Merkle-Hellman cryptosystem

Algoritmos e Estruturas de Dados

Professor Tomás Oliveira e Silva

Professor Pedro Lavrador

Professor Joaquim Silvestre Madeira

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

João Pedro Ferreira Monteiro

(102690) joao.mont@ua.pt

Índice

Introdução

Este relatório foi realizado no âmbito de reportar os resultados do **trabalho prático número 1** da cadeira do 2º ano, **Algoritmos e Estruturas de Dados** (**AED**). O objetivo deste trabalho centrou-se na resolução de um problema de somas de subconjuntos, através da utilização do criptossistema público de *Merkle-Hellman*. Para isso, foi dado a cada estudante um arquivo com o nome a conter o número mecanográfico do aluno e que tem a extensão *.h* com **20 somas** obtidas através da soma de termos de uma **sequência *p*** para cada um dos **n**, contidos entre **10** e **57**. O objetivo do problema é descobrir quais dos termos foram utilizados para obter cada uma das somas, através de quatro métodos diferentes.

Métodos Utilizados

1º Método - Brute Force

Este é o primeiro método pedido para resolver o problema. Como o nome sugere, é através de *brute force*, conhecido em português como abordagem exaustiva ou de força bruta. É o método mais lento, comparativamente aos próximos, pois tem de percorrer todas as opções para encontrar a solução certa.

O que o código faz é receber como argumentos da função recursiva *brute force*: um **n** inteiro que indica o número do problema em que estamos; o vetor **p,** que aponta para o array de inteiros do problema; a **desired_sum,** que é a soma a que queremos chegar; um inteiro **next_index,** que indica o ponto de partida do ciclo na iteração atual; a **partial_sum**, que indica quando chegamos à soma desejada, e o vetor **b** que é onde vão ser colocados os 0 ou 1 para formar a sequência binária da solução do problema.

Observação: Para que o código não ficasse demasiado extenso, foram retirados os prints para levantamento de dados, os comentários e a variável de controle de tempo que usa a função fornecida nas aulas cpu_time, no entanto, estes encontram-se presentes no código final.

```
int main(void) {
    for (int prob = 0; prob < n_problems; prob++){
        for (int sum = 0; sum < n_sums; sum++){
            int b[all_subset_sum_problems[prob].n];
            brute_force(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum], 0,
0, b);
            for(int index = 0; index < all_subset_sum_problems[prob].n;
index++){fprintf(f, "%d", b[index]);}
        }
    }
    return 0;
}
```

Código 1: Chamada da função brute_force na main.

```
int brute_force(int n,integer_t *p, integer_t desired_sum,int
next_index,integer_t partial_sum, int *b){
        if(desired_sum==partial_sum){
          for(int i=next_index;i<n;i++){ b[i]=0;}
          return 1;}
        if(next_index==n) return 0;
        b[next_index]=0;
```

```
        int result = brute_force(n,p,desired_sum,
next_index+1,partial_sum,b);
        if(result==1){return 1;}
        b[next_index]=1;
        return
brute_force(n,p,desired_sum,next_index+1,partial_sum+p[next_index],b);}
```

Código 2: Função brute_force

A função **brute_force** começa por comparar os valores dos argumentos **desired_sum** e **partial_sum** e, caso estes sejam iguais, os índices restantes serão preenchidos com 0 e é retornado 1, dado que a soma que foi encontrada é igual à pretendida. Caso o índice utilizado na iteração atual iguale o valor de **n**, a função irá devolver o valor 0 devido a não ter encontrado a solução pretendida.

Por fim, se nenhuma das condições se verificar, é atribuído o valor 0 e é chamada novamente, para result, a função **brute_force** com o incremento no **next_index**. Caso o valor de **result** seja igual a 1, a iteração atual retorna o mesmo. Se tal não se verificar, chamar-se-á novamente a função **brute_force** com incremento de 1 em **next_index** e adicionar-se-á ao valor da **partial_sum** o valor em **p[next_index]**.

2º Método - Clever Brute Force (Branch and bound)

O segundo método, como o nome sugere, é também através de *brute force*, mas, desta vez, "esperta" (*clever*). Este método evita que a recursão continue quando já é possível descobrir a solução do problema, não sendo necessário analisar todas as somas do problema.

O que o código faz é receber como argumentos da função recursiva *brute_force*: um **n** inteiro que indica o número do problema em que estamos; o vetor **p,** que aponta para o array de inteiros do problema; a **desired_sum**, que é a soma a que queremos chegar; um inteiro **next_index,** que indica o ponto de partida do ciclo na iteração atual; a **partial_sum**, que indica quando chegamos à soma desejada, e o vetor **b** que é onde vão ser colocados os 0 ou 1 para formar a sequência binária da solução do problema.

```
int main(void) {
  for (int prob = 0; prob < n_problems; prob++){
    for (int sum = 0; sum < n_sums; sum++){
      int b[all_subset_sum_problems[prob].n];
      branch_and_bound(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum], 0,
0, b);
      for(int index = 0; index < all_subset_sum_problems[prob].n;
index++){fprintf(f, "%d", b[index]);}
        }
      }
    return 0;
}
```

Código 3: Chamada de branch_and_bound na função main

```
int branch_and_bound(int n, integer_t *p, integer_t desired_sum, int
next_index, integer_t partial_sum, int *b){
    if(desired_sum == partial_sum){
        for(int i = next_index; i < n; i++){
            b[i] = 0;
        }
        return 1;
    }
    if(next_index == n) return 0;
    if(partial_sum > desired_sum){
        return 0;
    } else {
```

```
        integer_t remaining_sum = 0;
        for(int k = next_index; k < n; k++){
            remaining_sum += p[k];
        }
        if((partial_sum + remaining_sum) < desired_sum){
            return 0;
        } else {
            // set next bit to zero
            b[next_index] = 0;
                            int  result  =  branch_and_bound(n,p,desired_sum,
next_index+1,partial_sum,b);
            if(result == 1) return 1;
            // set next bit to one
            b[next_index] = 1;
                                                                        return
branch_and_bound(n,p,desired_sum,next_index+1,partial_sum+p[next_index],b);
        }
    }
}
```

Código 4: Função branch_and_bound

Na função ocorre, inicialmente, a verificação, tanto da **partial_sum**, como da **next_index** em semelhança ao que foi feito na função *brute force*. De seguida, verifica-se se a **partial_sum** excede a **desired_sum** e, caso isto aconteça, a função retorna 0. Contrariamente, criamos a variável **remaining_sum**, cujo valor corresponde à soma dos elementos de **p** do índice dado por **next_index** até ao último. Posteriormente, averigua-se se a soma da **partial_sum** com a **remaining_sum** é menor do que **desired_sum**, a eventualidade de a condição ser verdadeira, significa que este valor não vai ser atingido com os elementos de **p** e, portanto, a função retorna 0.

3º Método - Horowitz and Sahni technique (meet-in-the-middle)

O terceiro método, é conhecido como *método de Horowitz e Sahni*, no entanto, em prol de facilitismo e comodidade na escrita, no decorrer do relatório, referir-me-ei a este como *meet-in-the-middle* .

O que o código faz é receber como argumentos da função *meet_in_the_middle*: um **n** inteiro, que indica o número do problema em que estamos; o vetor **p,** que aponta para o array de inteiros do problema, e o vetor **sums,** que indica a soma dentro de n.

```
meet_in_the_middle(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum]);
```

Código 5: Chamada da função meet_in_the_middle dentro da função main

Partindo para a explicação do método, este começa por dividir o array **p** de cada problema **n** em dois arrays de tamanhos iguais ou semelhantes, **p1** e **p2**.

- ➔ Quando número de termos em **p** for par:

```
sizeof(*p1)==sizeof(*p2)
```

- ➔ Quando o número de termos em **p** for ímpar:

```
sizeof(*p1)=sizeof(*p2)-1
```

Com os arrays divididos, será descoberto o número de somas possíveis entre termos de cada array. Por exemplo, se o array **p1** tiver **tamanho 3**, vai ter termos na posição *0, 1, 2*. Para descobrir as possíveis somas, em primeiro lugar, é necessário saber o número de casos possíveis, que é $2^{(\text{tamanho do array } \mathbf{p1})}$.

```
int na = pow(2, size1);
int nb = pow(2, size2);
```

Código 6: Descobrir quantidade de possíveis somas

Depois, com as funções auxiliares gera-se para cada array as somas possíveis, as quais são guardadas em arrays **a** e **b**.

```
int insertIntoArray(long int *arr, int n, integer_t *p, integer_t *array)
{
    long long int a = 0;
    for (long long int i = 0; i < n; i++)
    {
        if (arr[i] == 1)
        {
            a = a + p[i];
        }
    }
    /*printf(" - > sum : %d",a);
    array[controlo] = a;
```

```
    controlo = controlo + 1;
    return a;
}

void gerarStringBinaria(int n, long int arr[], int i, integer_t *p, integer_t
*array)
{
    if (i == n)
    {
        insertIntoArray(arr, n, p, array);
        return;
    }
    arr[i] = 0;
    gerarStringBinaria(n, arr, i + 1, p, array);
    arr[i] = 1;
    gerarStringBinaria(n, arr, i + 1, p, array);
}
```

Código 7: Descobrir as somas

Seguidamente, é feita a ordenação, por valor, das somas dos arrays **a** e **b** para cada índice através da função **sortArray** e **compare** e, ao mesmo tempo, é feito um *backup* do array original **a** e **b** nos arrays do tipo **integer_t d** e **f**, que serão utilizados mais à frente para encontrar a sequência binária final da solução.

```
int compare(const void *s1, const void *s2)
{

    if (*(integer_t *)s1 < *(integer_t *)s2)
        return -1;
    if (*(integer_t *)s1 > *(integer_t *)s2)
        return 1;
    return 0;
}

void sortArray(int n, integer_t *a, integer_t *g)
{
    for (int i = 0; i < n; i++)
    {
        g[i] = a[i];
    }
    qsort(a, n, sizeof(integer_t), compare);
}
```

Código 8: Sort dos arrays a e b e criação de um backup em d e f

Por fim, dentro da função *meet_in_the_middle*, no ciclo **do while**, é somado o valor de **a[x]** e **b[nb-1-y]**. Se o valor for maior que **desired_sum**, significa que o valor da soma desejada é menor que o da soma obtida, logo, adiciona-se **1** ao **y** para baixar uma posição no índice do array **b** e soma-se 1 ao **controlo_while**, que atua como variável de controlo do ciclo. Se o valor de **a[x]** e **b[nb-1-y]** for menor que o da **desired_sum** é somado **1** ao **x** para que este suba uma posição no índice do array **a** e adiciona-se, também, 1 ao **controlo_while**. Finalmente, se **a[x]** e **b[nb-1-y]** igualar a **desired_sum**, salva-se as posições finais dentro dos arrays **a** e **b**, que irão corresponder ao valor de **x** e **posB = nb - 1 - y** e abandona-se o ciclo **do while**.

```
long int x = 0;
long int y = 0;
long int i = 0;
long int controlo_while = 1;
long int salvar_posB;
do
{
    long int posB = nb - 1 - y;
    if ((a[x] + b[posB]) > desired_sum)
    {
        y++;
        controlo_while++;
    }
    if ((a[x] + b[posB]) < desired_sum)
    {
        x++;
        controlo_while++;
    }
    if (a[x] + b[posB] == desired_sum)
    {
        salvar_posB = posB;
        break;
        controlo_while++;
    }
} while (i < controlo_while);
```

Código 9: Descobrir valores de a[x] e b[posB] onde a sua soma iguala a desired_sum

A seguir, encontrar-se-ão os índices correspondentes para os valores de **a[x]** e **b[posB]** nos arrays **d** e **f**, que serão salvos nas variáveis **save_d** e **save_f**.

```
int save_d = 0;
int save_f = 0;
for (int i = 0; i < na + 1; i++){
```

```
      if (d[i] == a[x]){save_d = i;}
        if (f[i] == b[salvar_posB]){save_f = i;}
      }
```

Código 10: Descobrir posições de a[x] e b[posB] em d e f

No final, os valores **save_d** e **save_f**, serão transformados em binários através da chamada da função *decToBinary* dentro da *meet_in_the_middle* para obter a solução do problema. Em seguida, utiliza-se a função *free* para dar *deallocate* à memória dos arrays utilizados e dá-se *reset* ao controlo.

```
decToBinary(save_d, size1 - 1);
decToBinary(save_f, size2 - 1);
    free(a);
    free(b);
    free(p1);
    free(p2);
    free(d);
    free(f);
    controlo = 0;
```

Código 11: Chamada da decToBinary para obter os valores binários da solução e dar reset às variáveis utilizadas

```
void decToBinary(int n, int maximo)
{
    for (int i = maximo; i >= 0; i--)
    {
        int k = n >> i;
        if (k & 1)
        {
            printf("1"); // print na consola
            fprintf(h, "1"); // print no ficheiro
        }
        else
        {
            printf("0"); // print na consola
            fprintf(h, "0"); // print no ficheiro
        }
    }
}
```

Código 12: Função decToBinary

Resultados

Para os três métodos foram realizados testes com duas entradas diferentes. Primeiro, com o modelo contido no ficheiro *000000.h,* através da correspondência dos resultados obtidos com os dados, ser averiguado o funcionamento do programa. O segundo teste efetuado para cada método foi a utilização do ficheiro com os problemas correspondentes ao número mecanográfico do aluno, no caso, *102690.h*.

Para auxiliar no levantamento dos dados durante os testes, foram criados três ficheiros: um que levantava os tempos para cada problema, outro que levantava o problema e, por fim, um ficheiro que englobava todos os dados e a solução para cada soma dentro dos problemas.

Com os dados levantados e organizados, foram elaborados gráficos que expõem a evolução do **tempo de resolução** de cada problema em função do problema **n** dado.

Resultados do método Brute Force

Gráfico 1: Brute Force para 000000.h

Gráfico 2: Brute Force para 102690.h

problema	tempos em segundos 000000.h	tempos em segundos 102690.h
10	0	0
11	0	0
12	0	0
13	0,015625	0
14	0	0
15	0	0
16	0	0
17	0	0,015625
18	0	0
19	0,015625	0,03125
20	0,015625	0,03125
21	0,0625	0,078125
22	0,140625	0,125
23	0,21875	0,21875
24	0,609375	0,53125
25	0,828125	1,125
26	1,828125	2,1875
27	3,90625	3,484375
28	7,765625	9,484375
29	12,65625	17,03125
30	36,375	29,96875
31	74,59375	69,015625
32	128,65625	158,21875
33	173,203125	263,328125
34	513,09375	472,25

Tabela 1: Dados levantados da função Brute Force para 000000.h e 102690.h

Analisando os gráficos do tempo de execução das funções em função **n** do problema dado e a tabela final, para a função **Brute Force**, conclui-se que, com o aumentar do **n,** aumenta exponencialmente o tempo de execução, tornando-se o resultado pretendido, a partir do **n = 35**, bastante demorado de ser calculado, já que este vai se manter na ordem de **10^3 segundos** ou superior.

Resultados do método Branch and Bound

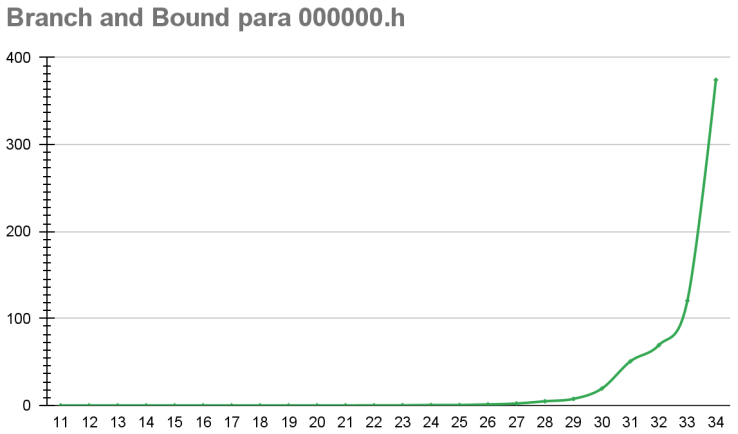

Gráfico 3: Branch and Bound para 000000.h

Gráfico 4: Branch and Bound para 000000.h

problema	tempos em segundos 000000.h	tempos em segundos 102690.h
10	0	0,015625
11	0	0
12	0	0
13	0	0

14	0	0
15	0,015625	0
16	0	0
17	0	0
18	0	0
19	0	0,015625
20	0,03125	0,03125
21	0,046875	0,046875
22	0,09375	0,0625
23	0,171875	0,171875
24	0,484375	0,40625
25	0,484375	0,765625
26	1,15625	1,375
27	2,203125	2,40625
28	4,796875	7,5
29	7,515625	10,015625
30	19,4375	19,953125
31	50,78125	50,125
32	69,359375	90,703125
33	120,453125	183,15625
34	374,34375	263,3125

Tabela 2: Dados levantados da função Branch and Bound para 000000.h e 102690.h

Analisando, agora, os gráficos do tempo de execução das funções em função **n** do problema dado e a tabela final, para a função **Branch and Bound**, conclui-se que, tal como na função **Brute Force**, com o aumentar do **n,** aumenta exponencialmente o tempo de execução, tornando-o, para valores superiores, também bastante demorado. No entanto, como este método é uma melhoria do **Brute Force**, mostra-se mais rápido a resolver os problemas.

Resultados do método Meet in the Middle

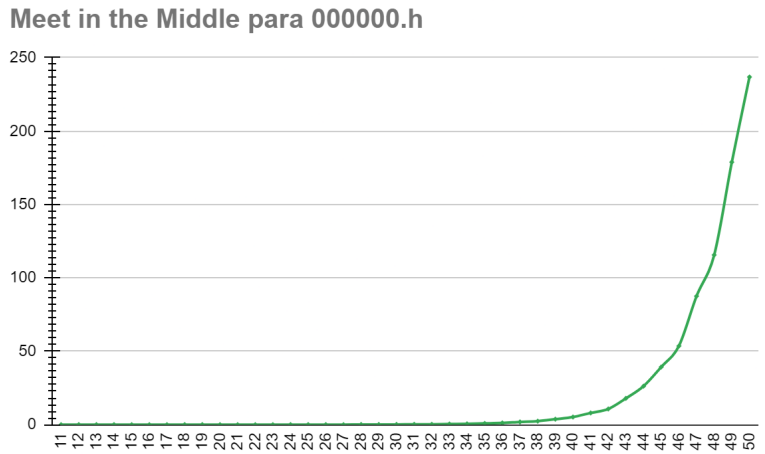

Gráfico 5: Meet in the Middle para 000000.h

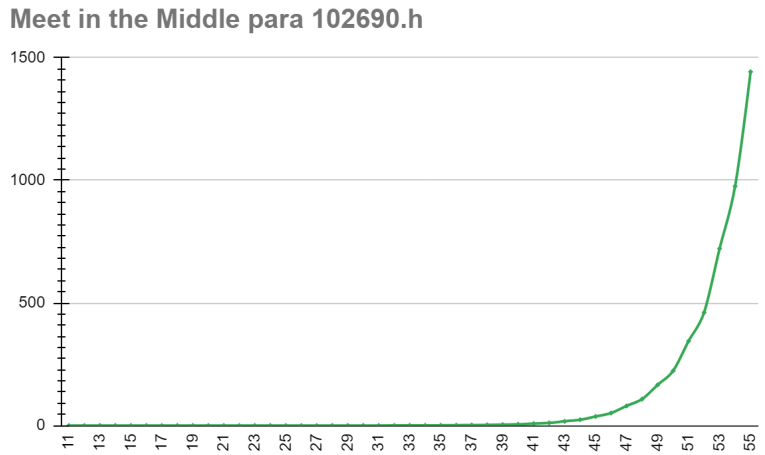

Gráfico 6: Meet in the Middle para 102690.h

problema	tempos em segundos 000000.h	tempos em segundos 102690.h
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0,015625
16	0	0
17	0	0
18	0	0
19	0,015625	0

20	0	0
21	0	0,015625
22	0,015625	0
23	0	0,015625
24	0,015625	0,015625
25	0,03125	0,03125
26	0,015625	0,03125
27	0,03125	0,046875
28	0,0625	0,0625
29	0,09375	0,078125
30	0,125	0,125
31	0,203125	0,171875
32	0,234375	0,265625
33	0,40625	0,421875
34	0,53125	0,546875
35	0,84375	0,859375
36	1,1875	1,1875
37	1,84375	1,859375
38	2,390625	2,453125
39	3,703125	3,6875
40	5,15625	4,984375
41	7,96875	7,953125
42	10,671875	10,71875
43	17,90625	17,296875
44	26,234375	23,90625
45	39,265625	36,984375
46	53,578125	51,1875
47	87,5	80,265625
48	115,546875	107,78125
49	178,796875	165,75
50	236,84375	223,09375
51	sem dados	344,734375
52	sem dados	461,046875
53	sem dados	721,65625
54	sem dados	975,890625
55	sem dados	1442,8125

Tabela 2: Dados levantados da função Meet in the Middle para 000000.h e 102690.h

Relativamente aos gráficos do tempo de execução das funções em função **n** do problema dado e a tabela final, para a função **Meet in the Middle**, conclui-se que, com o aumentar do **n,** aumenta o tempo de execução, mas de forma menos acentuada do que nos métodos anteriores, fazendo com que os valores dos tempos de execução permaneçam na ordem dos $\mathbf{10^2}$ **segundos** até ao **n=54** e, daí para a frente, na ordem de $\mathbf{10^3}$ **segundos**.

Comparação dos métodos utilizados

Gráfico 7: Comparação de todos os dados obtidos para 000000.h

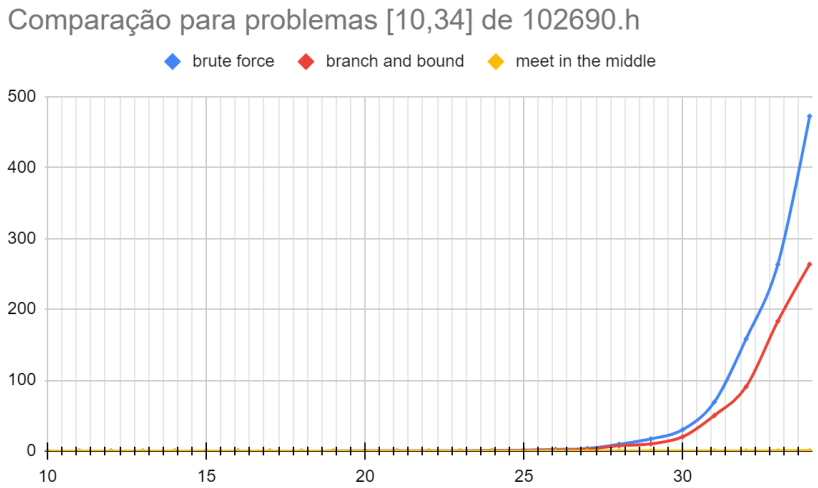

Gráfico 8: Comparação de todos os dados obtidos para 102690.h

Observando os gráficos com os resultados obtidos para os três métodos para os problemas de **000000.h** e **102690.h**, é possível concluir que o método **Meet in the Middle** é o que obtém melhores resultados para o tempo de execução em função do **n** problema a ser resolvido.

Código Final

```
//
// AED, November 2021
//
// Solution of the first practical assignement (subset sum problem)
//
// Place your student numbers and names here
// João Pedro Ferreira Monteiro   102690
#if __STDC_VERSION__ < 199901L
#error "This code must must be compiled in c99 mode or later (-std=c99)" //
to handle the unsigned long long data type
#endif
#ifndef STUDENT_H_FILE
#define STUDENT_H_FILE "102690.h"
#endif
#include <stdio.h>
#include <stdlib.h>
#include "elapsed_time.h"
#include STUDENT_H_FILE
#include <math.h>
FILE *h;
FILE *f;
FILE *t;
FILE *no;
long long int controlo;
// 1. FUNÇÃO RECURSIVA BRUTE FORCE
int brute_force(int n, integer_t *p, integer_t desired_sum, int next_index,
integer_t partial_sum, int *b)
{
    if (desired_sum == partial_sum)
    {
        for (int i = next_index; i < n; i++)
        {
            b[i] = 0;
        }
        return 1;
    }
    if (next_index == n)
        return 0;
    // set next bit to zero
    b[next_index] = 0;
    int result = brute_force(n, p, desired_sum, next_index + 1, partial_sum,
b);
    if (result == 1)
```

```
    { return 1;}
    // set next bit to one
    b[next_index] = 1;
    return brute_force(n, p, desired_sum, next_index + 1, partial_sum +
p[next_index], b);
}
// 2. BRANCH AND BOUND
int branch_and_bound(int n, integer_t *p, integer_t desired_sum, int
next_index, integer_t partial_sum, int *b)
{
    if (desired_sum == partial_sum)
    {
        for (int i = next_index; i < n; i++)
        {
            b[i] = 0;
        }
        return 1;
    }
    if (next_index == n)
        return 0;
    if (partial_sum > desired_sum)
    {
        return 0;
    }
    else
    {
        integer_t remaining_sum = 0;
        for (int k = next_index; k < n; k++)
        {
            remaining_sum += p[k];
        }
        if ((partial_sum + remaining_sum) < desired_sum)
        {return 0;}else{
            // set next bit to zero
            b[next_index] = 0;
            int result = branch_and_bound(n, p, desired_sum, next_index + 1,
partial_sum, b);
            if (result == 1)
                return 1;
            // set next bit to one
            b[next_index] = 1;
            return branch_and_bound(n, p, desired_sum, next_index + 1,
partial_sum + p[next_index], b);
        }
    }
}

// 3. MEET IN THE MIDDLE
```

```
// FUNÇÃO QUE PRINTA OS BINÁRIOS
void decToBinary(int n, int maximo)
{
    for (int i = maximo; i >= 0; i--)
    {
        int k = n >> i;
        if (k & 1)
        {
            printf("1");
            fprintf(h, "1");
        }
        else
        {
            printf("0");
            fprintf(h, "0");
        }
    }
}
int insertIntoArray(long int *arr, int n, integer_t *p, integer_t *array)
{
    long long int a = 0;
    for (long long int i = 0; i < n; i++)
    {
        // printf("%d", arr[i]);
        if (arr[i] == 1)
        {
            a = a + p[i];
        }
    }
    /*printf(" - > sum : %d",a);
    printf("\n");
    printf("PARA %d - >", a);*/
    array[controlo] = a;
    // printf(" ARRAY[%d] : %lld\n",controlo, array[controlo]);
    controlo = controlo + 1;
    return a;
}
void gerarStringBinaria(int n, long int arr[], int i, integer_t *p, integer_t
*array)
{
    if (i == n)
    {
        insertIntoArray(arr, n, p, array);
        return;
    }
    arr[i] = 0;
    gerarStringBinaria(n, arr, i + 1, p, array);
    arr[i] = 1;
```

```
    gerarStringBinaria(n, arr, i + 1, p, array);
}
// QSORT
int compare(const void *s1, const void *s2)
{

    if (*(integer_t *)s1 < *(integer_t *)s2)
        return -1;
    if (*(integer_t *)s1 > *(integer_t *)s2)
        return 1;
    return 0;
}

void sortArray(int n, integer_t *a, integer_t *g)
{
    for (int i = 0; i < n; i++)
    {
        g[i] = a[i];
    }
    qsort(a, n, sizeof(integer_t), compare);
}

// FUNÇÃO PRINCIPAL
int meet_in_the_middle(int n, integer_t *p, integer_t desired_sum)
{
    int size1 = n / 2;     // size 2
    int size2 = n - size1; // size 2
    //  reservar memória
    integer_t *p1 = (integer_t *)malloc(size1 * sizeof(integer_t));
    integer_t *p2 = (integer_t *)malloc(size2 * sizeof(integer_t));

    for (int i = 0; i < size1; i++)
    {
        p1[i] = p[i];
        p2[size2 - 1 - i] = p[n - 1 - i];     }
    if (n / 2 != 0)
        p2[0] = p[size1];
    int na = pow(2, size1);
    int nb = pow(2, size2);
    integer_t *a = (integer_t *)malloc(na * sizeof(integer_t));
    integer_t *b = (integer_t *)malloc(nb * sizeof(integer_t));
    long int arr[size1];
    // printf("A: ");
    gerarStringBinaria(size1, arr, 0, p1, a);

    // BACKUP DO ARRAY PARA ENCONTRAR O BINARIO
    integer_t *d = (integer_t *)malloc(na * sizeof(integer_t)); // backup de
a
```

```
    integer_t *f = (integer_t *)malloc(nb * sizeof(integer_t)); // backup de
b

    sortArray(na, a, d);
    controlo = 0; // resetar a contagem
    long int arr2[size2];

    gerarStringBinaria(size2, arr2, 0, p2, b);
    sortArray(nb, b, f);
    long int x = 0;
    long int y = 0;
    long int i = 0;
    long int controlo_while = 1;
    long int salvar_posB;
    do
    {
        long int posB = nb - 1 - y;
        if ((a[x] + b[posB]) > desired_sum)
        {
            y++;
            controlo_while++;
        }
        if ((a[x] + b[posB]) < desired_sum)
        {
            x++;
            controlo_while++;
        }
        if (a[x] + b[posB] == desired_sum)
        {
            salvar_posB = posB;
            break;
            controlo_while++;
        }
    } while (i < controlo_while);

    // ENCONTRAR O BINARIO DO VALOR DE A[X] E B[POSB]
    int save_d = 0;
    int save_f = 0;
    for (int i = 0; i < na + 1; i++)
    {
        if (d[i] == a[x])
        {
            save_d = i;
        }
        if (f[i] == b[salvar_posB])
        {
            save_f = i;
        }
```

```
    }

    decToBinary(save_d, size1 - 1);
    decToBinary(save_f, size2 - 1);
    printf("\n");     // print na consola
    fprintf(h, "\n"); // print no ficheiro
    free(a);
    free(b);
    free(p1);
    free(p2);
    free(d);
    free(f);
    controlo = 0;
    return 0;
}

int main(void)
{
    // 1 . ------------------------------------------------
    // DESCOMENTAR PARA TESTAR BRUTE FORCE RECURSIVA
    /*
    f = fopen("brute_force_solution_102690.txt", "a");
    // para fazer os gráficos
    t = fopen("tempos_brute_force_solution_102690.txt", "a");
    no = fopen("problemas_brute_force_solution_102690.txt", "a");
    for (int prob = 0; prob < n_problems; prob++)
    { // (int prob = 0; prob < n_problems; prob++)
        fprintf(f, "n -> %d\n", all_subset_sum_problems[prob].n);
        printf("n -> %d\n", all_subset_sum_problems[prob].n);
        double tempo = cpu_time();

        for (int sum = 0; sum < n_sums; sum++)
        {
            int b[all_subset_sum_problems[prob].n];
            brute_force(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum], 0,
0, b);
            for (int index = 0; index < all_subset_sum_problems[prob].n;
index++)
            {
                fprintf(f, "%d", b[index]);
                printf("%d", b[index]);
            }
            fprintf(f, "\n");
            printf("\n");
        }

        tempo = cpu_time() - tempo;
```

```
        fprintf(f, "%d -> %0.12f segundos\n",
all_subset_sum_problems[prob].n, tempo);
        printf("%d -> %0.12f segundos\n", all_subset_sum_problems[prob].n,
tempo);
        fprintf(t, "%0.10f\n", tempo);
        fprintf(no, "%d\n", all_subset_sum_problems[prob].n);
    }
    */
    // 2 . ------------------------------------------------
    // DESCOMENTAR PARA TESTAR BRANCH AND BOUND
    /*
    f = fopen("branch_and_bound_solution_102690.txt", "a");
    // para fazer os gráficos
    t = fopen("tempos_branch_and_bound_solution_102690.txt", "a");
    no = fopen("problemas_branch_and_bound_solution_102690.txt", "a");
    for (int prob = 0; prob < n_problems; prob++)
    { // (int prob = 0; prob < n_problems; prob++)
        fprintf(f, "n -> %d\n", all_subset_sum_problems[prob].n);
        printf("n -> %d\n", all_subset_sum_problems[prob].n);
        double tempo = cpu_time();

        for (int sum = 0; sum < n_sums; sum++)
        {
            int b[all_subset_sum_problems[prob].n];
            branch_and_bound(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum], 0,
0, b);
            for (int index = 0; index < all_subset_sum_problems[prob].n;
index++)
            {
                fprintf(f, "%d", b[index]);
                printf("%d", b[index]);
            }

            fprintf(f, "\n");
            printf("\n");
        }
        tempo = cpu_time() - tempo;
        // dados consola + ficheiros
        printf("%d -> %0.12f segundos\n", all_subset_sum_problems[prob].n,
tempo);
        fprintf(f, "%d -> %0.12f segundos\n",
all_subset_sum_problems[prob].n, tempo);
        fprintf(t, "%0.10f\n", tempo);
        fprintf(no, "%d\n", all_subset_sum_problems[prob].n);
    }
    */
    // 3 . ------------------------------------------------
```

```
    // DESCOMENTAR PARA TESTAR MEET IN THE MIDDLE
    /*
    h = fopen("meet_in_the_middle_solution_102690.txt", "a");
    // para levantar dados para os gráficos
    t = fopen("tempos_meet_in_the_middle_102690.txt", "a");     // salvar
tempos
    no = fopen("problemas_meet_in_the_middle_102690.txt", "a"); // salvar
problema
    for (int prob = 0; prob < n_problems; prob++)               // (int prob
= 0; prob < n_problems; prob++)
    {
        fprintf(h, "n -> %d\n", all_subset_sum_problems[prob].n);
        double tempo = cpu_time();
        printf("n -> %d\n", all_subset_sum_problems[prob].n);
        for (int sum = 0; sum < n_sums; sum++)
        {
            double tempo_indivual = cpu_time();
            meet_in_the_middle(all_subset_sum_problems[prob].n,
all_subset_sum_problems[prob].p, all_subset_sum_problems[prob].sums[sum]);
            tempo_indivual = cpu_time() - tempo_indivual;
        }
        tempo = cpu_time() - tempo;
        fprintf(no, "%d\n", all_subset_sum_problems[prob].n);
        fprintf(t, "%0.10f\n", tempo);
        fprintf(h, "%d -> %0.12f segundos\n",
all_subset_sum_problems[prob].n, tempo);
        printf("%d -> %0.12f segundos\n", all_subset_sum_problems[prob].n,
tempo);
    }
    // printf("%d -> %0.12f segundos\n", all_subset_sum_problems[47].n,
tempo);
    */
    // ------------------------------------------------
    fclose(f);
    fclose(t);
    fclose(no);
    fclose(h);
    return 0;
}
```

Conclusão

Para terminar, posso afirmar que consegui implementar uma solução para o problema proposta de forma eficiente, uma vez que obtive os resultados pretendidos através do teste com os problemas fornecidos no ficheiro **000000.h** e comparação com os resultados que estavam no código comentado.

Quanto ao quarto método que foi pedido, não consegui encontrar uma maneira de resolver os problemas ao utilizar *heaps*.

Em suma, com as funções que desenvolvi para cada método e face ao que me foi pedido pelos professores para resolver o problema, consegui completar as três primeiras tarefas com sucesso e obter resultados favoráveis.

Bibliografia

Para a realização deste trabalho consultei os slides teóricos e os guiões práticos resolvidos durante as aulas práticas, assim como os seguintes sites. Para complementar utilizei também os seguintes sites:

QuickSort - GeeksforGeeks

C Files I/O: Opening, Reading, Writing and Closing a file