Análise de Algoritmos - Trabalho 1
Clara de Mattos Szwarcman - 1310351
Lucas Ribeiro Borges -
Guilherme Simas Abinader -

1 - Controle de Qualidade na Produção de Frascos de Vidro

1)

A altura será dividida em raiz de n intervalos, aonde cada intervalo possui raiz de n degraus. O primeiro frasco será jogado de raiz de n em raiz de n degraus. Quando o frasco quebrar, jogaremos o segundo frasco a partir do início desse intervalo de degrau em degrau até que ele quebre.

Pseudo Código:

$Degrau\_2\_frascos(x, n)$

$raiz\_n = sqrt(n);$

$for\ i = 0;\ i < n;\ i+ = raiz\_n$

$if\ i >= x$

$for\ j = i - raiz_n;\ j < i;\ j++$

$if\ j == x$

$return\ j;$

Quando o primeiro frasco quebrar teremos um intervalo de tamanho $\sqrt{n}$ que com certeza contém a altura em que o frasco quebra, visto que se o frasco não quebrou no degrau anterior ao início do intervalo, ele também não quebra em nenhum dos degraus abaixo dele. Assim, ao percorrermos o intervalo de um em um, encontraremos a altura x.

No pior dos casos, o frasco quebra no último degrau, portanto, o primeiro frasco será jogado de todos os intervalos ($\sqrt{n}$ vezes.) Como ele quebrou no último degrau, será conferido se ele quebra em algum degrau pertencente ao último intervalo. Dessa maneira, o segundo frasco será jogado em cada degrau do intervalo ($\sqrt{n}$ vezes), até finalmente quebrar no último degrau. Sendo assim, foi jogado $2 * \sqrt{n}$.

$$O(2 * \sqrt{n}) = O(\sqrt{n})$$

2)

Tendo 3 frascos:

O primeiro frasco será jogado em intervalos de $n^{\frac{2}{3}}$ até quebrar. O segundo frasco será jogado em intervalos de $n^{\frac{1}{3}}$, no intervalo de $n^{\frac{2}{3}}$ encontrado. O terceiro frasco será jogado de degrau em degrau no intervalo de $n^{\frac{1}{3}}$ encontrado.

Tendo 4 frascos:

O primeiro frasco será jogado em intervalos de $n^{\frac{3}{4}}$ até quebrar. O segundo frasco será jogado em intervalos de $n^{\frac{2}{4}}$, no intervalo de $n^{\frac{3}{4}}$ encontrado. O terceiro frasco será jogado em intervalos de $n^{\frac{1}{4}}$, no intervalo de $n^{\frac{2}{4}}$ encontrado. O quarto frasco será jogado de degrau em degrau no intervalo de $n^{\frac{1}{4}}$ encontrado.

Tendo k frascos:

A altura ($(\sqrt[k]{n})^k$ degraus)é dividida em $\sqrt[k]{n}$ intervalos de tamanho $(\sqrt[k]{n})^{k-1}$. Jogamos o primeiro frasco em intervalos de $(\sqrt[k]{n})^{k-1}$ degraus. Quando o frasco quebrar, o último intervalo ($(\sqrt[k]{n})^{k-1}$ degraus) será dividido em $\sqrt[k]{n}$ intervalos de tamanho $(\sqrt[k]{n})^{k-2}$ degraus e o segundo frasco será jogado em intervalos de $(\sqrt[k]{n})^{k-2}$ degraus. Isso ocorrerá sucessivamente para todos os k frascos. No frasco k teremos um intervalo de tamanho $(\sqrt[k]{n})^{k-(k-1)}$, que é igual a $\sqrt[k]{n}$. O frasco será jogado em intervalos de $(\sqrt[k]{n})^{k-k}$, ou seja, de degrau em degrau, até quebrar.

Pseudo Código:

$Degrau\_k\_frascos(x, n, k)$

$raiz\_kesima = raiz(n, k)$

$inicio = 0;$

$fim = n;$

$incremento = pow(raiz\_kesima, k - 1)$

$for\ i = 0;\ i < k;\ i++$

$for\ j = inicio;\ j < fim;\ j+ = incremento$

$if\ j >= x$

$if\ incremento == 1$

$return\ j;$

$inicio = j - incremento;$

$fim = j;$

$incremento = incremento/raiz\_kesima;$

$break;$

Segue a premissa do primeiro, aonde sempre se tem certeza do intervalo em que ocorre a quebra, porém com mais frascos para serem utilizados. Portanto, podemos inicialmente dividir a altura em intervalos maiores com mais subdivisões, assim postergando a procura de um em um, que será feita em um intervalo menor.

Para cada frasco estamos realizando no máximo $\sqrt[k]{n}$ testes, visto que para o frasco i temos um espaço de $(\sqrt[k]{n})^{k-(i-1)}$ degraus e o jogaremos em intervalos de $(\sqrt[k]{n})^{k-i}$ degraus. Como temos k frascos, isso será realizado k vezes. Assim, o número total de quedas será $k * \sqrt[k]{n}$.

$$O(k * \sqrt[k]{n})$$

Se k, é um número fixo, a complexidade será O($\sqrt[k]{n}$), como provado anteriormente.

3)

A menor complexidade assintótica possível é de $O(logn)$.

O algoritmo realiza uma busca binária ao longo da escada, jogando um frasco a cada comparação, se o frasco quebra, busca-se na metade inferior, do contrário busca-se na metade superior. Quando o intervalo é de 1 degrau, podemos garantir que encontramos a altura correta.

Pseudo Código:

$Degrau\_logn\_frascos(x, n)$

$busca\_binaria(x, n)$

Resultados

Nome do Arquivo	Degraus	Frascos	cpu time (ms)
bignum_32_01	2^{32}	1	7115744.463ms
bignum_32_01	2^{32}	2	194.01299999999998ms
bignum_32_01	2^{32}	4	1.7309999999999999ms
bignum_32_01	2^{32}	8	0.40499999999999914ms
bignum_32_01	2^{32}	16	0.4539999999999995ms
bignum_32_01	2^{32}	32	0.7480000000000004ms
bignum_32_02	2^{32}	1	5761534.251999999ms
bignum_32_02	2^{32}	2	198.36ms
bignum_32_02	2^{32}	4	1.7560000000000007ms
bignum_32_02	2^{32}	8	0.3969999999999998ms
bignum_32_02	2^{32}	16	0.4469999999999995ms
bignum_32_02	2^{32}	32	0.7430000000000005ms
bignum_64_01	2^{64}	2	13550835.350000001ms
bignum_64_01	2^{64}	4	409.81199999999995ms
bignum_64_01	2^{64}	8	3.5249999999999986ms
bignum_64_01	2^{64}	16	0.7790000000000001ms
bignum_64_01	2^{64}	32	0.8859999999999997ms
bignum_64_02	2^{64}	4	429.72499999999997ms
bignum_64_02	2^{64}	8	4.684999999999998ms
bignum_64_02	2^{64}	16	0.9440000000000004ms
bignum_64_02	2^{64}	32	1.0919999999999992ms
bignum_128_01	2^{128}	8	898.033ms
bignum_128_01	2^{128}	16	9.008999999999997ms
bignum_128_01	2^{128}	32	2.252999999999998ms
bignum_128_02	2^{128}	8	896.0749999999999ms
bignum_128_02	2^{128}	16	9.394ms
bignum_128_02	2^{128}	32	1.906999999999999ms
bignum_192_01	2^{192}	16	125.40499999999999ms
bignum_192_01	2^{192}	32	5.802000000000001ms
bignum_192_02	2^{192}	16	125.98100000000001ms
bignum_192_02	2^{192}	32	5.7680000000000025ms
bignum_256_01	2^{256}	16	1973.921ms
bignum_256_01	2^{256}	32	19.016ms
bignum_256_02	2^{256}	16	1875.7930000000001ms
bignum_256_02	2^{256}	32	19.178ms

2. Problema da Mochila Fracionária (pode-se colocar parte de um objeto na mochila)

1.a)

Os objetos são ordenados por seu valor por peso com merge sort. São então adicionados a mochila de um em um começando pelo com maior valor por peso, até a capacidade ser atingida.

Pseudo Código:

$struct\ objeto$

$int\ valor;$

$int\ peso;$

$float\ densidade;$

$int\ indice;$

$mochila\_frac1(w, v, n, W)$

$for\ i = 0;\ i < n;\ i++$

$objetos[i] = cria\_objeto(v[i], w[i], i);$

$objetos\_selecionados[i] = 0;$

$mergesort\_densidade(objetos, n);$

$sum\_peso = 0;$

$for\ j = n - 1;\ j >= 0;\ j--$

$if\ sum\_peso == W$

$return\ objetos\_selecionados;$

$indice = objetos[j] -> indice;$

$peso = objetos[i] -> peso;$

$if\ peso + sum\_peso < W$

$objetos\_selecionados[indice] = peso;$

$sum\_peso+ = peso;$

$else$

$objetos\_selecionados[indice] = W - sum\_peso;$

$sum\_peso+ = W - sum\_peso;$

$return\ objetos\_selecionados;$

Com os elementos ornados por valor por peso, podemos facilmente sempre escolher quais resultarão em um maior valor na mochila.

Inicializar e percorrer os vetores é realizado em $O(n)$. A ordenação com mergesort é realizada em $O(n\ logn)$.

$$O(c*n + n \log n) = O(n \log n)$$

1.b)

Encontra-se a mediana do valor por peso e particiona-se o vetor.

- Se a metade de maior valor cabe na mochila, todos os objetos são colocados na mochila e realiza-se o algoritmo na metade de menos valor.
- Se a metade de maior valor não cabe na mochila, realiza-se o algoritmo na metade de maior valor.
- Se a metade de maior valor possui apenas um objeto, coloca-se a fração dele que cabe na mochila.

Pseudo Código:

$struct\ objeto$

$int\ valor;$

$int\ peso;$

$float\ densidade;$

$int\ indice;$

$mochila\_frac2(w, v, n, W)$

$for\ i = 0;\ i < n;\ i++$

$objetos[i] = cria\_objeto(v[i], w[i], i);$

$objetos\_selecionados[i] = 0;$

$mochila\_frac2\_rec(objetos, W, 0, n, objetos\_selecionados);$

$return\ objetos\_selecionados;$

$mochila\_frac2\_rec(objetos, W, ini, fim, objetos\_selecionados)$

$if\ ini > fim$

$return;$

$if\ ini == fim$

$indice = objetos[ini] -> indice;$

$if\ objetos[ini] -> peso >= W$

$objetos_selecionados[indice] = W;$

$else$

$objetos_selecionados[indice] = objetos[ini] -> peso$

$return;$

$meio = ini + fim/2$

$k = kesima\_densidade(objetos, meio);$

$part\_inv\_densidade(ini, fim, objetos, k);$

$soma = somar\_peso(objetos, ini, meio);$

$if\ soma > W$

$mochila\_frac2\_rec(objetos, W, ini, meio, objetos\_selecionados);$

$else$

$for\ i = ini;\ i < meio;\ i++;$

$indice = objetos[i] -> indice;$

$peso = objetos[i] -> peso;$

$objetos\_selecionados[indice] = peso;$

$mochila\_frac2\_rec(objetos, W - soma, meio, fim, objetos\_selecionados)$

Com o particionamento dos objetos pela mediana do valor por peso, asseguramos que sempre que uma metade é colocada na mochila, esta é a metade de maior valor.

Achar a mediana, particionar e somar são realizados em $O(n)$. Assim, temos a seguinte relação de recorrência:

$$f(n) \leqslant \begin{cases} c,\ n = 1 \\ f(\frac{n}{2}) + c'n,\ n > 1 \end{cases}$$

Pelo Teorema Mestre :

$a = 1$
$b = 2$
$k = 1$
$a < b^k$

$$O(n^k) = O(n)$$

1.c)

Pseudo Código:

struct objeto

int valor;

int peso;

float densidade;

int indice;

$mochila\_frac3(w, v, n, W)$

$for\ i = 0;\ i < n;\ i++$

$objetos[i] = cria\_objeto(v[i], w[i], i);$

$objetos\_selecionados[i] = 0;$

$mochila\_frac3\_rec(objetos, W, 0, n, objetos\_selecionados);$

return objetos_selecionados;

$mochila\_frac3\_rec(objetos, W, ini, fim, objetos\_selecionados)$

$if\ ini > fim$

$return;$

$if\ ini == fim$

$indice = objetos[ini] -> indice;$

$if\ objetos[ini] -> peso >= W$

$objetos_selecionados[indice] = W;$

$else$

$objetos_selecionados[indice] = objetos[ini] -> peso$

$return;$

$meio = ini + fim/2$

$valor_pivot = media_densidade(objetos, ini, fim);$

$part\_inv\_densidade(ini, fim, objetos, valor_pivot);$

$soma = somar\_peso(objetos, ini, meio);$

$if\ soma > W$

$mochila\_frac3\_rec(objetos, W, ini, meio, objetos\_selecionados);$

$else$

$for\ i = ini;\ i < meio;\ i++;$

$indice = objetos[i] -> indice;$

$peso = objetos[i] -> peso;$

$objetos\_selecionados[indice] = peso;$

$mochila\_frac3\_rec(objetos, W - soma, meio, fim, objetos\_selecionados)$

Agora podemos ter um caso de particionamento desbalanceado, onde sempre um lado tem 1 elemento e o outro tem n-1 elementos. Se todos elementos menos uma fração do último cabem na mochila, todos serão percorridos. Assim, teremos intâncias de $n-1, n-2, ..., 1$, elementos. Portanto a soma de todas as iterações é igual a $\frac{n-1(n-2)}{2}$.

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

Resultados

Nome do Arquivo	Num exec	cpu/Alg 1	cpu/Alg 2	cpu/Alg 3
m50.in	10000	0.000309375	0.00073125	0.00008125
m58.in	8621	0.0003842362	0.0008808433	0.00007793469
m67.in	7463	0.0004438564	0.0009526162	0.0001046831
m78.in	6411	0.0003436476	0.001013882	0.000112112
m91.in	5495	0.0006596906	0.001632166	0.0001393312
m106.in	4717	0.0007751219	0.00190468	0.0001192495
m124.in	4033	0.0009182061	0.002405932	0.0001820915
m145.in	3449	0.0006886054	0.001857422	0.0001948028
m169.in	2959	0.001378211	0.003284471	0.000264025
m197.in	2539	0.001643117	0.003877019	0.0002338519
m230.in	2174	0.001918986	0.004585442	0.0003449862
m269.in	1859	0.001395239	0.003353618	0.0003446073
m314.in	1593	0.002854284	0.006287272	0.0004413842
m367.in	1363	0.003335932	0.007348221	0.0004126926
m429.in	1166	0.003953152	0.008871141	0.0006164237
m501.in	999	0.002846597	0.006475225	0.000609985
m586.in	854	0.005818208	0.01266101	0.0008233314
m685.in	730	0.006913527	0.01502568	0.0007491438
m801.in	625	0.0081	0.01705	0.00115
m937.in	534	0.005559457	0.01331344	0.001228933
m1096.in	457	0.01182987	0.02424097	0.001572757
m1282.in	391	0.01414642	0.02709399	0.001478581
m1499.in	334	0.01665419	0.03424401	0.002198728
m1753.in	286	0.0111451	0.02545892	0.002239948
m2051.in	244	0.02407787	0.04482582	0.003009734
m2399.in	209	0.02900718	0.05554725	0.002766148
m2806.in	179	0.03421788	0.06407123	0.004189944
m3283.in	153	0.0223652	0.04381127	0.004289216
m3841.in	131	0.0492605	0.08277672	0.005605916
m4493.in	112	0.05929129	0.1060268	0.005580357
m5256.in	96	0.0703125	0.1220703	0.008463542
m6149.in	82	0.04668445	0.0929878	0.008384146
m7194.in	70	0.1008929	0.1747768	0.01183036
m8416.in	60	0.1208333	0.2036458	0.01041667
m9846.in	51	0.1433824	0.2352941	0.01593137
m11519.in	44	0.09019886	0.1420455	0.01669034
m13477.in	38	0.2088816	0.3322368	0.02179276
m15768.in	32	0.2524414	0.3867188	0.02148438
m18448.in	28	0.3024554	0.4665179	0.03459821
m21584.in	24	0.1855469	0.4733073	0.03515625
m25253.in	20	0.440625	0.65	0.05
m29546.in	17	0.5303309	0.7702206	0.05238971
m34568.in	15	0.634375	0.8895833	0.078125
m40444.in	13	0.3858173	0.8389423	0.07451923
m47319.in	11	0.9488636	1.227273	0.1122159
m55363.in	10	1.157812	1.45625	0.1015625
m64774.in	8	1.386719	1.730469	0.1640625
m75785.in	7	0.8415179	1.366071	0.1495536
m88668.in	6	2.125	2.416667	0.2265625
m103741.in	5	2.559375	2.828125	0.078125

É possível observar que o terceiro algoritmo é sempre o de menor tempo. Acreditamos que os possíveis motivos para que isso ocorra são:

- A função do kesimo e/ou o MergeSort foram implementados de forma ineficiente.
- O pior caso descrito na última questão não ocorre.
- Embora o algoritmo que calcula o kesimo seja linear, a constante pela qual n é multiplicado é muito grande.