Vinicius Campos Tinoco Ribeiro
Vitor Rodrigues Greati

Análise empírica de algoritmos

Brasil
2015, v-1.0

Vinicius Campos Tinoco Ribeiro
Vitor Rodrigues Greati

Análise empírica de algoritmos

Relatório técnico apresentado à disciplina de Estrutura de Dados Básicas I, como requisito parcial para obtenção de nota referente à unidade I.

Universidade Federal do Rio Grande do Norte - UFRN

Instituto Metrópole Digital - IMD

Bacharelado em Tecnologia da Informação

Brasil

2015, v-1.0

Sumário

1 Introdução

Problemas computacionais são solucionados através de algoritmos, sequências de passos bem definidos capazes de prover uma saída a partir do processamento de um conjunto de entradas. Para um mesmo problema, é possível que existam diversos algoritmos corretos, os quais, embora culminem em resultados idênticos, podem se diferenciar pelo consumo de recursos do computador, sendo o processamento e a memória os mais considerados. Esse consumo é comumente mensurado por análises de complexidade, que permitem apontar as soluções mais eficientes - e, portanto, as melhores - em termos de tempo e espaço (SZWARCFITER MARKENZON, 1994).

A complexidade de um algoritmo pode ser determinada por meio de estudos empíricos de execução do programa com entradas de tamanhos crescentes, verificando-se, para cada uma, os consumos de tempo e memória, bem como a maneira como eles crescem. Gerando gráficos a partir dos resultados obtidos, é possível observar a qual função a complexidade do algoritmo obedece, o que pode ser confirmado por meio de análises matemáticas. A partir disso, basta comparar as ordens de crescimento dos diferentes algoritmos e identificar qual é o melhor como solução para o problema.

Este relatório objetiva realizar análises empíricas e comparações de algoritmos que resolvem dois problemas computacionais: o da soma máxima de uma subsequência e o da compactação de um arranjo. Foram implementados dois algoritmos para cada, um quadrático e um linear, e as análises de tempo foram executadas em um mesmo computador, em dois sistemas operacionais distintos. Todos os valores foram registrados em tabelas e plotados em gráficos, permitindo fácil comparação.

Nas seções seguintes, é apresentado o método seguido, abrangendo os materiais e ferramentas utilizadas, bem como o pseudocódigo dos algoritmos implementados; depois, são mostrados os resultados obtidos e, por fim, as discussões geradas a partir deles. O único apêndice traz a implementação em C++ dos algoritmos escolhidos.

2 Metodologia

2.1 Materiais utilizados

2.1.1 Computador

A Tabela 1 apresenta as principais especificações do computador utilizado nos testes:

Computador	Especificações
Dell Inspiron 14R N4110	Processador Intel Core i5-2430M 2.40 GHz 1 x 4 GB DDR3 1333MHz HD de 750 GB 5400 RPM Intel HD 3000 Graphics (DirectX 10.1 SM 4.1) e HD 6470M 1GB (DirectX 11.0 e SM 5.0)

Tabela 1: Principais especificações técnicas do computador utilizado.

Nesse mesmo computador, dois sistemas operacionais - Windows e GNU/Linux - foram utilizados como ambientes para os testes. A Tabela 2, abaixo, traz os detalhes técnicos de ambos.

Sistema	Especificações
GNU/Linux	Ubuntu 14.04 trusty (x86-64) Kernel 3.13.0-57-generic Cinnamon 2.6.12
Windows	Windows 10 Pro 64bits

Tabela 2: Detalhes técnicos dos sistemas operacionais adotados.

2.1.2 Ferramentas de programação

Os quatro algoritmos escolhidos foram implementados na linguagem C++, padrão ISO/IEC 14882:2011, ou simplesmente C++11.

No ambiente GNU/Linux, os códigos foram compilados pelo *G++*[1], na versão 4.8.4 (2013), através da linha:

g++ -std=c++11 -Wall <arquivos .cpp> -o <executável>

[1] http://www.cprogramming.com/g++.html

No Windows, o projeto *Cygwin*[2] foi utilizado em sua versão 4.9.2 para a execução dos algoritmos. Nele, o G++ foi executado através do comando:

g++ -std=c++11 <arquivos .cpp>

A biblioteca *chrono*[3] foi responsável pelas medições do tempo de execução.

2.1.3 Algoritmos

2.1.3.1 Problema da soma máxima de uma subsequência

Neste trabalho, o problema da soma máxima de uma subsequência foi definido como segue:

> Dada uma sequência de inteiros (possivelmente negativos) $A_1, A_2, \ldots, A_N$, encontrar o máximo valor de $\sum_{k=i}^{j} A_k$.
>
> Por conveniência, a soma máxima de uma subsequência é zero se todos os inteiros forem negativos. Por exemplo, para a entrada $(-2, 11, -4, 13, -5, -2)$, a resposta é 20 (A_2 até A_4).

Uma possível solução consiste em, para cada elemento, a partir do primeiro, realizar os testes para todas as subsequências que começam por ele, armazenando a maior soma para aquele conjunto de arranjos. Por fim, compara-se tal valor com a maior soma de todos os arranjos possíveis, a qual corresponde ao resultado esperado. Essa possibilidade está descrita no Algoritmo 1, no qual a presença de dois laços aninhados aponta para uma complexidade quadrática - a ser confirmada pelos testes e pela análise matemática,

[2] https://www.cygwin.com/
[3] http://www.cplusplus.com/reference/chrono/

dispostos nas próximas seções.

Algoritmo 1: Primeira solução para o problema da soma máxima de uma subsequência.

Input: Um arranjo $A = \{A_1, A_2, \ldots, A_N\}$
Result: Valor máximo de $\sum_{k=i}^{j} A_k$
begin
    // c1
    $soma\_total \leftarrow 0$;
    // c2
    for $i \leftarrow 1$ **to** N **do**
        // c3
        $soma\_local \leftarrow 0$;
        // c4
        for $j \leftarrow i$ **to** N **do**
            // c5
            $soma\_local \leftarrow soma\_local + A_j$;
            // c6
            $soma\_total \leftarrow max\{soma\_local, soma\_total\}$;
        end
    end
end
// c7
return *soma_total*

Outra solução traz uma abordagem mais simples. Partindo-se do primeiro elemento, somam-se os próximos, um a um, verificando se isso torna maior ou menor a soma calculada até então. Essa soma local, ao ser atualizada, é transmitida à soma geral, caso a maximize. Como mostrado no Algoritmo 2, itera-se sobre o arranjo apenas uma vez,

sugerindo uma complexidade linear.

Algoritmo 2: Segunda solução para o problema da soma máxima de uma subsequência.

Input: Um arranjo $A = \{A_1, A_2, \ldots, A_N\}$
Result: Valor máximo de $\sum_{k=i}^{j} A_k$
begin
    // c1
    $soma\_total \leftarrow 0$;
    // c2
    $soma\_local \leftarrow 0$;
    // c3
    for $i \leftarrow 1$ **to** N **do**
        // c4
        $soma\_local \leftarrow max\{soma\_local + A_i, 0\}$;
        // c5
        $soma\_total \leftarrow max\{soma\_local, soma\_total\}$;
    end
end
// c6
return *soma_total*

2.1.3.2 Problema da compactação de um arranjo

Para este problema, foi dada a seguinte definição:

> O problema da compactaçao de um arranjo consiste em percorrer um arranjo com números inteiros e "retirar"dele todos os elementos nulos ou negativos - este processo é denominado de compactação. A remoção, na verdade, consiste em mover os elementos positivos para a frente do vetor, preservando sua ordem relativa. Ao final da operação, o arranjo poderá ter seu tamanho lógico redefinido, caso alguma compactação tenha de fato ocorrido.
>
> Por exemplo, para um vetor com os elementos (-2, 11, -4, 0, -5, 21) com 6 elementos, a compactação resultaria no vetor (11, 21) com apenas 2 elementos.

A primeira solução é simples e ingênua: com um laço começando do último elemento do arranjo e terminando no primeiro, para cada $A_i \leq 0$, mover todos os elementos do arranjo atualizado a partir de A_{i+1} para a esquerda. Após essa operação, decrementa-se o tamanho do arranjo e se continua a percorrer a partir de A_{i-1} para a esquerda. O Algoritmo 3 demonstra como seria essa implementação, utilizando dois laços aninhados,

evidenciando o aspecto quadrático da solução.

Algoritmo 3: Primeira solução para o problema da compactação de um arranjo.

Input: Um arranjo $A = \{A_1, A_2, \ldots, A_N\}$
Result: O arranjo A sem elementos menores ou iguais a zero.
Output: O novo tamanho do arranjo A.

```
begin
    // c1
    tamanho_final ← N;
    // c2
    for i ← N to 1 do
        // c3
        if A_i ≤ 0 then
            // c4
            for j ← i to tamanho_final − 1 do
                // c5
                A_j ← A_{j+1};
            end
            // c6
            tamanho_final ← tamanho_final − 1;
        end
    end
end
// c7
return tamanho_final;
```

A outra solução para esse problema trabalha com dois índices, i e k, sendo o primeiro mais rápido, percorrendo todo o arranjo, e o segundo mais lento, apontando para a posição onde se deve inserir A_i quando $A_i > 0$, momento este em que k avança. Dessa maneira, todo $A_i \leq 0$ é ignorado. O Algoritmo 4 mostra como se utiliza apenas um

laço nessa abordagem, apontando para uma solução linear.

Algoritmo 4: Segunda solução para o problema da compactação de um arranjo.

Input: Um arranjo $A = \{A_1, A_2, \ldots, A_N\}$
Result: O arranjo A sem elementos menores ou iguais a zero.
Output: O novo tamanho do arranjo A.

```
begin
    // c1
    k ← 0;
    // c2
    for i ← 1 to N do
        // c3
        if A_i > 0 then
            // c4
            A_k ← A_i;
            // c5
            k ← k + 1;
        end
    end
end
// c6
return k;
```

2.2 Método de comparações

Os algoritmos foram comparados segundo os critérios de tempo de execução e número de passos (quantidade de execuções da operação dominante).

Nos testes, utilizaram-se 25 instâncias de cada problema, com o tamanho da entrada $n \in [100, 100.000]$ e incrementado em intervalos crescentes. Para cada valor de n, os algoritmos foram executados 100 vezes sobre uma instância aleatória. Calculou-se, em seguida, a média aritmética dos tempos de execução e do número de passos, para análise nos próximos capítulos.

3 Resultados

Para cada problema, são exibidos os resultados dos testes de tempo e de passos em tabelas e gráficos de acordo com o sistema operacional. Há gráficos para os algoritmos separadamente e outros que tornam a comparação entre eles mais evidente. No Capítulo 4, esses dados são melhor analisados e discutidos.

3.1 Problema da soma máxima de uma subsequência

3.1.1 GNU/Linux

A Tabela 3 apresenta os resultados dos testes realizados com os dois algoritmos deste problema, em ambiente GNU/Linux.

Tamanho da entrada	Algoritmo 1		Algoritmo 2	
	Tempo (s)	Passos	Tempo (s)	Passos
100	0.079621	5050	0.001678	100
200	0.141525	20100	0.003218	200
300	0.27774	45150	0.005243	300
400	0.483658	80200	0.006091	400
500	0.754683	125250	0.007694	500
600	1.07749	180300	0.008686	600
700	1.46482	245350	0.010663	700
800	1.94305	320400	0.013106	800
900	2.44865	405450	0.013879	900
1000	2.99772	500500	0.01577	1000
1500	6.72688	1125750	0.022514	1500
2000	11.9881	2001000	0.021802	2000
2500	20.0968	3126250	0.02491	2500
3000	27.9293	4501500	0.029577	3000
4000	47.9656	8002000	0.039945	4000
5000	75.6699	12502500	0.053522	5000
10000	302.367	50005000	0.095658	10000
15000	670.136	112507500	0.143061	15000
20000	1195.27	200010000	0.191471	20000
30000	2854.12	450015000	0.281089	30000
40000	4945.71	800020000	0.370226	40000
50000	8086.35	1250025000	0.46028	50000
60000	11789.3	1800030000	0.561571	60000
80000	22346.1	3200040000	0.742661	80000
100000	32419.4	5000050000	0.925754	100000

Tabela 3: Tempos e passos para os dois algoritmos do primeiro problema, em ambiente GNU/Linux.

3.1.1.1 Análise de tempo

Gráfico 1 - Análise de tempo para o Algoritmo 1, em GNU/Linux

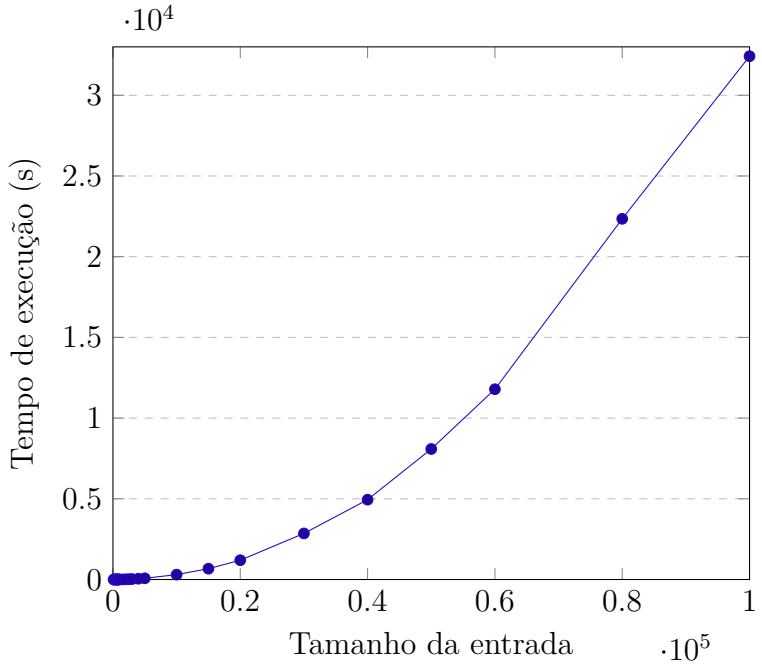


Gráfico 2 - Análise de tempo para o Algoritmo 2, em GNU/Linux

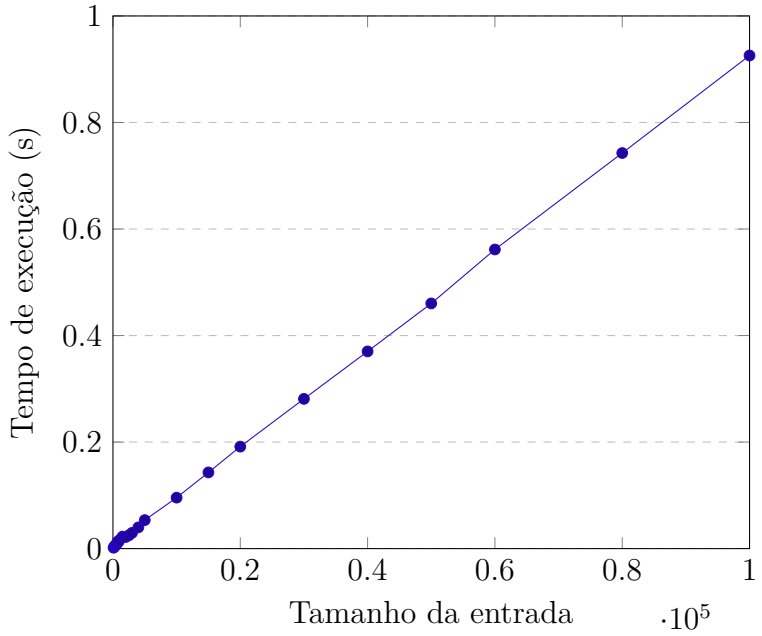
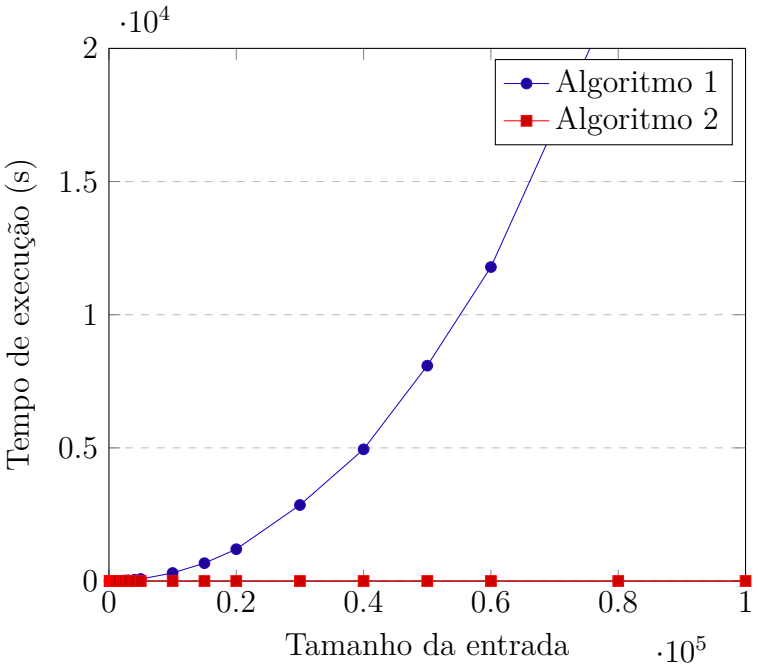


Gráfico 3 - Análises de tempo para os Algoritmos 1 e 2, em GNU/Linux



3.1.1.2 Análise de passos

Gráfico 4 - Análise de passos para o Algoritmo 1, em GNU/Linux

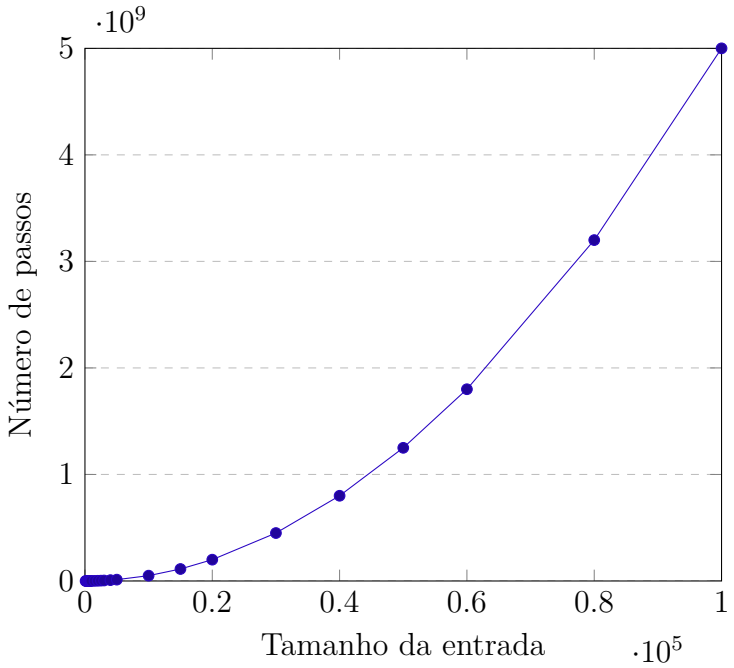


Gráfico 5 - Análise de passos para o Algoritmo 2, em GNU/Linux

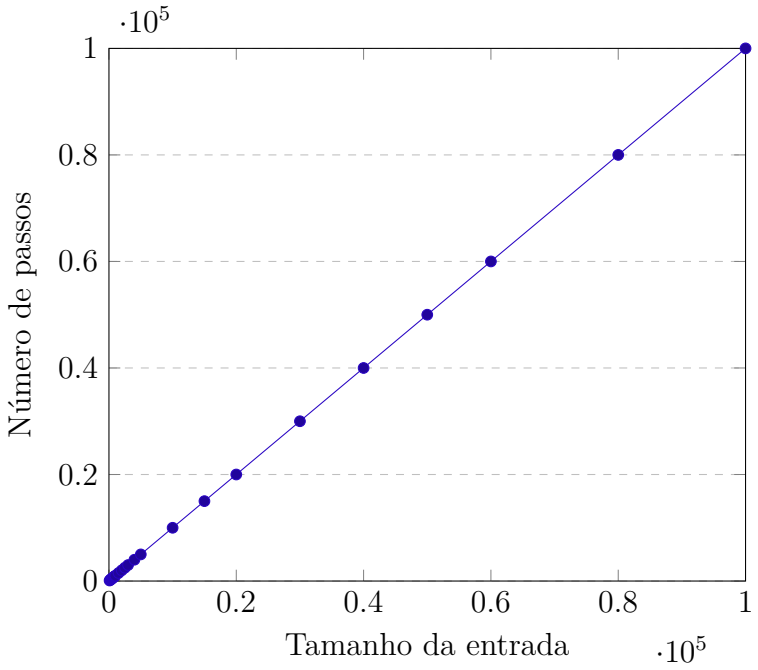
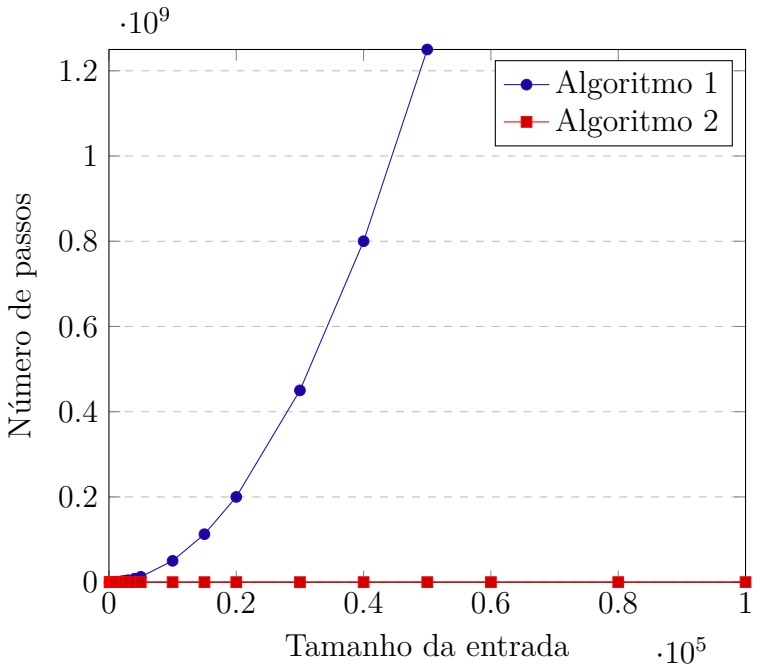


Gráfico 6 - Análises de passos para os Algoritmos 1 e 2, em GNU/Linux



3.1.2 Windows

A Tabela 4 abaixo apresenta os resultados dos testes realizados com os dois algoritmos deste problema executados no sistema operacional Windows.

Tamanho da entrada	Algoritmo 1		Algoritmo 2	
	Tempo (s)	Passos	Tempo (s)	Passos
100	0.03708	5050	0.00193	100
200	0.14578	20100	0.00342	200
300	0.28603	45150	0.00446	300
400	0.52439	80200	0.00539	400
500	0.78309	125250	0.00644	500
600	1.10563	180300	0.00746	600
700	1.52928	245350	0.00918	700
800	2.00395	320400	0.01016	800
900	2.51431	405450	0.01074	900
1000	3.12785	500500	0.01229	1000
1500	6.97031	1125750	0.02016	1500
2000	12.2985	2001000	0.02389	2000
2500	19.302	3126250	0.04071	2500
3000	28.0907	4501500	0.04609	3000
4000	49.0195	8002000	0.04986	4000
5000	76.5493	12502500	0.06375	5000
10000	338.828	50005000	0.11517	10000
15000	729.116	112507500	0.15228	15000
20000	2475.56	200010000	0.18993	20000
30000	5373.67	450015000	0.30345	30000
40000	9149.67	800020000	0.3839	40000
50000	14233.0	1250025000	0.47558	50000
60000	14946.8	1800030000	0.56444	60000
80000	25276.3	3200040000	0.76476	80000
100000	40097.9	5000050000	0.96624	100000

Tabela 4: Tempos e passos para os dois algoritmos do primeiro problema, em ambiente Windows.

3.1.2.1 Análise de tempo

Gráfico 7 - Análise de tempo para o Algoritmo 1, em Windows

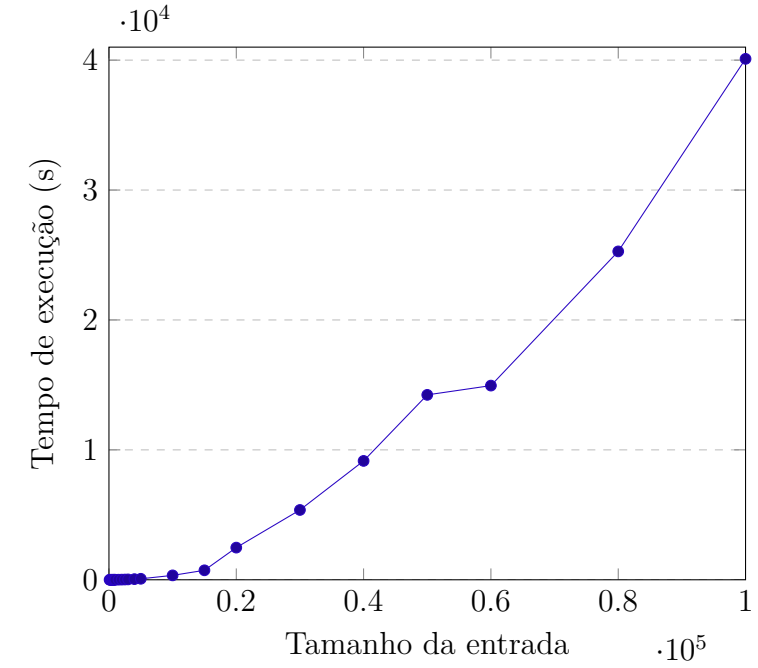


Gráfico 8 - Análise de tempo para o Algoritmo 2, em Windows

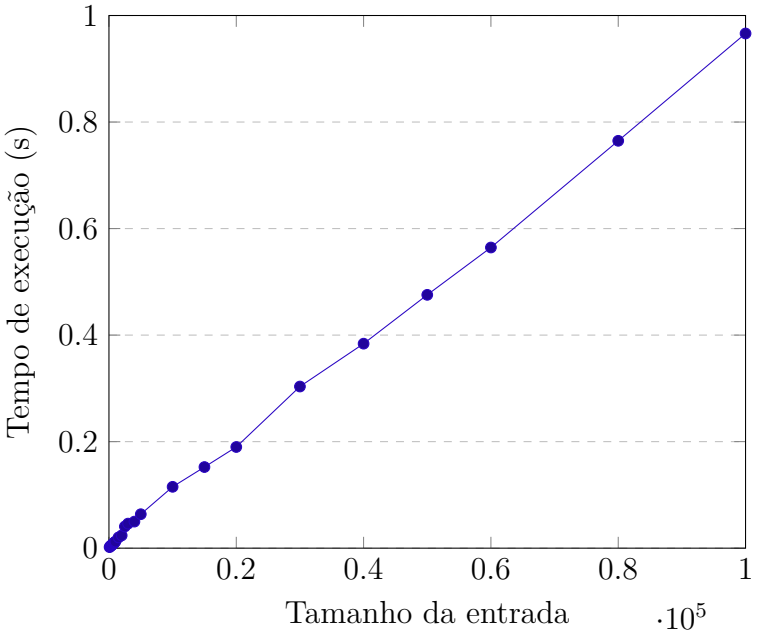
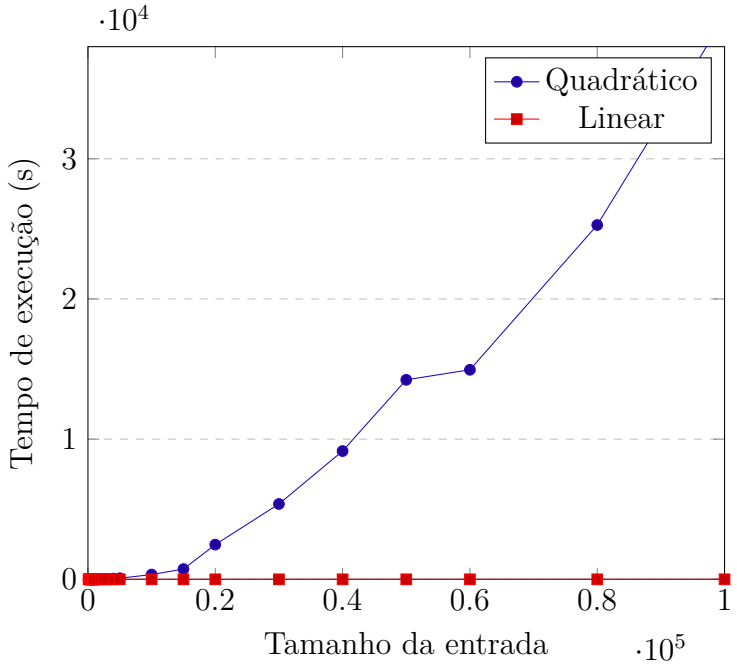


Gráfico 9 - Análise de tempo para os Algoritmos 1 e 2, em Windows



3.1.2.2 Análise de passos

Gráfico 10 - Análise de passos para o Algoritmo 1, em Windows

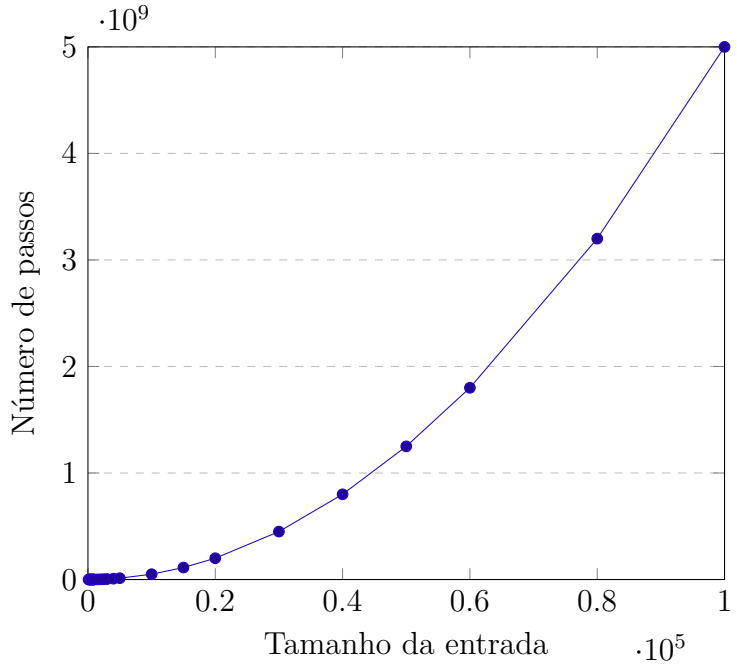


Gráfico 11 - Análise de passos para o Algoritmo 2, em Windows

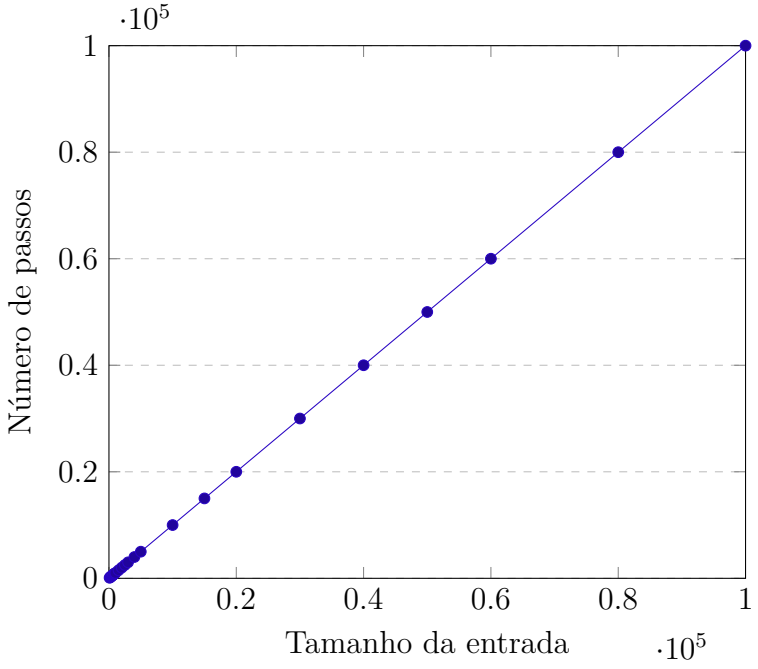
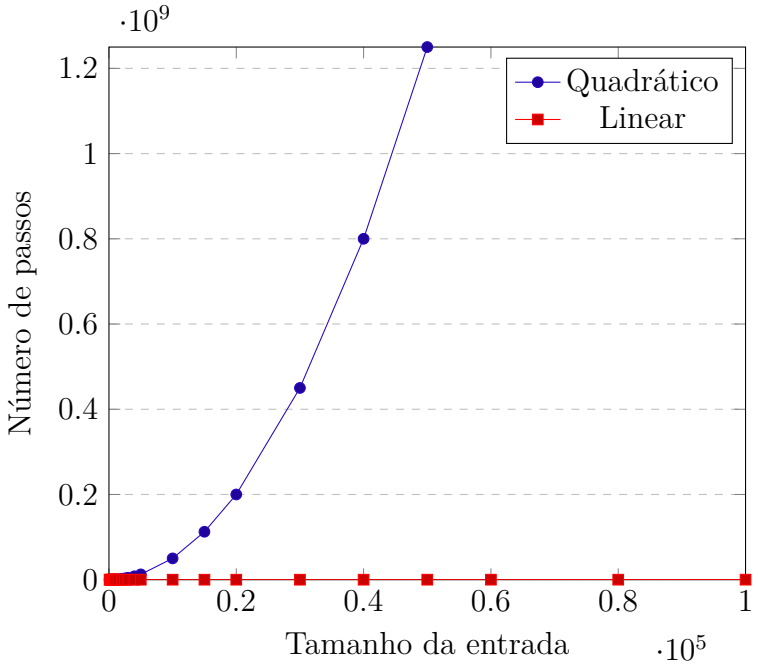


Gráfico 12 - Análise de passos para os Algoritmos 1 e 2, em Windows



3.2 Problema da compactação de um arranjo

3.2.1 GNU/Linux

A Tabela 5 abaixo apresenta os resultados dos testes de tempo e passos para as resoluções do segundo problema, executando em ambiente GNU/Linux.

Tamanho da entrada	Algoritmo 2		Algoritmo 4	
	Tempo (s)	Passos	Tempo (s)	Passos
100	0.03337	1159	0.001537	100
200	0.036195	5491	0.002921	200
300	0.048919	12055	0.004311	300
400	0.077406	20414	0.005911	400
500	0.119236	31194	0.00617	500
600	0.157536	45380	0.002666	600
700	0.220354	63075	0.002814	700
800	0.281709	82030	0.003376	800
900	0.352706	104146	0.003933	900
1000	0.422199	125498	0.004421	1000
1500	0.955652	279411	0.008547	1500
2000	1.7272	506788	0.012675	2000
2500	2.65735	781179	0.015354	2500
3000	3.80134	1128593	0.019504	3000
4000	6.77431	2008197	0.029548	4000
5000	10.4267	3102561	0.037854	5000
10000	42.3132	12469972	0.081724	10000
15000	99.4856	28078596	0.125604	15000
20000	170.314	49890900	0.169299	20000
30000	382.53	112821543	0.254883	30000
40000	705.335	199585412	0.339884	40000
50000	1034.21	311618350	0.424333	50000
60000	1488.08	448232072	0.508079	60000
80000	2654.71	800603518	0.677554	80000
100000	4146.1	1250123357	0.851603	100000

Tabela 5: Tempos e passos para os dois algoritmos do segundo problema, em GNU/Linux.

3.2.1.1 Análise de tempo

Gráfico 13 - Análise de tempo para o Algoritmo 3, em GNU/Linux

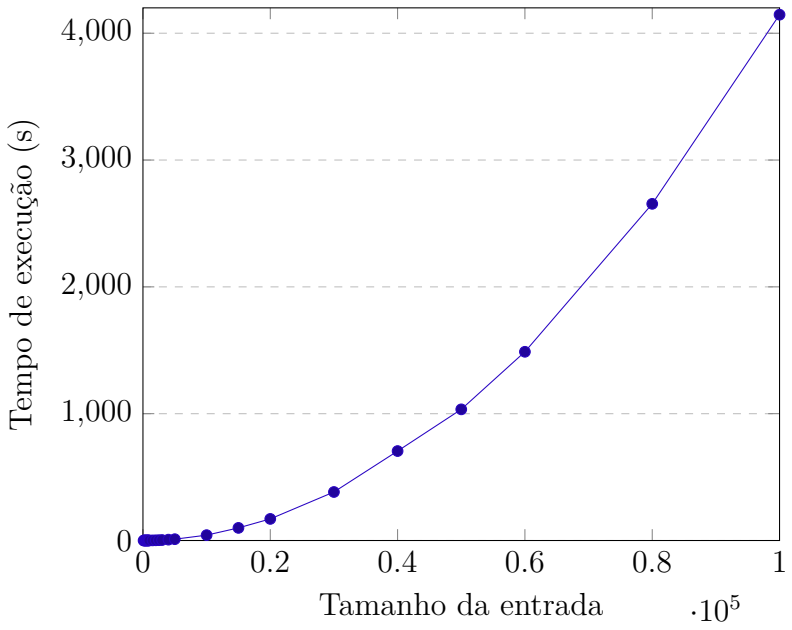


Gráfico 14 - Análise de tempo para o Algoritmo 4, em GNU/Linux

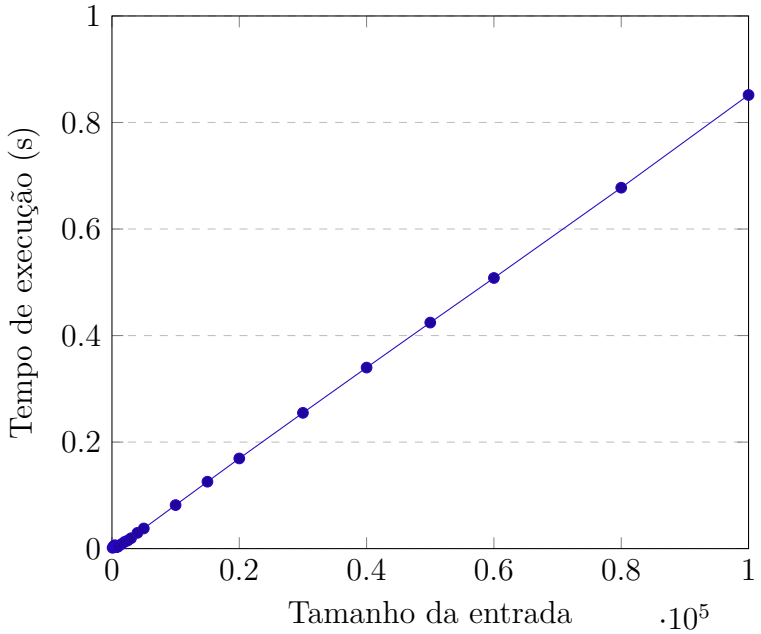
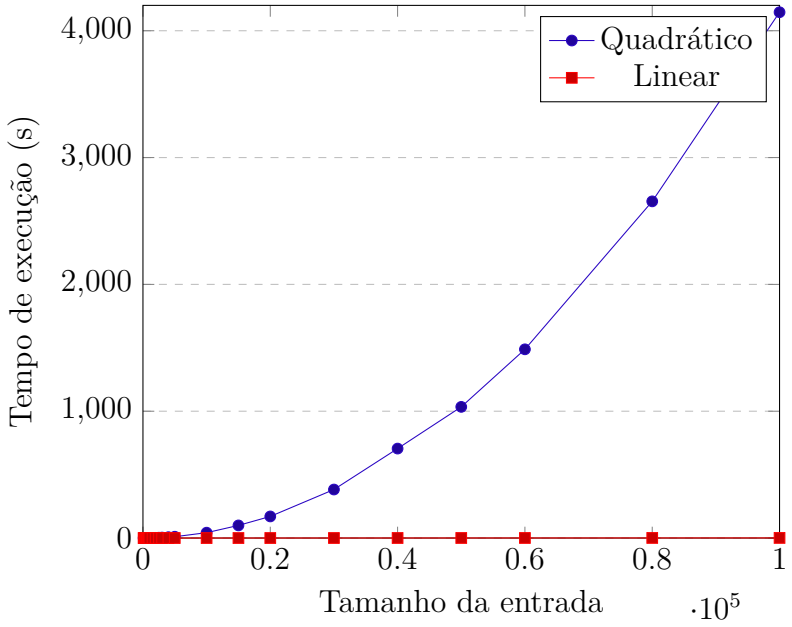


Gráfico 15 - Análise de tempo para os Algoritmos 3 e 4, em GNU/Linux



3.2.1.2 Análise de passos

Gráfico 16 - Análise de passos para o Algoritmo 3, em GNU/Linux

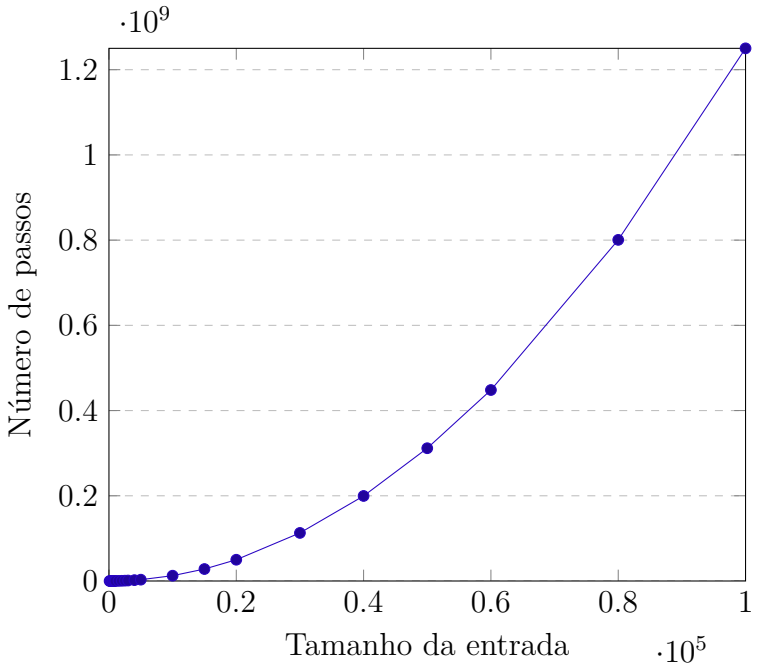


Gráfico 17 - Análise de passos para o Algoritmo 4, em GNU/Linux

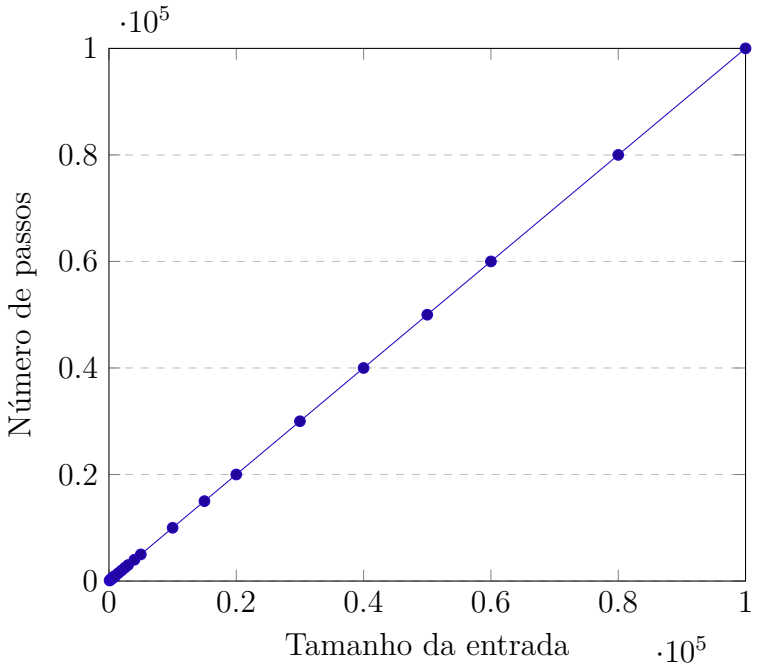
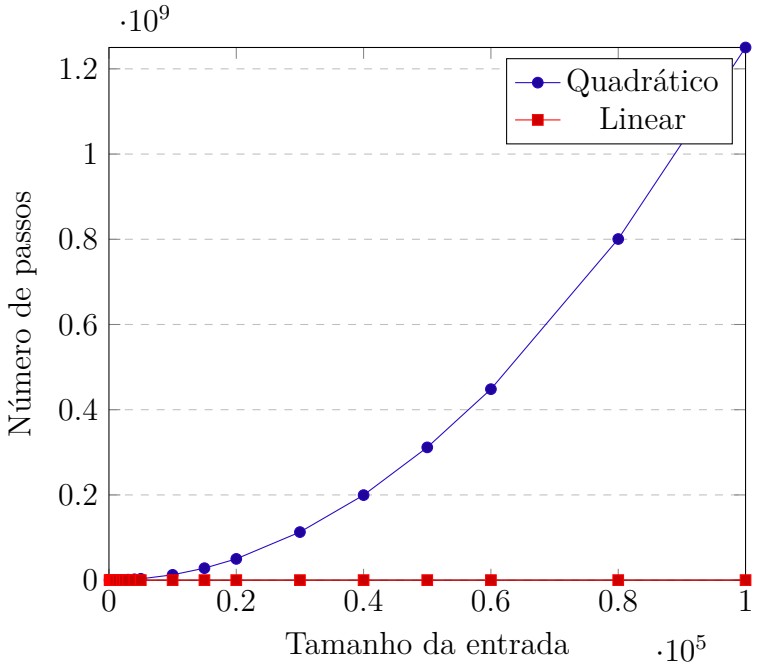


Gráfico 18 - Análise de passos para os Algoritmos 3 e 4, em GNU/Linux



3.2.2 Windows

A Tabela 6 abaixo apresenta os resultados dos testes realizados com os dois algoritmos do segundo problema, executados no sistema operacional Windows.

Tamanho da entrada	Algoritmo 3		Algoritmo 4	
	Tempo (s)	Passos	Tempo (s)	Passos
100	0.03337	1159	0.00101	100
200	0.036195	5491	0.0019	200
300	0.048919	12055	0.00166	300
400	0.077406	20414	0.00311	400
500	0.119236	31194	0.00281	500
600	0.157536	45380	0.00383	600
700	0.220354	63075	0.00436	700
800	0.281709	82030	0.00525	800
900	0.352706	104146	0.00622	900
1000	0.422199	125498	0.00683	1000
1500	0.955652	279411	0.01142	1500
2000	1.7272	506788	0.01496	2000
2500	2.65735	781179	0.01918	2500
3000	3.80134	1128593	0.02325	3000
4000	6.77431	2008197	0.03172	4000
5000	10.4267	3102561	0.04168	5000
10000	42.3132	12469972	0.0834	10000
15000	99.4856	28078596	0.12984	15000
20000	170.314	49890900	0.17041	20000
30000	382.53	112821543	0.25278	30000
40000	705.335	199585412	0.34059	40000
50000	1034.21	311618350	0.42183	50000
60000	1488.08	448232072	0.50744	60000
80000	2654.71	800603518	0.68913	80000
100000	4146.1	1250123357	0.86356	100000

Tabela 6: Tempos e passos para os dois algoritmos do segundo problema, em Windows.

3.2.2.1 Análise de tempo

Gráfico 19 - Análise de tempo para o Algoritmo 3, em Windows

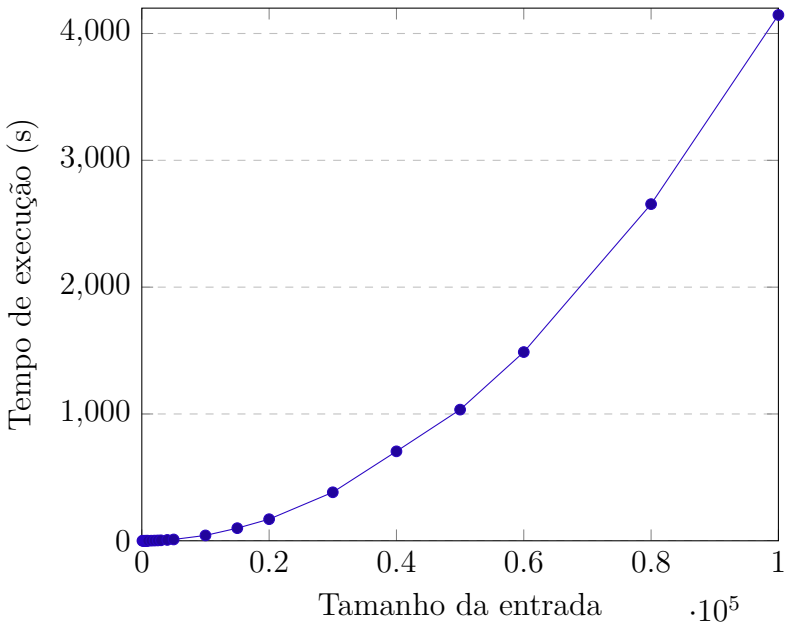


Gráfico 20 - Análise de tempo para o Algoritmo 4, em Windows

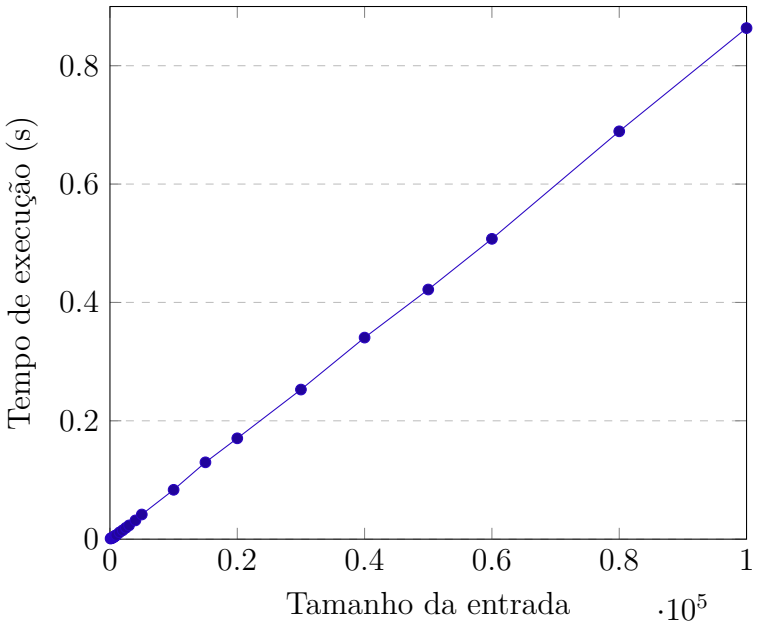
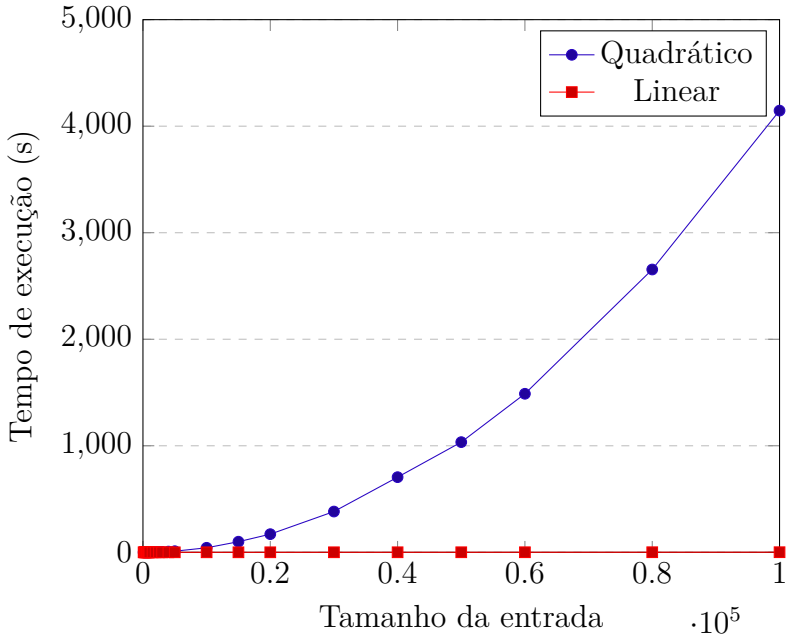


Gráfico 21 - Análise de tempo para os Algoritmos 3 e 4, em Windows



3.2.2.2 Análise de passos

Gráfico 22 - Análise de passos para o Algoritmo 3, em Windows

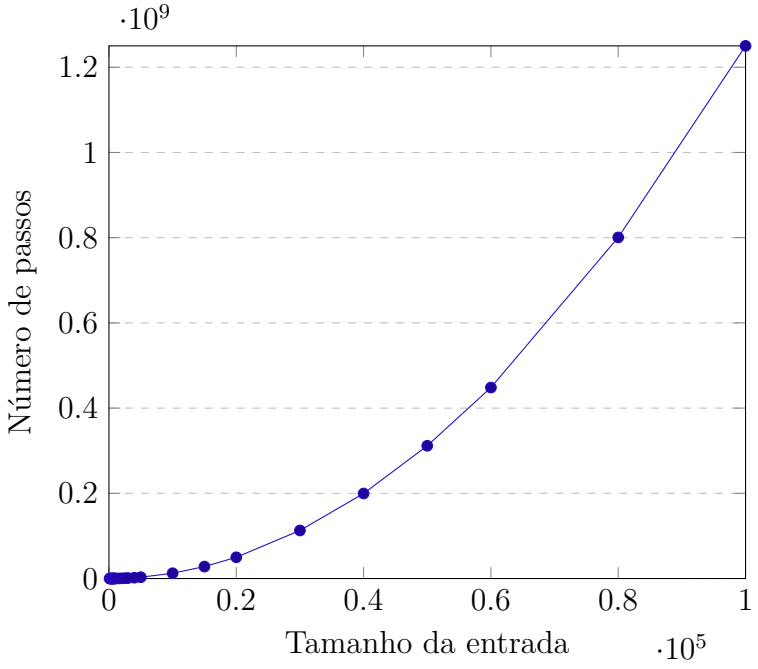


Gráfico 23 - Análise de passos para o Algoritmo 4, em Windows

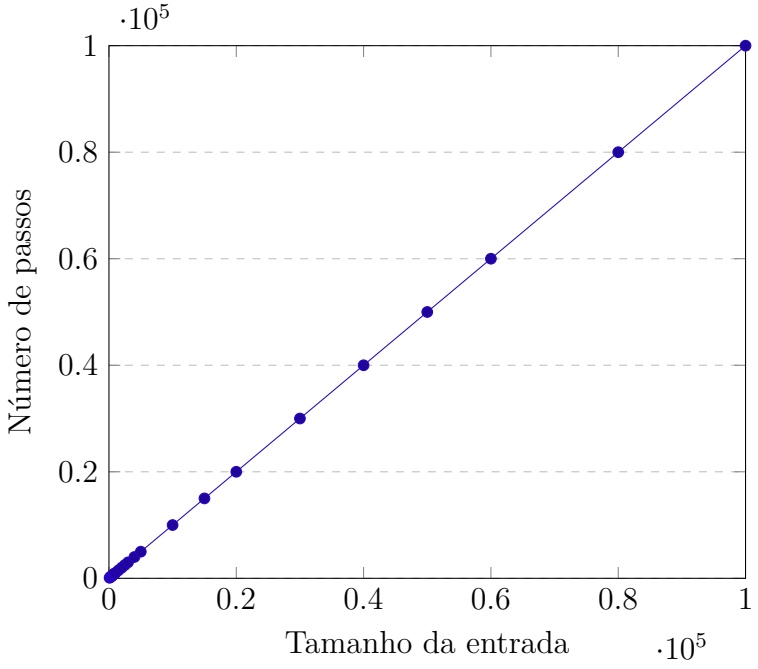
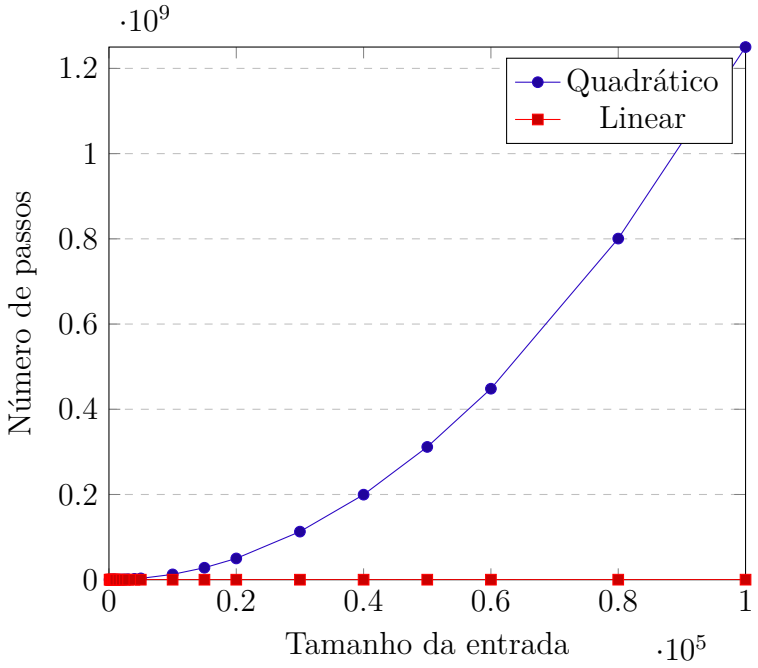


Gráfico 24 - Análise de passos para os Algoritmos 3 e 4, em Windows

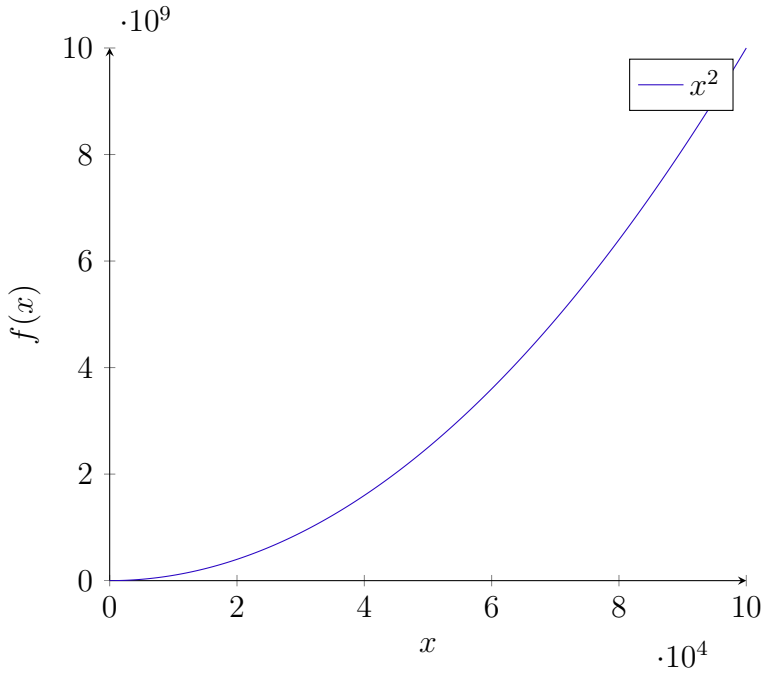


4 Discussão

4.1 Análise dos gráficos

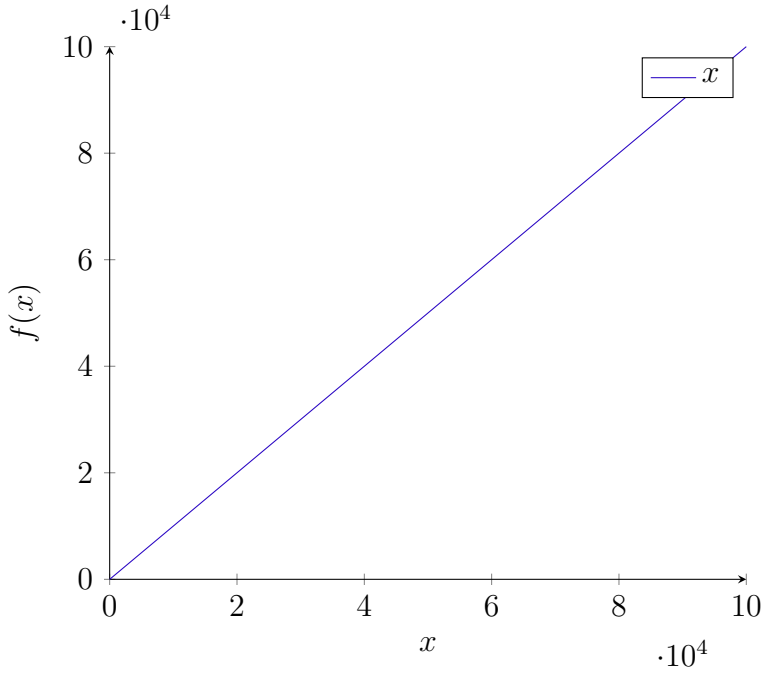
Para ambos os sistemas operacionais, tanto na análise de tempo, quanto na de passos, os Algoritmos 1 e 3 geraram gráficos (Gráficos 1, 4, 7, 10, 13, 16, 19 e 22) cujo comportamento se assemelha ao da função quadrática $f(x) = ax^2 + bx + c, a \neq 0$, exemplificada no Gráfico 25. Tal complexidade quadrática pode ser facilmente explicada observando-se os pseudocódigos, pela presença de dois laços aninhados dependentes do tamanho dos dados de entrada.

Gráfico 25 - Exemplo de função quadrática



Já os Algoritmos 2 e 4 geraram gráficos (Gráficos 2, 5, 8, 11, 14, 17, 20 e 23) cujo comportamento obedece ao da função linear $f(x) = ax, a \neq 0$, como exemplificado no Gráfico 26. Neles, existe apenas um laço dependende do tamanho da entrada, característica geral de um algoritmo linear.

Gráfico 26 - Exemplo de função linear



4.1.1 Comportamento anômalo no ambiente Windows

Ao observar os gráficos de tempo (Gráficos 7, 8 e 9) para a execução, em Windows, dos algoritmos do primeiro problema, é possível perceber um desvio em relação aos Gráficos 1, 2 e 3, que também representam o tempo de execução dos mesmos algoritmos, porém no ambiente Linux e em maior alinhamento com o comportamento quadrático.

A possível causa desse efeito foi a instabilidade de consumo de memória e processamento de alguns processos nativos do próprio sistema. Durante os procedimentos de execução, foram constatados alguns processos que oscilaram entre 10% e 30% no consumo de memória, principalmente no momento de testes com valores mais altos. Outro ponto a se levantar está no fato de o Windows 10 ainda estar na sua fase inicial de uso, o que pode levantar a suspeita da existência de algumas falhas de processamento de dados, podendo ter influenciado indiretamente na realização dos testes.

4.2 Correspondência com as análises matemáticas

As análises matemáticas abaixo usam as constantes marcadas nos comentários dos algoritmos descritos na Seção 2.1.3. Cada comentário liga uma constante à operação da linha seguinte. Com elas, pode-se corroborar a relação entre os gráficos dos testes empíricos do Capítulo 3 e os das funções quadrática e linear.

4.2.1 Algoritmos 1 e 3 - quadráticos

Para o Algoritmo 1, não se identificam melhor e pior casos, pois nenhuma condição explícita previne a execução dos dois laços. Por isso, a seguinte análise matemática é suficiente para demonstrar o comportamento quadrático desse algoritmo:

$$T(N) = c1 + (c2 + c3)N + (c5 + c6)\sum_{i=1}^{N} i + c7 \tag{4.2.1}$$

$$= c1 + (c2 + c3)N + (c5 + c6)\frac{N(N-1)}{2} + c7 \tag{4.2.2}$$

$$= \Theta(N^2) \tag{4.2.3}$$

Para o Algoritmo 3, são identificados o pior e o melhor caso. O pior caso - aquele em que a primeira metade do vetor contém somente elementos negativos - pode ser matematicamente analisado como segue:

$$T(N) = c1 + (c2 + c3 + c6)N + \frac{N}{2}\{\frac{N}{2}(c4 + c5)\} + c7 \tag{4.2.4}$$

$$= \mathcal{O}(N^2) \tag{4.2.5}$$

No melhor caso, o segundo laço do algoritmo não é executado, e a análise matemática se resume a:

$$T(N) = c1 + (c2 + c3)N + c7 \tag{4.2.6}$$

$$= \Omega(N) \tag{4.2.7}$$

Estão justificados matematicamente, portanto, os comportamentos dos gráficos das análises empíricas para esses dois algoritmos.

4.2.2 Algoritmos 2 e 4 - lineares

Para o Algoritmo 2, é suficiente a seguinte análise para mostrar a sua complexidade linear, visto que não há melhor ou pior caso:

$$T(N) = c1 + c2 + (c3 + c4 + c5)N + c6 \tag{4.2.8}$$

$$= \Theta(N) \tag{4.2.9}$$

O Algoritmo 4, por sua vez, tem seu pior caso assim analisado:

$$T(N) = c1 + (c2 + c3 + c4 + c5)N + c6 \tag{4.2.10}$$

$$= \mathcal{O}(N) \tag{4.2.11}$$

Para o melhor caso:

$$T(N) = c1 + (c2 + c3)N + c6 \tag{4.2.12}$$

$$= \Omega(N) \tag{4.2.13}$$

Dessa maneira, pode-se generalizar que $T(N) = \Theta(N)$.

Novamente, as análises matemáticas estão coerentes com as análises empíricas realizadas.

4.3 Comparando os algoritmos

Tendo em vista as análises empíricas de tempo e passos, bem como as análises matemáticas, é evidente que as soluções mais eficientes são os Algoritmos 2 e 4. No pior caso - geralmente o mais relevante -, apresentam complexidade linear, contrastando com o crescimento quadrático dos Algoritmos 1 e 3, que leva estes a um consumo insustentável de recursos do computador conforme o tamanho da entrada de dados cresce. A comparação se torna ainda mais clara nos Gráficos 3, 6, 9, 12, 15, 18, 21 e 24.

Essa constatação independe do ambiente de execução dos testes, dado que é da natureza do projeto desses algoritmos apresentarem esses comportamentos, como foi matematicamente demonstrado.

Referências

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de Dados e seus Algoritmos. Rio de Janeiro: LTC, 1994. 320p.

Apêndices

APÊNDICE A – Implementação dos algoritmos em C++

A.1 Problema da soma máxima de uma subsequência

A.1.1 Quadrático

```
int maxSumQuadratic(int *v, int tam){
    int geralSum = 0;
    int localSum = 0;
    for(int i = 0; i < tam; ++i){
        localSum = 0;
        for(int j = i; j < tam; ++j){
            localSum += v[j];
            geralSum = std::max(geralSum, localSum);
        }

    }
    return geralSum;
}
```

A.1.2 Linear

```
int maxSumLinear(int *v, int tam){
    int geralSum = 0;
    int localSum = 0;
    for(int i = 0; i < tam; ++i){
       localSum = std::max(localSum + v[i], 0);
       geralSum = std::max(geralSum, localSum);
    }
    return geralSum;
}
```

A.2 Problema da compactação de um arranjo

A.2.1 Quadrático

```
void compactArrayQuadratic(int *v, int *tam) {
    for (int i = (*tam) - 1; i >= 0; --i) {
        if (v[i] <= 0) {
            for (int j = i; j < (*tam) - 1; ++j){
                v[j] = v[j + 1];
            }
            --(*tam);
        }
    }
}
```

A.2.2 Linear

```
void compactArrayLinear(int *v, int *tam) {
    int k = 0;
    for (int i = 0; i < (*tam); ++i)
        if (v[i] > 0) v[k++] = v[i];
    *tam = k;
}
```