15-150 Spring 2012
Homework 07

Out: March 21, 2012
Due: April 4, 2012, 09:00

1 Introduction

In this homework, you will first get some practice using exceptions and doing analysis of sequence code. In the main problem on the assignment, you will implement the Barnes-Hut approximation algorithm for the n-body problem.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your `hw07.pdf` and modified `exceptions.sml` and `barnes-hut.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw07/
```

Your files must be named exactly `hw07.pdf` and `exceptions.sml` and `barnes-hut.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/07/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `exceptions.sml` and `barnes-hut.sml` files must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Methodology

Unless otherwise specified, you must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose or tests even if the implementation of the function is correct.

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on April 4, 2012, 09:00. Barnes-Hut is a larger piece of code than you have written thus far this semester; start early! Remember that this deadline is final and that we do not accept late submissions.

2 Sequence Library

We have provided an implementation of a sequence library that provides the functions listed below. This assignment will be phrased mostly in terms of these functions and types, rather than the more familiar SML base types.

- `Seq.length : 'a Seq.seq -> int`
  `Seq.length s` evaluates to the number of items in `s`.
- `Seq.empty : unit -> 'a Seq.seq`
  `Seq.empty ()` evaluates to the sequence of length zero.
- `Seq.cons : 'a -> 'a Seq.seq -> 'a Seq.seq`
  If the length of `xs` is `l`, `Seq.cons x xs` evaluates to a sequence of length `l+1` whose first item is `x` and whose remaining `l` items are exactly the sequence `xs`.
- `Seq.singleton : 'a -> 'a Seq.seq`
  `Seq.singleton x` evaluates to a sequence of length 1 where the only item is `x`.
- `Seq.append : 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`
  If `s1` has length l_1 and `s2` has length l_2, `Seq.append` evaluates to a sequence with length $l_1 + l_2$ whose first l_1 items are the sequence `s1` and whose last l_2 items are the sequence `s2`.
- `Seq.tabulate : (int -> 'a) -> int -> 'a Seq.seq`
  `Seq.tabulate f n` evaluates to a sequence `s` with length `n` where the i^{th} item of `s` is the result of evaluating `(f i)`. `Seq.tabulate f i` raises `Range` if `n` is less than zero.
- `Seq.nth : int -> 'a Seq.seq -> 'a`
  `nth i s` evaluates to the i^{th} item in `s`. This is zero-indexed. `Seq.nth i s` will raise `Range` if `i` is negative or greater than `(Seq.length s)-1`.
- `Seq.filter : ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter p s` returns the longest subsequence `ss` of `s` such that `p` evaluates to `true` for every item in `ss`.[1]
- `Seq.map : ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`
  `Seq.map f s` maps `f` over the sequence `s`. That is to say, it evaluates to a sequence `s'` such that `s` and `s'` have the same length and the i^{th} item in `s'` is the result of applying `f` to the i^{th} item of `s`.
- `Seq.reduce : (('a * 'a) -> 'a) -> 'a -> 'a Seq.seq -> 'a`
  `Seq.reduce c b s` combines all of the items in `s` pairwise with `c` using `b` as the base case. `c` must be associative, with `b` as its identity.

[1] Here we use the term "subsequence" to mean any subsequence of a sequence, not necessecarily one whose elements are consecutive in the original sequence. For example, $\langle\rangle$, $\langle 3\rangle$, and $\langle 2, 4\rangle$ are subsequences of $\langle 1, 2, 3, 4\rangle$.

- `Seq.mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`
  `Seq.mapreduce l e n s` is equivalent to `Seq.reduce n e (Seq.map l s)`.

- `Seq.toString : ('a -> string) -> 'a Seq.seq -> string`
  `Seq.toString ts s` evaluates to a string representation of `s` by using `ts` to convert each item in `s` to a `string`.

- `Seq.repeat : int -> a -> 'a Seq.seq`
  `Seq.repeat n x` evaluates to a sequence consisting of exactly `n`-many copies of `x`.

- `Seq.flatten : 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten ss` is equivalent to `reduce append (empty ()) ss`

- `Seq.zip : ('a Seq.seq * 'b Seq.seq) -> ('a * 'b) Seq.seq`
  `Seq.zip (s1,s2)` evaluates to a sequence whose n^{th} item is the pair of the n^{th} item of `s1` and the n^{th} item of `s2`.

- `Seq.split : int -> 'a Seq.seq -> 'a Seq.seq * 'a Seq.seq`
  If `s` has at least `i` elements, `Seq.split i s` evaluates to a pair of sequences `(s1,s2)` where `s1` has length `i` and `append s1 s2` is the same as `s`. Otherwise it raises `Range`.

- `Seq.take : int -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.take i s` evaluates to the sequence containing exactly the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

- `Seq.drop : int -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.drop i s` evaluates to the sequence containing all but the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

3 Exceptions

On Homework 3, you wrote `subset_sum_cert`, which we have revised to use options here:

```
(* PURPOSE
 *
 * given a multiset and a target value,
 * returns SOME (a submultiset whose members sum to the target value)
 * if there is one, or NONE if not
 *)
fun subset_sum_op (l : int list, s : int) : int list option =
    case l
        of [] => (case inteq (s, 0) of
                      true => SOME []
                    | false => NONE)
      | x::xs => (case subset_sum_op (xs, s - x) of
                      SOME c1 => SOME (x::c1)
                    | NONE => subset_sum_op (xs, s))
```

Task 3.1 (10%). Rewrite this function to use an exception to signal that there is no subset that sums to the target:

```
(* PURPOSE
 *
 * given a multiset and a target value,
 * returns a submultiset whose members sum to the target value if there is one,
 * or raises NoSubset if not
 *
 * E.g. subset_sum_exn ([2,3,2], 4) ==> [2,2]
 *      subset_sum_exn ([2,4,6], 7) raises NoSubset
 *)
exception NoSubset
fun subset_sum_exn (l : int list, s : int) : int list = ...
```

You may not use `subset_sum_op` or any other helper functions.

Task 3.2 (10%). Rewrite this function to use an exception to return the certificate:

```
(* PURPOSE
 *
 * given a multiset and a target value,
 * raises Certificate with a submultiset whose members sum to the target value,
 *     if there is one,
 * or returns () if not
 *
 * E.g. subset_sum_exn2 ([2,3,2], 4) raises (Certificate [2,2])
 *      subset_sum_exn2 ([2,4,6], 7) ==> ()
 *
 *)
exception Certificate of int list
fun subset_sum_exn2 (l : int list, s : int) : unit = ...
```

You may not use `subset_sum_op` or `subset_sum_exn` or any other helper functions. See Section 2 of the Lecture 15 notes for a similar use of exceptions.

4 Analysis

This problem asks you to analyze the running time of code that uses sequences; see Lecture 17 for the costs of the various sequence operations. See the analysis of `count` on page 8 of the notes for an example of what we expect as an explanation in following tasks.

4.1 Append and Reverse

The following function is from the Lab 9 solutions:

```
fun myAppend (s1 : 'a Seq.seq, s2 : 'a Seq.seq) : 'a Seq.seq =
    Seq.tabulate (fn i => (case i < Seq.length s1 of
                       true => Seq.nth i s1
                     | false => Seq.nth (i - (Seq.length s1)) s2))
                 (Seq.length s1 + Seq.length s2)
```

Task 4.1 (2%). Give a tight O-bound for the work of `myAppend`. Make sure you explicitly state what quantities you are analyzing the work in terms of. Briefly explain why your answer is correct.

Task 4.2 (2%). Give a tight O-bound for the span of `myAppend`. Make sure you explicitly state what quantities you are analyzing the span in terms of. Briefly explain why your answer is correct.

We implemented `reverse` using `tabulate` in Lecture 17, with $O(n)$ work and $O(1)$ span. Here is an alternate implementation:

```
fun reverse' (s : 'a Seq.seq) : 'a Seq.seq =
    Seq.mapreduce (fn x => Seq.singleton x)
                  (Seq.empty())
                  (fn (x,y) => myAppend (y, x))
                  s
```

`Seq.singleton` and `Seq.empty` take constant time.

Task 4.3 (2%). Give a tight O-bound for the work of `reverse'`, in terms of the length of `s`. Briefly explain why there is a discrepancy between this and the work of `reverse`.

Task 4.4 (2%). Give a tight O-bound for the span of `reverse'`, in terms of the length of `s`. Briefly explain why there is a discrepancy between this and the span of `reverse`.

4.2 Stocks

Recall the stock market code from Lectures 10 and 17:

```
fun suffixes (s : 'a Seq.seq) : ('a Seq.seq) Seq.seq =
    Seq.tabulate (fn x => Seq.drop (x + 1) s) (Seq.length s)

val maxS : int Seq.seq -> int = Seq.reduce Int.max minint
val maxAll : (int Seq.seq) Seq.seq -> int = maxS o Seq.map maxS

fun withSuffixes (t : int Seq.seq) : (int * int Seq.seq) Seq.seq =
    Seq.zip (t, suffixes t)

val bestGain : int Seq.seq -> int =
      maxAll
    o (Seq.map (fn (buy,sells) => (Seq.map (fn sell => sell - buy) sells)))
    o withSuffixes
```

Task 4.5 (3%). Give a tight O-bound for the work of `suffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.6 (3%). Give a tight O-bound for the span of `suffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.7 (2%). Give a tight O-bound for the work of `withSuffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.8 (2%). Give a tight O-bound for the span of `withSuffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.9 (3%). Give a tight O-bound for the work of

$$\texttt{maxAll}\langle\langle x_1^1, \ldots, x_{k_1}^1\rangle, \ldots, \langle x_1^n, \ldots, x_{k_n}^n\rangle\rangle$$

(i.e. the i^th inner sequence has length k_i and the outer sequence of sequences has length n) in terms of $k_1, \ldots, k_n$ and n. Briefly explain why your answer is correct.

Task 4.10 (3%). Give a tight O-bound for the span of

$$\texttt{maxAll}\langle\langle x_1^1, \ldots, x_{k_1}^1\rangle, \ldots, \langle x_1^n, \ldots, x_{k_n}^n\rangle\rangle$$

in terms of $k_1, \ldots, k_n$ and n. Briefly explain why your answer is correct.

Task 4.11 (3%). Give a tight O-bound for the work of `bestGain s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.12 (3%). Give a tight O-bound for the span of `bestGain s`, in terms of the length of `s`. Briefly explain why your answer is correct.

5 n-Body Simulations

5.1 Two Planes, One Simulation

5.1.1 Big Picture

The main portion of this programming assignment is modeling movements of bodies through a universe represented by a two-dimensional Euclidian plane. To make this model, we must pick an SML representation of points in the plane that allows us to meaningfully measure the distance between points—that is to say, we must pick a way to measure the universe.

The obvious choice, and the one we made in lecture, is to say that a point in the plane is a pair of real numbers, represented by a pair of values of type `real`. Values of type `real` are relatively fast to compute with, come with many helpful functions because they are built into SML, and work well with the visualizer we wrote. Critically, though, they are a floating point precision approximation to real numbers—*not* actual real numbers in the mathematical sense. In particular, addition and multiplication on values of type `real` are not always associative, and multiplication does not always distribute over addition. This means that you can do the same sequence of operations in two slightly different orders and get drastically different results:

```
- 10E30 + (~10E30 + 1.0);
val it = 0.0 : real
- (10E30 + ~10E30) + 1.0;
val it = 1.0 : real
```

We saw as early as Homework 1 that this property makes testing programs that compute with `real`s hard. Two completely correct implementations of a particular algorithm that use slightly different associations will, in general, produce different results.

A less obvious representation is to say that a point in the plane is a pair of rational numbers. Rationals are not built into SML, but can be represented: we gave an implementation of rationals in the type `Rational.t` from Homework 5. Values of type `Rational.t` actually do represent mathematical rational numbers, so all the operations that should be associative and distributive actually are. Critically, this means that the results produced by rational arithmetic are easily testable: any correct implementation of an algorithm using values of type `Rational.t` will produce the same output.

There is, however, a price to pay: computation involving values of type `Rational.t` is typically very slow because the type is implemented with arbitrary precision integer arithmetic. It's so slow that you can't use a simulation implemented using `Rational.t` to simulate anything very large or very interesting. More damningly, abstract mathematical rational numbers don't support Euclidian distance: if x and y are rational numbers, it is not necessarily the case that

$$\sqrt{x^2 + y^2}$$

is a rational number. Another definition of distance—the so-called Manhattan Distance—is an appropriate definition of distance mathematically speaking in that it makes pairs of rationals into a metric space, but does not intuitively model the universe we know.

5.1.2 Avoiding A Choice

So we have two possible ways to represent points in space, each with benefits and drawbacks. Instead of picking one over the other, we'll use both as they suit us. The reason that this is valid is that both pairs of rationals and pairs of reals form a mathematical object called a metric space: if we only program using properties of metric spaces in general, it doesn't matter which particular metric space we choose when we run the code.

To that end, you will write your code for this assignment without committing to either representation of points and using only functions that work on an abstract idea of points in general.

For testing, we have included implementations of a plane built from both real and rational points. This gives you two choices:

- If you want to run your code to see if your algorithm is correct: use the rational plane and compare your output to ours. (Details in Section 5.5.1.)
- If you want to run your code, once you've decided that it's correct, to see the results of the simulation, or if you want to see how far off your code is: use the real plane and pipe the output into the visualizer. (Details in section 5.4.3.)

Your grade for this assignment will factor in the behaviour of your code on *both* representations of the plane. We will test that your implementation produces exactly the same results as ours with rational points and also that it produces roughly the same results in the real metric space. This means that your solution must not use operations specific to any metric space and must cleanly compile with either.

5.2 The Plane

To make your code work with both implementations of the plane, you will write it in terms of types named `Plane.scalar`, `Plane.point`, and `Plane.vec`, which represent numbers, points, and vectors, respectively. We have provided two implementations of these types, one with scalars implemented by reals and the normal Euclidean distance metric, and one with scalars implemented by rationals and Manhattan distance.

The implementation of the plane using reals is in `realplane.sml`; the implementation using rationals is `ratplane.sml`. The file `space.sig` lists all of the functions common to both of these implementations of the plane, with comments describing their behavior. We suggest you read the summary in `space.sig` first, and then read `realplane.sml` if you are confused about what an operation does—the rationals implementation is a little harder to read. We describe some of the operations in more detail here:

5.2.1 Scalars

The type `Plane.scalar` is equipped with the following functions:

- `Plane.s_plus : Plane.scalar * Plane.scalar -> Plane.scalar` which computes the sum of two scalars.

- `Plane.s_minus : Plane.scalar * Plane.scalar -> Plane.scalar` which computes the difference of two scalars.
- `Plane.s_times : Plane.scalar * Plane.scalar -> Plane.scalar` which computes the product of two scalars.
- `Plane.s_divide : Plane.scalar * Plane.scalar -> Plane.scalar` which computes the quotient of two scalars.
- `Plane.s_dist : Plane.scalar * Plane.scalar -> Plane.scalar * Plane.scalar -> Plane.scalar` which computes the distance between two pairs of scalars.
- `Plane.s_compare : Plane.scalar * Plane.scalar -> order` which computes the ordering between two scalars.
- `Plane.s_fromRatio : IntInf.int * IntInf.int -> Plane.scalar`
  `Plane.s_fromRatio (x,y)` evaluates to the value of type `Plane.scalar` which represents $\frac{x}{y}$.
- `Plane.s_toString : Plane.scalar -> string` which computes a string representation of a scalar.

There are other helper functions in the file implemented in terms of these; see `space.sig` for a description. By using only the above operations, your code will work with either implementation of the plane.

5.2.2 Points and Vectors

In order to write our implementation of the Barnes-Hut algorithm, we need several operations on vectors and points in space, many of which we discussed in Lecture 18. The type `Plane.point` is used to represent a point in space, and the type `Plane.vec` is used to represent vectors of velocity, acceleration, etc. Whatever the definition of `Plane.scalar` is, we define the type of points and vectors as in lecture:

```
type Plane.point = Plane.scalar * Plane.scalar
type Plane.vec = Plane.scalar * Plane.scalar
```

This uses `Plane.scalar` for numbers. Here we give a brief description of some of the functions you may need on this assignment:

- `Plane.distance : Plane.point -> Plane.point -> Plane.scalar`
  `Plane.distance p1 p2` evaluates to the distance between the points `p1` and `p2`.
- `Plane.midpoint : Plane.point -> Plane.point -> Plane.point`
  `Plane.midpoint p1 p2` evaluates to the midpoint of the points `p1` and `p2`.

- `Plane.head : Plane.vec -> Plane.point`
  `Plane.head v` evaluates to the point that corresponds to the displacement of `v` from the origin.

- `Plane.sum : ('a -> Plane.vec) -> 'a Seq.seq -> Plane.vec`
  `Plane.sum f s` evaluates to the vector that corresponds to the sum of the sequence of vectors that results from mapping `f` on `s`. Recall that we used this in lecture to correspond to the mathematical notion of a sum of vectors, Σ.

5.3 Barnes-Hut

In lecture on Thurday, we will discuss how to solve the n-body problem in the naïve, quadratic manner. The code for this is given in `mechanics.sml` and `naiveNBody.sml`. Recall that the pieces of information we need about a body in space are its mass, location, and velocity. This is represented by the type definition

```
type body = Plane.scalar * Plane.point * Plane.vec
```

The type `body` is used to represent the different bodies in the n-body simulation. Specifically, in an expression `(m, p, v)` of type `body`, `m` is the mass of the body, `p` is its position, and `v` is the vector representing its velocity.

The naïve, quadratic implementation of an n-body simulation is given by the function

```
accelerations : body Seq.seq -> Plane.vec Seq.seq
```

in `naiveNBody.sml`. This function transforms a sequence of bodies into a sequence in which the element at position `i` represents the acceleration for the element at position `i` of the sequence of bodies.

One of the vital helper functions for this is

```
accOn : body * body -> Plane.vec
```

found in `mechanics.sml`. Recall the specification is that `accOn b1 b2` calculates the acceleration on `b1` due to `b2`. Using this function, the calculation is fairly straightforward:

```
fun accelerations (bodies : body Seq.seq) : Plane.vec Seq.seq =
    Seq.map (fn b1 => Plane.sum (fn b2 => accOn (b1, b2)) bodies) bodies
```

However, on large inputs, this implementation is accurate, but unacceptably slow for an actual simulation. There are many different approximations that have been developed; the one we will look at is called Barnes-Hut.

5.3.1 The algorithm

In short, Barnes-Hut groups bodies by quadrants (in the two-dimensional case) and uses a threshold value θ to determine whether each individual body is "far enough" away from a group of other bodies. If it is, it groups the other bodies into a big *pseudobody* and uses that for the acceleration calculation instead of each individual body composing the pseudobody. This results in a loss of accuracy, but a dramatic speedup in terms of runtime—while the old algorithm had work in $O(n^2)$, this algorithm's work is in $O(n \log n)$ if the threshold value is well-chosen.

To calculate the effect of a pseudobody on another body, it is important to know the total mass of all the bodies represented by the pseudobody and also their center of mass or *barycenter*. Therefore, when we form a pseudobody, we will compute a tuple `(m, c)` such that `m : Plane.scalar` is the total mass of the bodies and `c : Plane.point` is the barycenter. To compute the barycenter, we compute a weighted average of the vectors corresponding to the displacement of each body's position from the origin. For example, if the positions are given by the set $\{p_i \mid i \in I\}$ and the corresponding masses are given by the set $\{m_i \mid i \in I\}$ then we compute the following vector[2]:

$$\mathbf{R} = \frac{\sum_{i \in I} m_i \mathbf{r}_i}{\sum_{i \in I} m_i}$$

where $\mathbf{r}_i$ is the vector corresponding to the displacement of position p_i from the origin. The barycenter is then the point that results from displacing the origin point by $\mathbf{R}$.

Given the total mass and barycenter, we approximate the acceleration due to all the bodies in the group as the acceleration due to a single body located at the barycenter with mass equal to the total mass.

5.3.2 Computing the barycenter

We will begin by writing the `barycenter` function to compute the total mass and barycenter of a sequence of pairs of masses and points.

Task 5.1 (10%). Write the function

```
barycenter : (Plane.scalar * Plane.point) Seq.seq
           -> Plane.scalar * Plane.point
```

that computes the pair `(m, c)` where `m` is the total mass of the bodies in the sequence (*i.e.*, the sum of the first components of the pairs) and `c` is the barycenter. You should use the provided `scale_point` function to compute a weighted vector from a mass and position. You should also use the `Plane.sum` function.

[2] taken from Wikipedia

5.3.3 Grouping bodies

We still have not discussed exactly *how* to group bodies. There are many different ways of doing so, but the most straightforward is by grouping things into quadrants (for the 2D case). That is, starting at the center of the area, we divide the field into quadrants, then recursively group the bodies in each quadrant, stopping when a region has either zero or one body in it. This yields a tree-structured division of space. Unsurprisingly, we can represent this tree structure as a datatype in SML.

```
datatype bhtree =
    Empty
  | Single of body
  | Cell of (Plane.scalar * Plane.point) * BB.bbox * (bhtree Seq.seq)
```

`Empty` represents a region with no bodies in it. `Single b` represents a region with exactly the body `b` in it. `Cell ((m, c), bb, sq)` is somewhat more complicated:

- `m` is the total mass of the bodies contained in the region.
- `c` is the barycenter of the bodies contained in the region.
- `bb` is the bounding box of the region. The type `BB.bbox` represents a rectangular region in two-dimensional space. You will want to use the functions whose types and specs are given in `bbox.sig`. These functions are implemented in `bbox.sml`.
- `sq` is the sequence of the four quadrants in the region. The invariant is that this sequence is always of length four and that the four child `bhtree`'s are, in order, the top-left, top-right, bottom-left, and bottom-right quadrants of the region, respectively.

As a first step in constructing this tree, we will write the `quarters` function to split a bounding box into four equally sized quadrants.

Task 5.2 (10%). Write the function

```
quarters : BB.bbox -> BB.bbox Seq.seq
```

to compute a sequence of four bounding boxes that correspond to the top-left, top-right, bottom-left, and bottom-right quadrants of the argument bounding box. You may find some of the functions in `bbox.sml` helpful. In particular, note that the `BB.vertices` function computes the sequence of the top-left, top-right, bottom-left, and bottom-right corners of a bounding box.

Once we have the four quadrants, we need to partition the bodies between them. To do this we will use the function

```
seqPartition : ('a -> int) -> 'a Seq.seq -> int -> 'a Seq.seq Seq.seq
```

The expression `seqPartition f s n` evaluates to a sequence of `n` sequences such that the elements of the $\texttt{i}^{\text{th}}$ sequence are those elements of `s` which evaluated to `i` when supplied as an argument to `f`.

Therefore, we need a function to supply as the first argument to `seqPartition`. This function must determine the quadrant in which each point in a sequence should be placed.

Task 5.3 (10%). Write the function

```
firstMatch : BB.bbox Seq.seq -> Plane.point -> int
```

`firstMatch s p` should return the index of the first bounding box in `s` which contains `p`. If `p` is not in any bounding box in `s`, the function should raise an exception (*e.g.*, `raise Fail "Invariant violation"`). You should use the function `BB.contained` to determine if a point is contained in a bounding box.

Observe that supplying `firstMatch` applied to the sequence of four quadrants to `seqPartition` enables the partitioning of the points in a bounding box we desire. The specification of `firstMatch` entails that ties (*i.e.*, points that are on the line dividing two quadrants) will be broken in favor of the first of the two quadrants in the sequence.

5.3.4 Growing the tree

We now have the tools we need to compute a `bhtree` from a sequence of points and a bounding box.

Task 5.4 (20%). Write the function

```
compute_tree : body Seq.seq -> BB.bbox -> bhtree
```

such that `compute_tree s bb` evaluates to `T`, where `T` is the tree decomposition of `s` in the bounding box `bb`. Futhermore, we have the invariant that all of the bodies in `s` are within the bounding box `bb` and that no two bodies in `s` occupy the same position (*i.e.*, have equal position components). Your implementation does not have to handle arguments that violate these invariants. **You must use `Seq.map` to compute the recursive calls in parallel!**

Note: If a large set of bodies is partitioned into several subsets and the barycenter of each subset is known, the barycenter of the whole set can be more efficiently computed as the barycenter of the barycenters of the subsets. In this instance, the barycenter of the bodies in a bounding box can be computed as the barycenter of the four quadrants' barycenters. You should use this observation to compute the barycenter in the recursive case of `compute_tree` by applying the `barycenter` function you wrote earlier to the sequence of the barycenters of the quadrants given by the results of the recursive calls. You may find it helpful to write a helper function `center_of_mass : bhtree -> Plane.scalar * Plane.point` to project the relevant data from the sequence of recursive results.

5.3.5 Computing acceleration

Now that we can calculate the tree determined by a group of bodies, we can use it to efficiently compute an approximation of the acceleration of all the bodies at this particular timestep. This brings us back to the threshold value θ mentioned above.

The reason Barnes-Hut is more efficient than the naïve approach is that it does not compute the exact acceleration—instead, it uses θ to determine exactly how precise to be. Whenever your algorithm reaches a region with more than one body in it (that is, a `Cell` in the tree), it checks to see if $\frac{m}{d} \leq \theta$ (where `m` is the total mass of the region and `d` is the distance from the body being checked to the region's barycenter). If it is, then the region is treated as one large body located at its barycenter (which we have conveniently already calculated!). Otherwise, the region gets decomposed into quadrants and the respective accelerations from the bodies in each quadrant are computed recursively, combined, and returned.

Task 5.5 (10%). Write the function

```
too_far : Plane.point -> Plane.scalar * Plane.point -> Plane.scalar -> bool
```

such that given a point `p`, the mass and location `(m,c)` of a pseudobody, and a threshold `t`, `too_far p (m,c) t` evaluates to `true` if $\frac{m}{d} \leq \theta$ and `false` otherwise. Recall that in this equation, m refers to the mass of the pseudobody, d refers to the distance between the point and the pseudobody, and θ corresponds to the threshold argument `t`. You may find the function `Plane.distance` to be useful.

There is one more thing that we need to take into consideration. Keep in mind that if the body whose accelerations we are computing is within the bounding box of the current region, then we *need* to decompose the region into its quadrants so we do not accidentally treat the body as affecting its own acceleration. Just because this is an approximation does not mean that we can be sloppy! This check then subsumes the threshold check, which should then be made only if the body does not inhabit the current region.

Task 5.6 (20%). Write the function

```
bh_acceleration : bhtree -> Plane.scalar -> body -> vec
```

such that `bh_acceleration T threshold b` computes the acceleration on `b` from the tree `T` according to the algorithm described above. (**Hint:** The function `accOnPoint` in `mechanics.sml` may be useful.) **You should use `Plane.sum` to compute the recursive calls in parallel and add the resulting accelerations!**

The provided function `barnes_hut` uses your `compute_tree` and `bh_acceleration` functions to form the Barnes-Hut tree for a sequence of bodies and then use it to compute the acceleration on each body in the sequence.

5.4 Running the Simulation

Now that you have written all of the important code, you can actually visualize an n-body simulation with some of the infrastructure that we have given you!

5.4.1 Choosing a plane

To load your code using the floating point implementation of the plane, at the REPL issue the command

```
- CM.make "sources-real.cm";
```

To load your code using the rational implementation of the plane, at the REPL issue the command

```
- CM.make "sources-rational.cm";
```

`CM.make` uses the SMLNJ *compilation manager* to load many different files, including all of our support code and the code that you write. Once you have done `CM.make` to compile your code once, you can `use "barnes-hut.sml"` to re-load the file you are working on. However, before generating transcripts, you should re-run `CM.make` to refresh the support code that depends on your implementation. An alternative is to use `CM.make` each time you want to load your code.

5.4.2 Choosing an acceleration function and generating a transcript

Once you have done the `CM.make`, you can use the functions `Simulation.runPairwise` and `Simulation.runBH`, which simulate the n-body problem, using the naïve algorithm covered in class and your implementation of the Barnes-Hut algorithm, respectively. These functions create a file containing transcripts of the results of the simulation at each time step. Note that the Barnes-Hut algorithm that you implement is not the same as the naïve quadratic algorithm! The former is an approximation of the latter. Therefore, you should expect the visualization of the output of your code to be roughly the same as the visualization of the output of the naïve code, but the outputs themselves will not agree.

```
Simulation.runBH : Mechanics.body list -> Plane.scalar -> int -> string -> unit
Simulation.runPairwise : Mechanics.body list -> Plane.scalar -> int -> string -> unit
```

A call `Simulation.runBH bodies dt niters filename` simulates the system containing `bodies` for `niters` steps, with a time-step of `dt` seconds, and writes the transcript to `filename`.

In addition, in the file `solars.sml`, we have provided you with data for the sun and the planets of the solar system, which are useful for constructing test simulations.

For example (**Note: This is different from the transcripts we provided.**)

```
(* Simulates the earth-sun system for a year *)
Simulation.runBH Solars.solar_system
                 (Plane.s_fromInt 86400) (* seconds in a day *)
                 365                     (* days in a year *)
                 "transcript.txt";
```

This produces a file `transcript.txt` in the current working directory. The values `Solars.sun`, `Solars.earth`, and so on define the planets, and the simple solar systems `Solars.one_body` and `Solars.two_body` may be helpful for testing.

5.4.3 Running the Visualizer

Once you have produced a transcript file, you can visualize it by navigating to

```
http://www.cs.cmu.edu/~15150/visualizer/
```

You can then load a transcript file in one of two ways: either dragging and dropping the transcript file into the dashed box, or using the file browser to select the file manually. **The visualizer will only work with floating point transcripts!** Once you select a transcript, click 'Go!' to run the visualizer.

In the visualization, objects are displayed as circles. Note that the scale of the display will be chosen automatically so that all bodies are visible on-screen at all times. If one body is very far away from the others at some step of the simulation, this can result in objects at different positions appearing very close or even being indistinguishable in the visualization.

5.5 Testing

Barnes-Hut is more difficult to test than the previous homeworks, because the data structures are more complex and examples are harder to write out by hand. We encourage you to test your code using all of the following techniques:

- Run the visualizer! The existence of many bugs will be pretty easy (and funny) to spot.
- Because your solution consists of several functions that build on each other, it is in your interest to test each function in isolation, which will help you figure out where your bugs are. For Tasks 5.1-5.4, we have provided some helper functions and some hard-coded points/bounding boxes/bodies, along with suggestions about what we expect for tests. You are of course welcome to write more tests than the suggestions describe, but you will receive full testing points for following the suggestions.
- For the final Task 5.6, you should test your overall implementation by running on the rational plane and `diff`ing the output against ours, as described below. You do not need to include hard-coded tests in your SML file. Running the visualization with the output from the floating point scalars should give you a rough idea of if your code is correct, but comparing with our output on the rational scalars will be more precise.

5.5.1 Testing with the Rational Plane Simulation

We have provided the output of our code built from the rational scalars on a few examples. These are in the `tests` directory. The file `tests/README.txt` describes how each file was created so that you can run the same command and compare the output. Note that some of these tests will take a while to run: we tried to simulate the whole solar system with rationals for three days but killed the job after a half an hour. You do not need to exactly match the times above, but you shouldn't be way off.

To do this, produce a transcript file after doing `CM.make "sources-rational.cm"` as described above. Instead of giving this file to the visualizer, use the UNIX command `diff` to compare it to the appropriate file in the tests directory. `diff` takes two file names as arguments on the command line. For example, if you created a transcript file named `student.txt` from simulating the whole solar system for one day, you would check to see if it's correct by doing

```
diff student.txt tests/sun-earth.1day.txt
```

If `diff` doesn't make any output, as in

```
bovik@unix34 hw08 % diff student.txt tests/sun-earth.1day.txt
bovik@unix34 hw08 %
```

then the files are exactly the same and your code agrees with ours on that case. If `diff` does produce output, as in

```
bovik@unix34 hw08 % diff student.txt tests/sun-earth.1day.txt
2c2
< (0, 0),(0, 57909100000),(0, ~108208930000),(0, ~149600000000),(0,
227939100000),(0, 778547200000),(0, ~1433449370000),(0, 2876679082000),(0,
~45)
---
> (0, 0),(0, 57909100000),(0, ~108208930000),(0, ~149600000000),(0,
227939100000),(0, 778547200000),(0, ~1433449370000),(0, 2876679082000),(0,
~4503443661000)
bovik@unix34 hw08 %
```

then your code does not agree with ours on that case.[3]

[3] `diff` is a robust and useful UNIX utility for comparing text; you can learn more about it starting at http://en.wikipedia.org/wiki/Diff.