**Algoritmos e Estruturas de Dados 2021/2022**

**Multi-Ordered Binary Trees**

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

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

**Índice**

**Introdução1**

Escrever título do capítulo (nível 2)2

Escrever título do capítulo (nível 3)3

**Escrever título do capítulo (nível 1)4**

Escrever título do capítulo (nível 2)5

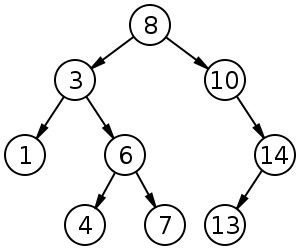
Escrever título do capítulo (nível 3)6

Introdução

Uma das áreas mais importantes da Ciência da Computação e no desenvolvimento de Software é o armazenamento e processamento de informação de forma eficaz e pouco dispendiosa. Diferentes tipos de estruturas de dados oferecem diversas vantagens e desvantagens. Neste trabalho focaremos apenas em Árvores Binárias, nas vantagens da sua utilização tal como na implementação de diversos algoritmos para percorrer e criar estas Árvores.

Árvore Binária:

Neste caso falaremos de Árvores Binárias de Busca que podem ser definidas por um conjunto de Nodes onde cada node possuí a seguinte informação

* Dados a guardar
* Ponteiro para o ramo da esquerda (left child)
* Ponteiro para o ramo da direita (right child)
* Ponteiro para o Parente **(Opcional)**
* Todos os Nodes da subárvore da esquerda possuem um valor inferior ao da raiz e todos os Nodes da subárvore da direita possuem um valor superior ao da raiz

Neste trabalho prático foi nos pedido a organização em árvores binarias de Nodes com três campos:

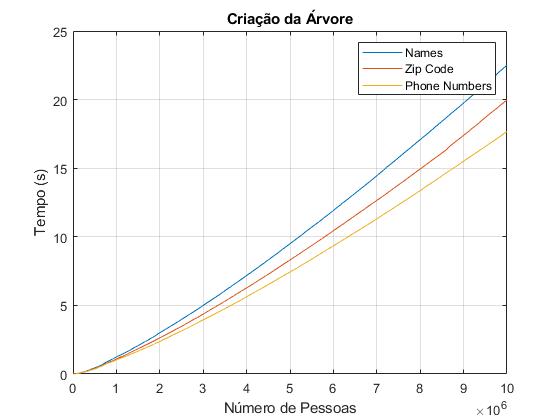
* Nome
* Zip Code
* Número de Telefone

É necessária a criação de 3 árvores, uma por cada campo, sendo que deverá ser possível percorrer, pesquisar e saber o tamanho de cada Árvore criada.

Inserção de elementos

A operação de inserção deve ter em cuidado os princípios da árvore binária de busca, sendo isto adicionando novos nodes sempre como folhas da árvore.

* Percorremos a árvore recursivamente comparando o valor adicionar com o valor do node onde nos encontramos. Caso o node tiver um valor maior que o node atual tentamos adicionar à direita caso contrário tentamos adicionar a esquerda. Desde modo o node é sempre inserido numa posição de folha não pondo em causa a estrutura da árvore
* A comparação é realizada relativamente ao campo específico através de uma função que recebe as duas Pessoas e o índex relativo há arvore que estamos a criar comparando destes modos os campos desejados. Se os campos forem iguais a função irá comparar outro campo.
* As três árvores diferentes são criadas através de um array de ponteiros, onde cada índex é referente a uma árvore diferente com diferentes campos a serem comparados

Esta operação tem de complexidade temporal em média O(log(n)) sendo o pior caso O(n). A criação completa da árvore tem de complexidade O(n\*log(n)).

Procura

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.

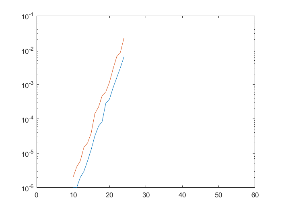
1. Consideremos que temos uma árvore onde a cada ramo é o valor de Soma + C[xi]\*0 ou Soma + C[xi]\*1 onde i é o nível da árvore. Neste caso iremos gerar todos os 2n ramos comparando pelo caminho a soma de cada iteração e aumentando a combinação.
2. Se o algoritmo chegar ao nível n da árvore, ou seja, 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.

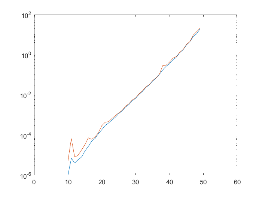
Este método é bastante semelhante ao anterior com algumas diferenças:

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.

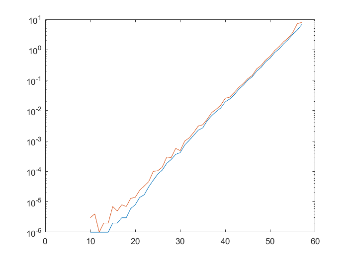
1. Separa o conjunto em 2 subconjuntos com n/2 elementos cada (n representa o número de elementos totais do conjunto).
2. Cria e guarda todas as combinações e somas possíveis para cada subconjunto.
3. Ordena os dois subconjuntos de somas criados.
4. Percorre o subconjunto com as menores somas de baixo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

Este algoritmo tem de complexidade temporal O(22/n \* (n/2)) mas requer uma complexidade espacial maior O(2n/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;

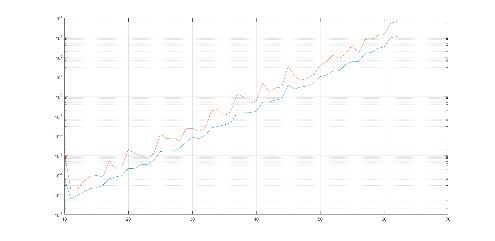
1. Separa o conjunto em 2 subconjuntos de somas ordenadas com 2n/2 elementos cada (n representa o número de elementos totais do conjunto).
2. Percorre o subconjunto com as menores somas de baixo para cima e o conjunto com as maiores somas no sentido contrário “encontrando-se no meio”.

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 2n/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(2n/4). É mais lento que o algoritmo anterior devido à necessidade de gerar os heaps necessários tendo uma complexidade temporal O((n/4) x 2n/2).

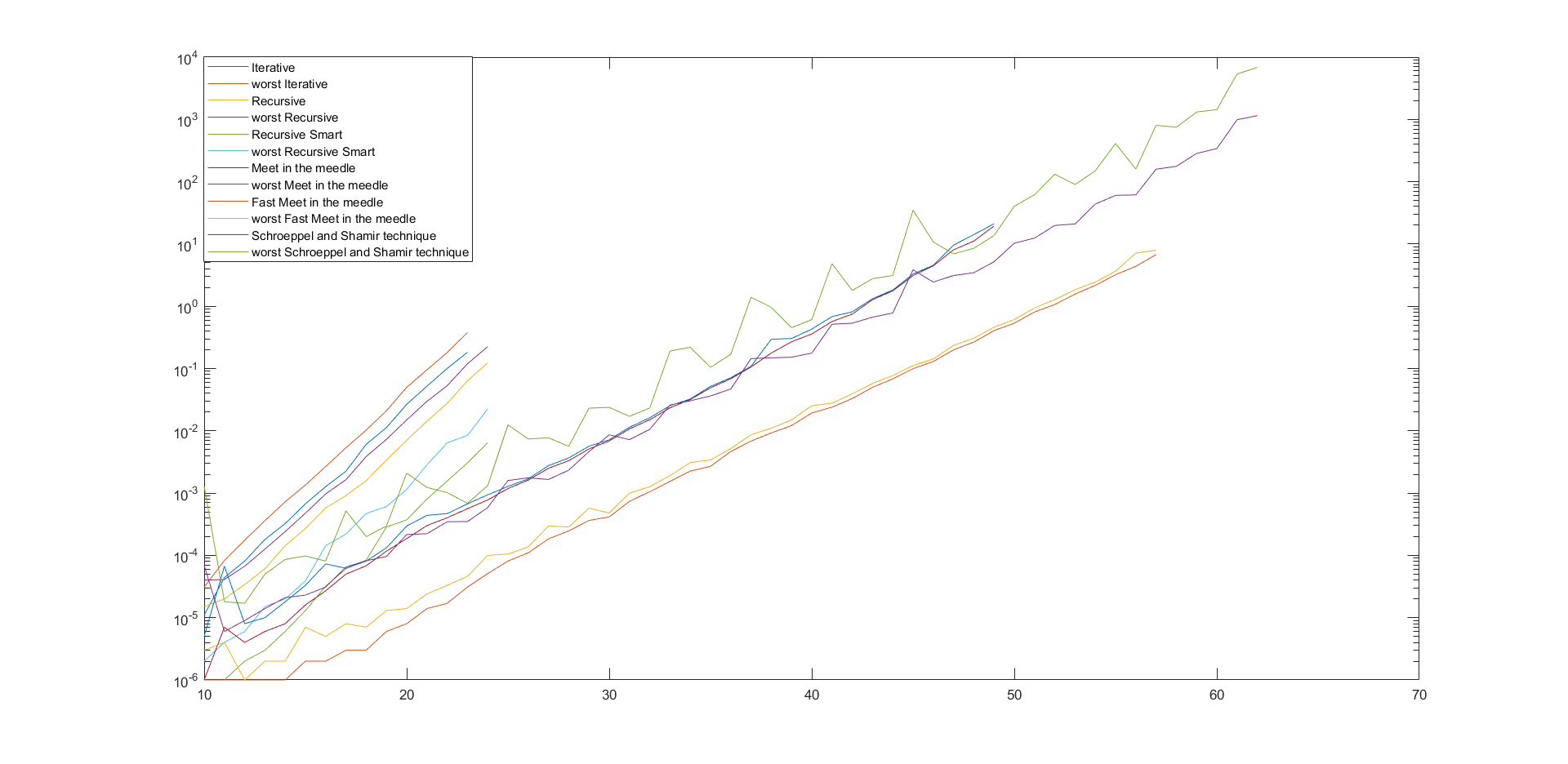
Observações e conclusão

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

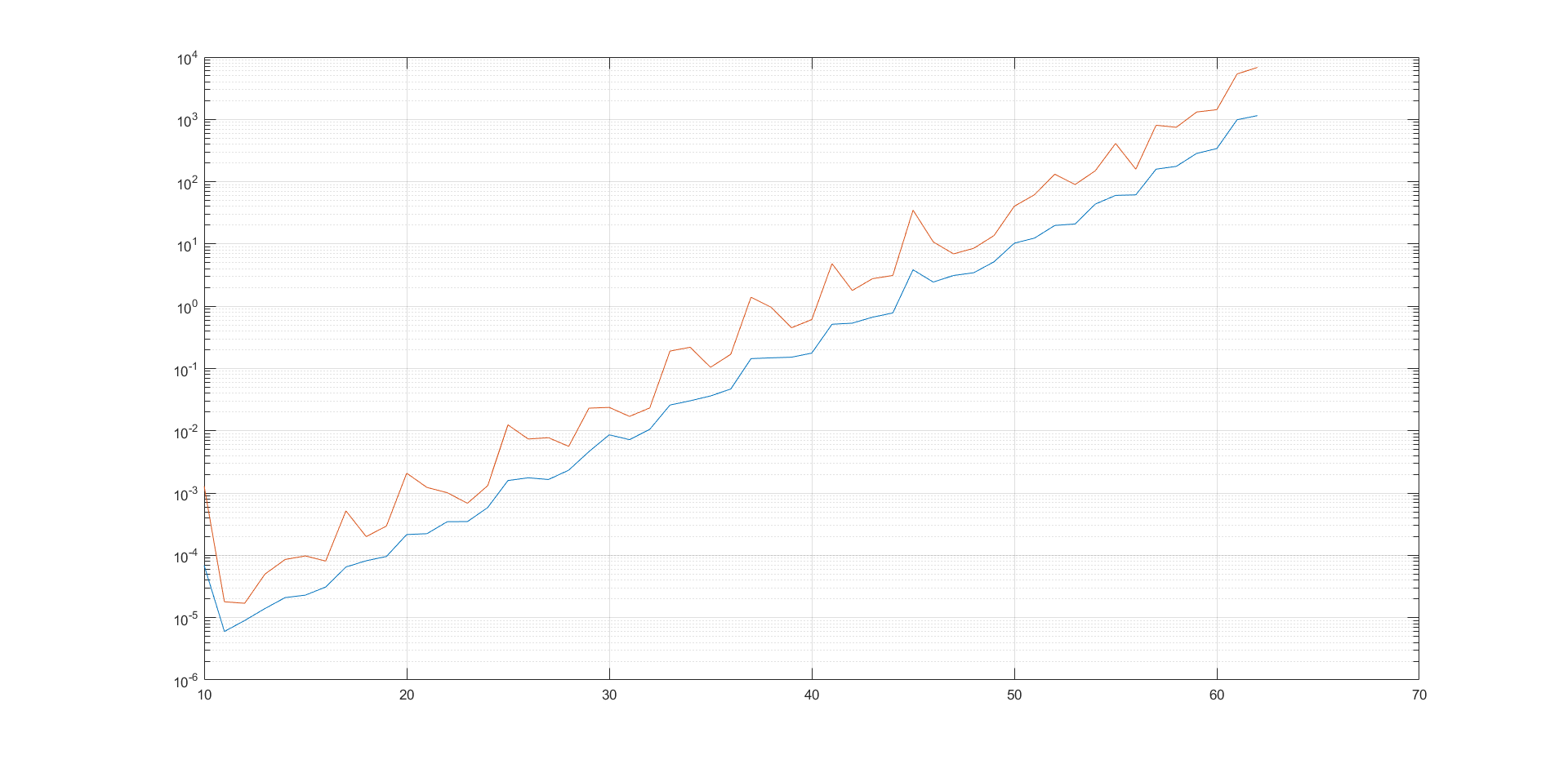
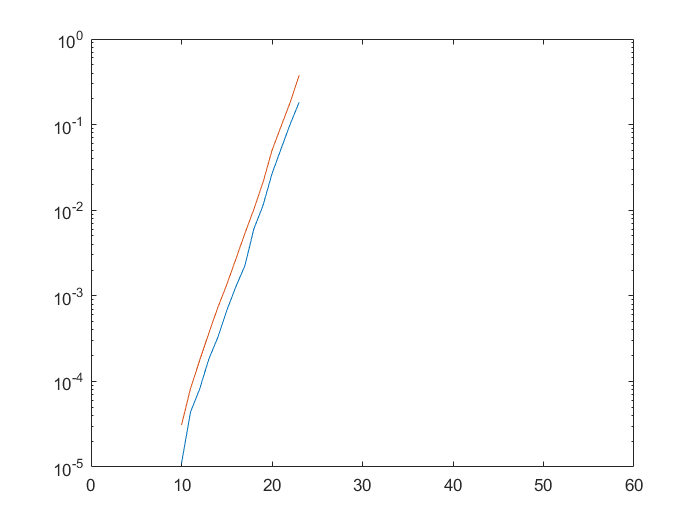
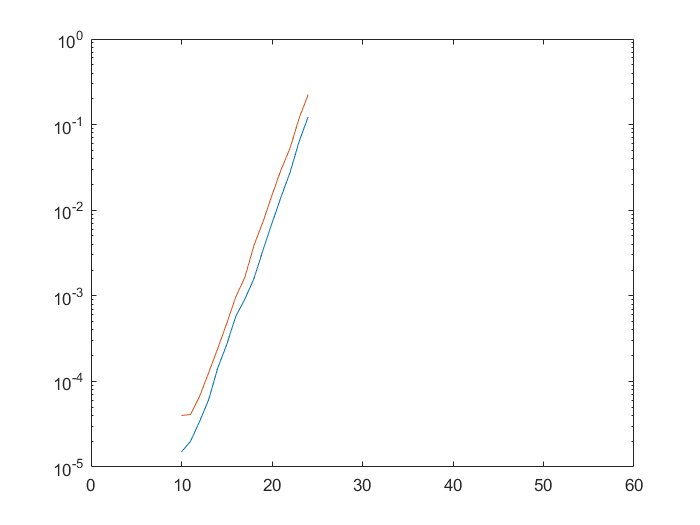


Figura 1. Todos os algoritmos

Figura 2. Schroeppel and Shamir



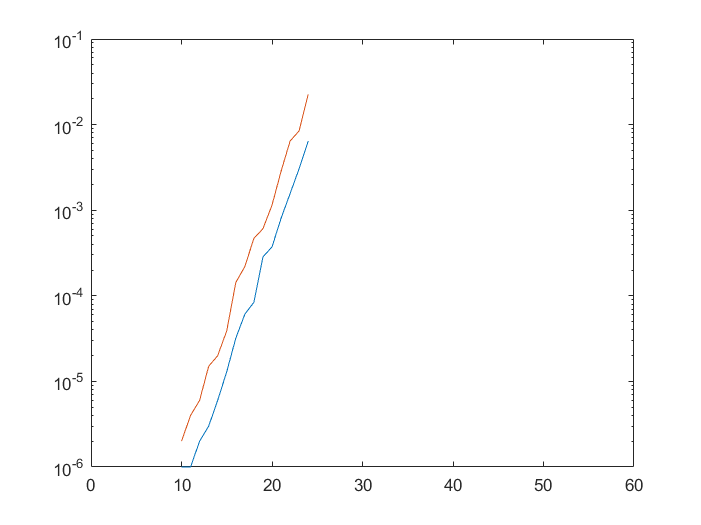
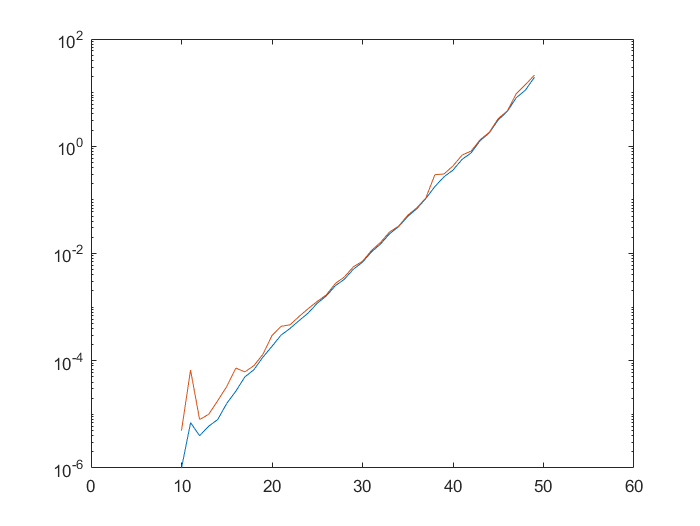


Figura 4. Algoritmo Recursivo

Figura 3. Algoritmo Iterativo

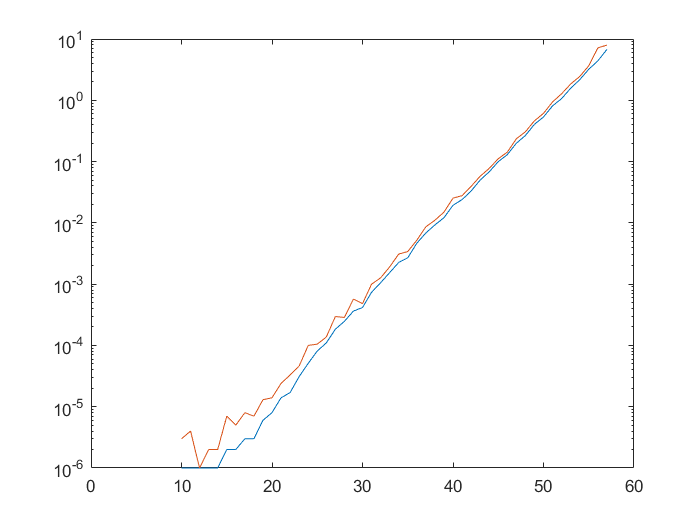


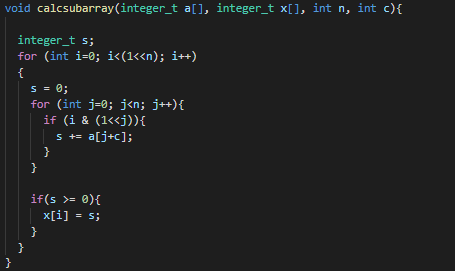
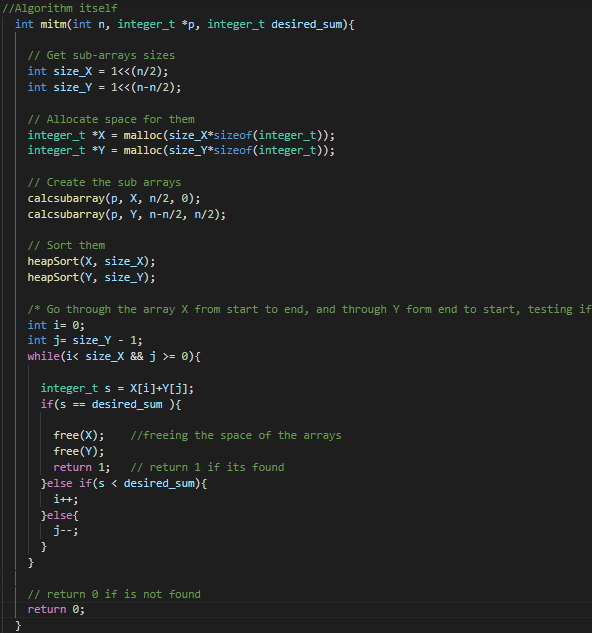
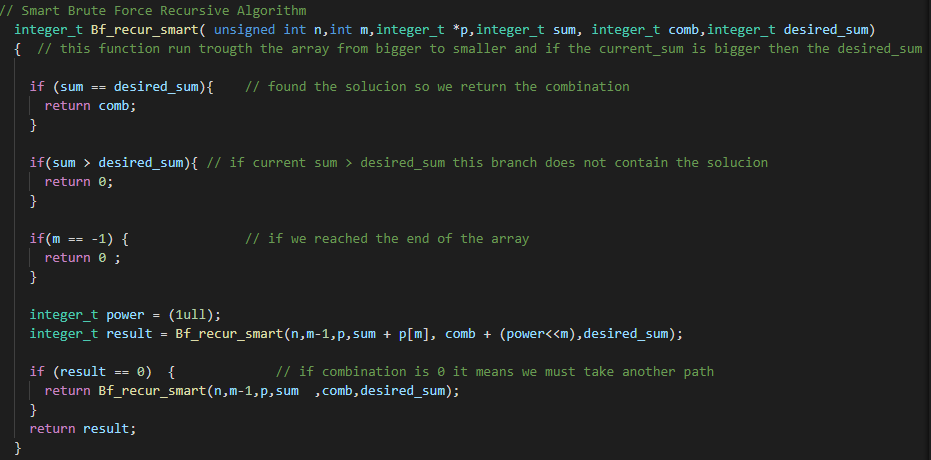
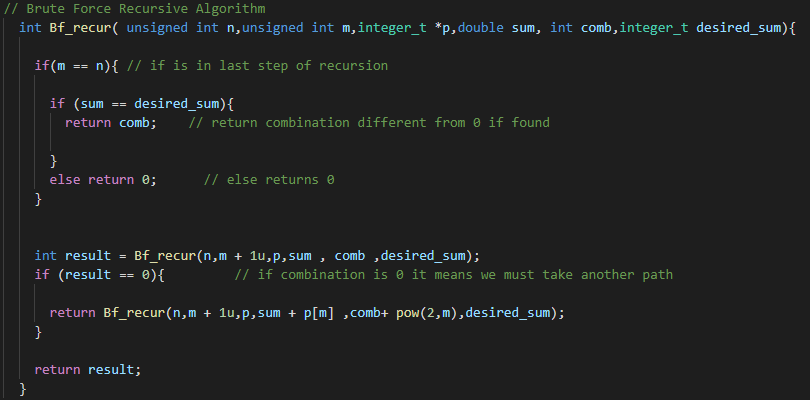
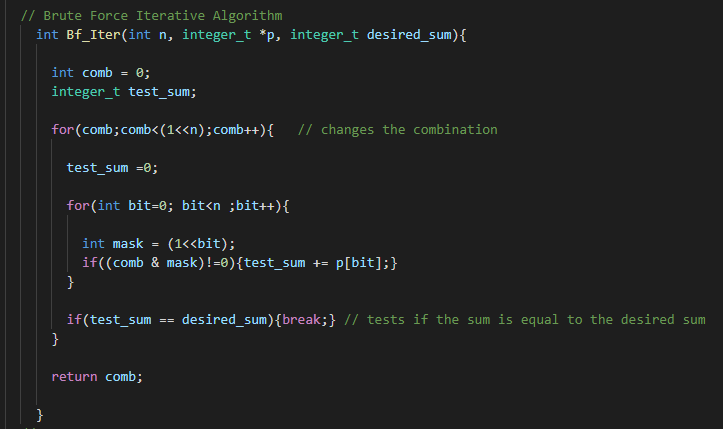
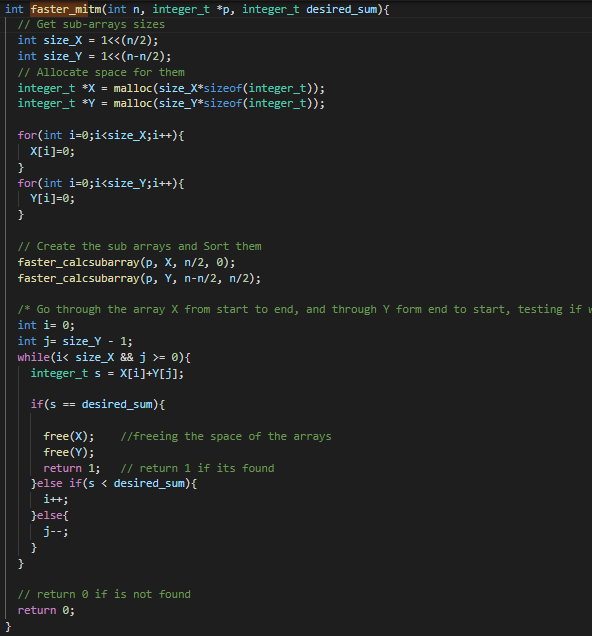
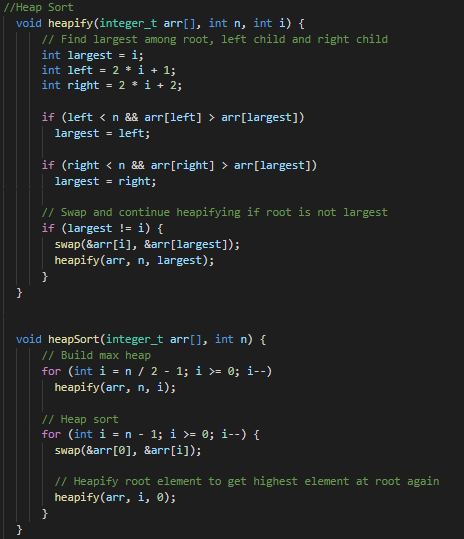
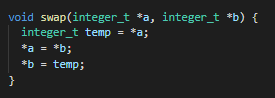
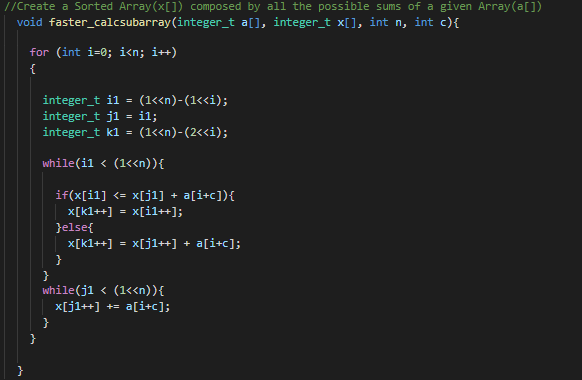
Figura 6. Algoritmo Meet in the Middle

Figura 5. Algoritmo Branch and Bound

Figura 7. Algoritmo Faster Meet in the Middle

Código utilizado

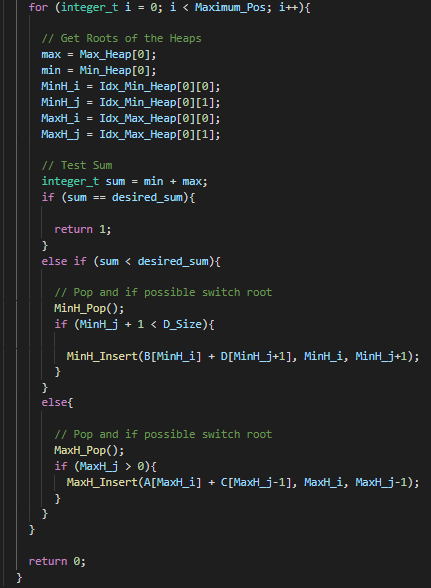
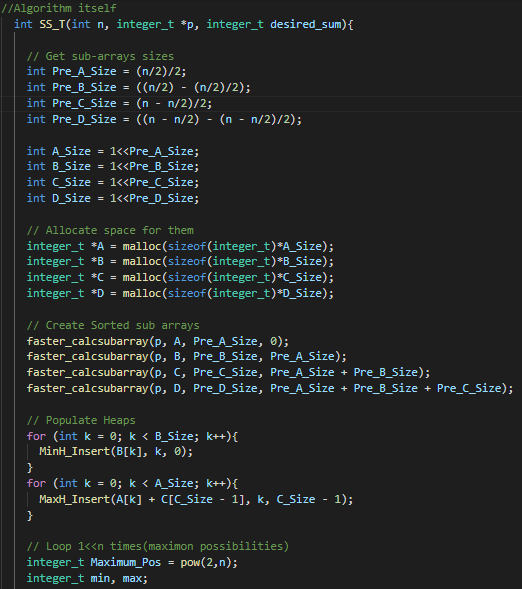
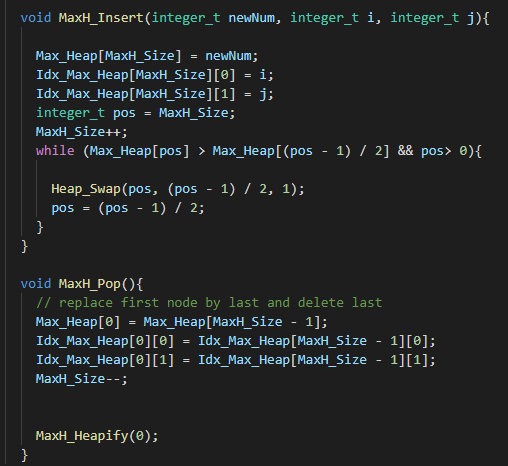
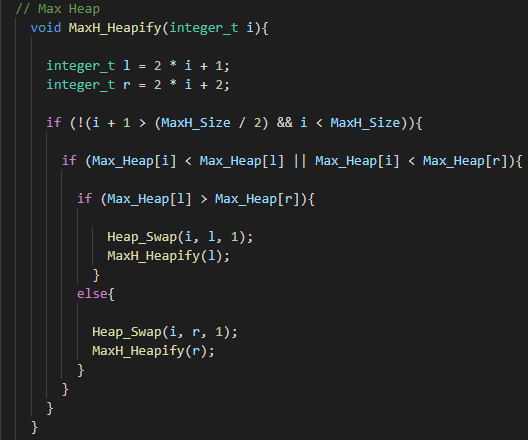
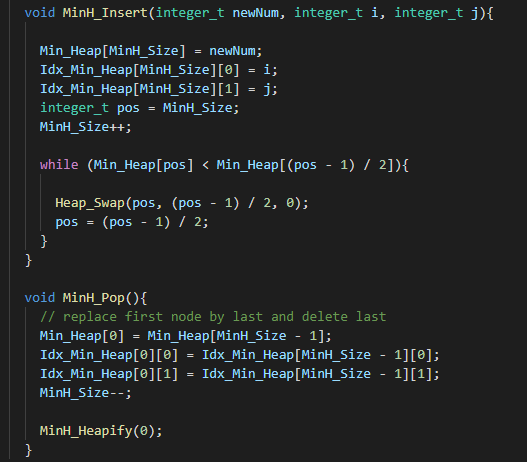
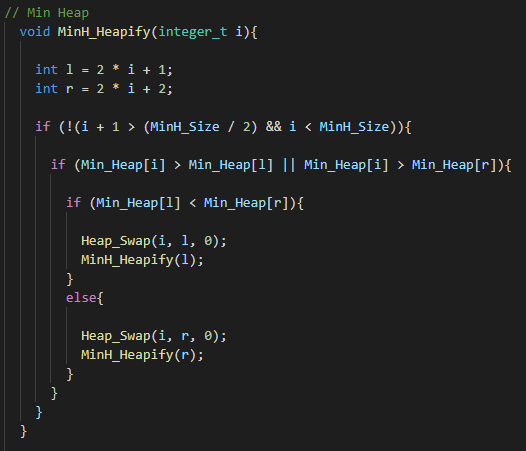
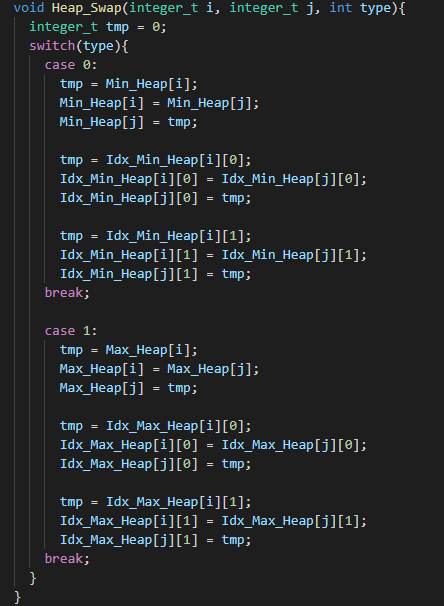
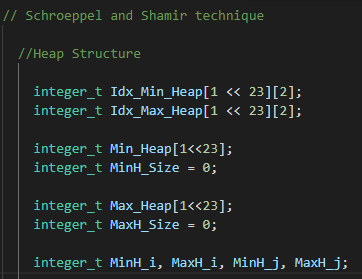
**Brute Forces**

 **

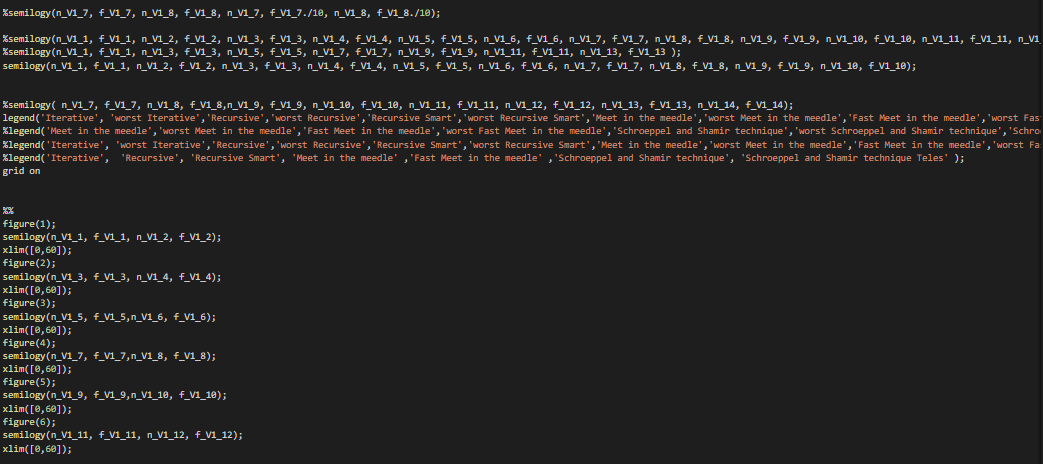
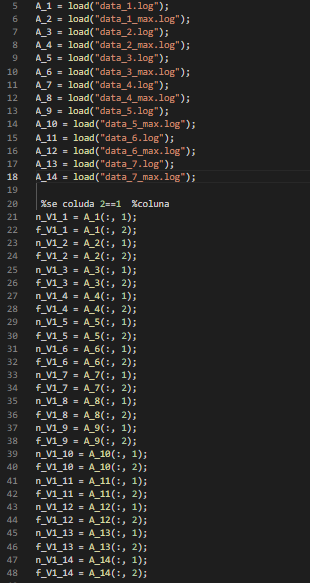
**Meet in the middle**

**Faster Meet in the middle**

**Schroeppel and Shamir**

**

Código Matlab

**

Algumas Soluções Obtidas

**Rafael Remígio 102435 João Correia 104360**

Primeiras 8 soluções para n = 40

1001111010110010001010101000110011101001

0011001111011001100011001010111111011100

0110111000111101011011110110110110000101

0000000001111111010010111010010101101000

1000011011000101101000010111011000110111

0000110111010101001001111100110100110101

1100010011000010001010000001111111010000

1011001100001011110111010111100100000010

Primeira solução n = 57

101010101010111111001000110000010010100100110111110011111

Primeiras 8 soluções para n = 40

1110101110001100011111110001011001111110

0001111101011010110010100001100011110111

0011101001000000011000111111011000001101

1111000100010011101010101110000111001100

1101111010010000110000110000111101011011

1101011011000001110011001011111011100000

0010100101011000101001110001000101110111

0110010001111100110001110001011010010000

Primeira solução n = 57

010001010010010010100100011101001101011101100001101101111

**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/>