SelectionSort

for i in range(len(lista)): #Percore o tamanho da lista O(N) com relação ao tamanho da entrada

menor = i

for j in range(i+1,len(lista)): percorre de i+1 até o tamanho da lista, sendo assim um somatório de n-1+n-2+n-3+n-4....+1

if lista[j] < lista[menor]:

menor = j

if lista[i] != lista[menor]:

aux = lista[i]

lista[i] = lista[menor]

lista[menor] = aux

Logo, por pa(Progressão Aritmética) temos (1+(n−1)), dentro do laço interno e n no laço externo, sendo o resultado final, (1+(n−1))\*n, logo O(n2).

Não utiliza memoria auxiliar.

Merge Sort, se analisando os ponto principais

def mergeSort(lista):

if len(lista) > 1:

meio = len(lista)//2

listaDaEsquerda = lista[:meio]

listaDaDireita = lista[meio:]

mergeSort(listaDaEsquerda) Chamado do tipo T(n/2)

mergeSort(listaDaDireita) Chamado do tipo T(n/2)

i = 0

j = 0

k = 0

while i < len(listaDaEsquerda) and j < len(listaDaDireita): os códigos abaixo percorrem, em contextos diferentes, todos valores presentes na lista, sendo o passo do merge.

if listaDaEsquerda[i] < listaDaDireita[j]:

lista[k]=listaDaEsquerda[i]

i += 1

else:

lista[k]=listaDaDireita[j]

j += 1

k += 1

while i < len(listaDaEsquerda):

lista[k]=listaDaEsquerda[i]

i += 1

k += 1

while j < len(listaDaDireita):

lista[k]=listaDaDireita[j]

j += 1

k += 1

return lista

Logo, temos que a complexidade é o das chamadas recursivas mais o merge, respectivamente, sendo 2T(n/2) + O(n), pelo teorema mestre simplificado, O(n)=n Log n.

E possui complexidade N para o espaço, pois as chamadas recursivas em sua própria estrutura demandam de memoria auxiliar e existe o vetor auxiliar de tamanho n

Radix Sort

def radix\_sort(lista):

tamanho\_maximo = max([len(str(palavra)) for palavra in lista])

for pos in range(tamanho\_maximo\*1, 1, -1): Percorre um loop de tamanho máximo igual ao numero de caracteres do dado, vamos chamar de k

baldes = [list() for y in range(MAX\_CHARS)]

for palavra in lista: Percorre todas os dados da lista original, complexidade do tamanho da entrada

balde = numero\_do\_balde(palavra, pos)

baldes[balde] += [palavra]

lista = sum(baldes, [])

return lista

def numero\_do\_balde(palavra, pos):

if (pos >= len(str(palavra))): return 0

ch = str(palavra)[pos]

if (ch >= 'A' and ch <= 'Z'): return ord(ch) - ord('A') + 1

if (ch >= 'a' and ch <= 'z'): return ord(ch) - ord('a') + 1

return MAX\_CHARS-1

Logo, temos complexidade de tempo igual ao produto do numero de caracteres do dado e o tamanho da entrada, sendo, O(nk).

E com relação a espaço temos complexidade de O(n+s), n ou s, onde temos um vetor auxiliar que “conta” o número de ocorrências, o tamanho desse vetor vai se igual o tamanho da entrada ou igual o tamanho do alfabeto.