Práctica 3. Divide y vencerás

Luis José Quintana Bolaño
luisjquintana@gmail.com
Teléfono: 956535843
NIF: 49073584w

18 de enero de 2015

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

   Para todos los casos se ha usado la misma estructura:

```
typedef std::vector <std::pair <std::pair<int, int>, float> > values_t;
```

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

```
values_t merge(values_t l_left, values_t l_right){
        values_t result(l_left.size()+l_right.size());
        values_t::iterator ileft=l_left.begin(),
                                        iright=l_right.begin(),
                                        iresult=result.begin();
        while(ileft!=l_left.end() && iright!=l_right.end()){
                if(ileft->second>=iright->second){
                        *iresult=*ileft;
                        ++ileft;
                } else {
                        *iresult=*iright;
                        ++iright;
                }
                ++iresult;
        }
        while(ileft!=l_left.end()){
                *iresult=*ileft;
                ++iresult;++ileft;
        }
        while(iright!=l_right.end()){
                *iresult=*iright;
                ++iresult;++iright;
        }
        return result;
}

values_t mergeSort(values_t list){
        //Base case
        if (list.size()<=1){
                return list;
        } else {
        //Recursive case
                values_t l_left(list.begin(),list.begin()+(list.size()/2)),
                                l_right(list.begin()+(list.size()/2),list.end());
                values_t fl_left=mergeSort(l_left);
                values_t fl_right=mergeSort(l_right);

                return merge(fl_left,fl_right);
        }
}
```

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

```
values_t concatenate(values_t less, values_t equal, values_t greater){
        values_t result(greater.begin(), greater.end());
        result.insert(result.end(),equal.begin(),equal.end());
        result.insert(result.end(),less.begin(),less.end());
        return result;
}

values_t quickSort(values_t list){
        values_t less,equal,greater;
        if (list.size()>1){
                float pivot=(list.begin()+(list.size()/2))->second;
                for(values_t::iterator i=list.begin();i!=list.end();++i){
                        if(i->second>pivot)
                                greater.push_back(*i);
                        else if(i->second==pivot)
                                equal.push_back(*i);
                        else
                                less.push_back(*i);
                }
                return concatenate(quickSort(less),equal,quickSort(greater));
        } else {
                return list;
        }
}
```

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.

```
bool comparePair(const std::pair <std::pair<int, int>, float>& a, const std::pair <std::pair<
    int, int>, float>& b) {
  return a.second > b.second;
}

bool is_sorted (values_t::iterator first, values_t::iterator last)
{
  if (first==last) return true;
  values_t::iterator next = first;
  while (++next!=last) {
    if (comparePair(*next,*first))
      return false;
    ++first;
  }
  return true;
}
/***/
#ifdef PRUEBAS_CAJA_NEGRA
        for(int i=0;i<10;++i) {
                cellVal.push_back(std::make_pair(std::make_pair((float)i,(float)i),(float)i))
                    ;
        }
        std::sort(cellVal.begin(),cellVal.end(),comparePair);
        while( next_permutation(cellVal.begin(),cellVal.end(),comparePair)){
                values_t copiacellVal=mergeSort(cellVal);
                if(!is_sorted(copiacellVal.begin(),copiacellVal.end())){
                        std::cout<< "ERROR EN LA ORDENACION POR FUSION" <<std::endl;
                }
        }
        std::sort(cellVal.begin(),cellVal.end(),comparePair);
        while( next_permutation(cellVal.begin(),cellVal.end(),comparePair)){
                values_t copiacellVal=quickSort(cellVal);
                if(!is_sorted(copiacellVal.begin(),copiacellVal.end())){
                        std::cout<< "ERROR EN LA ORDENACION RAPIDA" <<std::endl;
                }
        }
#endif
```

5. Comente y justifique a continuación los resultados esperados en cada caso. Suponga un terreno de batalla cuadrado en todos los casos.

   Tomando un tablero de n casillas y m defensas:

   Para el caso sin preodenación, se realizarán en el peor caso (si ninguna posición es factible para alguna de las torres) $\sum_{i=0}^{n} n - i$, ya que tras cada acceso se elimina de la estructura que contiene las casillas el elemento comprobado. Por tanto el algoritmo es de orden $O(n^2/2$ en el peor caso y $O(n)$ en el mejor.

   Tanto en el caso preordenado por fusión como en ordenacióin rápida y por montículo, el algoritmo es de orden $O(n \log n)$ en el mejor caso, siendo $O(n^2)$ el peor caso para quicksort (en muy raras ocasiones).

   Con estos datos, los resultados esperados son que todos, exceptuando quizá el caso sin preordenación, estarán muy igualados.

6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.001 y un error relativo de valor 0.001). Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500. Incluya en el análisis los planetas que considere oportunos para mostrar información relevante.

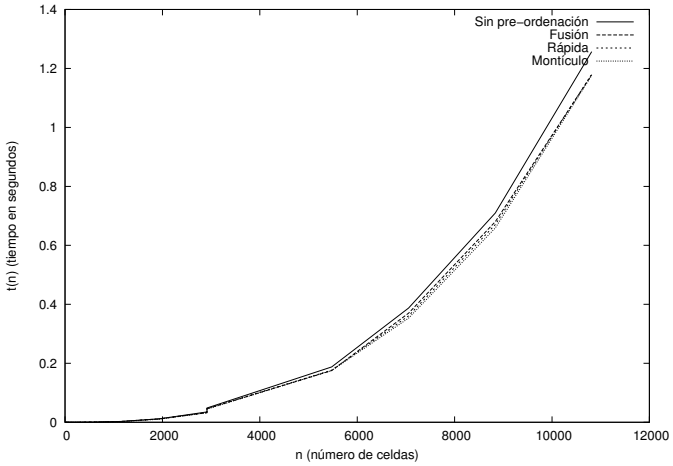

Figura 1: Gráfica de tiempos.

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.