Práctica 3. Divide y vencerás

AGUSTIN ENEAS HARISPE LUCARELLI
correo@servidor.com
Teléfono: xxxxxxxx
NIF: 55080402W

15 de junio de 2023

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

   He utilizado dos vectores de tamaño N = nCellsWidth*nCellsHeight, celvalues para el guardado defaultCellValue y el segundo guarda las posiciones i de estas celdas. A la hora de ordenar el primer vector, se le aplican las mismas permutaciones al segundo. Así se obtiene un vector de celdas ordenadas en función de los valores de defaultCellValue.

   En el caso de la ordenación por montículo se ha usado un vector de pares ¡flaot, int¿, en el primer elemento se guarda el resultado de defaultCellValue y en el segundo la posicion a la que corresponde.

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

```
void merge(float *array, int izq, int medio, int der, int* indices)
{
    //variables
    int i, j, k;
    int tam1 = (medio -izq) + 1;
    int tam2 = (der - medio);
    float *arr_izq, *arr_der;
    int *ind_izq, *ind_der;
    //creamos subarrays
    arr_izq = (float*)malloc(tam1 * sizeof(float));
    arr_der = (float*)malloc(tam2 * sizeof(float));
    ind_izq = (int*)malloc(tam1 * sizeof(int));
    ind_der = (int*)malloc(tam2 * sizeof(int));
    //se copian los datos
    for (i = 0; i < tam1; i++)
    {
        arr_izq[i] = *(array + izq + i);
        ind_izq[i] = *(indices + izq + i);
    }
    for (j = 0; j < tam2; j++)
    {
        arr_der[j] = *(array + medio + j + 1);
        ind_der[j] = *(indices + medio + j + 1);
    }
    i = 0;
    j = 0;
    //Fusion de datos
    for (k = izq; k < der + 1; k++)
    {
        if (i == tam1)
        {
            *(array + k) = *(arr_der + j);
            *(indices + k) = *(ind_der + j);
            j =  j+ 1;
        }
        else if(j == tam2)
        {
```

```
            *(array + k) = *(arr_izq + i);
            *(indices + k) = (*ind_izq + i);
            i = i + 1;
        }
        else
        {
            if (*(arr_izq + i) > *(arr_der + j))
            {
                *(array + k) = *(arr_izq+ i);
                *(indices + k) = *(ind_izq+ i);
                i = i + 1;
            }
            else
            {
                *(array + k) = *(arr_der + j);
                *(indices + k) = *(ind_der + j);
                j = j + 1;
            }
        }
    }
}
void merge_sort(float *array, int izq, int der, int * indices)
{
    if (izq < der)
    {
        int medio = (izq + der)/2;

        merge_sort(array, izq, medio,indices);
        merge_sort(array, medio + 1, der,indices);

        merge(array, izq, medio, der, indices);
    }
}
```

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

```
void ordenacion_rapida(float *array, int izq, int der, int *indices){
  int i = izq;
  int j = der;
  float aux = *(array + izq); //pivote
  int ind_aux = *(indices + izq);

  if(izq < der){
    while(i<j){
      while(*(array + j) <= aux && i<j){j--;}
      *(array + i) = *(array + j);
       *(indices + i) = *(indices + j);
      while(*(array + i)>= aux && i<j){i++;}
      *(array + j)=*(array + i);
      *(indices + j)=*(indices+ i);
    }
    *(array + i) = aux;
    *(indices + i) = ind_aux;
    ordenacion_rapida(array,izq,i-1,indices);
    ordenacion_rapida(array,j+1,der,indices);
  }
}
```

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.

   Se hacen algunas sencillas pruebas de caja negra, el código no relevante o repetitivo se obvia.

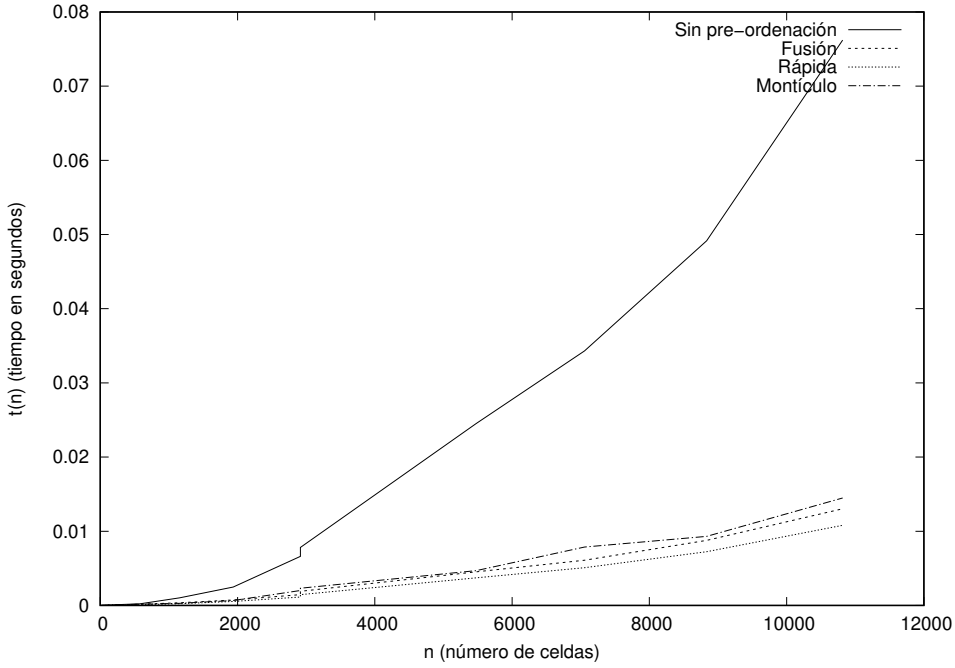
```
void cajanegra(){
  //aqui se realizaran las pruebas con diferentes inputs
  float p02[] = {}; int i02[] = {};
```

```
  float p1 = 1;  int i1 = 1;
  float p2[] = {0,0,0,0,0,0,0}; int i2[] = {0,1,2,3,4,5,6};
  float p3[] = {-1,2,3,4,5,6,7,8}; int i3[] = {0,1,2,3,4,5,6,7};
  float p4[] = {8,7,6,5,4,3,2,1}; int i4[] = {0,1,2,3,4,5,6,7};
  //[...] mas inicializaciones
  std::vector<std::pair<float,int>> p44;
  for(int i=0; i<8; i++){
      p44.emplace_back(p4[i],i);
  }

  //por fusion
  merge_sort(&p1,0,0,&i1); //un unico elemento
  printf("%f\n",p1);
  merge_sort(p2,0,6,i2); //elementos repetidos
  for (int i = 0 ; i <*(&p2 + 1) - p2; i++)
    std::cout << p2[i] << '\t' << i2[i] << std::endl;
  printf("\n");
  merge_sort(p3,0,7,i3); //ordenado ascendentemente
  for (int i = 0 ; i <*(&p3 + 1) - p3; i++)
    std::cout << p3[i] << '\t' << i3[i] << std::endl;
  printf("\n");
  merge_sort(p4,0,7,i4); //ordenado descendentemente
  for (int i = 0 ; i <*(&p4 + 1) - p4; i++)
    std::cout << p4[i] << '\t' << i4[i] << std::endl;
  printf("\n\n");

  //rapida
  //[...] igual que fusion
  //monticulo
  ordenar_monticulo(p14); //un unico elemento
  printf("%f %i\n",p14[0].first, p14[0].second);
  ordenar_monticulo(p24);  //elementos repetidos
  for (int i = 0 ; i <7; i++)
    std::cout << p24[i].first << '\t' << p24[i].second << std::endl;
  printf("\n");
  ordenar_monticulo(p34); //ordenado ascendentemente
  for (int i = 0 ; i <8; i++)
    std::cout << p34[i].first << '\t' << p34[i].second << std::endl;
  printf("\n");
  ordenar_monticulo(p44); //ordenado descendentemente
  for (int i = 0 ; i <8; i++)
    std::cout << p44[i].first << '\t' << p44[i].second << std::endl;
  printf("\n\n");

}
```

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.

   Escriba aquí su respuesta al ejercicio 5.

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 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.



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.