Análise Temporal - Projeto 3

Docente: João Dias

Discente: Diogo Fonseca nº79858

Comparação do Recursive Sort com o Quicksort

n	Recursive Sort (ns)	Quick Sort (ns)	Nº testes
10	2171	2015	100000
15	2546	2281	100000
25	3593	3242	100000
35	3867	3554	100000
37	4648	4648	100000
40	4277	4843	100000
50	5375	5781	100000
100	8125	11562	100000
500	60937	79687	100000
1000	162500	165625	100000
2000	346875	426562	100000
5000	765625	1281250	100000
8000	1296875	1828125	1000
10000	2000000	2562500	1000
12000	2671875	3546875	1000
1000000	356562500	595156250	100

Tabela 1. Tempos médios em relação ao tamanho da array

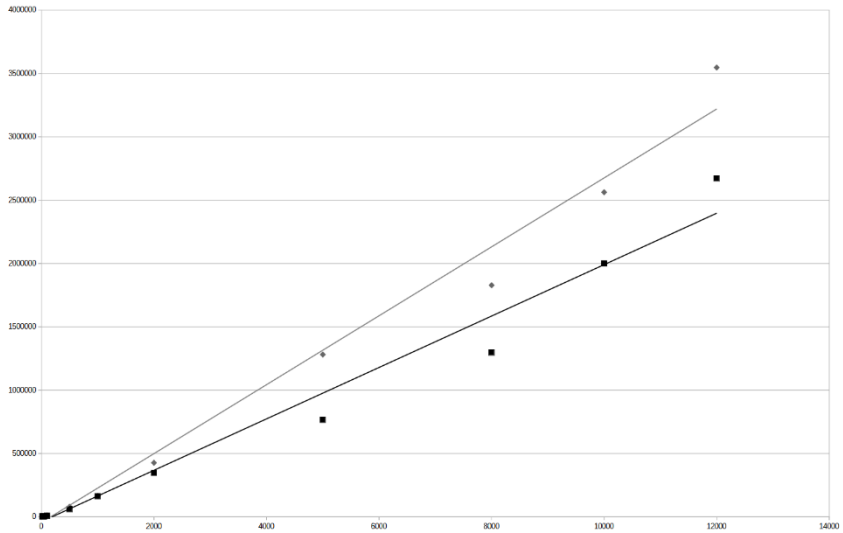

Gráfico I. Tempos médios (Quicksort – cinzento) (Recursive Sort – Preto)

Para cada entrada na tabela 1, n sendo o tamanho do array, foram realizados ~100000 testes ao qual o tempo médio de execução é a média geométrica dos tempos.

Os valores de n foram escolhidos semi-aleatóriamente de modo a encontrar o valor de n em que o quicksort demora o mesmo tempo que o recursive sort. Encontrando também os valores de n para quais o quicksort é mais lento e mais rápido que o recursive sort.

É evidenciado pelos dados da tabela 1 que para quase qualquer caso o Recursive Sort é melhor que o quicksort (dado que o recursive sort só ordena strings, e sacrifica muita memória para o fazer). Para arrays menores que 37 o quicksort é em média mais rápido que o Recursive Sort. Observando o gráfico I também é fácil deduzir que (a partir de ~37) o recursive sort é bastante mais rápido que o quicksort (e a diferença só aumenta). Deve ser notado que nenhum dos dois gráficos é linear, mas neste caso esta simplificação permite ver a sua diferença melhor.

Os arrays gerados são arrays onde cada string é composta por caracteres alfanuméricos (minúsculos e maiúsculos, total de 62 caracteres diferentes) os quais variam aleatoriamente de tamanho entre 1 e 1000 caracteres.

Complexidade temporal do Recursive Sort

n	Tempo Médio de Execução (ms)	Razão
250	0.2	0.0
500	0.5	3.0
1000	0.2	0.3
2000	0.5	3.0
4000	0.0	0.0
8000	1.7	0.0
16000	2.5	1.5
32000	6.3	2.5
64000	14.5	2.3
128000	33.6	2.3
256000	74.4	2.2
512000	148.1	2.0
1024000	321.9	2.2
2048000	755.3	2.3

Tabela 2. Ensaios de razão dobrada do recursive sort

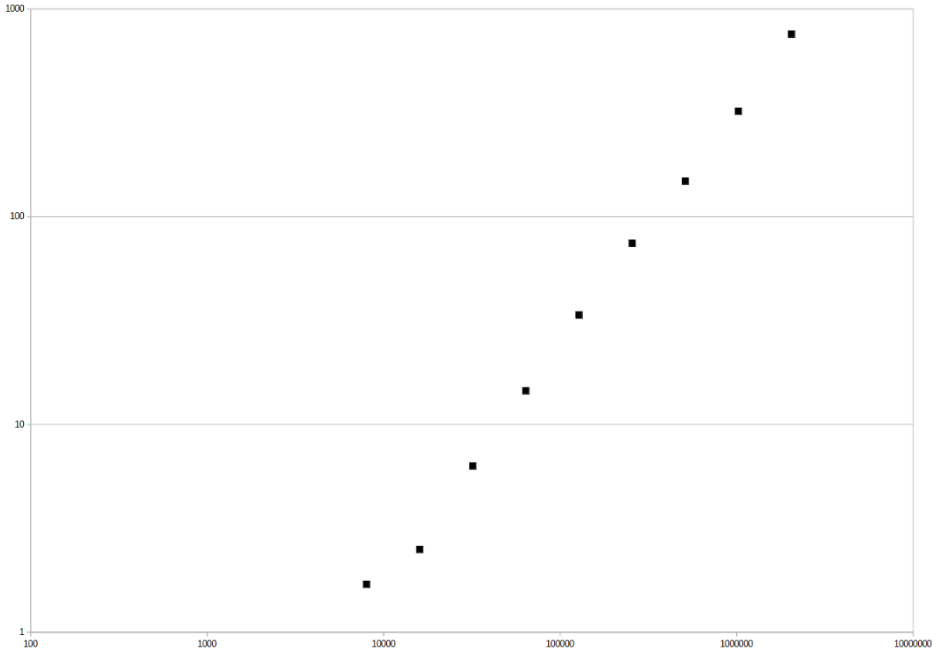

Gráfico II. (Escala log) Ensaios de razão dobrada do recursive sort

Para cada entrada na tabela 2 foram realizados 100 testes aos quais o tempo médio de execução é a média geométrica dos tempos dos testes.

Os exemplos gerados para os ensaios de razão dobrada são iguais aos da comparação anterior: entre 1 e 1000 caracteres alfanuméricos (de uma seleção de 62 caracteres únicos) para cada elemento do array de tamanho n.

Observando a tabela 2 podemos concluir que a razão flutua por valores acima de 2 (~2.3), com base nestes valores podemos assumir que a complexidade média assintótica é acima de O(n) mas menor que $O(n^2)$. Podemos então fazer uma estimativa muito crude que a complexidade talvez seja algo da ordem ~nlogn. O gráfico II também apresenta uma "reta" que aparenta ser mais inclinada que uma função linear, mas menos inclinada que uma parábola, confirmado a análise anterior.

Analisando o algoritmo podemos verificar que, tirando a recursividade, cada chamada ao método recursive sort vai percorrer 3n. Uma vez para determinar os limites, outra para gerar e distribuir as strings pelos baldes, e por fim outra para por as strings todas dos baldes "filhos" por ordem. Dada esta análise podemos concluir que uma chamada ao método vai pelo menos percorrer 3n elementos (assintoticamente n). Para além disto ao ordenar cada balde "filho" a função apela à recursividade, chamando-se a si própria para ordenar os seus baldes.

No caso médio podemos assumir que para cada profundidade, ou seja o index na string que a função está a percorrer, vão haver k elementos: k sendo o número de caracteres únicos nas strings dadas, isto claro parte do princípio que todos os caracteres têm uma distribuição uniforme, e a chance de encontrar cada um é igual.

O algoritmo divide então n por k baldes, mais uma vez assumindo que cada balde tem um número igual de strings no caso médio, cada balde vai ter $\frac{n}{k}$ elementos. Dentro de cada um destes baldes, os seus próprios baldes terão $\frac{\frac{n}{k}}{k}$ elementos. Dada esta análise podemos concluir que no caso médio a função vai a uma profundidade de $log_k(n)$. Pode também ser deduzido que, se para cada chamada recursiva, ela itera sobre $\frac{3n}{k}$ elementos, então todas as chamadas subsequentes vão também iterar sobre $\sum_1^k \frac{3n}{k} = 3n$ elementos.

Isto significa que para cada profundidade vão haver 3n iterações, se a profundidade do caso médio é dada por $log_k(n)$, então a complexidade do algoritmo para o caso médio tem de ser $3nlog_k(n)$, ou assintoticamente:

$$\boldsymbol{nlog_k(n)}$$

(onde k é o número de caracteres únicos que podem aparecer na string).

Dada esta análise podemos construir um gráfico o qual o eixo x é o tamanho da string (n) e y é dado por:

$$f(n) = nlog_k(n) \times \frac{\text{tempo de execução}}{n_{testado} log_k(n_{testado})}$$

Obtendo então uma estimativa para os tempos de execução interpolados através da complexidade deduzida dado um par de tempo de execução para um n. Se a análise da complexidade estiver correta, os pontos interpolados deverão ser bastante próximos dos valores obtidos experimentalmente.

Calculando os valores desta função para um ponto escolhido ao acaso da tabela 2: $(n_{testado} = 512000, tempo\ de\ execução = 148.1)$, obtemos os seguintes pontos interpolados:

n	Tempo Médio de Execução (ms)
8000	1.6
16000	3.4
32000	7.3
64000	15.6
128000	33.1
256000	70.1
512000	148.1
1024000	311.8
2048000	654.9

Tabela 3. Pontos interpolados a partir de n=512000

Obs: para n<8000 os valores não foram calculados pois os valores experimentais não têm grande precisão devido a limitações do java.

Com a tabela 3 conseguimos observar que deveras os valores interpolados são muito próximos dos valores reais, significando que a análise assintótica está correta e que a função apresenta a complexidade assintótica $nlog_k(n)$. Graficamente podemos observar este fenómeno mais facilmente:

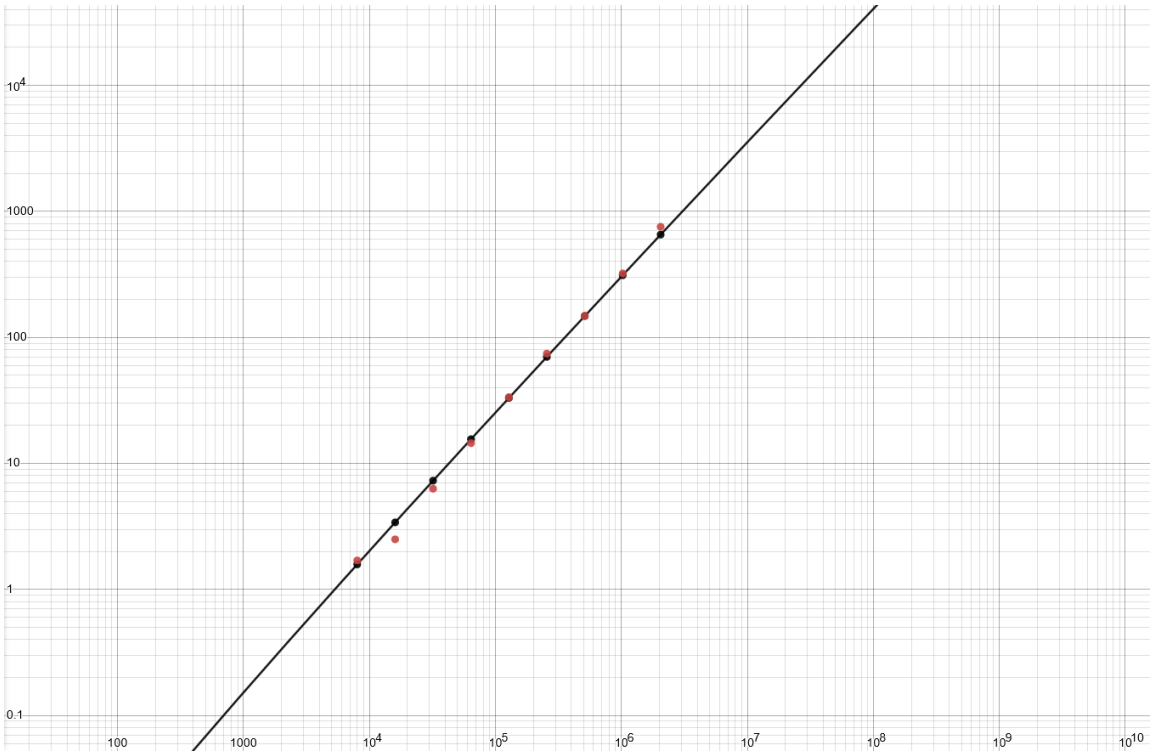

Gráfico III. (Escala log) Interpolação do tempo médio a partir de n

Obs: os valores interpolados estão a preto, enquanto os valores vermelhos são os pontos obtidos experimentalmente. A linha preta representa a complexidade assintótica calculada.

Pior caso do Recursive Sort

Analiticamente podemos concluir que independentemente do valor das strings, a função continua a iterar 3 vezes sobre n, logo 3n, ou n assintoticamente. A parte recursiva da função, por outro lado, tem dois casos em que se porta relativamente mal.

O primeiro caso não é muito mau, sendo ainda assintoticamente tão eficiente como o quicksort, em que o número de caracteres únicos na string (previamente denominado k) é um valor baixo (i.e. 2). Conforme a base do logaritmo é menor, a função cresce mais rapidamente, o que tem um impacto negativo na sua eficiência.

O segundo caso, e o real "pior caso" é um caso muito pior que o primeiro caso, seria um caso em que uma grande quantidade dos caracteres iniciais, ou mesmo todos os caracteres sejam iguais, isto faz com que o algoritmo acabe por não dividir as strings em baldes diferentes, passando só para o próximo balde com o tamanho das strings $(l, length)$ reduzido por um. Por outras palavras o algoritmo vê-se obrigado a ir até à profundidade máxima da string para a ordenar. Num array com n strings exatamente iguais de tamanho l, a sua complexidade assintótica vai então ser:

$$\boldsymbol{l \times n}$$

Esta dedução não é intuitiva quando vista de um ponto de vista assintótico, pois parece que a sua complexidade temporal é melhor que a do caso médio! Isto é verdade quando n tende para infinito, mas a realidade é que estamos limitados pela velocidade computacional, onde n vai ser muito menor que infinito. Desde que l seja um valor minimamente alto (i.e. 1000) para qualquer valor de n computável, o caso médio vai ser extremamente mais rápido que o pior caso. Isto dá-se porque $l \gg log_k(n)$ para qualquer caso relevante.

n	Tempo Médio de Execução (ms)	Razão
250	8.8	0
500	11.4	1.3
1000	24.1	2.1
2000	47.8	2.0
4000	93.8	2.0
8000	184.4	2.0
16000	341.6	1.9
32000	702.7	2.1
64000	1326.4	1.9
128000	3146.1	2.4
256000	6779.8	2.2
512000	14234.1	2.1
1024000	30663.9	2.2

Tabela 4. Razão dobrada do pior caso para $l = 1000$

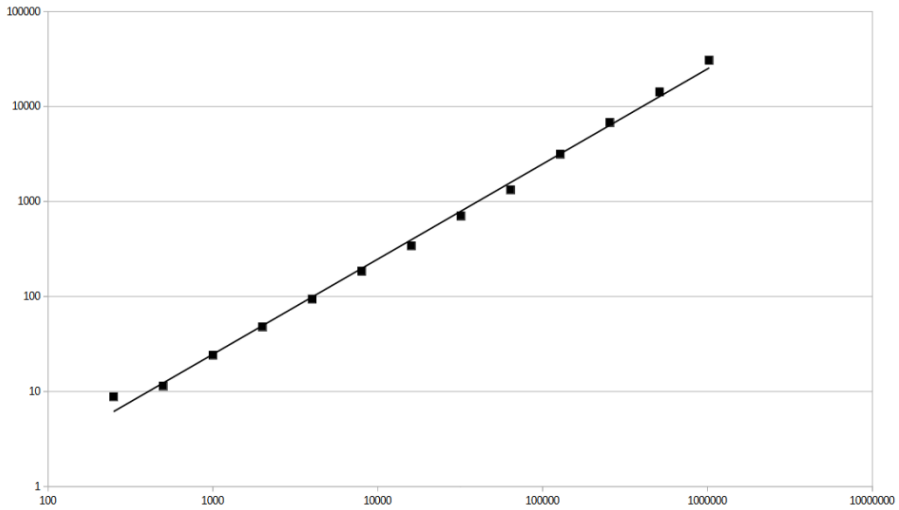

Gráfico IV. (Escala log) Razão dobrada do pior caso para $l = 1000$

n	Tempo Médio de Execução (ms)	Razão
250	6.6	0
500	5.6	0.9
1000	10.0	1.8
2000	19.8	2.0
4000	39.7	2.0
8000	79.4	2.0
16000	157.2	2.0
32000	318.1	2.0
64000	626.4	2.0
128000	1373.1	2.2
256000	3313.9	2.4
512000	6811.4	2.1
1024000	14702.5	2.2

Tabela 5. Razão dobrada do pior caso para $l = 500$

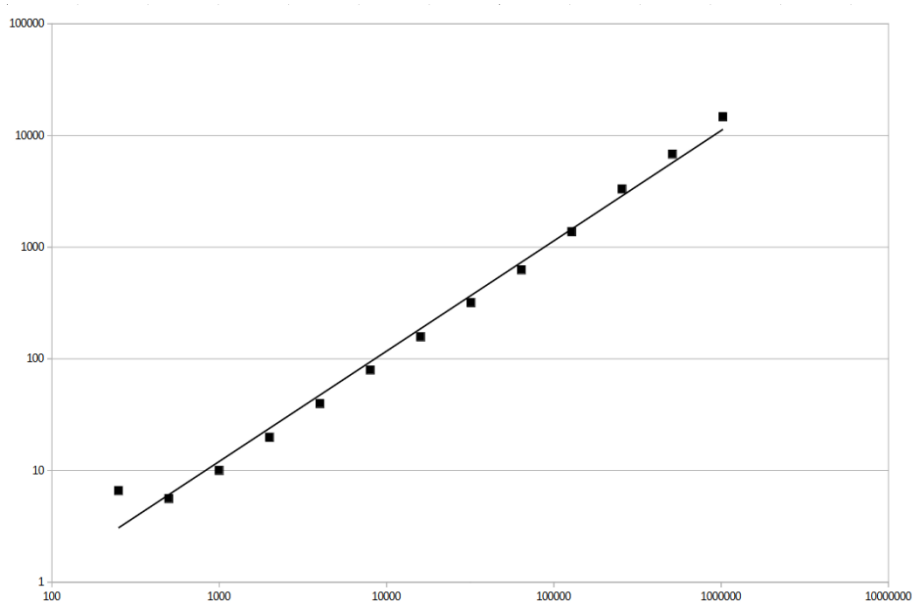

Gráfico V. (Escala log) Razão dobrada do pior caso para $l = 500$

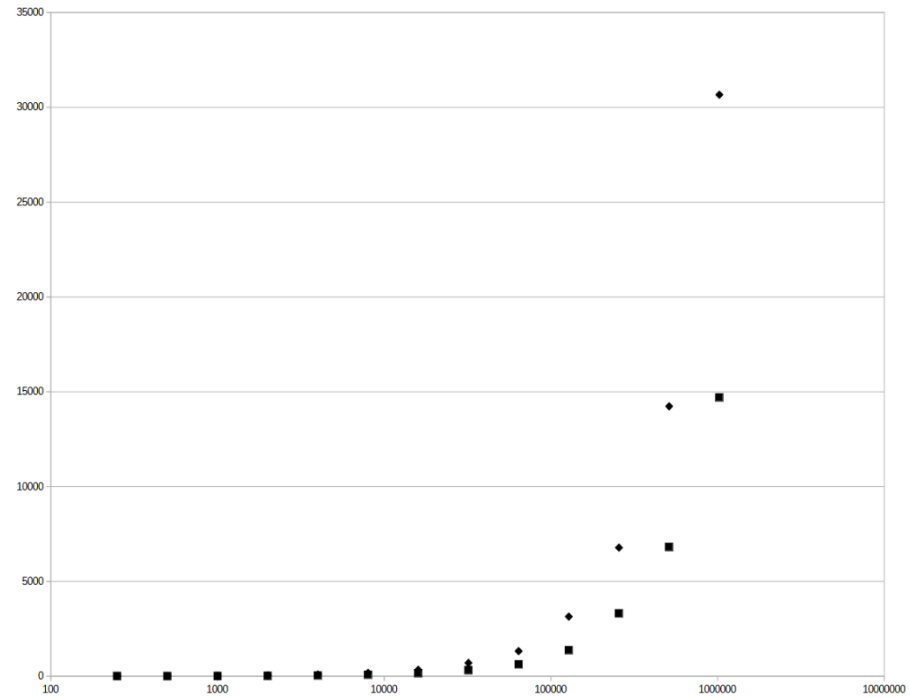

Gráfico IV-V. (Escala log de x) junção dos gráficos IV e V

Obs: Diamantes – gráfico IV ($l = 1000$); Quadrados – gráfico V ($l = 500$)

Os testes usados são análogos aos anteriores exceto para o tamanho das strings: n strings são geradas com k = 62 caracteres únicos uniformemente aleatórios, onde na tabela 4 o tamanho das strings, l é 1000, e na tabela 5 l é 500 (metade).

Com os testes representados nas tabelas 3, 4 e respetivamente nos gráficos IV e V para além do gráfico IV-V, é muito simples observar tanto a sua linearidade, como relação linear entre o l e o n no pior caso.

Com as tabelas 3, 4 e os gráficos IV, V conseguimos facilmente observar a linearidade assintótica da função no pior caso, a duplicação do valor anterior obtém uma aproximação muito boa do valor seguinte.

Também é fácil concluir que é linear pelo facto da razão rondar o valor 2.0, indicando que a complexidade é da ordem $O(n^1)$ ($\log_2(2) = 1$).

Com o gráfico IV-V e comparando as tabelas 3 e 4, nota-se que quando l passa para a metade, vê-se o tempo de execução médio também a baixar para a metade, isto é especialmente fácil de ver no gráfico IV-V onde para cada valor de x, (representando cada valor de n), o tempo de execução no gráfico de $\frac{l}{2}$ **é equivalente a** $\frac{\textbf{tempo de execução}}{2}$. Provando então a sua relação linear. Por outras palavras, o facto dos valores da tabela 5 serem metade dos da tabela 4, provam a relação $l \times n$.

Complexidade espacial do Recursive Sort

n	Memória Utilizada (mb)	Razão
250	0.0	0
500	0.0	2.7
1000	0.1	3.9
2000	0.2	1.6
4000	0.4	1.7
8000	0.5	1.3
16000	0.7	1.5
32000	2.1	2.9
64000	8.3	4.0
128000	14.7	1.8
256000	22.1	1.5
512000	34.6	1.6
1024000	55.6	1.6
2048000	169.9	3.1

Tabela 6. Razão dobrada da memória utilizada

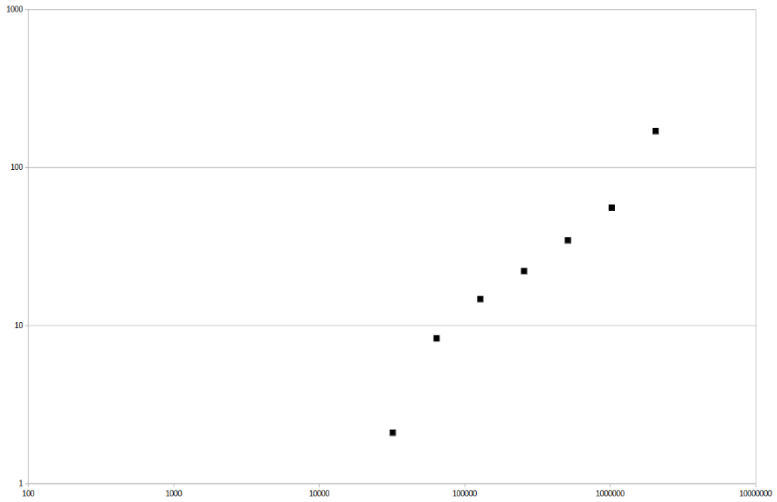

Gráfico VI. (Escala log) Razão dobrada da memória utilizada

Os testes usados são iguais aos para a complexidade temporal média: n strings são geradas com k = 62 caracteres únicos uniformemente aleatórios, onde o tamanho de cada string é aleatório (uniformemente) entre 1 e 1000.

É difícil assumir uma complexidade com base nos valores obtidos, a razão flutua bastante. A média geométrica da razão é 2.25, se assumirmos este valor como a razão "real" da complexidade espacial, então chegamos à conclusão que a complexidade temporal deverá estar entre O(n) e O(n^2).

Analisando o algoritmo no caso médio, podemos concluir que cada balde será preenchido por $\frac{n}{k}$ strings. Com uma lógica análoga à da complexidade temporal média, concluímos que o somatório de todos os baldes para cada profundidade vai em média ter n elementos. Tendo n elementos a serem guardados por profundidade, só nos resta saber a profundidade média para obter a média dos elementos a serem guardados. Mais uma vez como visto no cálculo da complexidade temporal, foi demonstrado que a profundidade será igual a $log_k(n)$, logo a complexidade espacial assintótica vai ser da ordem de:

$$\boldsymbol{nlog_k(n)}$$

Embora assintoticamente o cálculo anterior esteja correto, deve ser mencionado que o array inicial é passado "por referência" (pelo valor da referência), e como consequência o algoritmo em si não guarda o array inicial, logo não gasta quase memória na primeira profundidade. Tomando isto em conta e chega-se à expressão $n(log_k(n) - 1)$, o que embora não seja a expressão assintótica, bate com os dados obtidos experimentalmente muito melhor, pois $log_k(n)$ vai sempre ser um valor muito baixo, e o -1 vai fazer um impacto significativo quando n é um valor real e não infinito.

Outra observação notável é que também vai haver overhead do algoritmo, para além da recursividade e da chamada de uma função adicional, também há várias variáveis a ser declaradas e usadas nos cálculos intermédios do algoritmo, mas este valor é constante e vai ter um impacto negligenciável comparativamente com os arrays de strings.

Dada esta análise podemos construir um gráfico o qual o eixo x é o tamanho da string (n) e y é dado por:

$$f(n) = n(log_k(n) - 1) \times \frac{memoria\ utilizada}{n_{testado}(log_k(n_{testado}) - 1)}$$

Obtendo então uma estimativa para a memória usada pelo algoritmo interpolada através da complexidade deduzida, dado um par de valores (memoria utilizada, n). Se a análise da complexidade estiver correta, os pontos interpolados deverão ser bastante próximos dos valores obtidos experimentalmente.

Calculando os valores desta função para um ponto escolhido ao acaso da tabela 6: $(memoria\ utilizada = 0.7, n_{testado} = 16000)$, obtemos os seguintes pontos interpolados:

n	Memória Utilizada (mb)
1000	0.0
2000	0.1
4000	0.1
8000	0.3
16000	0.7
32000	1.6
64000	3.5
128000	7.7
256000	17.0
512000	36.4
1024000	78.4
2048000	167.9

Tabela 7. Razão dobrada da memória utilizada

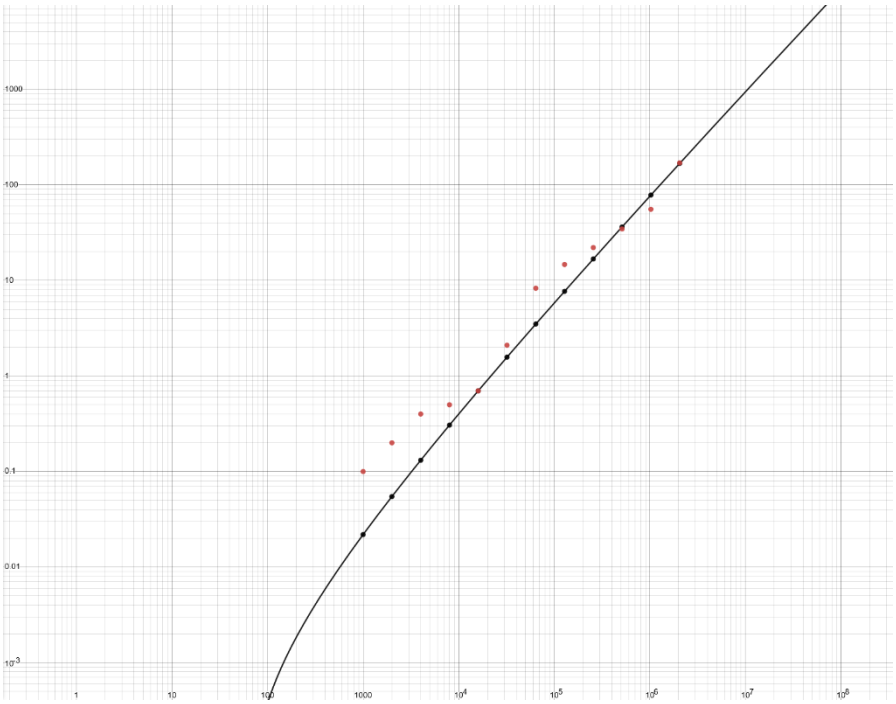

Gráfico VII. (Escala log) Interpolação da memória utilizada a partir de n

Obs: os valores interpolados estão a preto, enquanto os valores vermelhos são os pontos obtidos experimentalmente. A linha preta representa a complexidade assintótica calculada.

Por análise gráfica, observando o gráfico VII é possível concluir que a interpolação está a estimar bem a memória utilizada em função de n. Mostrando que a complexidade espacial é de ordem $n(log_k(n) - 1)$.

Pode ser observado uma oscilação nos pontos experimentais, tal oscilação é provável que seja causada pelo garbage collector do java, que embora tenha tentado ser mitigado nos testes, não consegue ser desligado e, portanto, dependendo de n. Este agente vai alterar de

forma periódica os valores estimados experimentalmente, o que é representado por oscilações gráficas e por uma razão errática.