INF01124 - Classificação e Pesquisa de Dados - Exercício 3

Felipe de Almeida Graeff
00261606

1 Hybrid Sort

A implementação do merge sort desenvolvida pode ser conferida abaixo. Para a utilização híbrida do algoritmo de merge com o de inserção, é avaliado se o tamando da sublista que está sendo ordenada é menor ou igual ao argumento k fornecido. Caso positivo a sublista é ordenada utilizando-se o algoritmo de insertion sort fornecido, com algumas modificações para a utilização da classe *vector* do C++.

A escolha do valor ótimo k depende de diversos fatores, como linguagem de programação e implementação utilizadas. O merge sort utiliza recursão e , quando não é feito in-place, memória auxiliar. O overhead gerado pelas chamadas recursivas e pelo gerenciamento da memória auxiliar pode fazer com que, pricipalmente para entradas pequenas, o insertion sort seja mais rápido. O valor ideal de k é o menor tamanho de entrada em que, mesmo com o overhead, o merge sort termina em menos tempo que o insertion sort. Como esse overhead varia de acordo com a linguagem e a implementação, esse valor só pode ser obtido através de experimentação.

Para o cômputo dos tempos de execução mostrados no gráfico abaixo foi utilizado o valor arbitrário 10 para k.

Listing 1: merge_sort.hpp

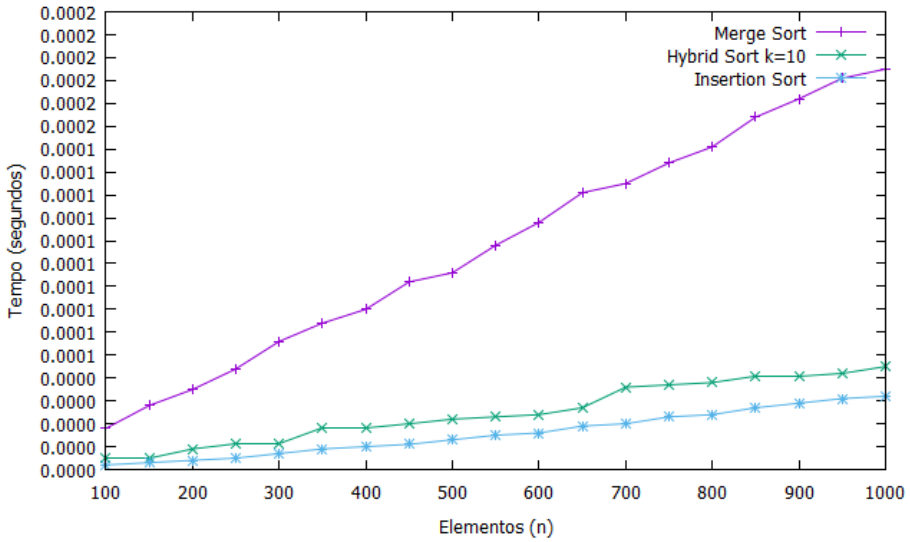
```
8  void mergeSort(std::vector<int>& l,int k=0);
9  void mergeSort(std::vector<int>& l,int k,int low,int high);
```

Listing 2: merge_sort.cpp

```
48 void mergeSort(std::vector<int>& l,int k){
49     mergeSort(l,k,0,l.size()-1);
50 }
51
52 void mergeSort(std::vector<int>& l,int k,int low,int high){
53     int size = high - low + 1;
54     if(size <= k){
55         binary_insertion_sort(l,low,high);
56     }else if(size > 1){
57         int middle = size / 2;
58         mergeSort(l,k,low,middle+low-1);
59         mergeSort(l,k,low+middle,high);
```

```
60
61              std::vector<int> aux(size,0);
62
63              std::merge(l.begin()+low,l.begin()+low+middle,
64                      l.begin()+low+middle,l.begin()+high+1,
65                      aux.begin());
66
67              for(int i = 0, j = low; i < size; i++, j++){
68                  l[j] = aux[i];
69              }
70          }
71      }
```

Figura 1: Tempos de execução



2 Radix Sort

A implementação do radix sort foi feita em Python e pode ser vista abaixo.
O código feito pode ser dividido em 3 partes:

1. Para ordenar uma lista de inteiros positivos é executado o trecho da linha 13 até a linha 23. Primeiramente é calculada a quantidade de iterações necessárias para a ordenação baseando-se no número de dígitos do maior elemento da lista. Após isso o algoritmo de radix sort propriamente dito é executado.

2. Para números de tipo *float* foi utilizado o princípio de que, para números em ponto flutuante de precisão simples, ou seja, de 32 bits, a precisão máxima é de 6 casas decimais. Portanto, nas linhas 11 e 12, todos os números da lista são multiplicados por 10^6 e transformados em inteiros. Assim obtem-se uma lista de inteiros e pode-se utilizar o mesmo algoritmo do item anterior. Ao final, nas linhas 24 e 25, todos os elementos da lista

já ordenada são divididos por 10^6 para que se obtenha os valores originais novamente.

3. Por fim, para poder ordenar também números negativos, da linha 4 à linha 9 os elementos da lista inicial são divididos em uma lista de positivos e outra de negativos. O algoritmo dos itens anteriores foi colocado em um laço for na linha 10 que executa para as duas listas. Ao final do algoritmo a lista ordenada de positivos é concatenada à lista ordenada de negativos e retornada.

Listing 3: cpdsort.py

```
3  def radixSort(li,radix=10):
4      l2 = [[],[]]
5      for i in li:
6          if i < 0:
7              l2[0].append(i)
8          else:
9              l2[1].append(i)
10     for l in l2:
11         for i,j in enumerate(l):
12             l[i] = int(j * (10**6))
13         try:
14             passes = int(round(log(max(abs(min(l)),max(l)),radix)) + 1)
15         except ValueError:
16             passes = 0
17         for i in range(passes):
18             buckets = [[] for _ in range(radix)]
19             for n,x in enumerate(l):
20                 buckets[x // radix**i % radix].append(x)
21             l.clear()
22             for m in buckets:
23                 l.extend(m)
24         for i,j in enumerate(l):
25             l[i] = j / (10**6)
26     return l2[0] + l2[1]
```

Como o número de iterações do radix sort depende da quantidade de dígitos dos números a serem ordenados, uma representação binária de um float exigiria um pior caso de 32 iterações, portanto uma representação em bases maiores executaria, em teoria, mais rapidamente.
Em geral, para dados númericos, a ordem lexicográfica obtida com a ordenação MSD não é interessante, então o algoritmo LSD seria preferível nesse caso.