Universidade Federal Fluminense
Instituto de Computação
Nomes: Fernando Costa Rodrigues e Lucas Alexandre Siqueira
Análise e Projeto de Algoritmos – 2018.1

Relatório T4: análise de algoritmos para a resolução do problema de subset sum.

1 - Introdução:

Neste relatório serão abordados dois algoritmos para a resolução do problema *subset sum*, que consiste em dizer se, dados um conjunto qualquer de números inteiros V, é possível formar um subconjunto no qual a soma de seus elementos resulte em um valor dado S.

2 - Implementações:

2.1 - Algoritmo 1:

O primeiro algoritmo dado se resume em uma abordagem linear para o problema citado utilizando programação dinâmica, e além disso sua implementação foi relativamente simples visto que o mesmo utiliza apenas de poucos casos tratados com as seguintes condições:

- Se a soma atual for maior que o valor de s, e o elemento for negativo, adicione ele a soma.
- Caso ele seja positivo e a soma atual for menor que s, adicione ele a soma.
- Quando a soma for igual ao valor de s, um subconjunto foi encontrado. Se a soma nunca for igual a s, então não existe um subconjunto.

Por fim, um importante fator na implementação a ser considerado foi a utilização de uma variável utilizada para saber se a soma havia sido modificada, de forma que não caíssemos no caso onde a soma inicial é 0 pelo fato de um subconjunto vazio sempre estar contido em qualquer subconjunto, o que não deveria ser considerado de acordo com o problema.

2.1.1 - Complexidade de Tempo:

Podemos concluir que a complexidade temporal do algoritmo é O(n). Ele percorre o vetor de tamanho N de forma linear, tendo seu melhor caso quando na primeira iteração a soma que calculamos é igual a que desejávamos verificar (O(1)), e seu pior caso seria quando ele precisa percorrer todo vetor para verificar a existência ou não da soma ((O(n))

2.1.2 - Complexidade de Espaço:

O espaço realmente necessário para o funcionamento desse algoritmo é apenas das variáveis auxiliares, que na implementação em questão são *modified* e *actual*. Visto que a memória utilizada não depende do tamanho da entrada temos como complexidade de espaço O(1), ou seja, constante.

2.1.3 - Corretude:

O primeiro erro que conseguimos identificar é quando a soma (S) que desejamos verificar é 0. Com o algoritmo implementado dessa forma, ele já começa retornando verdadeiro mesmo sem verificar os elementos dos vetores, porque ele não entra em nenhuma condição e no final teremos que a soma temporária é igual a que desejávamos (igual a 0).

Lembrando que isso só é um erro se não considerarmos o conjunto vazio como um conjunto válido nesse caso.

Analisando de forma mais profunda, conseguimos identificar erros ainda mais comprometedores. A forma linear que o algoritmo é executado faz com que a distribuição dos elementos dentro do conjunto influencia diretamente o resultado, ou seja, um mesmo conjunto de elementos distribuídos em ordem diferente podem apresentar resultados diferentes. Esse é um problema grave, que faz com que nosso algoritmo funcione apenas para casos bem específicos.

2.2 - Algoritmo 2:

O segundo algoritmo dado também envolve o uso de programação dinâmica para tentar solucionar o problema, mas de forma diferente. Analisando a questão, vemos que ela é bem parecida com um problema conhecido na computação, o Subset-Sum. No caso, a única diferença do nosso problema para o Subset-Sum convencional é que em nosso caso temos um conjunto de números que pode conter tanto números positivos quanto negativos, e a forma convencional amplamente conhecida engloba apenas um conjunto contendo apenas números positivos.

Para contornar essa dificuldade pensamos em uma forma eficiente de adaptar o problema que nós não sabíamos solucionar para outro conhecido. Para isso foi necessário pensar em uma lógica para adaptar o problema sem prejudicar a corretude do algoritmo, e para exemplificar melhor nossa lógica, irei dividi-la em passos:

1 - percorrer o conjunto de entrada, e calcular a soma de todos números positivos (armazenando-a em P) e a soma de todos números negativos (armazenando-a em N). Caso a soma de subconjunto que queremos verificar seja menor que N ou maior que P, já sabemos que ela não existe. Porém, se ela estiver dentro desse intervalo, partimos para o próximo passo.

2 - O próximo passo foi criar uma função auxiliar, a qual dado um valor de um elemento do conjunto, ele retornava seu respectivo valor de índice na matriz. Nossa intenção é conseguir em C um comportamento parecido com python na questão de manipulação de vetores. Quando tivermos que acessar um índice negativo, a intenção é que comecemos a iterar pelo vetor de trás pra frente. Um exemplo seria o elemento -1. Nesse caso, olharíamos para última coluna de nossa matriz.

3 - Com essa função auxiliar, podemos representar os números positivos na matriz como sendo as colunas [0 a P], [P+1] representando 0 e [P+2 até (P-N+1)] representando os números negativos. Com isso, passamos o problema do Subset-Sum com números negativos que não sabíamos resolver para a forma convencional, conseguindo assim solucionar a questão.

No final da implementação foi observado que a forma com que o problema foi tratado pelo algoritmo era muito parecida com a forma de resolver um problema bastante conhecido no mundo da computação utilizando programação dinâmica, o problema da mochila booleana. A diferença deste que estamos resolvendo para o citado é que, ao invés de maximizarmos a solução de forma que a mesma (soma dos elementos de um sub-vetor) seja menor igual a um custo, esta seja exatamente igual a este custo.

2.2.1 - Complexidade de tempo:

A complexidade deste algoritmo pode ser facilmente descoberta visto que o mesmo trabalha em cima de uma estrutura auxiliar, neste caso uma matriz, iterando por todos os valores da mesma. Ao fim de dois laços, um indo de 0 até n+1 (n o tamanho da entrada) e outro de 0 até P-N+1 (N a soma dos valores negativos e P dos positivos), temos que sua complexidade dependerá destas três variáveis. Considerando que para cada laço de um dos tipos teremos um do outro tipo, teremos como complexidade temporal para o pior caso o produto entre ambos, de forma que obtemos O(n*(P-N+1)) ou O(n*(P-N)). É importante comentar que este algoritmo é considerado Pseudo Polinomial justamente por não depender somente do tamanho de sua entrada, mas sim do tamanho dos valores encontrados em seu decorrer.

2.2.2 - Complexidade de espaço:

A complexidade de espaço desse algoritmo está diretamente relacionada com a complexidade temporal do mesmo, isso porque o tempo gasto por ele depende diretamente do tamanho da estrutura que está sendo utilizada de forma auxiliar, logo se relacionando com o espaço de memória usado. Desta forma temos que a complexidade deste algoritmo dependerá , no pior caso, do tamanho da matriz utilizada na programação dinâmica, visto que é o uso mais significativo de memória do mesmo. Assim teremos que a complexidade de espaço será de O(n*(P-N)) novamente. É importante salientar que esse algoritmo pode ser bastante custoso com relação a memória gasta em casos onde N, P e n são muito grandes, desta forma é bom conhecer a entrada antes de utilizá-lo.

2.2.3 - Corretude:

Diferentemente do primeiro algoritmo apresentado, neste nós temos garantia de que a solução correta será encontrada, visto que em seu decorrer analisamos uma estrutura a qual possui um representação para cada valor do vetor de entrada, assim explorando todas as possibilidades possíveis, porém de uma forma mais inteligente e não repetitiva.

3- Analisando teste a teste:

3.1 - Tabela

Teste:	Algoritmo 1:	Algoritmo 2:
1	1	1
2	0	0
3	0	1
4	1	1
5	1	1
6	0	1
7	0	1

8	1	1
9	0	1
10	0	1 (*)

3.2 - Analisando os resultados

Verificando os testes, vemos que em alguns casos os algoritmos apresentam resultados diferentes para um mesmo teste. Isso se dá pelo fato de que o Algoritmo 01 (o linear) funcionar apenas para casos específicos, enquanto o Algoritmo 02 para qualquer caso (lembrando também da limitação com relação a memória desse tipo de implementação).

Para os testes 1 e 2, temos que o Algoritmo 01 conseguiu achar a resposta correta. Além disso, a quantidade de memória desnecessária de memória e processamento para esses casos simples no Algoritmo 02 é bem significativo (dado que a complexidade do input não é grande). Por isso, para esses testes, optaria pelo algoritmo 01.

No teste 3, vemos que agora o algoritmo 01 não apresentou a resposta correta, ou seja, esse caso em específico não é abrangido pela forma linear da implementação do algoritmo 01. O algoritmo 02 conseguiu se dar bem com essa situação, já que N e P serão valores pequenos, e o uso de memória possivelmente não representará um problema. Optaria pelo algoritmo 02.

No teste 4 e 5, temos que o algoritmo 01 apresenta novamente resultados corretos. Contudo, analisando os inputs, vemos que os dois testes caem em um caso onde o algoritmo 01 não engloba, que é quando queremos um subvetor com soma 0. Por coincidência os resultados bateram, mas a melhor opção para esse caso é o algoritmo 02, que além de apresentar realmente o resultado correto (baseado em uma lógica correta), também não fará uso de muita memória auxiliar nesse caso (principalmente no teste 04, onde N e P apresentam valores bem baixos).

Nos testes 6, 7, 9 e 10, vemos que agora os valores de N e P tendem a crescer, e a memória passa a se tornar um problema (*em especial para o teste 10, que para rodar fez-se necessário um computador com grande quantidade de memória). Contudo, temos que o algoritmo 01 não conseguiu ser correto para esses casos de teste, por isso para essa situação, apesar da quantidade de memória usada no processo, o algoritmo 02 é a melhor opção.

No teste 8, o algoritmo linear novamente conseguiu apresentar um resultado correto, e dessa vez não foi por coincidência. Sua lógica realmente abrange esse caso, ou seja, não temos problemas com a corretude, rodamos ele em tempo linear e não fazemos uso de quase nenhuma memória auxiliar. Por isso, optaria pelo algoritmo 01.

4 - Conclusão:

Por fim temos que o Algoritmo 01 baseia-se em heurística, de forma que não necessariamente vá encontrar o valor esperado, porém utiliza da mesma para que tenha uma eficiência temporal e espacial maior. Em contrapartida, o Algoritmo 02 apresenta o valor esperado para todos os casos, contudo é Pseudo Polinomial, ou seja, depende dos valores do elemento da entrada para determinar o quanto gastará de recursos (podemos ter uma entrada com poucos elementos, mas se esses elementos tiverem um valor elevado,

gastaríamos uma grande quantidade de recursos para resolver um problema relativamente simples, que outros algoritmos mais “burros” poderiam solucionar de forma mais eficiente). Baseando-se nestas duas informações temos que é de grande importância o conhecimento dos dados de entrada para optar entre um ou outro, com a finalidade de tentar utilizá-los de forma que nos retornem respostas corretas e eficientemente.