Universidade Federal de São Paulo - Campus São José dos Campos
Disciplina: Projeto e Análise de Algoritmos

Problema 2: Engenharia Reversa

Rosângela Miyeko Shigenari, 92334

12 de maio de 2018

1 Introdução

Este trabalho visa a resolução de um problema de engenharia reversa com a aplicação dos conceitos de *backtracking*, vistos em aula, em linguagem C/C++, com um tempo limite de 2s, e em seguida, realizar o estudo de sua complexidade.

2 Problema

Vários compostos de outro planeta foram encontrados recentemente e a composição dos compostos ainda é um mistério. Sabe-se apenas a massa desses compostos. Sem a existência de um equipamento experimental capaz de quebrar os compostos nos diferentes elementos básicos que os compõem. Baseado nas massas de um conjunto de elementos básicos conhecidos, a sua tarefa é escrever um programa que verifique, através de busca exaustiva, se é possível que o composto possa ser formado por alguns desses elementos básicos. Para esse problema, considere que cada elemento presente no composto está presente com apenas uma unidade nesse composto.

Entrada:
A entrada do problema consiste de quatro linhas. A primeira linha contém um número N ($1 \leq N \leq 20$) de possíveis elementos. A segunda linha contém N inteiros positivos com a massa de cada elemento considerado, sendo que cada elemento possui massa de até 2000, inclusive. A terceira linha contém um número M ($1 \leq M \leq 200$) de diferentes compostos a serem verificados. A quarta linha contém M inteiros positivos correspondentes a diferentes valores de massa de compostos a serem verificados, separados por espaços e cada composto podendo ter valor de massa de até 2000, inclusive.

Saída: Seu programa deve imprimir M linhas com os resultados das verificações na ordem das consultas realizadas, sendo um resultado de consulta por linha. Para cada composto, se for possível formá-lo com alguns dos elementos básicos apresentados, imprima a mensagem "yes". Caso contrário, imprima "no".

Exemplos de entrada e saída

- **Entrada**:
  5
  1 4 8 16 11
  20
  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

  Saída:

yes
no
no
yes
yes
no
no
yes
yes
no
yes
yes
yes
no
yes
yes
yes
no
yes
yes

3 Implementação da Solução

A solução se baseia na implementação da busca por força bruta (*backtracking*), sendo uma adaptação problema do subconjunto.
Para verificar se um dado composto pode ser formado a partir da combinação dos elementos dados, basta gerar subconjuntos desse conjunto de elementos, que seriam as combinações possíveis, e para cada subconjunto formado, calcular a soma desses itens e comparar se é igual a massa do composto, retornando "yes", caso positivo ou "no", caso contrário.

- **Geração de um subconjunto**
  A função *backtracking_ subconjunto* é do tipo inteiro, com os parâmetros mostrados na figura 1.

```
int backtracking_subconjunto(int conjunto[], int N, int index, int massa, int vet_solucao[], int somaNo) {
```

Figura 1: Declaração da função do *Backtracking*

Os parâmetros consistem em: um conjunto dado como entrada (o conjunto de elementos básicos inseridos pelo usuário), o número de elementos deste conjunto, uma variável *index* que percorre os níveis da árvore de soluções, a massa do composto (a ser verificado), um vetor de inteiros *vet_ solução*, que representará cada nó da árvore de soluções e a variável inteira *somaNo*, que armazena a soma dos elementos de um subconjunto formado a partir do vetor *conjunto*.
Na figura 2 podemos observar a função que retorna 1 caso o composto inserido como entrada possa ser formado com os elementos do vetor de entrada e 0 caso contrário.

Primeiramente, é realizada a verificação se o vetor dos elementos de entrada possui tamanho menor ou igual a 0, se não for, o método é aplicado. É criado uma árvore de soluções, onde cada nível representa a inserção ou não de um elemento em um subconjunto, se o elemento da posição *index* for inserido no *vet_ solucao*, a posição *index* deste vetor é preenchido com 1, senão com 0, a figura 3 mostra a esquematização da árvore de soluções.

Podemos observar que ao chegar em uma folha, obtemos um subconjunto do conjunto de entrada. Por exemplo, para a entrada dada como exemplo {1,4,8,16,11}, se o vetor *vet_ solucao* for [1,0,0,1,1], temos o subconjunto {1,16,11}.
Como pode ser observado na figura 2 realizamos chamadas recursivas a direita e a esquerda na árvore de soluções, incrementando o valor de *index* (para percorrer os níveis da árvore).
Na função, também temos um contador *somaNo*, que soma os os elementos ao passo que estão sendo

```
10  int backtracking_subconjunto(int conjunto[], int N, int index, int massa, int vet_solucao[], int somaNo) {
11      int i, result;
12      if(N>0)
13    {
14        if (somaNo == massa) {
15          aux = 1;
16          index = N;
17        }
18        else if (index!=N) {
19         vet_solucao[index] = 1;// seleciona um elemento
20         somaNo += conjunto[index];
21         if(aux!=1)//se a soma ja foi encontrada nao precisa percorrer o outro lado
22          backtracking_subconjunto(conjunto, N, index + 1, massa, vet_solucao, somaNo);
23         somaNo -= conjunto[index];
24         vet_solucao[index] = 0;// nao seleciona
25         if(aux!=1)//se a soma ja foi encontrada nao precisa percorrer o outro lado
26          backtracking_subconjunto(conjunto, N, index + 1, massa, vet_solucao, somaNo);
27       }
28
29    }
30      result = aux;
31      return result;
32  }
```

Figura 2: Função do *Backtracking*

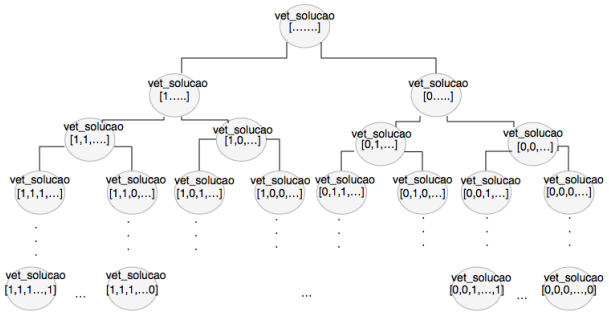

Figura 3: Árvore de soluções

"inseridos"no vetor *vet_solução*, ou seja, quando a posição *index* for marcada como 1, o valor da massa desse elemento é acrescido no contador. Durante as chamadas recursivas, se o valor de *somaNo* for igual ao número de massa do composto, uma variável global *aux* recebe o valor de 1 (a variável é declarada como global e não local para não ser alterada durante as chamadas recursivas). Ao final, a função retorna o valor de *aux*.

- **Podagem**
  A fim de diminuir o tempo de execução, é possível realizar uma poda na árvore de busca de subconjuntos (soluções) ao realizar uma inclusão de comparadores antes das chamadas recursivas, caso seja encontrada uma solução, ou seja, haja uma combinação possível entre os elementos que resulte no composto inserido como entrada, a busca é terminada não necessitando percorrer outros níveis ou lados na árvore, como pode ser observado no código da figura 2 nas linhas 21 e 25, diminuindo o tempo de execução do algoritmo.

- **Função *main***
  A função *main* esta representada na figura 4. Temos como entrada o número de elementos(*n_elementos*), um vetor *massa_e* de tamanho *n_elementos*, que armazena a massa de cada um dos elementos. Outras entradas também são o número de compostos (n_compostos) a serem analisados, e as massas desses compostos, para cada um desses compostos, a função *backtracking_subconjunto* é chamada, para realizar a verificação, retornando "yes", caso a variável global seja 1 (combinação encontrada) ou "no", caso contrário. A cada iteração a variável *aux* é zerada para que não mantenha o valor obtido anteriormente. O custo dessa iteração para inserção das massas dos compostos é O(n_compostos).

4 Análise do Algoritmo

Neste algoritmo é possível obter um pior caso, ou seja, não ser encontrada nenhuma solução, gerando assim todos os subconjuntos possiveis para ser realizada a verificação ou um caso mais favorável onde uma solução é encontrada, não necessitando realizar outras chamadas recursivas.

```c
int main()
{
  int n_elementos, n_compostos;
  int massa_c;
  int result, somaNo, i;

  scanf("%d\n",&n_elementos);

  int massa_e[n_elementos];
  int vet_aux [n_elementos];
  if(n_elementos>0)
  {
    for(i =0; i<n_elementos; i++)
    scanf("%d",&massa_e[i]);
  }

  scanf("%d", &n_compostos );
  if(n_compostos>0)
  {
    for(i = 0; i<n_compostos; i++){
      scanf("%d", &massa_c);
      aux = 0;
      result = backtracking_subconjunto(massa_e, n_elementos, 0, massa_c, vet_aux, somaNo);
      if(result==1)
        printf("yes\n" );
      else
        printf("no\n");
    }
}
```

Figura 4: Função *main*

- **Pior Caso**
  O algoritmo, sem considerar a podagem, gera todos os subconjuntos possíveis com o vetor de entrada (vetor *conjunto*), ou seja gera 2^n soluções, portanto o custo deste algoritmo, será:

$$Complexidade = \begin{cases} \theta(1), & \text{se } N \leq 0 \\ O(2^n), & \text{caso contrário} \end{cases}$$

- **Caso de Poda**
  Para este caso temos a seguinte recorrência.

$$T(indice) = \begin{cases} \theta(1), & \text{se } N \leq 0 \\ 2 * T(indice - 1), & \text{caso contrário} \end{cases}$$

  No método *backtracking* podemos observar que há 2 chamadas recursivas, uma para o lado esquerdo da árvore e outra para o esquerdo, resultando em T(indice+1) + T(indice+1) = 2*T(indice+1) para valores de N (número de elementos básicos de entrada) maiores que 0. O custo dependerá em qual nível (valor de *indice*) a poda ocorrerá, podendo variar de 0 até N. Se *indice* for igual a N resultará no pior caso.