Segundo Parcial de Programación 3 (2/12/2008)

Instituto de Computación, Facultad de Ingeniería

- Este parcial dura 4:00 horas y contiene 5 carillas. El total de puntos es **60**.
- En los enunciados llamamos C^* a la extensión de C al que se agrega el operador de pasaje por referencia &, y las sentencias *new*, *delete* y el uso de *cout* y *cin*.
- NO se puede utilizar ningún tipo de material de consulta. Salvo que se indique lo contrario, puede usarse todo lo visto en el teórico, práctico y laboratorio sin demostrarlo, indicando claramente lo que se está usando.

Se requiere:

i. Numerar todas las hojas e incluir en cada una el nombre y la cédula de identidad.
ii. Utilizar las hojas de un sólo lado y escribir con lápiz.
iii. Iniciar cada ejercicio en hoja nueva.
iv. Poner en la carátula la cantidad de hojas entregadas, y un índice indicando en qué hojas respondieron cada problema.

Ejercicio 1. (20 puntos)

a) **(10 puntos)** Un comerciante romano ha conseguido un lote de *n* animales (identificados por un valor en el rango *0..n-1*) para ser embarcados en uno de sus barcos a los efectos de ser vendidos en otra ciudad. De cada animal dispone de un único ejemplar.

El comerciante conoce el peso, en kilos, de cada animal y sabe que el barco puede transportar un peso máximo de G kilos sin hundirse.

El comerciante piensa vender cada animal a un precio de d_i $(0 \leq i < n)$ denarios y desea saber cual es la mayor cantidad de denarios que puede obtener por la venta de los animales que podría transportar en su barco sin que éste se hunda.

Dar una fórmula **recursiva** para solucionar el problema. Llame a la fórmula recursiva *f*.

b) **(10 puntos)** Resuelva la siguiente recurrencia:

$$\begin{cases} T(1)=1 \\ T(n)=T(n/2)+n+\log_2(n) \quad con \quad n>1 \end{cases}$$

En caso de suponer *n* como potencia de una base conveniente, se debe indicar la misma.

Recuerde que:

- $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$
- *log(a.b) = log(a) + log(b)*

Ejercicio 2. (13 puntos)

Considere el tipo *Elemento* como predefinido con las operaciones usuales de comparación.
Dada una secuencia *A* de *N* elementos a ordenar, considere el algoritmo de ordenación de la Burbuja (BubbleSort). La estrategia se basa en comparar elementos consecutivos de la secuencia y desplazar el mayor hacia la derecha. Por ejemplo, usando notación de arreglos para la secuencia, se efectúa en primer lugar la comparación de A[1] con A[2], y suponiendo que A[1] < A[2] continúa con el siguiente paso: compara entonces A[2] con A[3], en caso contrario intercambia A[1] por A[2] para luego seguir con la comparación de A[2] con A[3](con los elementos ya cambiados) y así sucesivamente.
También puede utilizarse la misma estrategia pero de atrás hacia delante, de acuerdo al siguiente seudocódigo:

```
//A: entrada de elementos a ordenar
//N: tamaño de A

BubbleSort(arreglo A, int N)
      n_elem = N;
      mientras n_elem > 1 :
            para i de 1 a n_elem-1:
                  si A[i] > A[i+1]:
                        intercambiar A[i] y A[i+1];
            n_elem--;
```

a. **(10 puntos)** Considerando la entrada ***A* = *[a, b, c],*** dar el árbol de decisión del algoritmo indicando en cada paso el estado de la entrada.
Explique el significado de los distintos nodos del árbol. En particular para este árbol indique la validez de lo representado por cada una de sus hojas.

b. **(3 puntos)**
   i. Defina *estabilidad* de un algoritmo de ordenación.

   ii. Teniendo en cuenta el algoritmo anterior de *BubbleSort*, indique si el mismo es *estable* o no. Justifique.

c. **(4 puntos)** Calcule el costo $T_W(N)$ (peor caso) y $T_A(N)$ (caso promedio) de *BubbleSort* en función de la *cantidad de comparaciones de Elementos*. Indique cual es el (los) peor caso. Justifique sus respuestas.

d. **(3 puntos)** Considerando el ***problema de ordenación*** defina optimalidad de un algoritmo (para el peor caso y caso medio). Indique y justifique si *BubbleSort* es óptimo en algún caso.

Ejercicio 3. (13 puntos).

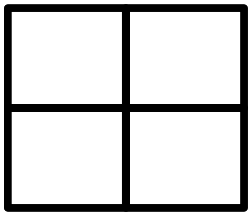
Considere el siguiente juego, en el cual se dispone de:

- Un tablero cuadrado de tamaño $N \times N$
- M bloques (con $M \geq N \times N$), cada bloque está subdividido en cuatro partes y cada una de las partes tiene un número mayor que cero.

Los bloques se insertan en el tablero (***en cualquiera de las cuatro rotaciones posibles***) cumpliendo con la restricción de que los números de las zonas de contacto entre los bloques adyacentes (por sus lados) debe coincidir. Por cada una de estas coincidencias se suma el número al puntaje obtenido por el jugador hasta el momento.

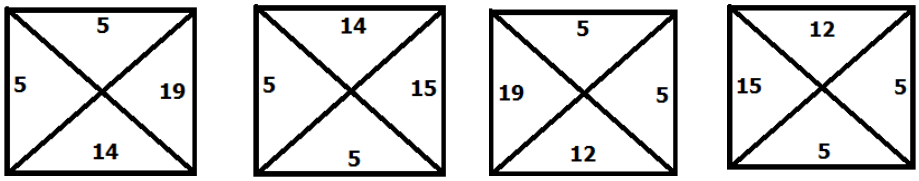
El objetivo del juego es disponer en el tablero $N \times N$ bloques, tomados de entre los M disponibles y sin repetir bloques (es decir, un bloque sólo puede ser utilizado una vez, independientemente de la rotación), de forma que el puntaje obtenido por el jugador sea menor que un valor dado P, pero lo más cercano posible al mismo.

Por ejemplo, dados $N = 2$, $M = 4$ y $P = 61$

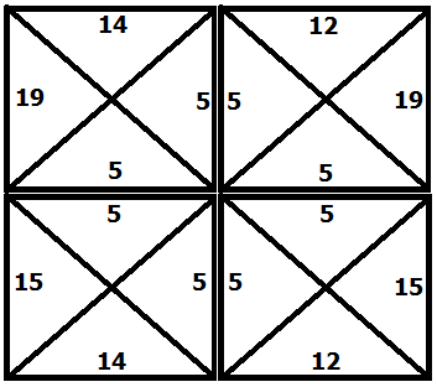


tablero de 2*2

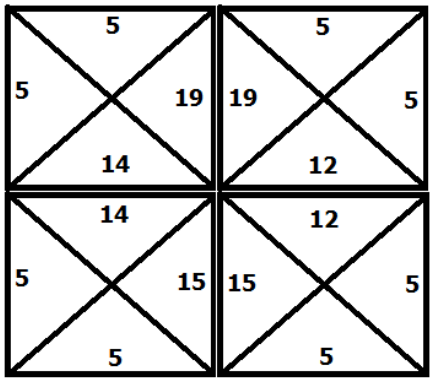
y los bloques:



una posible disposición de los bloques, con un puntaje de 20 es:



una mejor solución, dado que el puntaje es menor que P pero más cercano al mismo, con un puntaje de 60 es:



Se pide:

a) (**10 puntos**) Formalizar el problema en términos de ***Backtracking***. Indicar: forma de la solución, restricciones explícitas e implícitas, función objetivo y predicados de poda.

Nota

- La casilla ubicada en la fila ***i*** y columna ***j***, $\forall\ i,j\ 0 \leq i,j \leq N-1$ de un tablero de tamaño $N \times N$ corresponde a la posición $i \times N + j$ cuando se representa el tablero en forma lineal. Análogamente la posición ***k*** de la representación lineal corresponde a la casilla ubicada en la fila $k\ div\ N$ y la columna $k\ mod\ N$ del tablero.

b) (**10 puntos**) Implemente en C* la función:

```
Tablero* Juego (int N, int P, ListaBloque* lista);
```

utilizando ***Backtracking***.

En caso de que no exista solución al problema, debe retornarse el tablero de tamaño $N \times N$ vacío. En caso contrario, se deberá retornar el tablero solución. En caso de existir más de un tablero en las condiciones pedidas puede retornar cualquiera.

Identifique e indique las porciones de código que se corresponden con las diferentes partes de la formalización de la parte a).

Nota:

- **No se corregirá la parte b**) si no se realizó **satisfactoriamente** la parte a)

TADs auxiliares:

```
// TAD bloque

// Marca el bloque como usado en el tablero
void BloqueMarcar (Bloque* &bloque);
// Devuelve verdadero si el bloque esta marcado y falso en caso contrario
bool BloqueEstaMarcado (Bloque* bloque);
// Desmarca el bloque
void BloqueDesmarcar (Bloque* &bloque);
// Realiza una rotación hacia la izquierda del bloque
void BloqueRotarIzq (Bloque* &bloque);
```

```
// TAD ListaBloque

// Retorna verdadero si la lista de bloques es vacía y falso en caso
// contrario
bool ListaBloqueEsVacia (ListaBloque* lista);
// Retorna el primer bloque de la lista de bloques
// Precondición:!EsVaciaLista(lista)
Bloque* ListaBloqueHead (ListaBloque* lista);
// Retorna la lista de bloques sin el primer bloque
// Precondición:!EsVaciaListaBloque(lista)
ListaBloque* ListaBloqueTail (ListaBloque* lista);
```

```
// Tad Tablero

// Crea un tablero vacío con la dimensión indicada
Tablero* TableroCrear (int dimension);
// Inserta el bloque en el tablero, en la casilla más a la izquierda de
// la primer fila con casillas disponibles.
// El tablero hace un alias del bloque
// Precondición: el tablero tiene al menos una casilla libre.
void TableroInsertarBloque(Tablero* tab, Bloque* bloque);
// Quita el bloque del tablero, de la casilla más a la izquierda de
// la última fila ocupada
// Precondición: se realizó un llamada previa a
// TableroInsertarBloque(Tablero tab, Bloque bloque)
void TableroQuitarBloque(Tablero* tab);
// Retorna verdadero si todas las casillas tienen bloques y falso en
// caso contrario
bool TableroLleno (Tablero* tab);
// Verifica que los bloques contenidos en el tablero cumplen que los
// números de las zonas de contacto entre los bloques adyacentes (por sus
// lados) coincide
bool TableroCumpleAdyacencia(Tablero* tab);
// Retorna el puntaje del tablero
// TableroEstaLleno (Tablero tab) no tiene porque ser verdadero
int TableroPuntaje(Tablero* tab);
// Retorna una copia del tablero en profundidad sin compartir memoria
Tablero* TableroCopia (Tablero* tab);
// Destruye el tablero y también todos los bloques contenidos en el mismo
void TableroDestruir (Tablero* &tab);
```