Jorge Constanzo De la Vega Carrasco
A01650285

Primer Parcial

Los algoritmos que seleccioné para el examen parcial fueron: Merge Sort (Recursivo), Quick Sort (Divide y Vencerás) e Insertion Sort (cíclico).

Merge Sort (Recursivo)

<u>Pseudocódigo:</u>

```
function merge sort (list m)
   Base case. A list of zero or one elements is sorted, by definition.
   if length of m ≤ 1 then
      return m

   var left: = empty list
   var right: = empty list
   for each x with index i in m do
      if i < (length of m)/2 then
         add x to left
      else
         add x to right

   left: = merge sort(left)
   right: = merge sort(right)

   return merge (left, right)
```

<u>Complejidad</u>

Peor Caso:

Merge Sort

Worst case

$$T(N) = 2 + \left(\frac{n}{2}\right) + N - 1$$
$$T(N) = 2\left[2 + \left(\frac{n}{4}\right) + \left(\frac{n}{2}\right) - 1\right] + N - 1$$
$$T(N) = 4\left[2 + \left(\frac{N}{8}\right) + \frac{N}{4} - 1\right] + 2N - 3$$
$$T(N) = 8 + \left(\frac{N}{8}\right) + N + N + N - 4 - 2 - 1$$
$$T(N) = 2^k + \left(\frac{N}{2^k}\right) + kN - (2^k - 1)$$
$$T(1) = 0$$
$$2^k = N$$
$$k = \log_2 N$$
$$T(N) = N\log\frac{N}{2} - N + 1$$
$$= N\log N$$

Mejor Caso:

Best case

$$T(N) = 2 + \left(\frac{N}{2}\right) + \frac{N}{2}$$
$$T(N) = 2\left[2 + \frac{N}{4} + \frac{N}{4}\right] + \frac{N}{2}$$
$$T(N) = 4\left(2 + \left(\frac{N}{8}\right) + \frac{N}{8}\right] + N$$
$$T(N) = 2^k + \left(\frac{N}{2^k}\right) + \frac{kN}{2}$$
$$T(N) = \frac{N}{2}\log_2 N$$
$$= N\log N$$

Caso Intermedio:

Average case

$T(N) = N \log N$

Caso base $= N=1$

$$T(2N) = 2T(N) + 2N$$
$$= 2N \log N + 2N$$
$$= 2N(\log(2N) - 1) + 2N$$
$$= 2N \log(2N)$$
$$T = N \log N$$

En los 3 casos (Mejor, Peor e Intermedio) la complejidad es la misma: N log N

Implementación en Python de Merge Sort:

```python
# Merge Sort(Recursivo)
def mergeSort(arr):

    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]

        #recursion
        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0

        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                arr[k]=lefthalf[i]
                i=i+1
            else:
                arr[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            arr[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            arr[k]=righthalf[j]
            j=j+1
            k=k+1

    return arr
```

Quick Sort (Divide y Vencerás)

<u>Pseudocodigo:</u>

```
Algorithm quicksort (A, lo, hi) is
    if lo < hi then
        p: = partition (A, lo, hi)
        quicksort (A, lo, p - 1)
        quicksort (A, p + 1, hi)

Algorithm partition (A, lo, hi) is
    pivot: = A[hi]
    i: = lo - 1
    for j: = lo to hi - 1 do
        if A[j] < pivot then
            i: = i + 1
            swap A[i] with A[j]
    swap A [i + 1] with A[hi]
    return i + 1
```

<u>Complejidad:</u>

Mejor Caso:

Best case

$k = \frac{n}{2} \quad n-k = \frac{n}{2}$

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + \alpha n \\ &= 2(2T(\tfrac{n}{4} + \alpha\tfrac{n}{2}) + \alpha \\ &= 2^2 T(\tfrac{n}{4}) + \alpha 2n \\ &= 2^2(2T(\tfrac{n}{8}) + \alpha\tfrac{n}{4}) + \alpha 2n \\ &= 2^3 T(\tfrac{n}{8}) + 3\alpha n \qquad n = 2^k \\ &= 2^k T(\tfrac{n}{2^k}) + k\alpha n \qquad k = \log n \\ &= nT(1) + \alpha n \log n \\ &= n \log n \end{aligned}$$

Peor Caso:

Worst case

$T(1) = 1 \quad k = n-1$

$$\begin{aligned} T(n) &= T(q-1) + T(n-q) + N \\ T(n) &= T(0) + T(n-1) + n = 1 + T(n-1) + n \\ &= T(n-1) + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \end{aligned}$$

$\frac{C_N}{N+1}$

$$T(n-k) + \sum_{i=-1}^{k-2} (n-i)$$

$$T(n) \leq T(1) + \sum_{i=-1}^{n-3} (n-i)$$

$$= \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} = n^2$$

Caso Intermedio:

Average case

$$C_n = N+1+\frac{1}{N}\sum_{1\le k\le N}(C_{k-1}+C_{n-k})$$

$$= N+1+\frac{1}{N}\sum_{1\le k\le N}(C_{k-1}+C_{n-k}$$

$$C_n = N+1+\frac{2}{N}\sum_{1\le k\le N} C_{k-1}$$

$$NC_n-(N-1)C_{n-1} = N(N+1) - (N-1)N+2C_{n-1}$$

$$NC_N = (N+1)C_{N-1}+2N$$

$$NC_N = (N+1)C_{N-1}+2N$$

$$NC_N = (N+1)C_{n-1}+2N$$

$$\frac{C_N}{N+1} = \frac{C_{n-1}}{N}+\frac{2}{N+1}$$

$$= \frac{C_{n-2}}{N-1}+\frac{2}{N}+\frac{2}{N+1}$$

$$= \frac{C_2}{3}+\sum_{3\le k\le N}\frac{2}{k+1}$$

$$\frac{C_N}{N+1} = 2\sum\frac{1}{k} = 2\int^{N}\frac{1}{x}\,dx = 2\ln N$$

$$2N\ln N = 1.39N\log N$$

$$= n\log n$$

En el intermedio y mejor caso el resultado fue N Log N, en el peor de los casos fue n^2

<u>Implementación en Python de Quick Sort:</u>

```python
# Quicksort(Divide y Venceras)
def partition(arr, begin, end):
    pivot = begin
    for i in range(begin+1, end+1):
        if arr[i] <= arr[begin]:
            pivot += 1
            arr[i], arr[pivot] = arr[pivot], arr[i]
    arr[pivot], arr[begin] = arr[begin], arr[pivot]
    return pivot



def quicksort(arr, begin=0, end=None):
    if end is None:
        end = len(arr) - 1
    def _quicksort(arr, begin, end):
        if begin >= end:
            return
        pivot = partition(arr, begin, end)
        _quicksort(arr, begin, pivot-1)
        _quicksort(arr, pivot+1, end)
    return _quicksort(arr, begin, end)
```

Insertion Sort (Cíclico)

Pseudocódigo:

```
insertion Sort (A)
    for j = 2 to n
        key ← A [j]
        // Insert A[j] into the sorted sequence A  [1 … j-1]
        j ← i – 1
       while i > 0 and A[i] > key
            A[i+1] ← A[i]
             i ← i – 1
       A[j+1] ← key
```

Complejidad:

Peor Caso:

Insertion sort

Worst case

$$\sum_{i=1}^{n-1} i = 1+2+3+\dots+(n-1) = \frac{(n-1)N}{2}$$
$$= \frac{1}{2}(n^2-n)$$
$$= O(N^2)$$

Mejor Caso:

Best case

$$\sum_{i=1}^{n-1} 1 = n-1 = \Omega(N)$$

Caso Intermedio:

Average case

$$\sum_{i=1}^{N-1} \frac{i+1}{2} = \frac{(N-1)N}{4} + \frac{N-1}{2}$$
$$= \frac{(N-1)(N+2)}{4}$$
$$= \Theta(N^2)$$

En el peor e intermedio de los casos la complejidad fue la misma: n^2. En el mejor caso fue: n.

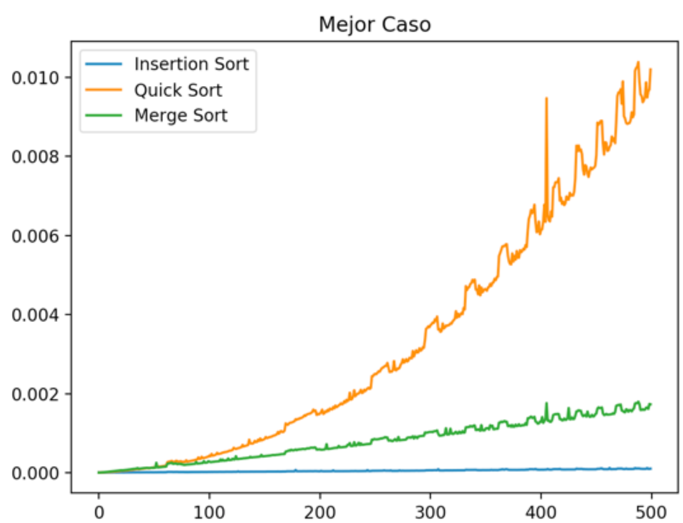
Implementación en Python de Insertion Sort:

```python
# Insertion Sort(Ciclico)
def insertionSort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
    return arr
```

Graficas

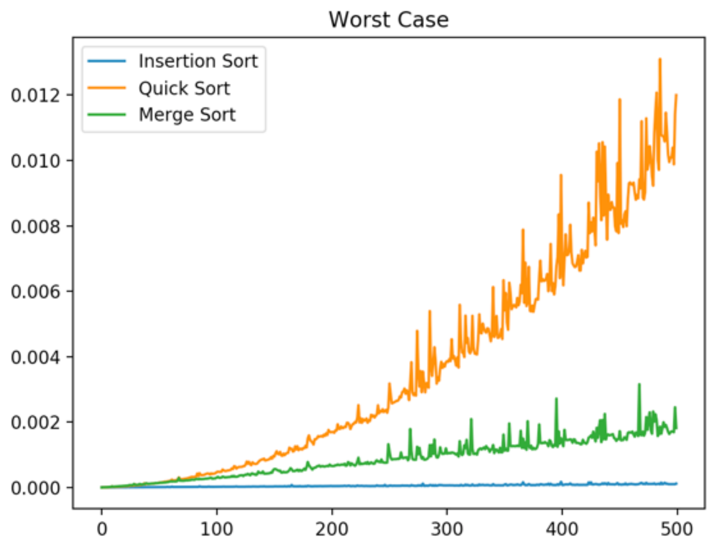
Las gráficas están hechas de 0 a n. El lado X de la gráfica se compone por el tamaño del arreglo y el lado Y por el tiempo procesado.

Mejor Caso:



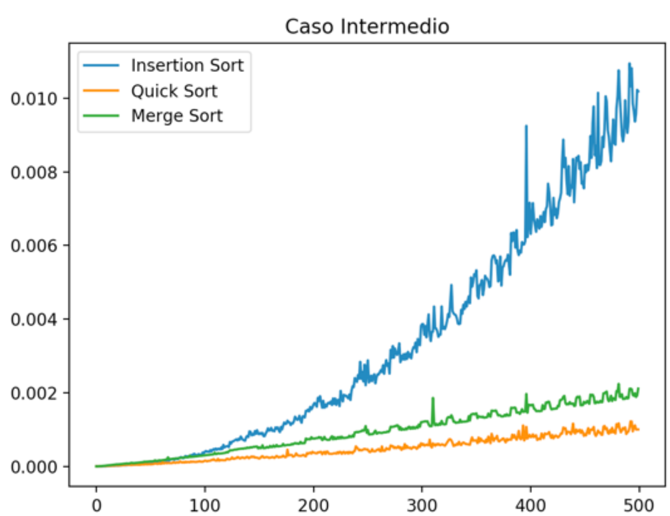
En el mejor de los casos Quick Sort toma el mayor tiempo de ejecución, además de tener mucho ruido en su gráfica. En el caso de Merge Sort, se encuentra en un rango intermedio entre los 3 algoritmos de ordenamiento, con poco ruido y finalmente Insertion Sort es el que más rápido ordena pues se mantiene casi en una lista recta.

Peor Caso:



En el peor caso el algoritmo que más tiempo consume en procesar los datos es Quick Sort, este contiene mucho ruido. El algoritmo Merge Sort se encuentra en medio de los otros 2 algoritmos de ordenación, contiene ruido considerable. Finalmente, Insertion Sort es el algoritmo que procesa los datos de forma más rápida en el peor de los casos.

Caso Intermedio:

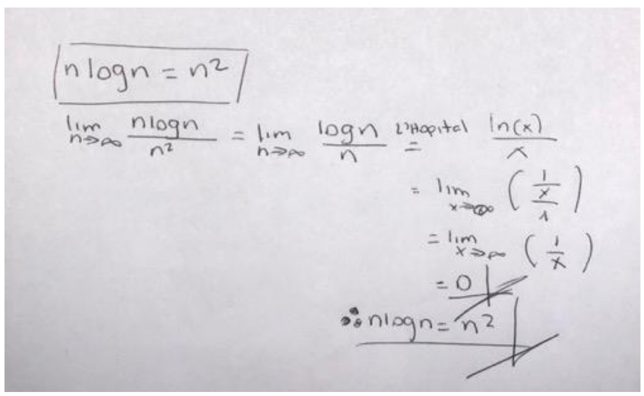


En el caso intermedio el algoritmo Insertion Sort es el que más tarda en ordenar el arreglo, no con tanto ruido en comparación de los otros casos al ser el más lento. Merge Sort es intermedio y no cuenta con mucho ruido, el algoritmo que más rápido procesa los datos es Quick Sort, este algoritmo se empareja mucho con Merge Sort y tienen un ruido aproximado entre ambos.

Intersecciones:

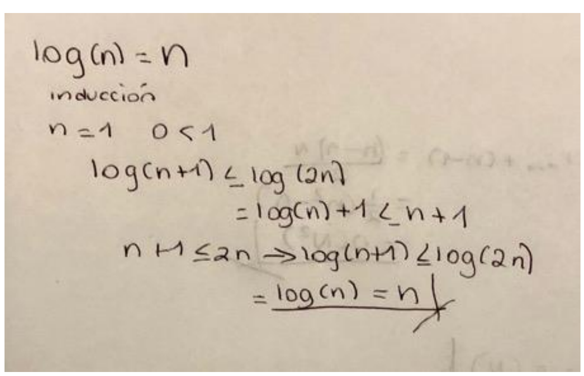
En la gráfica intermedia Insertion Sort y Merge Sort se intersectan entre 0 y 100.
Lo mismo que en Quick Sort y Merge sort en el peor de los casos, ambas tienen la misma complejidad que en Insertion Sort por lo que la intersección es la misma.

N log N = n^2



En el mejor de los casos Quick Sort se empalma con Merge Sort. Aquí la diferencia es que Quick Sort la complejidad es N log N con variación con N. Debido a que la variación es N entonces puede encontrarse la intersección.

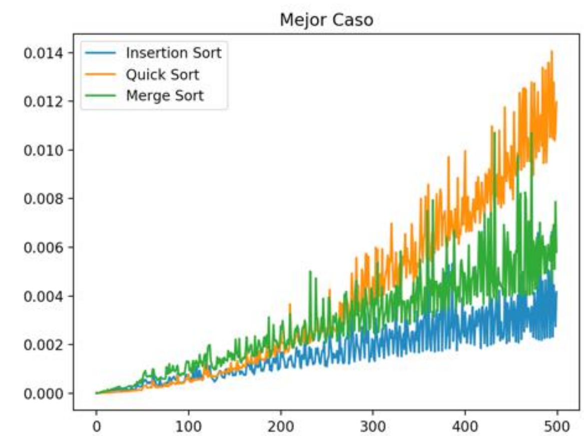
N log N = n



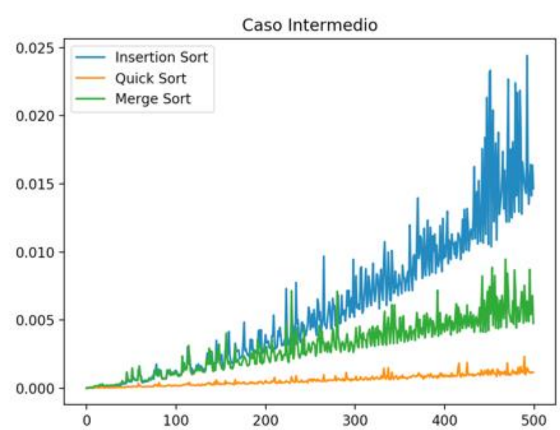
Velocidad de Ejecución

<u>Aumentando la Velocidad de Ejecución a 0.00005</u>

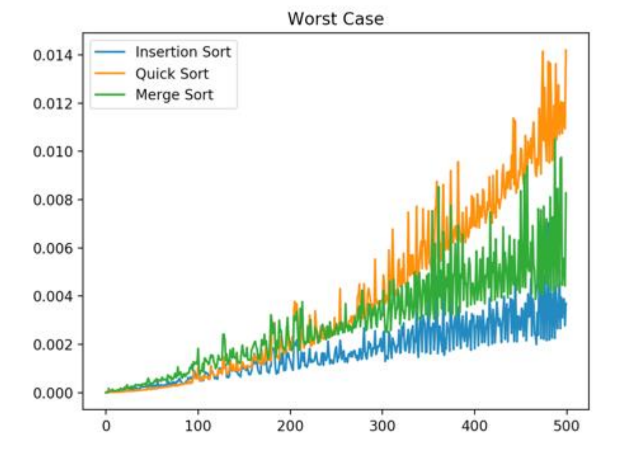
Mejor Caso:



Caso Intermedio:



Peor Caso:



Como se puede apreciar en las 3 gráficas, el crecimiento asintótico continua a pesar de que ahora existe más ruido que antes. Lo que hay que hacer notar es que ahora es más evidente las intersecciones que hay entre las gráficas. Los resultados de las intersecciones se encuentran en la parte de arriba.