Análisis y Algoritmos

Sergio Juan Diaz Carmona
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv17c.sdiaz@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 5, 2019

1) Algoritmos de Ordenamiento
   En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada.
   Desarrollamos dos metodos de ordenamientos, ”bubble sort” e ”insertion sort”, ambos tendran las mismas entradas; una entrada en orden acendente, una entrada en orden random y una entrada en orden decendente, todas con la misma cantidad de datos.
   sin embargo en las tablas solo se tendran en cuenta el mejor y el peor caso, que serian la acendente y la decenten, en ese orden. en ambos casos el tamaño de la entrada sera en ”n”

2) Algoritmos de Busqueda Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez. La variante más simple del problema es la búsqueda de un número en un vector.

Merge-Sort

Merge sort. Algoritmo basado en la técnica de diseño de algoritmos Divide y Vencerás. Consiste en dividir el problema a resolver en subproblemas del mismo tipo que a su vez se dividirán, mientras no sean suficientemente pequeños o triviales.
La complejidad es de O(n log n), y en master method es de $n^{(log(base2)2)}$
$elcodigoeselsiguiente$

```
std::vector<int> Merge(std::vector<int> left, std::vector<int> right)
{
        std::vector<int> result;
        while (left.size() > 0 || right.size() > 0)
        {
                if (left.size() > 0 && right.size() > 0)
                {
                        if (left[0] <= right[0])
                        {
                                result.push_back(left[0]);
                                left.erase(left.begin());
                        }
                        else
                        {
                                result.push_back(right[0]);
                                right.erase(right.begin());
                        }
                }
                else if (left.size() > 0)
                {
                        for (int i = 0; i < left.size(); i++)
                                result.push_back(left[i]);
```

```
                        break;
                }
                else if (right.size() > 0)
                {
                        for (int i = 0; i < right.size(); i++)
                                result.push_back(right[i]);
                        break;
                }
        }
        return result;
}

std::vector<int> MergeSort(std::vector<int> vec)
{
        if (vec.size() <= 1)
                return vec;

        std::vector<int> left, right, result;
        int middle = ((int)vec.size() + 1) / 2;

        for (int i = 0; i < middle; i++) {
                left.push_back(vec[i]);
        }

        for (int i = middle; i < (int)vec.size(); i++) {
                right.push_back(vec[i]);
        }

        left = MergeSort(left);
        right = MergeSort(right);
        result = Merge(left, right);

        return result;
}
}
```

QUICK-SORT

Quicksort (a veces llamado clasificación de intercambio de particiones) es un algoritmo de clasificación eficiente , que sirve como un método sistemático para colocar los elementos de un archivo de acceso aleatorio o una matriz en orden. Quicksort es una ordenación de comparación , lo que significa que puede ordenar elementos de cualquier tipo para los que se define una relación ”menor que” (formalmente, un orden total). En implementaciones eficientes, no es una clasificación estable , lo que significa que no se conserva el orden relativo de los elementos de igual clasificación.

El análisis matemático de quicksort muestra que, en promedio , el algoritmo toma comparaciones O (n log n) para ordenar n elementos. En el peor de los casos , hace comparaciones de O (n 2), aunque este comportamiento es raro. En master method T(n)=2T(n/2)+(n)-¿ T(n) = (nlogn)

el codigo es el siguiente

```
std::vector<int> insertion(std::vector<int> vec)
{
int partition(std::vector<int> &vec, int min, int max)
{
        int pivot = vec[max];     // pivot
        int i = (min - 1);  // Index of smaller element

        for (int j = min; j <= max - 1; j++)
        {

                if (vec[j] <= pivot)
```

```
                {
                        i++;
                        int swap = vec[i];
                        vec[i] = vec[j];
                        vec[j] = swap;
                }
        }
        int swap = vec[i + 1];
        vec[i + 1] = vec[max];
        vec[max] = swap;
        return (i + 1);
}

void quickSort(std::vector<int>& vec, int min, int max)
{
        if (min < max)
        {
                int mid = partition(vec, min, max);

                quickSort(vec, min, mid - 1);
                quickSort(vec, mid + 1, max);
        }
}
}
```

BINARY-SEARCH

la búsqueda binaria, también conocida como búsqueda de intervalo medio1 o búsqueda logarítmica,2 es un algoritmo de búsqueda que encuentra la posición de un valor en un array ordenado.34 Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

La complejidad es T(n) = T(n/2) + c, dando como resultado en el master method O (log n) comparaciones, donde n es el número de elementos del arreglo y log es el logaritmo.

el codigo es el siguiente

```
int binarySearch(std::vector<int> vec, int lefht, int right, int value)
{
        if (right >= lefht) {
                int mid = lefht + (right - lefht) / 2;

                if (vec[mid] == value)
                        return mid;
                if (vec[mid] > value)
                        return binarySearch(vec, lefht, mid - 1, value);

                return binarySearch(vec, mid + 1, right, value);
        }

        return -1;
}
```

Linear

una búsqueda lineal o una búsqueda secuencial es un método para encontrar un elemento dentro de una lista . Comprueba secuencialmente cada elemento de la lista hasta que se encuentra una coincidencia o se ha buscado en toda la lista.

Su complejidad es de O(n) y el resultado con el master ,ethod es n

el codigo es el siguiente

```
int linearSearch(std::vector<int> vec, int value)
{
```

```
        for (int i = 0; i < vec.size(); i++)
        {
                if (vec[i] == value)
                {
                        return i;
                }
        }
        return -1;
}
```