Tutorial 9 solutions

1. If the price of the stock on day $i-1$ is p', and we sell s units on stock on day i, then the price p of the stock on day i will be $\mathbf{p'}$ (the price on day $i-1$) $+(\mathbf{p_i} - \mathbf{p_{i-1}})$ (the expected price drop from day $i-1$ to day i) $-\mathbf{f(s)}$ (the price drop due to the s shares we're selling on day i). Isolating p', we get $p' = p + p_{i-1} - p_i + f(s)$. This tells us what the price of the stock must have been on day $i-1$ if it ends up equalling p on day i. Our recurrence then considers all of the possible number of shares we might sell on day i, and pick the ones that gives the highest income:

$$P(x,i,p) = \begin{cases} \max_{s=0}^{x}\{s \times p + P(x-s, i-1, p + p_{i-1} - p_i + f(s)\} & \text{if } i > 1 \\ x \times p & \text{if } i = 1 \text{ and } p = p_1 - f(x) \\ -\infty & \text{if } i = 1 \text{ and } p \neq p_1 - f(x) \end{cases}$$

2. We will store the choice for which the objective function is maximized in order to avoid having to again loop over possible choices in our answer to question 4.3.

```
Algorithm GeneratePriceTable(x, n, p_1, ..., p_n)
  create empty 3D table P[0 ... x, 1 ... n, 0 ... p_1]
  initialize every element of P to −1

  //
  // We know the price of the shares on day n is at most p_n
  //
  for p ← 0 to p_n do
      GPTH(x, n, p)

Algorithm GPTH(x, i, p)
  //
  // The recursion might end up with a price per share larger than
  // p_1, because we're computing the price on day i − 1
  // from the price on day i instead of the other way around.
  // We need to make sure this does not affect the result.
  //
  if p > p_1
    return −∞

  //
  // Ok, now we proceed normally.
  //
  if P[x, i, p] = -1 then
    if i = 1 and p = p_1 - f(x)
      P[x, i, p] ← x × p
```

```
    else
      P[x, i, p] ← −∞
      if i > 1
        for s ← 0 to x do
          currentIncome ← s × p + GPTH(x - s, i - 1, p + p_{i−1} - p_i + f(s))
          if currentIncome > P[x, i, p]
            P[x, i, p] ← currentIncome
            S[x, i, p] ← s
          endif
        endfor
      endif
    endif
  endif
  return P[x, i, p]
```

3.

```
Algorithm DetermineShares(x, n, p_n)

// Before we start backtracking, we need to find the value
// of p that maximimizes P[x, n, p].
maxIncome ← -1
for i ← 0 to p_n
  if P[x, n, i] > maxIncome
    maxIncome ← P[x, n, i]
    p ← i

while n > 0
  Sell[n] ← S[x, n, p]
  x ← x - Sell[n]
  n ← n - 1

return Sell
```

4. The space complexity is in $\Theta(xnp_1)$ because of the table. The time complexity is in $O(x^2np_1)$ because each table entry requires $O(x)$ time to compute.