# Задача

Дана последовательность цен на акцию. Необходимо на всей этой последовательности найти две сделки покупка-продажа (естественно непересекающиеся по времени), дающих максимальную выгоду.

# Алгоритм

Решение выглядит так: относительно каждого элемента в списке можно найти максимальное значение купли-продажи на ценах справа от этого элемента (включая сам элемент) и слева от него.

Соответственно, пройдя по списку и получив пары правых и левых макс. выгод, дальше можно выбрать ту пару, сумма которой максимальна. Получается сложость О(n).

Однако возникает вопрос: «Как получить значение максимальной выгоды на оставшихся ценах, когда мы смотрим на элемент в середине списка, не пробегая его постоянно до конца?». Ответ: «Можно сделать два прохода, на первом проходе мы определяем все левые максимумы, а возвращаясь обратно дополняем их правыми и выясняем какая пара даёт максимальную сумму».

# Имплементация

(см. Файл CSharpMaxProfits.cs в прилогаемом солюшене)

Можно, конечно, по-банальному сделать две итерации взад-вперёд и запомнить нужные значения. Но мы не ищем лёгких путей и стремимся открывывать новые горизонты, а посему можем применить (и в этом нам помогут функуиональные возможности языков .Net) технику continuation – передача функции как парамера с целью её отложенного вызова.

Весь алгоритм реализуется на двух функциях:

* *StepForward* – рекурсивно проходит список вперёд.
* *StepBackward –* обеспечивает обратный проход по списку.

На каждом шаге *StepForward* получает минимальное из всех предыдущих просмотренных цен (*prevMinValue*), а также значение макс. выгоды (*prevLeftMaxDiff*), которая была определена на всех предыдущих ценах (слева). На основании этих знаний и текущего элемента определяется новая пара из минимальной цены с учётом текущего элемента и макс. выгоды на этом интервале. После этого происходит рекурсивный вызов *StepForward*, в который передаются вычесленные параметры, а также continuation-функция и оставшиеся элементы списка.

Сontinuation-функция – это функция, которая на обратном проходе списка, используя данные, запомненные на первом проходе, определяет пару макс. выгод с наибольшей суммой.

*StepForward* создаёт continuation используя частичное применение параметров к функции *StepBackward* (текущий элемент, макс. выгода слева и значение предыдущей continuation-функции). При этом остаются неопределёными три параметра, которые ожидает *StepBackward* (максимальная цена справа и макс. выгода справа, а также текущая максимальная сумма выгод среди всех пар просмотренных по мере обратных вызовов continuation-функций.

***Важно понимать****, что параметр* ***cont*** *функции StepForward имеет тип Func<int,int,int,int> (т.е. (int,int,int) -> int!). Mожет показаться, что мы вроде как используем функцию StepBackward у которой шесть параметров, но на самом деле в момент рекурсивного вызова StepForward мы создём новую continuation*-функцию, *применяя к StepBackward три параметра (которые запоминаются за счёт замыкания). Таким образом, при вызове эта continuation будет ожидать три int и возвращать int.*

Когда функция *StepForward* доходит до последнего элемента в списке она инициирует цепочку обратных вызовов *StepBackward*, запомненных на предыдущих вызовах *StepForward.*

Здесь происходит первый вызов continuation – просто инициализация начальных значений для последующей цепочки обратных вызовов *StepBackward*. При этом забавно (ну для меня), что при прямом проходе списка мы берём значения цен из листа, а на обратном пути эти значения берутся из замыканий в параметре *currPrice* *continuation*-функции. Это, кстати, очень удобно при использовании «true» листа, который применяется в F#, поскольку к этому моменту исходный лист уже полностью опустошён. Последняя *continuation* в цепочке обратных вызовов определяется первой ещё в момент начального вызова *StepForward* :

switch (prices.Count)

{

case 0: return 0;

**case 1: return cont(prices[0], 0, 0);**

default: …

}

return StepForward(l[0], 0, (x, y, z) => z, l.Skip(1).ToList());

Её задача, просто вернуть нужное значение результата.

Больше всего сносит крышу от рекурсивного определения *continuation*.

Для последовательности [**1, 7, 2, 3, 9**] на первом шаге у нас получается (**красным** выделены применённые на шаге переменные):

(x, y, z) => z

На втором:

(x, y, acc) => StepBackward(**7, 6**, **(x, y, z) => z,** x, y, acc)

На третьем:

(x, y, acc) => StepBackward(**2, 6, (x, y, acc) => StepBackward(7, 6, (x, y, z) => z, x, y, acc)**, x, y, acc)

На четвёртом:

(x, y, acc) => StepBackward(**3, 6**, **(x, y, acc) => StepBackward(2, 6, (x, y, acc) => StepBackward(7, 6, (x, y, z) => z, x, y, acc), x, y, acc)**, x, y, acc)

Когда же цепочка обратных вызовов разворачивается обратно, внутри *StepBackward* делается прямой вызов к самой последней *continuation* c полученными параметрами и выходит:

StepBackward(3, 6, (x, y, acc) => StepBackward(2, 6, (x, y, acc) => StepBackward(7, 6, (x, y, z) => z, x, y, acc), x, y, acc), **9, 0, 0**)

затем...

StepBackward(2, 6, (x, y, acc) => StepBackward(7, 6, (x, y, z) => z, x, y, acc), **9, 6, 12**)

потом...

StepBackward(7, 6, (x, y, z) => z, **9, 7, 13**)

и апосля всё кончается... трам-пара-пам-пам...

**9, 1, 13** -> **13**