Universidade Federal Rural de Pernambuco
Bacharelado em Ciência da Computação

EP 1 - Análise de Algorítmos Recursivos e Iterativos

Aluno: Bruno Henrique Gusmão Vasconcelos
Professor: Rodrigo de Souza

Outubro
2017

Universidade Federal Rural de Pernambuco
Bacharelado em Ciência da Computação

Relatório

Primeiro Relatório de Algoritmos e Estruturas de Dados do Curso de Ciência da Computação ofertado pela Universidade Federal Rural de Pernambuco.

Aluno: Bruno Henrique Gusmão Vasconcelos

Professor: Rodrigo de Souza

Outubro
2017

Conteúdo

1 Resumo

Este relatório tem como objetivo analisar algoritmos recursivos implementados e compará-los com suas versões iterativas através da quantidade de operações executadas, ou seja, utilizando método experimental para obtenção de resultados.

2 Descrição de atividades

Para obtenção de resultados, utilizaremos algoritmos recursivos e suas versões iterativas, escritos na linguagem C e compilados com o compilador TDM-GCC 4.9.2 64-bit da IDE Dev-C++, e faremos uma comparação entre o número de operações realizadas entre estes algoritmos.
Os algoritmos utilizados na análise foram:

- MAX-REC e MAX-IT, sendo o primeiro um algoritmo recursivo do tipo divisão-e-conquista e o segundo sua versão iterativa, para encontrar o valor máximo de um vetor de inteiros.
- CRESC-REC e CREC-IT, sendo o primeiro um algoritmo recursivo do tipo divisão-e-conquista e o segundo sua versão iterativa, para ordenar um vetor de forma crescente.
- LOC-REC e LOC-IT, sendo o primeiro um algoritmo recursivo do tipo divisão-e-conquista e o segundo sua versão iterativa, para encontrar a posição de um inteiro x em um vetor crescente de inteiros.
- SEG-REC e SEG-IT, sendo o primeiro um algoritmo recursivo do tipo divisão-e-conquista e o segundo sua versão iterativa, para calcular o segmento de soma máxima de um vetor de inteiros.

3 Análise dos Resultados

3.1 MAX-REC

O algoritmo MAX-REC tem como objetivo encontrar o valor máximo de um vetor de inteiros de forma recursiva, utilizando divisão-e-conquista.

Código:

```
int max_rec(int* A, int p, int r)
{
        if(p == r)
        {
                COUNT++;
                return A[p];
        }
        else
        {
                int result, x, y;
                int q = (p + r) / 2;

                x =  max_rec(A, p, q);
                COUNT++;
                y = max_rec(A, q+1 , r);
                COUNT++;

                result = max( x , y );

                return result;
        }
}
```

O algoritmo MAX-REC divide o vetor em dois segmentos repetidamente até que sobre um segmento de um ou dois valores, a partir disso ele retorna para a chamada anterior o maior valor local, tomando dois a dois, até que a primeira função retorne o maior valor absoluto.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento linear n.

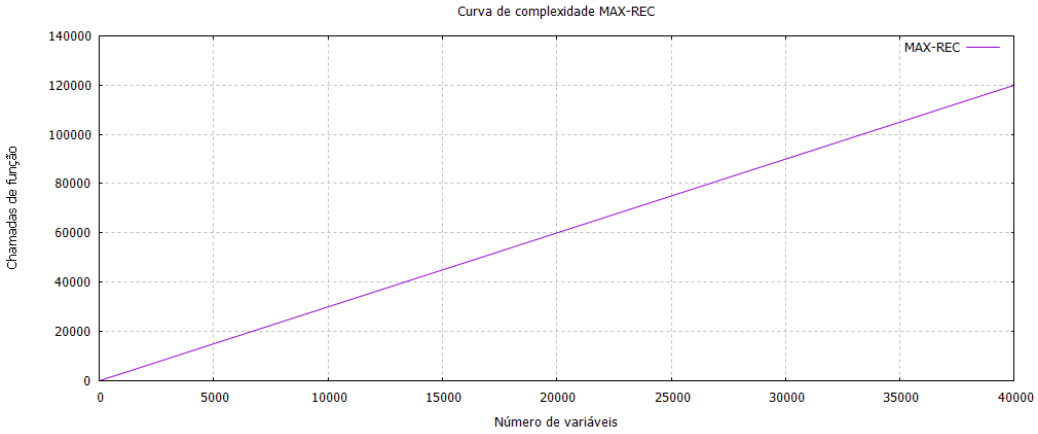

Figura 1: Curva de complexidade MAX-REC

3.2 MAX-IT

O algoritmo MAX-IT tem como objetivo encontrar o valor máximo de um vetor de inteiros de forma iterativa.

Código:

```
int max_it(int* A, int tamA){
        int i;
        int max = A[0];

        for(i = 0 ; i < tamA ; i++)
        {
                COUNT++;
                if(A[i] > max)
                {
                        COUNT++;
                        max = A[i];
                }
        }

        return max;
}
```

O algoritmo MAX-IT percorre todo o vetor uma única vez e compara o valor do índice atual com o valor máximo até aquele momento, se o valor do índice atual for maior que o máximo até então, o valor máximo recebe o valor do índice atual até que percorra todo o vetor.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento linear n.

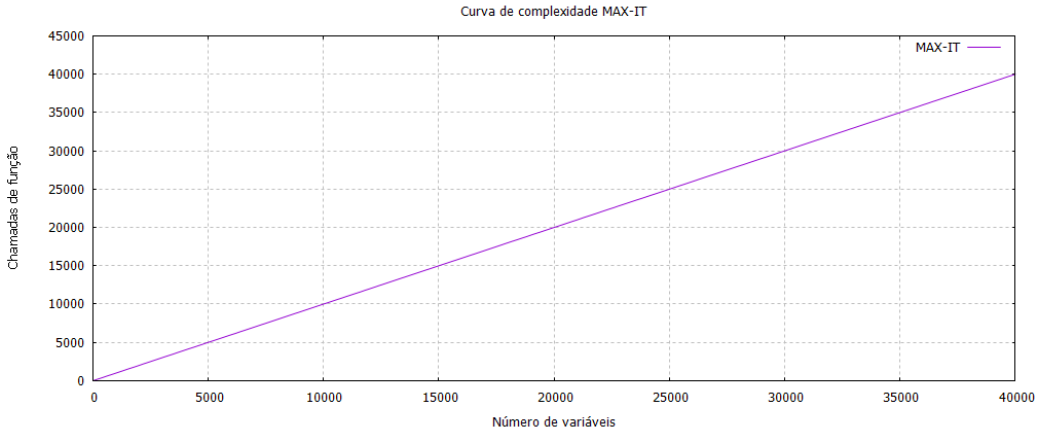

Figura 2: Curva de complexidade MAX-IT

3.3 CRESC-REC

O algoritmo CRESC-REC tem como objetivo ordenar um vetor de inteiros de forma recursiva, utilizando divisao-e-conquista.

Código:

```
1  void cresc_rec(int* arr, int l, int r)
2  {
3      if (l < r)
4      {
5          int m = l+(r-l)/2;
6
7          cresc_rec(arr, l, m);
8          COUNT++;
9          cresc_rec(arr, m+1, r);
10         COUNT++;
11
12         merge(arr, l, m, r);
13         COUNT++;
14     }
15 }
16
17 void merge(int* arr, int l, int m, int r)
18 {
19     int i, j, k;
20     int n1 = m - l + 1;
21     int n2 =  r - m;
22
23     int L[n1], R[n2];
24
25     for (i = 0; i < n1; i++)
26         L[i] = arr[l + i];
27     for (j = 0; j < n2; j++)
28         R[j] = arr[m + 1+ j];
29
30     i = 0;
31     j = 0;
32     k = l;
33     while (i < n1 && j < n2)
34     {
35         if (L[i] <= R[j])
36         {
37             arr[k] = L[i];
```

```
38              i++;
39          }
40          else
41          {
42              arr[k] = R[j];
43              j++;
44          }
45          k++;
46      }
47
48      while (i < n1)
49      {
50          arr[k] = L[i];
51          i++;
52          k++;
53      }
54
55      while (j < n2)
56      {
57          arr[k] = R[j];
58          j++;
59          k++;
60      }
61  }
```

O algoritmo CRESC-REC divide o vetor em dois segmentos repetidamente até que sobre um segmento de um ou dois valores, a partir disso ele chama a função *merge* que é a continuação da função de forma encapsulada, ela tem como objetivo criar dois subarrays e ordená-los entre si, como o vetor foi destrinchado em segmentos de um ou dois valores, essa função ordena localmente dois a dois até que chegue de volta na primeira chamada da função e retorne a ordenação absoluta.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento linear n.

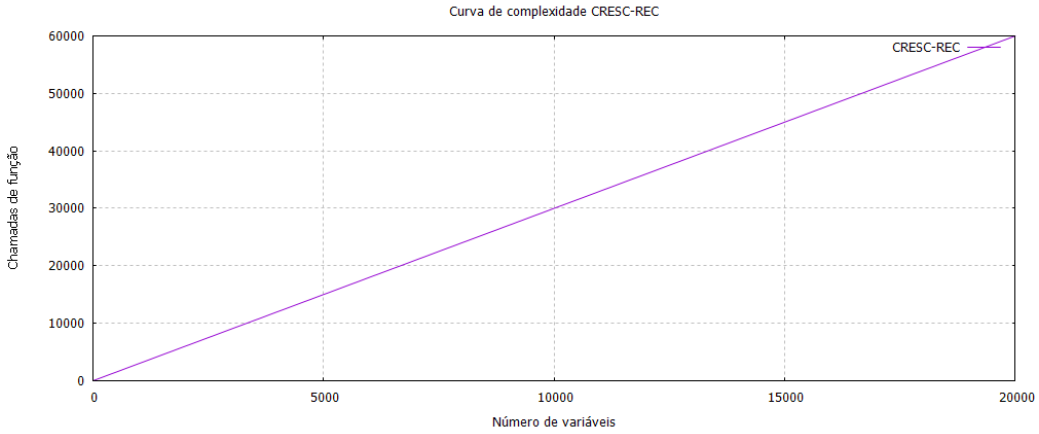

Figura 3: Curva de complexidade CRESC-REC

3.4 CRESC-IT

O algoritmo CRESC-IT tem como objetivo ordenar um vetor de inteiros de forma iterativa.

Código:

```
1  void cresc_it(int* A, int p, int n)
2  {
3          int i, j, aux;
4          for(j = 0 ; j <= n ; j++)
5          {
6                  COUNT++;
7                  for(i = 1 ; i <= n ; i++)
8                  {
9                          COUNT++;
10                         if(A[i] < A[i-1])
11                         {
12                                 aux = A[i];
13                                 A[i] = A[i-1];
14                                 A[i-1] = aux;
15                         }
16                 }
17         }
18 }
```

O algoritmo CRESC-IT percorre todo o vetor comparando o valor do índice atual com o valor do índice anterior, se o valor do índice atual for menor

que o anterior eles trocam de posição, esse processo é repetido o número de vezes igual ao número de variáveis de entrada.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento n^2.

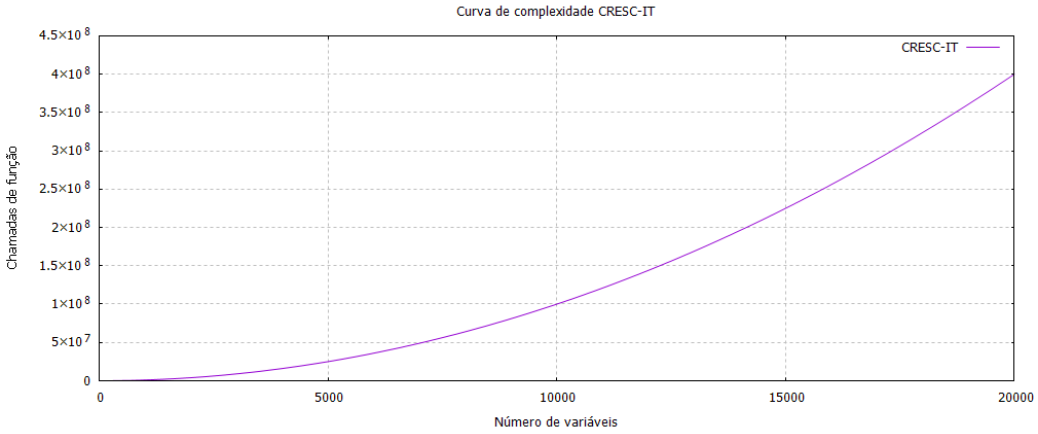

Figura 4: Curva de complexidade CRESC-IT

3.5 LOC-REC

O algoritmo LOC-REC tem como objetivo encontrar um valor x em um vetor de inteiros ordenado de forma recursiva, utilizando divisão-e-conquista.

Código:

```
1
2  int loc_rec(int* a, int p, int r, int x)
3  {
4          if(p == r-1)
5          {
6                  COUNT++;
7                  return r;
8          }
9          else
10         {
11                 int q = (p + r) / 2;
12                 if(a[q] < x)
13                 {
14                         COUNT++;
15                         return loc_rec(a, q, r, x);
16                 }
17                 else
18                 {
19                         COUNT++;
20                         return loc_rec(a, p, q, x);
21                 }
22         }
23 }
```

O algoritmo LOC-REC divide o vetor em dois segmentos e analisa se o valor procurado é menor ou maior que o valor da metade do segmento, se o valor for maior ele repete o processo a partir da metade do segmento, se for menor ele repete o processo até a metade do segmento. Este processo é repetido até que o valor intermediário seja o procurado.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento $nlog(n)$.

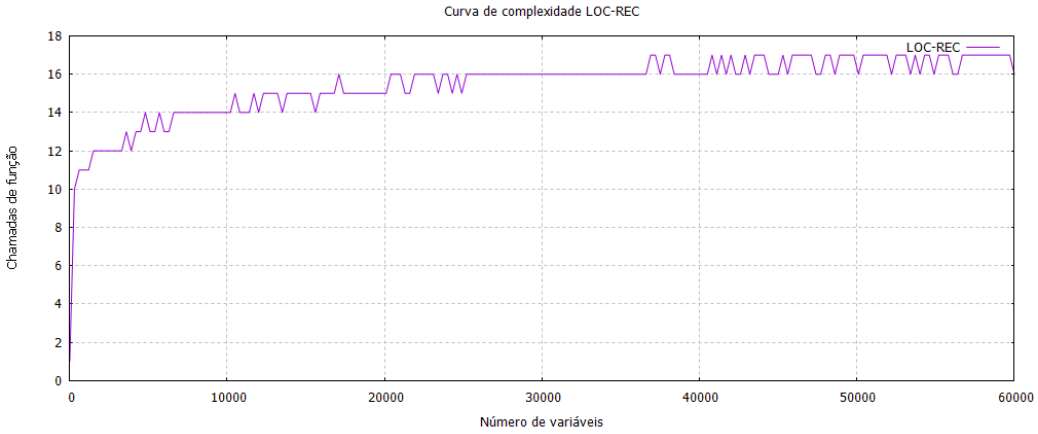

Figura 5: Curva de complexidade LOC-REC

3.6 LOC-IT

O algoritmo LOC-IT tem como objetivo encontrar um valor x em um vetor de inteiros ordenado de forma iterativa.

Código:

```
1  int loc_it(int* a, int n, int x)
2  {
3          int p = 0;
4          int r = n - 1;
5          while(p < r - 1)
6          {
7                  COUNT++;
8                  int q = (p + r) / 2;
9                  if(a[q] < x)
10                 {
11                         p = q;
12                 }
13                 else
14                 {
15                         r = q;
16                 }
17         }
18         return r;
19 }
```

O algoritmo LOC-IT divide o índice máximo por 2 e analisa se o valor procurado é menor ou maior que o intermediário do segmento, se o valor for

maior ele repete o processo a partir da metade do segmento, se for menor ele repete o processo até a metade do segmento. Este processo é repetido até que o valor intermediário seja o procurado.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um crescimento $nlog(n)$.

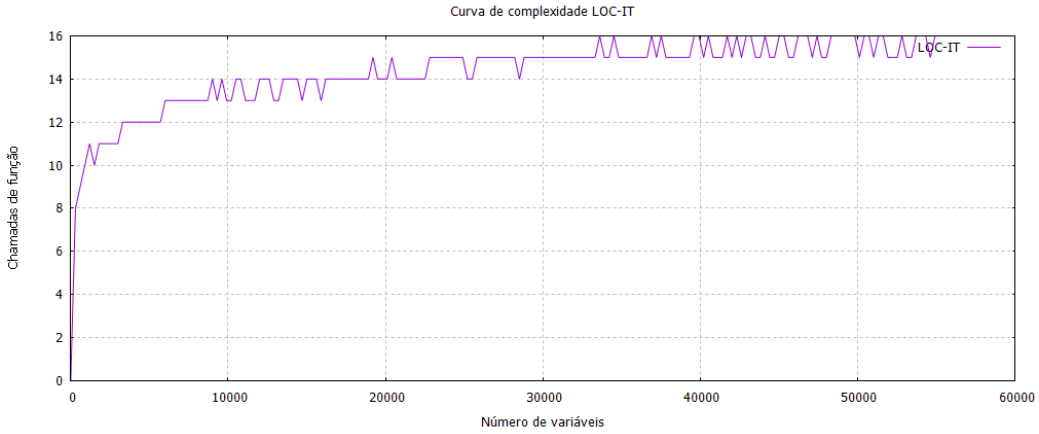

Figura 6: Curva de complexidade LOC-IT

3.7 SEG-REC

O algoritmo SEG-REC tem como objetivo encontrar o segmento de soma máxima de um vetor de inteiros de forma recursiva, utilizando divisão-e-conquista.

Código:

```
1  int seg_rec(int* a, int p, int r)
2  {
3          if(p == r)
4          {
5                  return a[p];
6          }
7          else
8          {
9                  int x1, x2;
10                 int q = (p + r) / 2;
11
12                 x1 = seg_rec(a, p, q);
13                 COUNT++;
14
15                 x2 = seg_rec(a, q+1, r);
16                 COUNT++;
17
18                 int s = a[q];
19                 int y1 = s;
20                 int i;
21                 for(i = q - 1 ; i > p ; i--)
22                 {
23                         s += a[i];
24                         if(s > y1)
25                         {
26                                 y1 = s;
27                         }
28                 }
29                 s = a[q+1];
30                 int y2 = s;
31                 int j;
32                 for(j = q + 2 ; j < r ; j++)
33                 {
34                         s += a[j];
35                         if(s > y2)
36                         {
```

```
37                                      y2 = s;
38                                  }
39                              }
40                              int x = max(max(x1 , y1 + y2), x2);
41                              return x;
42                      }
43  }
```

O algoritmo SEG-REC divide o vetor em dois segmentos repetidamente até que sobre um valor e o armazena, a cada retorno da função ele verifica se a soma com o próximo valor é maior que a anterior, se sim ele continua retornando os valores até obter o maior segmento.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um comportamento $f(n) = n$.

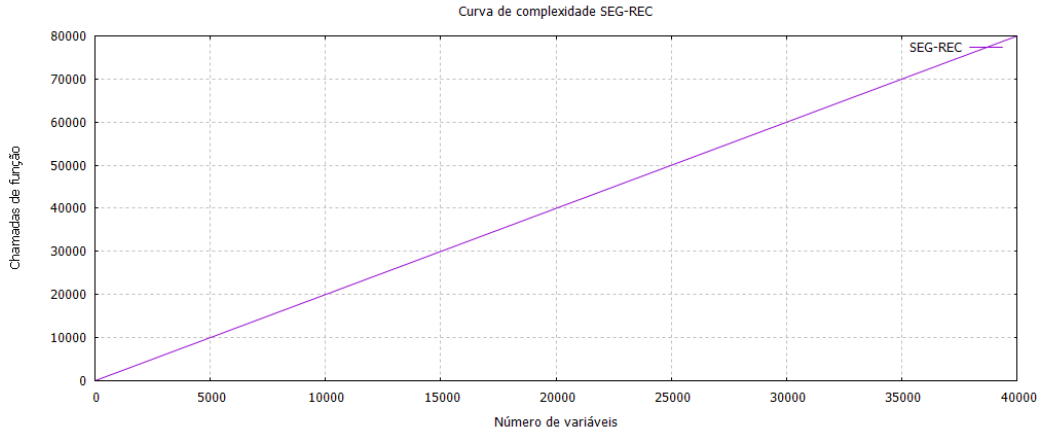

Figura 7: Curva de complexidade SEG-REC

3.8 SEG-IT

O algoritmo SEG-IT tem como objetivo encontrar o segmento de soma máxima de um vetor de inteiros de forma iterativa.

Código:

```
int seg_it(int* a, int p, int r)
{
        int x = a[r];
        int q;
        int s;
        int j;
        for(q = r - 1 ; q >= p ; q--)
        {
                COUNT++;
                s = 0;
                for(j = q ; j <= r ; j++)
                {
                        COUNT++;
                        s = s + a[j];
                        if(s > x)
                        {
                                x = s;
                        }
                }
        }
        return x;
}
```

O algoritmo SEG-IT percorre o vetor verificando os elementos um a um, depois verifica a soma dois a dois, depois a soma três a três até verificar a soma do vetor inteiro e retorna o maior segmento de soma calculado.

Abaixo podemos observar o gráfico de complexidade deste algoritmo e concluir que ele possui um comportamento n^2.

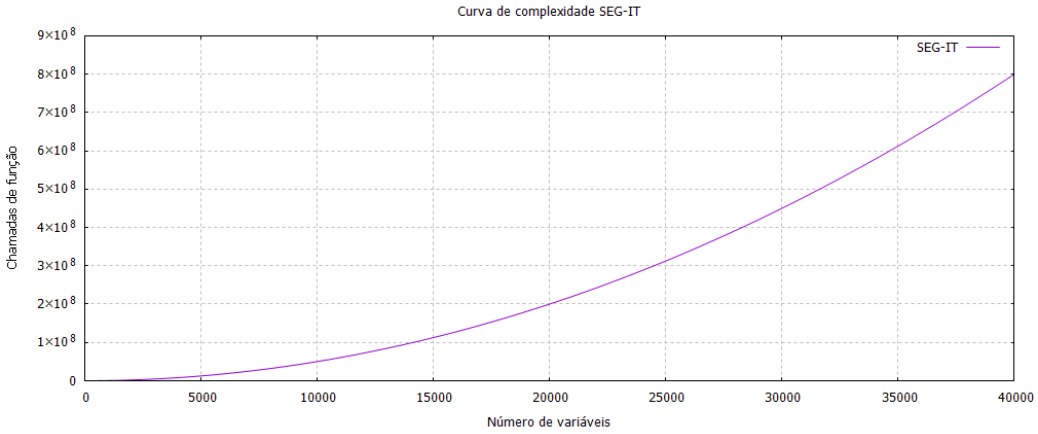

Figura 8: Curva de complexidade SEG-IT

3.9 Comparações

Podemos comparar os gráficos de crescimento de cada algoritmo recursivo com sua versão iterativa e tirar conclusões sobre suas complexidades e eficiências.

3.9.1 MAX

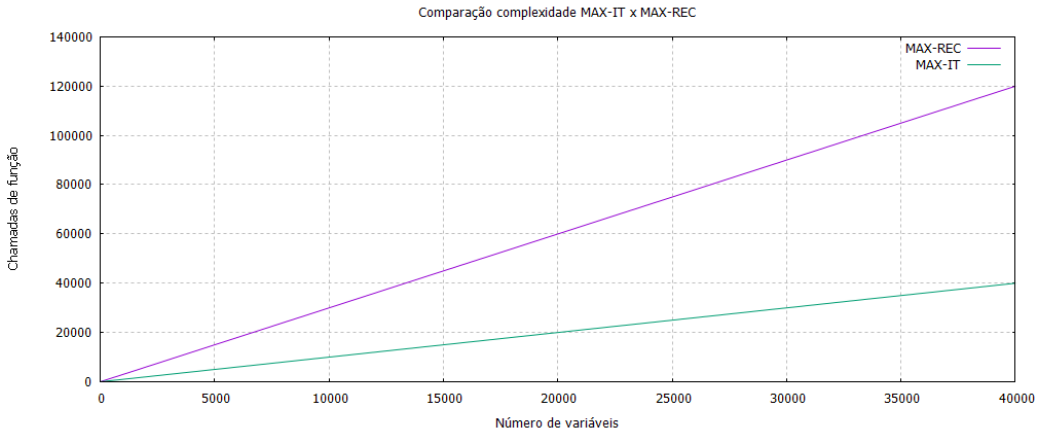

Figura 9: Comparação de complexidade MAX-REC x MAX-IT

Pelo gráfico traçado podemos observar que o algoritmo iterativo tem menor ângulo de inclinação, resultando numa maior eficiência.

Porém, se formos analisar a complexidade de forma grosseira, eles obedecem a mesma regra de complexidade n. O que é facilmente contornável com um processamento mais potente.

3.9.2 CRESC

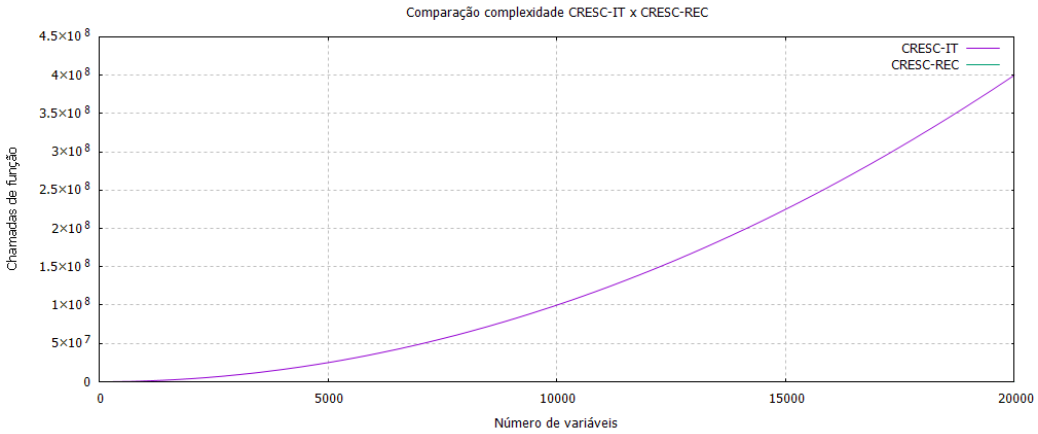

Figura 10: Comparação de complexidade CRESC-REC x CRESC-IT

Na comparação dos algoritmos CRESC-REC para CRESC-IT já podemos ver uma significante diferença. Olhando para o gráfico, quase não conseguimos observar o traço de CRESC-REC por ele ter uma complexidade n, enquanto o CRESC-REC possui complexidade n^2. Isso mostra que, neste caso, a solução recursiva é muito mais eficiente que a iterativa.

3.9.3 LOC

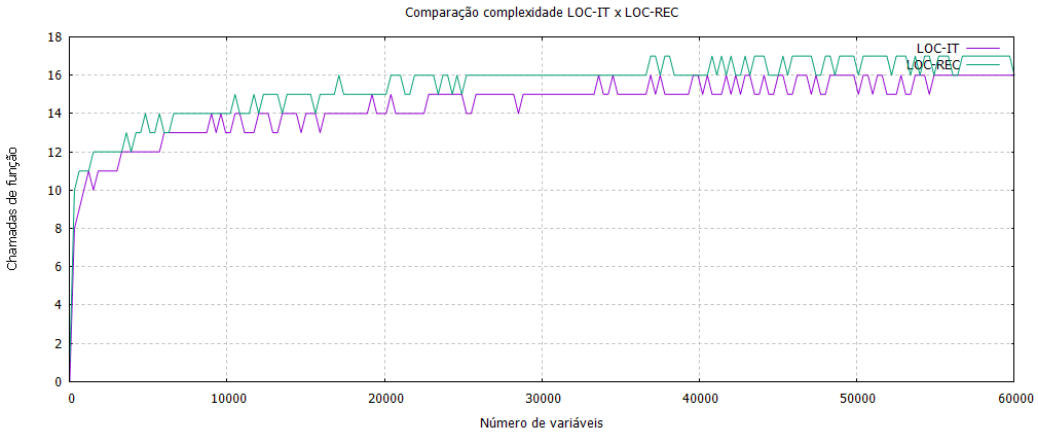

Figura 11: Comparação de complexidade LOC-REC x LOC-IT

Quando comparamos o gráfico dos algoritmos LOC-REC e LOC-IT, podemos ver que eles acompanham basicamente a mesma curva de complexidade $nlog(n)$, logo, são igualmente eficientes.

3.9.4 SEG

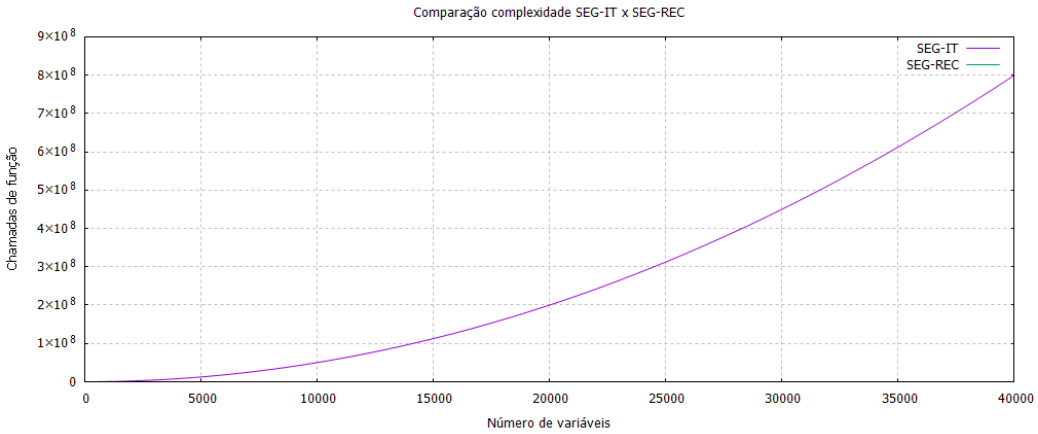

Figura 12: Comparação de complexidade SEG-REC x SEG-IT

Na comparação dos gráficos dos algoritmos SEG-REC e SEG-IT, assim como os algoritmos de ordenação CRESC, vemos que a curva de complexidade da forma iterativa cresce muito mais que a recursiva. Isso se dá porque o número de chamadas SEG-REC tem complexidade n, enquanto a SEG-IT possui complexidade n^2.

4 Conclusão

Pela observação dos dados obtidos e aspectos analisados podemos concluir que embora a solução recursiva não tenha sido a melhor em cem porcento dos casos, a facilidade de contornar uma situação é muito maior que um algoritmo iterativo, pois sua eficiência diferencia em apenas uma constante. Já nos casos onde a solução recursiva foi melhor, pudemos observar que a curva de complexidade diferenciava significantemente em relação ao modo iterativo, sendo muito difícil contornar a situação através de melhoria de processamento, tendo a necessidade de recorrer a um algoritmo mais eficiente.

Bibliografia

Referências Bibliográficas nas Normas ABNT de Livros e Sites (links) – Como Fazer. Disponível em: $< https : //www.normaseregras.com/normas - abnt/referencias >$. Acesso em: 29 out. 2017.

WILLIAMS, T.; KELLEY, C. gnuplot 5.0, An Interactive Plotting Program. Disponível em: $< http : //www.gnuplot.info/docs_5.0/gnuplot.pdf >$. Acesso em: 23 out. 2017.

FEOFILOFF, Paulo. Minicurso de Análise de Algoritmos. Disponível em: $< https : //www.ime.usp.br/\ pf/livrinho - AA/AA - BOOKLET.pdf >$.

Merge Sort. Disponível em: $< http : //www.geeksforgeeks.org/merge - sort/ >$. Acesso em: 18 out. 2017.