Algoritmos e Estruturas de Dados 2021/2022

Subset-Sum-Problem

Primeiro Trabalho Prático da disciplina de AED.
Professores: Tomás Oliveira e Silva; Pedro Lavrador

Trabalho Realizado por: 102435 Rafael Remígio 50%;
104360 João Correia 50%;

universidade de aveiro

Índice

Introdução

O sistema criptográfico Merkle-Hellman proposto em 1974 foi um dos mais antigos sistemas criptográficos de chave pública e era baseado no problema **knapsack,** mais especificamente no **problema subset-sum.** Em 1981, foi publicado por **Adi Shamir** ***"A Polynomial-Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem"*** mostrando que o Sistema criptográfico não era seguro.

No nosso trabalho o Sistema criptográfico é substituído pelo Subset-Sum Problem, um caso especial do problema Knapsack. O problema envolve, a partir de conjunto ordenado de inteiros positivos C e de um valor inteiro K, descobrir qual combinação de constituintes de C tem soma igual a K.

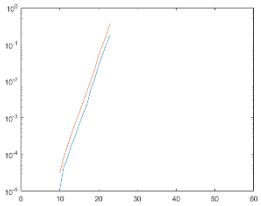
Neste relatório explicamos e comparamos diferentes métodos e algoritmos capazes de solucionar o problema.

Brute Force Iterativo

Um método Brute Force consiste numa busca exaustiva da chave. Neste caso simplesmente iteramos por todas as combinações possíveis até eventualmente encontrarmos a combinação de algoritmos que iguala a soma desejada.

1. Geramos a combinação iterando por todas as combinações (2^n)
2. Iteramos pelo conjunto somando os inteiros correspondentes a combinação
3. Comparamos esta soma com o soma desejada

Este método tem o benefício de ser fácil de implementar e de eventualmente chegar sempre a uma solução, porém é muito demoroso tendo a maior complexidade algorítmica de todos os algoritmos de $O(n*2^n)$.

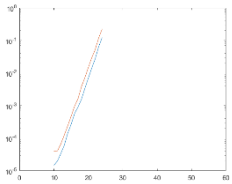


Brute Force Recursiva

Este método consiste também numa busca intensiva onde percorremos todas as combinações possíveis, no entanto neste caso a chave da combinação e a soma são geradas através de um algoritmo recursivo de “backtracking”. Consideremos o conjunto de inteiros ordenados P e a soma desejada K.

1. Consideremos que temos uma árvore onde a cada ramo é o valor de Soma + $C[x_i]*0$ ou Soma + $C[x_i]*1$ onde i é o nível da árvore. Neste caso iremos gerar todos os 2^n ramos comparando pelo caminho a soma de cada iteração e aumentando a combinação.
2. Se o algoritmo chegar ao nível n da árvore, ou seja, chegar ao fim do array e não encontrar solução, retornara 0 e será gerado e corrido o outro ramo.
3. Se a soma igualar a soma desejada a função retornará a combinação chave correspondente.

Este método é também bastante intuitivo e fácil de implementar, no entanto mais uma vez possuí uma alta complexidade algorítmica de O(2^n).



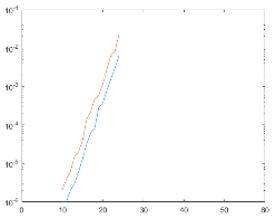
Clever Brute Force (Ramificar e limitar)

O método “Branch and Bound” é uma otimização matemática usada em vários tipos de problemas. Consiste em mais uma vez percorrer uma árvore, onde exploramos os ramos da árvore sendo que se não for possível criar uma solução a partir deste ramo ele é simplesmente ignorado.

Este método é bastante semelhante ao anterior com algumas diferenças:

1. Percorremos o conjunto dos números maiores para os mais pequenos construindo a árvore também desta forma.
2. Se a Soma corrente for maior que a Soma desejada o ramo é então ignorado pois é impossível, a partir deste ramo, encontrar uma solução.
3. Se a solução for encontrada e retornada a combinação tal como no anterior.

Este método é bastante mais eficaz que o anterior, mas não possui uma complexidade algorítmica definida sendo um algoritmo bastante difícil de analisar.

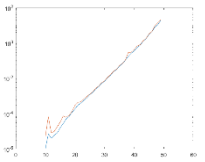


Meet-in-the-middle (Horowitz e Sahni)

Este algoritmo é muito mais eficaz temporalmente com a desvantagem de alocar muita mais memoria que os outros algoritmos.

1. Separa o conjunto em 2 subconjuntos com n/2 elementos cada (n representa o número de elementos totais do conjunto).
2. Cria e guarda todas as combinações e somas possíveis para cada subconjunto.
3. Ordena os dois subconjuntos de somas criados.
4. Percorre o subconjunto com as menores somas de baixo para cima e o conjunto com as maiores somas no sentido contrário “encontrando-se no meio”.

Este algoritmo tem de complexidade temporal $O(2^{2/n} * (n/2))$ mas requer uma complexidade espacial maior $O(2^{n/2})$. A criação dos subconjuntos de somas tal como a sua ordenação também terão efeito na complexidade temporal e espacial do algoritmo



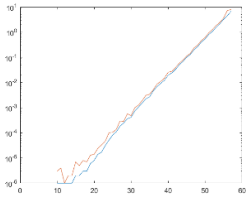
Faster Meet-in-the-middle

Este algoritmo é fundamentalmente igual ao anterior sendo a unica diferença na criação e ordenação dos subconjuntos antes falados.

Desta vez ao invés de cria-los e depois usar uma rotina de ordenação iremos no momento de criação colocar as somas já por ordem usando os principios de um merge sort;

1. Separa o conjunto em 2 subconjuntos de somas ordenadas com $2^{n/2}$ elementos cada (n representa o número de elementos totais do conjunto).
2. Percorre o subconjunto com as menores somas de baixo para cima e o conjunto com as maiores somas no sentido contrário “encontrando-se no meio”.

Este algoritmo terá uma complexidade espacial igual á anterior, e uma complexidade temporal menor.



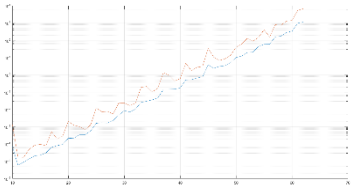
Schroeppel and Shamir

Este algoritmo é muito semelhante ao algoritmo Horowitz e Sahni mas usando significativamente menos memória. Em vez de mantermos os 2 subconjuntos com as somas, organizamos a informação num minHeap e maxHeap gerados sempre que os queremos acessar. Os somas de **p** são divididas em 4 subconjuntos de igual tamanho.

1. Separa o conjunto em 4 subconjuntos de somas ordenadas com $2^{n/4}$ elementos cada (n representa o número de elementos totais do conjunto).
2. Criamos 2 Max Heaps e 2 Min Heaps (um para guardar as somas e o outro para guardar os índices “i”,”j” correspondestes no subconjunto)
3. Sendo “A”,”B”,”C”,”D” os subconjuntos, preenche 2 minHeap com (D[0]+B[j]) e (0,j), e 2 maxHeap com (C[C_Size-1]+A[j]) e (C_Size-1,j)
4. Depois vamos percorrendo os heaps dando pop e se possível adicionando (D1[i+1],B[j]) no caso do min heap e (C[i-1]+A[j]) no caso do maxHeap. Até encontrarmos a soma.

Este algoritmo tem a vantagem de ser possível usar para conjuntos com **n** bastante elevados apesar de ser bastante complicado de implementar.

Comparando com o algoritmo anterior terá uma complexidade espacial substancialmente menor usando apenas $O(2^{n/4})$. É mais lento que o algoritmo anterior devido à necessidade de gerar os heaps necessários tendo uma complexidade temporal $O((n/4) \times 2^{n/2})$.



Observações e conclusão

Foi alcançado o objetivo trabalho implementando todos os algoritmos necessários e descobrindo as soluções também necessárias. Os gráficos são demonstrativos da eficiência e da complexidade temporal de cada algoritmo.

Este trabalho contribui-o para o nosso conhecimento da linguagem de programação C tal como as estruturas de dados necessárias para construir os algoritmos. Uma grande parte do código usado provém das aulas praticas onde este nos foi lecionado e serve como base de certas funções.

Foi também tentada a criação de um “Greedy algorithm” mas não achamos relevante suficiente para acrescentar extensivamente no relatório visto não encontrar este sempre uma solução e ser muito complicado de definir em complexidade algorítmica.

Concluindo, pela análise da complexidade temporal e espacial, podemos dizer que para diferentes números de instâncias diferentes algoritmos são preferíveis. Branch and Bound e Meet in the Middle são os únicos recomendados para um pequeno número de instâncias sendo que o algoritmo Meet in the Middle será muito mais dispendioso em termos espaciais, mas exponencialmente mais rápido. Para reduzir os elevados custos espaciais e por isso também conseguir um número de instâncias mais elevado é necessário a utilização do algoritmo de Schroeppel and Shamir.

Graphs

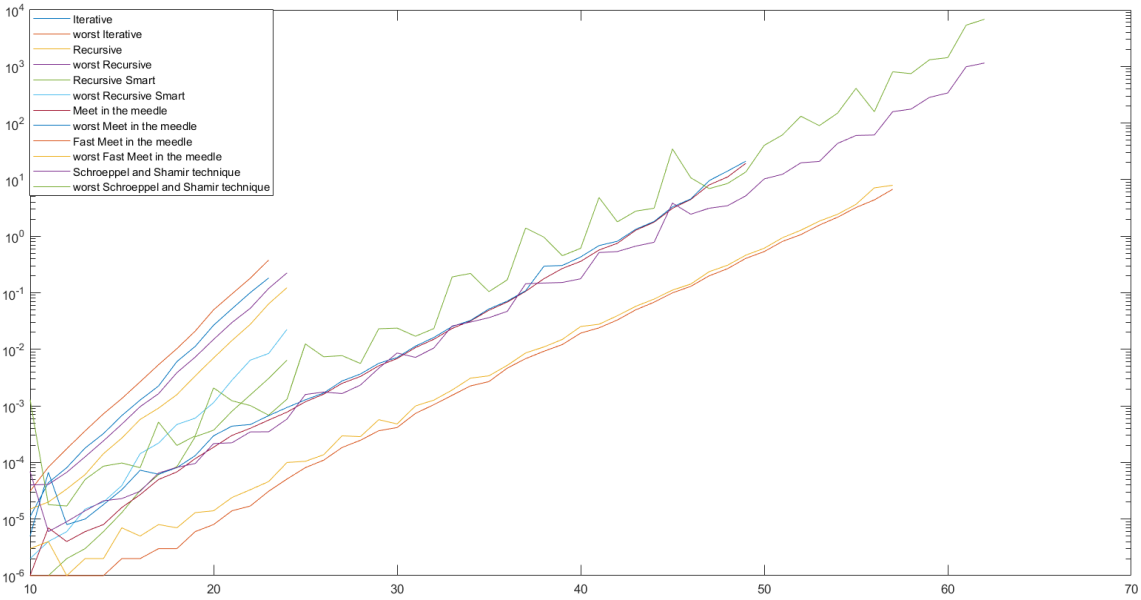


Figura 1. Todos os algoritmos

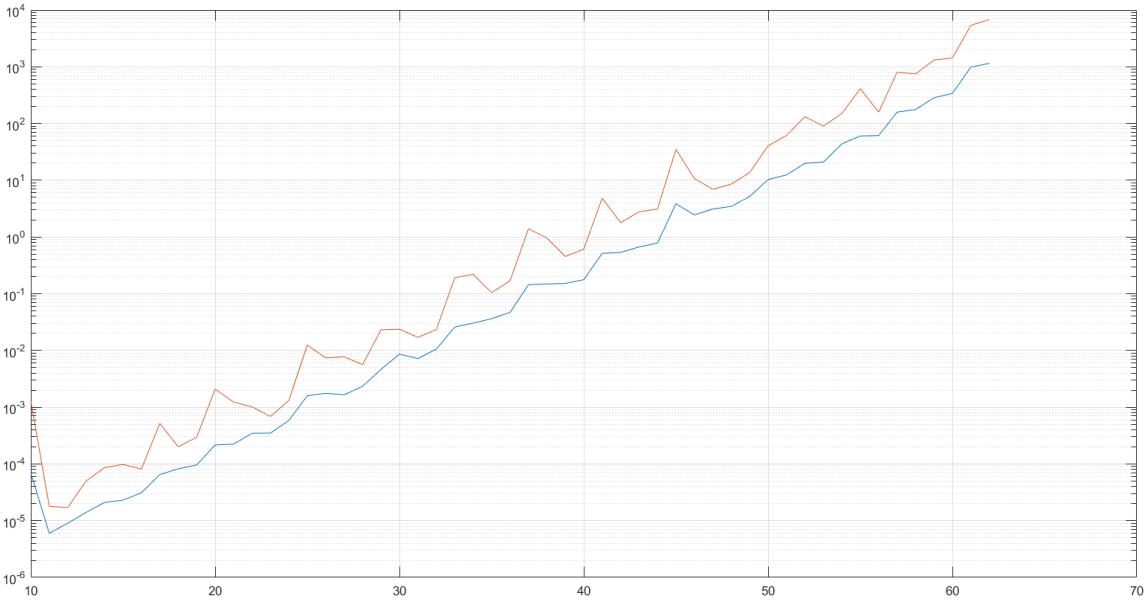


Figura 2. Schroeppel and Shamir

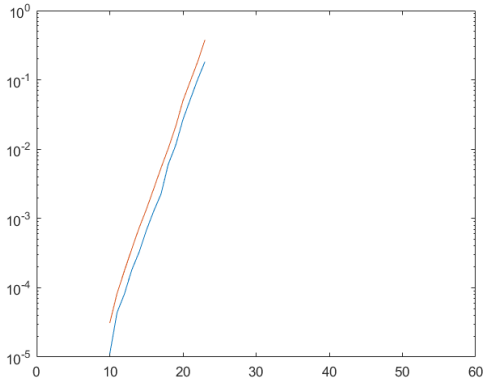


Figura 3. Algoritmo Iterativo

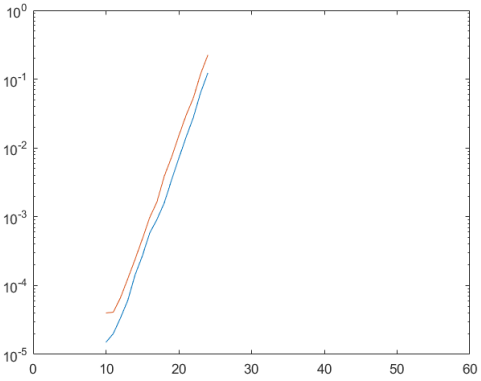


Figura 4. Algoritmo Recursivo

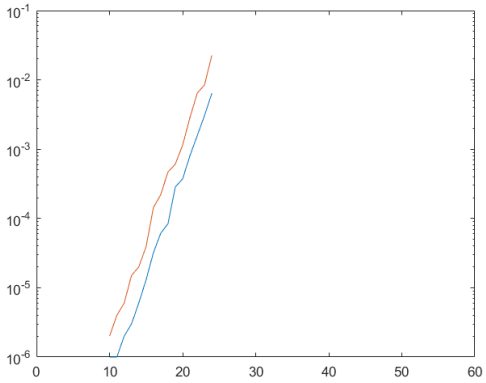


Figura 5. Algoritmo Branch and Bound

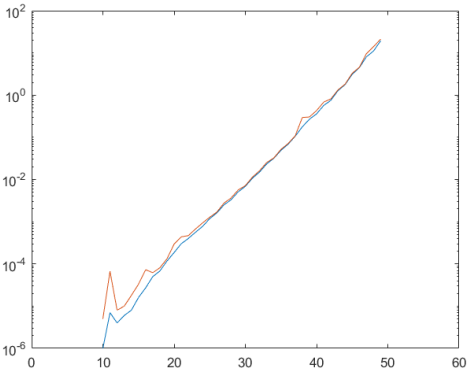


Figura 6. Algoritmo Meet in the Middle

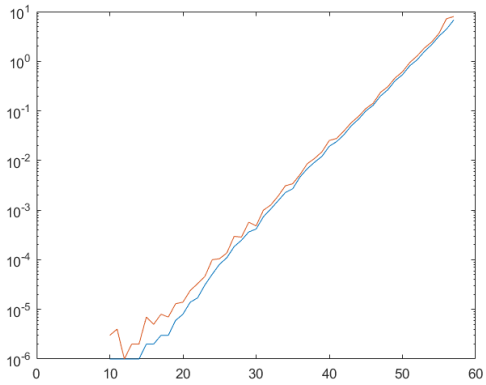


Figura 7. Algoritmo Faster Meet in the Middle

Código utilizado

Brute Forces

```c
// Brute Force Iterative Algorithm
int Bf_Iter(int n, integer_t *p, integer_t desired_sum){

  int comb = 0;
  integer_t test_sum;

  for(comb;comb<(1<<n);comb++){   // changes the combination

    test_sum =0;

    for(int bit=0; bit<n ;bit++){

      int mask = (1<<bit);
      if((comb & mask)!=0){test_sum += p[bit];}
    }

    if(test_sum == desired_sum){break;} // tests if the sum is equal to the desired sum
  }

  return comb;

}
```

```c
// Brute Force Recursive Algorithm
int Bf_recur( unsigned int n,unsigned int m,integer_t *p,double sum, int comb,integer_t desired_sum){

  if(m == n){ // if is in last step of recursion

    if (sum == desired_sum){
      return comb;    // return combination different from 0 if found

    }
    else return 0;      // else returns 0
  }


  int result = Bf_recur(n,m + 1u,p,sum , comb ,desired_sum);
  if (result == 0){         // if combination is 0 it means we must take another path

    return Bf_recur(n,m + 1u,p,sum + p[m] ,comb+ pow(2,m),desired_sum);
  }

  return result;
}
```

```c
// Smart Brute Force Recursive Algorithm
integer_t Bf_recur_smart( unsigned int n,int m,integer_t *p,integer_t sum, integer_t comb,integer_t desired_sum)
{  // this function run trougth the array from bigger to smaller and if the current_sum is bigger then the desired_sum

  if (sum == desired_sum){    // found the solucion so we return the combination
    return comb;
  }

  if(sum > desired_sum){ // if current sum > desired_sum this branch does not contain the solucion
    return 0;
  }

  if(m == -1) {              // if we reached the end of the array
    return 0 ;
  }

  integer_t power = (1ull);
  integer_t result = Bf_recur_smart(n,m-1,p,sum + p[m], comb + (power<<m),desired_sum);

  if (result == 0)  {            // if combination is 0 it means we must take another path
    return Bf_recur_smart(n,m-1,p,sum  ,comb,desired_sum);
  }
  return result;
}
```

Meet in the middle

```c
//Algorithm itself
int mitm(int n, integer_t *p, integer_t desired_sum){

  // Get sub-arrays sizes
  int size_X = 1<<(n/2);
  int size_Y = 1<<(n-n/2);

  // Allocate space for them
  integer_t *X = malloc(size_X*sizeof(integer_t));
  integer_t *Y = malloc(size_Y*sizeof(integer_t));

  // Create the sub arrays
  calcsubarray(p, X, n/2, 0);
  calcsubarray(p, Y, n-n/2, n/2);

  // Sort them
  heapSort(X, size_X);
  heapSort(Y, size_Y);

  /* Go through the array X from start to end, and through Y form end to start, testing if
  int i= 0;
  int j= size_Y - 1;
  while(i< size_X && j >= 0){

    integer_t s = X[i]+Y[j];
    if(s == desired_sum ){

      free(X);    //freeing the space of the arrays
      free(Y);
      return 1;   // return 1 if its found
    }else if(s < desired_sum){
      i++;
    }else{
      j--;
    }
  }

  // return 0 if is not found
  return 0;
}
```

```c
void calcsubarray(integer_t a[], integer_t x[], int n, int c){

  integer_t s;
  for (int i=0; i<(1<<n); i++)
  {
    s = 0;
    for (int j=0; j<n; j++){
      if (i & (1<<j)){
        s += a[j+c];
      }
    }

    if(s >= 0){
      x[i] = s;
    }
  }
}
```

```c
void swap(integer_t *a, integer_t *b) {
  integer_t temp = *a;
  *a = *b;
  *b = temp;
}
```

```c
//Heap Sort
  void heapify(integer_t arr[], int n, int i) {
      // Find largest among root, left child and right child
      int largest = i;
      int left = 2 * i + 1;
      int right = 2 * i + 2;

      if (left < n && arr[left] > arr[largest])
        largest = left;

      if (right < n && arr[right] > arr[largest])
        largest = right;

      // Swap and continue heapifying if root is not largest
      if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
      }
  }


  void heapSort(integer_t arr[], int n) {
      // Build max heap
      for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

      // Heap sort
      for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);

        // Heapify root element to get highest element at root again
        heapify(arr, i, 0);
      }
  }
```

Faster Meet in the middle

```c
int faster_mitm(int n, integer_t *p, integer_t desired_sum){
  // Get sub-arrays sizes
  int size_X = 1<<(n/2);
  int size_Y = 1<<(n-n/2);
  // Allocate space for them
  integer_t *X = malloc(size_X*sizeof(integer_t));
  integer_t *Y = malloc(size_Y*sizeof(integer_t));

  for(int i=0;i<size_X;i++){
    X[i]=0;
  }
  for(int i=0;i<size_Y;i++){
    Y[i]=0;
  }

  // Create the sub arrays and Sort them
  faster_calcsubarray(p, X, n/2, 0);
  faster_calcsubarray(p, Y, n-n/2, n/2);

  /* Go through the array X from start to end, and through Y form end to start, testing if
  int i= 0;
  int j= size_Y - 1;
  while(i< size_X && j >= 0){
    integer_t s = X[i]+Y[j];

    if(s == desired_sum){

      free(X);    //freeing the space of the arrays
      free(Y);
      return 1;   // return 1 if its found
    }else if(s < desired_sum){
      i++;
    }else{
      j--;
    }
  }

  // return 0 if is not found
  return 0;
}
```

```c
//Create a Sorted Array(x[]) composed by all the possible sums of a given Array(a[])
  void faster_calcsubarray(integer_t a[], integer_t x[], int n, int c){

    for (int i=0; i<n; i++)
    {

      integer_t i1 = (1<<n)-(1<<i);
      integer_t j1 = i1;
      integer_t k1 = (1<<n)-(2<<i);

      while(i1 < (1<<n)){

        if(x[i1] <= x[j1] + a[i+c]){
          x[k1++] = x[i1++];
        }else{
          x[k1++] = x[j1++] + a[i+c];
        }
      }
      while(j1 < (1<<n)){
        x[j1++] += a[i+c];
      }
    }

  }
```

Schroeppel and Shamir

```
// Schroeppel and Shamir technique

 //Heap Structure

  integer_t Idx_Min_Heap[1 << 23][2];
  integer_t Idx_Max_Heap[1 << 23][2];

  integer_t Min_Heap[1<<23];
  integer_t MinH_Size = 0;

  integer_t Max_Heap[1<<23];
  integer_t MaxH_Size = 0;

  integer_t MinH_i, MaxH_i, MinH_j, MaxH_j;
```

```
void Heap_Swap(integer_t i, integer_t j, int type){
  integer_t tmp = 0;
  switch(type){
    case 0:
      tmp = Min_Heap[i];
      Min_Heap[i] = Min_Heap[j];
      Min_Heap[j] = tmp;

      tmp = Idx_Min_Heap[i][0];
      Idx_Min_Heap[i][0] = Idx_Min_Heap[j][0];
      Idx_Min_Heap[j][0] = tmp;

      tmp = Idx_Min_Heap[i][1];
      Idx_Min_Heap[i][1] = Idx_Min_Heap[j][1];
      Idx_Min_Heap[j][1] = tmp;
    break;

    case 1:
      tmp = Max_Heap[i];
      Max_Heap[i] = Max_Heap[j];
      Max_Heap[j] = tmp;

      tmp = Idx_Max_Heap[i][0];
      Idx_Max_Heap[i][0] = Idx_Max_Heap[j][0];
      Idx_Max_Heap[j][0] = tmp;

      tmp = Idx_Max_Heap[i][1];
      Idx_Max_Heap[i][1] = Idx_Max_Heap[j][1];
      Idx_Max_Heap[j][1] = tmp;
    break;
  }
}
```

```
// Min Heap
  void MinH_Heapify(integer_t i){

    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (!(i + 1 > (MinH_Size / 2) && i < MinH_Size)){

      if (Min_Heap[i] > Min_Heap[l] || Min_Heap[i] > Min_Heap[r]){

        if (Min_Heap[l] < Min_Heap[r]){

          Heap_Swap(i, l, 0);
          MinH_Heapify(l);
        }
        else{

          Heap_Swap(i, r, 0);
          MinH_Heapify(r);
        }
      }
    }
  }
```

```
  void MinH_Insert(integer_t newNum, integer_t i, integer_t j){

    Min_Heap[MinH_Size] = newNum;
    Idx_Min_Heap[MinH_Size][0] = i;
    Idx_Min_Heap[MinH_Size][1] = j;
    integer_t pos = MinH_Size;
    MinH_Size++;

    while (Min_Heap[pos] < Min_Heap[(pos - 1) / 2]){

      Heap_Swap(pos, (pos - 1) / 2, 0);
      pos = (pos - 1) / 2;
    }
  }

  void MinH_Pop(){
    // replace first node by last and delete last
    Min_Heap[0] = Min_Heap[MinH_Size - 1];
    Idx_Min_Heap[0][0] = Idx_Min_Heap[MinH_Size - 1][0];
    Idx_Min_Heap[0][1] = Idx_Min_Heap[MinH_Size - 1][1];
    MinH_Size--;

    MinH_Heapify(0);
  }
```

```
// Max Heap
  void MaxH_Heapify(integer_t i){

    integer_t l = 2 * i + 1;
    integer_t r = 2 * i + 2;

    if (!(i + 1 > (MaxH_Size / 2) && i < MaxH_Size)){

      if (Max_Heap[i] < Max_Heap[l] || Max_Heap[i] < Max_Heap[r]){

        if (Max_Heap[l] > Max_Heap[r]){

          Heap_Swap(i, l, 1);
          MaxH_Heapify(l);
        }
        else{

          Heap_Swap(i, r, 1);
          MaxH_Heapify(r);
        }
      }
    }
  }
```

```
  void MaxH_Insert(integer_t newNum, integer_t i, integer_t j){

    Max_Heap[MaxH_Size] = newNum;
    Idx_Max_Heap[MaxH_Size][0] = i;
    Idx_Max_Heap[MaxH_Size][1] = j;
    integer_t pos = MaxH_Size;
    MaxH_Size++;
    while (Max_Heap[pos] > Max_Heap[(pos - 1) / 2] && pos> 0){

      Heap_Swap(pos, (pos - 1) / 2, 1);
      pos = (pos - 1) / 2;
    }
  }

  void MaxH_Pop(){
    // replace first node by last and delete last
    Max_Heap[0] = Max_Heap[MaxH_Size - 1];
    Idx_Max_Heap[0][0] = Idx_Max_Heap[MaxH_Size - 1][0];
    Idx_Max_Heap[0][1] = Idx_Max_Heap[MaxH_Size - 1][1];
    MaxH_Size--;


    MaxH_Heapify(0);
  }
```

```c
//Algorithm itself
int SS_T(int n, integer_t *p, integer_t desired_sum){

  // Get sub-arrays sizes
  int Pre_A_Size = (n/2)/2;
  int Pre_B_Size = ((n/2) - (n/2)/2);
  int Pre_C_Size = (n - n/2)/2;
  int Pre_D_Size = ((n - n/2) - (n - n/2)/2);

  int A_Size = 1<<Pre_A_Size;
  int B_Size = 1<<Pre_B_Size;
  int C_Size = 1<<Pre_C_Size;
  int D_Size = 1<<Pre_D_Size;

  // Allocate space for them
  integer_t *A = malloc(sizeof(integer_t)*A_Size);
  integer_t *B = malloc(sizeof(integer_t)*B_Size);
  integer_t *C = malloc(sizeof(integer_t)*C_Size);
  integer_t *D = malloc(sizeof(integer_t)*D_Size);

  // Create Sorted sub arrays
  faster_calcsubarray(p, A, Pre_A_Size, 0);
  faster_calcsubarray(p, B, Pre_B_Size, Pre_A_Size);
  faster_calcsubarray(p, C, Pre_C_Size, Pre_A_Size + Pre_B_Size);
  faster_calcsubarray(p, D, Pre_D_Size, Pre_A_Size + Pre_B_Size + Pre_C_Size);

  // Populate Heaps
  for (int k = 0; k < B_Size; k++){
    MinH_Insert(B[k], k, 0);
  }
  for (int k = 0; k < A_Size; k++){
    MaxH_Insert(A[k] + C[C_Size - 1], k, C_Size - 1);
  }

  // Loop 1<<n times(maximon possibilities)
  integer_t Maximum_Pos = pow(2,n);
  integer_t min, max;
  for (integer_t i = 0; i < Maximum_Pos; i++){

    // Get Roots of the Heaps
    max = Max_Heap[0];
    min = Min_Heap[0];
    MinH_i = Idx_Min_Heap[0][0];
    MinH_j = Idx_Min_Heap[0][1];
    MaxH_i = Idx_Max_Heap[0][0];
    MaxH_j = Idx_Max_Heap[0][1];

    // Test Sum
    integer_t sum = min + max;
    if (sum == desired_sum){

      return 1;
    }
    else if (sum < desired_sum){

      // Pop and if possible switch root
      MinH_Pop();
      if (MinH_j + 1 < D_Size){

        MinH_Insert(B[MinH_i] + D[MinH_j+1], MinH_i, MinH_j+1);
      }
    }
    else{

      // Pop and if possible switch root
      MaxH_Pop();
      if (MaxH_j > 0){
        MaxH_Insert(A[MaxH_i] + C[MaxH_j-1], MaxH_i, MaxH_j-1);
      }
    }
  }

  return 0;
}
```

Código Matlab

```
A_1 = load("data_1.log");
A_2 = load("data_1_max.log");
A_3 = load("data_2.log");
A_4 = load("data_2_max.log");
A_5 = load("data_3.log");
A_6 = load("data_3_max.log");
A_7 = load("data_4.log");
A_8 = load("data_4_max.log");
A_9 = load("data_5.log");
A_10 = load("data_5_max.log");
A_11 = load("data_6.log");
A_12 = load("data_6_max.log");
A_13 = load("data_7.log");
A_14 = load("data_7_max.log");

 %se coluda 2==1  %coluna
n_V1_1 = A_1(:, 1);
f_V1_1 = A_1(:, 2);
n_V1_2 = A_2(:, 1);
f_V1_2 = A_2(:, 2);
n_V1_3 = A_3(:, 1);
f_V1_3 = A_3(:, 2);
n_V1_4 = A_4(:, 1);
f_V1_4 = A_4(:, 2);
n_V1_5 = A_5(:, 1);
f_V1_5 = A_5(:, 2);
n_V1_6 = A_6(:, 1);
f_V1_6 = A_6(:, 2);
n_V1_7 = A_7(:, 1);
f_V1_7 = A_7(:, 2);
n_V1_8 = A_8(:, 1);
f_V1_8 = A_8(:, 2);
n_V1_9 = A_9(:, 1);
f_V1_9 = A_9(:, 2);
n_V1_10 = A_10(:, 1);
f_V1_10 = A_10(:, 2);
n_V1_11 = A_11(:, 1);
f_V1_11 = A_11(:, 2);
n_V1_12 = A_12(:, 1);
f_V1_12 = A_12(:, 2);
n_V1_13 = A_13(:, 1);
f_V1_13 = A_13(:, 2);
n_V1_14 = A_14(:, 1);
f_V1_14 = A_14(:, 2);
```

```
%semilogy(n_V1_7, f_V1_7, n_V1_8, f_V1_8, n_V1_7, f_V1_7./10, n_V1_8, f_V1_8./10);

%semilogy(n_V1_1, f_V1_1, n_V1_2, f_V1_2, n_V1_3, f_V1_3, n_V1_4, f_V1_4, n_V1_5, f_V1_5, n_V1_6, f_V1_6, n_V1_7, f_V1_7, n_V1_8, f_V1_8, n_V1_9, f_V1_9, n_V1_10, f_V1_10, n_V1_11, f_V1_11, n_V1
%semilogy(n_V1_1, f_V1_1, n_V1_3, f_V1_3, n_V1_5, f_V1_5, n_V1_7, f_V1_7, n_V1_9, f_V1_9, n_V1_11, f_V1_11, n_V1_13, f_V1_13 );
semilogy(n_V1_1, f_V1_1, n_V1_2, f_V1_2, n_V1_3, f_V1_3, n_V1_4, f_V1_4, n_V1_5, f_V1_5, n_V1_6, f_V1_6, n_V1_7, f_V1_7, n_V1_8, f_V1_8, n_V1_9, f_V1_9, n_V1_10, f_V1_10);


%semilogy( n_V1_7, f_V1_7, n_V1_8, f_V1_8,n_V1_9, f_V1_9, n_V1_10, f_V1_10, n_V1_11, f_V1_11, n_V1_12, f_V1_12, n_V1_13, f_V1_13, n_V1_14, f_V1_14);
legend('Iterative', 'worst Iterative','Recursive','worst Recursive','Recursive Smart','worst Recursive Smart','Meet in the meedle','worst Meet in the meedle','Fast Meet in the meedle','worst Fas
%legend('Meet in the meedle','worst Meet in the meedle','Fast Meet in the meedle','worst Fast Meet in the meedle','Schroeppel and Shamir technique','worst Schroeppel and Shamir technique','Schro
%legend('Iterative', 'worst Iterative','Recursive','worst Recursive','Recursive Smart','worst Recursive Smart','Meet in the meedle','worst Meet in the meedle','Fast Meet in the meedle','worst Fa
%legend('Iterative',  'Recursive', 'Recursive Smart', 'Meet in the meedle' ,'Fast Meet in the meedle' ,'Schroeppel and Shamir technique', 'Schroeppel and Shamir technique Teles' );
grid on


%%
figure(1);
semilogy(n_V1_1, f_V1_1, n_V1_2, f_V1_2);
xlim([0,60]);
figure(2);
semilogy(n_V1_3, f_V1_3, n_V1_4, f_V1_4);
xlim([0,60]);
figure(3);
semilogy(n_V1_5, f_V1_5,n_V1_6, f_V1_6);
xlim([0,60]);
figure(4);
semilogy(n_V1_7, f_V1_7,n_V1_8, f_V1_8);
xlim([0,60]);
figure(5);
semilogy(n_V1_9, f_V1_9,n_V1_10, f_V1_10);
xlim([0,60]);
figure(6);
semilogy(n_V1_11, f_V1_11, n_V1_12, f_V1_12);
xlim([0,60]);
```

Algumas Soluções Obtidas

Rafael Remígio 102435

Primeiras 8 soluções para n = 40

1110101110001100011111110001011001111110

0001111101011010110010100001100011110111

0011101001000000011000111111011000001101

1111000100010011101010101110000111001100

1101111010010000110000110000111101011011

1101011011000001110011001011111011100000

0010100101011000101001110001000101110111

0110010001111100110001110001011010010000

Primeira solução n = 57

0100010100100100101001000111010011010111011
00001101101111

João Correia 104360

Primeiras 8 soluções para n = 40

1001111010110010001010101000110011101001

0011001111011001100011001010111111011100

0110111000111101011011110110110110000101

0000000001111111010010111010010101101000

1000011011000101101000010111011000110111

0000110111010101001001111100110100110101

1100010011000010001010000001111111010000

1011001100001011110111010111100100000010

Primeira solução n = 57

1010101010101111110010001100000100101001001
10111111011111

Bibliografia/Webgrafia

1. Optimal Sequential Multi-Way Number Partitioning Richard E. Korf, Ethan L. Schreiber, and Michael D. Moffitt
2. Shamir's Attack on the Basic Merkle-Hellman Knapsack Cryptosystem
3. https://rjlipton.wpcomstaging.com/2012/12/19/branch-and-bound-why-does-it-work/
4. https://en.wikipedia.org/