Kevin Alanis Pineda 2018083622
Adrián López Vásquez 2016108981

Tarea Extraclase #3: Ordenes de crecimiento

- **Logaritmico Log(n)**

Búsqueda Binaria: Tal como se observa en el siguiente código el algoritmo búsqueda binaria divide el array en dos mitades y va buscando en las sublistas comparando el pivote con el valor que se debe buscar para decidir en que lado de la lista buscar y repite esto hasta encontrar el valor. Como al dividir la lista se va durar menos que una búsqueda lineal el comportamiento será log(n).

```
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;
      if (arr[mid] == x)
         return mid;
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);
      return binarySearch(arr, mid + 1, r, x);
   }
    return -1;
}
```

Para ver si un algoritmo es O(log(n)) se debe ver si este realiza una cierta cantidad de iteraciones menores que n independientemente de su entrada.

- **Lineal (n)**

Búsqueda Lineal: Este algoritmo va comparando el elemento en cada posición con el valor a buscar y retorna el valor a buscar. El peor caso es que el elemento este en la última posición por lo que va a realizar n comparaciones hasta llegar al valor siendo n el largo del array, por ende, es un orden de crecimiento lineal.

```
int search(int arr[], int n, int x)
{
   int i;
   for (i = 0; i < n; i++)
      if (arr[i] == x)
         return i;
   return -1;
}
```

Para saber si un algoritmo es O(n) se debe observar si el peor caso requiere la misma cantidad de iteraciones que su entrada.

- **Lineal logarítmico (nlog(n))**

Mergesort: Es un algoritmo de ordenamiento que consiste en dividir la lista a la mitad para seleccionar un pivote y realiza una comparación si los valores no están ordenados entonces realiza una nueva sublista tomando en cuenta todas las posiciones en el medio de los valores que se compararon y realiza esto hasta que queden sublistas de 1 elemento y luego concatena en orden las sublistas. Se comporta como un nlog(n) es log(n) ya que al dividir con un pivote va a recorrer la lista menos que n veces y como se debe volver a formar la lista ordenada va a recorrer n veces la lista mientras la crea.

```
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;
   int L[n1], R[n2];
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
```

```
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
     while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
  void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

Este orden de crecimiento se basa en un n y un log(n) entonces se deben identificar cada uno por separado entendiendo el funcionamiento del algoritmo. Siendo este mayor que n pero menor que n^2.

- **Cuadrático (n^2)**

Quicksort: Es un algoritmo de ordenamiento que se basa en la elección de un pivote que puede ser aleatorio. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha. Este proceso se realiza de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados. En el peor caso al ser el pivote aleatorio el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$ ya que la lista se va a recorre n veces para comparar cada posición con el pivote y luego se debe recorrer n veces para unir las sublistas ordenadas.

```
int partition(int arr[], int low, int high)
{
    int pivot = arr[low];
    int i = low - 1, j = high + 1;

    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);

        if (i >= j)
            return j;

        swap(arr[i], arr[j]);
    }
}

// Generates Random Pivot, swaps pivot with
int partition_r(int arr[], int low, int high)
{
    srand(time(NULL));
    int random = low + rand() % (high - low);
    swap(arr[random], arr[low]);

    return partition(arr, low, high);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        int pi = partition_r(arr, low, high);
        quickSort(arr, low, pi);
        quickSort(arr, pi + 1, high);
```

```
    }
}
```

Este tipo de orden de crecimiento se puede observar si un algoritmo realiza dos for() o realiza dos iteraciones de orden n.

- **Cúbico(n^3)**

Floyd-Warshall: Este es un algoritmo para buscar las rutas más cortas entre cada par de vértices. El algoritmo recibe un grafo de tamaño V entonces crea una copia y se empieza a recorrer como se muestra en el código. Se recorre 3 veces n ya que son 3 for() anidados que permiten calcular todas las posibles rutas cortas entre cada par de vértices.

```
void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    printSolution(dist);
}
```

Para identificar un $O(n^n)$ se debe realizar n^n iteraciones o lo que n for() que involucren una n cantidad de iteraciones por cada uno.

- **Exponencial (2^n)**

Cálculo de Fibonacci: Fibonacci es una serie sucesiva recursiva que se basa en la siguiente ecuación F(n)=F(n-1)+F(n-2).

Cálculo de Fibonacci 5:

F(5)=F(4)+F(3) ->2^5

F(4)=F(3)+F(2) ->2^4

F(3)=F(2)+F(1) ->2^3

F(2)=F(1)+F(0) ->2^2

F(1)=F(0)=1 ->2^1

F(2)=1+1=2

F(3)=2+1=3

F(4)=3+2=5

F(5)=5+3=8

Kevin Alanis Pineda 2018083622
Adrián López Vásquez 2016108981

Para realizar el cálculo se debe hacer 2^n iteraciones tal como se ve en el ejemplo anterior ya que para 5 se ocupa fibonacci de 4 y de 3 y el de 4 ocupa el de 3 y el de 2 y así sucesivamente por lo que se cálculo va a estar regido por este comportamiento.

```
int fib(int n)
{
  int f[n+2];   // 1 extra to handle case, n = 0
  int i;
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

Un $O(2^n)$ para cada iteración ocupara dos elementos y hará n iteraciones.

- **Factorial (n!)**

Problema de vendedor ambulante (TSP): Suponiendo que un vendedor ambulante debe repartir distintos negocios pasando por todas las ciudades. El algoritmo consiste en encontrar la ruta más corta posible que visite cada ciudad exactamente una vez y regrese al punto de partida. Este es un problema NP y no existe solución polinomial. Como se ve tiene for() y while() lo que aumenta el tiempo de ejecución.

```
int travllingSalesmanProblem(int graph[][V], int s)
{
    vector<int> vertex;
    for (int i = 0; i < V; i++)
        if (i != s)
            vertex.push_back(i);

    int min_path = INT_MAX;
    do {
        int current_pathweight = 0;
        int k = s;
        for (int i = 0; i < vertex.size(); i++) {
            current_pathweight += graph[k][vertex[i]];
            k = vertex[i];
        }
        current_pathweight += graph[k][s];

        min_path = min(min_path, current_pathweight);

    } while (next_permutation(vertex.begin(), vertex.end()));

    return min_path;
}
```

Cuando el tiempo de calculo es extremadamente grande como en el caso de las permutaciones o problemas NP que son realmente complejos siguen un orden de crecimiento factorial. Este tipo de ordenes son raros y solo se encuentran en problemas de muy alta exigencia o un alto nivel de iteraciones.