Problema da Mochila Relacional

André Almeida *164047*
MC658 - Lab 02

I. O problema

São dados n itens, $N = 1, ..., n$, e uma mochila de capacidade C. Cada item $i \in N$ tem um peso $s_i > 0$ e valor v_i e cada par de itens $i, j \in N$ tem uma relação r_{ij} (não necessariamente positiva). O objetivo é encontrar um subconjunto $S \subseteq N$ tal que $\sum_{i \in S} s_i \leqslant C$ e $\sum_{i \in S} v_i + \sum_{i \in S} \sum_{j \in S, j > i} r_{ij}$ é máximo.

A entrada do problema é dada por:

- Número de itens;
- Matriz de relações de itens;
- Vetor de valor de itens;
- Vetor de peso de itens;
- A capacidade da mochila e
- O tempo máximo disponível para o cálculo da solução

Todos os resultados mostrandos foram executados em uma máquina com processador Intel Core i7-3537U CPU @ 2.00GHz, rodando com GNU/Linux 4.15. O compilador usado foi o GCC 7.3.0, usando as *flags* de compilação do Makefile disponibilizado.

II. Algoritmo exato

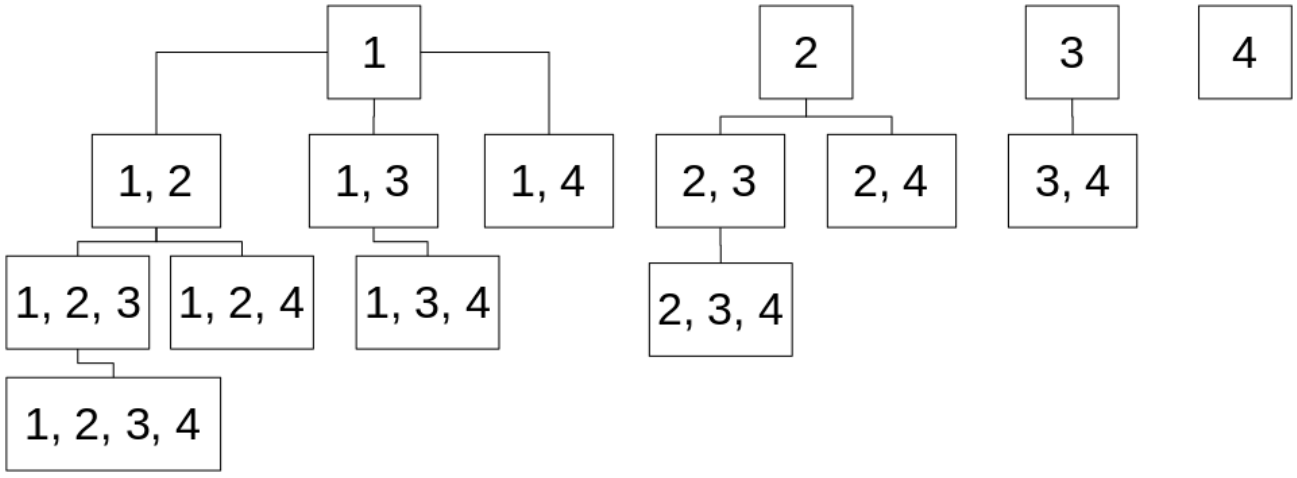
A abordagem exata para o problema foi criar uma estrutura de solução global, com a soma dos itens e um vetor de índices de itens da solução. Esta estrutura é atualizada sempre que é encontrada uma solução melhor.

O algoritmo cria uma árvore de combinações de índices para cada item da entrada. As combinações geradas por cada nó inserem somente índices que sejam maiores que o último índice do nó pai, de forma a evitar permutações. Por exemplo, seja n o número de itens e dado um nó com índices, $i_1, i_2, i_3, ..., i_j$, onde $i_k < i_{k+1}$, esse nó gerará $n - i_j$ folhas. Para cada $m \in [i_j + 1, n]$, o nó ira uma folha inserindo em cada folha os índices do nó pai ($i_1, i_2, i_3, ..., i_j$) mais o índice m.

Se os itens de um nó não cabem na mochila, esse nó vira uma folha e não gera mais combinações. Por exemplo, se um conjunto de itens (i_1, i_3, i_7) já ocupa a capacidade máxima da mochila, nenhum item i_k (já que o peso de i_k é maior que 0) será capaz de ocupar a mochila e não faz sentido calcularmos soluções com o conjunto inicial.

Agora, dada uma raíz (que pela definição anterior possúi apenas um item), verificamos se este item cabe na mochila e, se cabe, chamamos uma rotina para calcular os valores de seus nós-filhos. Essa rotina passa por parâmetro o valor do item e a capacidade total da mala subtraída do peso do item. Para cada filho que couber na mala, é calculada a soma dos itens selecionados (levando em conta a soma das relações entre os itens) e é feita uma chamada recursiva passando a soma atual e reduzindo a capacidade da mala com o peso do novo item.

Por exemplo, para uma entrada com 4 itens, o algoritmo irá gerar a seguinte árvore de combinações:



a. Implementação

A implementação do algoritmo é feita da seguinte maneira, começando pela rotina principal:

> *1.* Iniciar o alarme com o tempo máximo e ordenar decrescentemente os `n` itens pelo seu *valor relativo*[1].
> *2.* Criar uma estrutura global `solução`, com a soma igual à 0 e o conjunto de itens vazio
> *3.* Chamar a rotina `calculaSolução(items, valores, pesos, relações, capacidade)`
> *4.* Quando a rotina acabar ou quando o alarme acabar, retorna o conjunto de items e a soma da solução atual

[1]O *valor relativo* é calculado fazendo a soma do valor de um item mais todas os valores de suas relações e dividindo pelo seu peso:

> valor_relativo = (somaTodasRelações(item) + valor[item]) / peso[item]

Usamos um vetor ordenado para, no caso em que o tempo estourar, ter uma chance que tenhamos começado por um item que venham a ter soluções mais próximas da ótima.

A rotina `calculaSolução(...)` por sua vez é definida por:

> *1.* Para cada índice `i` no vetor ordenado:
> *2.* Verificar se o item cabe na mochila e, se couber:
> *3.* Cria uma solução com apenas este elemento e compara com a solução global. Se for melhor, substituiu ela.
> *4.* Se ainda houver espaço na mochila, chama a rotina `calculaFilhos([i], valor[i], capacidade - v[i])`

Nessa rotina `calculaSolução(...)` criamos soluções com apenas um item e, caso ainda exista espaço disponível, chamamos uma subrotina para verificar as combinações que são "filhas" desta rotina, de forma a percorrer todas as combinações que são válidas. Para cada `i`, passamos o `valor[i]` como uma soma parcial e reduzimos a capacidade da mochila subtraindo o peso de `i`. A rotina `calculaFilhos(itens[], soma_parcial, capacidade)` funciona de maneira

semelhante, mas sem a etapa de buscar os itens ordenados de forma a evitar consultas a vetor que iriam pesar no desempenho do algoritmo.

Rotina `calculaFilhos(itens[], soma_parcial, capacidade_parcial)`:

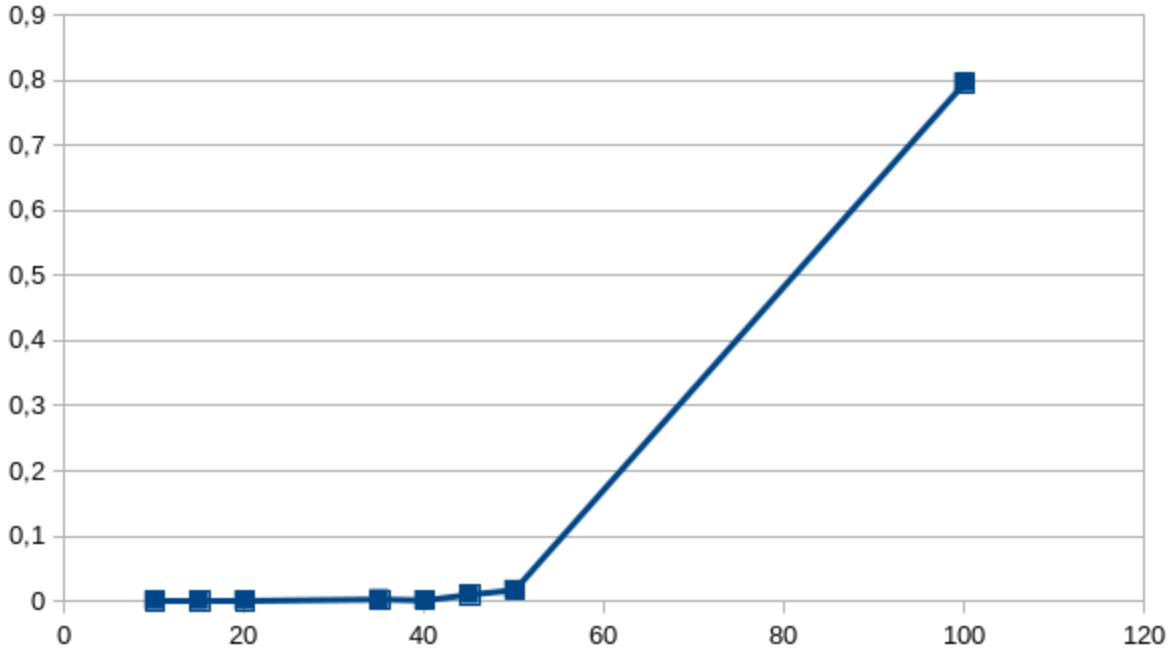
> *1.* Para `j`, começando no último índice de `itens[] + 1` até `número total de itens`:
> *2.* Se o alarme esgotou, retornamos a solução global do momento
> *3.* Se o `peso[j]` cabe na mochila:
> *4.* Adiciona o `j` no vetor `itens`
> *5.* Calculamos a `soma_atual`, somando a `soma_parcial` + `valor[j]` + `valorRelacoes(j, itens)`
> *6.* Se esta soma for maior do que a solução global, substituiu a global por ela
> *7.* Se ainda existe espaço na mochila:
> *8.* Chamamos `calculaFilhos(itens[], soma_atual, capacidade - peso[j])`
> *9.* Retiramos `j` do vetor `itens`

O cálculo na linha *5* de `valorRelacoes()` é feito tomando cuidado para não somar duas vezes a mesma relação. Somamos apenas as relações na coluna da matriz do item novo, pegando os itens que já estão na mocilha. Como a coluna do item novo nunca foi percorrida antes, os itens que iremos somar não serão repetidos.

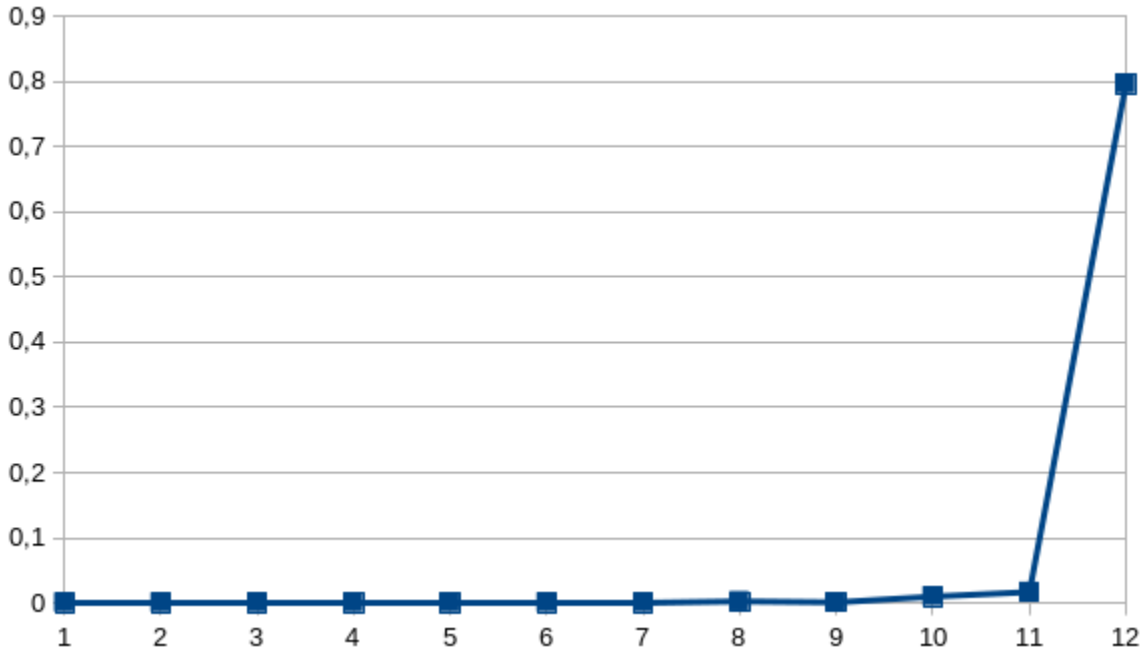
b. Resultados

Casos pequenos

Para os casos de teste de 01 à 12, o algoritmo exato executou abaixo de um segundo e obteve a resposta esperada. O tempo de execução do algoritmo crescia exponencialmente conforme o número de itens da entrada, como é possível ver nos gráficos abaixo:



Tempo de execução (seg) x Número de itens

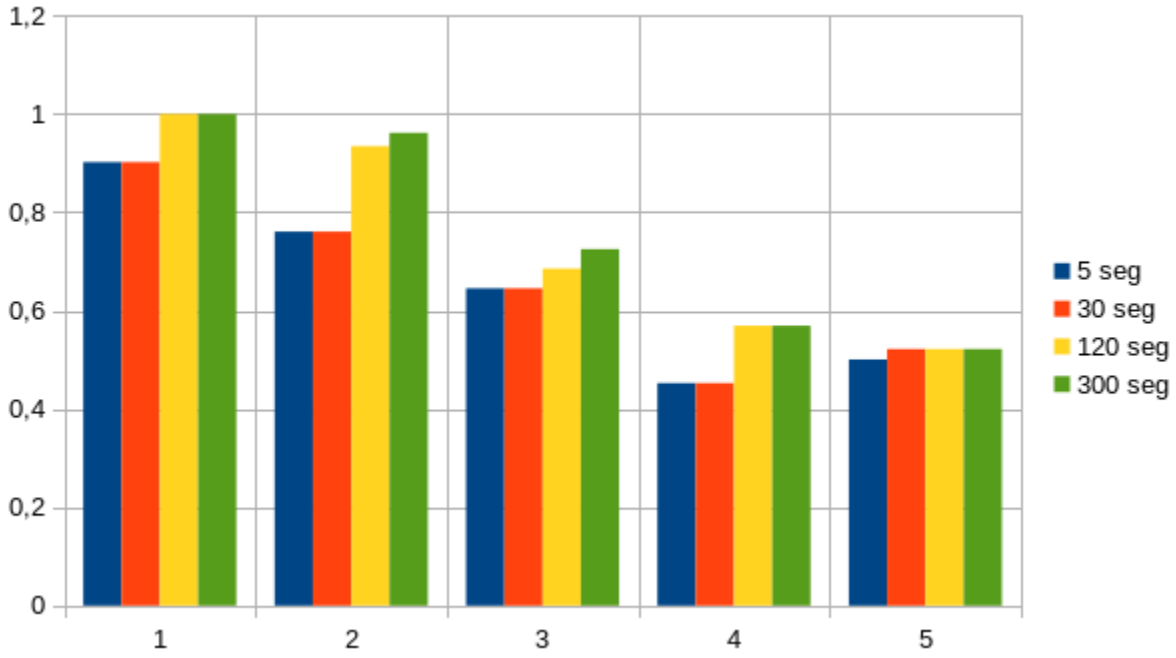


Tempo de execução (seg) x Caso teste

Casos grandes

Para os casos grandes, o algoritmo não conseguia ser executado em tempo hábil. Foram testados vários tempos de corte para o algoritmo. Conforme aumenta o tamanho da entrada, a influência de cada segundo diminui para a aproximação do

resultado exato, mostrando o caráter exponencial do algoritmo.



Precisão X Grupos de caso teste

Precisão é o valor obtido dividido pelo valor esperado, variando de 0 à 1, equivalente à porcentagem. Os grupos de casos são, respectivamente, $1 \equiv 200, 2 \equiv 300, ..., 5 \equiv 600$.

III. Algoritmo aproximado

O algoritmo aproximado ordena decrescentemente os itens pelo valor relativo, definido na seção II.:

> valor_relativo = (somaTodasRelações(item) + valor[item]) / peso[item]

Após a ordenação, ele insere o maior item não inserido que cabe na mochila até que ela esteja cheia e toma estes itens como a solução. Essa heurística foi escolhida porque privilegia os itens com maior potencial para contribuirem com uma soma de valores próxima da ótima. A divisão pelo peso garante que o valor relativo cresça conforme o quanto da mochila que irá ocupar.

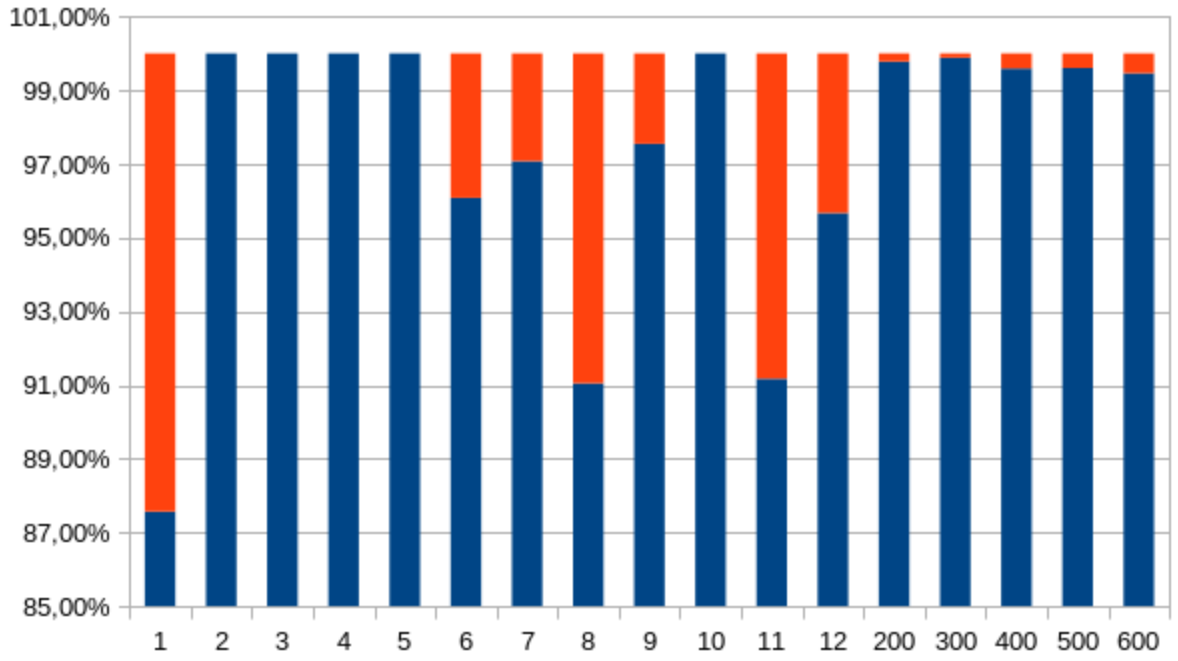
a. Implementação

A implementação do algoritmo heurístico é dada por:

> *01.* Inicia o alarme com o tempo máximo e ordena decrescentemente o vetor `itens` com os `n` itens pelo seu *valor relativo*[1]
> *02.* Inicializa `soma` como 0
> *04.* Inicializa `capacidade_parcial` com o valor da `capacidade`
> *05.* Inicializa `solucao` como um vetor vazio
> *06.* Para `ì`, de `0` até `n`:
> *07.* Se o alarme acabou, retorna a soma e solução atual
> *08.* Se `peso[i] <= capacidade_parcial`:
> *09.* Coloca `i` em `solucao`
> *10.* Adiciona `valor[i]` em `soma`
> *11.* Adiciona `valorRelacoes(i, solucao)` em `soma`
> *12.* Subtrai `peso[i]` de `capacidade`
> *13.* Retorna `soma` e `solucao`

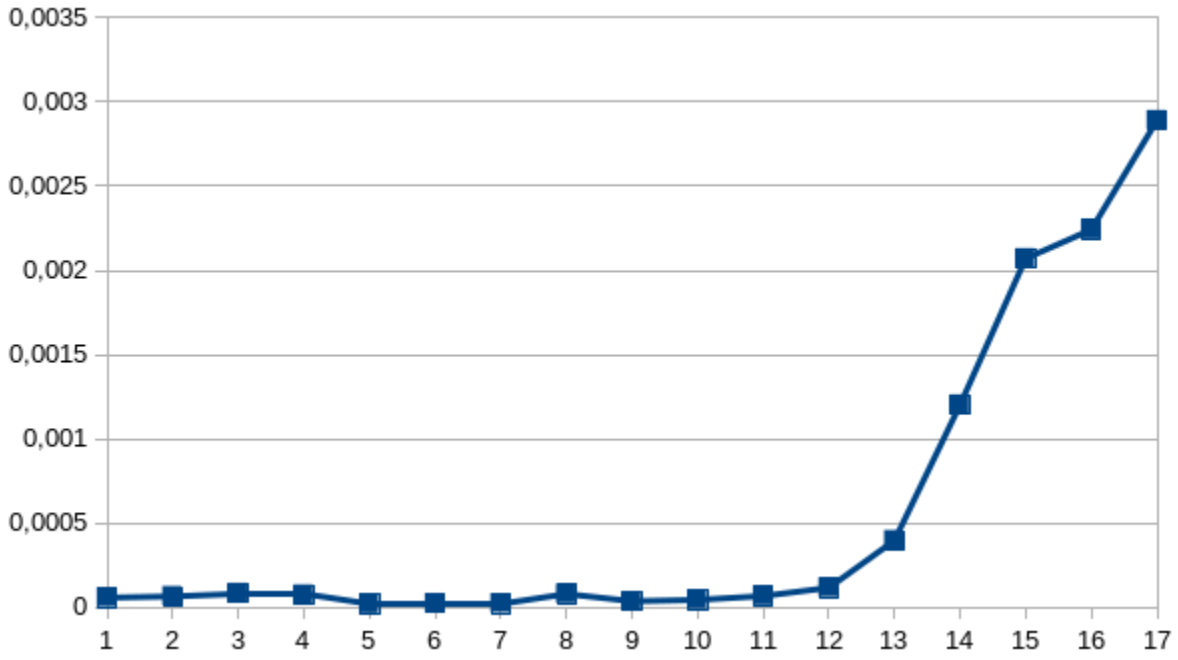
b. Resultados

Todas as entradas tiveram tempo de execução muito similar, atingindo um pico em 0,003 segundos no maior caso de teste. A precisão das respostas mostrou evoluir de maneira quase proporcional ao tamanho da entrada. Todos os casos testes grandes obtiveram 99% de precisão, e alguns pequenos atingiram 100%. A precisão mínima obtida foi de 87%. Devido a complexidade das relações no problema, é improvável que entradas muito grandes atinjam 100%.



Precisão X Caso teste

Embora o tempo de execução seja relativamente curto em comparação ao algoritmo exato, é possível observar no gráfico abaixo um comportamento quadrático (n_2) conforme a entrada cresce. Dada uma entrada suficientemente grande, o algoritmo aproximado também será muito ineficiente, sendo necessário um corte pelo tempo de execução.



Tempo de execução (seg) X Casos teste

IV. Conclusão

Para entradas pequenas, vale a pena executar os algoritmos exatos, já que estes apresentam resultados 100% precisos em tempo suficientemente pequenos, enquanto o heurístico, apesar de apresentar tempo menor, não garante precisão de 100% em todos os casos. Para as entragas grandes a escolha é usar o algoritmo aproximado, já que em pouquissimo tempo consegue retornar resultados com 99% de precisão, enquanto o algoritmo exato, mesmo com 10_6 mais tempo não conseguiu alcançar 50% de precisão para algumas entradas.

Resultados finais

Exato

Resultados finais para o algoritmo exato.
Resultado contem porcentagem de precisão, se não for 100%.

Entrada	Resultado	Tempo (seg)
1	193	0,000111
2	330	$4,4 * 10\_5$
3	260	0,000104
4	549	$7,3 * 10\_5$
5	337	$5,9 * 10\_5$
6	893	0,000287
7	787	0,000158
8	2044	0,002405
9	1875	0,000913

Entrada	Resultado	Tempo (seg)
10	1773	0,009742
11	3398	0,016658
12	7736	0,796084
200	22929	120
300	36264 (96%)	300
400	46420 (72%)	300
500	54768 (56%)	300
600	66557 (52%)	300

Heurístico

Resultados finais para o algoritmo Heurístico.

Entrada	Resultado (% Precisão)	Tempo (seg)
1	169 (87,56%)	$5,8 * 10\_5$
2	330 (100,00%)	$6,6 * 10\_5$
3	260 (100,00%)	$8,5 * 10\_5$
4	549 (100,00%)	$7,5 * 10\_5$
5	337 (100,00%)	$2 * 10\_5$
6	858 (96,08%)	$25 * 10\_5$
7	764 (97,08%)	$24 * 10\_5$
8	1861 (91,05%)	$8 * 10\_5$
9	1829 (97,55%)	$3,7 * 10\_5$

Entrada	Resultado (% Precisão)	Tempo (seg)
10	1773 (100,00%)	$4.5 * 10\_5$
11	3098 (91,17%)	$6.8 * 10\_5$
12	7401 (95,67%)	0,000117
200	22878 (99,78%)	0,000398
300	37666 (99,88%)	0,001203
400	63748 (99,59%)	0,002071
500	95808 (99,60%)	0,002242
600	126865 (99,46%)	0,00289