Universidad Nacional Autonoma de Mexico

Facultad de Ingeniería	Estructura de Datos y Algoritmos II

4to Examen Parcial

Profesor: Jorge A. Solano

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 2

Integrante: Barrero Olguin Adolfo Patricio

Semestre: 2019-1

Fecha de entrega: 26/nov/2018

Observaciones:

CALIFICACIÓN: _____

4to Examen Parcial

Análisis de las funciones de la estructura Vector

```
int VECTOR_obtener_medio(Vector A, int p, int r){
    int prom = 0;                                                    c1
    int i;                                                           c2
    for(i = p; i < r + 1; i++){                                      c3*n
        prom = prom + VECTOR_ver_elemento_en_posicion_a(A, i);       c4*(n-1)
    }                                                                c5
    int resta = r + 1 - p;                                           c6
    prom = ceil(prom / resta);                                       c7
    int ant = 0;                                                     c8
    int pos = 0;                                                     c9
    for (i = 0; i < resta; i++) {                                    c10*n
        int act = prom - VECTOR_ver_elemento_en_posicion_a(A, i + p); c11*(n-1)
        if (ant == 0 || ant >  VECTOR_abs(act)) {                    c12*(n-1)
            ant = VECTOR_abs(act);                                   c13*(n-1)
            pos = i + p;                                             c14*(n-1)
        }                                                            c15*(n-1)
    }                                                                c16*(n-1)
    return pos;                                                      c17
}                                                                    c18
```

Por lo tanto la complejidad de la función es O(n), siendo n = r-p

```
int VECTOR_tamano_vector(Vector vector){                             c1
    return vector.cantidad_elementos;                                c2
}                                                                    c3
```

Por lo tanto la complejidad de la función es O(1)

```
Vector VECTOR_crear_vector(int num_elementos){                       c1
    Vector vector;                                                   c2
    vector.conjunto_numeros = (int*) calloc(num_elementos, sizeof(int));  c3
    vector.cantidad_elementos = num_elementos;                       c4
    return vector;                                                   c5
}                                                                    c6
```

Por lo tanto la complejidad de esta función en el peor caso es O(1).

```
int VECTOR_ver_elemento_en_posicion_a(Vector vector, int posicion){
    int tamano = VECTOR_tamano_vector(vector);                       c1
    int num = 0;                                                     c2
    if(tamano > posicion){                                           c3
        num = vector.conjunto_numeros[posicion];                     c4
    }                                                                c5
    return num;                                                      c6
}                                                                    c7
```

Por lo tanto la complejidad de esta función para el peor caso es O(1).

Análisis del Algoritmo Forma Serial

```
int partition_original(Vector* arr, int low, int high){
    int i = low - 1;                                                 c1
    int pivot = VECTOR_ver_elemento_en_posicion_a(*arr, high);       c2
    int j, elemento;                                                 c3
```

```
    for(j = low; j < high; j++){                                        c4*(n)
        elemento = VECTOR_ver_elemento_en_posicion_a(*arr, j);          c5*(n-1)
        if(elemento <= pivot){                                          c6*(n-1)
            i = i + 1;                                                  c7*(n-1)
            VECTOR_intercambio(arr, i, j);                              c8*(n-1)
        }                                                               c9*(n-1)
    }                                                                   c10
    i = i + 1;                                                          c11
    VECTOR_intercambio(arr, i, high);                                   c12
    return i;                                                           c13
}
```

Por lo tanto la complejidad de esta función es O(n) siendo n = high - low

```
void quickSort_original(Vector* arr, int low, int high){
    if(low < high){                                                     c1*log(n)
        int pi = partition_original(arr, low, high);                    c2*n*(log(n)-1)
        quickSort_original(arr, low, pi - 1);                           c3*(log(n)-1)
        quickSort_original(arr, pi + 1, high);                          c4*(log(n)-1)
    }
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
void QuickSortOriginal(Vector* A){
    int extremo_derecho = VECTOR_tamano_vector(*A) - 1;                 c1
    quickSort_original(A, 0, extremo_derecho);                          c2*log(n)*n
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
int partition(Vector* arr, int low, int high){
    int i = low - 1;                                                    c1
    int pivot = VECTOR_ver_elemento_en_posicion_a(*arr, high);          c2
    int j, elemento;                                                    c3
    for(j = low; j < high; j++){                                        c4*(n)
        elemento = VECTOR_ver_elemento_en_posicion_a(*arr, j);          c5*(n-1)
        if(elemento <= pivot){                                          c6*(n-1)
            i = i + 1;                                                  c7*(n-1)
            VECTOR_intercambio(arr, i, j);                              c8*(n-1)
        }                                                               c9
    }                                                                   c10
    i = i + 1;                                                          c11
    VECTOR_intercambio(arr, i, high);                                   c12
    return i;                                                           c13
}
```

Por lo tanto la complejidad de esta función es O(n) siendo n = high - low

```
void quickSort(Vector* arr, int low, int high){
    if(low < high){                                                     c1*log(n)
        int pi = partition(arr, low, high);                             c2*n*(log(n)-1)
        quickSort(arr, low, pi - 1);                                    c3*(log(n)-1)
        quickSort(arr, pi + 1, high);                                   c4*(log(n)-1)
    }
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
void QuickSort(Vector* A){
    int extremo_derecho = VECTOR_tamano_vector(*A) - 1;                          c1
    quickSort(A, 0, extremo_derecho);                                            c2*log(n)
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
double DoubleQuickSort(Vector* arreglo){
    double tiempo = omp_get_wtime();                                             c1
    int i;                                                                       c2
    int contdir=0;                                                               c3
    int continv=0;                                                               c4
    int tamano = VECTOR_tamano_vector(*arreglo);                                 c5
    for (i = 1; i < tamano; i++) {                                               c6*(n)
        int elemento1 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i-1);        c7*(n-1)
        int elemento2 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i);          c8*(n-1)
        if(elemento1 <= elemento2){                                              c9*(n-1)
            contdir += 1;                                                        c10*(n-1)
        }                                                                        c11*(n-1)
    }                                                                            c12
    if(contdir == tamano - 1){                                                   c13
        tiempo = omp_get_wtime() - tiempo;                                       c16
        return tiempo;                                                           c15
    }                                                                            c16
    for (int i = 0; i < tamano; i++) {                                           c17*(n)
        int elemento1 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i-1);        c18*(n-1)
        int elemento2 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i);          c19*(n-1)
        if(elemento1 >= elemento2){                                              c20*(n-1)
            continv += 1;                                                        c21*(n-1)
        }                                                                        c22*(n-1)
    }                                                                            c23
    if(continv == tamano - 1){                                                   c24
        int longitud = floor(tamano / 2); //Redondeo hacia arriba                c25
        for(i = 0; i < longitud; i++){                                           c26*(n)
            VECTOR_intercambio(arreglo, i, tamano - 1 - i);                      c27*(n-1)
        }                                                                        c28*(n-1)
        tiempo = omp_get_wtime() - tiempo;                                       c29
        return tiempo;                                                           c30
    }                                                                            c31
    int division = ceil((tamano - 1) / 1.7);                                     c32
    if(continv >= division || contdir >= division){                              c33
        QuickSort(arreglo);                                                      c34*log(n)*n
    }else{                                                                       c35
        QuickSortOriginal(arreglo);                                              c36*log(n)*n
    }                                                                            c37
    tiempo = omp_get_wtime() - tiempo;                                           c38
    return tiempo;                                                               c39
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

Análisis del Algoritmo Forma Paralela

Para hallar la complejidad en el modelo PRAM se halla igual que en RAM y se divide entre la cantidad de hilos que posee el procesador donde se ejecuta el programa.

```
int partition_original(Vector* arr, int low, int high){
```

```
    int i = low - 1;                                                    c1
    int pivot = VECTOR_ver_elemento_en_posicion_a(*arr, high);          c2
    int j, elemento;                                                    c3
    for(j = low; j < high; j++){                                        c4*(n)
        elemento = VECTOR_ver_elemento_en_posicion_a(*arr, j);          c5*(n-1)
        if(elemento <= pivot){                                          c6*(n-1)
            i = i + 1;                                                  c7*(n-1)
            VECTOR_intercambio(arr, i, j);                              c8*(n-1)
        }                                                               c9*(n-1)
    }                                                                   c10
    i = i + 1;                                                          c11
    VECTOR_intercambio(arr, i, high);                                   c12
    return i;                                                           c13
}
```

Por lo tanto la complejidad de esta función es O(n) siendo n = high - low

```
void quickSort_original(Vector* arr, int low, int high){
        if(low < high){                                                 c1*log(n)
        int pi = partition(arr, low, high);                             c2*n*(log(n)-1)
        if(omp_get_num_threads() < omp_get_max_threads() ){             c3*(log(n)-1)
            #pragma omp parallel
            {
                #pragma omp single nowait
                {
                    quickSort(arr, low, pi - 1);                        c4*(log(n)-1)
                }
                #pragma omp single nowait
                {
                    quickSort(arr, pi + 1, high);                       c5*(log(n)-1)
                }
            }
        }else{
            quickSort(arr, low, pi - 1);                                c6*(log(n)-1)
            quickSort(arr, pi + 1, high);                               c6*(log(n)-1)
        }

    }
}
```

Por lo tanto la complejidad de esta función es 1/pO(nlog(n)), siendo p la cantidad de hilos que posee el procesador donde se ejecuta el programa.

```
void QuickSortOriginal(Vector* A){
    int extremo_derecho = VECTOR_tamano_vector(*A) - 1;                 c1
    quickSort_original(A, 0, extremo_derecho);                          c2*log(n)*n
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
int partition(Vector* arr, int low, int high){
    int i = low - 1;                                                    c1
    int pivot = VECTOR_ver_elemento_en_posicion_a(*arr, high);          c2
    int j, elemento;                                                    c3
    for(j = low; j < high; j++){                                        c4*(n)
        elemento = VECTOR_ver_elemento_en_posicion_a(*arr, j);          c5*(n-1)
        if(elemento <= pivot){                                          c6*(n-1)
            i = i + 1;                                                  c7*(n-1)
            VECTOR_intercambio(arr, i, j);                              c8*(n-1)
        }                                                               c9
```

```
    }                                                                   c10
    i = i + 1;                                                          c11
    VECTOR_intercambio(arr, i, high);                                   c12
    return i;                                                           c13
}
```

Por lo tanto la complejidad de esta función es O(n) siendo n = high - low

```
void quickSort(Vector* arr, int low, int high){
    if(low < high){                                                     c1*log(n)
        int pi = partition(arr, low, high);                             c2*n*(log(n)-1)
        if(omp_get_num_threads() < omp_get_max_threads() ){             c3*(log(n)-1)
            #pragma omp parallel
            {
                #pragma omp single nowait
                {
                    quickSort(arr, low, pi - 1);                        c4*(log(n)-1)
                }
                #pragma omp single nowait
                {
                    quickSort(arr, pi + 1, high);                       c5*(log(n)-1)
                }
            }
        }else{
            quickSort(arr, low, pi - 1);                                c6*(log(n)-1)
            quickSort(arr, pi + 1, high);                               c6*(log(n)-1)
        }

    }
}
```

Por lo tanto la complejidad de esta función es 1/pO(nlog(n)), siendo p la cantidad de hilos que posee el procesador donde se ejecuta el programa.

```
void QuickSort(Vector* A){
    int extremo_derecho = VECTOR_tamano_vector(*A) - 1;                 c1
    quickSort(A, 0, extremo_derecho);                                   c2*log(n)
}
```

Por lo tanto la complejidad de esta función es O(nlog(n))

```
double DoubleQuickSort(Vector* arreglo){
    double tiempo = omp_get_wtime();                                    c1
    int i;                                                              c2
    int contdir=0;                                                      c3
    int continv=0;                                                      c4
    int tamano = VECTOR_tamano_vector(*arreglo);                        c5
    #pragma omp parallel for private(i) reduction(+:contdir)
    for (i = 1; i < tamano; i++) {                                      c6*(n)
        int elemento1 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i-1);   c7*(n-1)
        int elemento2 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i);     c8*(n-1)
        if(elemento1 <= elemento2){                                     c9*(n-1)
            contdir += 1;                                               c10*(n-1)
        }                                                               c11*(n-1)
    }                                                                   c12
    if(contdir == tamano - 1){                                          c13
        tiempo = omp_get_wtime() - tiempo;                              c16
        return tiempo;                                                  c15
    }                                                                   c16
```

```
    #pragma omp parallel for private(i) reduction(+:continv)
    for (int i = 0; i < tamano; i++) {                                          c17*(n)
        int elemento1 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i-1);       c18*(n-1)
        int elemento2 = VECTOR_ver_elemento_en_posicion_a(*arreglo, i);         c19*(n-1)
        if(elemento1 >= elemento2){                                             c20*(n-1)
            continv += 1;                                                       c21*(n-1)
        }                                                                       c22*(n-1)
    }                                                                           c23
    if(continv == tamano - 1){                                                  c24
        int longitud = floor(tamano / 2); //Redondeo hacia arriba               c25
        #pragma omp parallel for private(i)
        for(i = 0; i < longitud; i++){                                          c26*(n)
            VECTOR_intercambio(arreglo, i, tamano - 1 - i);                     c27*(n-1)
        }                                                                       c28*(n-1)
        tiempo = omp_get_wtime() - tiempo;                                      c29
        return tiempo;                                                          c30
    }                                                                           c31
    int division = ceil((tamano - 1) / 1.7);                                    c32
    if(continv >= division || contdir >= division){                             c33
        QuickSort(arreglo);                                                     c34*log(n)*n
    }else{                                                                      c35
        QuickSortOriginal(arreglo);                                             c36*log(n)*n
    }                                                                           c37
    tiempo = omp_get_wtime() - tiempo;                                          c38
    return tiempo;                                                              c39
}
```

Por lo tanto la complejidad de esta función es 1/pO(nlog(n)), siendo p la cantidad de hilos que posee el procesador donde se ejecuta el programa.