CS 109A Project #2

Due March 17, 1995

Modeling the Stock Market

This project is about investing in the stock market and optimal strategies given knowledge of the future. The price of share of stock for any company varies from day to day, and we can make profits by buying shares at a particular price and selling it later for a higher price. While shares of many different companies are bought and sold on the stock-market, we will simplify the problems involved by considering the buying and selling of shares of only one company.

An important component in determining our investment decisions is the price of the share on each day. We can represent the history of prices of the particular stock by a list of length n, giving the price of each share for n consecutive days. For example, the list `[24,36,15]` means the price of a share was $24 on the first day, rose to $36 on the second day, and dropped to $15 on the third day.

As an investor, on each day, we could chose to buy some shares (if we have any money), sell some shares (if we have any shares), or chose to do nothing. However, we have to pay a brokerage fee every time we buy or sell any shares. To simplify the problem, let's assume that we have to pay a fixed amount of money regardless of the number of shares we buy or sell.

For example, suppose the stock's price is given by the list `[25,35,60,20,40]`, we start with $500 in hand, and the fee is $25 for each transaction. We could invest all the $500 on the first day to buy 19 shares (remember that we need to spend $25 to buy or sell any shares, leaving us with only $475 to spend on the cost of the shares themselves). Then, we cannot buy any shares on the second day since we have no money in hand. However, we may chose to sell 10 of our shares on the third day and have $600 - $25 = $575 in hand. We could then buy 20 shares on the fourth day for $400 + $25, thus leaving us with $150 and 29 shares in hand. On the fifth day, we could sell all of our shares for $1160 and end up with ($1160 - $25) + $150 = $1285. We can represent this sequence of activities by the list `[(1,"buy",19),(3,"sell",10),(4,"buy",20),(5,"sell",29)]` to indicate that we bought 19 shares on day 1, sold 10 shares on day 3, and so on.

In general, we can represent our activities as an investor by a list of tuples with three components where the first component is a natural number indicating the day of the transaction, the second component is a string indicating the nature of the transaction (buying or selling), and the third component is a natural number indicating the number of shares that were bought or sold.

> **Question 1.** Write an ML function `financial-status` such that given the tuple `(history, init, fee, activity)` as argument, returns the money we have at the end. Here, the stock price history is given by the list `history`, the initial cash holdings is `init`, the fee for making any transaction on the stock market is `fee`, and the investing activities are given by the list `activity`. Thus, `financial-status` will have type `int list * int * int * ((int * string * int) list) -> int`.

Getting the Biggest Bang for our Buck

For any particular history of stock-prices over a period, initial cash holdings, and cost of each stock market transaction, we wish to find the sequence of activities that give us the most money at the end of the period. Since we always only have some finite amount of money, there are only finitely many choices for the number of shares we can buy on any day, and since we only possess a finite number of shares on any day there are only finitely many choices for the number of shares that we can sell. Thus there are only finitely many different possible sequence of activities. Then, one obvious algorithm to find the optimal sequence of activities is the brute force one, *i.e.*, we examine all possible (finitely many) sequences of activities and use the function `financial-status` to chose the one that leaves us with the greatest money at the end. However, even though the number of sequences that we have to examine is finite. It is still a large number, and the resulting algorithm is extremely inefficient.

One way to obtain a more efficient algorithm is to analyze some properties that the sequence of activities with the maximum financial return always possesses — we then need to look only among sequences satisfying these properties which would hopefully be much smaller in number.

The optimal sequence of investment activities has the following two properties:

1. We only buy shares at times when no shares are held, and the number of shares bought is the maximum possible with the money held on that day
2. On any day that shares are sold, all the shares held on that day are sold.

As an interesting exercise, you can try convincing yourself of the above statements. Otherwise, you can also just assume their truth. The above properties are useful because in looking for the optimal sequence we do not have to consider all possible choices of number of shares to be bought or sold on any day; once we have decided to buy/sell on a day the number of shares to be bought/sold is automatically determined by our cash/share holdings on that day. Stated another way, in representing optimal sequences we can drop the third component from the previously described representation; *i.e.*, we can represent an optimal sequence as a list of tuples of just two components where the first component gives the day of the transaction and the second component gives the nature of the transaction (*i.e.*, it is either `"buy"` or `"sell"`).

The main part of this programming project is to design an *efficient* algorithm to find the optimal, clairvoyant investment sequence. More precisely, given as inputs the stock history, our initial cash holdings, and the fee for making any stock-market transaction, you need to output the optimal sequence with maximum returns. While the two properties above are helpful for your task, note that simply searching among all sequences with the above property would not be a useful way to proceed. In trying to enumerate all possible sequences with the above stated property we note that on any particular day we can do one of two things — (a) sell or buy, depending on whether we possess shares or not respectively, or (b) do nothing. Thus, for a history of n days, there are 2^n such sequences, and examining all such sequences would still lead to an exponential-time and consequently inefficient algorithm.

Question 2. Write a function `optimize` of type `int list * int * int -> (int * string) list` such that `optimize (history, init, fee)` returns the optimal sequence of investment activities represented as a list of tuples of the form `(int * string)` where the first component indicates the day of the transaction, the second one indicates the nature of the transaction, *i.e.* "buy" or "sell", (you can skip the days when you do nothing). The given parameters are: `history` is a list representing the history of stock-prices, `init` gives the initial cash-holdings, and `fee` is the cost of each transaction on the stock-market.

Hint: Consider the following two ways of computing the Fibonacci numbers:

```
(* fibExp(n) returns the nth Fibonacci number *)
fun fibExp 0 = 1
 |  fibExp 1 = 1
 |  fibExp n = fibExp(n-1) + fibExp(n-2);

(* fibLin(n) returns the nth and (n-1)st Fibonacci numbers *)
fun fibLin 1 = (1,1)
 |  fibLin n = let
                  val (fim1, fim2) = fibLin(n-1);
               in
                  (fim1+fim2, fim1)
               end;
```

The function `fibExp` is the naive way of implementing the recursive definition of Fibonacci numbers. The running time of `fibExp` can be analyzed to be exponential in its argument. Intuitively, the source of the inefficiency of `fibExp` can be traced to the fact that we compute `fibExp(n-2)` twice, once in the call to `fibExp(n-1)` and once in the call to `fibExp(n-2)`. However, if we use the already computed result of the call `fibExp(n-2)` in computing `fibExp(n-1)` we would obtain a more efficient algorithm. One way of achieving this is to define a function to return all the fibonacci numbers that we need to use in any recursive call. This is illustrated by the efficient (linear-time) function `fibLin` which returns the pair `(fibLin(n), fibLin(n-1))`.

How does this help you solve the given problem? On any day, in deciding whether you should be "in" the market (*i.e.* you buy new shares or continue to hold some shares) or "out" of the market (*i.e.* you sell all your shares or continue to hold no share), you should keep all the "important" information up to that day (see how `fibLin` keeps the information). For example, the "important" information can be the maximum amount of money you have if you are "out", and the maximum number of shares if you are "in" plus the remainder.

From that information, you can then decide whether you should "buy", "sell", or do nothing on that day. However, you should design your function to compute more than just the needed answer, just like `fibLin` computes more than it needs, so that future calls to it can get all the information *they* need.

What You Have to Do

To do this project, **you are permitted to organize into teams of up to three students each.** Here are a few guidelines:

- In writing the functions given in Questions 1, 2, you **cannot** change the names and the types of the functions and the arguments.
- Your finished program must be able to run under SML/NJ on the Sweet Hall machines.

The following procedure to hand in the project on-line is similar to that of Project #1.

1. Choose one of your team member's leland account name to be the filename for your finished solution.

2. Log onto one of the Sweet Hall machines and copy your finished program using the name you chose in the preceding step into /usr/class/cs109a/dropoff.
   For example, if you name your file "makebigbuck.sml" and the account name of one of the team members is "wallstreetwizard", you can type
   `cp makebigbuck.sml /usr/class/cs109a/dropoff/wallstreetwizard`
   After you've done this, you will not be able to modify, read, or delete the file in the dropoff directory. You will, however, be able to use `ls` to see that your file got copied successfully.

In addition, you need to turn in:

1. The name you used for your on-line file in the dropoff directory.

2. The printout of the above two functions and all helper functions you wrote.

3. A few paragraphs explaining the algorithm you use to solve this problem and why you think it works (this shouldn't take more than a page). Analyze the complexity of your algorithm as a function of the length of the list `history`.

To Ease Your Task

There is a file named `"helpers.sml"` in the class directory which contains a few functions that help you test your programs. In particular, the function `generator`, given an integer n, generates a random list of n numbers ranging from 10 to 200, and the function `printlist` prints out the given int list.