Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Estruturas de Dados Básicas I

Arthur Ferreira de Holanda
José Maia Da Silva Neto

Análise Empırica de Algoritmos de Ordenação

Natal
2023

Arthur Ferreira de Holanda
José Maia Da Silva Neto

Análise Empırica de Algoritmos de Ordenação

Relatório de aula prática apresentado como avaliação da disciplina Estruturas de Dados Básicas I ministrada pelo professor Dr. Selan Rodrigues Dos Santos para o curso de Bacharelado em Tecnologia Da Informação do Instituto Metrópole Digital da Universidade Federal do Rio Grande do Norte - Campus Central (Natal).

Natal
2023

Sumário

1 Introdução

1.1 Propósito

Este relatório tem como propósito realizar um estudo por meio da análise empírica de algoritmos de ordenação, visando auxiliar na seleção do algoritmo ideal para realizar a ordenação de um arranjo. Serão examinados os seguintes algoritmos: insertion sort, selection sort, bubble sort, shell sort, quick sort, merge sort e radix sort. Através da avaliação desses algoritmos, será possível identificar suas características, desempenho e eficiência em relação à ordenação de conjuntos de dados. Com base nesses resultados, poderemos tomar decisões embasadas na escolha do algoritmo mais adequado para atender às necessidades específicas de ordenação.

2 Métodos

2.1 Metodologias

Foram realizados testes utilizando seis tipos de amostras: não crescente, não descrescente, totalmente aleatória, ordenada em 25%, ordenada em 50% e ordenada em 75%. Cada algoritmo de ordenação foi executado 25 vezes para cada tipo de amostra. O tamanho inicial dos elementos foi de 100 e foi incrementado em 4000 elementos até alcançar um total de 100000 elementos. Os resultados desses testes foram registrados e armazenados em um arquivo de dados.

2.2 Materiais

As características técnicas do computador são: um processaor intel®Pentium(R) dual CPU T2330@1.60GHZx2, 3.0 Giga bytes de memoria RAM e 120.0 Giga bytes de armazenamento. O sistema operacional é do tipo 64-bit, de nome Ubuntu 22.04.2LTS.A principal linguagem de programação adotada foi C++ com o auxilio de CMake. Os algoritmos de ordenação estudados a seguir foram insertion sort, selection sort, bubble sort, shell sort, quick sort, merge sort e radix sort.

2.3 Algoritimos

2.3.1 Insertion sort

O insertion sort é um algoritmo de ordenação simples. Ele percorre uma lista de elementos, inserindo cada elemento na posição correta em relação aos elementos anteriores já ordenados. A lista é percorrida da esquerda para a direita, e em cada iteração, um elemento é comparado com os anteriores e colocado em sua posição correta. O processo continua até que toda a lista esteja ordenada. Embora seja fácil de entender e implementar, o insertion sort pode ser lento para grandes conjuntos de dados, mas funciona bem para listas pequenas ou quase ordenadas. Segue o codigo:

```
template <typename T>
void insertion(T *first, T *last)
{
    if (first == last)
        return;

    for (int *fast = first + 1; fast != last; ++fast) {
        int val = *fast;
        int *runner = fast;
        while (runner != first && val < *(runner - 1)) {
```

```
            *runner = *(runner - 1);
            --runner;
        }
        *runner = val;
    }
}
```

2.3.2 Selection sort

O selection sort é um algoritmo de ordenação simples. Ele percorre a lista em busca do menor elemento e o coloca na posição correta. A cada iteração, o menor elemento é selecionado e trocado com o próximo elemento da lista não ordenada. Esse processo continua até que toda a lista esteja ordenada. Embora seja fácil de entender e implementar, o selection sort também tem um desempenho lento para grandes conjuntos de dados. No entanto, é eficiente para listas pequenas. Segue o codigo:

```
template <typename T>
void selection(T *first, T *last)
{
    int min_idx;
    int n = distance(first, last);

    for(int i = 0; i < n-1; i++){
        min_idx = i;
        for(int j = i + 1; j < n; j++){
            if(*(first+j) < *(first+min_idx))
                min_idx = j;
        }
        if(min_idx != i)
            iter_swap(first+min_idx, first+i);
    }
}
```

2.3.3 Bubble sort

O selection sort é um algoritmo de ordenação simples. Ele percorre a lista em busca do menor elemento e o coloca na posição correta. A cada iteração, o menor elemento é selecionado e trocado com o próximo elemento da lista não ordenada. Esse processo continua até que toda a lista esteja ordenada. Embora seja fácil de entender e implementar, o selection sort também tem um desempenho lento para grandes conjuntos de dados. No entanto, é eficiente para listas pequenas. Segue o codigo:

```
template <typename T>
void bubble(T *first, T *last)
{
    int n = distance(first, last);
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < n - i - 1; j++){
            if(*(first+j) > *(first+(j+1)))
                iter_swap(first+j, first+(j+1));
        }
    }
}
```

2.3.4 Shell sort

O shell sort é um algoritmo de ordenação que divide a lista em grupos menores e realiza comparações e trocas dentro desses grupos. Ele repete esse processo com grupos cada vez menores até que a lista esteja ordenada. Embora seja mais eficiente do que o bubble sort e o insertion sort, o shell sort ainda é menos eficiente do que outros algoritmos mais avançados. É uma opção viável quando a simplicidade de implementação é priorizada em vez do melhor desempenho. Segue o codigo:

```
template <typename T>
void shell(T *first, T *last)
{
  int tamanho = last - first;
  for (int gap = tamanho / 2; gap > 0; gap /= 2) {
    for (int i = gap; i < tamanho; i++) {
      int temp = first[i];
      int j;
      for (j = i; j >= gap && temp < first[j - gap]; j -= gap
         ) {
        first[j] = first[j - gap];
      }
      first[j] = temp;
    }
  }
}
```

2.3.5 Quick sort

O quick sort é um algoritmo de ordenação eficiente que divide a lista em subgrupos menores usando um pivô e rearranja-os de forma que os menores fiquem à esquerda

e os maiores à direita. Esse processo é aplicado recursivamente até que a lista esteja completamente ordenada. Embora possa ter um pior caso ineficiente, o quick sort é amplamente utilizado por seu bom desempenho médio em conjuntos de dados grandes. Segue o codigo:

```
template <typename T>
void quicksort(T *first, T *last)
{
  if (std::distance(first, last) >= 2) {
    auto* q = partition(first, last, last - 1);
    quicksort(first, q);
    quicksort(q + 1, last);
  }
}
```

O quick sort usa a função auxiliar partition. No quick sort, a função "partition"seleciona um pivô na lista e rearranja os elementos de forma que os menores fiquem à esquerda e os maiores à direita. A função retorna o índice do pivô, que é usado para dividir a lista em subgrupos menores. Essa divisão é repetida até que todos os subgrupos sejam de tamanho unitário, resultando na lista completamente ordenada. A função "partition"desempenha um papel fundamental na eficiência do quick sort.

```
template <typename T>
int* partition(T *first, T *last, T *pivot)
{
  auto slow = first;
  auto fast = first;
  while (fast < pivot) {
    if (*fast < *pivot) {
      std::iter_swap(slow, fast);
      slow++;
    }
    fast++;
  }
  std::iter_swap(pivot, slow);
  return slow;
}
```

2.3.6 Merge sort

O merge sort é um algoritmo de ordenação eficiente baseado na estratégia de "dividir para conquistar". Ele divide a lista em duas metades, recursivamente ordena cada metade e, em seguida, mescla as duas metades ordenadas para obter a lista final ordenada. segue

o codigo:

```
template <typename T>
void merge(T *first, T *last)
{
    auto len = std::distance(first, last);
    if (len >= 2) {
        auto* m = first + len / 2;
        merge(first, m);
        merge(m, last);

        T* temp = new T[len];
        merge_aux(first, m, m, last, temp);
        std::copy(temp, temp + len, first);
        delete[] temp;
    }
}
```

O merge sort usa a função auxiliar *merge_aux*. A função "mergeaux"no merge sort combina duas sublistas ordenadas em uma única lista ordenada. Ela compara os elementos das sublistas e os coloca em ordem crescente na lista resultante. A função "merge"é essencial para a eficiência e a corretude do merge sort.

```
template <typename T>
void merge_aux(T* l_first, T* l_last, T* r_first, T* r_last,
   T* a_first)
{
    while (l_first != l_last) {
        if (r_first == r_last) {
            std::copy(l_first, l_last, a_first);
            return;
        }
        *a_first++ = (*l_first < *r_first) ? *l_first++ : *
           r_first++;
    }
    std::copy(r_first, r_last, a_first);
}
```

2.3.7 Radix sort

O radix sort é um algoritmo de ordenação que classifica os elementos com base em seus dígitos. Ele agrupa os elementos em baldes de acordo com seus valores de dígitos e os rearranja até que a lista esteja ordenada. O radix sort é eficiente para números inteiros

ou strings alfanuméricas e sua complexidade de tempo é linear. Segue o codigo:

```
template <typename T>
void radixsort(T *first, T *last) {
  T max = get_max<int>(first, last);

  for(int place = 1; max / place > 0; place *= 10)
    countSort<int>(first, last, place);
}
```

O Radix sort usa a função auxiliar countSort. Ela conta o número de ocorrências de cada elemento e os coloca em suas posições corretas na lista ordenada. O countSort é repetidamente usado no radix sort para ordenar os elementos por dígito.

```
template <typename T>
void countSort(T *first, T *last, T place) {
    const int max = 100000;
    T output[max];
    T count[max];

    int n = distance(first, last);

    for(int i = 0; i < max; i++)
        count[i] = 0;

    for(int i = 0; i < n; i++)
        count[(*(first+i) / place) % 10]++;


    for(int i = 1; i < max; i++)
        count[i] += count[i - 1];

    for(int i = n - 1; i >= 0; i--) {
        output[count[(*(first+i) / place) % 10] - 1] = *(
           first+i);
        count[(*(first+i) / place) % 10]--;
    }

    for(int i = 0; i < n; i++)
        *(first+i) = output[i];
}
```

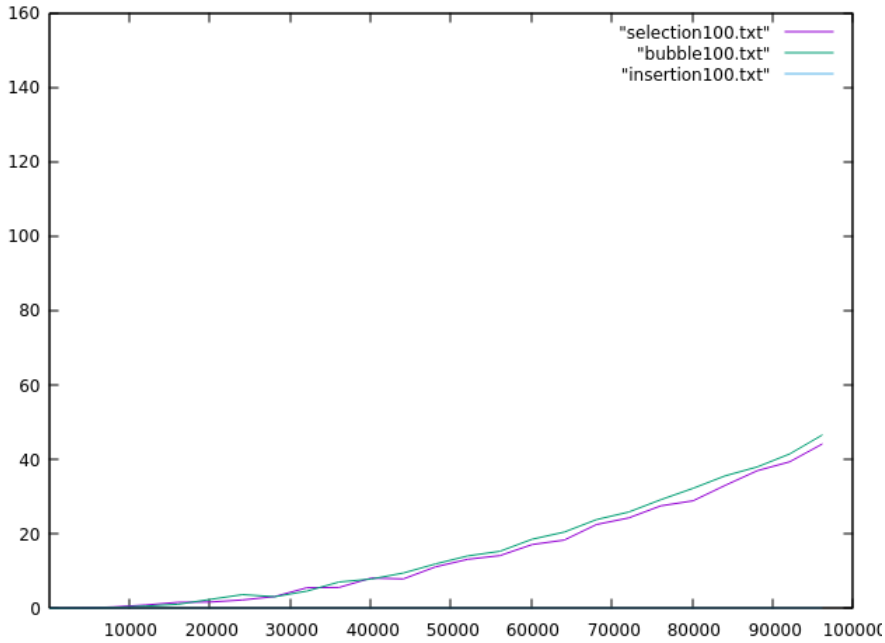
3 Resultados alcançados

3.1 Lista não decrescente

3.1.1 Quadráticas

Para listas não decrescente as funções quadraticas tiveram desempenho semelhante, Com o bubble sort se demonstrando mais demorado em relação aos outros.

Figura 1: Quadraticas

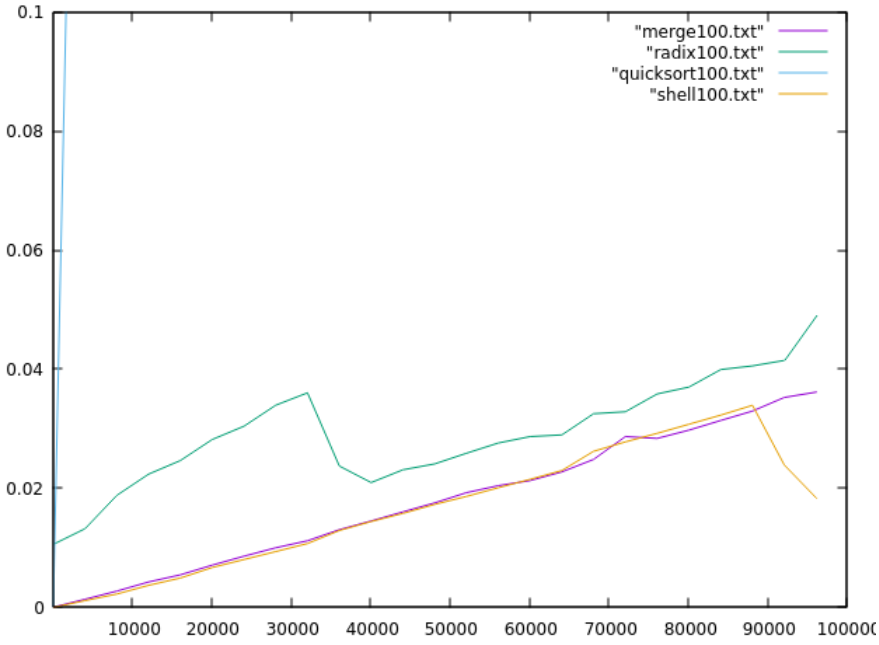


Fonte: Elaborado pelo autor (2023).

3.1.2 Radix e logarítmicos

Para listas não decrescente as funções logaritimicas variaram bastante, principalmente o quick sort que teve um pico muito alto, e o shell se mostrando mais rapido para as amostras maiores ao final.

Figura 2: logarıtmicas



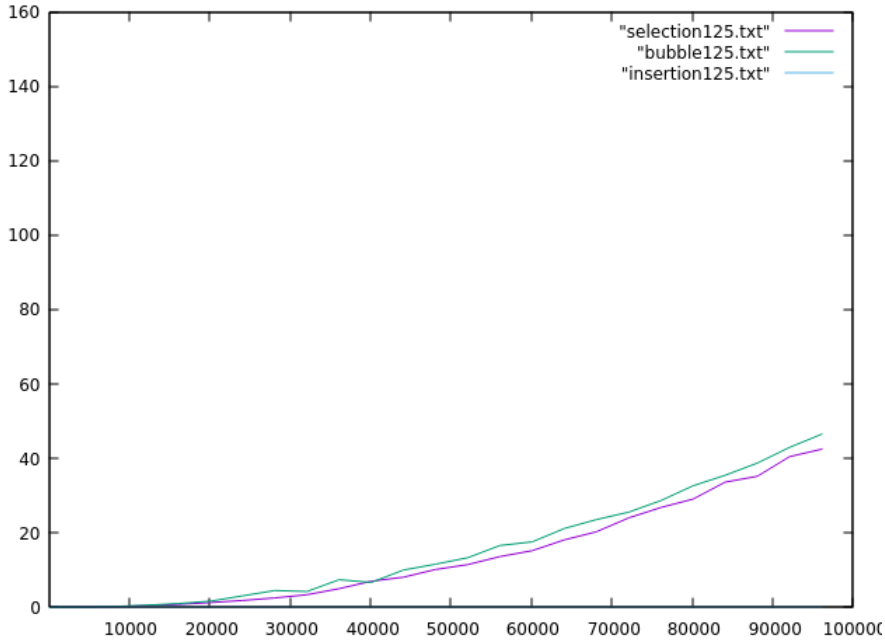
Fonte: Elaborado pelo autor (2023).

3.2 Lista não crescente

3.2.1 Quadráticas

Para listas não crescente as funções quadraticas tiveram desempenho semelhante, Com o bubble sort se demonstrando mais demorado em relação aos outros.

Figura 3: Quadraticas

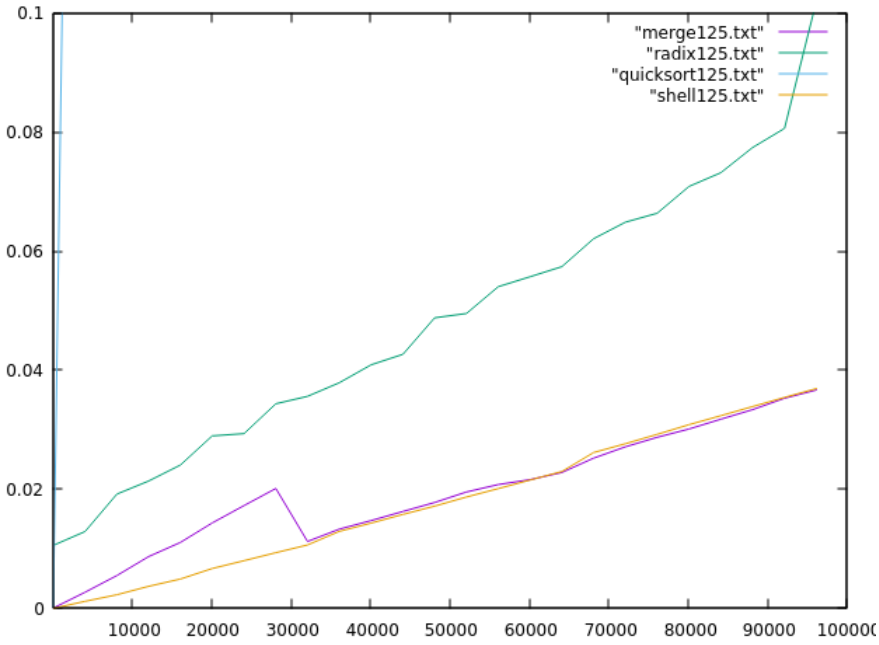


Fonte: Elaborado pelo autor (2023).

3.2.2 Radix e logarítmicos

Para listas não crescente as funções logaritimicas variaram bastante, principalmente o quick sort e o radix sort que demoravam muito mais do que o shell e o merge.

Figura 4: logarıtmicas



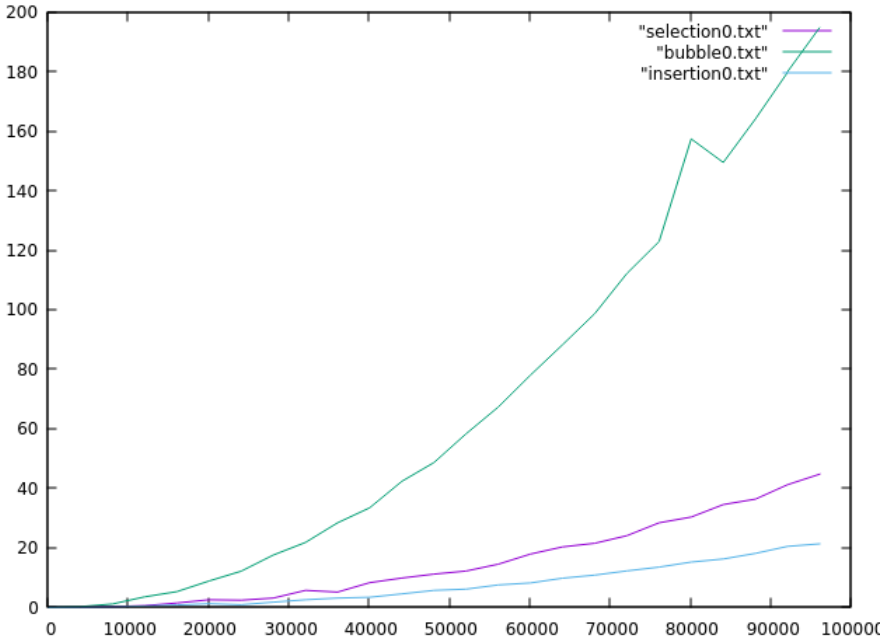
Fonte: Elaborado pelo autor (2023).

3.3 Lista totalmente embaralhada

3.3.1 Quadráticas

Para listas totalmente embaralhadas as funções quadraticas tiveram desempenho distinto, Com o bubble sort tendo uma taxa de crescimento muito maior em relação aos outros.

Figura 5: Quadraticas

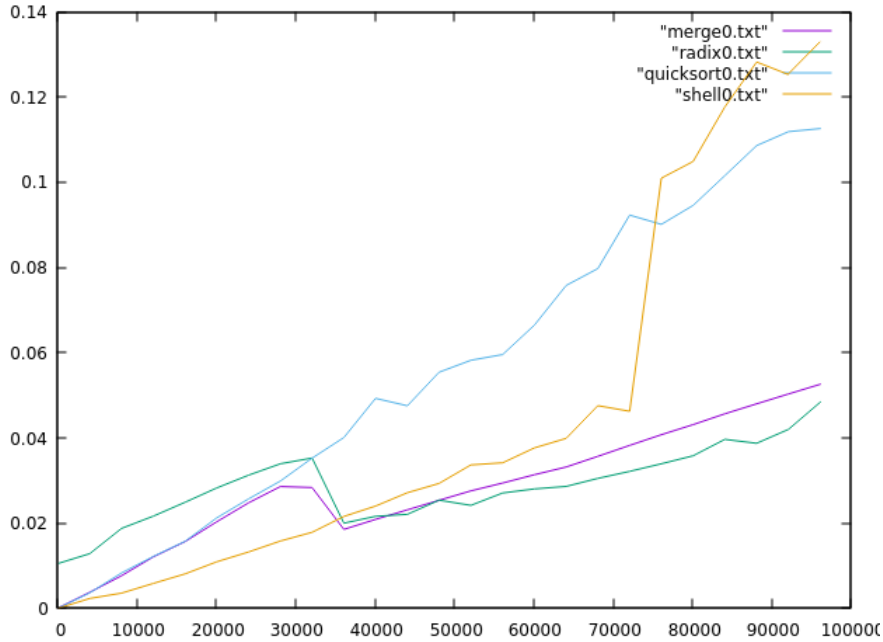


Fonte: Elaborado pelo autor (2023).

3.3.2 Radix e logarítmicos

Para listas totalmente embaralhadas as funções logaritimicas variaram bastante, com um pico abrupto na função shell sort e o radix que se demonstrou mais rapido que os outros com amostras maiores.

Figura 6: logarıtmicas



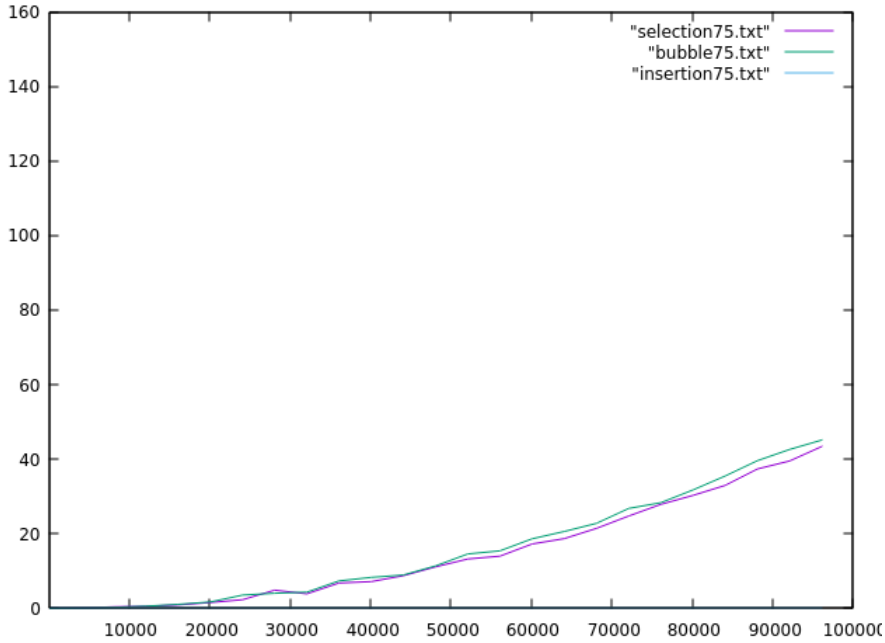
Fonte: Elaborado pelo autor (2023).

3.4 Lista 75% ordenada

3.4.1 Quadráticas

Para listas 75% ordenadas as funções quadraticas tiveram desempenho semelhante, tendo um crescimento continuo.

Figura 7: Quadraticas

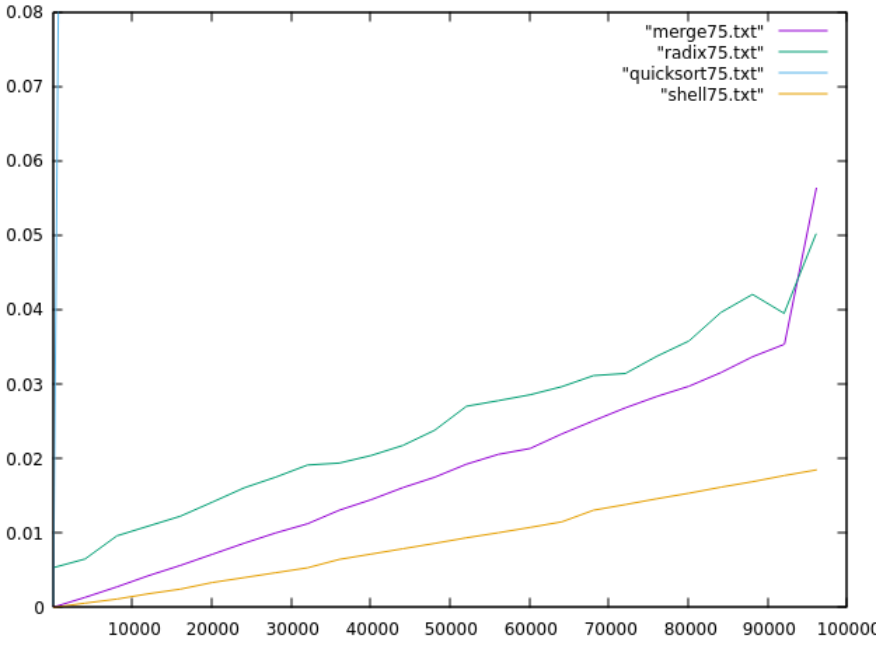


Fonte: Elaborado pelo autor (2023).

3.4.2 Radix e logarítmicos

Para listas 75% ordenadas as funções logaritimicas variaram bastante, com um pico muito alto desde o inicio da função quick sort, e um pico menor ao final da merge. Com a shell se demonstrando mais rapida.

Figura 8: logarıtmicas



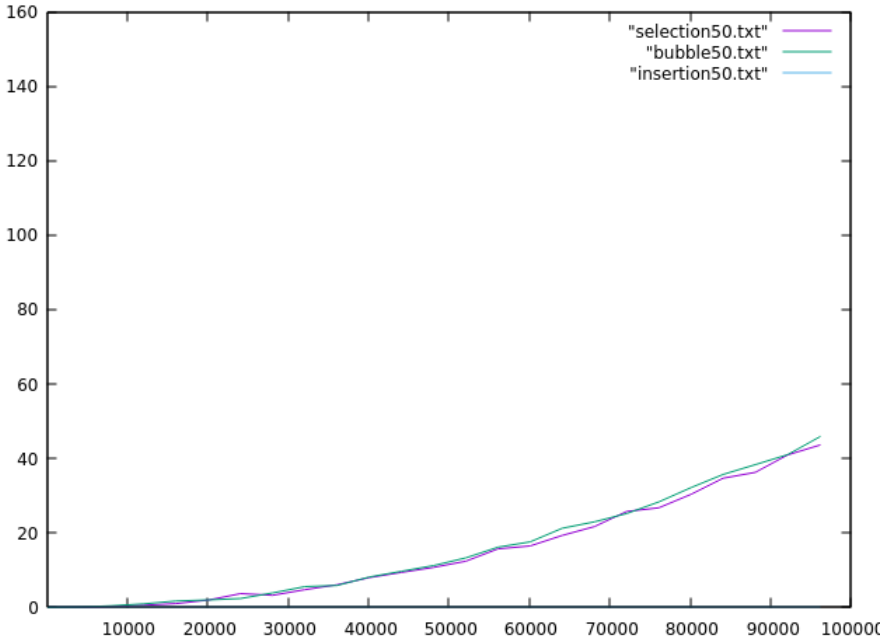
Fonte: Elaborado pelo autor (2023).

3.5 Lista 50% ordenada

3.5.1 Quadráticas

Para listas 50% ordenadas as funções quadraticas tiveram desempenho semelhante, tendo um crescimento continuo.

Figura 9: Quadraticas

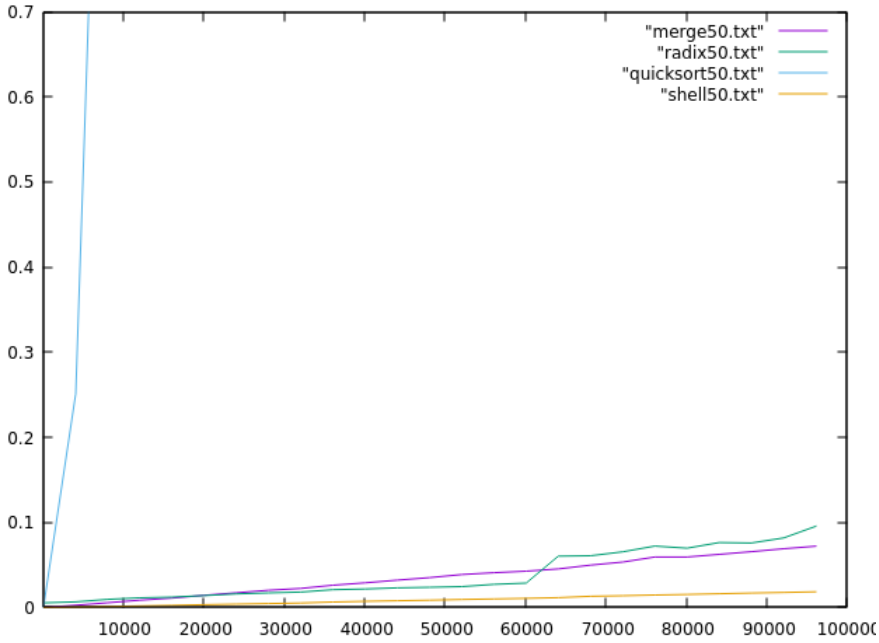


Fonte: Elaborado pelo autor (2023).

3.5.2 Radix e logarítmicos

Para listas 50% ordenadas as funções logaritimicas variaram bastante, com um pico muito alto desde o inicio da função quick sort. Com a shell se demonstrando mais rapida, e as outras duas tendo um desempenho similar.

Figura 10: logarıtmicas



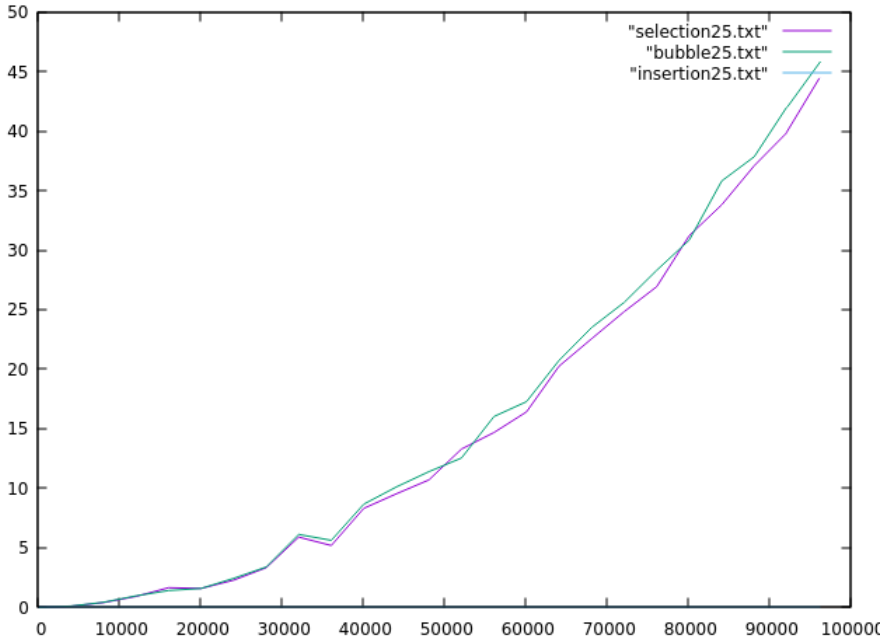
Fonte: Elaborado pelo autor (2023).

3.6 Lista 25% ordenada

3.6.1 Quadráticas

Para listas 25% ordenadas as funções quadraticas tiveram desempenho semelhante, tendo um crescimento continuo.

Figura 11: Quadraticas

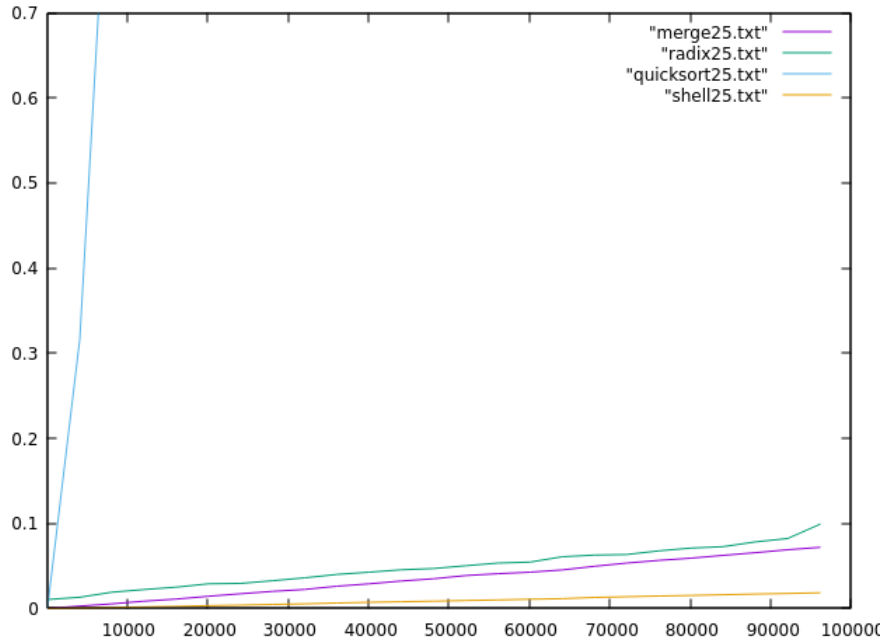


Fonte: Elaborado pelo autor (2023).

3.6.2 Radix e logarítmicos

Para listas 25% ordenadas as funções logaritimicas variaram bastante, com um pico muito alto desde o inicio da função quick sort. Com a shell se demonstrando mais rapida, e as outras duas tendo um desempenho similar.

Figura 12: logarıtmicas



Fonte: Elaborado pelo autor (2023).

4 Discussão dos Resultados

4.1 Observações gerais

O estudo desses algoritmos permite explorar diferentes técnicas de ordenação, podendo assim entender conceitos de complexidade e eficiencia na prática, e comparando-os entendende-se como a quantidade de elementos afeta o desempenho de cada algoritmo.Isso premite reconhecer casos de uso adequados levando em consideração o tamanho da entrada, o estado de ordenação inicial e outros fatores.

4.2 Algoritmos para cada caso

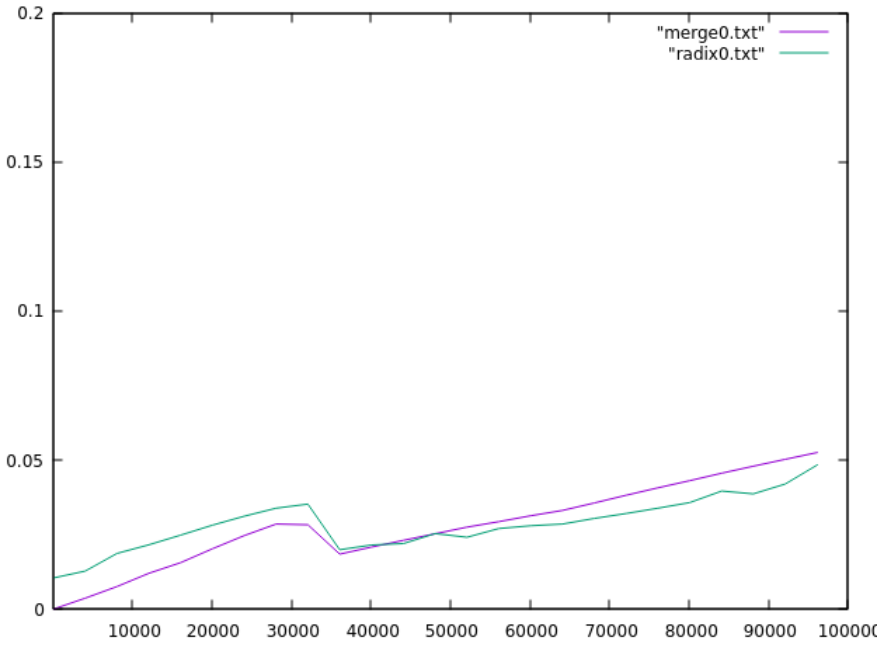
Com base naos dados coletados, chegamos a conclusão de que lidando conjuntos pequenos ou quase ordenados seria mais viavel usar o Insertion Sort e Selection Sort, enqaunto para conjuntos pequenos ou com pouca desordem, o Bubble Sort, e por fim, para conjuntos de dados maiores: Shell Sort, Quick Sort, Merge Sort e o Radix Sort.

4.3 Decomposição de chave(radix) Vs comparação de chaves

Embora a Complexidade de tempo do radix seja inferior a daos outros algoritmos baseados em comparação de chaves, a escolha entre o radix sort e os algoritmos baseados em comparação de chaves depende das características do problema em questão. Se a ordenação envolve uma relação de ordem complexa e variada ou tipos de dados diversos, os algoritmos baseados em comparação de chaves são mais adequados. Por outro lado, se a ordenação é baseada em dígitos individuais ou caracteres e o tamanho da chave é pequeno em relação ao número de elementos, o radix sort pode ser uma opção mais eficiente.

4.4 Quick Sort Vs Merge Sort

O Quick Sort, na prática, geralmente é mais rápido que o Merge Sort. Isso se deve à eficiente manipulação de elementos em um array, uso eficiente da memória cache e menor número de comparações e trocas. No entanto, o desempenho pode variar dependendo dos dados de entrada. O Merge Sort pode ser mais rápido em casos específicos, como quando o array está quase ordenado ou possui muitos elementos repetidos. A escolha do algoritmo mais adequado depende das características dos dados a serem ordenados.



4.5 Acontecimentos inesperados dos gráficos

Na criação dos graficos obtivemos um pico inexperado com a função da Quick sort para as amostras que já estavam parcialmente ordenadas.

4.6 Estimativa de tempo

Para a estimativa de tempo de (10^{12}) elementos na amostra, tendo como base os resultados anteriores, podemos calcular a taxa de crescimento do tempo em relação ao número de elementos, tal que:
$taxa = x/(10^5)$
Onde x é o tempo necessário para uma amostra de 10^5 elementos.
Então, podemos estimar o tempo necessário para ordenar 10^{12} elementos utilizando essa taxa de crescimento:

Tempo estimado $= Taxa * (10^{12}) = (x/(10^5)) * (10^{12}) = (x * 10^7)$ segundos
Sendo assim, para os algoritimos temos que:

- Bubble: $1,9 * 10^{16}$ segundos
- Selection: $4,4782 * 10^{15}$ segundos
- Insertion: $2,137 * 10^{15}$segundos
- Merge: 12231271 segundos
- Shell: 30887624 segundos

- Radix: 11248741 segundos
- Quicksort: 26178790 segundos