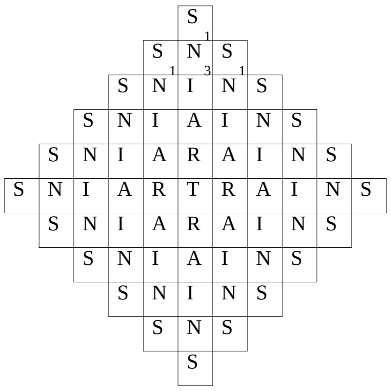
CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 5
Dynamic Programming

Jeff Edmonds

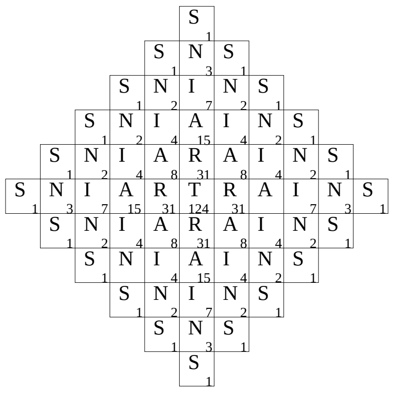
1. In one version of the scrabble game, an input instance consists of a set of letters and a board and the goal is to find a word that returns the most points. A student described the following recursive backtracking algorithm for it. The bird provides the best word out of the list of letters. The friend provides the best place on the board to put the word. Why are these bad questions?

   - Answer: Asking to provide the best word is not a "little question" for the bird. She would be doing most of the work for you. Asking the friend to provides the best place on the board to put the word is not a subinstance of the same problem as that of the given instance.

2. I saw this puzzle on a Toronto subway. The question is how many times the word "TRAINS" appears, winding snaking. We could count them but this might be exponential in the number of squares. Instead, for each box do a constant amount of work and write one integer. In the end, the answer should appear in the box with a "T". You should give a few sentences explaining the order you fill the boxes and how you do it and how much work it is.



   - Answer: In each box with an "S" write a 1. In each box with an "N" write the number of times "NS" appears starting in that box. In each box with an "I" write the number of times "INS" appears starting in that box. Similarly, "AINS", "RAINS", and "TRAINS". As an example, consider the one with "T" in it. Each time "TRAINS" appears starting this box, it must first move from the box "T" to a neighboring box with an R in it and then continue on as "RAINS" starting from there. Hence, the number of times "TRAINS" appears starting in our first box is the sum of the numbers written in the neighboring boxes with an R, i.e. $124 = 31 + 31 + 31 + 31$. The boxes should be filled in order S, N, I, A, R, and finally T. No work should be redone. A constant amount of time is spent for each box (or proportional to its degree). Hence, the running time is linear in the number of boxes.


S 1
S N S 1 3 1
S N I N S 1 2 7 2 1
S N I A I N S 1 2 4 15 4 2 1
S N I A R A I N S 1 2 4 8 31 8 4 2 1
S N I A R T R A I N S 1 3 7 15 31 124 31 7 3 1
S N I A R A I N S 1 2 4 8 31 8 4 2 1
S N I A I N S 1 4 15 4 2 1
S N I N S 1 2 7 2 1
S N S 1 3 1
S 1

3. (Answer is in the slides) A classic optimization problem is the *integer-knapsack problem.* For the problem in general, no polynomial algorithm is known. However, if the volume of the knapsack is a small integer, then dynamic programming provides a fast algorithm.

Integer-Knapsack Problem:

Instances: An instance consists of $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i, and its price is p_i.

Solutions: A solution is a subset $S \subseteq [1..n]$ of the objects that fit into the knapsack, i.e., $\sum_{i\in S} v_i \leq V$.

Measure Of Success: The cost (or success) of a solution S is the total value of what is put in the knapsack, i.e., $\sum_{i\in S} p_i$.

Goal: Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

- Answer: The Knapsack Problem:

  Algorithm using Trusted Bird and Friend: Consider a particular instance, $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$ to the knapsack problem. The little bird knows a solution S to it.

  2) Question for Bird: I ask the little bird whether or not this optimal solution contains the n^{th} item from the store.

  Possible Answers from Bird: There are $K = 2$ possible answers, Yes and No.

  No:

  3) Constructing Subinstances: If the little bird tells me *not* to put the n^{th} item into my solution, then we simply delete this last item from consideration. This leaves us with the smaller instance, $subI_{no} = \langle V, \langle v_1, p_1\rangle, \ldots, \langle v_{n-1}, p_{n-1}\rangle\rangle$, which we give to a friend. He gives me an optimal solution *optSubSol* for it.

  4) Constructing a Solution for My Instance: Trusting both the bird and the friend, my solution and its value/cost are same as his.

  Yes:

  3) Constructing Subinstances: If, on the other hand, she says to include the last item, then we can take this last item and put it into the knapsack first. This leaves a volume of $V - v_n$ in the knapsack. We determine how best to fill the rest of the knapsack with the remaining items by asking our friend to solve the smaller instance $subI_{no} = \langle V - v_n, \langle v_1, p_1\rangle, \ldots, \langle v_{n-1}, p_{n-1}\rangle\rangle$. Here too, he gives me an optimal solution *optSubSol* for it.

  4) Constructing a Solution for My Instance: Trusting both the bird and the friend, my solution is the same as my friends, except I add the n^{th} item in the space of volume v_n left for it.

  5) Costs of Solution: Having the extra item in it, the value of my solution is p_n more then the value of my friend's.

  Yes but wrong:

  3) Constructing Subinstances: If the last item does not fit into the knapsack because $v_n > V$ and the bird says to include it, then we politely tell her that she is wrong.

  5) Costs of Solution: We set the value of this solution to be $-\infty$ to ensure that it is not selected as the best one.

  Recursive Back Tracing Algorithm:

  6) Best of the Best: I can trust the friend because he is a recursive version of myself. Not actually having a little bird, I try all her answers and take best of best.

  7) Base Cases: If there are $n = 0$ items or the volume of the knapsack is $V = 0$, then the only solution is to put nothing in the knapsack for a value of zero.

Dynamic Programming Algorithm:

1) The Set of Subinstances: The set of subinstances *subI* **(Describe)**. By tracing the recursive algorithm, we see that the set of subinstances ever given to me, my friends, their friends is $\{\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_i\rangle\rangle \mid V' \in [0..V], i \in [0..n]\}$. Note that the items considered are a contiguous prefix of the original items, indicating a polytime algorithm. However, in addition we are considering every possible smaller knapsack size.

Closed: Applying the sub-operator to an arbitrary subinstance $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_i\rangle\rangle$ from this set constructs subinstances $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_{i-1}, p_{i-1}\rangle\rangle$ and $\langle V' - v_i, \langle v_1, p_1\rangle, \ldots, \langle v_{i-1}, p_{i-1}\rangle\rangle$, which are contained in the stated set of subinstances. (The second one is not when $V - v_n < 0$, but we don't actually recurse in this case.) Therefore, this set contains all subinstances generated by the recursive algorithm.

Generating: For some instances, these subinstances might not get called in the recursive program for every possible value of V'. However, as an exercise you could construct instances for which each such subinstance was called.

3) Construct a Table Indexed by Subinstances: The table indexed by the above set of subinstances will have a dimension for each of the parameters i and V' used to specify a particular subinstance. The tables will be $optCost[0..V, 0..n]$ and $birdAdvice[0..V, 0..n]$.

6) The Order in which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait. (from smaller to larger instances). This could be one row at a time with both i and V' increasing, one column at a time, or even diagonally.

8) Code:

algorithm *Knapsack* $(\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle)$

$\langle$***pre−cond***$\rangle$**:** V is the volume of the knapsack. v_i and p_i are the volume and the price of the i^{th} objects in a store.

$\langle$***post−cond***$\rangle$**:** *optSol* is a way to fill the knapsack with the greatest possible total price. *optCost* is its price.

```
begin
    % Table: $subI[V', i]$ denotes the subinstance of optimally filling a knapsack with vol-
            ume $V'$ with the first $i$ objects.
            $optSol[V', i]$ would store an optimal solution for it, but it is too big. Hence,
            we store only the bird's advice $birdAdvice[V', i]$ given for the subinstance
            and the cost $optCost[V', i]$ of an optimal solution.
    $table[0..V, 0..n]\ birdAdvice, optCost$

    % Base Cases: The base cases are when the number of objects is zero.
      For each, the solution is the empty knapsack with cost zero.
      (If the knapsack has zero volume, then nothing will fit in it until there are zero items.)
    loop $V' = 0..V$
        % $optSol[V', 0] = \emptyset$
        $optCost[V', 0] = 0$
        $birdAdvice[V', 0] = \emptyset$
    end loop

    % General Cases: Loop over subinstances in the table.
    loop $i = 1$ to $n$
        loop $V' = 0$ to $V$
            % Solve instance $subI[V', i]$ and fill in the table at index $\langle V', i\rangle$.
```

```
                % The bird and Friend Alg: The bird tells us either (1) exclude
                  the $i^{th}$ item from the knapsack or (2) include it. Either way, we
                  remove this last object, but in case (2) we decrease the size of
                  the knapsack by the space needed for this item. Then we ask
                  the friend for an optimal packing of the resulting subinstance.
                  He gives us (1) $optSol[V', i-1]$ or (2) $optSol[V' - v_i, i-1]$
                  which he had stored in the table. If the bird had said we were
                  to include the $i^{th}$ item, then we add this item to the friend's
                  solution. Denote this resulting solution by $optSol_{\langle\langle V',i\rangle,k\rangle}$. It is
                  a best packing for our instance $subI\ \langle V', i\rangle$ from amongst those
                  consistent with the bird's $k^{th}$ answer.
                % Try each possible bird answers.
                % cases $k = 1, 2$ where 1=exclude 2=include
                    % $optSol_{\langle\langle V',i\rangle,1\rangle} = optSol[V', i-1]$
                    $optCost_{\langle\langle V',i\rangle,1\rangle} = optCost[V', i-1]$
                    if($V' - v_i \geq 0$) then
                        % $optSol_{\langle\langle V',i\rangle,2\rangle} = optSol[V' - v_i, i-1] \cup i$
                        $optCost_{\langle\langle V',i\rangle,2\rangle} = optCost[V' - v_i, i-1] + p_i$
                    else
                        % Bird was wrong
                        % $optSol_{\langle\langle V',i\rangle,2\rangle} = ?$
                        $optCost_{\langle\langle V',i\rangle,2\rangle} = -\infty$
                    end if
                % end cases
                % Having the best, $optSol_{\langle\langle V',i\rangle,k\rangle}$, for each bird's answer $k$,
                  we keep the best of these best.
                $k_{max}$ = "a $k$ that maximizes $optCost_{\langle\langle V',i\rangle,k\rangle}$"
                % $optSol[V', i] = optSol_{\langle\langle V',i\rangle,k_{max}\rangle}$
                $optCost[V', i] = optCost_{\langle\langle V',i\rangle,k_{max}\rangle}$
                $birdAdvice[V', i] = k_{max}$
            end for
        end for
        $optSol = KnapsackWithAdvice\left(\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle, birdAdvice\right)$
        return $\langle optSol, optCost[V, n]\rangle$
    end algorithm
```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time: The number of subinstances is $\Theta(V \cdot n)$ and the bird chooses between two options: to include or not to include the object. Hence, the running and the space requirements are both $\Theta(V \cdot n)$. However, this running time should be expressed as a function of input size. The number of bits needed to represent the instance $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$ is $N = |V| + n \cdot (|v| + |p|)$, where $|V|$, $|v|$, and $|p|$ are the the numbers of bits needed to represent V, v_i, and p_i. Expressed in these terms, the running time is $T(|\text{instance}|) = \Theta(nV) = \Theta(n2^{|V|})$. This is quicker than the brute force algorithm because its running time is polynomial in the number of items n. In the worst case, however, V is large and the time can be exponential in the number of bits N. I.e., if $|V| = \Theta(N)$, then $T = \Theta(2^N)$. In fact, the knapsack problem is one of the classic NP complete problems, which means that it is generally believed that not polynomial time algorithm exists for it.

4. Stock Market Prices You are very lucky to have a time machine bring you the value each day of a set of stocks. The input instance to your problem consists of $I = \langle T, S, Price\rangle$, where T is an integer indicating your last day to be in the market, S is the set of $|S|$ stocks that you consider, and $Price$ is a table such that $Price(t, s)$ gives the price of buying one share of stock s on day t. Buying stocks

costs an overhead of 3%. Hence, if you buy p dollars worth of stock s on day t, then you can sell them on day t' for $p \cdot (1-0.03) \cdot \frac{Price(t',s)}{Price(t,s)}$. You have one dollar on day 1, can buy the same stock many times, and must sell all your stock on day T. You will need to determine how you should buy and sell to maximize your profits.

Because you know exactly what the stocks will do, there is no advantage in owning more than one stock at a time. To make the problem easier, assume that at each point time there is at least one stock not going down and hence at each point in time you alway own exactly one stock. A solution will be viewed a list of what you buy and when. More formally, a solution is a sequence of pairs $\langle t_i, s_i\rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1}. (Here $i \geq 1$, $t_1 = 1$ and $t_{last+1} = T$.) For example, the solution $\langle\langle 1,4\rangle, \langle 10,8\rangle, \langle 19,2\rangle\rangle$ means that on day 1 you put your one dollar into the 4th stock, on day 10 you sell all of this stock and buy the 8th stock, on day 19 you sell and buy the 2nd stock, and finally on day T you sell this last stock. The value of this solution is $\Pi_i \left[(1-0.03) \cdot \frac{Price(t_{i+1},s_i)}{Price(t_i,s_i)}\right] = 1 \cdot (1-0.03) \cdot \frac{Price(10,4)}{Price(1,4)} \cdot (1-0.03) \cdot \frac{Price(19,8)}{Price(10,8)} \cdot (1-0.03) \cdot \frac{Price(T,2)}{Price(19,2)}$. (Note that the symbol Π_i works the same as $\sum_i$ except for product.)

Design for a dynamic programming algorithm for this stock buying problem. Be sure to include ALL the steps given in the solution for the assignment. Hint: Ask the bird for the last "object" in the solution. Be sure to explain what this means.

- Answer:

  1) Specifications: (See question) An input instance consists of $I = \langle T, S, Price\rangle$, where T is the day that I must sell, S is the set of stocks, and $Price$ gives the prices. A solution is a sequence of pairs $\langle t_i, s_i\rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1}. (Here $i \geq 1$, $t_1 = 1$ and $t_{last+1} = T$.) The value of this solution is $\Pi_i \left[(1-0.03) \cdot \frac{Price(t_{i+1},s_i)}{Price(t_i,s_i)}\right]$.

  Algorithm using Trusted Bird and Friend: I have my instance $I = \langle T, S, Price\rangle$. The little bird knows a solution to it.

  2) Question for Bird: I ask the little bird what is the last object $\langle t_k, s_k\rangle$ in the solution, namely what stock $s_k \in S$ I should buy last and one what day $t_k \in [1..T-1]$ I should buy it. Note I will sell this stock on day T.

  2') Possible Answers from Bird: There are $T \cdot |S|$ different answers $\langle t_k, s_k\rangle$ that she might give.

  3) Constructing Subinstances: Given the bird wants me to buy on day t_k and I sell and buy on the same day, I need my friend to tell me what and when to buy and sell so as to sell on day t_k. Therefore, I give him the subinstance $subI = \langle t_k, S, Price\rangle$. Note, I do not need to change the set of stocks considered and even though he wont use the entire table $Price$, we don't need to change it either. He gives me an optimal solution $optSubSol$ for it.

  4) Constructing a Solution for My Instance: I produce an optimal solution $optSol$ for my instance I from the bird's answer k and the friend's solution $optSubSol$ simply by tacking the last object $\langle t_k, s_k\rangle$ on to the end of the friend's solution. This means that my friend and I buy and sell the same stocks on the same days, we both sell on day t_k, then I continue on to buy stock s_k on this same day t_k, later to sell it on day T.

  5) Costs of Solution: If $optSubCost$ is the cost of my friend's optimal solution $optSubSol$ for his instance $subI$, then my cost $optCost$ to my solution $optSol$ is $optSubCost \times \left[(1-0.03) \cdot \frac{Price(T,s_k)}{Price(t_k,s_k)}\right]$

  Recursive Back Tracing Algorithm:

  6) Best of the Best: I can trust the friend because he is a recursive version of myself. Not actually having a little bird, I try all her answers and take best of best.

  7) Base Cases: The base case instance is when we have to sell on day one. Its solution is to never buy or sell anything. It's value is one because we still have the dollar that we started with.

Dynamic Programming Algorithm:

1) The Set of Subinstances: We determine the set of subinstances $subI = \langle t, S, Price\rangle$ ever given to me, my friends, their friends. For each day $t \in [1..T]$, there is a subinstance that asks what to buy and sell so that on day t you sell your last stock and maximize the amount of money you have on that day. There are T such subinstances. Note that this set is closed under this "sub"-operator and all of these subinstances are needed.

3) Construct a Table Indexed by Subinstances: The table is simply indexed by $t \in [1..T]$. We don't actually store the solution $optS[t]$ for the subinstance $subI[t]$, but $optCost[t]$ is the cost of this solution and $birdAdvice[t]$ stores the birds advice given on this subinstance.

6) The Order in which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait. (from smaller to larger instances). This is simply $for\ t = 1 \ldots T$.

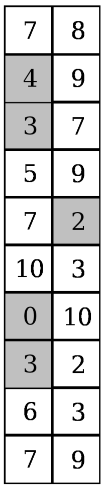
8) Code:

```
algorithm $Stocks\,(T, S, Price)$
⟨pre−cond⟩: $T$ is the day that I must sell, $S$ is the set of stocks, and $Price$ gives the
  prices.
⟨post−cond⟩: $optSol$ is an optimal valid schedule. It is a sequence of pairs $\langle t_i, s_i\rangle$,
  meaning that stock $s_i$ is bought on day $t_i$ and sold on day $t_{i+1}$. $optCost$ is its cost.
begin
    % Table: $subI[t]$ denotes the subinstance of finding an optimal schedule ending on
            day $t$.
            $optSol[t]$ would store an optimal solution for it, but it is too big. Hence,
            we store only the bird's advice $birdAdvice[t]$ given for the subinstance and
            the cost $optCost[t]$ of an optimal solution.
    $table[1..T]\ optCost, birdAdvice$

    % Base Case: The only base case is for the optimal set ending on day $t = 1$.
      It's solution consists of the empty set with value 1.
    % $optSol[1] = \emptyset$
    $optCost[1] = 1$
    $birdAdvice[1] = \emptyset$

    % General Cases: Loop over subinstances in the table.
    for $t = 2$ to $T$
        % Solve instance $subI[t]$.
        % Try each possible bird answer.
        for each $t_k \in [1..t-1]$
            for each $s_k \in S$
                % The bird and Friend Alg: I want to finish on day $t$. I ask the bird
                  what stock $s_i \in S$ I should buy last and what day $t_i \in [1..t-1]$
                  I should buy it. She answers $\langle t_k, s_k\rangle$. I ask my friend to solve
                  the subinstance that ends on day $t_k$. I produce an optimal solu-
                  tion $optSol$ for my instance $subI[t]$ from the bird's answer $k$ and
                  the friend's solution $optSubSol$ simply by tacking the last object
                  $\langle t_k, s_k\rangle$ on to the end of the friend's solution. This means that my
                  friend and I buy and sell the same stocks on the same days except
                  after we both sell everything on day $t_k$, I buy stock $s_k$ later to sell
                  it on day $T$.
                  Denote this resulting solution by $optSol_{\langle t,\langle t_k,s_k\rangle\rangle}$. It is a best so-
                  lution for our instance $subI[t]$ from amongst those consistent with
                  the bird's $\langle t_k, s_k\rangle^{th}$ answer.
                % $optSol_{\langle t,\langle t_k,s_k\rangle\rangle} = optSol[t_k] + \langle t_k, s_k\rangle$
                $optCost_{\langle t,\langle t_k,s_k\rangle\rangle} = optCost[t_k] \cdot \left[(1-0.03)\cdot \frac{Price(t,s_k)}{Price(t_k,s_k)}\right]$
            end for
            % Having the best, $optSol_{\langle t,\langle t_k,s_k\rangle\rangle}$, for each bird's answer $\langle t_k, s_k\rangle$, we keep the best
              of these best.
            $\langle t_{max}, s_{max}\rangle$ = "a $\langle t_k, s_k\rangle$ that maximizes $optCost_{\langle t,\langle t_k,s_k\rangle\rangle}$"
            % $optSol[t] = optSol_{\langle t,\langle t_{max},s_{max}\rangle\rangle}$
            $optCost[t] = optCost_{\langle t,\langle t_{max},s_{max}\rangle\rangle}$
            $birdAdvice[t] = \langle t_{max}, s_{max}\rangle$
        end for
        $optSol = SchedulingWithAdvice\,(T, S, Price, birdAdvice)$
        return $\langle optSol, optCost[T]\rangle$
    end algorithm
```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time:
The number of subinstances in the table is T.
The number of bird answers is $T \cdot |S|$.
The running time is the product of these $\mathcal{O}(T^2 \cdot |S|)$.

5. Dynamic Programming the Narrow Art Gallery Problem:
(See ACM contest open.kattis.com/problems/narrowartgallery):
A long art gallery has $2N$ rooms. The gallery is laid out as N rows of 2 rooms side-by-side. Doors connect all adjacent rooms (north-south and east-west, but not diagonally). The curator has been told that she must close off r of the rooms because of staffing cuts. Visitors must be able to enter using at least one of the two rooms at one end of the gallery, proceed through the gallery, and exit from at least one of the two rooms at the other end. Therefore, the curator must not close off any two rooms that would block passage through the gallery. That is, the curator may not block off two rooms in the same row or two rooms in adjacent rows that touch diagonally. Furthermore, she has determined how much value each room has to the general public, and now she wants to close off the set r rooms that minimize the sum of the values of the rooms closed, without blocking passage through the gallery.



7	8
4	9
3	7
5	9
7	2
10	3
0	10
3	2
6	3
7	9

Figure 1: Shows an example of an art gallery of $N = 10$ rows and 2 columns of rooms. The number of rooms to close is $r = 5$. The number 1-10 in each room gives its value. The gray rooms indicate which should be closed in the optimal solution.

(a) Algorithmic Paradigms:
Tell me which one is **wrong** or say all are right.

A: An *iterative* algorithm takes one step at a time. During each it makes a little progress while maintaining a loop invariant.
A *recursive* algorithm asks his friends any instance that is smaller and meets the precondition. It is best not to micro managing these friends.

B: A *greedy* algorithm grabs the next best item and commits to a decision about it without concern for long term consequences.

C: A *recursive backtracking* algorithm tries various things. For each thing tried it recurses. Then it backtracks and tries something different.

D: A *dynamic programming* algorithm fills in a table with a cell for each of a set of subinstances. Each is solved in the same way as one stack frame of the recursive backtracking algorithm.

E: They are all right.

- Answer: E: They are all right.

(b) Specification of the Narrow Art Gallery problem:
Tell me which one is **wrong** or say all are right.

A: An *instance* is specified by $I = \langle N, r, value(1..N, 1..2)\rangle$ where N is the number of rows. 2 is the number of columns. r is the number of rooms to close. For $n \in [1..N]$ and $side \in \{left, right\}$, $value(n, side)$ is the value of this room, i.e. the cost of closing it.

B: A *solution* is a subset of the $2N$ rooms of size r.

C: A solution is *valid* if a visitor traveling only east-west or north-south is able to enter the top of the gallery and leave though the bottom without traveling through a closed room. See the white path of rooms in the figure. For example, they closed the room of value 3 instead of the less valuable room next to it of value 2 because otherwise the public could not walk through.

D: The *cost* of such a solution is the sum of the values of the rooms closed. The goal is to minimize this cost.

E: They are all right.

- Answer: E: They are all right.

(c) What is the nature of the bird?
Tell me which one is **wrong**.

A: There is no bird. There is no spoon (Quote from Matrix)

B: She helps us pose what we want to try.

C: We imagine her giving us a little answer about the solution.

D: She represents an algorithmic technique for learning part of the solution.

E: She helps us trust what we are trying so that we can go on.

- Answer: D is wrong.

(d) What is the nature of the friend?
Tell me which one is **wrong**.

A: There is no friend. There is no spoon (Quote from Matrix)

B: I know you. You are just like me.

C: You don't micro manage him because it's too confusing.

D: You can give him anything that meets the precondition and that is smaller.

E: Recursion

- Answer: A is wrong.

(e) Why does the bird tell you something about the end of the solution instead of the beginning?

A: It far too confusing the other way.

B: Esthetically the dynamic programming algorithm looks better going forward.

C: Esthetically the dynamic programming algorithm looks better going backwards.

D: It makes the algorithm faster.

E: The bird only knows about the beginning.

- Answer: B is right.

(f) Given the instance in the figure, I will ask bird:
"Should I close the bottom left room (i.e. labeled 7), the bottom right room (i.e. labeled 9), or neither of them."
Tell me which one is **wrong** or say all are right.

A: It is ok not to worry here about r or the higher rooms.

B: The answer she gives is $k \in \{left, right, none\}$. The number of bird answers we will have to try is 3.

C: I don't just ask just about the bottom right room, because if we just delete it, the friend's instance would be a funny shape.

D: I don't include the option of closing both of them, because that is not valid.

E: They are all right.

- Answer: E: They are all right.

(g) One option is that we delete the bottom row of rooms (i.e. labeled 7&9) and we give the friend the first $N-1$ rows of rooms.
Tell me which one is **wrong** or say all are right.

A: We must also give the friend the new number r_{friend} of rooms to close which is our number r to close minus the number the bird closed.

B: We expect the friend to find the optimal solution for this.

C: The obvious solution for our instance of N rows is to close the rooms that our friend told us to close and to close the rooms that the bird told us to close.

D: The problem with this obvious solution is that if the bird tells us to close the bottom left room (i.e. labeled 7) and the friend tells us to close the room on the right in the second last row (i.e. labeled 3), then the solution will not be valid because the public would not be able to exit out of the bottom of the gallery.

E: They are all right.

- Answer: E: They are all right.

Our buildings will always be rectangular with two columns. The public is allowed to enter either via the top left or the top right room. However, we are going to restrict the way that the public is allowed to leave the bottom of the gallery. We change the problem so that in addition to what has been specified above, the instance includes a parameter *door* $\in \{left, right, both\}$ which tells us whether the public can leave the gallery though the bottom left room but not the bottom right, the bottom right room but not the left, or both the bottom left and the bottom right. For our initial instance, *door* = *both*.

(h) Suppose we are given the instance in the figure and the bird tells us either to close the bottom left, bottom right or none of the rooms on the bottom row. Tell me about the instance that we give our friend.
Tell me which one is **right** or say all are wrong.

A: The set of rooms he gets is the same as the set of rooms we get.

B: We change some of the values of the rooms for our friend's instance.

C: Because the bird told us the solution for this last row of rooms, we delete this last row from our friend's instance. We give the friend our first $N_{friend} = N-1$ rows.

D: We remove just the room that the bird closes.

E: They are all wrong.

- Answer: C is right.

(i) Suppose we are given the instance in the figure and the bird tells us to close the bottom left room, i.e. $k = left$. Tell me about the instance that we give our friend.
Tell me which one is **right** or say all are wrong.

A: The number of rooms r that must be closes is fixed by the boss and does not change.

B: We do not need to tell the friend how many rooms to close because his bird will tell him that.

C: The bird tells us the total number of rooms to close.

D: The number of that my friend's instance needs closed is one less than the number we must close, i.e. $r_{friend} = r-1$.

E: They are all wrong.

- Answer: D is right.

(j) Suppose we are given the instance in the figure and the bird tells us to close the bottom left room, i.e. $k = left$. Tell me about the instance that we give our friend.
Tell me which one is **right** or say all are wrong.

A: There is no need for doors.

B: The friend's instance will have no door on the bottom left, i.e. $door_{friend} = right$. This models the fact there is a pseudo room below his bottom left room that is closed and hence the public cannot enter his bottom left room in this way.

C: The friend's instance will have no door on the bottom right, i.e. $door_{friend} = left$. This models the fact there is a pseudo room below his bottom left room that is closed and hence the public cannot enter his bottom left room in this way.

D: In this case, the friend should be given a door both on the left and on the right. This gives the public the correct level of access.

E: They are all wrong.

- Answer: B is right.

(k) Suppose we are given the instance in the figure except for the fact that there is no door on the bottom left, i.e. $door = right$. Tell me about the instance that we give our friend.
Tell me which one is **wrong** or say all are right.

A: The whole business with the doors is done to avoid the following bug. If the bird tells us to close the bottom left room (i.e. labeled 7) and the friend tells us to close the room on the right in the second last row (i.e. labeled 3), then the solution will not be valid because the public would not be able to exit out of the bottom of the gallery.

B: Closing the only room on the bottom with a door will prevent the public from leaving. Hence, we will not allow the bird to close the bottom right room, i.e. if she says that $k = right$, then we politely tell her that she is wrong.

C: We politely tell the bird she is wrong by setting $optCost_{\langle I,k \rangle} = \infty$. Being a minimization problem, this option will never be selected.

D: Sometimes when writing a recursive program, we need to change the preconditions so that the friend gives us the answer that meets our needs. When converting this into a dynamic programming algorithm, this makes a larger set of subinstances.

E: They are all right.

- Answer: E: They are all right.

(l) Dynamic Programming:
Tell me which one is **wrong** or say all are right.

A: This recursive back tracking algorithm effectively tries every possible solution, i.e. brute force. The time is exponential.

B: Recursive back tracking is a common algorithmic technique that is used in artificial intelligence (certainly before machine learning).

C: A dynamic programming algorithm saves time by in a way similar to that the greedy algorithm works, i.e. if the existence of an optimal solution consistent with decision A implies the existence of an optimal solution consistent with decision B, then decision A does not need to be tried.

D: A dynamic programming algorithm saves time by not solving the same subinstance more than once.

E: They are all right.

- Answer: C is wrong: It is right for greedy algorithms, but this is not done in dynamic programming.

(m) We start the dynamic programming algorithm by setting up a table indexed by all of the subinstances that some friend friend friend will have to solve.
Tell me which one is **wrong** or say all are right.

A: This table will have a dimension that is indexed by the number of rows $N' \in [0..N]$ that the friend will consider.

B: This table will have a dimension that is indexed by the number of rooms $r' \in [0..r]$ that the friend will have to close.

C: This table will have a dimension indicating the values of the each room, i.e. the numbers 0 to 10 in the figure.

D: This table will have a dimension that is indexed by the parameter $door \in \{left, right, both\}$.

E: They are all right.

- Answer: C is wrong.

(n) What needs to be true about the set S of subinstances being solved?
Tell me which one is **wrong** or say all are right.

A: Include our original instance. (Invite the bride and groom.)

B: Closed under the friend operation. For every subinstance $I' \in S$, all of I''s friends must also be in S. (If you invite your aunt, then you must invite her friends.)

C: Don't have too many things in S that are not asked by some friend's friend.

D: Sometimes it is too hard to tell if some I' is actually asked by some friend's friend. Then we just put it in S for good luck.

E: They are all right.

- Answer: E: They are all right.

(o) The number of subinstance that we must solve is:

A: $N \times r \times 10 \times 2N$

B: $2N$

C: $3rN$

D: $\mathcal{O}(r + N)$

E: exponential (say in N, r, or the number of bits to write down the values of the rooms.)

- Answer: C is right.

(p) Each cell of the table (tables)
Tell me which one is **wrong** or say all are right.

A: Is indexed by a subinstance to be solved.

B: Stores the optimal solution for that subinstance.

C: Stores the cost of optimal solution for that subinstance.

D: Stores the bird's advice for that subinstance.

E: They are all right.

- Answer: B is wrong.

(q) What order should the subinstances in the table be completed.
Tell me which one is **wrong** or say all are right.

A: Smallest to largest.

B: In an order that nobody waits, i.e. when a subinstance is solved, all of his friends have already been solved. When its your aunt's job, her friends are already done and gotten drunk.

C: If the table is two dimentional, you have to fill it in diagonally.

D: Basecases first. Our instance last.

E: They are all right.

- Answer: C is wrong.

(r) What are the base cases?
Tell me which one is **wrong** or say all are right.

A: The smallest subinstances in your table.

B: Subinstances that don't have any friends.

C: Subinstances for which the generic code does not work.

D: Subinstances that you can solve easily on your own.

E: They are all right.

- Answer: E: They are all right.

(s) The base cases are handled as follows.
Tell me which one is **right** or say all are wrong.

A: if($r \leq 0$) then return(no rooms need to be closed)

B: if($N \leq 0$) then return(no rooms to close)

C: if($N < r$) then return(more rooms to close than can be closed)

D: % The cost of the solution is the sum of that values of the rooms closed. Hence, if there are $r = 0$ rooms to close, then the cost is zero.
for all entries of the table for which $r' = 0$, $optCost[...r;..] = 0$

E: They are all wrong.

- Answer: D is right: The case that $N < r$ has no solution and is handled in the code.

(t) Dynamic Programming
Tell me which one is **wrong** or say all are right.

A: To solve our instance, we recursively ask friends to solve smaller subinstances.

B: We loop over all subinstances from smallest to largest.

C: The word "Memoization" comes from the word "Memo," i.e. to write nodes about what has happened already.

D: To solve our instance, we look in the table to see what our friends have stored about smaller subinstances.

E: They are all right.

- Answer: A is wrong.

(u) The first thing a dynamic program does is:
Tell me which one is **right** or say all are wrong.

A: Ask the bird about an optimal solution of the inputted instance.

B: Set up the table.

C: Check if the input is a base case.

D: Check if the input has the correct format.

E: They are all wrong.

- Answer: B is right.

(v) We combine the cost of our friend's solution and the cost of our bird's solution k to get:
Tell me which one is **right** or say all are wrong.

A: The cost of an optimal solution $optCost_{I'}$ for the instance I'.

B: The cost of an optimal solution $optCost_{\langle I',k\rangle}$ for the instance I' from amongst those that are consistent with this bird's answer k.

C: We need to compute the optimal solution.

D: The whole idea of combining is misrepresents the technique.

E: They are all wrong.

- Answer: B is right.

(w) Consider a dynamic programming routine named *RoomClosures*. A key line of its is:
Tell me which one is **right** or say all are wrong.

A: $optCost_{..} = optCost[I_{friend}] + GetBirdsCost(...)$

B: $optCost_{..} = RoomClosures(I_{friend}) + optCost_{bird}$
where if($k = left$) then $opCost_{bird} = value(N', left)$.

C: $optCost_{..} = optCost[I_{friend}] + optCost_{bird}$
where if($k = none$) then $opCost_{bird} = 0$.

D: $optCost_{..} = RoomClosures(I_{friend}) + GetBirdsCost(...)$

E: They are all wrong.

- Answer: C is right: The friend's answer is looked up in the table and the bird's answer is the current one we are trying.

(x) Dynamic Programming
Tell me which one is **wrong** or say all are right.

A: We try all bird answers.

B: Having the best, $optSol_{\langle I',k\rangle}$, for each bird's answer k, we keep the best of these best.

C: The following is reasonable pseudo code.
k_{min} = "a k that minimizes $optCost_{\langle I',k\rangle}$"

D: Options A, B, and C need to be done in dynamic programming but not in recursive backtracking.

E: They are all right.

- Answer: D is wrong.

(y) Finishing up the dynamic programming algorithm.
Tell me which one is **wrong** or say all are right.

A: The following is where the memo is being taken.
$optCost[I'] = optCost_{\langle I',k_{min}\rangle}$.

B: The following is key line of how Jeff does dynamic programming, but you likely won't hear anyone else talk about it. $birdAdvice[I'] = k_{min}$.

C: The algorithm $optSol = AlgWithAdvice\,(I, birdAdvice)$ reruns the algorithm but this time it is fast because now there really is a bird.

D: The algorithm $AlgWithAdvice$ and the greedy algorithm both follow one path down the decision tree.

E: They are all right.

- Answer: E: They are all right.

(z) Running time of dynamic programming algorithms.
Tell me which one is **wrong** or say all are right.

A: The running time of a dynamic programming algorithm is the number of subinstances times the number of bird answers.

B: If an optimal solution is found in the inner loop then the time is multiplied by the number of bits to write down the solution.

C: Finding an optimal solution when the bird's advice is known is fast. The time is in linear in the number of bits to write down the solution.

D: If the input consists of n objects and each subset of these objects is a subinstance, then the number of subinstances is exponential. This occurring with the obvious dynamic programming algorithm is a very common reason for a problem to be NP-complete, i.e. no polynomial algorithm for it is known.

E: They are all right.

- Answer: E: They are all right.

6. More questions past z.

(a) Which is **true** about the running time of our RoomClosures dynamic program?

A: The running time of this algorithm is proportional to the number of rows of rooms N. The size of the input includes the $\log N$ bits to represent N. This means that this dynamic programming algorithm runs in exponential time just like the Knapsack problem.

B: The running time of this algorithm is proportional to the number of rooms to close r. The size of the input includes the $\log r$ bits to represent r. This means that this dynamic programming algorithm runs in exponential time just like the Knapsack problem.

C: A and B.

D: The running time of this algorithm is quadratic in the size of the input.

E: They are all wrong.

- Answer: D is right: The size the input is dominated by the table *value* whose description requires at least N bits. Lets say $size = \Theta(N)$. The number of rooms to close is at most the number of rows, i.e. $r \leq N$ or else the solution is impossible. The running time is the number of subinstances times the number of bird answers which is $Time(size) = 3Nr \times 3 = \mathcal{O}(N^2) = \mathcal{O}(size^2)$.

(b) Suppose instead of the number of columns being restricted to 2, the gallery could have width w. Consider extending the same basic algorithm done above.
Tell me which one is **wrong** or say all are right.

A: The running time would increase by a factor of w.

B: We would still ask the bird which rooms to close on the bottom row.

C: The number of bird answers would now be $\Theta(2^w)$.

D: This problem is NP-complete, i.e. no polynomial algorithm for it is known.

E: They are all right.

- Answer: A is wrong.

(c) Write out the code for the *RoomClosure* algorithm developed here.

```
algorithm $DynamicProgrammingAlgforRoomClosures\ (N, r, value(1..N, 1..2))$
⟨pre−cond⟩: $N$ is the number of rows. 2 is the number of columns. $r$ is the number of rooms to
    close. For $n \in [1..N]$ and $side \in \{left, right\}$, $value(n, side)$ is the value of this room, i.e. the
    cost of closing it.
⟨post−cond⟩: A solution is a subset of the $2N$ rooms of size $r$.
    A solution is valid if a visitor traveling only east-west or north-south is able to enter the top
    of the gallery and leave though the bottom without traveling through a closed room. See the
    white path of rooms in the figure.
    The cost of such a solution is the sum of the values of the rooms closed. The goal is to minimize
    this cost.
    The program returns $optSol$ which is an optimal solution for this instance and $optCost$ which
    is it's cost.
begin
    % Table: $subI[N', r', door']$ denotes the subinstance in which we consider the first $N' \in [0..N]$ columns
            of the rooms, we close $r' \in [0..r]$ of the rooms, and $door' \in \{left, right, both\}$ specifies which
            of the rooms on the bottom have doors.
            $optSol[N', r', door']$ would store an optimal solution for it, but it is too big. Hence, we
            store only the bird's advice $birdAdvice[N', r', door']$ given for the subinstance and the cost
            $optCost[N', r', door']$ of an optimal solution.
    $table[0..N, 0..r, 0..2]\ optCost, birdAdvice$

    % Base Cases: When the number of rooms $r'$ to close is zero, the solution is to close zero
      rooms with a cost of zero. We won't worry about the number $N'$ of rows being zero because
      we will make sure that $N' \geq r'$.
    for $N' \in [0..N]$, for $door' \in \{left, right, both\}$
        % $optSol[N', 0, door']$ = no rooms
        $optCost[N', 0, door'] = 0$
        $birdAdvice[N', 0, door'] = none$
    end loop

    % General Cases: Loop over subinstances in the table.
    for $N' \in [0..N]$, for $r' \in [0..min(r, N')]$, for $door' \in \{left, right, both\}$
        % Solve instance $subI[N', r', door']$ and fill in table entry $\langle N', r', door' \rangle$.
        % Try each possible bird answer.
        for $k \in \{left, right, none\}$
            % The bird and Friend Alg: Our instance either has $N'$ rows of rooms, $r'$ rooms to close
              and a door on the $door' \in \{left, right, both\}$. We ask the bird whether to close the
              bottom left room, the bottom right, or neither. She answers $k \in \{left, right, none\}$.
              Given the bird has handled the bottom row indexed $N'$, we ask the friend about
              the first $N_{friend} = N' - 1$ rows. If the bird says to close a room, then we ask the
              friend to close one few doors, i.e. $r_{friend} = r' - 1$. Similarly, if she says to close no
              rooms, then $r_{friend} = r'$. If the bird answers to delete the bottom left room, then
              we give the friend the instance no door on the bottom left, i.e. if $k = left$ then
              $door_{friend} = right$. Similarly, if the bird answers $k = right$, then $door_{friend} = left$.
              If the bird closes no rooms on the bottom then we leave both of the friend's doors
              open, i.e. if $k = none$ then $door_{friend} = both$. If we must close one room per row,
              i.e. $N' = r'$, then we make sure the bird closes at least one room. This ensures that
              $N_{friend} \geq r_{friend}$. If we are given the instance with no door on the bottom left,
              then we will not allow the bird to close the bottom right room, otherwise, the public
              can't get out, i.e. it is not the case that $door' = right$ and $k = right$. Otherwise,
              we combine our friend's room closures solution with that of the bird.
            % Create friend's instance and bird's cost.
            $N_{friend} = N' - 1$
            if( $k = left$ ) then
                $r_{friend} = r' - 1$
                $door_{friend} = right$
                $opCost_{bird} = value(N', left)$
            elseif( $k = right$ ) then
                $r_{friend} = r' - 1$
                $door_{friend} = left$
                $optCost_{bird} = value(N', right)$
            elseif( $k = none$ ) then
                $r_{friend} = r'$
                $door_{friend} = both$
                $optCost_{bird} = 0$
            endif
            % Build our solution and cost from friend's and bird's solutions.
            if( $N_{friend} < r_{friend}$ or ($door' = left$ and $k = left$) or ($door' = right$ and $k = right$) ) then
                % $optSol_{\langle\langle N',r',door'\rangle,k\rangle} = error$
                $optCost_{\langle\langle N',r',door'\rangle,k\rangle} = \infty$
            else
                % $optSol_{\langle\langle N',r',door'\rangle,k\rangle} = optSol[N_{friend}, r_{friend}, door_{friend}] + k$
                $optCost_{\langle\langle N',r',door'\rangle,k\rangle} = optCost[N_{friend}, r_{friend}, door_{friend}] + optCost_{bird}$
            endif
        end for
        % Having the best, $optSol_{\langle\langle N',r',door'\rangle,k\rangle}$, for each bird's answer $k$, we keep the best of these best.
        $k_{min}$ = "a $k$ that minimizes $optCost_{\langle\langle N',r',door'\rangle,k\rangle}$"
        % $optSol[N', r', door'] = optSol_{\langle\langle N',r',door'\rangle,k_{min}\rangle}$
        $optCost[N', r', door'] = optCost_{\langle\langle N',r',door'\rangle,k_{min}\rangle}$
        $birdAdvice[N', r', door'] = k_{min}$
    end for
    $optSol = AlgWithAdvice\,(N, r, value(N, 2), birdAdvice)$
    return $\langle optSol, optCost[N, r, both]\rangle$
end algorithm
```

7. Stock Market Prices You are very lucky to have a time machine bring you the value each day of a set of stocks. The input instance to your problem consists of $I = \langle T, S, Price\rangle$, where T is an integer indicating your last day to be in the market, S is the set of $|S|$ stocks that you consider, and $Price$ is a table such that $Price(t, s)$ gives the price of buying one share of stock s on day t. Buying stocks costs an overhead of 3%. Hence, if you buy p dollars worth of stock s on day t, then you can sell them on day t' for $p \cdot (1 - 0.03) \cdot \frac{Price(t',s)}{Price(t,s)}$. You have one dollar on day 1, can buy the same stock many times, and must sell all your stock on day T. You will need to determine how you should buy and sell to maximize your profits.

   Because you know exactly what the stocks will do, there is no advantage in owning more than one stock at a time. To make the problem easier, assume that at each point time there is at least one stock not going down and hence at each point in time you alway own exactly one stock. A solution will be viewed a list of what you buy and when. More formally, a solution is a sequence of pairs $\langle t_i, s_i\rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1}. (Here $i \geq 1$, $t_1 = 1$ and $t_{last+1} = T$.) For example, the solution $\langle\langle 1, 4\rangle, \langle 10, 8\rangle, \langle 19, 2\rangle\rangle$ means that on day 1 you put your one dollar into the 4^{th} stock, on day 10 you sell all of this stock and buy the 8^{th} stock, on day 19 you sell and buy the 2^{nd} stock, and finally on day T you sell this last stock. The value of this solution is $\Pi_i \left[(1 - 0.03) \cdot \frac{Price(t_{i+1},s_i)}{Price(t_i,s_i)}\right] = 1 \cdot (1 - 0.03) \cdot \frac{Price(10,4)}{Price(1,4)} \cdot (1 - 0.03) \cdot \frac{Price(19,8)}{Price(10,8)} \cdot (1 - 0.03) \cdot \frac{Price(T,2)}{Price(19,2)}$. (Note that the symbol Π_i works the same as $\sum_i$ except for product.)

Design for a dynamic programming algorithm for this stock buying problem. Be sure to include ALL the steps given in the solution for the assignment. Hint: Ask the bird for the last "object" in the solution. Be sure to explain what this means.

- Answer:

  1) Specifications: (See question) An input instance consists of $I = \langle T, S, Price \rangle$, where T is the day that I must sell, S is the set of stocks, and $Price$ gives the prices. A solution is a sequence of pairs $\langle t_i, s_i \rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1}. (Here $i \geq 1$, $t_1 = 1$ and $t_{last+1} = T$.) The value of this solution is $\Pi_i \left[(1 - 0.03) \cdot \frac{Price(t_{i+1}, s_i)}{Price(t_i, s_i)}\right]$.

  Algorithm using Trusted Bird and Friend: I have my instance $I = \langle T, S, Price \rangle$. The little bird knows a solution to it.

  2) Question for Bird: I ask the little bird what is the last object $\langle t_k, s_k \rangle$ in the solution, namely what stock $s_k \in S$ I should buy last and one what day $t_k \in [1..T-1]$ I should buy it. Note I will sell this stock on day T.

  2') Possible Answers from Bird: There are $T \cdot |S|$ different answers $\langle t_k, s_k \rangle$ that she might give.

  3) Constructing Subinstances: Given the bird wants me to buy on day t_k and I sell and buy on the same day, I need my friend to tell me what and when to buy and sell so as to sell on day t_k. Therefore, I give him the subinstance $subI = \langle t_k, S, Price \rangle$. Note, I do not need to change the set of stocks considered and even though he wont use the entire table $Price$, we don't need to change it either. He gives me an optimal solution $optSubSol$ for it.

  4) Constructing a Solution for My Instance: I produce an optimal solution $optSol$ for my instance I from the bird's answer k and the friend's solution $optSubSol$ simply by tacking the last object $\langle t_k, s_k \rangle$ on to the end of the friend's solution. This means that my friend and I buy and sell the same stocks on the same days, we both sell on day t_k, then I continue on to buy stock s_k on this same day t_k, later to sell it on day T.

  5) Costs of Solution: If $optSubCost$ is the cost of my friend's optimal solution $optSubSol$ for his instance $subI$, then my cost $cost$ to my solution $optSol$ is $optSubCost \times \left[(1 - 0.03) \cdot \frac{Price(T, s_k)}{Price(t_k, s_k)}\right]$

  Recursive Back Tracing Algorithm:

  6) Best of the Best: I can trust the friend because he is a recursive version of myself. Not actually having a little bird, I try all her answers and take best of best.

  7) Base Cases: The base case instance is when we have to sell on day one. Its solution is to never buy or sell anything. It's value is one because we still have the dollar that we started with.

  Dynamic Programming Algorithm:

  1) The Set of Subinstances: We determine the set of subinstances $subI = \langle t, S, Price \rangle$ ever given to me, my friends, their friends. For each day $t \in [1..T]$, there is a subinstance that asks what to buy and sell so that on day t you sell your last stock and maximize the amount of money you have on that day. There are T such subinstances. Note that this set is closed under this "sub"-operator and all of these subinstances are needed.

  3) Construct a Table Indexed by Subinstances: The table is simply indexed by $t \in [1..T]$. We don't actually store the solution $optS[t]$ for the subinstance $subI[t]$, but $cost[t]$ is the cost of this solution and $birdAdvice[t]$ stores the birds advice given on this subinstance.

  6) The Order in which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait. (from smaller to larger instances). This is simply $for\ t = 1 \ldots T$.

  8) Code:

  algorithm $Stocks\,(T, S, Price)$

⟨pre−cond⟩: T is the day that I must sell, S is the set of stocks, and $Price$ gives the prices.

⟨post−cond⟩: $optSol$ is an optimal valid schedule. It is a sequence of pairs $\langle t_i, s_i\rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1}.

```
begin
    % Table: $optSol[t]$ stores an optimal schedule ending on day $t$ and $costSol[t]$ its cost.
    $table[1..T]\ cost, birdAdvice$

    % Base Case: The only base case is for the optimal set ending on day $t = 1$.
      It's solution consists of the empty set with value 1.
    % $optSol[1] = \emptyset$
    $cost[1] = 1$
    $birdAdvice[1] = \emptyset$

    % General Cases: Loop over subinstances in the table.
    for $t = 2$ to $T$
        % Solve instance $subI[t]$.
        % Try each possible bird answer.
        for each $t_k \in [1..t-1]$
            for each $s_k \in S$
                % The bird and Friend Alg: I want to finish on day $t$. I ask the bird what
                  stock $s_i \in S$ I should buy last and what day $t_i \in [1..t-1]$ I should by it.
                  She answers $\langle t_k, s_k\rangle$. I ask my friend to solve the subinstance that ends
                  on day $t_k$. I produce an optimal solution $optSol$ for my instance $subI[t]$
                  from the bird's answer $k$ and the friend's solution $optSubSol$ simply by
                  tacking the last object $\langle t_k, s_k\rangle$ on to the end of the friend's solution.
                  This means that my friend and I buy and sell the same stocks on the
                  same days except after we both sell everything on day $t_k$, I buy stock
                  $s_k$ later to sell it on day $T$.
                % $optSol_{\langle t_k,s_k\rangle} = optSol[t_k] + \langle t_k, s_k\rangle$
                $cost_{\langle t_k,s_k\rangle} = cost[t_k] \cdot \left[(1-0.03) \cdot \frac{Price(t,s_k)}{Price(t_k,s_k)}\right]$
            end for
        % Having the best, $optSol_{\langle t_k,s_k\rangle}$, for each bird's answer $\langle t_k, s_k\rangle$, we keep the best of
          these best.
        $\langle t_{max}, s_{max}\rangle$ = "a $\langle t_k, s_k\rangle$ that maximizes $cost_{\langle t_k,s_k\rangle}$"
        % $optSol[t] = optSol_{\langle t_{max},s_{max}\rangle}$
        $cost[t] = cost_{\langle t_{max},s_{max}\rangle}$
        $birdAdvice[t] = \langle t_{max}, s_{max}\rangle$
    end for
    $optSol = SchedulingWithAdvice\,(Price, birdAdvice)$
    return $\langle optSol, cost[T]\rangle$
end algorithm
```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time:
The number of subinstances in the table is T.
The number of bird answers is $T \cdot |S|$.
The running time is the product of these $\mathcal{O}(T^2 \cdot |S|)$.

A Second Answer: There is another possible answer, but I was discouraging it because is it harder. The running time is $\mathcal{O}(T \cdot |S|^2)$ instead of $\mathcal{O}(T^2 \cdot |S|)$. Which is better depends on whether there are more stocks $|S|$ or more time steps T. In this second solution, I ask the bird what stock to own on the last day. She answers $s_k \in S$. Note there are $|S|$ different answers. The friend's subinstance will now end on day $T-1$. But it now matters what stock he owns on this his last day because if he ends with any stock other than s_k then I get charged $(1 + 0.03)$ for selling what he has in order to buy s_k. This is where the hard part comes in. We need to change the pre and post conditions of the problem to indicate which stock the friend should own on his last day and require that he not sell this stock on this last day but continue to own it. The new subinstance will be $subI = \langle t, S, Price, s\rangle$, where as before t is his last day to be in the market (except that he does not sell on this day), S is the set of $|S|$ stocks considered, and $Price$ gives the prices, but now s indicates that last stock to own. This gives $T \cdot |S|$ subinstances and a two dimensional table. You wont get part marks for getting some of the parts of this answer correct unless you can argue about the need for your friend to not sell his stock.

(a) Deleted questions about reductions Three dynamic programming algorithms occure to me.

i. The most obvious one has mT subinstances (nodes in G) and mT bird answers (degree of nodes in G) given a time of $\mathcal{O}(m^2T^2)$.

ii. The second algorithm decreases the number of bird answers (degree of nodes in G) to m given a time of $\mathcal{O}(m^2T)$.

iii. The third algorithm instead decreases the number of subinstances (nodes in G) to T given a time of $\mathcal{O}(mT^2)$.

Clearly the first is the worst. Whether the second or third is best depends on whether there are more stocks m, or more time steps T. Answers each of the following questions (except for the first) three times, one for each of these algorithms.

i. Suppose at first that there is not the 3% overhead for selling stock. Give an easy algorithm for knowing what and when to buy and sell.

- Answer: Since there is not cost in buying and selling, we might as well sell and buy every day. Hence, each day t sell the stock you have and buy the stock s_t that makes the biggest profit $\frac{Price(t+1,s_t)}{Price(t,s_t)}$ between day t and day $t+1$.

ii. Read Edmonds' notes chapter 19.9 which describes how to design a dynamic program via reductions. This has not been covered in either class. Follow the steps in this chapter to reduce this new stock problem to the longest paths problem.

A. Given the table of prices $Price(t, s)$, how do you map this to a graph G? A path is a sequence of nodes which will correspond to a sequence of *states* that you might be in during your stock buying. Hence, each node of G will represent one such state. However, you what as few nodes as possible. Hence, you must identify a small set of bench mark states that are of interest. What are these states/nodes? What is the start node $\widehat{s}$ and the finishing node $\widehat{t}$ in your graph?

- Answer: In the first and second algorithms, there is a node $u_{s,t}$ for each stock $s \in [1..m]$ and each day $t \in [1..T]$. Being in this state means that on day t you own stock s.
  In the third algorithm, **************************** REWRITE *********************

  If you pause your buying-selling sequence after you buy a stock, then your state would be determined by the current day t_i and the stock s_i that you own. However, we choose to pause after you sell the stock. Hence, your state is determined by the current day. We will have a node in the graph for each day $t \in [1..T]$. Standing on node t means that it is day t, you have just sold some stock and you are about buy something else. The start node $\widehat{s}$ is day 1 and the finishing node $\widehat{t}$ is day T.

- Answer: If you pause your buying-selling sequence after you buy a stock, then your state would be determined by the current day t_i and the stock s_i that you own. However, we choose to pause after you sell the stock. Hence, your state is determined by the current day. We will have a node in the graph for each day $t \in [1..T]$. Standing on node t means that it is day t, you have just sold some stock and you are about buy something else. The start node $\widehat{s}$ is day 1 and the finishing node $\widehat{t}$ is day T.

B. What are the edges between the nodes?

- Answer: The edges out of node t consist of your options. For each stock $s \in [1..m]$ and each day $t' \in [t+1..T]$ put an edge from node t to node t'.

C. What is the length of each edge?

- Answer: The length of this edge will be $\log\left[(1-0.03)\cdot\frac{Price(t',s)}{Price(t,s)}\right]$.

D. Given the longest path from $\widehat{s}$ to $\widehat{t}$ in your graph G, how does this tell you what and when to buy stocks?

- Answer: If your i^{th} edge in the path goes from node t to node t' and buys and sells stock s then our i^{th} item in our solution will be $\langle t_i, s_i\rangle = \langle t, s\rangle$.

E. The value $cost_{path}$ of the path is the sum of the lengths of the edges in it. The value of your resulting stock solution is $cost_{stock}$ as defined above. Show the relationship between $cost_{path}$ and $cost_{stock}$

- Answer: $\log[cost_{stock}] = \log\left[\Pi_i\left[(1-0.03)\cdot\frac{Price(t_{i+1},s_i)}{Price(t_i,s_i)}\right]\right] = \sum_i \log\left[(1-0.03)\cdot\frac{Price(t_{i+1},s_i)}{Price(t_i,s_i)}\right] = cost_{path}$.

8. I have said at various times that all dynamic programming algorithms can be *reduced* to min $s{-}t$ path in a leveled graph. This test will cover both reductions and dynamic programming by examining this relationship. See my description of how to do this in the dynamic programming steps.

(a) We will warm up by giving an algorithm for a *product* version of $LeveledGraph\,(G,s,t)$ using an "oracle" for the standard *sum* version of it.

Sum Path: We covered the problem $LeveledGraph\,(G,s,t)$. Its input is $I_{sum} = \langle G_{sum}, s_{sum}, t_{sum}\rangle$ where G_{sum} is a weighted leveled graph and s_{sum} and t_{sum} are nodes. A solution S_{sum} is a path from s to t through G. The cost of a solution $cost_{sum}(S_{sum})$ is the **sum** of the weights of the edges in the given path. If we climb the mountain, the oracle $Oracle_{sum}$ will give us am optimal solution $Oracle_{sum}(I_{sum}) = S_{sum}$, for which we can compute its cost.

Prod Path: We now define a new problem $LeveledGraph_{prod}\,(G,s,t)$. Its inputs is $I_{prod} = \langle G_{prod}, s_{prod}, t_{prod}\rangle$ and solutions S_{prod} are the same as before. The only difference is that the cost of a solution $cost_{prod}(S_{prod})$ is now the **product** of the weights of the edges in the given path. Our goal is construct an algorithm Alg_{prod}. It takes I_{prod} as input. It maps instance I_{prod} to instance I_{sum} using our program $InstanceMap(I_{prod}) = I_{sum}$. It gives I_{sum} to $Oracle_{sum}(I_{sum})$ who gives us S_{sum}. Our algorithm then maps solution S_{sum} to solution S_{prod} using our program $SolutionMap(S_{sum}) = S_{prod}$.

Correct: All you need to do to prove that your algorithm Alg_{prod} works is to show that the solution map is a bijection $SolutionMap^{-1}(S_{prod}) = S_{sum}$ and show that this bijection keeps the solutions ordered with respect to cost, namely $cost_{prod}(S_{prod}) \geq cost_{prod}(S'_{prod})$ if an only if $cost_{sum}(S_{sum}) \geq cost_{sum}(S'_{sum})$.
Hey recursion is like a reduction to the same problem - expect with smaller instances.
Hey the first dynamic programming algorithm in Test 7 suffered from the fact that the solution bijection did not keeps the solutions ordered with respect to cost.

Test Questions:
- Your task is to define $InstanceMap(I_{prod}) = I_{sum}$
- and $SolutionMap(S_{sum}) = S_{prod}$.
- Also define $CostMap(cost_{prod}) = cost_{sum}$. Why is this cost right?
Hint: What function did you learn in highschool for which $f(x \times y) = f(x) + f(y)$?

- Answer: Given $I_{prod} = \langle G_{prod}, s_{prod}, t_{prod}\rangle$, we construct $I_{sum} = \langle G_{sum}, s_{sum}, t_{sum}\rangle$ simply by taking the logarithm of each weight.
The path S_{prod} returned by our algorithm will be identical to that S_{sum} returned by the oracle.
The cost $cost_{sum}$ is given by $\sum_{e\in path}\log(w_e) = \log(\Pi_{e\in path} w_e) = \log(cost_{prod})$.

(b) Now we will give an algorithm for the stock problem using an "oracle" for the *product* version of *LeveledGraph* (G, s, t). The resulting algorithm will be *different* than that in the assignment. I wonder if it will be faster?

Stock Problem: The only difference between the version of the problem considered here and that in the practice assignment is that we are going to allow people to hold cash if all the stocks are doing poorly.

You are very lucky to have a time machine bring you the value each day of a set of stocks. The input instance to your problem consists of $I = \langle T, S, Price\rangle$, where T is an integer indicating your last day to be in the market, S is the set of $|S|$ stocks that you consider, and *Price* is a table such that $Price(t, s)$ gives the price of buying one share of stock s on day t. Selling a stock costs an overhead of 3%. Hence, if you buy p dollars worth of stock s on day t, then you can sell them on day t' for $p \cdot (1 - 0.03) \cdot \frac{Price(t',s)}{Price(t,s)}$. You have one dollar on day 1, can buy the same stock many times, and must sell all your stock on day T. You will need to determine how you should buy and sell to maximize your profits. Because you know exactly what the stocks will do, there is no advantage in owning more than one stock at a time. If all the stocks are doing poorly, you can hold cash for a period. The only difference between cash and yet another stock is that you don't pay 3% for selling cash.

Graph G_{prod}: Recall our goal is to give an algorithm for the stock problem using an "oracle" for the *product* version of *LeveledGraph* (G, s, t). Given your instance I_{stock} to the stock problem, your first task is to define the instance $InstanceMap(I_{stock}) = I_{prod} = \langle G_{prod}, s_{prod}, t_{prod}\rangle$.

Nodes of G_{prod} are Subinstances: Because I want a completely different algorithm for this Stock problem than that in the assignment, I will give you the set of subinstance that I want you to solve.

Maximize Cash at time t: These are the same subinstances that we had in the assignment. For each time t, let $subI[t]$ ask to maximize the amount of cash you have on day t (at a point in time when all stocks have been sold). $optCost[t]$ will denote this maximized amount of cash.

Maximize Stock s at time t: These subinstances are new. For each time t and stock s, let $subI[t, s]$ ask to maximize the value of stock s you have on day t, assuming you only hold this stock on this afternoon. Lets suppose that on days that you sell a stock, you do it at time 1:00pm and on days that you buy, you do so at time 1:02pm. $subI[t]$ maximizes the cash you have at time 1:01 and $subI[t, s]$ maximizes value of s you have at time 1:03. To keep all our cost units in dollars, $optCost[t, s]$ will measure the dollar value of stock s that you hold at this time. One of the possible reasons that you own this stock at this time is because you just bought it on this same day. In this case, $optCost[t]$ denotes the amount you spent at 1:02 on stock s and $optCost[t, s]$ denotes what it is worth at 1:03. In this case, we will assume that these amounts $optCost[t, s] = optCost[t]$ are the same. On the other hand, if you own stock s at time 1:03 on day t' because you bought it on some earlier day t, then because the stock has changed in value, we have that $optCost[t', s] = optCost[t, s] \times \frac{Price(t',s)}{Price(t,s)}$. Because you are only charged the 3% commission when you sell, this is not included here.

Test Questions: Your tasks are the following:

i. **The Game of Life:** Given your instance I_{stock} to the stock problem, your first task is to define the instance $InstanceMap(I_{stock}) = I_{prod} = \langle G_{prod}, s_{prod}, t_{prod}\rangle$ to give to the oracle. Recall that in the Steps document, the nodes of the graph G_{prod} was described in three ways: as the cells in the dynamic programing table, as the subinstances, and as the "Graph of Life". I have told you the subinstances. Your are to describe G_{prod} as the Graph of Life.

Hint: See how this was done in the Steps document.

A. I have told you the subinstances are $subI[t]$ and $subI[t, s]$. You are to describe these exact same nodes as states $state[t]$ and $state[t, s]$ in this game of life.

B. There are a number of different types of atomic life action that can be taken to move between these states. For each do the following.
- Describe the action that is being taken within the life of stocks.
- Which states does the corresponding edge of G_{prod} traverse between?
- How does the value of your state change when you take this action?
- What should the weights of these edge be?
  Hint: Remember the weights are multiplicative, i.e. a ratio.

C. Which are the start and terminating nodes s_{prod} and t_{prod}?

ii. What is the running time of your stock algorithm? i.e. In terms of S and T, how many edges does your graph I_{prod} have? Is it better than the time of the algorithm presented in the solution of the assignment?

iii. The oracle gives you an optimal solution S_{prod}. It is a path through the graph I_{prod} from s_{prod} to t_{prod}. How do you map this to a solution S_{stock} to your stock problem?

iv. Recall that the cost of the path S_{prod} is the product of the weights of its edges. Recall that the cost of S_{stock} is amount of money you have on the last day. How do you map this cost of S_{prod} to this cost of S_{stock}? Why is this cost right?

- Answer:

i. Lets describe G_{prod} as the Graph of Life.

A. States/Nodes:

Cash at time *t*: For every day t, $state[t]$ is the state in which it is 1:01pm on day t and you are holding some p dollars, which is as much money as you have managed to muster so far.

Stock *s* at Time *t*: For every day t and stock s, $state[t, s]$ is the state in which it is 1:03pm on day t and you are holding some p dollars worth of stock s, which is as much as you have mustered.

B. Actions/Edges: The edges between these states consist of the action that transition you from one state to another. Certainly one possible action is buying a stock s and another is selling it. Another possible "action" is not doing either of those things.

Buy Stock s on day t: Suppose it is 1:01pm on day t and you are sitting on node/state $\boldsymbol{subI}[t]$ with p dollars. Suppose at 1:02pm you buy stock s. This moves you to being in the state of it being 1:03pm on day t and you own p worth of stock s, i.e. you are in node/state $\boldsymbol{subI}[t, s]$. This action does not increase your worth p at all and hence its multiplicative ratio is one. Hence, there is an edge $\langle \boldsymbol{subI}[t], \boldsymbol{subI}[t, s]\rangle$ between these states with weight one.
Note that if this action is taken then $optCost[t, s] = optCost[t]$.

Sitting on Cash on day t: Suppose it is 1:01pm on day t and you are sitting on node/state $\boldsymbol{subI}[t]$ with p dollars. Suppose you do not buy any stocks this day. A day goes by. This moves you to being in the state of it being 1:01pm on day $t+1$ and you are sitting on node/state $\boldsymbol{subI}[t+1]$ with the same p dollars. This action does not increase your worth p at all and hence its multiplicative ratio is one. Hence, there is an edge $\langle \boldsymbol{subI}[t], \boldsymbol{subI}[t+1]\rangle$ between these states with weight one.
Note that if this action is taken then $optCost[t+1] = optCost[t]$.

Sell Stock s on day $t+1$: Suppose it is 1:03pm on day t and you are sitting on node/state $\boldsymbol{subI}[t, s]$ with p dollars worth of stock s. You have nothing to do until the next day so you play with your kids. In the mean time, your stock has gone up by the multiplicative ratio $\frac{Price(t+1,s)}{Price(t,s)}$.
Suppose at 1:00pm on day t+1 you sell stock s. This moves you to being in the state of it being 1:01pm on day t+1 and you are sitting on node/state $\boldsymbol{subI}[t+1]$ with money. Because selling costs 3% the amount you have is multiplied by 1–0.03. Hence, there is an edge $\langle \boldsymbol{subI}[t, s], \boldsymbol{subI}[t+1]\rangle$ between these states with weight $(1–0.03) \times \frac{Price(t+1,s)}{Price(t,s)}$.

Note that if this action is taken then $optCost[t+1] = optCost[t,s] \times (1-0.03) \times \frac{Price(t+1,s)}{Price(t,s)}$.

Sitting on Stock s on day $t+1$: Suppose instead you do not sell at 1:00pm on day $t+1$. This moves you to being in the state of it being 1:03pm on day $t+1$ and you are sitting on node/state $subI[t+1,s]$ with stock s. Hence, there is an edge $\langle subI[t,s], subI[t+1,s]\rangle$ between these states with weight $\frac{Price(t+1,s)}{Price(t,s)}$.

Note that if this action is taken then $optCost[t+1,s] = optCost[t,s] \times \frac{Price(t+1,s)}{Price(t,s)}$.

C. s and t: We start with \$1 on day 1. Hence our start state/note is $s_{prod} = subI[1]$. Our goal is to maximize the money that we have on day T. Hence our termination state/note is $t_{prod} = subI[T]$.

ii. Running Time: For three of the types, there is an edge for every day t and stock s. For the remaining type, there is one for every t. Hence, the total number of edges is $3TS+T$. Being the number of edges, the running time of your stock algorithm is $\mathcal{O}(TS)$ which is much better than the time $\mathcal{O}(T^2 \cdot |S|)$ of the algorithm given in the assignment.

iii. Solutions: The oracle gives you an optimal solution S_{prod}. It is a path through the graph G_{prod} from $s_{prod} = subI[1]$ to $t_{prod} = subI[T]$. Each node in this path corresponds to a state that the stock buying process may be in and each edge in it corresponds to the action that takes you from the one state to the next. As such this path exactly specifies our solution S_{stock} to our stock problem.

iv. Costs: Recall that the cost of path S_{prod} is the product of the weights of its edges. The weight of each edge in the path corresponds to the multiplicative amount that your worth increases by the action corresponding to this edge. Hence, the product of these weights corresponds to the total amount that your worth increases from \$1 on day one to $subI[T]$ dollars in state $t_{prod} = subI[T]$ on day T.

(c) In this question, you are to do an example. The following is the price table that you should work from.

Day	1	2	3	4	5	6	7
Stock 0	5.00	6.00	6.06	6.18	6.00	6.53	6.47
Stock 2	2.00	4.60	4.55	4.47	6.70	6.83	6.90

You are going to draw the graph G_{prod}, its nodes, edges, weights, optimal costs $optCost[t]$ and $optCost[t,s]$, bird answers, and optimal path. Like this:

Excel: At first I assumed that you would draw this by hand or some drawing app (I am hooked on PowerPoint). But the numbers got too tedious for me. I want you to do it in Excel. (NOT coding it up). See my file test8.xlsx

Time (black): The days and time are given in the first to rows.

***Price(t, s)* (pink):** In rows 3 and 9, I copied in the price table for the two stocks $s = 0$ and $s = 2$. Index 1 is used for cash.

***state(t, s)* (blue):** In rows 4 and 8, I put blue circles to represent the graph G_{prod} nodes $state(t,s)$ in which I own the optimal amount of stock s on day t at time 1:03.

***state(t)* (orange):** In row 6, I put orange circles to represent the graph G_{prod} nodes $state(t)$ in which I own the optimal amount of cash on day t at time 1:01.

Edges: I drew six example edges. You will have to fill in the rest of them. Or if there are too many put a bunch of them so that the TA gets the pattern and then put a dot dot dot.

I never did figure out how to draw circles and arrows in Excel. The way I created these was by making them in PowerPoint and then copying them to Excel. To keep the colours, I have to click "paste keep formatting". And when these drawings get in the way of the Excel equations, I move the drawings out of the way.

Edge Weights (black): For each (or most) of the edge, put in an Excel cells close to the edge an equation the computes the weight of the edge from the prices.

***optCost*(*t*, *s*) (blue):** For each node $state(t, s)$, put in the Excel cell within my circle an equation that computes $optCost(t, s)$, namely the optimal value of stock s that you own on day t at time 1:03.

***optCost*(*t*) (orange):** For each node $state(t)$, put in the Excel cell within my circle an equation that computes $optCost(t)$, namely the optimal amount of cash you have on day t at time 1:01.

***birdAdvice*(*t*, *s*) and *birdAdvice*(*t*) (green):** Near each node $state(t, s)$, put in the Excel cell that returns the bird's advice $birdAdvice(t, s)$, namely which in-edge gave the best answer. Do the same for $birdAdvice(t)$.
Hint: Try something like $= IF(A1 > B2, 0, IF(C1 > B2, 0, 2))$.

Path (red): Draw the optimal path on top.

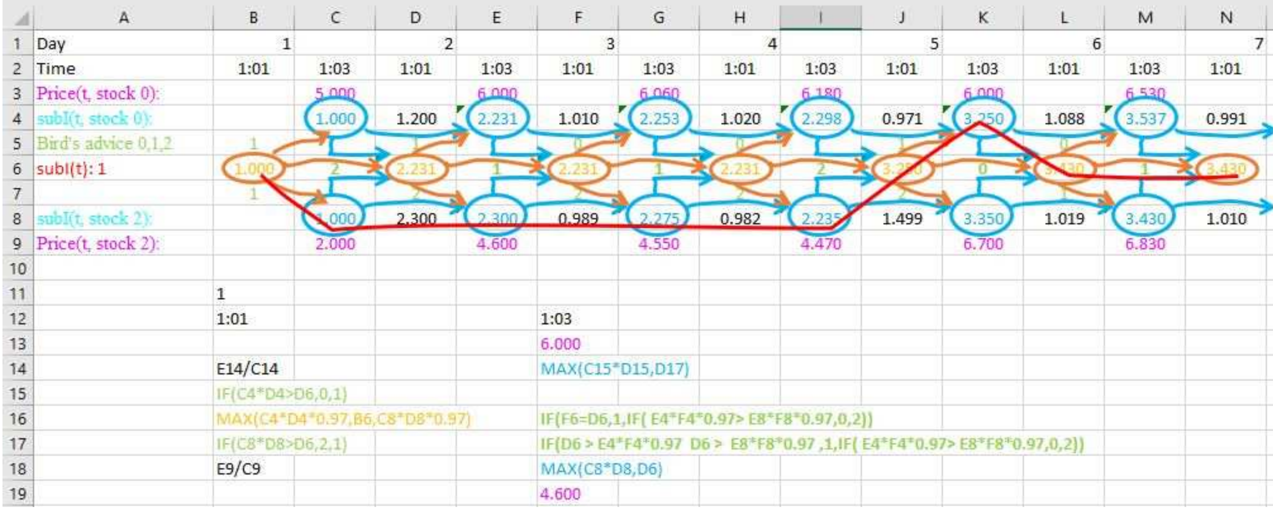
Formulas: I am giving you two copies.
- In the first you give the equation starting with = so that Excel computes the value.
- In the second you give the equation starting with no = so that Excel displays the equation as text so that the TA can easily read it.

You only need to include one example of every type of equation.

Hand In: Hand in a screen capture of this Excel file.

Fun: For fun, change the price numbers and everything else should change automatically.
I played with the prices so that the optimal thing was to own stock 2 even while it was dropping in price.

Answer:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Day	1		2		3		4		5		6		7
2	Time	1:01	1:03	1:01	1:03	1:01	1:03	1:01	1:03	1:01	1:03	1:01	1:03	1:01
3	Price(t, stock 0):		5.000		6.000		6.060		6.180		6.000		6.530	
4	subI(t, stock 0):		1.000	1.200	2.231	1.010	2.253	1.020	2.298	0.971	3.250	1.088	3.537	0.991
5	Bird's advice 0,1,2	1		1		0		0		1		0		
6	subI(t): 1	1.000	2	2.231	1	2.231	1	2.231	2	3.250	0	3.430	1	3.430
7		1		2		2		2		2		1		
8	subI(t, stock 2):		1.000	2.300	2.300	0.989	2.275	0.982	2.235	1.499	3.350	1.019	3.430	1.010
9	Price(t, stock 2):		2.000		4.600		4.550		4.470		6.700		6.830	
10														
11		1												
12		1:01				1:03								
13						6.000								
14		E14/C14				MAX(C15*D15,D17)								
15		IF(C4*D4>D6,0,1)												
16		MAX(C4*D4*0.97,B6,C8*D8*0.97)				IF(F6=D6,1,IF(E4*F4*0.97> E8*F8*0.97,0,2))								
17		IF(C8*D8>D6,2,1)				IF(D6 > E4*F4*0.97 D6 > E8*F8*0.97 ,1,IF(E4*F4*0.97> E8*F8*0.97,0,2))								
18		E9/C9				MAX(C8*D8,D6)								
19						4.600								