Assignment 3

Date of Submission - 10th March, Friday

The purpose of this assignment is twofold:

1. Programming practice involving list-comprehension and trees.
2. Illustrate how approximations based on real world situations (in this case the physics of the problem) can yield benefits in terms of computing speed with tolerable loss of precision.

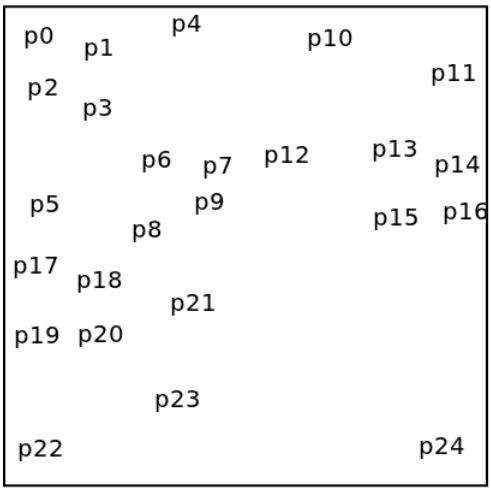
Introduction

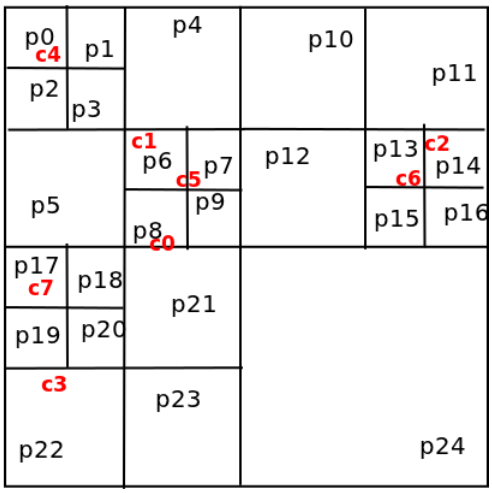
We turn to the problem of simulating the physical interaction due to gravitation between a number of particles in space and over a period of time. While we say *particles*, we mean objects of any size that can be influenced by gravitational forces. For simplicity, let us restrict ourselves to two dimensions. The particles have a mass, an initial velocity and an initial position in the region. Then, from elementary physics, you can find out the force on each of the particles. This enables you to find the new position and the new velocity after a small time Δt. This process can be repeated as many times as required.

This is fine except the computation of this process may take too long. The number of particles may be too large if one is simulating, for instance, the evolution of a galaxy. If the number of particles is n, then the force on any one of the particles is calculated by considering the other $n-1$ particles. Thus the computation at each time step takes $O(n^2)$ time. Can we reduce this time using a combination of data structures and the physics of the situation involved?

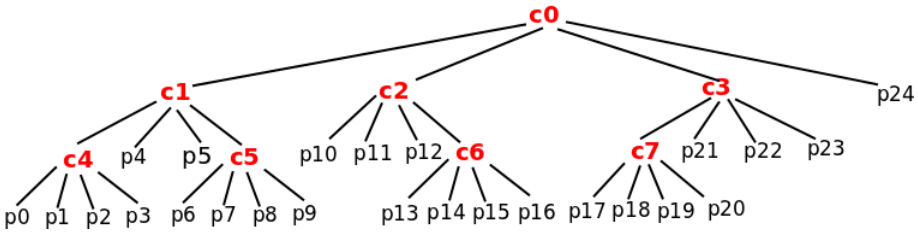
It turns out that a simplification can be achieved by the following principle: Consider a particle p and a group of particles G. If all the particles in G are in a region that is "sufficiently far away from p" in a well-defined sense, then the particles in G can be approximated by a single particle at its centroid. This is the idea that we shall use.

Assume that all the particles are within a square region. The first figure shown below shows the initial position of particles in the region.





We now group the particles as follows: If a square has more than one particle, divide it into four squares of equal size. Repeat this process recursively till each square has at most one particle (a quadrant may be empty) as shown in the second figure. If the particle falls on the horizontal dividing line, choose the upper quadrant. If the particle falls on the vertical dividing line, choose the quadrant on the right.



The grouping can be represented by the tree shown below. Note that a particle can be within several groups. For instance, the particle p_1 is in the group represented by the square c_4 as well as the containing squares c_1 and c_0. This is indicated by the fact that the leaf p_1 is under the interior nodes c_4, c_1 and c_0.

Now what do the interior nodes in the tree represent? In a sense, we can think of them as also being particles. The mass of a particle at an interior node is the sum of the particles represented by its sub-trees. Its position is the centroid of masses of the particles represented by the sub-trees of the node.

This is useful in the following way: Let us consider the force on the particle p_5. Assume that we have a notion of "being near" that we shall make precise later. We first ask whether p_5 is near the centroid c_0. If it is, we go to each of the children of c_0 and ask the same question. Assume that p_5 is near c_1 but far from the centroids c_2 and c_3. Then instead of considering the particles under c_2 or c_3, we consider the centroid itself for force calculation. Needless to say, we also stop on reaching a leaf node such as p_{24}. Going further, assume that p_5 is near c_5 but not c_4. Then the entire set of entities that would

be considered for calculating the force on p_5 are: c_4, p_4, p_6, p_7, p_8, p_9, c_2, c_3 and p_{24}. Clearly, many particles did not have to be considered.

Now all that remains to be done is to define the notion of near (or far for that matter). If the current square under consideration has a side s and the distance between two particles is d, then the particle is far from the centroid of the region, if d/s is greater than a quantity θ. Usually θ is taken to be 2. Note that if θ is very large, then no internal node is treated for force calculation, and the algorithm degenerates to the brute force $O(n^2)$ algorithm.

Programming

In this assignment you will calculate the positions of a set of initial particles and display them on a canvas. Your main code will look like:

```
(define (main ps)
  (define (main-helper ps i)
    (cond [(> i iter) (display "Done")]
          [else (let*
                    ([ps-next (singlestep ps)])
                  (if (= (remainder iter drawtime) 0)
                      (begin
                        (draw-particles ps)
                        (main-helper ps-next (+ i 1)))
                        (main-helper ps-next (+ i 1))))]))
  (main-helper ps 0))
(main testList1)
```

The heart of the program is the function `singlestep`:

```
(define (singlestep particles)
  (let* ([initialArea (bounding-box particles)]
         [tree (buildTree initialArea particles)]
         [forces (calcForces initialArea tree particles)])
    (moveparticles particles forces)))
```

`singlestep` takes a list of particles, and produces a list of particles with changed positions and velocities after a period `timeslice`. `timeslice` is a global variable.

If the region under consideration is fixed, particles can go out of it. To eliminate this possibility, the region is made variable. Given a list of particles, the function `bounding-box`

will calculate the smallest square (with its sides parallel to the axis) which can hold the particles. `bounding-box` is being provided. However, note that the canvas size and position on which particles are being drawn is fixed. Thus particles can move in and out of the canvas.

Your main job is to write the three functions `buildTree`, `calcForces` and `moveparticles`. Here is what they do:

- `buildTree` builds the tree, given the initial area and a list containing the current configuration of the particles.
- From this tree and the initial area, `calcForces` calculates the force on each particle. The result of this function is a list of forces.
- Finally `moveparticles` uses the forces to compute new configurations.

We are giving you the following four files:

1. A template file called `main.rkt`. This will containing the template for the main function.
2. A file that will be used for drawing called `drawing-routine.rkt`.
3. A file `declarations.rkt` containg all the `struct` declarations and other utility routines.
4. A file containing some testdata, `testdata.rkt`.

Your job will be to complete the program by writing the functions left out. Do not take collisions into account. Assume that the particles pass through each other.

The file `testcases.rkt` contains two test cases. The first test case contains just three particles and the second test case contains ten particles. We shall show our implementation on the test cases in the class and try to explain the results.