure date. In other words, we can assume that an entire trip can be planned before the boat leaves the "First Launch."

Parameters

The first step towards building our model was setting up some parameters. We knew that the number of sites, Y, could vary, making it the first parameter in our model. We knew how fast each boat could go, but we did not know how long each one could travel each day. We decided to leave this as a parameter, which we reasoned could vary between six and twelve hours per day.

Another issue we dealt with was how the boats were distributed. In our assumptions we said that requests are submitted well in advance of the departure date, and we organized those on a first come first served basis. However, we did not know what types of trips and trip lengths would be requested more often, so we initially assumed a completely uniform, albeit random, distribution of boats where each boat type and trip length had an equally likely chance of being requested. Later we modified the distribution and gave each type of boat a normally distributed trip length with different averages and variances. Another related issue was the ratio of oar boats to motor boats requested, which we varied across all possibilities.

Analysis of the Problem

We immediately identified two goals for our model. First, we wanted to maximize the number of complete trips that could take place during a single season of boating. Secondly, since the trips are supposed to be "wilderness experiences," without much contact with other people, we wanted to minimize the number of times a boat passes another boat on the river.

As we studied the problem further, we realized that there are two general ways that trips could be scheduled. One way is that the agency could "pre-schedule" all the trips, and groups would choose a trip that matched their preferences and schedule. The second way is that groups could submit a trip request, which would be scheduled as soon as possible in a first-come, first-served order. We will study this second scenario in the most detail, and then we will also show that our solution is highly efficient for the first scenario as well.

This led us to a third goal: if customers are submitting trip requests, and they must be fulfilled in first-come, first-served order, then while building the schedule, it is possible that we might not be able to fit a trip of a certain length into the schedule very easily at a certain time. In other words, the trip would end up more days than the group requested, which would force us to either reschedule the trip, or fail to meet the customer's request. Our third goal, then, is to minimize the number of these trips that have to be rescheduled.

Hypotheses

After considering our goals, we came up with a few different hypotheses for how to meet these goals. Our first problem dealt with how to decide who moves where in the river. From this issue arose the idea of priority. We hypothesized that if the boats who had the fewest days remaining chose where they camped for the night first, we would have the best chance of meeting our third goal, avoiding rescheduling. This stemmed from the idea that, in order for all the boats to come in on time, the boats who had the shortest time remaining would have to make up the most ground. Similarly, priority should also be given to the oar boats, since their limited speed makes it harder for them to catch up.

Linear thinking then tells us that we can assume that having less oar boats in the water will allow us to have more completed trips and less rescheduled trips.

The Model

We divided our model into two main components: the boats and the river. A boat in our model is synonymous with a trip; a boat consists of a method of transportation (motor or oar) and a desired number of nights for the trip. While in the river, a boat also keeps track both of the number of days remaining before the trip is scheduled to end, and the current campsite where it is resting.

The river is modeled as a ordered set of equally spaced campsites (each $\frac{225}{Y+1}$ miles apart, where Y is the number of campsites), which fits well with our assumptions. Based on this model, we can calculate the maximum number of sites a boat can advance in one day, S_k. If the maximum distance a boat can travel in one day is $v_k h$, where v_k is the boat's speed (8 mph for motor boats and 4 mph for oar boats) and h is the number of hours each boat spends on the water each day, then dividing the maximum distance in miles by the number of miles per campsite gives us the maximum number of campsites a boat can advance past in one day: $\Delta S_k \leq \lfloor \frac{v_k h(Y+1)}{225} \rfloor$.

We have chosen to use the Java programming language to implement our model. In our Java implementation, we maintain consistency with our model by breaking our code into a River class and a Boat class. We have included complete source code for both of these classes for reference in an Appendix.

Representation of a Schedule

We are representing a schedule, the goal of our model, as a table where the rows represent campsites and the columns represent nights. The entries of the table are the boats in the river. Low site numbers (indexed starting at zero) correspond to the beginning of the river, while high site numbers correspond to the end of the river. So if boat 19 is recorded in row 12 and column 22 of our schedule table, that means that boat 19 is resting in site 12 on night 22 of the boating season. In Java, this carries over as a two-dimensional array of Boat objects. This representation makes it easy for the scheduler to see at a glance where the boats are on a given day/night, how many sites are empty, and how far a given boat is from the end of the river.

It is important to note that in this representation, a column of the table represents a night, while an iteration of the scheduling algorithm represents a day. This is important because a 13-day trip consists of only 12 nights, and a trip with 18 nights actually has 19 days. We will generally use days and nights interchangeably when discussing our algorithm, but it is important when actually crunching the numbers to determine efficiency that we understand this distinction.

General Algorithm

We now define our main scheduling algorithm, which we have coded in java as a method called `generateSchedule()`. We will first show the code for this algorithm and then go into the details of how it works.

```java
public void generateSchedule()
{
    addBoats(0); // Add boats to the river for day one.

    // Start with the second day, and make the schedule day by day.
    for (int i = 1; i < NUM_DAYS; i++)
    {
        // Fill a priority queue with the boats already in the river on the previous day.
        PriorityQueue<Boat> boatsInRiver = new PriorityQueue<Boat>();
        for (int j = 0; j < numSites; j++)
        {
            if (schedule[i-1][j] != null)
            {
                boatsInRiver.add(schedule[i-1][j]);
            }
        }

        // Move the boats in order of priority
        while(!boatsInRiver.isEmpty())
        {
            // The next boat is on top in the priority queue.
            Boat nextBoat = boatsInRiver.poll();

            // Calculate the site the boat would "like" to go to.
            int nextSite = nextBoat.getPreferredNextSite(numSites);

            if (nextSite >= numSites && nextBoat.getDaysLeft() > 1)
            {
                // The boat shouldn't be leaving yet, but is trying to, so hold it back
                nextSite = numSites-1;
            }

            if (nextSite >= numSites)
            {
                nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                if (nextBoat.getDaysLeft() < 0)
                {
                    // Update stats for late boats
                    lateTrips++;
                    maxDaysLate = Math.max(maxDaysLate, -nextBoat.getDaysLeft());
                    totalDaysLate -= nextBoat.getDaysLeft();

                    // The boat arrived late, so we have to reschedule it.
                    reschedule(nextBoat);
                }
                else
                {
                    // Best case scenario: the boat is out of the river; add a point to our score
                    tripCount++;
                    boatsOut.add(nextBoat);
                }
            }
            else if (schedule[i][nextSite] == null)
            {
                // Nearly best case: there is an opening in the preferred site; move the boat there.
                nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                nextBoat.setCurrentSite(nextSite);
                schedule[i][nextSite] = nextBoat;
            }
            else
            {
                // Not best case: there is a boat in the site our boat would like to move to.

                // Keep track of whether we have found an alternate site.
                boolean found = false;

                // Check every site, moving up the river, until we get where the boat stopped last night.
                for (int j = nextSite - 1; !found && j >= nextBoat.getCurrentSite(); j--)
                {
                    if (schedule[i][j] == null)
                    {
                        // If we find an alternate site, we are in luck; the problem does not percolate.
                        nextBoat.setCurrentSite(j);
                        schedule[i][j] = nextBoat;
                        found = true;
                    }
                }

                // We are done moving this boat: either a site was found,
                // or the boat cannot move forward and is stuck.
                nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);

                if (!found)
                {
                    // Worst case scenario: A site was not found, so our boat and the boat currently in the
                    // spot our boat had yesterday swap roles, since our boat can't go backwards up the river.
                    Boat temp = nextBoat;
                    nextBoat = schedule[i][nextBoat.getCurrentSite()];
                    schedule[i][nextBoat.getCurrentSite()] = temp;
                }

                while (!found) // Repeat until the issue is resolved
                {
                    // Find the site where the boat we are dealing with now was yesterday.
                    int lastSite = -1;
                    for (int j = 0; j < numSites; j++)
                    {
                        if (schedule[i-1][j] == nextBoat)
                        {
                            lastSite = j;
                        }
                    }

                    // Try to back up this boat until it finds an empty site or
                    // a boat with lower priority whom it can swap roles with.
                    boolean swapped = false;
                    for (int j = nextBoat.getCurrentSite() - 1; !found && !swapped && j >= lastSite; j--)
                    {
                        if (schedule[i][j] == null)
                        {
                            // Empty site: we're done; set found to true so all the loops terminate.
                            nextBoat.setCurrentSite(j);
                            schedule[i][j] = nextBoat;
                            found = true;
                        }
                        else if (schedule[i][j].compareTo(nextBoat) > 0)
                        {
                            // Boat with lower priority: swap roles, but just break out of the
                            // inner loop since the lower priority boat still needs to find a site.
                            nextBoat.setCurrentSite(j);

                            Boat temp = nextBoat;
                            nextBoat = schedule[i][j];
                            schedule[i][j] = temp;
                            swapped = true;
                        }
                    }

                    if (!found && !swapped)
                    {
                        // No empty spots were found, and all boats have higher priority, so swap roles
                        // with the boat at the site our boat was at yesterday and continue with that boat.
                        nextBoat.setCurrentSite(lastSite);

                        Boat temp = nextBoat;
                        nextBoat = schedule[i][lastSite];
                        schedule[i][lastSite] = temp;
                    }
                }
            }
        }

```

```
143         // Now add more boats to the river
144         addBoats(i);
145     }
146
147     // Remove any boats from the schedule that didn't make it back by the last day of the season.
148     for (int i = 0; i < numSites; i++)
149     {
150         if (schedule[NUM_DAYS-1][i] != null)
151         {
152             reschedule(schedule[NUM_DAYS-1][i]);
153         }
154     }
155 }
```

To begin, we assume that we have a method called `addBoats()` which adds boats to the river to fill any empty spaces reachable from "First Launch," and takes one parameter, the day on which to add the boats. We will later propose a simple method by which this could be done, but there are many other ways in which this process could take place. We call this method for day zero (April 1st) on line 3 to populate the river with some boats.

Next we enter the main loop (line 6), which iterates from day 1 to 182, and make the schedule one day at a time. A priority queue is the data structure which will determine which boat's turn it is to "move." A priority queue, according to Weiss in *Data Structures and Problem Solving using Java*, supports three operations: `insert`, `findMin`, and `deleteMin`.[2] `insert` behaves as expected, inserting an object into the priority queue. `findMin` and `deleteMin` both find the item with highest priority in the queue, where `deleteMin` removes it from the queue, but `findMin` leaves it at the front of the queue. We do not need to worry about the details of how these methods would be implemented here, because Java provides an implementation of `PriorityQueue`, where the corresponding methods are `add()`, `peek()`, and `poll()`, respectively.[3]

To begin with, we populate the priority queue with all the boats that were in the river on the previous day of the schedule (lines 9 through 16). Next, an inner loop (line 19) iterates through all the boats in the order they come out of the priority queue. We assume that each boat has a certain site it would "prefer" to stop at, if there were no other boats in the river to compete with for the same site. We assume that the `Boat` class correspondingly implements a method called `getPreferredNextSite`, taking as a parameter the number of sites in the river. We call this method at line 25, and then, if it is not time for the boat to conclude its trip, we enforce in lines 27-31 that the boat must stop at the last site on the river (`numSites - 1`) until its trip is finished. For scenarios where the trips are pre-scheduled, so trip length is more or less arbitrary, lines 27 through 31 could be omitted.

At this point we break into several cases:

1. The best case (line 33) is that the trip is over, and the "site" the boat is trying to move to is past the end of the river. If the boat came out on time, we can add a point to our score by incrementing `tripCount`, and mark the trip as completed by adding the boat to `boatsOut`. However, if the boat was late (`nextBoat.getDaysLeft() < 0`), and the trip length is a hard deadline which must be met, then the trip was unsuccessful. After updating our statistics to record the late trip, we must find a way to fix our schedule so this late trip will not occur. We will discuss how this is done later, but for now, we assume that there is a method called `reschedule()`, which takes as a parameter the boat whose trip is to be rescheduled, and that after calling this method, the schedule will be in a legal state, with no late or early trips, so that the algorithm can procede. (Once again, if the trip length is fairly arbitrary, this line could be omitted).

2. The nearly best case (line 53) is that no boat currently occupies the site the current boat is trying to move to. Since this boat has priority over every other boat still in the priority queue, we can assume that this boat has the right to occupy this site. We update the boat to reflect another day passing and the new site where it is resting, and write it into the schedule for this site.

3. The third case (line 60) is that there is a boat with higher priority at the site where the current boat would like to rest, but there is an empty site between this site and the site where the current boat

rested the previous night, including the previous night's site. Lines 65 through 77 iterate over these sites to determine whether an open site exists, and if it does, the boat is assigned that site to rest.

4. The worst case is that there is a boat with higher priority at the site where the current boat would like to rest, and there are no sites between this site and the site where the boat rested the previous night, including the previous night's site, that are not occupied by a boat with higher priority. Since the boat cannot go backwards up the river, it must remain at the site where it rested the previous night, and the boat of higher priority who "wants" to rest there must find an alternate site. So now the lower priority boat searching for a site and the higher priority boat swap roles (lines 83-90), and the higher priority boat is searching for a site again. It can either find an open site to rest at, or find another lower priority boat to swap roles with and force the lower priority boat to find a new site. Once again, we do not want this boat to travel backwards, and since we have already lost track of where it was on the previous day, we must find this site and make sure it does not land upstream of it (lines 95-102). Once again, we iterate over all sites between the preferred site and the site the boat was at the night before (lines 107-127), and if the boat has not found a site, it is forced to swap roles with the boat at the last site (lines 129-138). This process can continue, with boats swapping roles with another boat of lower priority or a boat farther up river, until an open site is found. We are guaranteed that eventually there must be an open site, since no new boats have entered the river and at least one boat must have advanced a site to cause this problem, leaving an open site behind them.

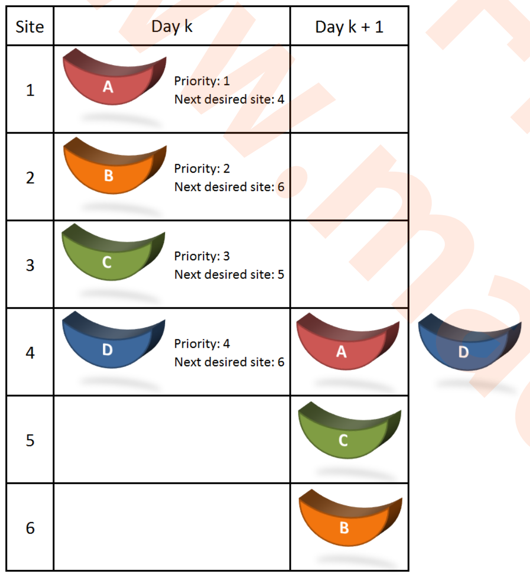

Figure 1: Case 4: Boat D cannot find a site to land.

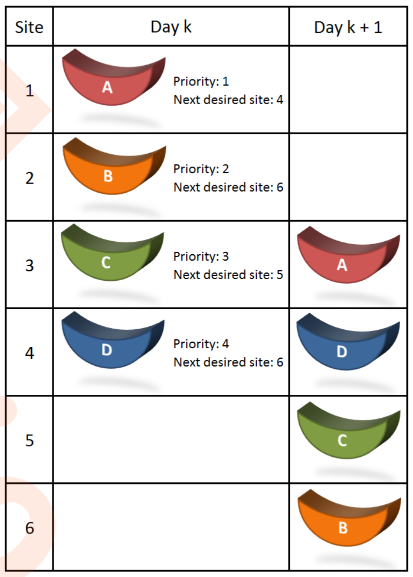

Figure 2: Resolution of Case 4: Boat A is forced to move back, even though it has higher priority.

Once all boats have found a new site to rest at, the algorithm calls `addBoats()` at line 144 to fill the top of the river with new boats. After iterating through the 183rd day, a complete schedule will have been created.

It is possible that this algorithm could have inadvertently allowed boats to go down the river that were not able to make it back by the end of the season. To avoid this, the program searches for such boats in lines 148-154 and calls `reschedule()` on them if they are found.

Calculating Movement

Our algorithm assumes we can calculate how far down the river a boat should travel under ideal conditions. Our initial attempt to find a formula for this was based on counting the minimum number of sites the boat would have to travel per day to exit on time. This simple movement formula for boat k is: $\Delta S_k = \lceil \frac{Y-S_k}{d_k} \rceil$, where S_k is the site where boat k is currently resting, Y is the number of sites, and d_k is the number of days left in boat k's trip. Of course, this formula must also be bounded by the maximum distance a boat can travel in a day: $\Delta S_k \leq \lfloor \frac{8h(Y+1)}{225} \rfloor$ for motor boats and $\Delta S_k \leq \lfloor \frac{4h(Y+1)}{225} \rfloor$ for oar boats, where h is the maximum number of hours traveled per day.

We eventually found that our movement formula can be improved by adding a constant m_C to keep the boats moving faster: $\Delta S_k = \lceil \frac{Y-S_k}{d_k} \rceil + m_C$. We will discuss how this constant can be selected to optimize our model in our Results section.

Adding Boats to the River

Our algorithm assumes there is a "waiting list" of people who would like to schedule a trip down the river. Our method of adding new boats to the river is very simple; boats are added on a "first-come, first-served" basis, by means of a queue (for which we use Java's `LinkedList`). The placement of these boats works the same as in the movement algorithm - the algorithm attempts to place them in their preferred site, but if it is unavailable, it checks upstream, iterating through each site one by one, until it finds an open site or reaches the top of the river. The method ends when it reaches the top of the river without placing a boat, because that means that there must not be an open site for the boat at the front of the queue. Our source code for `addBoats()` is shown below:

```java
private void addBoats(int day)
{
    boolean found = true;
    Boat nextBoat = null;
    while (!queue.isEmpty() && found) // Stop when a site is not found.
    {
        nextBoat = queue.poll();

        // Skip boats that can't get out by the end of the season.
        if (NUM_DAYS-day < nextBoat.getTotalDays()) continue;

        int nextSite = nextBoat.getPreferredNextSite(numSites);

        if (nextSite >= numSites)
        {
            // The boat is out of the river, add a point to our score
            // This shouldn't ever happen with our parameters, but it's here just in case they ever change.
            tripCount++;
            boatsOut.add(nextBoat);
        }
        else if (schedule[day][nextSite] == null)
        {
            // There is an open spot in the preferred site, move the boat there.
            nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
            nextBoat.setCurrentSite(nextSite);
            schedule[day][nextSite] = nextBoat;
        }
        else
        {
            // There is a boat in the spot our boat would like to move to.
            found = false; // Keep track of whether we have found an alternate spot.
            for (int j = nextSite - 1; !found && j >= 0; j--)
            {
                if (schedule[day][j] == null)
                {
                    // If we find an alternate spot, the boat can move there.
                    nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                    nextBoat.setCurrentSite(j);
                    schedule[day][j] = nextBoat;
                    found = true;
                }
            }
            // When a spot is not found, the found variable will be left as false, and the loop terminates.
        }
    }

    if (!found)
    {
        queue.push(nextBoat);
    }
}
```

Rescheduling

Our algorithm as it stands has a major deficiency: there is a significant probability that over the course of the year, several boats, despite our techniques of prioritizing, will be scheduled to arrive later than their intended arrival time. Other than referencing this issue in our general algorithm, we have not yet discussed how we are going to deal with this. The first method is that we could simply ignore the problem. This technique would be appropriate if all the trips are scheduled at the beginning of the year, and customers sign up for a trip that fits their personal schedule. In this case, we would just adjust the numbers and pretend that however long the trip took was how long we wanted it to be.

However, the river managers might decide instead to take requests throughout the year for individualized trips, in which case it would not be acceptable to schedule a boat to arrive later than requested. The simplest way to handle this is to just go through the schedule, and remove all references to the late boat, as if it never left the launch. Then the boat is reset and pushed onto the front of the queue to re-enter the river (Java's `LinkedList`is a double-ended queue, so this is allowed[3]) and we try again, hopefully with a better result. This is, in fact the solution we used in our implementation of `reschedule()`, as shown below in our source code:

```java
private void reschedule(Boat boat)
{
    // Remove all references to our boat in the schedule.
    for (int i = 0; i < NUM_DAYS; i++)
    {
        for (int j = 0; j < numSites; j++)
        {
            if (schedule[i][j] == boat)
            {
                schedule[i][j] = null;
            }
        }
    }

    // Reset the boat and put it at the front of the queue.
    boat.setDaysLeft(boat.getTotalDays());
    boat.setCurrentSite(-1);
    queue.push(boat);
}
```

An alternative approach, which we did not implement, would have been to roll back the entire schedule to before the boat entered the river, erasing several days of the schedule. This seemed an unnecessary amount of work considering the relatively low number of late arrivals, and we instead decided to focus on optimizing our model by reducing the number of rescheduled boats while increasing the total number of trips.

Pre-scheduling trips

We have generally been developing our algorithm as if we are required to meet the demands of the customer on a first-come, first-served basis. However, it may be that the manager does not want to allow customers to choose their own trips, but instead select from a schedule of predetermined trips. Under these conditions, it would in theory be easier to make an optimal schedule. However, in the time we spent attempting to develop a better algorithm for such a scenario, we were unable to get better results than those we got when simulating the random needs of customers. The solution we would therefore propose for such a situation would be to do what we have done, and generate random "requests" to build up a schedule of predetermined trips. To optimize further, this algorithm could be run thousands of times with the same parameters, until the best "random" schedule was generated (that is, the schedule with the most completed trips). This method should have very good results, and, as we will discuss later, will be very near the theoretically optimal efficiency for this problem.

Results

Optimization

When it comes to optimization, there is one important equation that we used that could be tweaked. This equation determines how far a boat tries to move on any given day: $\Delta S_k = \lceil \frac{Y - S_k}{d_k} \rceil + m_C$ where, for boat

k, $Y - S_k$ is the number of campsites remaining and d_k = the number of days left to end on schedule. The equation itself makes sure that boats move at an average pace to end their trip on time, but it rounds up to move boats out of the way of other boats so we can get more boats on the river sooner. The first time we used this equation it ran fairly well, but we noticed in the final schedule that the last few sites rarely get used. While trying to fix this we came up with a variety of solutions that prioritized filling the final few spots, but most of these were complicated and ended in failure. We found that the best solution was something much more simple. Adding a constant of one at the end of the equation increased the number of trips by about 10.7 % (from 557.91 average trips per year to 617.796) and reduced rescheduled trips from an average of .793 to about .166 per year. This makes sense because the boats are moving faster and we are able to get more through, and since they are moving faster they are less likely to be late.

Adding a coefficient of one helped, we hypothesized that adding a larger coefficient might get even better results. It turns out that adding a higher number will only improve efficiency up to a point, because as soon as this movement constant (which we call m_C) approaches infinity, it will exceed the maximum travel distance and thus the equation becomes less elegant. Also, as we increase m_C, the number of rescheduled trips goes up anywhere from 10 % to 150 %. In order to minimize this some tests were run while changing the number of total campsites, and for the most efficient results, it seems like we want m_C = 8-15 % of the maximum travel distance, with m_C being lower in the range for a lower number of sites, and higher in the higher number of sites. By this method, the number of trips is optimized, while the number of rescheduled ones is reduced.

Magic Numbers

Of all the discoveries we made, "magic numbers" were perhaps the strangest and most exciting. We were testing how the number of sites affected the number of trips and rescheduling when we noticed something interesting. Here is the graph relating the number of sites compared to average number of late trips:

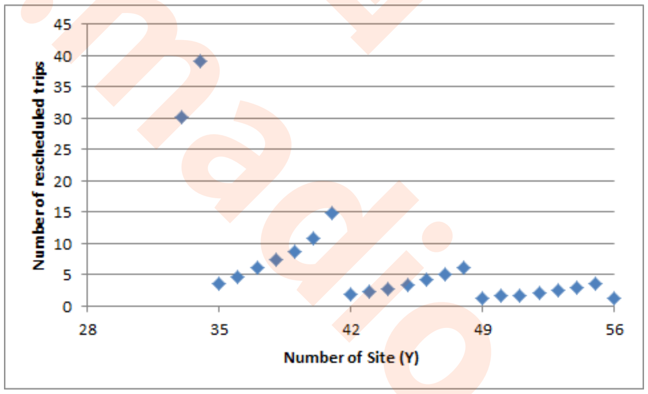

Figure 3: Graph of Magic Numbers

At first it looked like jumbled data, but that was hardly the case. There were **multiple** lines that seemed periodic with a period of 7. This phenomenon was so incredible that we labeled these numbers "magic numbers." Looking into it a little deeper, we noticed that it had a relation to our maximum movement equation. Plugging in the numbers to the equation with motor boat that traveled 8 hours maximum a day we noticed that these multiples of 7 where not mere happenstance, but mathematically efficient. Here are the results for 34 campsites, a poor case, and 35 campsites, a magic number: for $Y = 34$, $\lfloor\frac{4(2)(8)(34+1)}{225}\rfloor = \lfloor 9.955\rfloor = 9$, while for $Y = 35$, $\lfloor\frac{4(2)(8)(35+1)}{225}\rfloor = \lfloor 10.24\rfloor = 10$. We checked this for all magic numbers, but there ended up being a problem. The magic numbers were only hit when the maximum distance traveled by a motor boat was an even integer, not an odd. We discovered that this is because oar boats travel half as fast, and do not switch over on the odd integers of the motor boats. For example when $\lfloor\frac{4(2)(8)(38+1)}{225}\rfloor = \lfloor 11.093\rfloor = 11$ and $\lfloor\frac{4(2)(8)(38)}{225}\rfloor = \lfloor 10.809\rfloor = 10$, then 38 should be a magic number, but $\lfloor\frac{4(8)(38+1)}{225}\rfloor = \lfloor 5.547\rfloor = 5$ and $\lfloor\frac{4(2)(8)(38)}{225}\rfloor = \lfloor 5.404\rfloor = 5$ so it ends up not being one.

Later we tried a more optimal speed, 80 miles per day for motor boats and 40 miles per day for oar boats, and came up with magic numbers that happened to be multiples of 11. This held true for a while, but when the number of sites got high enough, not only did the efficiency barely change when we hit a magic number, but the campsites eventually got **off** the 11 pattern. Magic numbers lost their pattern of 11, but the idea still remained. Using this idea, we decided to test the rest of our results with 44 campsites. But another idea came up. When varying the maximum travel time per day, we predicted that 8.8 hours would be very efficient, while 8.7 hours would not. When we tested this, we lost 33% of the total trips we had. While 8.8 hours was a magic number, 8.7 hours we called a "black magic number." Also, we tested the length of the river just to see if this model would work for other rivers and found some more black magic numbers. At the normal length of 225 miles increased to about 257 miles, the number of trips stayed somewhat consistent, but as soon as we changed it to 258 miles we again lost 33% of our total trips. At that point the river campsites were too sparsely populated that the boats would barely miss the next site and would not be able to make it to the end with the maximum time constraints.

There is a formula for magic numbers and it is as follows: When $\lfloor \frac{8h(Y+1)}{225} \rfloor \neq \lfloor \frac{8hY}{225} \rfloor$ and $\lfloor \frac{8h(Y+1)}{225} \rfloor = 2\lfloor \frac{4h(Y+1)}{225} \rfloor$ is true, then h and Y are magic numbers. When deciding for other rivers how many campsites to have and how long to restrain the boats, magic numbers are a **necessary** parameter to consider.

Efficiency

Calculating the total number of trips in a season is not enough in and of itself to determine whether our model is efficient. For example, increasing the length of the season or the number of campsites should always improve the number of trips, but that does not mean that the efficiency has improved. Instead, how we will accurately measure efficiency is how effectively the campsites are used. To do this we go back to our table representation of a schedule. The schedule has as many rows as there are campsites, and as many columns as there are nights in the season. If the schedule were entirely full, the campsites would be used at maximum efficiency. For a 183-day season, the number associated with this maximum efficiency is $182Y$ where Y is the number of campsites (since there are 182 nights for 183 days). Then, to determine the actual efficiency of a schedule, we need to estimate the number of entries in the schedule table that are used. If we are randomly generating trip requests, we should know what the average length of a trip will be, which we will call $\overline{L}$ (once again, in this case we mean the number of nights, not the number of days). Since a trip of length L uses L entries in the table, on average, n trips of average length $\overline{L}$ will take up $n\overline{L}$ entries in the table. Therefore to convert number of trips to percent efficiency, we calculate: % efficiency $= \frac{n\overline{L}}{182Y} \times 100\%$. Alternatively, we can say that the maximum number of trips, M, of average length $\overline{L}$, with Y campsites, in a single season, is determined by $M = \frac{182Y}{\overline{L}}$.

Here are the parameters that gave us the best efficiency (using a schedule of 183 days and a 225-mile river):

1. The maximum distance a boat can travel per day is 80 miles for motor, and 40 miles for oar.
2. Set $m_C = 4$.
3. Oar boats and motor boats are equally represented.
4. Uniform distribution in trip length.
5. 44 sites (the standard of most of our testing).

With these parameters, we were running 659.371 (an average taken over 10 data points of 1,000 iterations) trips a year, which is about 98.8% efficiency.

Error/Sensitivity Analysis

We have just described our best possible outcome, and luckily our model is not too sensitive to things we cannot control. We will go in the order of the list above. The maximum distance a boat could travel related to the number of trips we could get across in the year was like a step function. Once it hit the magic number of 8.8 hours maximum travel per day, there was a little change until we increased it to above ten hours, but once we got below 8.8 our efficiency decreased from 630.515 trips to 439.586, about 30.28%. Fortunately we have an hour to spare before that happens, and even if we were to get down to 8.8 hours, it would only be about a 4% loss in efficiency. Increasing the time limit from 10 to 12 hours would bring us up to a few

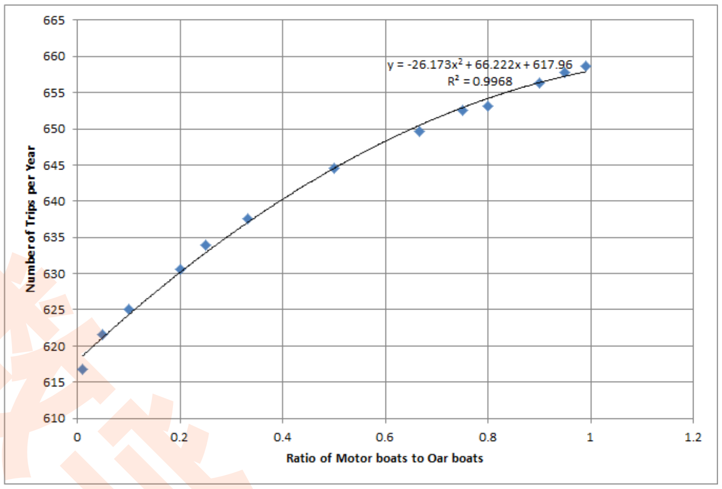

Figure 4: The ratio changes, and efficiency increases

more trips (because 44 is a magic number for 12 hours too) but, not enough to warrant the extra stress for the rowers and motors. Rowing more than 12 hours is not realistic, since in some parts of the country, the amount of daylight drops below 12 hours by the end of September.[4]

The next thing we checked was to see how affecting the ratio of motor boats to oar boats would change our efficiency. As can be seen in Figure 4, the efficiency **does** increase. However, it only changes by a few trips, and while our random number generator is good and we are running the program over 1,000 times for each data point, our estimates are not perfect and some higher results are more likely to be inaccurate. Because of this fact, we cannot trust the higher ratio of the data for accurate results. There does seem to be a trend however, and it makes sense that if we have more boats who cannot travel as far the efficiency would decrease and vice versa. In either case it is less than 4% decrease and a 2.3% increase.

The third thing we checked was how a uniform distribution of requests would compare with a Gaussian distribution. We said motors were normally distributed with a mean of 12 nights and standard deviation of 3 days, and that oars were normally distributed with a mean of 13 nights and standard deviation of 2 days. This is the conclusion we came to because people typically spend less time traveling on a motor boat, but could spend more time easily if they wanted to since they go at such a leisurely pace. Oars had a higher average and smaller standard deviation because it was likely that people would not go on the river to row as fast as they could for the 6 nights they were there. An actual rafting company's web site confirmed our reasoning that oar trips would tend to be longer than motor trips.[5] It was also less likely that the oar boats would want to spend all 18 nights there, but it was more likely than 6. In any case, efficiency was a tad bit lower; we lost about 27 trips for about a 4.1% loss in efficiency from when both oars and motors were equally likely to be chosen. Our biggest worry here was that we would be off on our estimations of the normal distributions, so we varied the averages and standard deviations. To our surprise, lowering the average of both oars and motors increased the number of trips by the same rate (about 23 more trips per day changed). The efficiency stayed the same however. The same happened when varying the standard deviations. The bigger the standard deviations, the more the Gaussian distributions looked like the uniform distribution and was close to reaching the efficiency of the top model. A note on efficiency: since these are normally distributed, we had to use "cut-offs" that would not allow the distribution to give us numbers outside of 6 and 18 nights. From that, our efficiency calculation would change, so we cannot get an accurate estimate of efficiency. But we know that the trips' average lengths were similar to that of the uniform distribution, but there were 30 less trips than the uniform, so it was definitely **not** as efficient as the uniform distribution. In conclusion, the results were not more efficient, and if we changed the average or standard deviation of our model, efficiency would more or less stay constant.

A few more parameters that were checked to see if this model could be implemented elsewhere are spoken of here. We decreased the number of days we would be open to about 25, and then modeled what would happen if we went year round. The results were excellent. There was a near perfect linear correlation between how many days we are open to how many trips we could get out. We gain about 3.5 trips per day we are open. Testing varying lengths of the river was also important. As river size decreases, the more total trips we can get with less and less having to be rescheduled. However, if we increase river size, the "black magic number" phenomenon happens, and to avoid it we would have to increase the number of sites or hours traveled. For our particular river, we could range anywhere from 257 mile long to 25 miles and we would only get a difference of about 11 trips, equating to a maximum of a 1.7% difference.

As far as sites go, we consistently tested around 44 because it was a magic number for multiple speeds. However, we tested down to 18 and still got wonderful results. Lowering or raising the number of sites linearly increases the number of trips, but **efficiency is not affected**. That means our model, so long as the boats can make the spacing, will work for any situation, as long as we can change the site count. Obviously we want to have as many sites as possible, but we want them to be at a magic number because that is where the rescheduling is put at a minimum. For this situation, going down to 18 sites does not affect efficiency because it does not go down enough for black magic numbers to really affect the data.

Analysis of our Model

Strengths

The biggest strength that our model has is its high efficiency of 98.8%. This is very good since the best possible efficiency is around 99%, due to the fact that no boat can reach the end of the river on the first day, leaving some unused campsites on the first few and last few days. Our model also gives the smallest amount of crossovers for a randomly selected group of trips. The reason that this is the case is because our priority scheduling prevents a boat from overtaking another unless the boat has a lower number of days left on its journey. To maintain a wide variety of trip lengths, this situation is inevitable.

Weaknesses

One of the weaknesses that our model has is that the boats are not put in the water in any particular order, but instead people who call in for reservations are served on a first-come, first-served basis. We tried to find a better technique to schedule boats, but our efforts ended up not getting as many people through as when they were randomly scheduled. Another weakness of our model is the fact that once in a while a trip has to be rescheduled. Ideally this would not be the case, but our algorithm does have to reschedule a trip about once every season.

Future Improvements

A future improvement that we could implement would be to schedule trips so that they are not on a first-come, first-served basis. This would be a different way in which someone could schedule a trip. Instead of calling in and asking for a specific date and trip length, there would be preset trips that would then be filled. By scheduling trips in this manner we might be able to reduce the number of boat crossovers or be able to improve our efficiency.

Currently in our algorithm, if a trip fails to reach the end by the desired time it is rescheduled for a different date. This is not the best way to do this because then a group has to find another time that works for them. Instead, another improvement we could make would be a rollback system. This would mean that if a trip ends and needs to be rescheduled, instead of rescheduling the trip, the algorithm rolls back to where the boat entered and starts over from that point. This would mean that every trip would leave and arrive when it is scheduled and no one would not have to reschedule their trips.

Conclusion

When scheduling a trip down the Big Long River, or any similar river for that matter, remembering a few simple rules will bring about the greatest efficiency. Boats which have the fewest days remaining to complete their trip should be given first priority in site location and those who cannot move as far should also be given priority in the case of a tie. How far boats should move is dependent on how many days they have left to complete the trip, how many sites are remaining, and partially on how many total sites there are. For maximum efficiency we need to make sure that we hit so-called "magic numbers" with our travel time, number of campsites, and length of the river - travel time being the most easily changed of the three. Following these conditions will create the most efficient boating/camping schedule that allows people to choose what length of experience they want on a first-come, first-served basis. With this model we got a 98.8% efficiency of utilizing campsites, while still having a low number of rescheduled trips. Tweaking our assumed conditions here and there seemed to vary minimally from the optimal results, and so we know that this model could be used under a multitude of conditions. Overall, we are quite proud of this model.

References

[1] State of Connecticut Department of Energy and Environmental Protection. *River Camping.* http://www.ct.gov/dep/cwp/view.asp?A=2716&Q=325048

[2] Weiss, Mark Allen. *Data Structures & Problem Solving Using Java.* Boston: Pearson Education.

[3] *Java Platform, Standard Edition 6, API Specification.* http://docs.oracle.com/javase/6/docs/api/

[4] NOAA Earth Science Research Laboratory, Earth Sciences Division. *Boulder Colorado: Sunrise/Sunset data and length of twilight.* http://www.esrl.noaa.gov/psd/boulder/boulder.sunset.html

[5] Arizona Raft Adventures. *Most Frequently Asked Questions.* http://www.azraft.com/gc_faqs.cfm

Appendix: Source Code

Boat.java

```java
/**
 * A class to model a Boat (or equivalently, a single trip down the river).
 */
public class Boat implements Comparable<Boat>
{
    /**
     * The type of boat (motor or oar).
     */
    public enum Type { Motor, Oar }

    /**
     * A constant defining how far a motor boat can travel in one day.
     */
    public static final double MOTOR_SPEED = 80;

    /**
     * A constant defining how far an oar boat can travel in one day.
     */
    public static final double OAR_SPEED = 40;

    /**
     * The constant used to tweak the movement formula.
     * The higher this constant, the farther boats move each day.
     */
    public static final int MOVEMENT_CONSTANT = 4;

    /**
     * The type of boat (motor or oar).
     */
    private Type type;

    /**
     * The name of the boat (used when printing tables).
     */
    private String name;

    /**
     * The total number of days in the trip.
     */
    private int totalDays;

    /**
     * The number of days remaining in the trip.
     */
    private int daysLeft;

    /**
     * The current camp site where the boat is resting.
     */
    private int currentSite;

    /**
     * Initializes the boat.
     * @param name The name of the boat (used when printing tables).
     * @param t The type of boat (Motor or Oar).
     * @param days The number of days in the trip.
     */
    public Boat(String name, Type t, int days)
    {
        this.name = name;
        type = t;
        totalDays = daysLeft = days;
        setCurrentSite(-1);
    }

    /**
     * Gets the type of the boat.
     * @return The type of boat (Motor or Oar).
     */
    public Type getType()
    {
        return type;
    }

    /**
```

```
     * Sets the type of the boat.
     * @param type The type of boat (Motor or Oar).
     */
    public void setType(Type type)
    {
        this.type = type;
    }

    /**
     * Gets the name of the boat.
     * @return The name of the boat (used when printing tables).
     */
    public String getName()
    {
        return name;
    }

    /**
     * Sets the name of the boat.
     * @param name The name of the boat (used when printing tables).
     */
    public void setName(String name)
    {
        this.name = name;
    }

    /**
     * Gets the total number of days in the trip.
     * @return The number of days in the trip.
     */
    public int getTotalDays()
    {
        return totalDays;
    }

    /**
     * Sets the total number of days in the trip.
     * @param totalDays The number of days in the trip.
     */
    public void setTotalDays(int totalDays)
    {
        this.totalDays = totalDays;
    }

    /**
     * Gets the number of days left.
     * @return The number of days remaining in the trip.
     */
    public int getDaysLeft()
    {
        return daysLeft;
    }

    /**
     * Sets the number of days left.
     * @param daysLeft The number of days remaining in the trip.
     */
    public void setDaysLeft(int daysLeft)
    {
        this.daysLeft = daysLeft;
    }

    /**
     * Gets the current site.
     * @return The current camp site where the boat is resting.
     */
    public int getCurrentSite()
    {
        return currentSite;
    }

    /**
     * Sets the current site.
     * @param currentSite The current camp site where the boat is resting.
     */
    public void setCurrentSite(int currentSite)
    {
        this.currentSite = currentSite;
    }

    /**
```

```
     * Returns the next site this boat would "prefer" to stop at.
     * @param numSites The number of sites in the river.
     * @return The index of the next site this boat will try to stop at.
     */
    public int getPreferredNextSite(int numSites)
    {
        return currentSite + (int)Math.min(
                Math.ceil((double)(numSites-currentSite)/Math.max(daysLeft, 0.0)) + 4,
                this.getMaxSitesPerDay(numSites));
    }

    /**
     * Gets the maximum number of sites the boat can travel each day.
     * @param numSites The number of sites in the river.
     * @return The maximum number of camp sites the boat can advance per day.
     */
    public int getMaxSitesPerDay(int numSites)
    {
        return (int)Math.floor(this.getSpeed()*(double)(numSites+1)/River.RIVER_LENGTH);
    }

    /**
     * Gets the maximum speed of the boat
     * @return Either MOTOR_SPEED or OAR_SPEED, depending on the type of boat.
     */
    public double getSpeed()
    {
        switch(type)
        {
        case Motor: return MOTOR_SPEED;
        case Oar: return OAR_SPEED;
        default: return 0.0;
        }
    }

    @Override
    public int compareTo(Boat boat)
    {
        // Compare by the number of days left in the trip, then by the type of boat.
        if (this.daysLeft == boat.daysLeft)
        {
            if (this.type == Type.Motor && boat.type == Type.Oar) return -1;
            else if (boat.type == Type.Motor && this.type == Type.Oar) return 1;
            else return 0;
        }
        else return this.daysLeft - boat.daysLeft;
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

River.java

```
import java.util.*;

/**
 * A class to model the river and create a schedule to manage the river's camp sites.
 */
public class River
{
    /**
     * A constant to define the number of days in the boating season.
     */
    public static final int NUM_DAYS = 183;

    /**
     * A constant to define the length of the river.
     */
    public static final double RIVER_LENGTH = 225.0;

    /**
     * The camp site schedule
     * The first index is the day of the year (from 0 to 182), the second index is the camp site number.
     */
    private Boat[][] schedule;

    /**
     * The queue of boats waiting to enter the river.
     */
    private LinkedList<Boat> queue;

    /**
     * The number of camp sites on the river.
     */
    private int numSites;

    /**
     * A list of all completed trips.
     */
    private LinkedList<Boat> boatsOut;

    /**
     * The number of completed trips.
     */
    private int tripCount;

    /**
     * The number of rescheduled trips.
     */
    private int lateTrips;

    /**
     * The maximum number of days late a trip would have been if it were not rescheduled.
     */
    private int maxDaysLate;

    /**
     * The total number of days late all trips would have been combined if no trips were rescheduled.
     */
    private int totalDaysLate;

    /**
     * Initialize the river.
     * @param numSites The number of sites on the river.
     * @param requests A collection of trip requests.
     */
    public River(int numSites, Collection<Boat> requests)
    {
        this.numSites = numSites;
        schedule = new Boat[NUM_DAYS][numSites];
        queue = new LinkedList<Boat>();
        queue.addAll(requests);
        tripCount = 0;

        boatsOut = new LinkedList<Boat>();
    }

    /**
     * A subroutine that adds boats to a certain day of the schedule, after all other boats have moved.
     * In this algorithm, first-come is first-served, the number of days requested and the method of propulsion
     * will not affect when a boat is added to the river.
     * This algorithm moves boats according to the same formula used to move them after they are in the river.
     * @param day The day of the schedule to add boats to.
     */
    private void addBoats(int day)
    {
        boolean found = true;
        Boat nextBoat = null;
        while (!queue.isEmpty() && found) // Stop when a site is not found.
        {
            nextBoat = queue.poll();

            // Skip boats that can't get out by the end of the season.
            if (NUM_DAYS-day < nextBoat.getTotalDays()) continue;

            int nextSite = nextBoat.getPreferredNextSite(numSites);

            if (nextSite >= numSites)
            {
                // The boat is out of the river, add a point to our score
                // This shouldn't ever happen with our parameters, but it's here just in case they ever change.
                tripCount++;
                boatsOut.add(nextBoat);
            }
            else if (schedule[day][nextSite] == null)
            {
                // There is an open spot in the preferred site, move the boat there.
                nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                nextBoat.setCurrentSite(nextSite);
                schedule[day][nextSite] = nextBoat;
            }
            else
            {
                // There is a boat in the spot our boat would like to move to.
                found = false; // Keep track of whether we have found an alternate spot.
                for (int j = nextSite - 1; !found && j >= 0; j--)
                {
                    if (schedule[day][j] == null)
                    {
                        // If we find an alternate spot, the boat can mover there.
                        nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                        nextBoat.setCurrentSite(j);
                        schedule[day][j] = nextBoat;
                        found = true;
                    }
                }
                // When a spot is not found, the found variable will be left as false, and the loop terminates.
            }
        }

        if (!found)
        {
            queue.push(nextBoat);
        }
    }

    /**
     * Removes all evidence of a boat's itinerary down the river from the schedule,
     * and puts the boat back at the front of the queue.
     * @param boat The boat to remove from the schedule
     */
    private void reschedule(Boat boat)
    {
        // Remove all references to our boat in the schedule.
        for (int i = 0; i < NUM_DAYS; i++)
        {
             for (int j = 0; j < numSites; j++)
             {
                 if (schedule[i][j] == boat)
                 {
                     schedule[i][j] = null;
                 }
             }
        }

        // Reset the boat and put it at the front of the queue.
        boat.setDaysLeft(boat.getTotalDays());
        boat.setCurrentSite(-1);
        queue.push(boat);
    }

    /**
     * Generates an entire schedule for a season of boating.
     */
    public void generateSchedule()
    {
        addBoats(0); // Add boats to the river for day one.
        for (int i = 1; i < NUM_DAYS; i++) // Start with the second day, and make the schedule day by day.
        {
            // Fill a priority queue with the boats already in the river on the previous day.
            PriorityQueue<Boat> boatsInRiver = new PriorityQueue<Boat>();
            for (int j = 0; j < numSites; j++)
            {
                if (schedule[i-1][j] != null)
                {
                    boatsInRiver.add(schedule[i-1][j]);
                }
            }

            // Move the boats in order of priority
            while(!boatsInRiver.isEmpty())
            {
                Boat nextBoat = boatsInRiver.poll(); // The next boat is on top in the priority queue.

                // Calculate the site the boat would "like" to go to.
                int nextSite = nextBoat.getPreferredNextSite(numSites);

                if (nextSite >= numSites && nextBoat.getDaysLeft() > 1)
                {
                    // The boat shouldn't be leaving yet, but is trying to, so hold it back
                    nextSite = numSites-1;
                }

                if (nextSite >= numSites)
                {
                    nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                    if (nextBoat.getDaysLeft() < 0)
                    {
                        // Update stats for late boats
                        lateTrips++;
                        maxDaysLate = Math.max(maxDaysLate, -nextBoat.getDaysLeft());
                        totalDaysLate -= nextBoat.getDaysLeft();

                        // The boat arrived late, so we have to reschedule it.
                        reschedule(nextBoat);
                    }
                    else
                    {
                        // Best case scenario: the boat is out of the river; add a point to our score
                        tripCount++;
                        boatsOut.add(nextBoat);
                    }
                }
                else if (schedule[i][nextSite] == null)
                {
                    // Nearly best case: there is an opening in the preferred site, move the boat there.
                    nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);
                    nextBoat.setCurrentSite(nextSite);
                    schedule[i][nextSite] = nextBoat;
                }
                else
                {
                    // Not best case: there is a boat in the site our boat would like to move to.
                    boolean found = false; // Keep track of whether we have found an alternate site.
                    // Check every site, moving up the river, until we get where the boat stopped last night.
                    for (int j = nextSite - 1; !found && j >= nextBoat.getCurrentSite(); j--)
                    {
                        if (schedule[i][j] == null)
                        {
                            // If we find an alternate spot, we are in luck, the problem does not percolate.
                            nextBoat.setCurrentSite(j);
                            schedule[i][j] = nextBoat;
                            found = true;
                        }
                    }

                    // We are done moving this boat: either a site was found,
                    // or the boat cannot move forward and is stuck.
                    nextBoat.setDaysLeft(nextBoat.getDaysLeft() - 1);

                    if (!found)
                    {
                        // Worst case scenario:
                        // A site was not found, so our boat and the boat currently in the spot our boat had
                        // yesterday swap roles, since our boat can't go backwards up the river.
                        Boat temp = nextBoat;
                        nextBoat = schedule[i][nextBoat.getCurrentSite()];
                        schedule[i][nextBoat.getCurrentSite()] = temp;
                    }

                    while (!found) // Repeat until the issue is resolved
                    {
                        // Find the site where the boat we are dealing with now was yesterday.
                        int lastSite = -1;
                        for (int j = 0; j < numSites; j++)
                        {
                            if (schedule[i-1][j] == nextBoat)
                            {
                                lastSite = j;
                            }
                        }

                        // Try to back up this boat until it finds an empty site or a boat with lower priority
                        // who it can swap roles with.
                        boolean swapped = false;
                        for (int j = nextBoat.getCurrentSite() - 1; !found && !swapped && j >= lastSite; j--)
                        {
```

```
                        if (schedule[i][j] == null)
                        {
                            // Empty site: we're done; set found to true so all the loops terminate.
                            nextBoat.setCurrentSite(j);
                            schedule[i][j] = nextBoat;
                            found = true;
                        }
                        else if (schedule[i][j].compareTo(nextBoat) > 0)
                        {
                            // Boat with lower priority: swap roles, but just break out of the inner loop
                            // since the lower priority boat still needs to find a site.
                            nextBoat.setCurrentSite(j);

                            Boat temp = nextBoat;
                            nextBoat = schedule[i][j];
                            schedule[i][j] = temp;
                            swapped = true;
                        }
                    }

                    if (!found && !swapped)
                    {
                        // No empty sites were found, and all boats have higher priority,
                        // so swap roles with the boat at the site our boat was yesterday
                        // and continue with that boat.
                        nextBoat.setCurrentSite(lastSite);

                        Boat temp = nextBoat;
                        nextBoat = schedule[i][lastSite];
                        schedule[i][lastSite] = temp;
                    }
                }
            }
        }

        // Now add more boats to the river
        addBoats(i);
    }

    // Remove any boats from the schedule that didn't make it back by the last day of the season.
    for (int i = 0; i < numSites; i++)
    {
        if (schedule[NUM_DAYS-1][i] != null)
        {
            reschedule(schedule[NUM_DAYS-1][i]);
        }
    }
}

/**
 * An algorithm to check that the schedule does not violate any rules of our model.
 */
public void checkSchedule()
{
    int lateBoats = 0, earlyBoats = 0, speedingBoats = 0, totalBoats = 0;
    for (Boat boat : boatsOut)
    {
        int firstDay = -1, site, lastSite = -1;
        for (int day = 0; day < NUM_DAYS; day++)
        {
            for (site = 0; site < numSites; site++)
            {
                if (schedule[day][site] == boat) break;
            }
            if (site != numSites)
            {
                if (firstDay == -1)
                {
                    firstDay = day;
                    if (site+1 > boat.getMaxSitesPerDay(numSites))
                    {
                        speedingBoats++;
                    }
                }
                else
                {
                    if (site-lastSite > boat.getMaxSitesPerDay(numSites))
                    {
                        speedingBoats++;
                    }
                }
```

```java
                }
                else
                {
                    if (firstDay != -1)
                    {
                        if (day - firstDay + 1 > boat.getTotalDays())
                        {
                            lateBoats++;
                        }
                        else if (day - firstDay + 1 < boat.getTotalDays())
                        {
                            earlyBoats++;
                        }
                        totalBoats++;
                        break;
                    }
                }
                lastSite = site;
            }
        }
        System.out.println("There␣were␣" + totalBoats + "␣boats␣total.");
        System.out.println(lateBoats + "␣were␣late.");
        System.out.println(earlyBoats + "␣were␣early.");
        System.out.println(speedingBoats + "␣were␣caught␣speeding.");
    }

    /**
     * A subroutine to count the number of crossovers in the schedule.
     * @return The number of crossovers.
     */
    public int countCrossovers()
    {
        int crossovers = 0;
        for (int i = 0; i < NUM_DAYS-1; i++)
        {
            for (int j = 0; j < numSites-1; j++)
            {
                if (schedule[i][j] != null)
                {
                    for (int k = j+1; k < numSites; k++)
                    {
                        if (schedule[i][k] != null)
                        {
                            for (int m = 0; m < numSites; m++)
                            {
                                if (schedule[i+1][m] == schedule[i][k])
                                {
                                    crossovers++;
                                    break;
                                }
                                else if (schedule[i+1][m] == schedule[i][j])
                                {
                                    break;
                                }
                            }
                        }
                    }

                    int k;
                    for (k = 0; k < numSites && schedule[i+1][k] != schedule[i][j]; k++);

                    for (k++; k < numSites; k++)
                    {
                        if (schedule[i+1][k] != null)
                        {
                            crossovers++;
                            for (int m = 0; m < numSites; m++)
                            {
                                if (schedule[i][m] == schedule[i+1][k])
                                {
                                    crossovers--;
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
        return crossovers;
    }

    /**
     * Prints a summary of the scheduling algorithm,
     * indicating how close the schedule came to meeting our three goals.
     */
    public void printSummary()
    {
        System.out.println("There␣were␣" + tripCount + "␣trips.");
        System.out.println("There␣were␣" + this.countCrossovers() + "␣crossovers.");
        System.out.println(lateTrips + "␣trips␣were␣rescheduled.");
    }

    /**
     * Prints out the schedule in ASCII form.
     */
    public void printTable()
    {
        System.out.print("␣␣␣");
        for (int i = 0; i < NUM_DAYS; i++)
        {
            System.out.printf("|%4d␣", i);
        }
        System.out.println();
        for (int i = 0; i < numSites; i++)
        {
            System.out.printf("%3d", i);
            for (int j = 0; j < NUM_DAYS; j++)
            {
                if (schedule[j][i] == null)
                    System.out.print("|␣---␣");
                else System.out.print("|"+schedule[j][i]);
            }
            System.out.println();
        }
    }

    /**
     * Tests the program with a uniform distribution in trip lengths.
     * @param numSites The number of camp sites to test with.
     */
    public static void testUniform(int numSites)
    {
        Random rand = new Random();
        LinkedList<Boat> requests = new LinkedList<Boat>();
        for (int i = 0; i < 2000; i++)
        {
            if (rand.nextInt() % 2 == 0)
            {
                requests.offer(new Boat(String.format("M%04d",i), Boat.Type.Motor,
                        Math.abs(rand.nextInt()) % 13 + 7));
            }
            else
            {
                requests.offer(new Boat(String.format("R%04d",i), Boat.Type.Oar,
                        Math.abs(rand.nextInt()) % 13 + 7));
            }
        }

        River river = new River(numSites, requests);
        river.generateSchedule();
        river.printTable();
        river.printSummary();
        river.checkSchedule();
    }

    /**
     * Tests the program with a Gaussian distribution in trip lengths.
     * @param numSites The number of camp sites to test with.
     */
    public static void testGaussian(int numSites)
    {
        Random rand = new Random();
        LinkedList<Boat> requests = new LinkedList<Boat>();
        for (int i = 0; i < 2000; i++)
        {
            if (rand.nextInt() % 2 == 0)
            {
                requests.offer(new Boat(String.format("M%04d",i), Boat.Type.Motor,
                        Math.min(Math.max((int)Math.round(rand.nextGaussian()*3)+13, 7), 19)));
            }
            else
            {
                requests.offer(new Boat(String.format("R%04d",i), Boat.Type.Oar,
                        Math.min(Math.max((int)Math.round(rand.nextGaussian()*2)+14, 7), 19)));
            }
        }

        River river = new River(numSites, requests);
        river.generateSchedule();
        river.printTable();
        river.printSummary();
        river.checkSchedule();
    }

    public static void main(String[] args)
    {
        System.out.println("Do␣you␣want␣a␣uniform␣or␣a␣Gaussian␣distribution␣in␣trip␣lengths?");
        Scanner scan = new Scanner(System.in);
        String distr = scan.next();
        if (distr.trim().toLowerCase().equals("uniform"))
        {
            System.out.println("How␣many␣sites?");
            int numSites = scan.nextInt();
            testUniform(numSites);
        }
        else if (distr.trim().toLowerCase().equals("gaussian"))
        {
            System.out.println("How␣many␣sites?");
            int numSites = scan.nextInt();
            testGaussian(numSites);
        }
        else
        {
            System.out.println("Unknown␣distribution.");
        }
    }
}
```