Facultad de Ingeniería de la UBA

75.29 Teoría de Algoritmos
Trabajo Práctico N°2

Cuatrimestre: Segundo de 2014

Fecha de entrega: 17 de Noviembre de 2014

Alumnos:

Apellido y Nombre	Padrón	Email
Parnisari, María Inés	92235	maineparnisari@gmail.com
Buffevant, César	76700	buffevant@gmail.com

Enunciado

Punto 1: Perfil de la ciudad

Dada una lista de edificios representados en el plano, se debe calcular el perfil de la ciudad. Cada edificio se representa por una terna (x_{1i}, h_i, x_{2i}), donde x_{1i} representa el inicio del edificio, x_{2i} el fin del edificio y h_i la altura del mismo. El perfil se debe calcular de forma que dado un x cualquiera se vea el edificio más alto en ese punto.

Se pide:

1. Implementar un algoritmo que resuelva el problema del perfil de la ciudad, utilizando la técnica de división y conquista.
2. Calcular el orden del algoritmo implementado.

Punto 2: Problema del Inventario

Somos dueños de una concesionaria que vende camiones, y contamos con predicciones de ventas para los próximos n meses. Para simplificar, asumimos que todas las ventas mensuales y compras se producen el primer día del mes, y que los camiones que no se vendieron deben almacenarse en un depósito hasta el primer día del próximo mes.

- d_i representa la cantidad de camiones que se esperan vender en el i-esimo mes.
- El depósito tiene capacidad para almacenar hasta S camiones.
- El costo de almacenar un camión es C.
- Para comprar camiones se debe librar una orden de compra, y por cada orden librada se debe pagar una tasa fija K, independientemente de la cantidad de camiones ordenados.
- En el momento inicial, antes del inicio del primer mes, no hay ningún camión almacenado en el depósito.
- Al finalizar el mes n no debe quedar ningún camión en el depósito.

Se pide:

1. Diseñar e implementar un algoritmo por programación dinámica, que determine cuándo y por qué cantidad de camiones se deben librar cada orden de compra, con el objetivo de satisfacer la demanda mensual esperada d_i y al mismo tiempo minimizar el costo total de todas las operaciones. El orden del algoritmo debe ser polinomial en la cantidad de meses (n) y la capacidad máxima del depósito (S).
2. Especificar las ecuaciones de recurrencias, los casos bases, y como rastrear la solución a partir de la tabla.
3. Calcular el orden del algoritmo.

Metodología de Trabajo

El presente trabajo se realizó en el lenguaje de programación C#, propietario de Microsoft. El mismo puede ejecutarse bajo Windows mediante Visual Studio, o en un entorno Linux con el framework de código abierto Mono.
Para el desarrollo implementamos tests unitarios y de integración automatizados utilizando el framework NUnit.
Para el versionado del código de fuente utilizamos un repositorio GitHub privado.

Análisis y Diseño

Punto 1

Lectura del archivo

La lectura del archivo básicamente lee línea por línea el archivo entero y construye instancias de la clase **Building**. Ésta lógica es implementada por la clase **BuildingsFileReader**.

Cálculo del perfil de la ciudad

Para el cálculo del perfil de la ciudad se implementó el algoritmo de MergeSort. La implementación de MergeSort en la clase **ProfileCityCalculator** toma cada uno de los edificios y los ordena de acuerdo a la coordenada X1. La clave del algoritmo está en la parte de recombinación de cada una de las mitades ordenadas (método **MergeWithSplit**). En éste método, además de ordenar las dos mitades, se chequea que ambos edificios no se solapen. Si se solapan, dependiendo de la altura de cada uno el edificio más bajo se recorta (el edificio más alto oculta esa parte del edificio más bajo). Si un edificio es ocultado completamente por otro, el primero se remueve de la lista sin agregarlo al set ordenado.

Punto 2

Lectura del archivo

La lectura del archivo lee línea por línea el archivo de entrada y construye una instancia de la clase **InventoryData**. Ésta lógica es implementada por la clase **InventoryManagerFileReader**.

Cálculo de la matriz de costos

Este problema también es conocido como el “problema de inventarios con modelo de revisión periódica con demanda variable”, con el agregado de que la capacidad de almacenamiento es limitada.

La observación clave que hacemos para encontrar un algoritmo eficiente es la siguiente: la estrategia óptima de compra sólo comprará camiones cuando no haya ninguno en stock. Si no fuera

así, estaríamos pagando de más en concepto de costo de almacenamiento (C). Entonces, en un mes dado, hay que tomar una decisión entre dos opciones:

- No comprar camiones y utilizar el stock del mes anterior, ó
- Comprar lo más que se pueda para suplir la demanda de este mes y el que sigue.

La elección va a depender de la relación que haya entre el costo de orden y el costo de almacenamiento, además de la capacidad de stock.

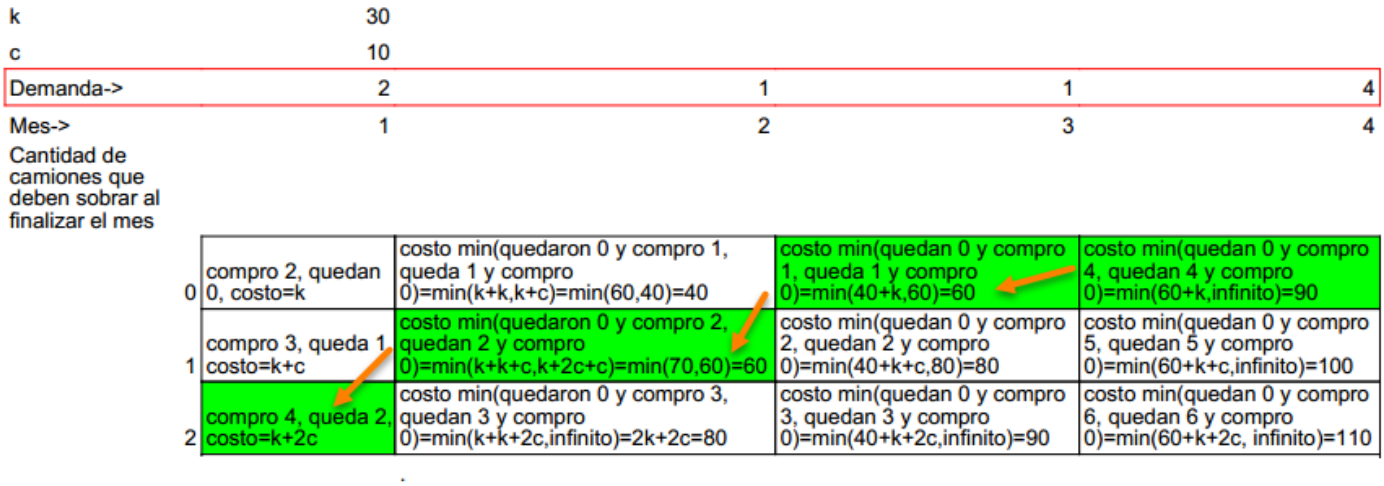
Para resolver el problema, primero se construye una matriz de s+1 filas y n columnas. Una celda (i,j) almacena el costo acumulado óptimo de satisfacer la demanda del mes j, y mantener i camiones en stock para el mes j+1.

Para completar la primera columna de la matriz (es decir, del mes 1), no hay que tomar decisiones: se compra lo suficiente para satisfacer la demanda d(1) y para mantener i camiones en stock. Para completar los meses restantes, se toma la decisión de comprar 0 camiones y abastecerse con lo del mes anterior, o comprar d(j) + i camiones, según cual alternativa sea de costo mínimo.

En esta matriz, además de ir almacenando los costos óptimos parciales acumulados, también almacenamos desde qué fila de la columna anterior llegamos a la celda actual, y cuántos camiones se compran en el mes.

El armado de esta matriz no proporciona una solución completa. Para obtener la misma, se parte de la observación de que en el último mes no deben quedar camiones en stock. Por lo tanto, luego de tener la matriz, se parte desde la celda (0, n) de la misma, y se busca el camino por el cual se llegó a esa celda. Como cada celda contiene el número de fila de la columna anterior por la cual arribamos, este camino es trivial.

La siguiente imagen muestra el armado de la matriz para un caso de ejemplo, y la búsqueda de la solución óptima en verde (partiendo desde la celda (0,4) e iterando hacia la izquierda).



k		30			
c		10			
Demanda->		2	1	1	4
Mes->		1	2	3	4
Cantidad de camiones que deben sobrar al finalizar el mes					
0	compro 2, quedan 0, costo=k	costo min(quedaron 0 y compro 1, queda 1 y compro 0)=min(k+k,k+c)=min(60,40)=40	costo min(quedan 0 y compro 1, queda 1 y compro 0)=min(40+k,60)=60	costo min(quedan 0 y compro 4, quedan 4 y compro 0)=min(60+k,infinito)=90
1	compro 3, queda 1 costo=k+c	costo min(quedaron 0 y compro 2, quedan 2 y compro 0)=min(k+k+c,k+2c+c)=min(70,60)=60	costo min(quedan 0 y compro 2, quedan 2 y compro 0)=min(40+k+c,80)=80	costo min(quedan 0 y compro 5, quedan 5 y compro 0)=min(60+k+c,infinito)=100
2	compro 4, queda 2, costo=k+2c	costo min(quedaron 0 y compro 3, quedan 3 y compro 0)=min(k+k+2c,infinito)=2k+2c=80	costo min(quedan 0 y compro 3, quedan 3 y compro 0)=min(40+k+2c,infinito)=90	costo min(quedan 0 y compro 6, quedan 6 y compro 0)=min(60+k+2c, infinito)=110

Ecuaciones de recurrencia

Sean:

- $C_i =$ costo total acumulado óptimo para el período i
- $M_{i,j} =$ costo de mantener i camiones en stock durante el mes *j* (para vender en el mes $j+1$).

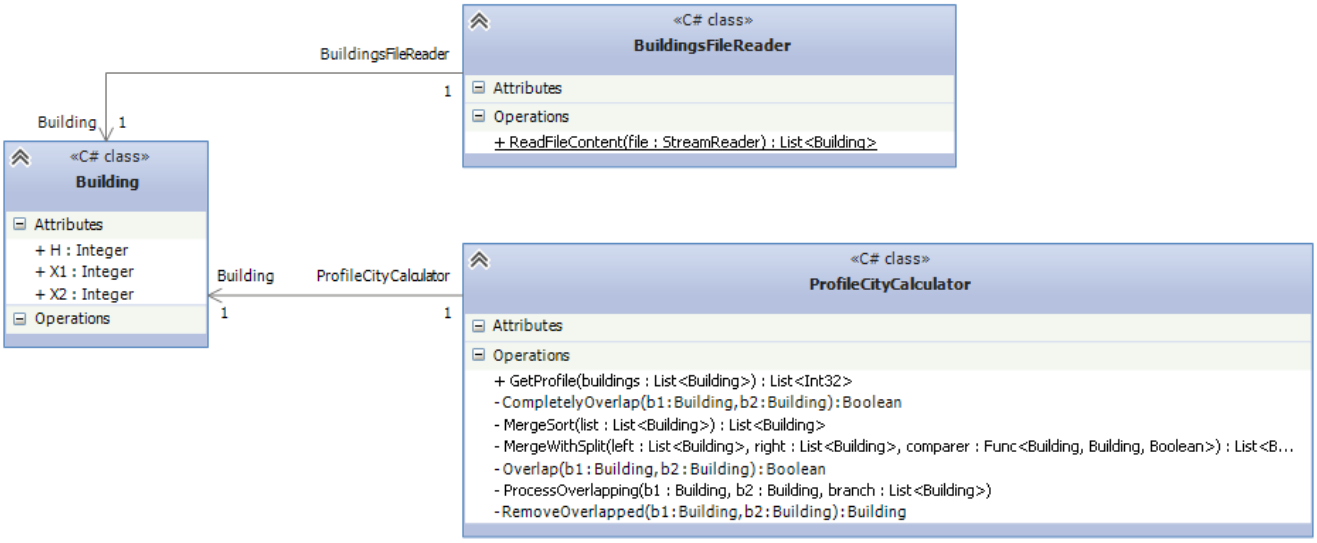
Tenemos que:

- $C_n = min[K + M_{0,n-1}, M_{d[n],n-1}]$
- $M_{i,mes} = min[M_{0,mes-1}, + K + C * i,\ M_{i+1,mes-1} + C * i]$
- $M_{i,1} = k + i * C$ (caso base)

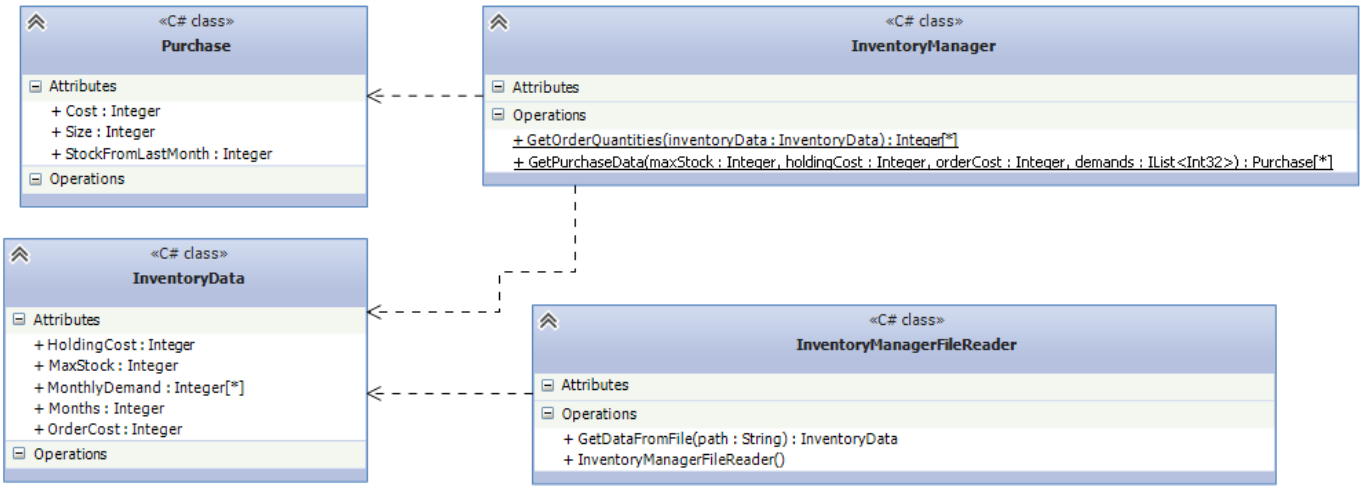
El objetivo es minimizar C_n.

Diagrama de Clases

Punto 1



Punto 2



Calculo de Ordenes

Punto 1

Lectura del archivo

La lectura del archivo es secuencial, se lee de una línea así que el orden es O(N), donde N es la cantidad de edificios.

Cálculo del perfil de la ciudad

El orden de la implementación del MergeSort es O(NLogN), ya que la recombinación sigue siendo de orden N. Luego, la lista ordenada se recorre una vez más para generar la lista de enteros requerida como output del programa, en orden O(N).

Punto 2

Lectura del archivo

La lectura del archivo es secuencial y el orden es O(N), donde N es la cantidad de meses a programar.

Cálculo del costo óptimo

La creación de la matriz de costos, de tamaño n * s, implica realizar una cantidad constante de operaciones para celda: calcular el mínimo entre dos valores. Por lo tanto, el armado de la matriz es de orden O(NS). Para obtener la solución óptima, se visita una celda por cada columna, por lo que este paso es de orden O(N). En total, como $O(NS) \geq O(N)$, el orden final es $O(NS)$.

Código Fuente

Main.cs

```
namespace ConsoleApplication
{
    using System;
    using System.Configuration;
    using System.Diagnostics;
    using System.IO;
    using System.Text;
    using TP2.CityProfile;
    using TP2.InventoryManager;

    public class MainClass
    {
        public static void Main()
        {
            var stopwatch = new Stopwatch();

            var logger = new Logger(ConfigurationManager.AppSettings["logFileName"]);
            try
            {
                // Punto 1
                logger.Log("Ingrese el nombre del archivo para el punto 1: ");
                var fileName = System.Console.ReadLine();
                stopwatch.Start();
                using (var stream = File.OpenRead(fileName))
                {
                    var buildings = BuildingsFileReader.ReadFileContent(new
StreamReader(stream));
                    var profileCityCalculator = new ProfileCityCalculator();
                    var profile = profileCityCalculator.GetProfile(buildings);
                    logger.Log("City Profile...");
                    logger.Log(BuildProfile(profile));
                    logger.Log(profile[profile.Count - 1].ToString());
                }
                stopwatch.Stop();

                // Punto 2
                logger.Log("Ingrese el nombre del archivo para el punto 2: ");
                fileName = System.Console.ReadLine();
                stopwatch.Start();
                var reader = new InventoryManagerFileReader();
                var inputData =
reader.GetDataFromFile(Path.Combine(Environment.CurrentDirectory, fileName));
```

```
                int[] orderQuantities = InventoryManager.GetOrderQuantities(inputData);
                for (int month = 0; month < inputData.Months; month++)
                {
                    logger.Log(string.Format("Mes {0}: comprar {1}", month,
orderQuantities[month]));
                }
                stopwatch.Stop();

            }
            catch (System.Exception)
            {
                logger.Log("Error al leer el archivo de entrada.");
            }
            finally
            {
                stopwatch.Stop();
                logger.Log("Tiempo de ejecucion total (en segundos): " +
stopwatch.Elapsed.TotalSeconds);
                logger.Log("Presione una tecla para finalizar...");
                logger.Dispose();
                System.Console.ReadLine();
            }
        }

        private static string BuildProfile(System.Collections.Generic.List<int> profile)
        {
            var result = new StringBuilder();
            for (var i = 0; i < profile.Count - 1; i++)
            {
                result.Append(profile[i].ToString());
                result.Append(",");
            }
            return result.ToString();
        }
    }
}
```

Punto 1

Building.cs

```
namespace TP2.CityProfile
{
    public class Building
    {
        public Building(int x1, int x2, int h)
        {
            this.X1 = x1;
            this.X2 = x2;
            this.H = h;
        }

        public int X1
        {
            get; set;
        }

        public int X2
        {
            get; set;
        }

        public int H
        {
            get; set;
        }
    }
}
```

BuildingsFileReader.cs

```
namespace TP2.CityProfile
{
    using System;
    using System.Collections.Generic;
    using System.IO;

    public class BuildingsFileReader
    {
        public static List<Building> ReadFileContent(StreamReader file)
        {
            var buildings = new List<Building>();
            while (!file.EndOfStream)
            {
                var line = file.ReadLine();
                var parts = line.Split(',');
                if (parts.Length == 3)
                {
                    buildings.Add(new Building(Int32.Parse(parts[0]), Int32.Parse(parts[2]),
Int32.Parse(parts[1])));
                }
            }
            return buildings;
        }
    }
}
```

ProfileCityCalculator.cs

```
namespace TP2.CityProfile
{
    using System;
    using System.Collections.Generic;
    using System.Linq;

    public class ProfileCityCalculator
    {
        /// <summary>
        /// Devuelve el profile de la ciudad.
        /// </summary>
        /// <param name="buildings"></param>
        /// <returns></returns>
        public List<int> GetProfile(List<Building> buildings)
        {
            var result = new List<int>();
            //O(NLogN)
            var orderedBuildings = MergeSort(buildings);

            // crea la lista con el profile - O(N)
            foreach (var b in orderedBuildings)
            {
                result.Add(b.X1);
                result.Add(b.H);
            }
            result.Add(orderedBuildings.Last().X2);
            return result;
        }

        /// <summary>
        /// Mergesort
        /// </summary>
        /// <param name="list">Lista a ordenar</param>
        /// <returns>La lista ordenada</returns>
        private List<Building> MergeSort(List<Building> list)
        {
            if (list.Count <= 1)
                return list;

            var middle = list.Count() / 2;
            var h1 = MergeSort(list.GetRange(0, middle));
            var h2 = MergeSort(list.GetRange(middle, list.Count - middle));
            return MergeWithSplit(h1, h2, (a, b) => a.X1 <= b.X1);
        }

        /// <summary>
```

```
        /// Devuelve True is b1 y b2 están parcialmente solapados.
        /// </summary>
        /// <param name="b1">Edificio 1</param>
        /// <param name="b2">Edificio 2</param>
        /// <returns>Devuelve True is b1 y b2 están parcialmente solapados. Sino,
False</returns>
        private bool Overlap(Building b1, Building b2)
        {
            return !(b1.X2 <= b2.X1 || b2.X2 <= b1.X1);
        }

        /// <summary>
        /// Devuelve True is b2 está completamente solapado a b1.
        /// </summary>
        /// <param name="b1">Edificio 1</param>
        /// <param name="b2">Edificio 2</param>
        /// <returns>Devuelve True si b2 está completamente solapado a b1.</returns>
        private bool CompletelyOverlap(Building b1, Building b2)
        {
            return (b1.X1 <= b2.X1 && b1.X2 >= b2.X2 && b1.H > b2.H);
        }

        /// <summary>
        /// Éste metodo remueve la intersección entre los dos edificios teniendo en cuenta la
altura.
        /// </summary>
        /// <param name="b1">Edificio 1</param>
        /// <param name="b2">Edificio 2</param>
        /// <returns>Devuelve el remanente luego del corte si lo hay. Sino null.</returns>
        private Building RemoveOverlapped(Building b1, Building b2)
        {
            var a = b1.H > b2.H ? b1 : b2;
            var b = a == b1 ? b2 : b1;

            if (a.X1 <= b.X1 && a.X2 >= b.X2)
            {
                return null;
            }

            if (a.X1 < b.X2 && b.X2 < a.X2)
            {
                b.X2 = a.X1;
                return null;
            }

            if (a.X1 < b.X1 && b.X1 < a.X2)
            {
                b.X1 = a.X2;
```

```
                return null;
            }

            if (b.X1 <= a.X1 && b.X2 >= a.X2)
            {
                var oldX2 = b.X2;
                b.X2 = a.X1;
                return new Building(a.X2, oldX2, b.H);
            }

            return null; // we should never reach this point
        }

        /// <summary>
        /// Verifica si hay solapamiento entre los dos edificios y
        /// lo elimina si es el caso.
        /// </summary>
        /// <param name="b1">Edificio 1</param>
        /// <param name="b2">Edificio 2</param>
        /// <param name="branch">Rama actual del mergesort.</param>
        private void ProcessOverlapping(Building b1, Building b2, List<Building> branch)
        {
            if (Overlap(b1, b2))
            {
                var remaining = RemoveOverlapped(b1, b2);
                if (remaining != null)
                {
                    branch.Add(remaining);
                }
            }
        }

        /// <summary>
        /// Fase de merge del mergesort que también remueve las interseccciones entre los
edificios.
        /// </summary>
        /// <param name="left">Rama izquierda</param>
        /// <param name="right">Rama derecha</param>
        /// <param name="comparer">Delegate para la comparación entre los edificios.</param>
        /// <returns>Devuelve la lista ordenada.</returns>
        private List<Building> MergeWithSplit(List<Building> left, List<Building> right,
Func<Building, Building, bool> comparer)
        {
            var result = new List<Building>(1000);
            while (left.Count > 0 || right.Count > 0)
            {
                if (left.Count > 0 && right.Count > 0)
                {
```

```
                var l = left.First();
                var r = right.First();
                if (comparer(l, r))
                {
                    // if r is completely overlapped by l, just remove it from the list.
                    if (CompletelyOverlap(l, r))
                    {
                        right = right.GetRange(1, right.Count - 1); // O(N)
                        continue;
                    }
                    // remove overlapped section
                    ProcessOverlapping(l, r, left);
                    // add it the result list
                    result.Add(l);
                    left = left.GetRange(1, left.Count - 1); // O(N)
                }
                else
                {
                    // if l is completely overlapped by r, just remove it from the list.
                    if (CompletelyOverlap(r, l))
                    {
                        left = left.GetRange(1, right.Count - 1); // O(N)
                        continue;
                    }
                    // remove overlapped section
                    ProcessOverlapping(r, l, right);
                    // add it the result list
                    result.Add(r);
                    right = right.GetRange(1, right.Count - 1); // O(N)
                }
            }
            else if (left.Any())
            {
                // add the remaining of the list.
                result.Add(left.First());
                left = left.GetRange(1, left.Count - 1); // O(N)
            }
            else if (right.Any())
            {
                // add the remaining of the list.
                result.Add(right.First());
                right = right.GetRange(1, right.Count - 1); // O(N)
            }
        }
        return result;
    }
}
```

```
}
```

CityProfileTest.cs

```
namespace TP2.Test
{
    using NUnit.Framework;
    using System.Collections.Generic;
    using TP2.CityProfile;

    [TestFixture()]
    public class CityProfileTest
    {
        [Test()]
        public void ShouldBuildCorrectProfile()
        {
            // arrange
            var buildings = new List<Building>
                    {
                            new Building(3, 8, 6),
                            new Building(0, 5, 4),
                            new Building(6, 10, 10),
                            new Building(1, 4, 7),
                            new Building(9, 11, 11),
                            new Building(7, 12, 8)
                    };
            var profileCityCalculator = new ProfileCityCalculator();

            // act
            var actualProfile = profileCityCalculator.GetProfile(buildings);

            // assert
            var expectedProfile = new List<int>
            {
                0, 4, 1, 7, 4, 6, 6, 10, 9, 11, 11, 8, 12
            };

            Assert.AreEqual(expectedProfile, actualProfile);
        }
    }
}
```

Punto 2

InventoryData.cs

```
namespace TP2.InventoryManager
{
    using System.Linq;

    public class InventoryData
    {
        public int Months { get; set; }
        public int MaxStock { get; set; }
        public int HoldingCost { get; set; }
        public int OrderCost { get; set; }
        public int[] MonthlyDemand { get; set; }

        public override bool Equals(object obj)
        {
            var isInventory = obj is InventoryData;
            if (isInventory)
            {
                InventoryData other = (InventoryData)obj;
                return this.MonthlyDemand.SequenceEqual(other.MonthlyDemand) &&
                       this.Months == other.Months &&
                       this.OrderCost == other.OrderCost &&
                       this.HoldingCost == other.HoldingCost &&
                       this.MaxStock == other.MaxStock;
            }
            return false;
        }
    }
}
```

Purchase.cs

```
namespace TP2.InventoryManager
{
    using System.Diagnostics;

    [DebuggerDisplay("Buy {Size} at ${Cost}. Prev stock {StockFromLastMonth}")]
    public class Purchase
    {
        public int Cost { get; set; }

        public int StockFromLastMonth { get; set; }

        public int Size { get; set; }
    }
}
```

InventoryManagerFileReader.cs

```
namespace TP2.InventoryManager
{
    using System;
    using System.IO;
    using System.Linq;

    public class InventoryManagerFileReader
    {
        private const int VARIABLES = 5;

        public InventoryData GetDataFromFile(string path)
        {
            var inventoryData = new InventoryData();

            string[] lines = File.ReadAllLines(path);
            inventoryData.Months = Convert.ToInt32(lines[0]);

            if (lines.Count() != VARIABLES + inventoryData.Months)
            {
                throw new Exception("Invalid file format");
            }

            inventoryData.MaxStock = Convert.ToInt32(lines[1]);
            inventoryData.HoldingCost = Convert.ToInt32(lines[2]);
            inventoryData.OrderCost = Convert.ToInt32(lines[3]);

            inventoryData.MonthlyDemand = new int[inventoryData.Months];

            for (int i = VARIABLES; i < inventoryData.Months + VARIABLES; i++)
            {
                inventoryData.MonthlyDemand[i - VARIABLES] = Convert.ToInt32(lines[i]);
            }

            return inventoryData;
        }
    }
}
```

InventoryManager.cs

```
namespace TP2.InventoryManager
{
    using System;
    using System.Collections.Generic;

    public static class InventoryManager
    {
        public static int[] GetOrderQuantities(InventoryData inventoryData)
        {
            Purchase[,] matrix = GetPurchaseData(
                inventoryData.MaxStock,
                inventoryData.HoldingCost,
                inventoryData.OrderCost,
                inventoryData.MonthlyDemand);

            int[] orderQuantities = new int[inventoryData.Months];

            // Start from the last month and work backwards.
            // Take the first row because we're not allowed to keep stock
            // after the last month
            Purchase currentMonthPurchase = matrix[0, inventoryData.Months - 1];

            int currentMonth = inventoryData.Months - 1;

            while (currentMonth > 0)
            {
                orderQuantities[currentMonth] = currentMonthPurchase.Size;
                currentMonthPurchase = matrix[currentMonthPurchase.StockFromLastMonth,
currentMonth - 1];
                currentMonth--;
            }

            orderQuantities[0] = currentMonthPurchase.Size;

            return orderQuantities;
        }

        public static Purchase[,] GetPurchaseData(int maxStock, int holdingCost, int
orderCost, IList<int> demands)
        {
            Purchase[,] matrix = new Purchase[maxStock + 1, demands.Count];

            // initalize first column (month 1)
            for (int row = 0; row <= maxStock; row++)
            {
                matrix[row, 0] = new Purchase
```

```
                {
                    Cost = orderCost + row * holdingCost,
                    StockFromLastMonth = -1,
                    Size = demands[0] + row
                };
            }

            // complete rest of the months
            for (int month = 1; month < demands.Count; month++)
            {
                // the row is the amount we want to keep in stock for next month
                for (int row = 0; row <= maxStock; row++)
                {
                    int amountToBuyNow = demands[month] + row;
                    int costBuyNow = matrix[0, month - 1].Cost + orderCost + holdingCost *
row;
                    int costDontBuyNow = int.MaxValue;

                    // if we have stock from the past month
                    // we can skip buying now
                    if (amountToBuyNow <= maxStock)
                    {
                        costDontBuyNow = matrix[row + 1, month - 1].Cost + holdingCost * row;
                    }

                    var cost = Math.Min(costBuyNow, costDontBuyNow);
                    var size = (cost != costBuyNow) ? 0 : amountToBuyNow;
                    var stockFromLastMonth = (size != 0) ? 0 : row + 1;

                    matrix[row, month] = new Purchase
                    {
                        Cost = cost,
                        Size = size,
                        StockFromLastMonth = stockFromLastMonth
                    };

                }
            }
            return matrix;
        }
    }
}
```

InventoryManagerTest.cs

```
namespace TP2.Test
{
    using NUnit.Framework;
    using TP2.InventoryManager;

    [TestFixture]
    public class InventoryManagerTest
    {
        [Test]
        public void ShouldBuildCorrectCostsMatrix()
        {
            // arrange
            var data = new InventoryData
            {
                MonthlyDemand = new[] { 2, 1, 1, 4 },
                HoldingCost = 10,
                MaxStock = 2,
                Months = 4,
                OrderCost = 30
            };

            // act
            var matrix = InventoryManager.GetPurchaseData(data.MaxStock, data.HoldingCost,
data.OrderCost, data.MonthlyDemand);

            // assert
            var expectedSizes = new[,]
            {
                {2, 0, 0, 4},
                {3, 0, 2, 5},
                {4, 3, 3, 6}
            };

            var expectedCosts = new[,]
            {
                {30, 40, 60, 90},
                {40, 60, 80, 100},
                {50, 80, 90, 110}
            };

            for (int row = 0; row <= data.MaxStock; row++)
            {
                for (int col = 0; col < data.Months; col++)
                {
                    Assert.AreEqual(expectedSizes[row, col], matrix[row, col].Size, "Size is
wrong for [" + row + "][" + col + "]");
```

```
                    Assert.AreEqual(expectedCosts[row, col], matrix[row, col].Cost, "Cost is
wrong for [" + row + "][" + col + "]");
                }
            }
        }

        [Test]
        public void ShouldGetOptimalSolution()
        {
            // arrange
            var data = new InventoryData
            {
                MonthlyDemand = new[] { 2, 1, 1, 4 },
                HoldingCost = 10,
                MaxStock = 2,
                Months = 4,
                OrderCost = 30
            };

            // act
            int[] optimalSolution = InventoryManager.GetOrderQuantities(data);

            // assert
            var expectedOptimalSolution = new[]
            {
                4, 0, 0, 4
            };

            CollectionAssert.AreEqual(expectedOptimalSolution, optimalSolution);
        }

        [Test]
        public void ShouldGetOptimalSolutionWhenOrderCostIsLowerThanHoldingCost()
        {
            // arrange
            var data = new InventoryData
            {
                MonthlyDemand = new[] { 10, 12, 9 },
                HoldingCost = 4,
                MaxStock = 20,
                Months = 3,
                OrderCost = 2
            };

            // act
            int[] optimalSolution = InventoryManager.GetOrderQuantities(data);

            // assert
```

```
            var expectedOptimalSolution = new[]
            {
                10, 12, 9
            };

            CollectionAssert.AreEqual(expectedOptimalSolution, optimalSolution);
        }
    }
}
```