30-5-2020

Néstor Monzón González - 735418

Andrés Otero García - 757755

PRÁCTICA 1

Radix Sort

Algoritmia Básica

1 – Implementación

En primer lugar, se siguió una implementación iterativa del algoritmo. Tras considerarlo, se decidió reformarlo en una función recursiva, ya que se reducía el número de copias del vector auxiliar al original.

El código principal de la función es el siguiente:

// v el vector de numeros (formato string), n\_dig el numero de digitos

// ordena v con el algoritmo de radix LSF

void radix\_rec(vector<string>& v, const int n\_dig, const int i\_dig) {

vector<int> cuenta = CUENTA\_INICIAL; // lleva la cuenta del numero de veces que aparece cada digito

int n\_eltos = v.size();

vector<string> aux(n\_eltos); // auxiliar del mismo tamaño que el original

for (auto e : v) {

int digito = e[i\_dig] - '0'; // i-ésimo digito

//cout << "digito: " << digito << endl;

cuenta[digito]++; // Contamos el digito correspondiente

}

// Acumulamos las cuentas de los digitos de izquierda a dcha:

acumular(cuenta);

// Y recolocamos cada elemento del original en el auxiliar según la cuenta:

for (int j = n\_eltos - 1; j >= 0; j--) { // (empezando por la dcha)

string elemento = v[j]; // Tomamos el j-esimo elemento

int digito = elemento[i\_dig] - '0'; // Y su i-esimo digito

int indice = cuenta[digito]--; // Posicion en el auxiliar (y reducimos esa cuenta)

aux[indice] = elemento;

}

// Finalmente, actualizamos el vector original:

If (i\_dig > 0) {

radix\_rec(aux, n\_dig, i\_dig-1);

}

}

Este código asume que todos los elementos tienen longitud n\_dig, por lo que requiere que el vector v se inicialice con elementos de esta longitud (los números con menos dígitos deberán tener ceros a la izquierda).

El coste temporal de esta función, a grandes rasgos será O(n\_dig(n\_eltos+base)), donde n\_dig es el número de dígitos, n\_eltos el número de números y base la base, en este caso 10. Cuando n\_eltos sea muy grande en comparación a la base (n\_eltos>>base), la base será despreciable y el orden será O(n\_dig\*n\_eltos).

Este coste se debe a que el primer bucle tiene n\_eltos iteraciones; acumular(cuenta), una función simple que suma en cada elemento de <cuenta> la cuenta de los elementos anteriores, es decir, unas 10 iteraciones, un coste despreciable; y el bucle final, de n\_eltos iteraciones de nuevo.

Así, en total, esto supone n\_eltos+10+n\_eltos=O(n\_eltos) por cada llamada no recursiva a esta función (hasta el if final). Añadiendo la recursión, esto se ejecuta un total de n\_dig veces, llevando al coste final de O(n\_dig\*n\_eltos).

2 – Pruebas y análisis

Se ha desarrollado el programa principal de forma que acepte varios tipos de invocaciones. Si se invoca sin parámetros, permitirá elegir entre el bucle de prueba predefinido y el modo interactivo, que permite un bucle infinito en el que se introducen los parámetros n\_digitos y n\_eltos y muestra los tiempos de la ordenación con nuestra función y con la función sort estándar de c++.

Además, permite elegir si se muestran los vectores antes y después de su ordenación, para así comprobar el correcto funcionamiento del algoritmo.

Por otra parte, invocándolo con dos argumentos, el primero corresponderá al número de dígitos, y el segundo al número de elementos. Se puede añadir un tercero (S/N) que determine si se mostrarán los vectores, como en el modo interactivo. Con este modo, se facilita la invocación desde scripts externos.

Para realizar las mediciones de los tiempos con precisión, se ha utilizado la biblioteca chrono de c++. Se miden los tiempos de ejecución de ambas funciones, y se muestran en pantalla para compararlos.

Antes de ejecutar las pruebas predefinidas, nuestra hipótesis es que el algoritmo de radix será más rápido que el sort de c++ cuando haya un gran número de elementos a ordenar con respecto al número de dígitos. Concretamente, ya que el coste del Quicksort es O(n\_eltos\*log(n\_eltos)), nuestro algoritmo será más rápido cuando el número de dígitos sea menor que log(n\_eltos):

Donde, de nuevo, es el número de elementos a ordenar y el número de dígitos de cada elemento.

En la siguiente ejecución de las pruebas preparadas en Practica1.cpp (invocándolo sin argumentos y seleccionando la opción 2), para ciertos valores de n\_digs y n\_eltos, se muestran los tiempos de los algoritmos, si radix es más rápido, el valor de log(n\_eltos), y si el log(n\_eltos)>n\_digs. Solo se muestran los últimos resultados por cuestión de espacio.

----------------------------------------------------------------

Ordenando con 4 digitos y 10000 elementos

Tiempo Radix: 0.0031568

Tiempo Sort: 0.0038234

radix es mejor: 1

log(n\_eltos)=9.21034

log(n\_eltos) > n\_digs = 1

----------------------------------------------------------------

Ordenando con 4 digitos y 100000 elementos

Tiempo Radix: 0.0245497

Tiempo Sort: 0.0478288

radix es mejor: 1

log(n\_eltos)=11.5129

log(n\_eltos) > n\_digs = 1

----------------------------------------------------------------

Ordenando con 4 digitos y 1000000 elementos

Tiempo Radix: 0.269179

Tiempo Sort: 0.513946

radix es mejor: 1

log(n\_eltos)=13.8155

log(n\_eltos) > n\_digs = 1

----------------------------------------------------------------

El radix era mejor cuando el logaritmo del numero de elementos era

mayor que el numero de digitos en 20 casos de 24 (0.833333)

Como se puede observar en el resumen final, la hipótesis se ha cumplido en un 83% de los casos. Aunque haya un cierto margen a considerar, se podría añadir una constante a la comparación para aumentar este porcentaje si se necesitara conocer el método óptimo para unos datos a priori.

De esta forma, se podría elegir entre un método de ordenación u otro dinámicamente en función de estos parámetros, asegurando la ordenación más rápida posible en todos los casos según las características de los datos de entrada.

En cuanto a la eficiencia en memoria, este método es siempre peor que otros como el Quicksort, ya que tiene que almacenar un vector adicional con el número de elementos de la base (es decir, uno para cada dígito posible).