

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MATEUS COUTINHO MARIM

CONTADOR DE FREQUÊNCIA DE PALAVRAS

TRABALHO DE ESTRUTURAS DE DADOS II

JUIZ DE FORA
2016

MATEUS COUTINHO MARIM

CONTADOR DE FREQUÊNCIA DE PALAVRAS

Trabalho da matéria Estruturas de
Dados II do curso de Ciência da
Computação da Universidade Federal
de Juiz de Fora.

Professor: Jairo Francisco de Souza

JUIZ DE FORA

2016

1. INTRODUÇÃO

Contar a frequência em textos muitas vezes se apresenta como um problema banal que pode ser facilmente resolvido, mas se não forem utilizados bons métodos para a resolução deste problema os custos para se chegar no resultado podem ser inviáveis, sendo estes custos de desempenho e gastos de memória.

O objetivo deste trabalho é encontrar uma estratégia eficiente para se contar a frequência de palavras em textos planos sem formatação combinando um bom desempenho com o menor gasto de memória possível.

2. TAD IMPLEMENTADO

O TAD escolhido para armazenar as palavras e contar as frequência foi uma árvore vermelho e preta, pois era necessário escolher uma estrutura de dados com uma maior flexibilidade na inserção para que se caísse menos vezes no pior caso da estrutura de $\Omega(2 * \log_2(N))$ e no final tivéssemos o desempenho das árvores binárias de busca, na ordem de $O(\log_2(N))$.

Como a árvore vermelho e preta na sua implementação mais simples apenas permite a contagem de frequência se fossem permitidas chaves duplicadas, foi necessário pensar em um jeito melhor para contar as frequências sem que precisasse percorrer toda a árvore para se obter o valor final das frequências.

Para resolver este problema a estrutura foi aumentada adicionando um contador nos nós da árvore, para que, toda vez que na inserção de árvore binária se chegasse em um nó com a mesma chave do nó que está sendo inserido, ao invés de se adicionar um nó duplicado, o contador do nó de chave igual iria ser incrementado e a inserção seria terminada, evitando a adição de nós duplicados.

2.1 REPRESENTAÇÃO DO TAD

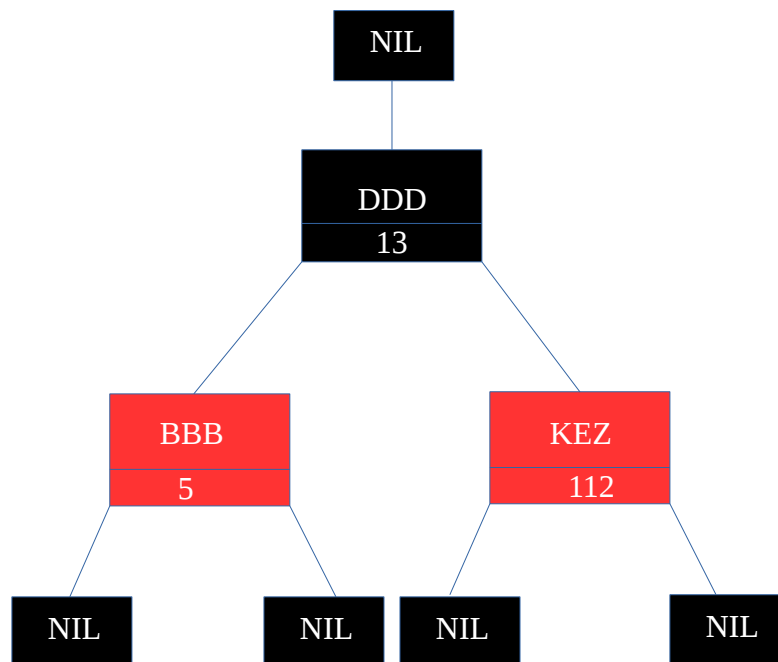


Figura 1. Representação da estrutura utilizada.

2.2 ANÁLISE DA COMPLEXIDADE NA INSERÇÃO

Como não existem nós duplicados na árvore podemos dizer que seu tamanho vai ser a quantidade de palavras únicas existentes nos textos processados ao qual vamos chamar de P , como vão ser feitas N inserções, o pior caso da inserção das chaves na estrutura vai passar a ser $\Omega(2 * N * \log_2(P))$.

2.3 ORDENAÇÃO LEXICOGRÁFICA

O próximo passo para resolver o problema e apresentá-lo como pedido foi fazer uma ordenação que deixasse as chaves em ordem alfabética, para isso nos aproveitamos das propriedades das árvores binárias de busca e executamos uma transversal In-order na árvore que de acordo com as propriedades vai imprimir os nós da árvore ordenados, como estamos lidando com strings, elas vão sair em ordem alfabética e vão ser armazenadas em uma lista.

Este procedimentos citados acima são utilizados pelo algoritmo treeSort representado na figura 2, onde a bolinha vermelha marca o início da caminhada e a verde marca o fim.

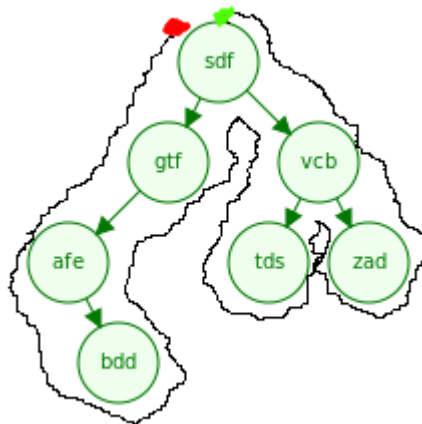


Figura 2. Representação do treeSort com início na bola vermelha e fim na verde, o resultado nessa árvore vai ser {bdd, afe, gtf, sdf, vcb, tds, zad}.

2.3.1 ANÁLISE DA COMPLEXIDADE DO TREESORT

O algoritmo treeSort tem como complexidade no pior caso para ordenar uma árvore binária balanceada $\Omega(N \cdot \log_2(N))$ supondo que fosse necessário construir a árvore com N nós, como já temos a árvore binária construída e balanceada e com tamanho P , então o algoritmo vai ter como complexidade no pior caso apenas $\Omega(P)$.

2.4 ORDENAÇÃO POR FREQUÊNCIA DAS CHAVES

Chegamos na parte em que temos que fazer com que a saída mostre as chaves ordenadas pela maior frequência, as que tiverem as mesmas frequências vão ser mostradas em ordem alfabética.

Para isso precisamos utilizar um algoritmo que depois de ordenar os nós pela frequência das chaves as mantenha na ordem que apareceram antes de serem ordenadas, esse tipo de algoritmo de ordenação é chamado de ordenação estável.

O algoritmo que iremos escolher é o mergeSort que satisfaz os requisitos de ser estável e ter uma complexidade aceitável para a nossa necessidade.

Exemplo da execução do mergeSort:

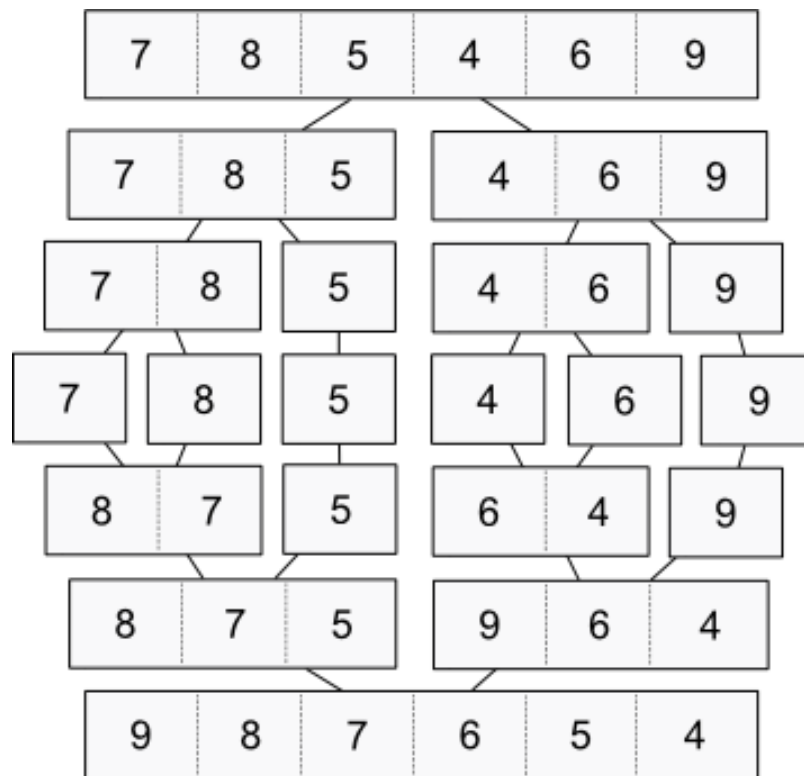


Figura 3: Árvore de execução do algoritmo mergeSort.

2.4.1 ANÁLISE DA COMPLEXIDADE DO MERGESORT

O algoritmo mergeSort tem como complexidade de ordenação no pior caso $\Omega(N \cdot \log_2(N))$ pois ele vai quebrando o vetor inicial em dois a cada chamada recursiva, gerando uma árvore binária de recursividade de altura $\log_2(N)$ e essa divisão é feita N vezes como mostrado na figura 3, como a nossa lista a ser ordenada tem tamanho P a complexidade no pior caso vai ser $\Omega(P \cdot \log_2(P))$.

2.5 ANÁLISE DO DESEMPENHO DA SOLUÇÃO

Seguindo os passos anteriores o problema do Contador de Frequência de Palavras é resolvido, mas ainda precisamos fazer a análise do desempenho da solução para verificar se os métodos utilizados são viáveis e satisfazem o requisito de ter um bom desempenho na execução.

Para fazermos esta análise primeiro vai ser feita a análise assintótica da solução e logo depois vão ser feitos experimentos suficientes para verificar se a curva gerada pelos experimentos batem com a ordem de complexidade achada para os procedimentos, caso sejam curvas equivalentes, pode-se dizer que a solução tem o desempenho esperado.

Primeiro é feita uma função $T(N, P)$ que descreve o comportamento assintótico da solução com N palavras e P palavras únicas:

$$T(N, P) = 2 * N * \log_2(P) + P + P * \log_2(P)$$

A fórmula acima é definida como a soma das fórmulas que representam a complexidade dos métodos utilizados na solução e com ela podemos isolar o termo de maior grau para termos a complexidade da solução:

$$T(N, P) = 2 * N * \log_2(P)$$

$$\Omega(N * \log_2(P))$$

Portanto, temos que a complexidade da solução no pior caso é $\Omega(N * \log_2(P))$.

2.5.1 ANÁLISE EXPERIMENTAL DO DESEMPENHO

Foram realizados 32 experimentos para gerar a curva representando o tempo de execução em função de N entradas, segue abaixo uma tabela com os resultados:

N	Palavras Únicas (P)	T(N)
0	0	0,000002
91738	9348	0,067151
110495	10798	0,080619
242983	15565	0,170938
260420	16775	0,183122
377324	21083	0,266776
725332	35961	0,531803
770447	37590	0,570903
787175	37753	0,579924
793748	37899	0,585812
878020	40105	0,650214
976491	42377	0,722959
993927	42497	0,735845
1095492	45126	0,810684
1140438	45758	0,842988
1157679	46352	0,859416
1181997	46852	0,872767
1339341	48793	0,986674
1397511	49453	1,02584
1587439	53964	1,16527
1605689	54266	1,17541
1696633	55218	1,24616
1713175	55469	1,26094
1849509	58505	1,3815
1874471	59392	1,38047
1943438	60837	1,4335
2035973	62739	1,50348
2148001	64251	1,58788
2262581	65391	1,66641
2381602	66816	1,75142
2462155	67909	1,81574
2572047	71438	1,91115
2634474	73296	1,96474

Tabela 1: Resultados dos experimentos.

E abaixo está o gráfico que representa os resultados acima:

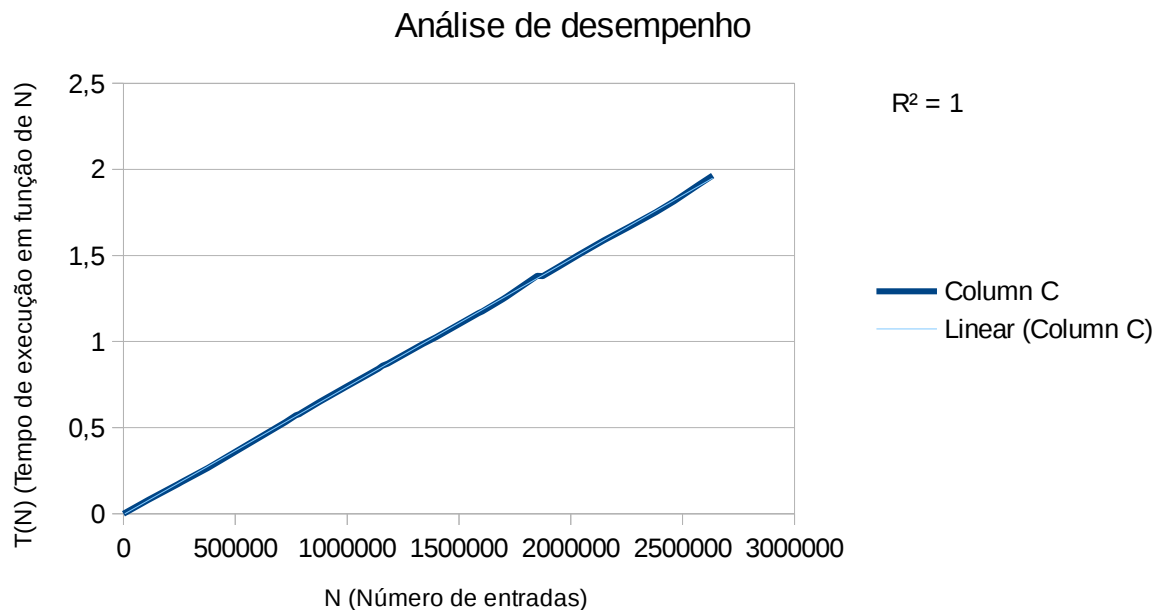


Gráfico 1: Curva representando os resultados dos experimentos.

Como podem ver o gráfico gerou uma curva linear, e para confirmar basta ver que o valor do coeficiente de determinação R^2 é muito próximo a 1 (o valor foi arredondado, mas $R^2=0,9998750807$), isso quer dizer que a equação linear gerada pelo gráfico consegue prever quase 100% dos resultados.

Esse comportamento linear consegue ser explicado pelo fato de que o valor da variável P cresce muito pouco conforme o valor de N cresce, isso se explica pelo “Princípio de Pareto”, ou mais comumente, a lei “os 80/20”, que é uma relação que descreve casualidades e resultados. Este princípio alega que aproximadamente 80% da saída é um resultado direto de 20% da entrada.

O “Princípio de Pareto” é utilizado para explicar o comportamento da curva pois ele é facilmente observado na frequência do uso de palavras nas linguagens de todo o mundo, incluindo a língua inglesa que é a língua das palavras da entrada do programa, portanto ele se aplica ao problema.

Portanto, quanto maior a entrada, maior a diferença entre N e P , fazendo com que na maior parte dos casos da inserção apenas uma pesquisa seja

feita e o contador do nó seja incrementado, fazendo com que a curva tenda a ser linear, se considerarmos que P com uma entrada muito grande se torne constante, então a complexidade no pior caso para grandes quantidades de dados vai ser de $\Omega(N)$.

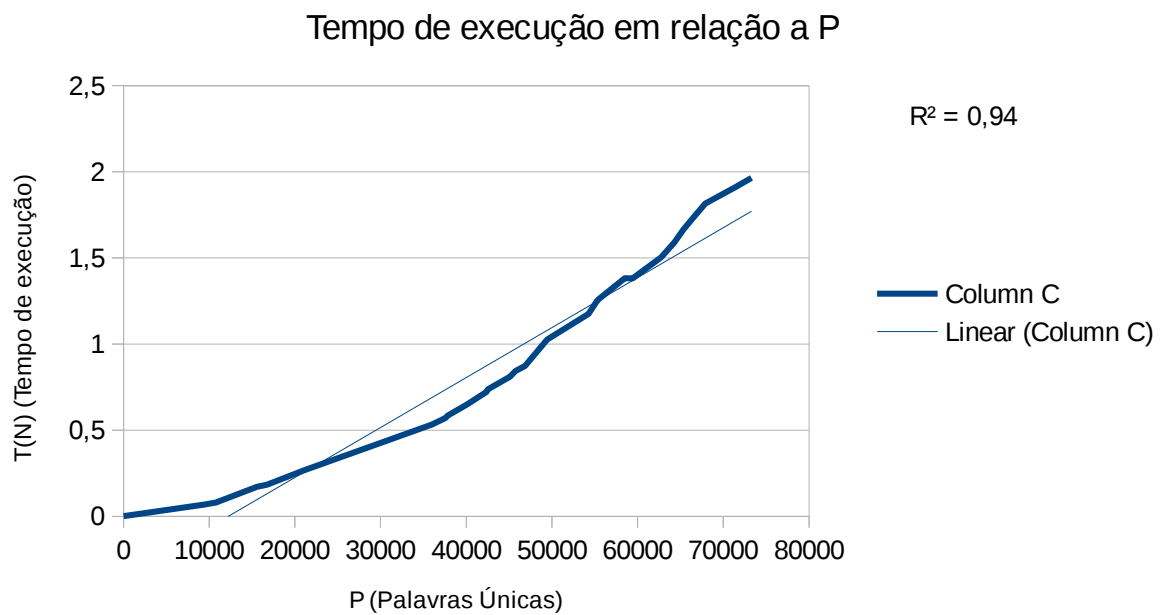


Gráfico 2: Tempo de execução em relação a quantidade de palavras únicas.

2.5.2 ANÁLISE DO GASTO DE MEMÓRIA

Para analisar o gasto de memória é necessário somar a complexidade da memória usada em cada passo da solução, a inserção na árvore vermelho e preto tem complexidade de memória no pior caso de $O(P)$, a ordenação treeSort $O(P)$ o mergeSort tem $O(P)$ mais $O(P)$ auxiliar durante a execução, a complexidade do gasto de memória vai ser $G(P)=4*P$ no pior caso, logo a complexidade de gasto de memória vai ser $O(P)$.

Como o gasto de memória vai depender principalmente do crescimento da quantidade de palavras únicas P , o crescimento da curva do gasto de memória vai ser proporcional ao crescimento de P em relação a N , como mostrado no Gráfico 3 e comprovado na análise experimental.

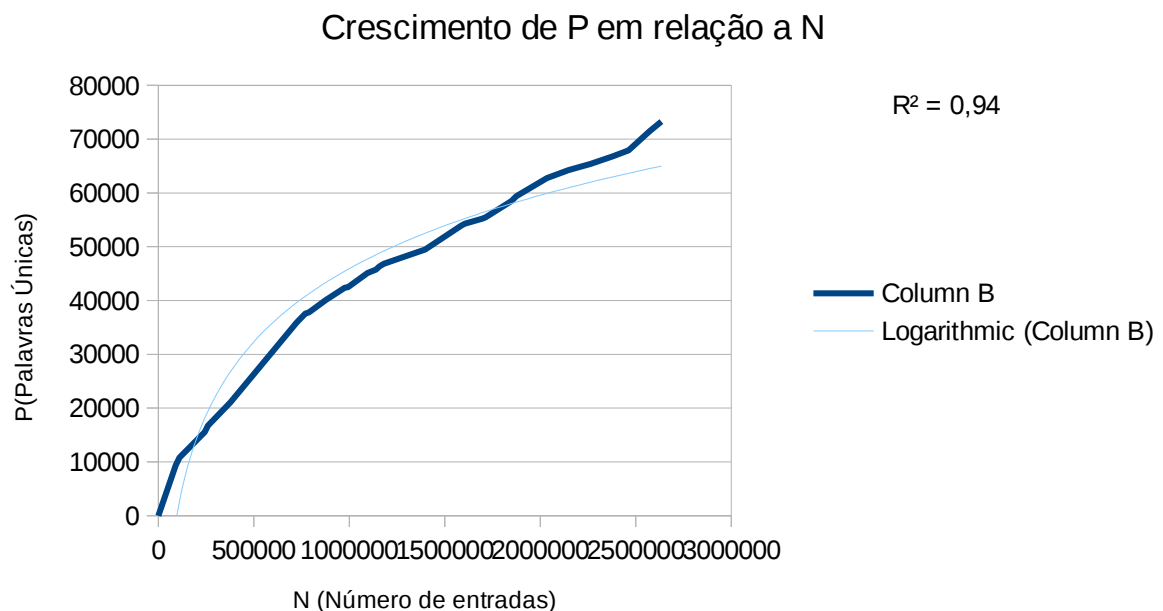


Gráfico 3: Crescimento de P em relação a N .

2.5.3 ANÁLISE EXPERIMENTAL DO GASTO DE MEMÓRIA

Foram realizados 32 experimentos para gerar a curva representando o gasto de memória do programa, segue abaixo uma tabela com os resultados:

N	P	Memória(Kb)
0	0	13312
62427	8000	14016
172319	16047	14712
217434	18409	14936
353768	23875	15416
472789	26556	15592
530959	28287	15748
720887	35604	16524
832915	38247	16696
839488	38461	16704
996832	40630	17000
1014268	40736	17000
1032518	41213	17012
1051275	41667	17016
1067817	42024	17156
1092135	42550	17160
1137081	43216	17172
1154518	43841	17312
1171759	44476	17320
1286339	45951	17476
1418827	47501	17636
1499380	49060	17788
1597851	50608	17944
1699416	52592	18108
1783688	54211	18268
1900592	56280	18428
2248600	66924	19520
2339544	67762	19540
2364506	68541	19680
2381234	68624	19680
2473769	70464	19840
2542736	71797	19864
2634474	73296	20016

E abaixo está o gráfico que representa os resultados:

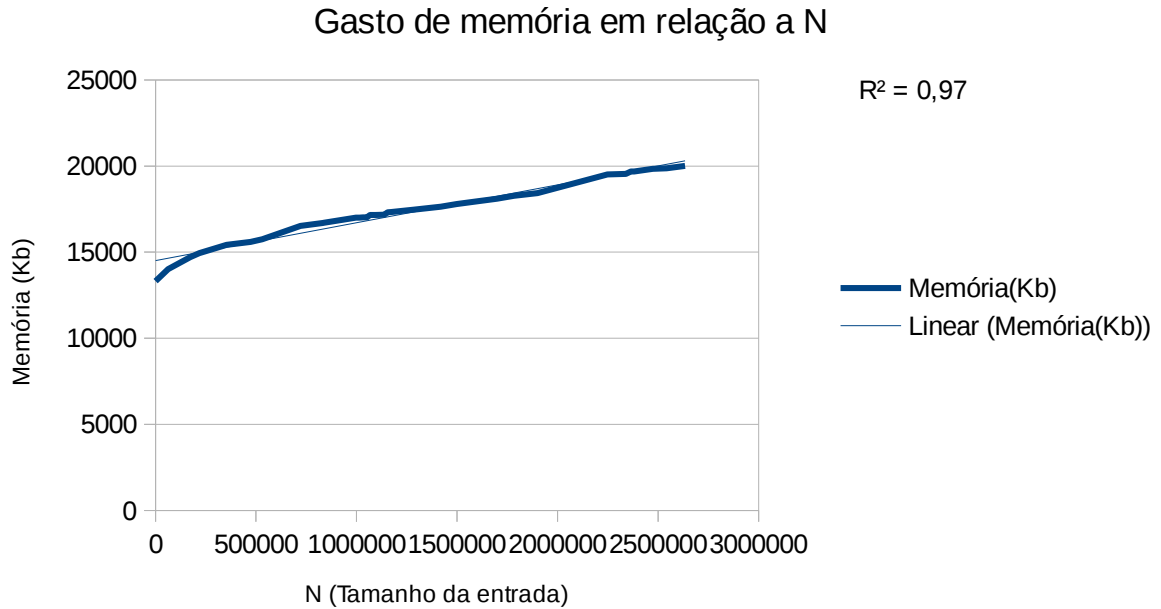


Gráfico 4: Gasto de memória em relação a N

Como é possível observar no gráfico, o gasto de memória tem uma taxa de crescimento maior com uma entrada menor, isso também se deve pelo Princípio de Pareto, pois após uma certa quantidade de inserções as palavras que aparecem mais frequentemente na língua inglesa se repetem mais, fazendo com que o gasto de memória aumente mais por causa dos incrementos de contadores e por causa de inserções de palavras mais específicas do texto.

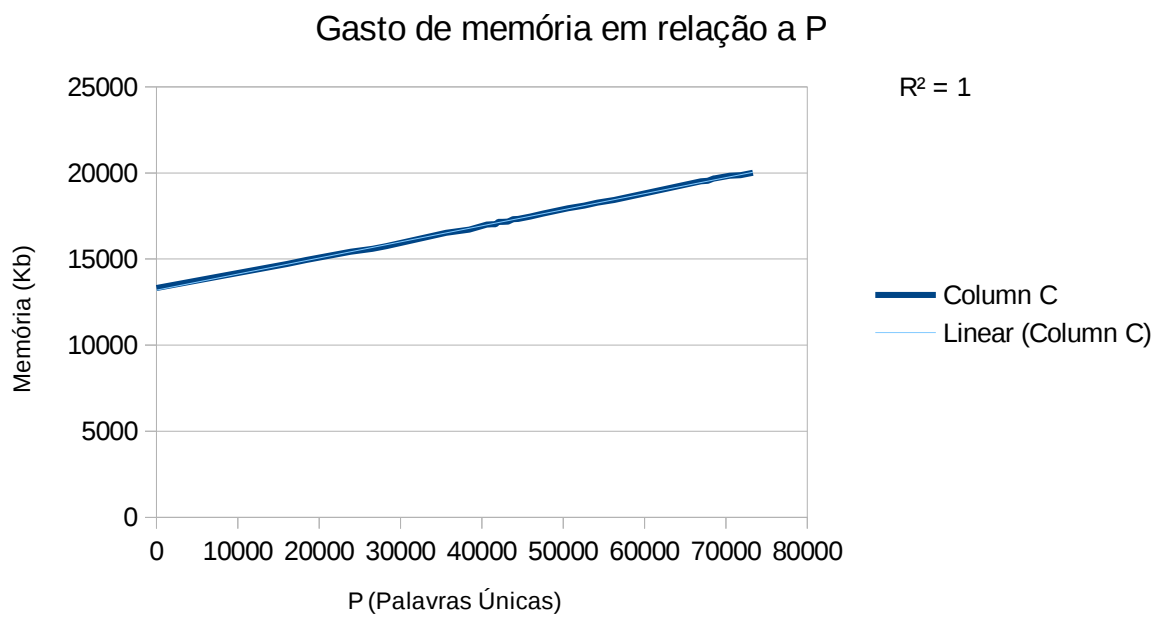


Gráfico 5: Gasto de memória em relação a P.

A curva gerada no Gráfico 5 prova a complexidade de memória de $\Omega(P)$.