Relatório de Estrutura de Dados

Eduardo Victor Nóbrega Fernandes

Maio, 2018

Algoritmos de busca e ordenação

Este relatório busca demonstrar a utilização dos algoritmos de busca e ordenação e suas respectivas complexidades. Para atingir tal objetivo utilizamos a linguagem C para implementação e testes dos mesmos.

Algoritmos de busca

Busca sequencial ou linear

A busca sequencial é a técnica mais simples de realizar uma busca em uma lista de dados desordenados. Ela visa procurar o valor através de comparações sucessivas a partir do primeiro elemento (ou último) até que se encontre o valor desejado ou até que os elementos da lista se esgotem. O algoritmo de busca sequencial não necessita que a lista esteja ordenado para funcionar.

```
int buscaLinear(int v[], int n, int x){
    int i;
    for(i = 0; i < n; i++){
        if(v[i] == x){
            return i;
        }
    }
    return -1;
}
```

Figure 1: Algoritmo de busca sequencial

Análise de complexidade

Melhor caso

O melhor caso ocorre quando o elemento a ser buscado está na primeira posição do vetor, sendo assim, fará apenas uma comparação, logo, o tempo de execução desse caso é constante.

$$T_b\,(n) = C1 + C2 + C3 + C4$$

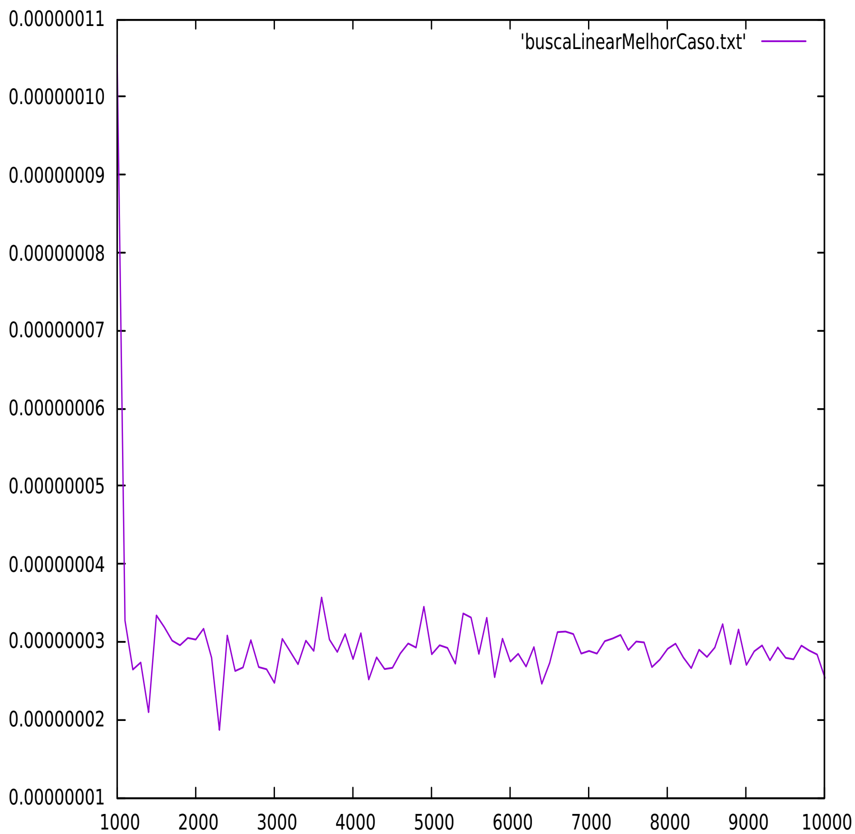

Figure 2: Comportamento do algoritmo de busca linear no melhor caso

Pior caso

O pior caso ocorre quando o elemento a ser buscado não está no vetor, sendo assim, terá que fazer todas as comparações possíveis, logo, será o que custará mais tempo. A complexidade deste algoritmo é de ordem $O\,(n)$.

$$T_w\,(n) = C1 + (n+1)C2 + nC3 + C5$$

$$T_w(n) = (C2 + C3)\,n + C1 + C2 + C5$$

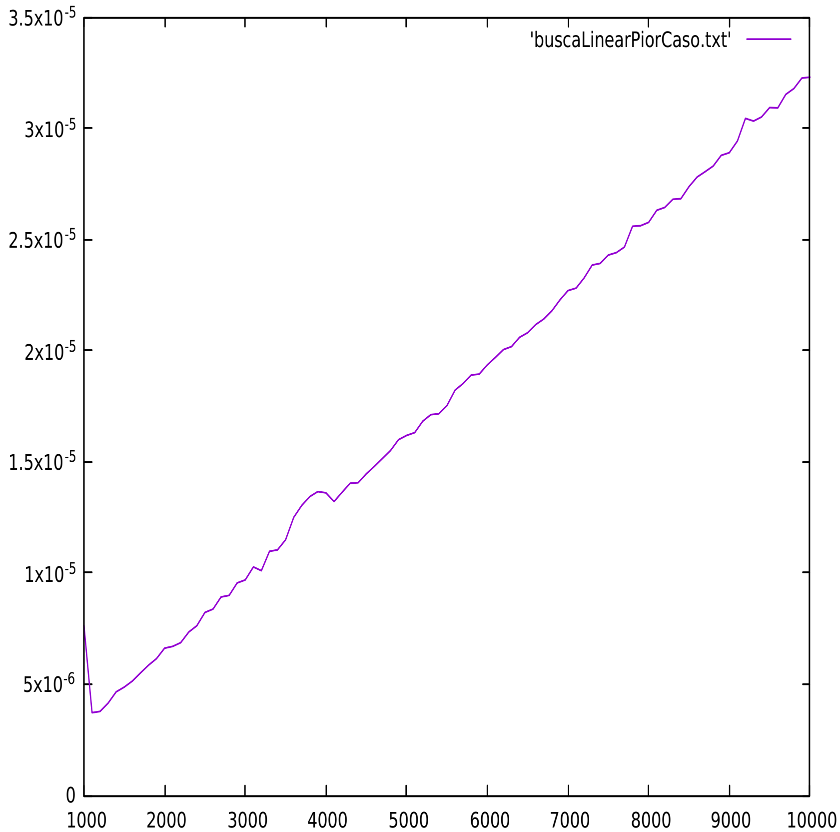

Figure 3: Comportamento do algoritmo de busca linear no pior caso

Caso médio

O caso médio ocorre quando o número a ser buscado é aleatório, podendo se dar no melhor ou pior caso. A complexidade deste algoritmo é de ordem $O(n)$.

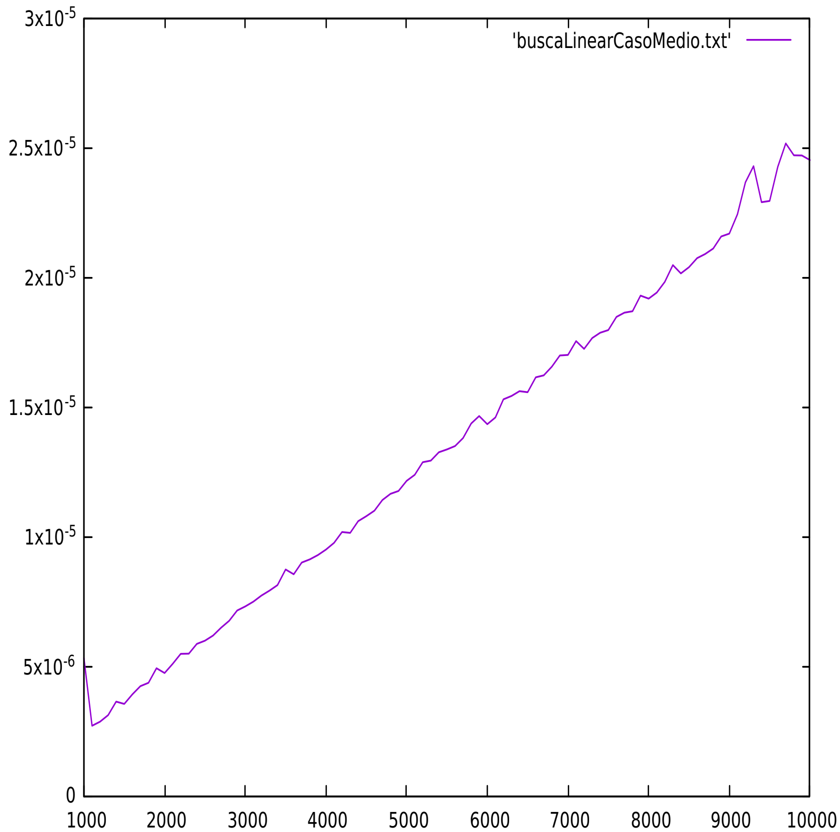

Figure 4: Comportamento do algoritmo de busca linear no caso médio

Comparativo dos casos

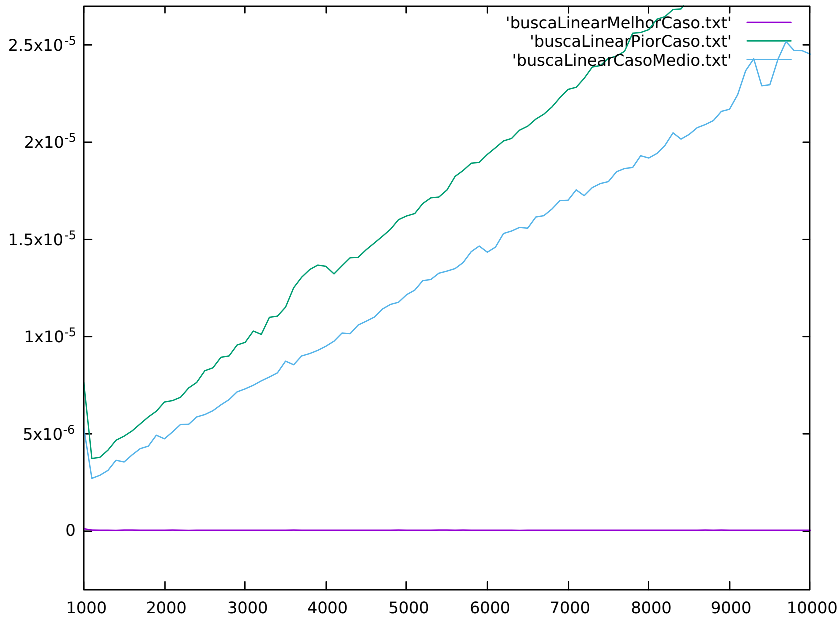

Figure 5: Comparativo dos casos de busca linear

Busca binária

A busca binária é um eficiente algoritmo para encontrar um item em um vetor ordenado de itens. Ela funciona dividindo repetidamente pela metade a porção do vetor que deve conter o item, até reduzir as localizações possíveis a apenas uma.

```
int buscaBinaria(int v[], int x, int s, int e){
     int m;
     if(s <= e){
         m = (s + e)/2;
         if(v[m] == x)
             return m;
         if(v[m] > x){
             return buscaBinaria(v, x, s, m-1);
         }
         return buscaBinaria(v, x, m+1, e);
     }
     return -1;
 }
```

Figure 6: Algoritmo de busca binária

Análise de complexidade

Melhor caso

O melhor caso da busca binária ocorre quando o elemento a ser buscado é sempre o elemento do meio do vetor, o que torna o tempo de execução constante.

$$T_b(n) = C1 + C2 + C3 + C4 + C5$$

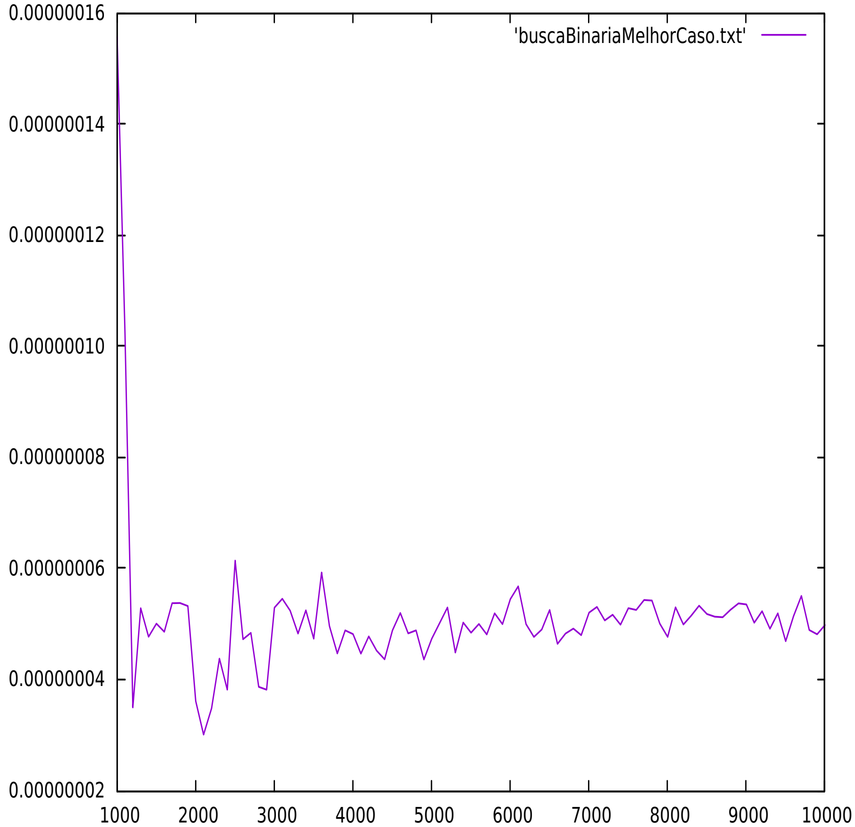

Figure 7: Comportamento do algoritmo de busca binária no melhor caso

Pior caso

O pior caso da busca binária ocorre quando o elemento buscado não está no vetor, e consequentemente o algoritmo terá que fazer o número máximo de processos. O tempo desse caso acaba sendo de ordem $O(\log n)$.

Considere $C1 + C2 + C3 + C4 + C6 + C_{7,8}$.

$$T_w(0) = C1 + C8$$

$$T_w(n) = a + T_w\left(\frac{n-1}{2}\right)$$

$$T_w\left(\frac{n-1}{2}\right) = a + T_w\left(\frac{n-3}{4}\right)$$

$$T_w(n) = a + \left[a + T_w\left(\frac{n-3}{4}\right)\right]$$

$$T_w(n) = 2a + T_w\left(\frac{n-3}{4}\right)$$

$$T_w\left(\frac{n-3}{4}\right) = a + T_w\left(\frac{n-7}{8}\right)$$

$$T_w(n) = 2a + \left[a + T_w\left(\frac{n-7}{8}\right)\right]$$

$$T_w(n) = 3a + T_w\left(\frac{n-7}{8}\right)$$

a partir disso identificamos o seguinte padrão

$$xa + T_w\left(\frac{n-(2^x-1)}{2^x}\right)$$

igualhando a equação da função T_w pelo valor que se encontra o caso base, temos que

$$\frac{n-(2^x-1)}{2^x} = 0$$

$$n - 2^x + 1 = 0$$

$$n + 1 = 2^x$$

$$\log_2^{2^x} = \log_2^{(n+1)}$$

$$x = \log_2^{(n+1)}$$

substituindo na equação original, encontra-se

$$T_w(n) = \left(\log_2^{(n+1)}\right) a + T_w\left(\frac{n-2^{\log_2^{(n+1)}}-1}{2^{\log_2^{(n+1)}}}\right)$$

$$T_w(n) = \left(\log_2^{(n+1)}\right) a + T_w(0)$$

$$T_w(n) = \log_2^{(n+1)} + C1 + C8$$

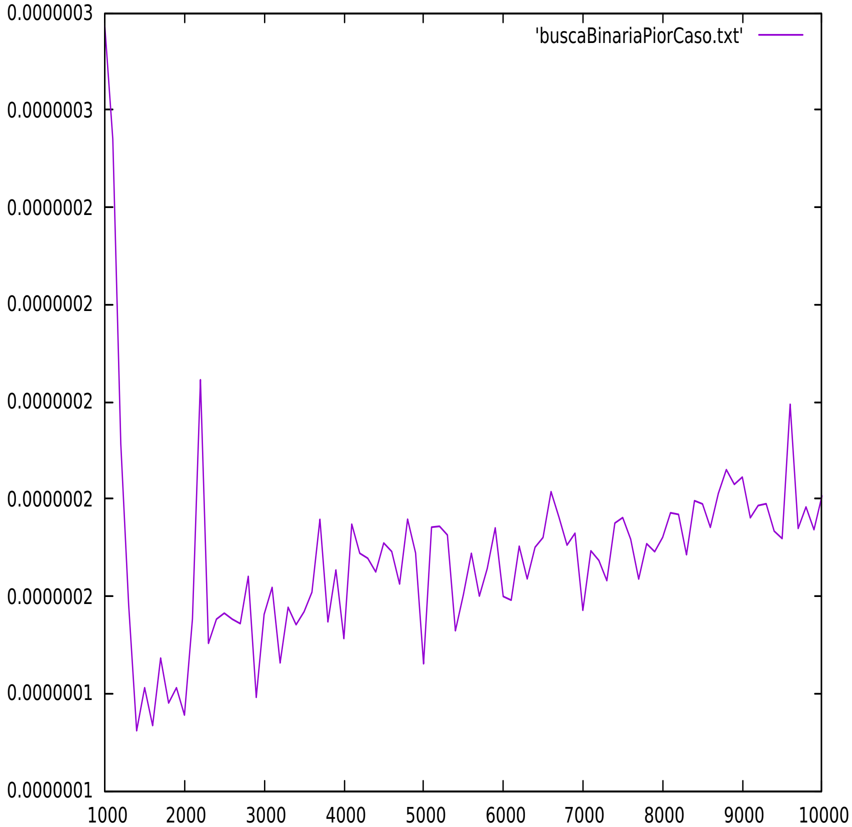

Figure 8: Comportamento do algoritmo de busca binária no pior caso

Caso médio

O caso médio da busca binária ocorre quando os elementos do vetor são gerados de forma aleatória e o elemento buscado pode ou não estar no vetor. O tempo execução desse caso é de ordem $O(\log n)$.

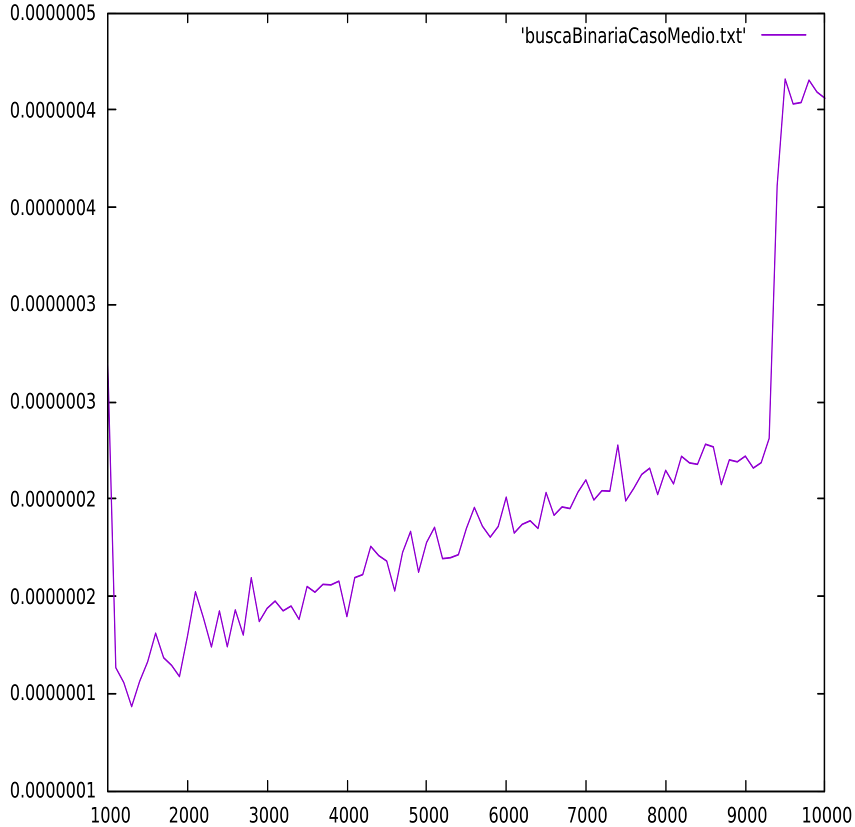

Figure 9: Comportamento do algoritmo de busca binária no caso médio

Comparativos finais

Binário vs sequencial (médio)

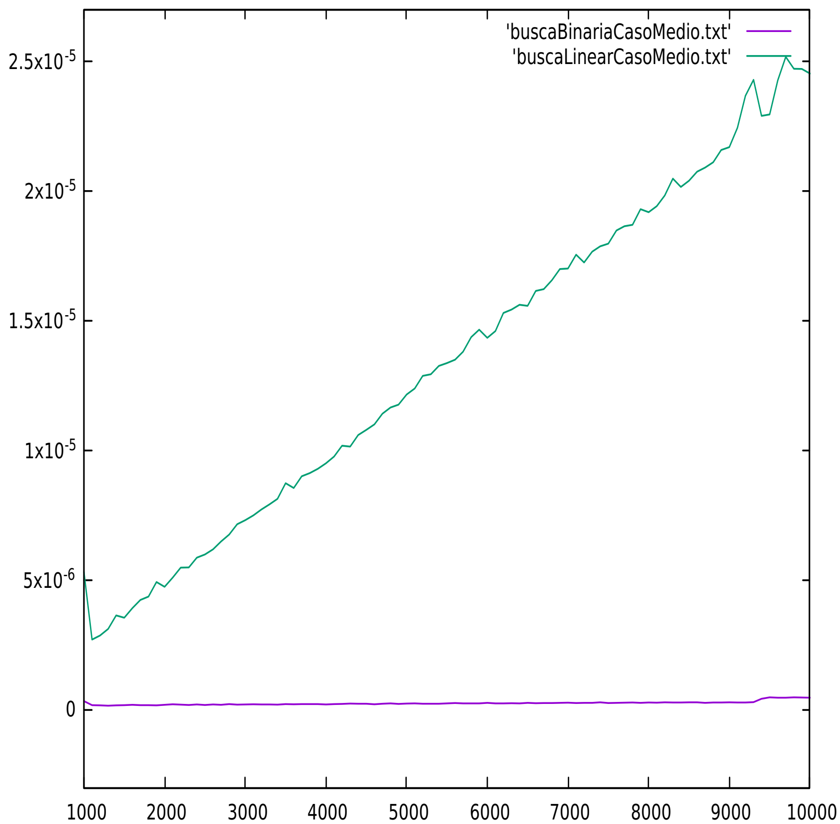

Figure 10: Comparativo dos algoritmos de busca nos casos médios

Algoritmos de ordenação

Bubble sort

O bubble sort é um algoritmo de ordenação simples. O processo ocorre por "flutuação", visto que o mesmo ordena de par em par, verificando se o primeiro termo do par é maior que o segundo, caso seja, troca-se os valores. Esse processo se repete até o final do vetor, por várias rodadas.

```
void bbsort(int v[], int n){
    int i, j, aux;
    for(i = 1; i < n; i++){
        for(j = 0; j < n - i; j++){
            if(v[j] > v[j + 1]){
                aux = v[j];
                v[j] = v[j + 1];
                v[j + 1] = aux;
            }
        }
    }
}
```

Figure 11: Algoritmo do bubble sort

Análise de complexidade

A complexidade desse algoritmo é de ordem $O\left(n^2\right)$, visto que necessita fazer n^2 operações.

$$T\left(n\right) = C1 + nC2 + C3\sum_{i=2}^{n} i + C4\sum_{i=1}^{n-1} i + \left(C5 + C6 + C7\right)\sum_{i=1}^{n-1} i$$

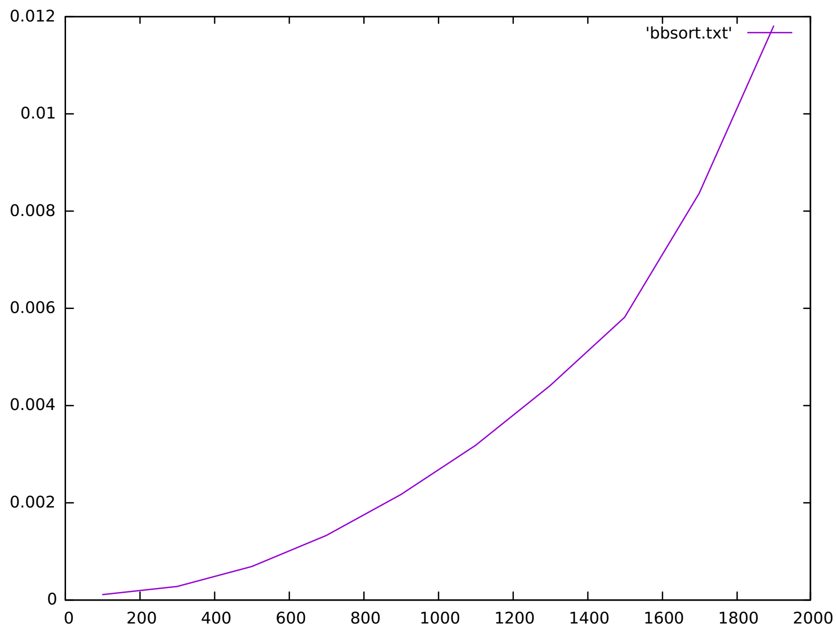

Figure 12: Comportamento do algoritmo bubble sort

Insertion sort

O algoritmo de ordenação por inserção é baseado na ideia de dividir o vetor em dois subvetores, um ordenado e outro desordenado, então, escolhe-se o primeiro elemento da região não ordenada e insere-se o mesmo na posição correta na região ordenada e assim por diante, até todo o vetor está ordenado.

```
void insertionSort(int v[], int n){
    int i = 0, j, aux;
    while(i < n){
        j = i;
        while((j > 0) && (v[j] < v[j-1])){
            aux = v[j];
            v[j] = v[j-1];
            v[j-1] = aux;
            j--;
        }
        i++;
    }
}
```

Figure 13: Algoritmo do insertion sort

Análise de complexidade

Melhor caso

O melhor caso ocorre quando o vetor já está ordenado e os únicos processos que ocorrerão são as verificações, descartando a necessidade de se fazer alguma troca. O tempo de execução desse caso é $O\left(n\right)$.

$$T_b\left(n\right) = C1 + \left(n+1\right)C2 + nC3 + nC4 + nC9$$

$$T_b\left(n\right) = \left(C2 + C3 + C4 + C9\right)n + C1 + C2$$

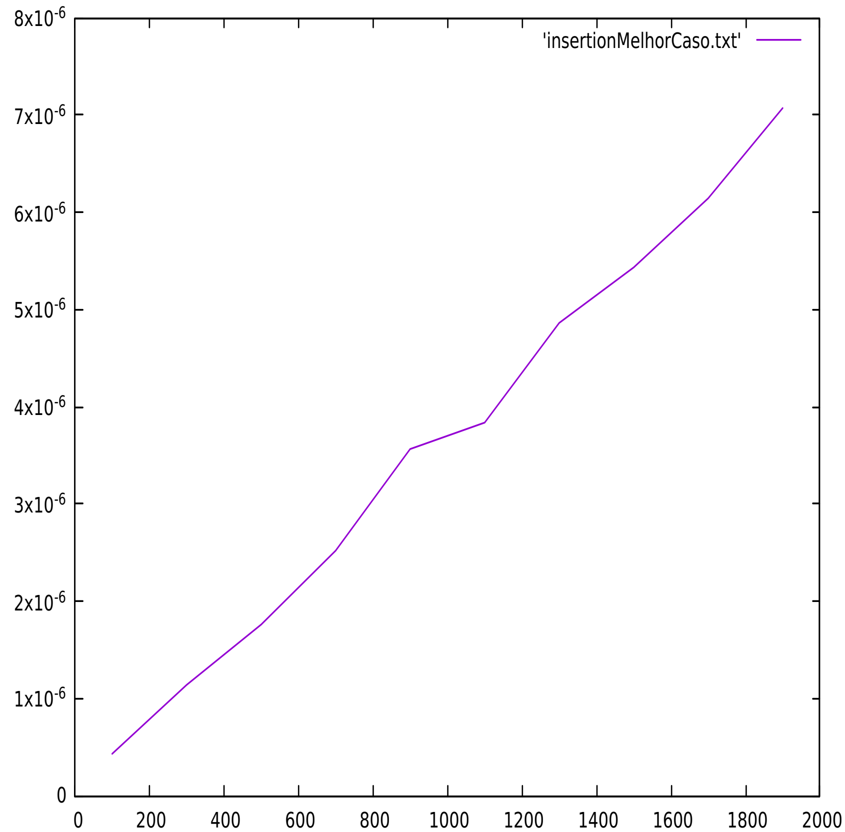

Figure 14: Comportamento do algoritmo insertion sort no melhor caso

Pior Caso

O pior caso ocorre quando o vetor está ordenado de forma decrescente e será feito o número máximo de verificações e trocas possíveis, resultando em um tempo de ordem $O\left(n^2\right)$.

$$T_w\left(n\right) = C1 + \left(n+1\right)C2 + nC3 + C4\sum_{i=0}^{n} i + \left(C5 + C6 + C7 + C8\right)\sum_{i=0}^{(n-1)} i + nC9$$

$$T_w\left(n\right) = C1 + C2 + \left(C2 + C3\right)n + C4\sum_{i=1}^{(n-1)} i + \right)\left(C5 + C6 + C7 + C8\right)\sum_{i=1}^{(n-2)} + nC9$$

$$T_w\left(n\right) = C1 + C2 + \left(C2 + C3\right)n + C4\left(\frac{n}{2}\left(n+1\right) - n\right) + \left(C5 + C6 + C7 + C8\right)\left(\frac{n}{2}\left(n+1\right) - 2n\right) + nC9$$

$$T_w(n) = C1 + C2 + (C2 + C3 + C9)\,n + C4\left(\frac{n(n+1)}{2} - n\right) + (C5 + C6 + C7 + C8)\left(\frac{n(n+1)}{2} - 2n\right)$$

$$T_w(n) = C1 + C2 + (C2 + C3 + C9)\,n + \frac{1}{2}C4n^2 + \frac{1}{2}\left(C5 + C6 + C7 + C8\right)\left(n^2 - n\right)$$

$$T_w(n) = C1 + C2 + (C2 + C3 + C9)\,n + \frac{1}{2}C4n^2 + \frac{1}{2}\left(C5n^2 - C5n + C6n^2 - C6n + C7n^2 - C7n + C8n^2 - C8n\right)$$

$$T_w(n) = C1 + C2 + (C2 + C3 + C9)\,n + \frac{1}{2}C4n^2 + \frac{1}{2}C5n^2 - \frac{1}{2}C5n + \frac{1}{2}C6n^2 - \frac{1}{2}C6n + \frac{1}{2}C7n^2 - \frac{1}{2}C7n + \frac{1}{2}C8n^2 - \frac{1}{2}C8$$

$$T_w(n) = \left[\frac{1}{2}C4 + \frac{1}{2}C5 + \frac{1}{2}C6 + \frac{1}{2}C7 + \frac{1}{2}C8\right]n^2 + \left[C2 + C3 - \frac{1}{2}C5 - \frac{1}{2}C6 - \frac{1}{2}C7 - \frac{1}{2}C8\right]n + C1 + C2$$

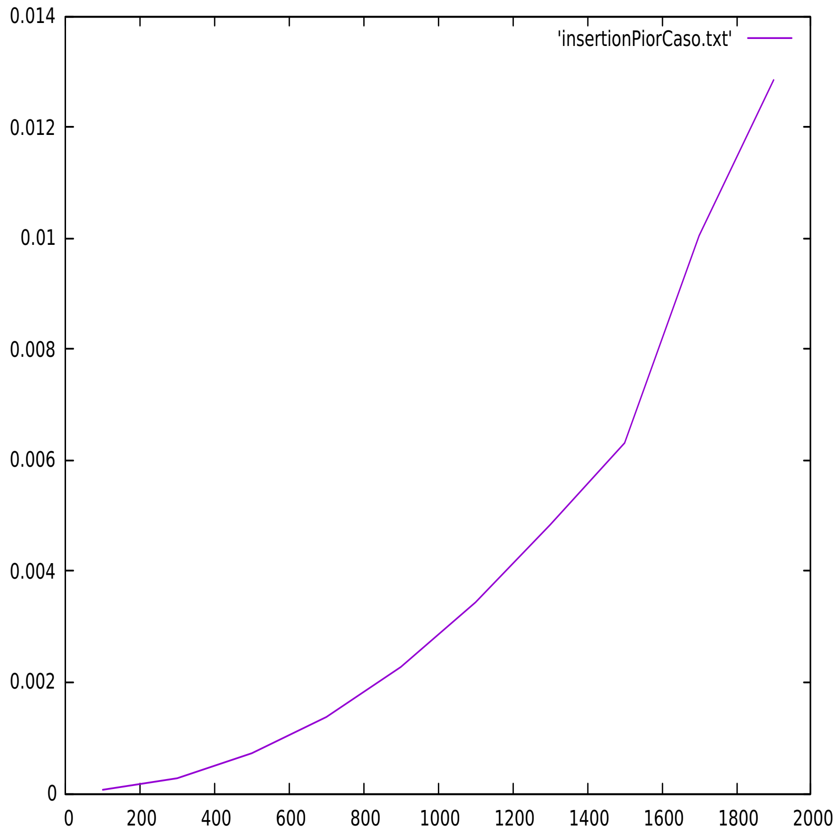

Figure 15: Comportamento do algoritmo insertion sort no pior caso

Caso médio

O caso médio ocorre quando os elementos do vetor são gerados de forma aleatória. A complexidade deste algoritmo é de ordem $O\left(n^2\right)$.

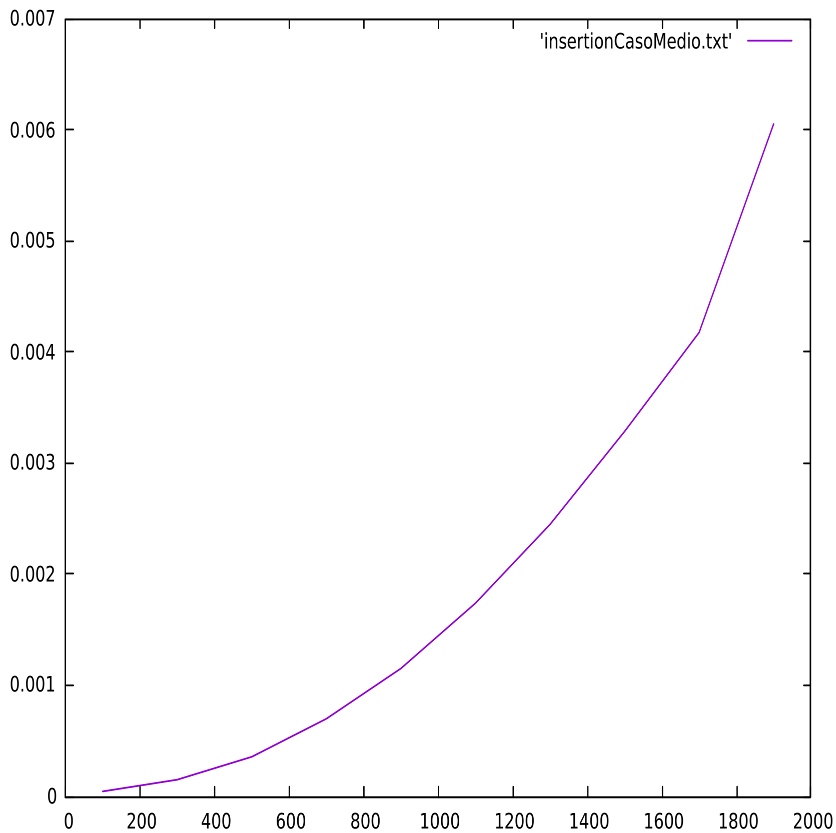

Figure 16: Comportamento do algoritmo insertion sort no caso médio

Comparativo dos casos

O melhor caso é tão rápido se comparado aos outros casos do mesmo que ao colocar os três gráficos em sobreposição o melhor caso que é linear, fica parecendo constante, tamanha que é essa diferença.

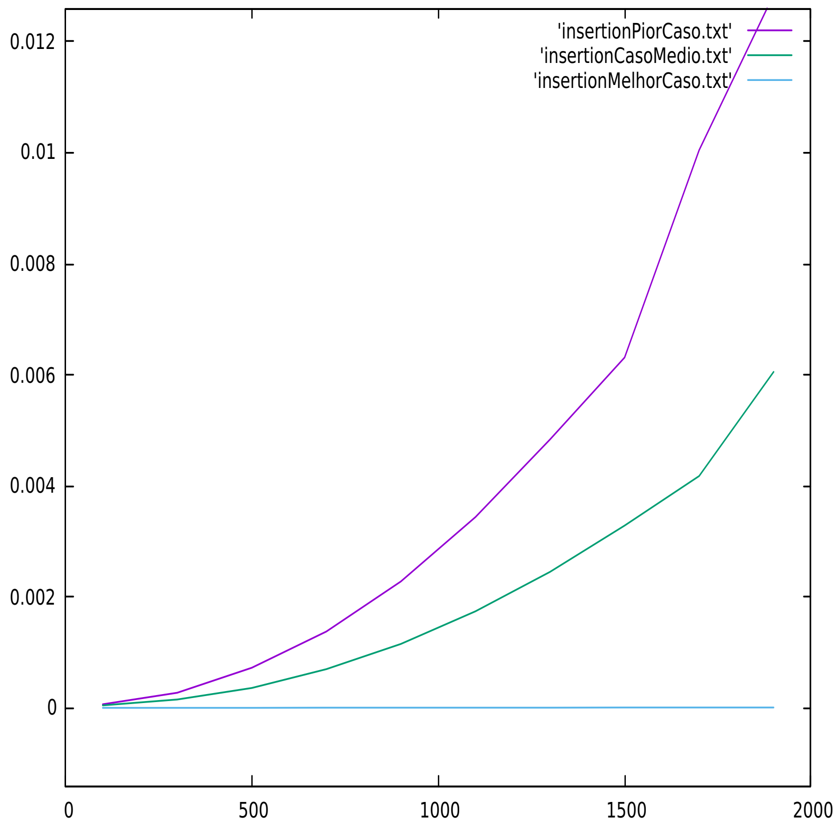

Figure 17: Comportamento do algoritmo insertion sort no caso médio

Distribution sort

A ideia é utilizar recipientes para organizar e classificar os dados e então retorna-los. Este tipo de algoritmo faz uso de um vetor auxiliar, onde é feito a separação e numeração das ocorrências dos dados de entrada, a qual os valores do vetor são usados como índices em um outro vetor. O distribution pode se tornar inviável para máquinas que não contém uma memória boa, pois aloca memória para um vetor do tamanho da diferência entre o maior e o menor termo. Caso tenha um vetor de 2 termos com os mesmos sendo 0 e 10, o algoritmo irá alocar memória da posição 0 até a posição 10.

A implementação de um algorítimo de distribution sort requer varias operações, em geral são usados as seguintes etapas:

1. Inicializar os elementos do vetor auxiliar com zeros.

2. Jogar os valores do vetor de entrada como índice no vetor auxiliar.

3. Ordenar o vetor auxiliar não vazios.
4. Transferir os valores do vetor auxiliar para o vetor de entrada.

```
int *distributionSort(int v[], int n){
    int maior = max(v, n);
    int i, k = maior + 1;
    int *w = zeros(k);
    int *y = zeros(n);
    for(i = 0; i < n; i++){
        w[v[i]]++;
    }
    for(i = 1; i < k; i++){
        w[i] += w[i - 1];
    }
    for(i = 0; i < n; i++){
        y[w[v[i]] - 1] = v[i];
        w[v[i]]--;
    }
    return y;
}
```

Figure 18: Algoritmo do distribution sort

Análise de complexidade

A complexidade desse algoritmo é de ordem $O(n + k)$.

$T(n) = C1 + C2 + C3 + C4 + (n + 1)C5 + nC6 + kC7 + (k - 1)C8 + (n + 1)C9 + nC10 + nC11 + nC12$

$T(n) = (C5 + C6 + C7 + C9 + 10 + C11)n + (C7 + C8)k + C1 + C2 + C3 + C4 + C5 - C8 + C12$

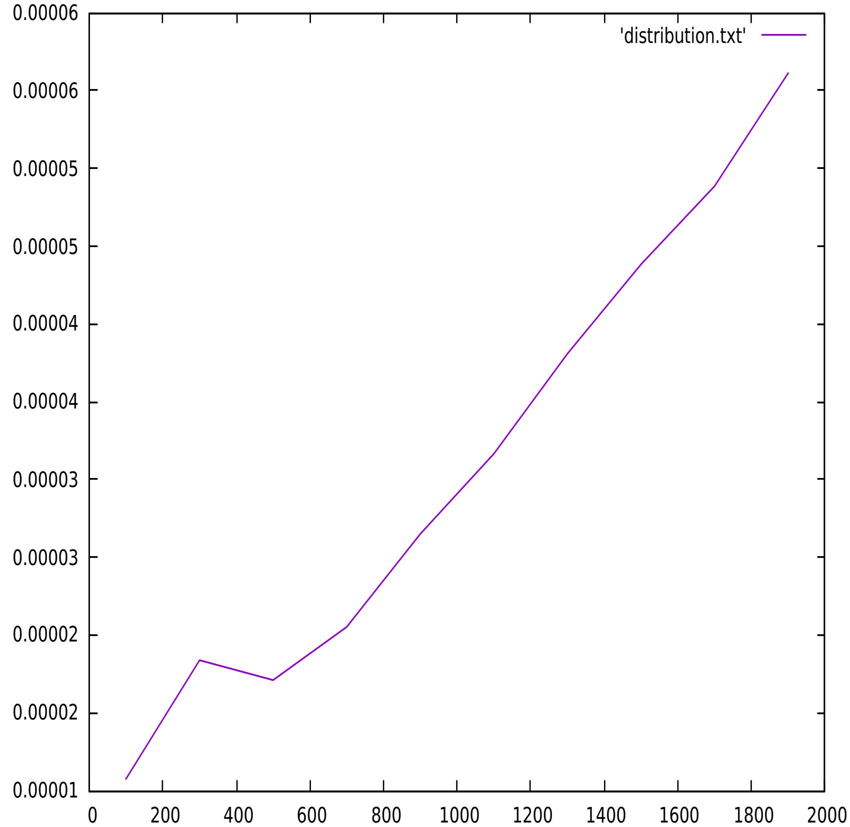

Figure 19: Comportamento do algoritmo distribution sort

Quick sort

O Quick Sort é um dos método mais rápidos de ordenação, apesar de às vezes partições desequilibradas poderem conduzir a uma ordenação lenta. Esse método de ordenação utiliza a técnica divide and conquer (dividir o problema inicial em dois subproblemas e resolver um problema menor utilizando a recursividade).

Este método baseia-se na divisão do vetor em dois subvetores, dependendo de um elemento chamado pivô. O pivô foi utilizado como sendo o último elemento do vetor. Um dos subvetores contém os elementos menores que o pivô enquanto a outra contém os maiores. O pivô é colocado entre ambas, ficando na posição correta. Os dois subvetores são ordenados de forma idêntica, até que se chegue o vetor com um só elemento.

O algoritmo de quick sort é composto pela função do mesmo e por uma função partition, essa função irá rearranjar o vetor recebido como parâmentro de tal modo que os elementos menores que o pivô fiquem a sua esquerda e os maiores a sua direita, ficando o pivô na posição central do vetor.

```
void quickSort(int v[], int s, int e){
    int p;
    if(s < e){
        p = partition(v, s, e);
        quickSort(v, s, p - 1);
        quickSort(v, p + 1, e);
    }
}

int partition(int v[], int s, int e){
    int l = s, i, aux;
    for(i = s; i < e; i++){
        if(v[i] < v[e]){
            aux = v[i];
            v[i] = v[l];
            v[l] = aux;
            l++;
        }
    }
    aux = v[e];
    v[e] = v[l];
    v[l] = aux;
    return l;
}
```

Figure 20: Algoritmo do quick sort

Análise de complexidade

Complexidade do partition para cálculos posteriores.

$$T^p\left(n\right) = C1 + \left(n+1\right)C2 + nC3 + \left(C4 + C5 + C6 + C7\right)\frac{n}{2} + C8 + C9 + C10 + C11$$

$$T^p\left(n\right) = \left(C2 + C3\right)n + \left(C4 + C5 + C6 + C7\right)\frac{n}{2} + C1 + C8 + C9 + C10 + C11$$

Melhor caso

O melhor caso de particionamento acontece quando ele produz duas listas de tamanho não maior que $\frac{n}{2}$, uma vez que uma lista terá tamanho $\frac{n}{2}$ e outra tamanho $\frac{n}{2} - 1$. Nesse caso, o quick sort é executado com maior rapidez. A complexidade desse caso é $O\left(n \log_2 n\right)$.

Considere $a = C1 + 2C2 + C3 + C4 + C5$.

$$T_b\left(0\right) = C1$$

$$T_b\left(1\right) = C1$$

$$T_b\left(n\right) = a + 2T_b\left(\frac{n-1}{2}\right) + T^P\left(n\right)$$

$$T_b\left(\frac{n-1}{2}\right) = a + 2T_b\left(\frac{n-3}{4}\right) + T^P\left(\frac{n-1}{2}\right)$$

$$T_b\left(n\right) = a + 2\left[a + 2T_b\left(\frac{n-3}{4}\right) + T^P\left(\frac{n-1}{2}\right)\right] + T^P\left(n\right)$$

$$T_b\left(n\right) = 3a + 4T_b\left(\frac{n-3}{4}\right) + 2T^P\left(\frac{n-1}{2}\right) + T^P\left(n\right)$$

$$T_b\left(\frac{n-3}{4}\right) = a + 2T_b\left(\frac{n-7}{8}\right) + T^P\left(\frac{n-3}{4}\right)$$

$$T_b\left(n\right) = 3a + 4\left[a + 2T_b\left(\frac{n-7}{8}\right) + T^P\left(\frac{n-3}{4}\right)\right] + 2T^P\left(\frac{n-1}{2}\right) + T^P\left(n\right)$$

$$T_b\left(n\right) = 7a + 8T_b\left(\frac{n-7}{8}\right) + 4T^P\left(\frac{n-3}{4}\right) + 2T^P\left(\frac{n-1}{2}\right) + T^P\left(n\right)$$

analisando tal relação de recorrência, encontramos o seguinte padrão:

$$T_b\left(n\right) = \left(2^x - 1\right)a + 2^x T_b\left(\frac{n - \left(2^x - 1\right)}{2^x}\right) + \sum_{i=0}^{x-1} T^p\left(\frac{n - 2^i - 1}{2^i}\right)$$

igualha-se a equação $\frac{n-(2^x-1)}{2^x}$ a 1, que é o termo deixa de existir recursividade,

$$\frac{n-(2^x-1)}{2^x}=1$$

$$n-(2^x-1)=2^x$$

$$n-2^x+1=2^x$$

$$n+1=2^x+2^x$$

$$2^{x+1}=n+1$$

aplica-se log nos dois lados

$$\log_2 2^{x+1}=\log_2(n+1)$$

$$x+1=\log_2(n+1)$$

$$x=\log_2(n+1)-1$$

substituindo o valor de x, temos

$$T_b(n)=\left(2^{\log_2(n+1)-1}-1\right)+2^{\log_2(n+1)-1}T_b(1)+\sum_{i=0}^{\log_2(n+1)-2}2^iT^p\left(\frac{n-2^i-1}{2^i}\right)$$

$$T_b(n)=(2^n-1)+2^nT_b(1)+\sum_{i=0}^{n-1}2^iT^p\left(\frac{n-2^i-1}{2^i}\right)$$

substituindo o T^p pelo valor calculado anteriormente, temos que

$$T_b(n)=(2^n-1)+2^nT_b(1)+\sum_{i=0}^{n-1}2^i\left((C2+C3)\,n+(C4+C5+C6+C7)\,\frac{n}{2}+C1+C8+C9+C10+C11\right)$$

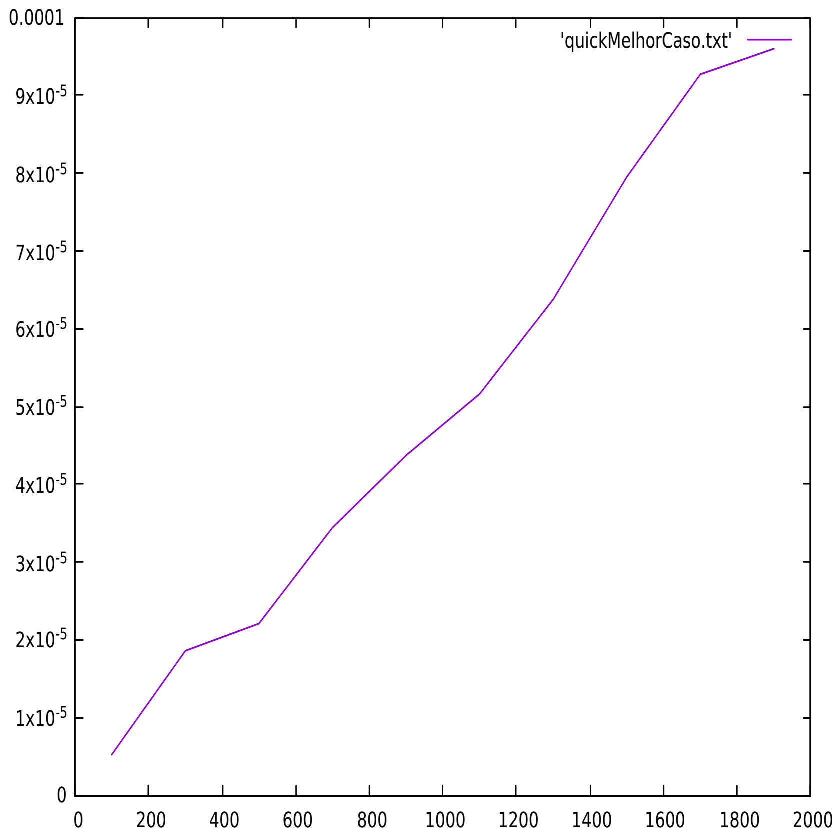

Figure 21: Comportamento do algoritmo quick sort no melhor caso

PiorCaso

O exemplo de melhor caso que foi utilizado para análise foi quando o pivô é o maior elemento ou o menor elemento da lista e a mesma já se encontra ordenada, ou inversamente ordenada. A complexidade desse caso é $O\left(n^2\right)$.

Teremos a seguinte relação de recorrência:

$$T_w\left(n\right) = C1 + 2C2 + C3 + C4 + C5 + T_w\left(n-1\right) + T^p\left(n\right)$$

Calculamos a $T_w\left(n-1\right)$

$$T_w\left(n-1\right) = C1 + 2C2 + C3 + C4 + C5 + T_w\left(n-2\right) + T^p\left(n-1\right)$$

e substituimos de volta na equação original

$$T_w\left(n\right) = C1{+}2C2{+}C3{+}C4{+}C5{+}\left[C1 + 2C2 + C3 + C4 + C5 + T_w\left(n-2\right) + T^p\left(n-1\right)\right]{+}T^p\left(n\right)$$

$T_w(n) = 2C1 + 4C2 + 2C3 + 2C4 + 2C5 + T_w(n-2) + T^p(n-1) + T^p(n)$

Calculamos agora $T_w(n-2)$

$$T_w(n-2) = C1 + 2C2 + C3 + C4 + C5 + T_w(n-3) + T^p(n-2)$$

e substituimos novamente na equaçao original

$T_w(n) = 2C1{+}4C2{+}2C3{+}2C4{+}2C5{+}[C1 + 2C2 + C3 + C4 + C5 + T_w(n-3) + T^p(n-2)]{+}T^p(n-1){+}T^p$

$T_w(n) = 3C1{+}6C2{+}3C3{+}3C4{+}3C5{+}T_w(n-3){+}T^p(n-2){+}T^p(n-1){+}T^p(n)$

ao definirmos $C1 + 2C2 + C3 + C4 + C5$ como sendo a, facilitamos o entendimento e idenficamos o seguinte padrão

$$xa + T_w(n-x) + \sum_{i=0}^{x-1} T^p(n-i)$$

agora temos que igualhar o valor dentro da chamada da função T_w a 1, que é o caso onde para de ocorrer a recursividade, temos que

$$n - x = 1$$

$$-x = 1 - n$$

ao multiplicar tudo por -1, encontramos

$$x = n - 1$$

se substituirmos de volta na equação levando em conta o a, temos que

$$T_w(n) = (n-1)\,a + T_w(1) + \sum_{i=0}^{(n-2)} T^p(n-i)$$

$$T_w(n) = na - a + C1 + \sum_{i=0}^{(n-2)} T^p(n-i)$$

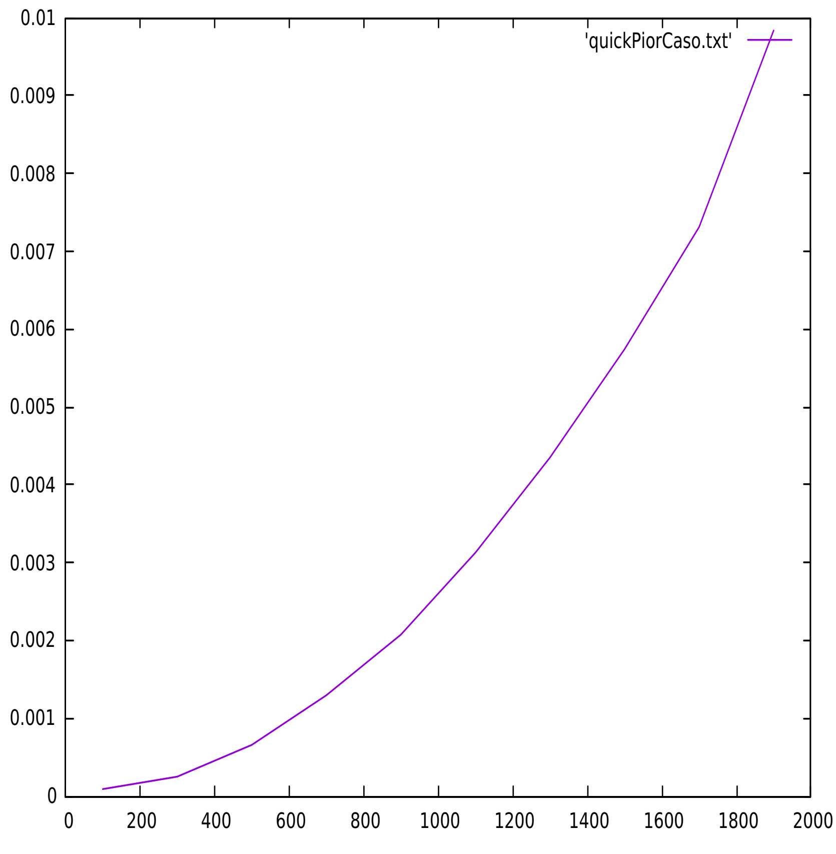

Figure 22: Comportamento do algoritmo quick sort no pior caso

Caso médio

O caso médio ocorre quando os elementos do vetor estão de forma aleatória. A complexidade desse caso é $O(n \log_2 n)$.

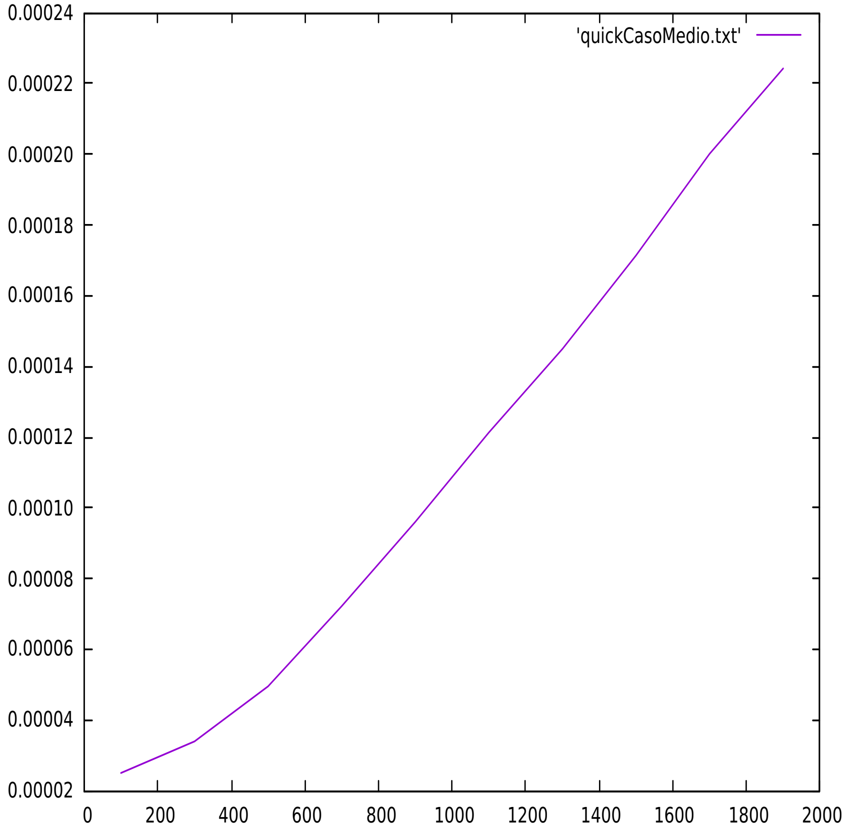

Figure 23: Comportamento do algoritmo quick sort no caso médio

Comparativo dos casos

Como pode-se observar, o pior caso, que é quadrático, é bem mais demorado que o restante, que são de ordem $O(n \log_2 n)$. Por mais parecidos que sejam, nota-se no gráfico do caso médio acaba sendo pior que o do melhor caso.

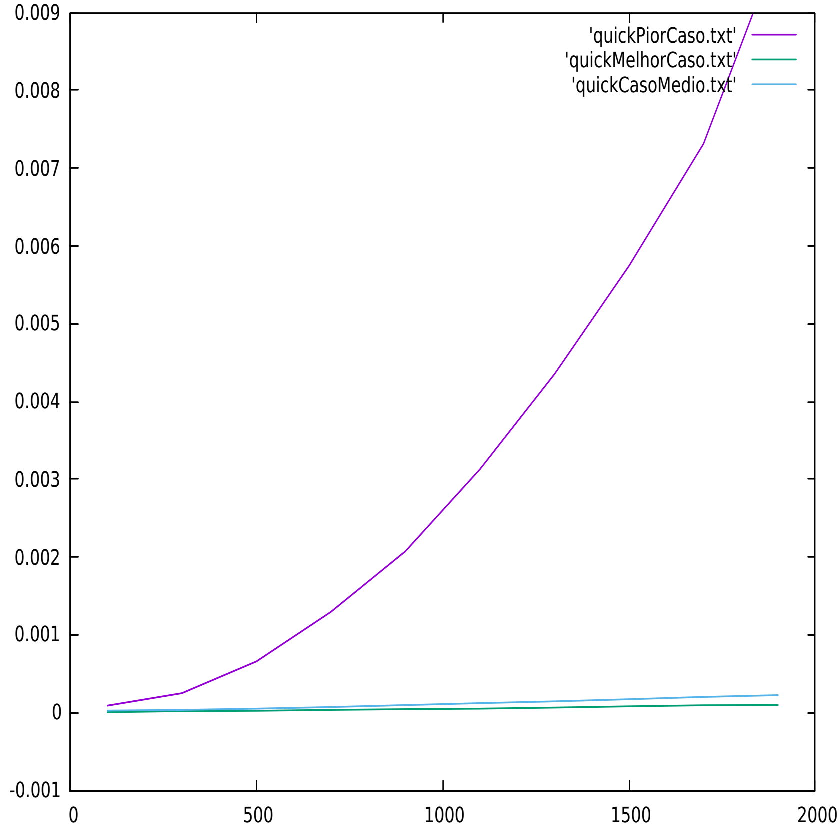

Figure 24: Comparativos dos casos do quick sort

Merge sort

Como o quick sort, o merge sort é um algoritmo divide and conquer. Baseia-se na ideia de dividir um vetor em várias sub-vetores até que cada sub-vetor consista de um único elemento e mesclar esses sub-vetores de uma maneira que resulte em um vetor ordenado.

```
void mergeSort(int v[], int s, int e){
    int m;
    if(s < e){
        m = (s + e)/2;
        mergeSort(v, s, m);
        mergeSort(v, m+1, e);
        merge(v, s, m, e);
    }
}

void merge(int v[], int s, int m, int e
    int i = s, j = m + 1, k = 0;
    int* w = (int*)malloc((e - s + 1) *
    while(k < (e - s + 1)){
        if((v[i] < v[j]) && ((i <= m) |
            w[k] = v[i];
            i++;
        }
        else{
            w[k] = v[j];
            j++;
        }
        k++;
    }
    k = 0;
    while(k < (e - s + 1)){
        v[s + k] = w[k];
        k++;
    }
    free(w);
}
```

Figure 25: Algoritmo do merge sort

Análise de complexidade

A complexidade desse algoritmo é de ordem $\Theta\left(n \log_2 n\right)$. Primeiro calculou-se a função merge:

$$T^m\left(n\right) = C1+C2+\left(n+1\right)C3+nC4+\left(C5+C6+C7+C8\right)\frac{n}{2}+nC9+C10+\left(n+1\right)C11+\left(C12+C13\right)n+C14$$

$$T^m\left(n\right) = \left(C3+C4+C9+C11+C12+C13\right)n+\left(C5+C6+C7+C8\right)\frac{n}{2}+C1+C2+C3+C11+C14$$

Considere $a = C1+2C2+C3+C4+C5$ na equação do merge sort:

$$T\left(1\right) = C1$$

$$T\left(n\right) = a+2T\left(\frac{n}{2}\right)+T^m\left(n\right)$$

$$T\left(\frac{n}{2}\right) = a+2T\left(\frac{n}{4}\right)+T^m\left(\frac{n}{2}\right)$$

$$T\left(n\right) = a+2\left[a+2T\left(\frac{n}{4}\right)+T^m\left(\frac{n}{2}\right)\right]+T^m\left(n\right)$$

$$T\left(n\right) = 3a+4T\left(\frac{n}{4}\right)+2T^m\left(\frac{n}{2}\right)+T^m\left(n\right)$$

$$T\left(\frac{n}{4}\right) = a+2T\left(\frac{n}{8}\right)+T^m\left(\frac{n}{4}\right)$$

$$T\left(n\right) = 3a+4\left[a+2T\left(\frac{n}{8}\right)+T^m\left(\frac{n}{4}\right)\right]+T^m\left(\frac{n}{2}\right)+T^m\left(n\right)$$

$$T\left(n\right) = 7a+8T\left(\frac{2}{8}\right)+4T^m\left(\frac{n}{4}\right)+T^m\left(\frac{n}{2}\right)+T^m\left(n\right)$$

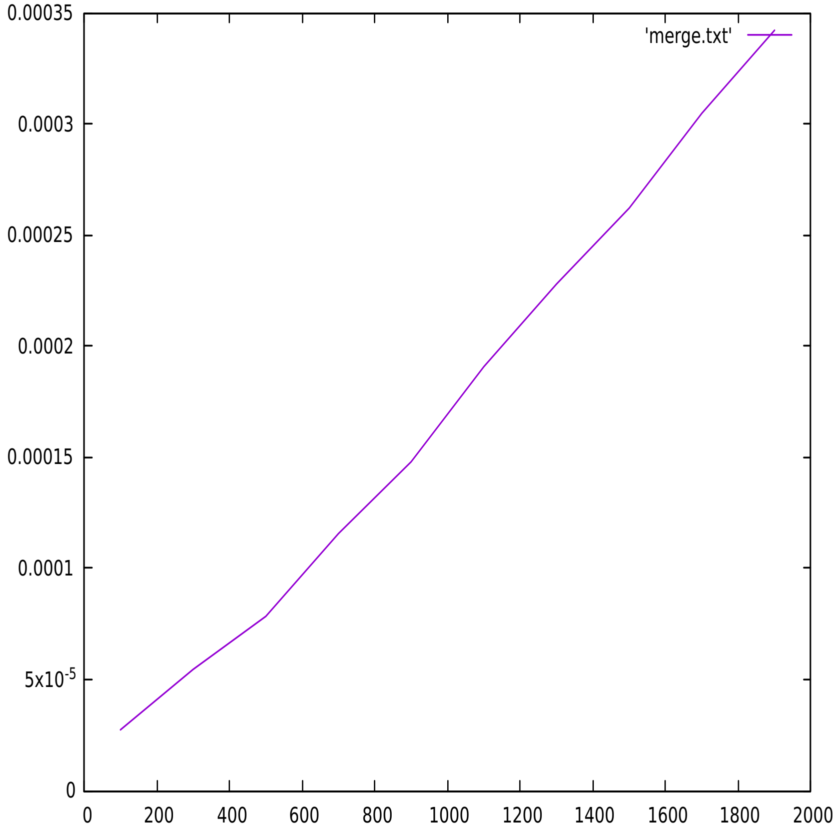

Figure 26: Comportamento do algoritmo do merge sort

Counting sort

O counting sort guarda cada ocorrência de elementos menores ao da posição que o vetor já se encontra e registra essa quantidade em um vetor auxiliar, após isso, o mesmo preenche outro vetor auxiliar a partir da posição de cada elemento.

```
int *countingSort(int v[], int n){
    int i, j, *c, *w;
    c = zeros(n);
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            if(v[j] < v[i]){
                c[i] = c[i] + 1;
            }
        }
    }
    w = zeros(n);
    for(i = 0; i < n; i++){
        w[c[i]] = v[i];
    }
    return w;
}
```

Figure 27: Algoritmo do counting sort

Análise de complexidade

A complexidade desse algoritmo é de ordem $O\left(n^2\right)$.

$$T\left(n\right) = C1{+}C2{+}2T^z\left(n\right){+}\left(n+1\right)C3{+}n\left(n+1\right)C4{+}c^2C5{+}\frac{n^2-n}{2}C6{+}\left(n+1\right)C7{+}nC8{+}C9$$

$$T\left(n\right) = n^2C5{+}\frac{n^2-n}{2}C6{+}n^2C4{+}\left(C3+C4+C7+C8\right)n{+}C1{+}C2{+}C3{+}C4{+}C9{+}2T^z$$

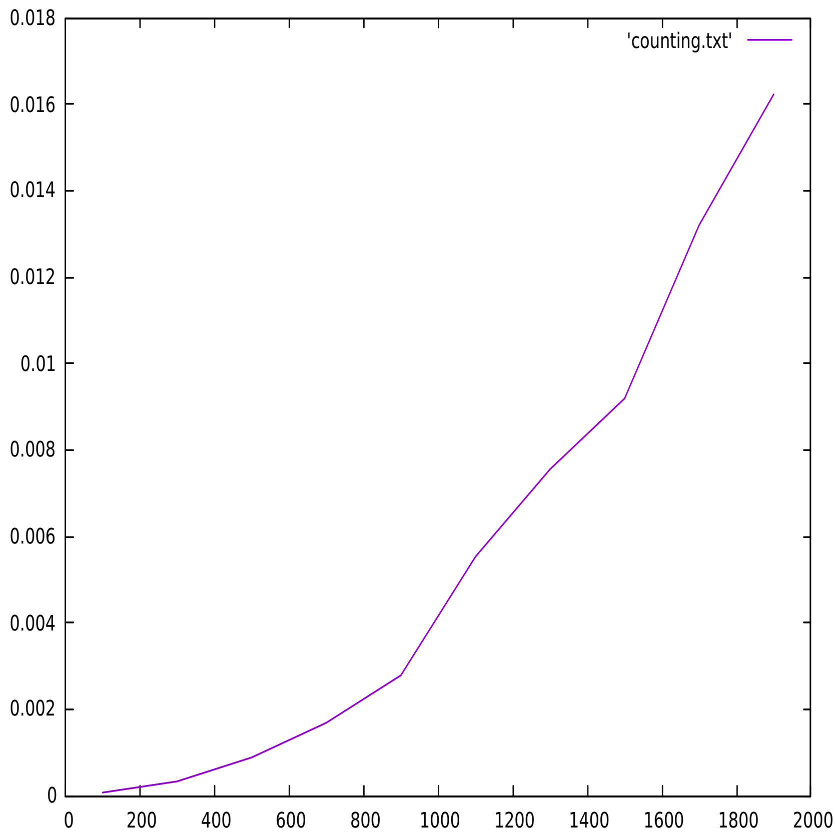

Figure 28: Comportamento do algoritmo do counting sort

Bogo sort

O algoritmo consiste no seguinte:

O vetor está ordenado? Se sim, então fim do algoritmo. Se não, embaralhe ele aleatoriamente e volte para o começo.

Este algoritmo é extremamente simples, mas também extremamente ineficiente. E consiste basicamente em embaralhar o vetor tantas vezes quanto forem necessárias até que por pura sorte e acaso, a ordenação aleatória dos elementos acabe sendo a correta. Mesmo ordenações parcialmente ordenadas ou quase ordenadas são completamente e cegamente descartadas em sua totalidade e de nada acabam ajudando.

```
void bogoSort(int v[], int n){
    while(sorted(v, n) != true){
        shuffle(v, n);
    }

}
bool sorted(int v[], int n){
    for(int i = 0; i < n - 1; i++){
        if(v[i] > v[i + 1]){
            return false;
        }
    }
    return true;
}
void shuffle(int v[], int n){
    for(int i = 0; i < n; i++){
        int j = myrand(0, n - 1);
        int aux = v[i];
        v[i] = v[j];
        v[j] = aux;
    }
}
```

Figure 29: Algoritmo do bogo sort

Análise de complexidade

Este algoritmo tem três funções, uma chamada no exemplo de sorted, que verifica se o vetor está ordenado, uma outra chamada shuffle que embaralha o vetor e a própria função bogoSort, que verifica através da função sorted se o vetor está ordenado, caso não esteja, chama a função shuffle para embaralhá-lo. A complexidade deste algoritmo é de ordem $O(n!)$, visto que necessita testar em média, todas as possíveis configurações do vetor.

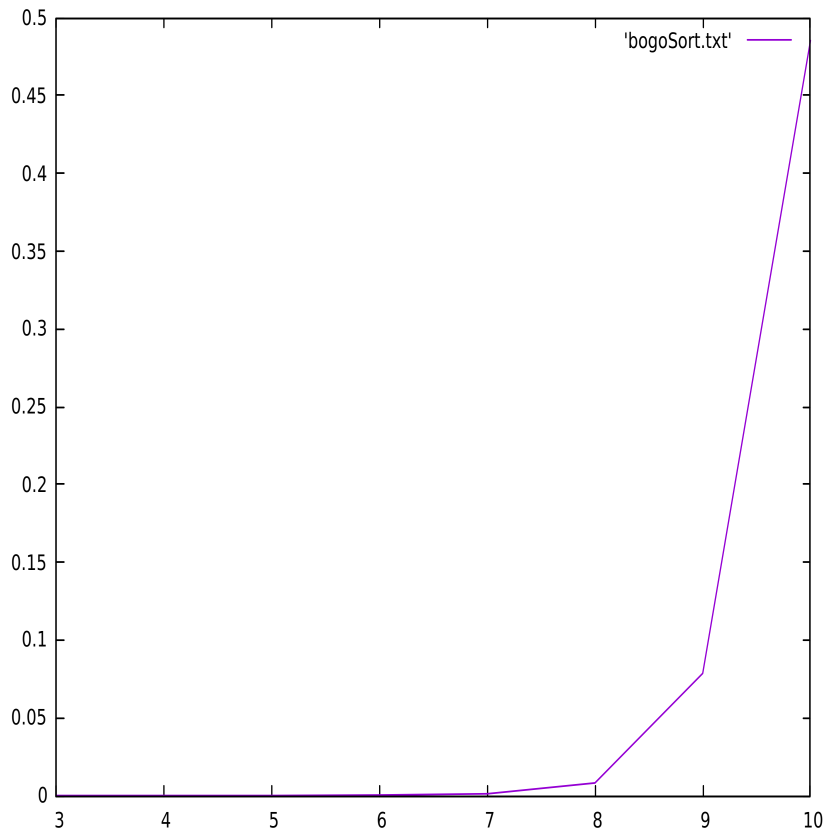

Figure 30: Comportamento do algoritmo do bogo sort

Comparações finais

Algoritmos básicos

Fazendo um comparativo dos dois algoritmos básicos percebe-se que o bubble sort necessita de um tempo bem maior pra ordenar os elementos.

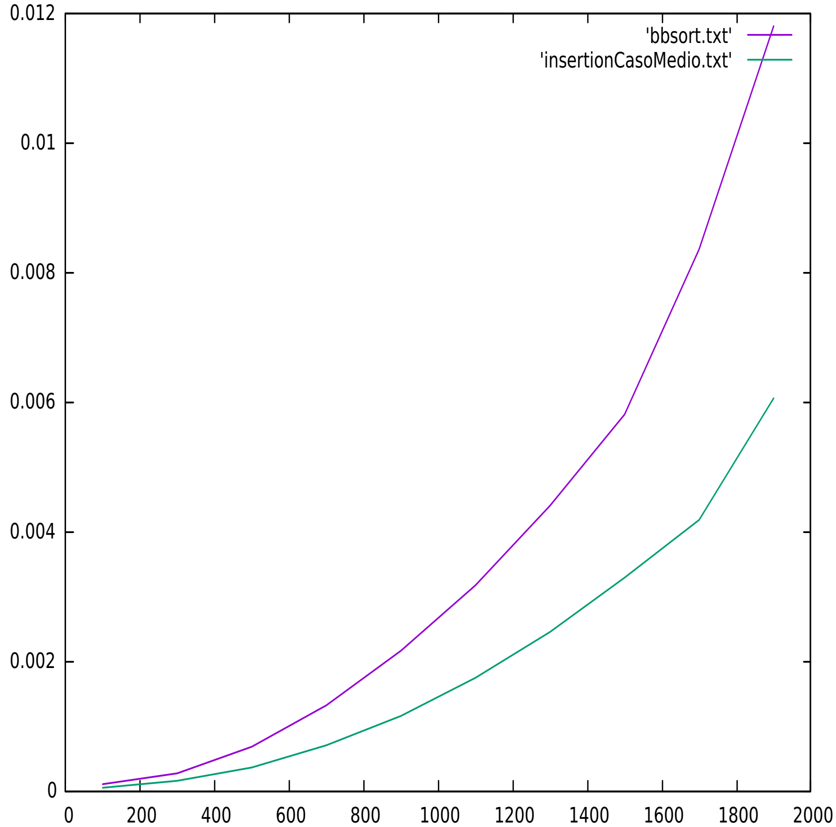

Figure 31: Algoritmos básicos

Melhores casos

Dentre os algoritmos que contém pior caso, o melhor é o quick sort, que acaba se sobressaindo para quantidades maiores.

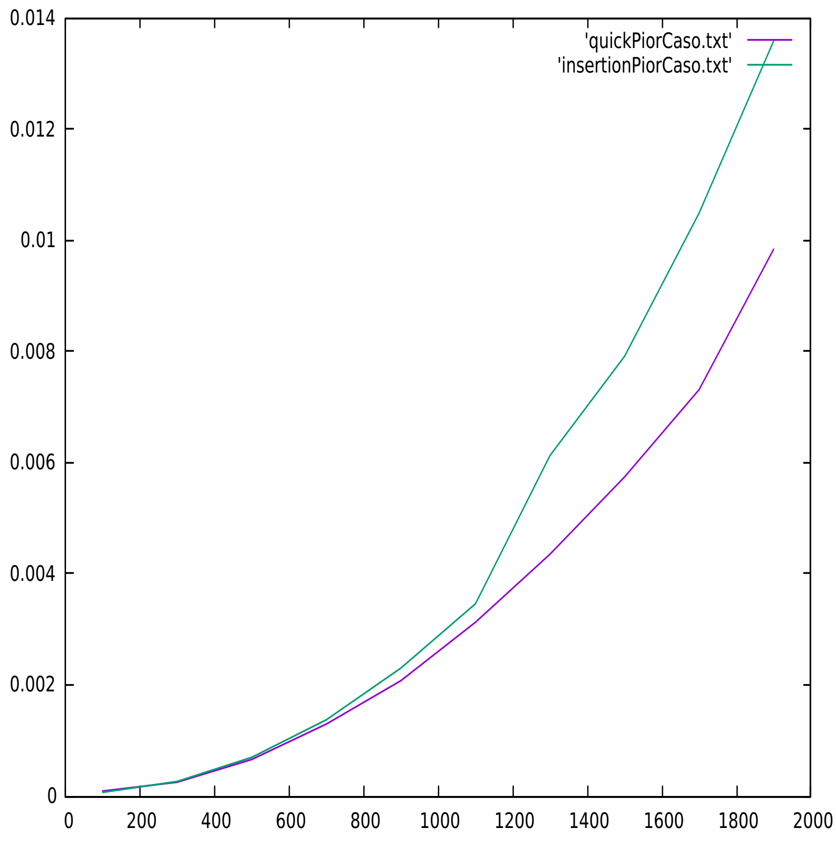

Figure 32: Insertion vs quick

Casos únicos

Nos algoritmos que contém apenas casos únicos, o distribution se sobressai sobre todos, e o counting é o que mais leva tempo para ser rodado, com excessão do bogo sort.

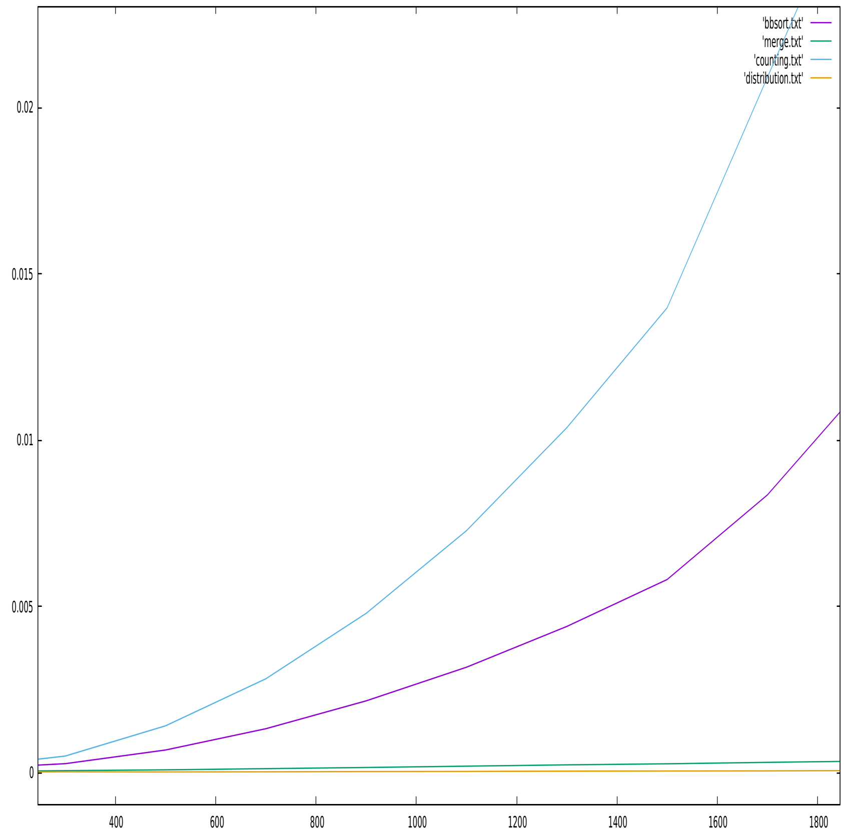

Figure 33: Counting vs bubble vs merge vs distribution

Quadráticos

Dos algoritmos quadráticos o que se destaca por sua rapidez é o insertion sort no caso médio. O pior acaba sendo o counting sort.

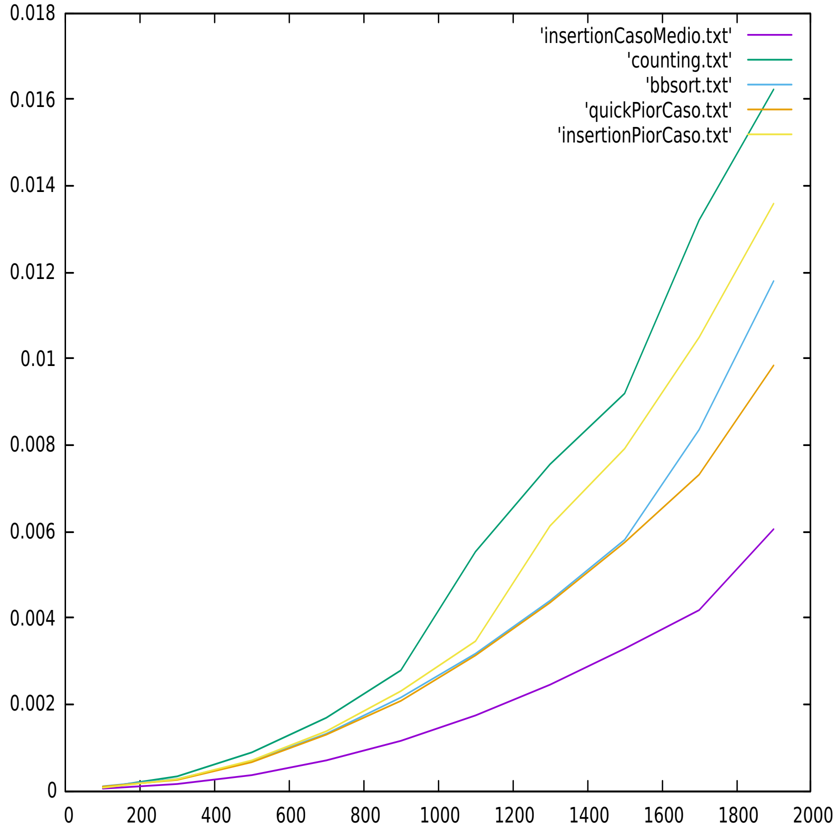

Figure 34: Comparativo dos algoritmos quadráticos

Recursivos

Dos algoritmos recursivos o quick se sobressai sobre o merge.

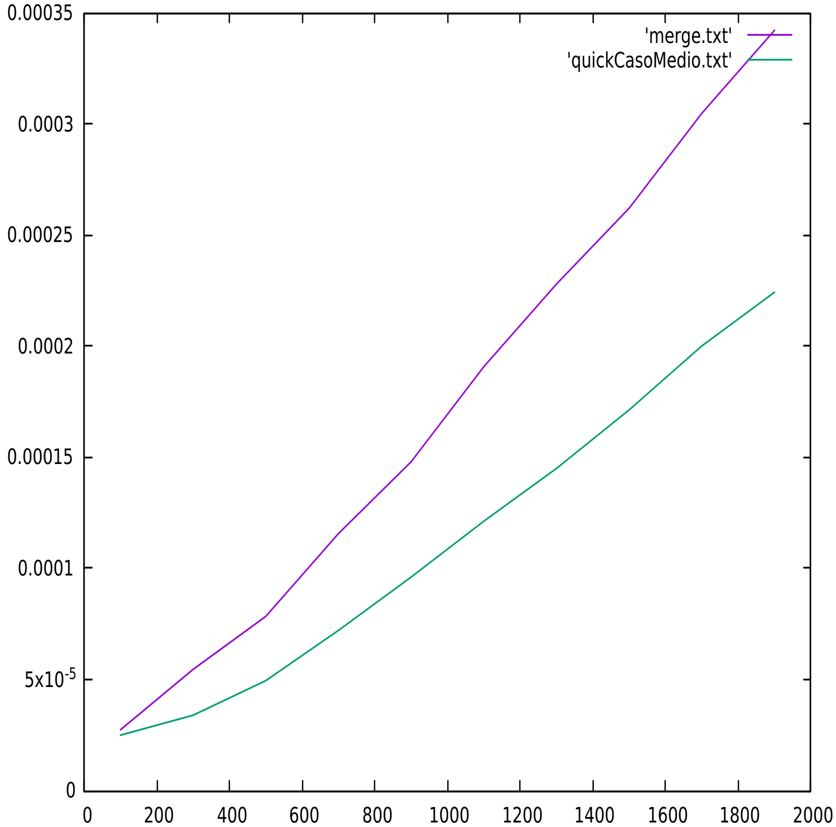

Figure 35: Quick vs merge

Todos os algoritmos

Dentre todos os algoritmos de ordenação citados o que supera o restante é o distribution, mas, por sua inviabilidade em máquinas com pouca memória, não é o melhor em todos os casos.

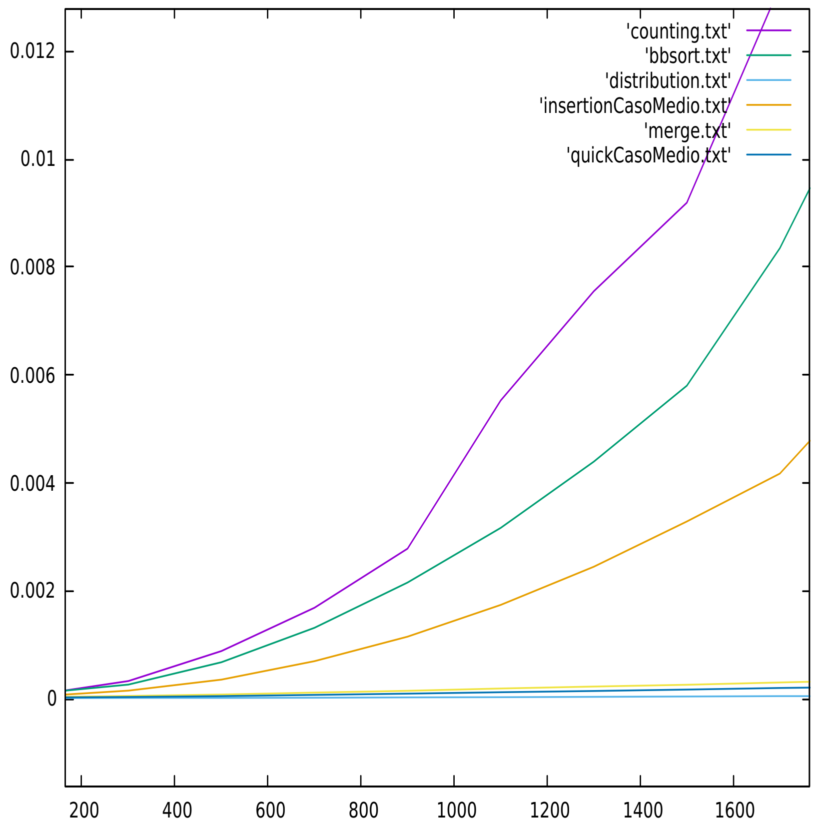

Figure 36: Todos os algoritmos de ordenação

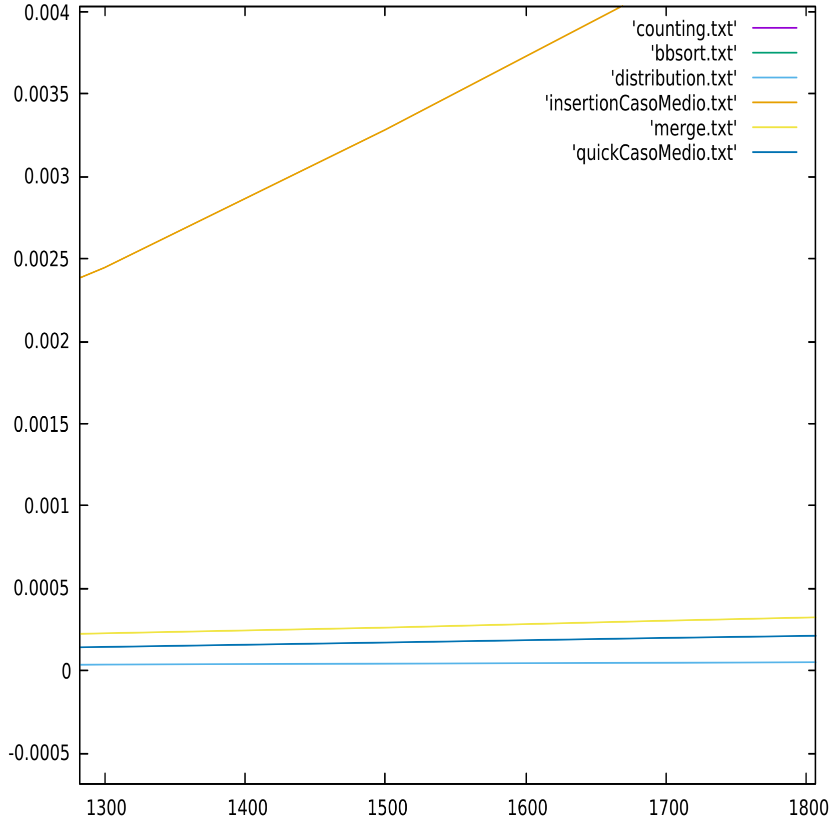

Figure 37: Zoom dos algoritmos mais rápidos