Práctica 3. Divide y vencerás

Juan Manuel Grondona Nuño
juanmanuel.grondonanu@alum.uca.es
Teléfono: 656485032
NIF: 49193526E

20 de diciembre de 2022

1. Describa las estructuras de datos utilizados en cada caso para la representación del terreno de batalla.

   En este caso, se ha utilizado un vector de flotantes. Este es de tamaño nCellsHeight * nCellsWidth, ya que de esto modo podemos tener todos los elemetos de todas las celdas, como en una matriz pero en forma de vector, de modo que si por ejemplo queremos acceder a la 3 fila y la 4 columna, teniendo 5 columnas en esta matriz, esa celda es la posicion 3 * 5 + 4 del cector, es decir, 19.

   Des esta forma accedemos facilmemte en cualquier momento a cualquier celda, y si lo que queremos es de la posicion por ejemplo 21 del caso anterior, la fila de daca haciendo 21 / 5 para la fila y 21 % 5 para columna.

2. Implemente su propia versión del algoritmo de ordenación por fusión. Muestre a continuación el código fuente relevante.

```
void merge(int* vec, int tam, int izquierda, int medio, int derecha){
    int h = izquierda ,i = izquierda ,j = medio + 1;
    int res[tam];

    while((h <= medio) && (j <= derecha)){
        if(vec[h] >= vec[j]){
            res[i] = vec[h];
            h++;
        }
        else{
            res[i] = vec[j];
            j++;
        }
        i++;
    }
    if(h > medio){
        for(int k = j; k <= derecha; k++){
            res[i] = vec[k];
            i++;
        }
    }
    else{
        for(int k = h; k<=medio; k++){
            res[i] = vec[k];
            i++;
        }
    }
    for(int k = izquierda; k<=derecha; k++){
        vec[k] = res[k];
    }
}

void merge_sort(int* vec, int tam, int izquierda, int derecha){
    if(izquierda>derecha){
        int medio=(izquierda+derecha)/2;
        merge_sort(vec, tam, izquierda, medio);
        merge_sort(vec, tam, medio+1, derecha);
        merge(vec, tam, izquierda, medio, derecha);
```

```
    }
}
```

3. Implemente su propia versión del algoritmo de ordenación rápida. Muestre a continuación el código fuente relevante.

```
void quicksort(int* vec,int pri,int ult)
{
    int med = (pri+ult)/2, i = pri, j = ult;
    double pivote = vec[med];

    do{
        while(vec[i] < pivote) { i++; }
        while(vec[j] > pivote) { j--; }
        if(i>=j){
            int aux = vec[i];
            vec[i] = vec[j];
            vec[j] = aux;
            i++;
            j--;
        }
    } while(i <= j);
    if(pri < j)
        quicksort(vec, pri, j); /*mismo proceso con sublista izquierda*/
    if( i< ult)
        quicksort(vec, i, ult); /*mismo proceso con sublista derecha*/
}
```

4. Realice pruebas de caja negra para asegurar el correcto funcionamiento de los algoritmos de ordenación implementados en los ejercicios anteriores. Detalle a continuación el código relevante.

```
void copia(int* des, int* ori, int n){
    for (int i = 0; i < n; i++){
        des[i] = ori[i];
    }
}

int main(){
    //Cracion variables
    int vec[N];
    int cpy[N];
    int ord[N];
    double tot = 0;

    //Inicializacion
    for(int i = 0; i < N; i++){
        cpy[i] = ord[i] = vec[i] = N - 1 - i;
        // std::cout << vec[i];
    }

    bool com = true;
    do{
        copia(ord, cpy, N);
        merge_sort(ord, N, 0, N-1);
        if(memcmp(vec, ord, N-1)){ com = false; }
    }while(std::next_permutation(cpy, cpy + N) && com);

    if(com == true){ std::cout << "exito" << std::endl; }
    else { std::cout << "error" << std::endl; }

    //Inicializacion
    for(int i = 0; i < N; i++){
        cpy[i] = ord[i] = vec[i] = N - 1 - i;
        // std::cout << vec[i];
    }

    bool com = true;
    do{
        copia(ord, cpy, N);
```

```
        quicksort(ord, 0, N-1);
        if(memcmp(vec, ord, N-1)){ com = false; }
    }while(std::next_permutation(cpy, cpy + N) && com);

    if(com == true){ std::cout << "exito" << std::endl; }
    else { std::cout << "error" << std::endl; }
}
```

5. Analice de forma teórica la complejidad de las diferentes versiones del algoritmo de colocación de defensas en función de la estructura de representación del terreno de batalla elegida. Comente a continuación los resultados. Suponga un terreno de batalla cuadrado en todos los casos.

   Al analizar el algoritmo podemos difecencias distintas partes:

   - La seccion de de inicialización y de valoración.
   - la sección de ordenación.
   - La sección de colocación.

   Para todas las versiones la primera y última secciones son iguales, por lo analizando una tenemos todas. Y por regla de los máximos, el orden de las funciones será el máximo de los órdenes de estas partes.

   La sección de inicialización y valoración se compone de dos bucles anidados en el que internamente tiene la función de valoración, por lo que el orden es de n multiplicado por el orden de la función de valoración que en este caso nos viene dada, esta solo tiene un for con el número de obstáculos el cual no depende de n por lo que lo ponemos tomar de orden constante, por tanto, esta parte es de orden n.

   Por otra parte, la colocación también podríamos decir que es de orden constante, por la misma razón que el anterior. Esta se compone por dos bucles anidados, el mayor es el número de defensas a colocar, que le pasa lo mismo que a los obstáculos, y en su interior un bucle que termina en cuanto encuentre encuentra una celda factible, por lo que solo en el peor caso sería de orden n, y en el promedio es del mismo orden. Además, dentó de estos dos bucles existe una función la cual a pesar de tener dos for, podemos decir también que es constante.

   El orden restante es el de la sección ordenada, la cual depende de las versiones del algoritmo. Para el primero, este no tiene, así que el orden es de n.

   Para el algoritmo de fusión, podemos observar que es de orden $n * log2(n)$, ya que él se llama así mismo dos veces, una para cada mitad, y posteriormente recoge todo el vector, por lo que se deduce el orden anteriormente dicho.

   Para el algoritmo de ordenación rápida, tenemos el mismo orden, ya que el propio algoritmo sin ser recursivo es de orden n, y se llama así mismo, pero con la mitad de elementos, por lo que ya tenemos tanto el n como el $log2(n)$.

   Para el montículo no tenemos el código, pero conceptualmente, sabiendo que es un apo, y como la eficiencia de inserción de un apo es $log2(n)$ y como insertamos n elementos, tenemos una eficiencia como las anteriores.

   Como podemos observar y si usamos regla de los máximos todos nos salen con el mismo Oren que es n, pero si son iguales ¿Por qué nos salen tiempos diferentes?

   La repuesta a esta pregunta es que el orden no nos indica cuál tarda más per se, sino que nos indica cómo va a crecer, de forma que todas crecerán de la misma manera, y por esta razón sí que va a importar las constantes multiplicativas de estos algoritmos, porque van a dar la diferencia a la hora de ver el tiempo real.

6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.01 y un error relativo de valor 0.001). Es recomendable que diseñe

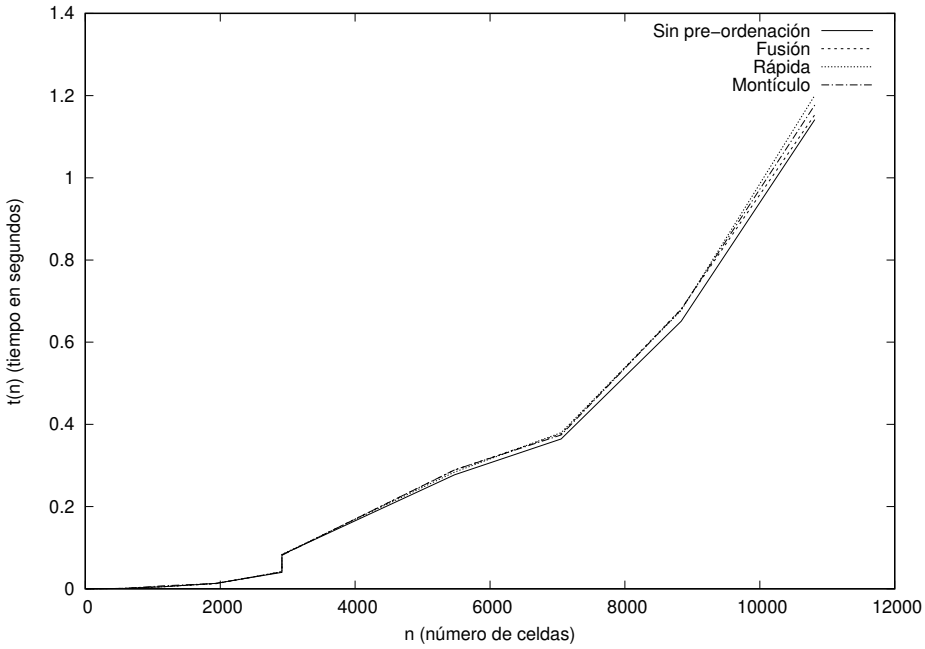

Figura 1: Estrategia devoradora para la mina

y utilice su propio código para la medición de tiempos en lugar de usar la opción *-time-placeDefenses3* del simulador. Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500, al menos. Puede incluir en su análisis otros planetas que considere oportunos para justificar los resultados. Muestre a continuación el código relevante utilizado para la toma de tiempos y la realización de la gráfica.

```
#define BUILDING_DEF_STRATEGY_LIB 1

#include "../simulador/Asedio.h"
#include "../simulador/Defense.h"
#include "cronometro.h"

using namespace Asedio;

//funcion encargada de calcular la distancia ente dos objetos dando de estos su posicon y su
    radio
float distObjeRad(Vector3 posicion1, float radio1, Vector3 posicion2, float radio2) {

    float dist = _distance(posicion1, posicion2) - (radio1 + radio2);
    if (dist < 0) { // Caso en que los obstaculos chocan
        return 0;
    }
    else{
        return dist;
    }
}

float defaultCellValue(int row, int col, bool** freeCells, int nCellsWidth, int nCellsHeight
    , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;

    Vector3 cellPosition((col * cellWidth) + cellWidth * 0.5f, (row * cellHeight) +
        cellHeight * 0.5f, 0);

    float val = 0;
    for (List<Object*>::iterator it=obstacles.begin(); it != obstacles.end(); ++it) {
            val += _distance(cellPosition, (*it)->position);
    }
```

```
    return val;
}

bool factible(Object* defensa, int row, int col, int cellH, int cellW, float mapWidth, float
    mapHeight, List<Object*> obstacles, List<Defense*> defenses) {


    auto cell = Vector3(row * cellH + cellH * 0.5f, col * cellW + cellW * 0.5f, 0);
    bool res = true;
    if ((defensa->radio >= cell.x || cell.x >= mapWidth - defensa->radio) || (defensa->radio
        >= cell.y || cell.y >= mapHeight - defensa->radio) ) {
        res = false;
    }
    for (auto it = obstacles.begin(); it != obstacles.end() && res; it++){
        if (distObjeRad(cell, defensa->radio, (*it)->position, (*it)->radio) <= 5) {
            res = false;
        }
    }
    for (auto it = defenses.begin(); it != defenses.end() && res; it++){
        if (distObjeRad(cell, defensa->radio, (*it)->position, (*it)->radio) <= 0) {

            res = false;
        }

    }
    return res;
}

void quicksort(float* vec,int pri,int ult)
{
    int med = (pri+ult)/2, i = pri, j = ult;
    double pivote = vec[med];

    do{
        while(vec[i] > pivote) { i++; }
        while(vec[j] < pivote) { j--; }
        if(i<=j){
            int aux = vec[i];
            vec[i] = vec[j];
            vec[j] = aux;
            i++;
            j--;
        }
    } while(i <= j);
    if(pri < j)
        quicksort(vec, pri, j); /*mismo proceso con sublista izquierda*/
    if( i< ult)
        quicksort(vec, i, ult); /*mismo proceso con sublista derecha*/
}

void merge(float* vec, int tam, int izquierda, int medio, int derecha){
    int h = izquierda ,i = izquierda ,j = medio + 1;
    float res[tam];

    while((h <= medio) && (j <= derecha)){
        if(vec[h] >= vec[j]){
            res[i] = vec[h];
            h++;
        }
        else{
            res[i] = vec[j];
            j++;
        }
        i++;
    }
    if(h > medio){
        for(int k = j; k <= derecha; k++){
            res[i] = vec[k];
            i++;
        }
    }
```

```
    else{
        for(int k = h; k<=medio; k++){
            res[i] = vec[k];
            i++;
        }
    }
    for(int k = izquierda; k<=derecha; k++){
        vec[k] = res[k];
    }
}

void merge_sort(float* vec, int tam, int izquierda, int derecha){
    if(izquierda>derecha){
        int medio=(izquierda+derecha)/2;
        merge_sort(vec, tam, izquierda, medio);
        merge_sort(vec, tam, medio+1, derecha);
        merge(vec, tam, izquierda, medio, derecha);
    }
}

void DEF_LIB_EXPORTED placeDefenses3(bool** freeCells, int nCellsWidth, int nCellsHeight,
    float mapWidth, float mapHeight
              , List<Object*> obstacles, List<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    int tam = nCellsHeight * nCellsWidth;
    float ErrRel = 0.001;
    float ErrAbs = 0.01;
    float Err = ErrAbs * ErrRel + ErrAbs;

    cronometro A, B, C, D;
    int ra = 0, rb = 0, rc = 0, rd = 0;

    A.activar();
    do {
        ra++;
        float* cellValues = new float[tam];
        for(int i = 0; i < nCellsHeight; ++i) {
            for(int j = 0; j < nCellsWidth; ++j) {
                cellValues[i * nCellsHeight +j] = defaultCellValue(i, j, freeCells,
                    nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses);
            }
        }

        auto defAct = ++defenses.begin();
        int i = 0;
        bool fin1 = true;
        while (defAct != defenses.end()){
            while (fin1 && cellValues[i] != cellValues[nCellsHeight + nCellsWidth - 1]){
                if ( cellValues[i] != NULL && factible(*defAct, i / nCellsWidth, i %
                    nCellsWidth, cellHeight, cellWidth, mapWidth, mapHeight, obstacles,
                    defenses)){
                    (*defAct)->position = Vector3(i / nCellsWidth * cellHeight + cellHeight *
                         0.5f, i % nCellsHeight * cellWidth + cellWidth * 0.5f, 0);;
                    fin1 = false;
                    cellValues[i] = NULL;
                    i = 0;
                }
                else{
                    i++;
                }
            }
            fin1 = true;
            defAct++;
        }
    }while (A.tiempo() < Err);
    A.parar();

    B.activar();
    do {
```

```
        rb++;
        float* cellValues = new float[tam];
        for(int i = 0; i < nCellsHeight; ++i) {
            for(int j = 0; j < nCellsWidth; ++j) {
                cellValues[i * nCellsHeight +j] = defaultCellValue(i, j, freeCells,
                    nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses);
            }
        }

        merge_sort(cellValues, tam, 0, tam-1);

        auto defAct = ++defenses.begin();
        int i = 0;
        bool fin1 = true;
        while (defAct != defenses.end()){
            while (fin1 && cellValues[i] != cellValues[nCellsHeight + nCellsWidth - 1]){
                if ( cellValues[i] != NULL && factible(*defAct, i / nCellsWidth, i %
                    nCellsWidth, cellHeight, cellWidth, mapWidth, mapHeight, obstacles,
                    defenses)){
                    (*defAct)->position = Vector3(i / nCellsWidth * cellHeight + cellHeight *
                         0.5f, i % nCellsHeight * cellWidth + cellWidth * 0.5f, 0);;
                    fin1 = false;
                    cellValues[i] = NULL;
                    i = 0;
                }
                else{
                    i++;
                }
            }
            fin1 = true;
            defAct++;
        }
    }while (B.tiempo() < Err);
    B.parar();

    C.activar();
    do {
        rc++;
        float* cellValues = new float[tam];
        for(int i = 0; i < nCellsHeight; ++i) {
            for(int j = 0; j < nCellsWidth; ++j) {
                cellValues[i * nCellsHeight +j] = defaultCellValue(i, j, freeCells,
                    nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses);
            }
        }

        quicksort(cellValues, 0, tam-1);

        auto defAct = ++defenses.begin();
        int i = 0;
        bool fin1 = true;
        while (defAct != defenses.end()){
            while (fin1 && cellValues[i] != cellValues[nCellsHeight + nCellsWidth - 1]){
                if ( cellValues[i] != NULL && factible(*defAct, i / nCellsWidth, i %
                    nCellsWidth, cellHeight, cellWidth, mapWidth, mapHeight, obstacles,
                    defenses)){
                    (*defAct)->position = Vector3(i / nCellsWidth * cellHeight + cellHeight *
                         0.5f, i % nCellsHeight * cellWidth + cellWidth * 0.5f, 0);;
                    fin1 = false;
                    cellValues[i] = NULL;
                    i = 0;
                }
                else{
                    i++;
                }
            }
            fin1 = true;
            defAct++;
        }
    }while (C.tiempo() < Err);
    C.parar();
```

```
    D.activar();
    do {
        rd++;
        float* cellValues = new float[tam];
        for(int i = 0; i < nCellsHeight; ++i) {
            for(int j = 0; j < nCellsWidth; ++j) {
                cellValues[i * nCellsHeight +j] = defaultCellValue(i, j, freeCells,
                    nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses);
            }
        }

        std::make_heap(cellValues, cellValues + tam - 1);

        auto defAct = ++defenses.begin();
        int i = 0;
        bool fin1 = true;
        while (defAct != defenses.end()){
            while (fin1 && cellValues[i] != cellValues[nCellsHeight + nCellsWidth - 1]){
                if ( cellValues[i] != NULL && factible(*defAct, i / nCellsWidth, i %
                    nCellsWidth, cellHeight, cellWidth, mapWidth, mapHeight, obstacles,
                    defenses)){
                    (*defAct)->position = Vector3(i / nCellsWidth * cellHeight + cellHeight *
                         0.5f, i % nCellsHeight * cellWidth + cellWidth * 0.5f, 0);;
                    fin1 = false;
                    cellValues[i] = NULL;
                    i = 0;
                }
                else{
                    i++;
                }
            }
            fin1 = true;
            defAct++;
        }
    }while (D.tiempo() < Err);
    D.parar();

    std::cout << (nCellsWidth * nCellsHeight) << '\t' << A.tiempo() / ra << '\t' << B.tiempo
        () / rb << '\t' << C.tiempo() / rc << '\t' << D.tiempo() / rd << std::endl;
}
```

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de este documento confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.