UNIVERSIDADE FEDERAL DE ALFENAS

MAXIMIANO CID NETO
TIAGO COSTA SOARES

Comparação entre Métodos de Ordenação

ALFENAS
2023

MAXIMIANO CID NETO
TIAGO COSTA SOARES

Comparação entre Métodos de Ordenação

Trabalho apresentado à disciplina de AEDS 1, ministrada pelo professor Dr. Paulo Alexandre Bressan, no curso de Ciência da Computação na Universidade Federal de Alfenas, Campus Santa Clara.

ALFENAS
2023

Resumo

Esse relatório tem como objetivo compreender as diferenças entre os métodos de ordenação não recursivos, conhecer uma forma de comparação de algoritmos e relatar os experimentos realizados, a fim de fixar o aprendizado do conteúdo ministrado durante a disciplina.

Palavras-chave: Métodos de ordenação.

SUMÁRIO

1. INTRODUÇÃO

Para o desenvolvimento deste trabalho foram utilizados o ambiente de programação NetBeans, sendo o código para o programa escrito na linguagem C++. O objetivo principal é comparar o desempenho dos diferentes métodos de ordenação não-recursivos.

Para isso foram fixados os seguintes parâmetros: Cada método de ordenação deve receber diferentes vetores, com números não repetidos e ordenados de três formas: crescente, aleatória e decrescente.

3 Métodos Implementados

3.1 Métodos de Ordenação Não Recursivos

Métodos de ordenação não recursivos são algoritmos de ordenação que não utilizam chamadas recursivas para realizar o processo de ordenação. Em vez disso, eles empregam estruturas de controles iterativas, como loops, para percorrer a lista e realizar as operações necessárias para obter uma lista ordenada.

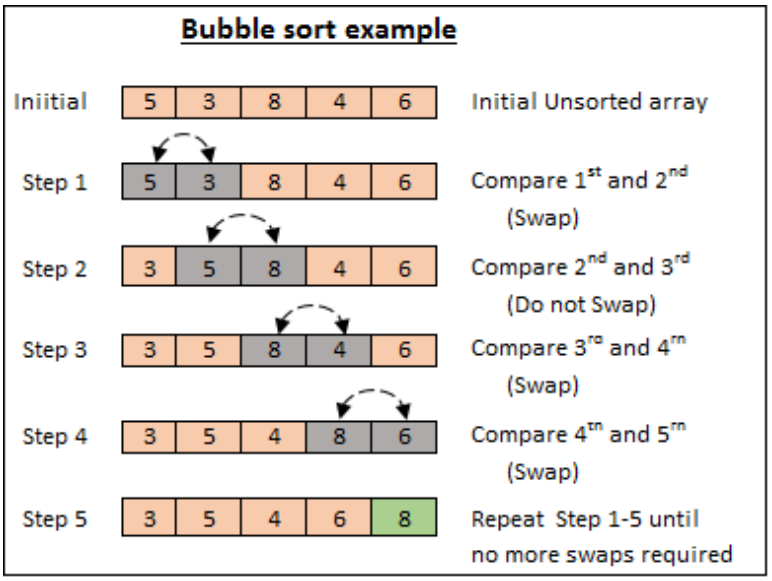
São métodos geralmente implementados utilizando laços de repetição, como loops “for” ou “while”, para percorrer a lista várias vezes e executar as operações de comparação e troca de elementos se necessário. Dessa forma, eles evitam o uso de chamadas recursivas, o que pode ser vantajoso em termos de eficiência de memória e velocidade de execução.

Os métodos de ordenação não recursivos são mais simples de implementar e entender do que os métodos recursivos. No entanto, podem ter uma complexidade de tempo maior em relação a certos cenários específicos ou quando aplicados a listas muito grandes. Os principais métodos não recursivos, e que foram utilizados no desenvolvimento do trabalho são o Bubble Sort, o Insertion Sort e o Selection Sort.

3.1.1 Bubble Sort

O Bubble Sort, também conhecido como ordenação por bolha, é um algoritmo simples de ordenação que percorre repetidamente uma lista, comparando pares de elementos adjacentes e os trocando se estiverem na ordem errada. Esse processo é repetido até que toda a lista esteja ordenada.

Ilustração do funcionamento do Bubble Sort.



A implementação no código se deu a partir da função:

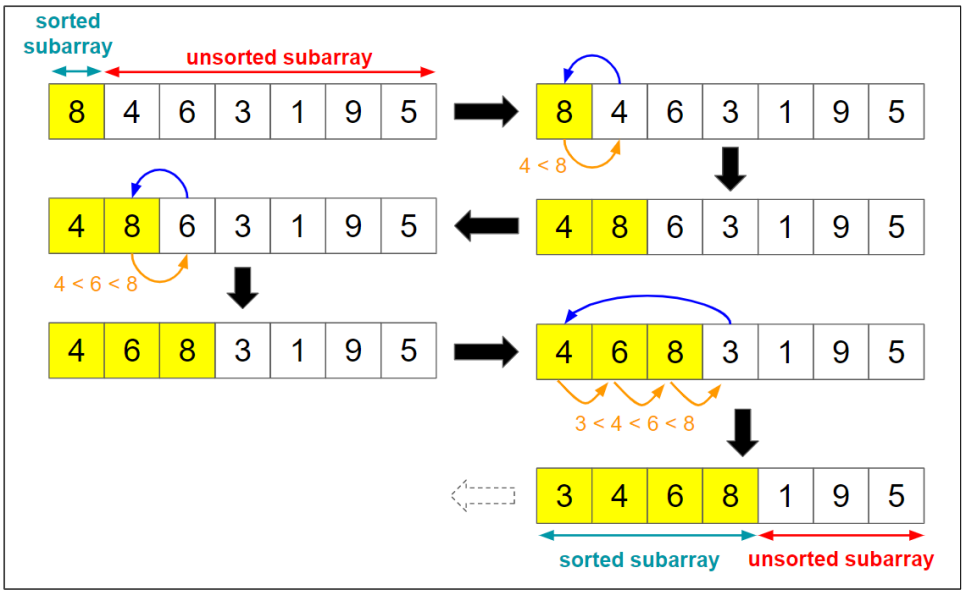
```
//Implementação do Bublle sort
void bubblesort(long long int v[], long long int n, const string& vectorType){
  double time_spent = 0.0;
  int i, j;
  long long int comparisonCount = 0; //Variável que calcula quantas comparações entre posições ocorreram
  long long int swapCount = 0;    //Variável que calcula quantas vezes ocorreram trocas de posição no vetor
  bool swapped;                   //Booleano que verifica se houve troca durante o percorrimento do vetor
  clock_t begin = clock();
  for(i = 0; i < n-1; i++){
    swapped = false;
    for(j = 0; j < n-i-1; j++){
      comparisonCount++;
      if(v[j] > v[j+1]){
        swapCount++;
        swap(v[j], v[j+1]);
        swapped = true;
      }
    }
    if(!swapped) {
        break;
    }
  }
  clock_t end = clock();
  time_spent += (double)(end - begin)/ CLOCKS_PER_SEC;

  printResults("Bubble Sort", vectorType, time_spent, comparisonCount, swapCount);
}
```

No primeiro for a lista é percorrida analisando a partir do primeiro elemento do vetor até o último. Nesse loop a variável bool swapped recebe valor false, caso o segundo loop seja percorrido completamente e ela continue com valor false o código pode ser interrompido, já que nenhuma troca foi realizada.

3.1.2 Insertion Sort

Insertion Sort, ou *ordenação por inserção*, é um algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Ilustração do funcionamento do Insertion Sort



A implementação do código ficou:

```
void insertionsort(long long int v[],long long int n, const std::string& vectorType){
  double time_spent = 0.0;
  int i, j, posicao;
  long long int comparisonCount = 0;
  long long int swapCount = 0;
  clock_t begin = clock();

  for (i = 1; i < n; i++) {
    posicao = v[i];
    j = i - 1;
    while (j >= 0 && v[j] > posicao){
      swapCount++;
      comparisonCount++;
      v[j+1] = v[j];
      j = j-1;
    }
    comparisonCount++;
    v[j+1] = posicao;
  }

  clock_t end = clock();
  time_spent += (double)(end - begin)/ CLOCKS_PER_SEC;

  printResults("Insertion Sort", vectorType, time_spent, comparisonCount, swapCount);
}
```

O algoritmo percorre a lista a partir do segundo elemento até o último. Para cada elemento, é encontrada sua posição correta na porção já ordenada da lista, e o elemento é inserido nessa posição. Ao final do processo, a lista estará ordenada.

3.1.3 Selection Sort

O algoritmo seleciona repetidamente o menor (ou maior) elemento da parte não classificada da lista e o troca pelo primeiro elemento da parte não classificada. Esse processo é repetido para a parte restante não classificada até que toda a lista seja classificada.

A implementação do código ficou:

```
void selectionsort(long long int v[],long long int n, const std::string& vectorType){
  double time_spent = 0.0;
  int i, j, menor, aux, valor;
  valor = n;
  long long int comparisonCount = 0;
  long long int swapCount = 0;
  clock_t begin = clock();

  for (i = 0; i < n-1; i ++) {
    menor = i;
    for ( j= i+1; j < n; j++) {
      comparisonCount++;
      if (v[j] < v[menor]){
        menor = j;
      }
    }
    if (menor != i){
        swapCount++;
        aux = v[i];
        v[i] = v[menor];
        v[menor] = aux;
    }
  }

  clock_t end = clock();
  time_spent += (double)(end - begin)/ CLOCKS_PER_SEC;

  printResults("Selection Sort", vectorType, time_spent, comparisonCount, swapCount);
}
```

O algoritmo começa definindo variáveis importantes, como um contador de comparações e um contador de trocas, juntamente com uma variável para medir o tempo de execução.

Em seguida, é iniciado um loop externo que percorre a lista a partir do primeiro elemento até o penúltimo elemento. Esse loop permite analisar todos os elementos da lista, um por um, para realizar as operações necessárias de comparação e troca.

Dentro do loop externo, uma variável chamada "menor" é definida com o valor do índice atual.Em seguida, é iniciado um loop externo que percorre a lista a partir do primeiro elemento até o penúltimo elemento.

Dentro desse loop externo, é definida uma variável chamada "menor" com o valor do índice atual. Em seguida, é iniciado um segundo loop interno que percorre a lista a partir do próximo elemento até o último elemento.

Durante o loop interno, o algoritmo compara o valor do elemento atual (v[j]) com o valor do elemento de menor índice (v[menor]). Essa comparação permite identificar se há algum elemento menor na parte restante da lista.

O loop externo continua até que todos os elementos sejam percorridos.
Após a conclusão do loop, o tempo de execução é calculado usando a função clock().

3.2 Métodos de captura

Para a implementação do código e posterior análise dos resultados, foi optado pela implementação de três métodos de captura para realizar as comparações de desempenho. Os métodos de captura foram através de tempo, através da quantidade de trocas realizadas na função e na quantidade de comparações realizadas nas funções.

3.2.1 Captura por tempo

A captura por tempo se refere a quanto tempo cada função referente ao método de ordenação foi executada, utilizando a biblioteca ctime para executar as seguintes linhas de código:

```c
double time_spent = 0.0;
int i, j, posicao;
long long int comparisonCount = 0;  //Variável que calcula quantas comparaçõe
long long int swapCount = 0;    //Variável que calcula quantas vezes ocorrera
clock_t begin = clock();

for (i = 1; i < n; i++) {
  posicao = v[i];
  j = i - 1;
  while (j >= 0 && v[j] > posicao){
    swapCount++;
    comparisonCount++;
    v[j+1] = v[j];
    j = j-1;
  }
  comparisonCount++;
  v[j+1] = posicao;
}

clock_t end = clock();
time_spent += (double)(end - begin)/ CLOCKS_PER_SEC;
```

Na função temos a inicialização da variável double time_spent que ao fim da execução receberá o valor do tempo decorrido, já begin = clock() é uma função da biblioteca ctime que inicia a contagem de um relógio. A contagem do relógio é encerrada em clock_t end = clock(), no entanto, essa contagem não é feita em segundos, mas sim em “tiques” de relógio, para obter os segundos foi feita uma divisão do total de tiques de relógio decorrido (dado por end - begin) por CLOCKS_PER_SEC, retornando quantos segundos totais foram gastos.

3.2.2 **Captura por comparações e por troca**

Captura por comparações: Refere-se ao número de comparações feitas entre elementos durante a execução do algoritmo. Quanto menor o número de comparações, mais eficiente é o algoritmo, pois indica que ele está realizando menos operações de comparação.

Captura por troca: Refere-se ao número de trocas de elementos que ocorrem durante a execução do algoritmo. Menor número de trocas indica um algoritmo mais eficiente, pois significa que ele está minimizando as operações de movimentação de elementos na lista.

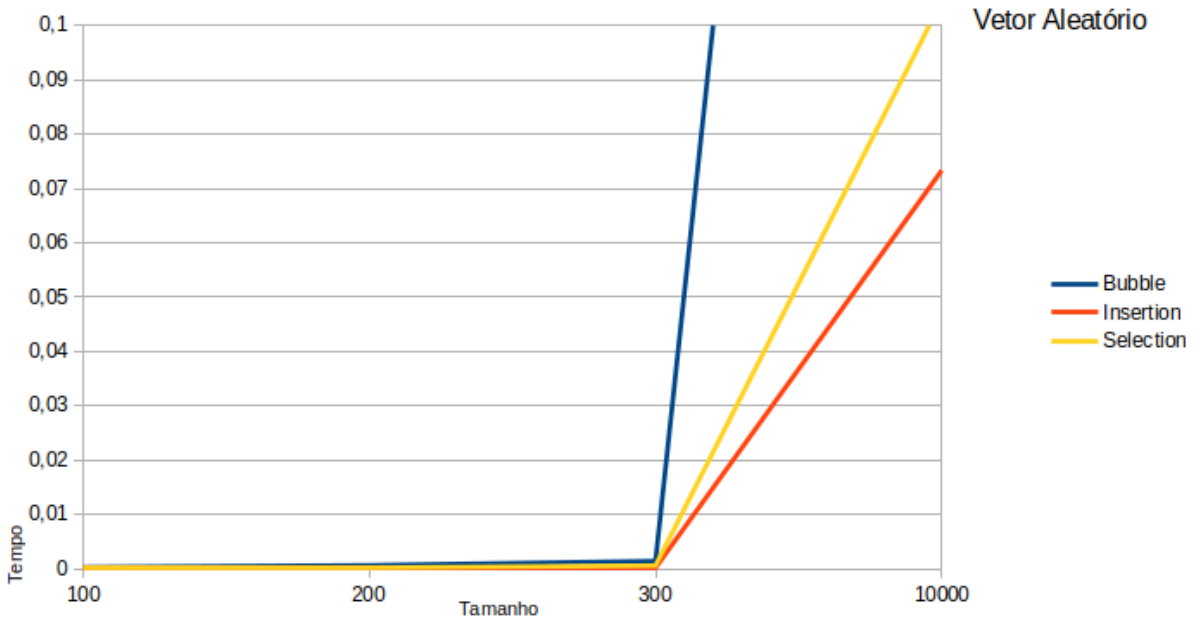
Essas medidas são essenciais para analisar o desempenho dos algoritmos de ordenação. Algoritmos com menos comparações e trocas tendem a ser mais rápidos e eficientes, especialmente em conjuntos de dados maiores.

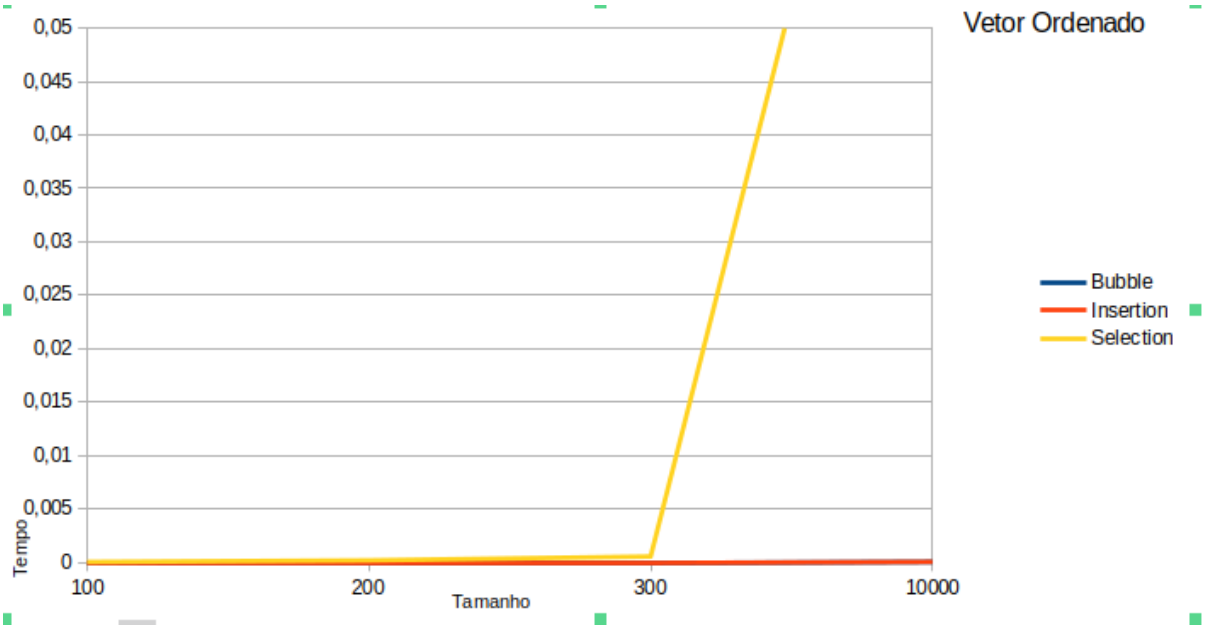
4 Resultados Obtidos

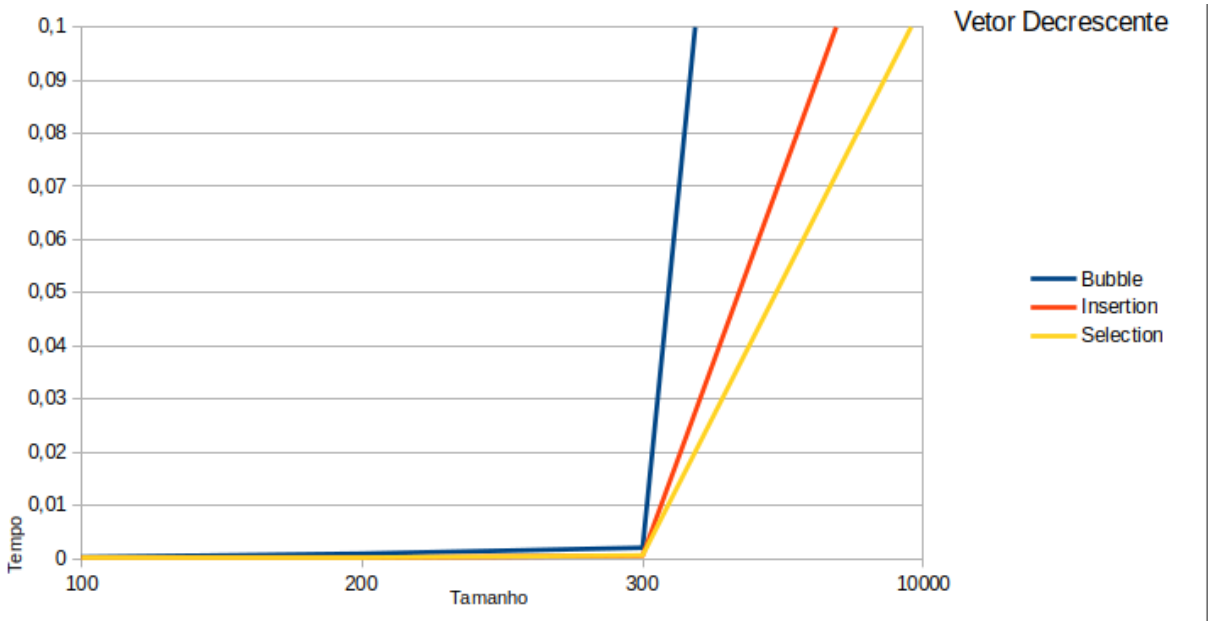
O Insertion Sort demonstrou um desempenho mais eficiente quando aplicado a listas que já estavam parcialmente ordenadas. Isso significa que o algoritmo teve um melhor tempo de execução e realizou menos comparações e trocas de elementos.

O Selection Sort apresentou melhor desempenho para listas em ordem decrescente. Nesse caso, o algoritmo foi capaz de ordenar a lista com menos comparações e trocas em comparação com os outros métodos.

O Bubble Sort, em contrapartida, mostrou-se menos eficiente em todos os casos testados. O algoritmo teve um tempo de execução mais lento e realizou um maior número de comparações e trocas de elementos.







5 Conclusão

Podemos concluir que a escolha do algoritmo de ordenação depende do estado inicial dos dados e das características específicas do problema. O Insertion Sort é uma boa opção quando a lista já está parcialmente ordenada, pois tem um desempenho mais eficiente nesse caso. Já o Selection Sort é mais adequado para listas em ordem decrescente, pois consegue lidar melhor com essa situação.

Em resumo, comparar os métodos de ordenação nos permite compreender as diferenças de desempenho entre eles e identificar os cenários em que cada um se destaca. Essa compreensão é fundamental para selecionar o algoritmo de ordenação mais adequado em diversas situações de programação e processamento de dados.

Ao considerar o estado inicial dos dados e as necessidades do problema, podemos otimizar a escolha do algoritmo, garantindo um processo de ordenação mais eficiente.

6 Referencial teórico

https://www.geeksforgeeks.org/insertion-sort/
https://brilliant.org/wiki/bubble-sort/
https://www.geeksforgeeks.org/selection-sort/
https://www.edureka.co/blog/sorting-algorithms-in-c/