**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%;**

**Índice**

***Introdução****\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2*

*Merkle-Hellman cryptosystem and the Subset Sum Problem\_\_\_\_\_\_2*

***Métodos/Algoritmos Utilizados****\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3*

*Brute Force Iterativa\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3*

*Brute Force Recursiva\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3*

*Clever Brute force\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4*

*Meet-in-the-middle\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5*

*Faster version of Meet-in-the-middle\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5*

*Schroeppel and Shamir\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5*

*Material Usado/Bibliografia/Webgrafia\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10*

Introdução

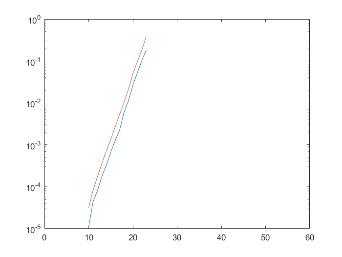
O sistema criptográfico Merkle-Hellman proposto em 1974 foi um dos mais antigos sistemas criptográficos de chave pública e era baseado no problema [**knapsack**](https://en.wikipedia.org/wiki/Knapsack_problem)**,** mais especificamente no **[problema subset-sum](https://en.wikipedia.org/wiki/Subset_sum_problem).** Em 1981, foi publicado por **[Adi Shamir](https://en.wikipedia.org/wiki/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 (2n)
2. Iteramos pelo conjunto somando os inteiros correspondentes a combinação
3. Comparamos esta soma com o soma desejada

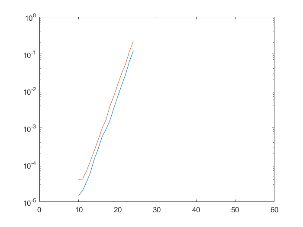
Este método tem o benefício de ser fácil de implementar e de eventualmente chegar sempre a uma solução, porém é muito demoroso tendo a maior complexidade algorítmica de todos os algoritmos de O(n\*2n).

Brute Force Recursiva

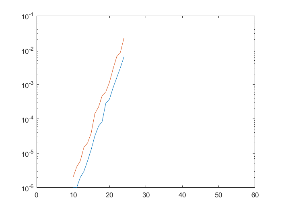
Este método consiste também numa busca intensiva onde percorremos todas as combinações possíveis, no entanto neste caso a chave da combinação e a soma são geradas através de um algoritmo recursivo de “backtracking”. Consideremos o conjunto de inteiros ordenados P e a soma desejada K.

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.

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.

Se a soma igualar a soma desejada a função retornará a combinação chave correspondente.

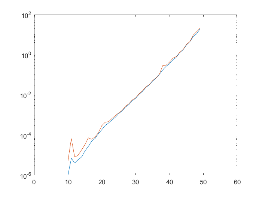
Este método é também bastante intuitivo e fácil de implementar, no entanto mais uma possuí uma alta complexidade algorítmica de O(2^n).

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 e todos 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.

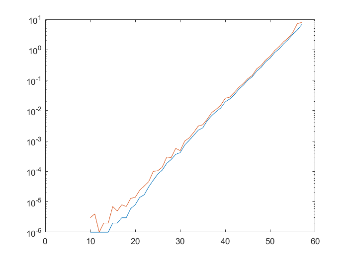
Este método é bastante mais eficaz que o anterior, mas não possui uma complexidade algorítmica definida sendo um algoritmo bastante difícil de analisar.

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 com n/2 elementos cada (n representa o número de elementos totais do conjunto).
2. Cria e, de forma ordenada, guarda todas as combinações e somas possíveis para cada subconjunto.
3. 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 tera 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 dados de **p** são guardados em 4 subconjuntos de igual tamanho.

1. Separa o conjunto em 4 subconjuntos com n/2 elementos cada (n representa o número de elementos totais do conjunto).
2. Cria e, de forma ordenada, guarda todas as combinações e somas possíveis para cada subconjunto.
3. Criamos 2 Max Heaps e 2 Min Heaps (um para guardar as somas e o outro para guardar os indices correspondestes no subconjunto ”i”, ”j”)
4. 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)
5. Depois vamos percorrendo os heaps dando pop e se possivel substituindo a root por (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 contribuo 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.

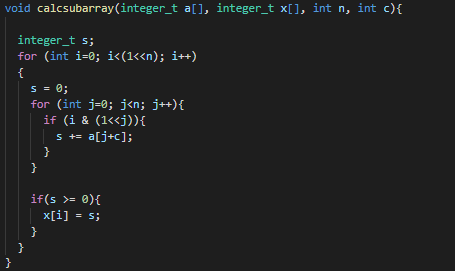
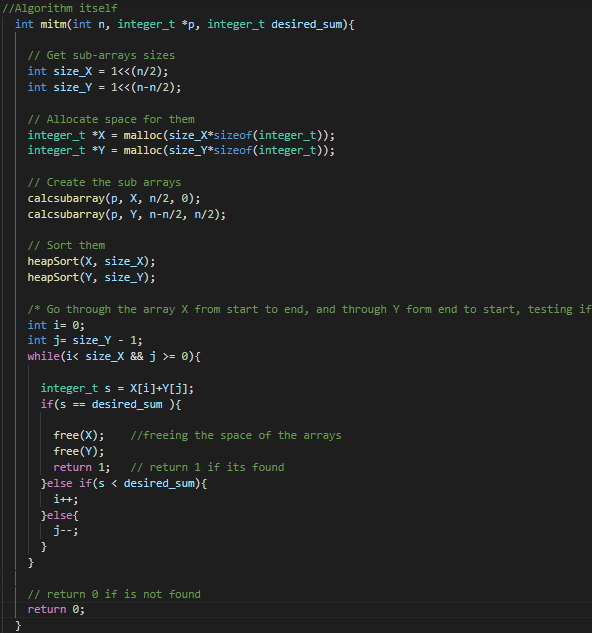
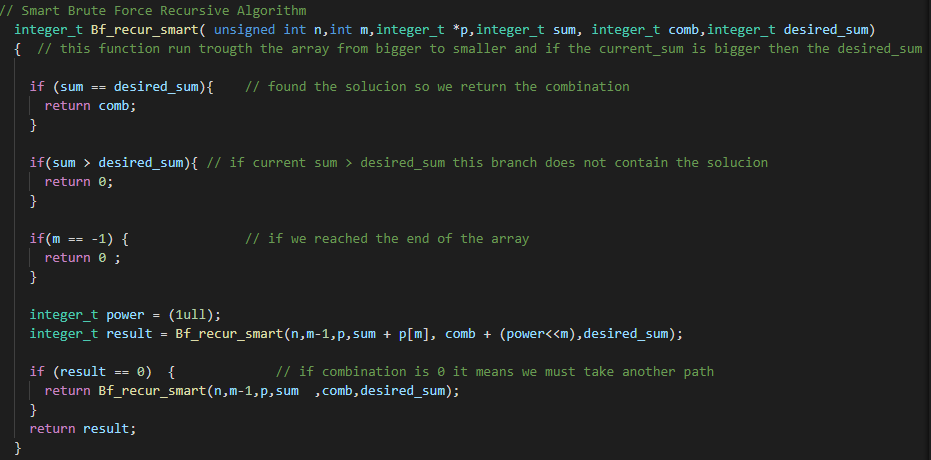
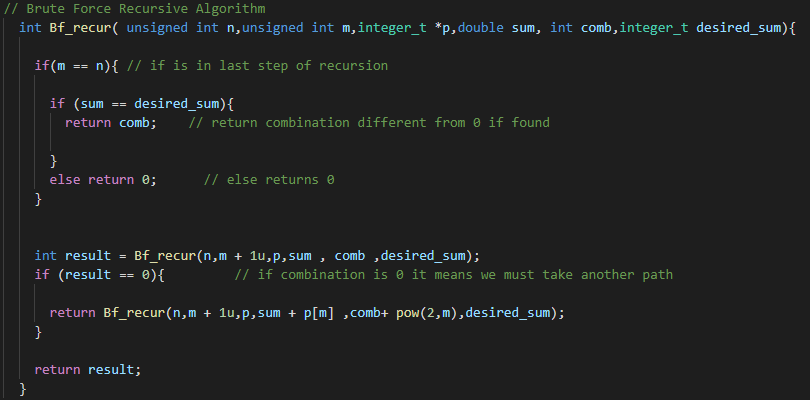
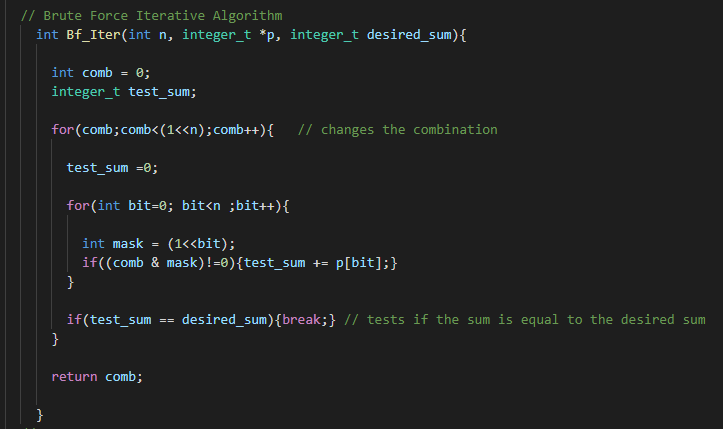
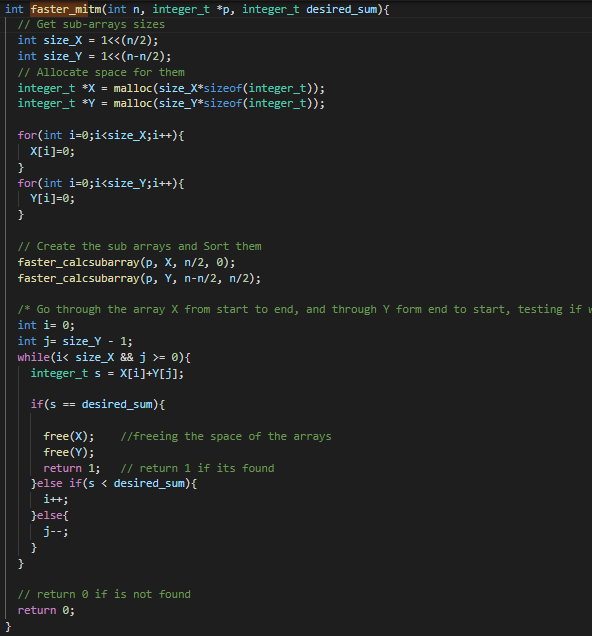
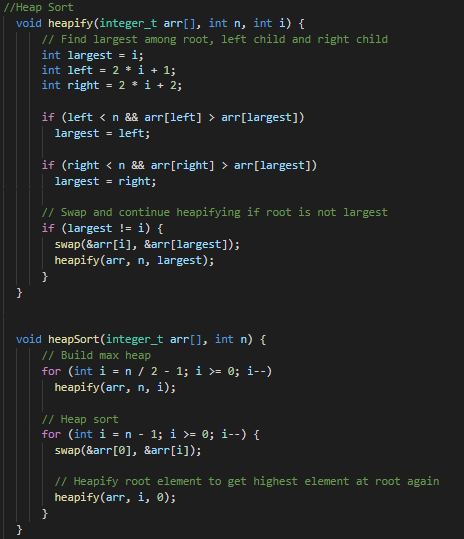
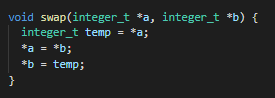
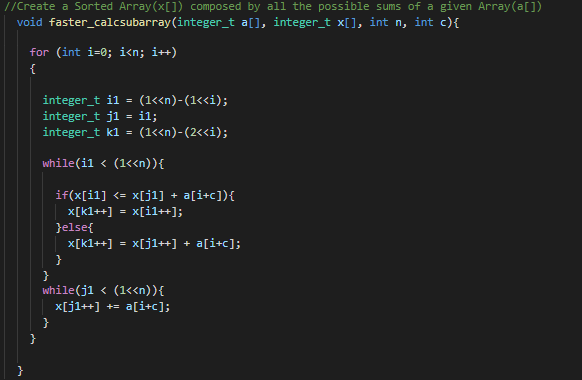
Concluindo, pela análise da complexidade temporal e espacial, podemos dizer que para diferentes números de instâncias diferentes algoritmos são preferíveis. Branch and Bound e Meet in the Middle são os únicos recomendados para um pequeno número de instâncias sendo que o algoritmo Meet in the Middle será muito mais dispendioso em termos espaciais, mas mais 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.

All

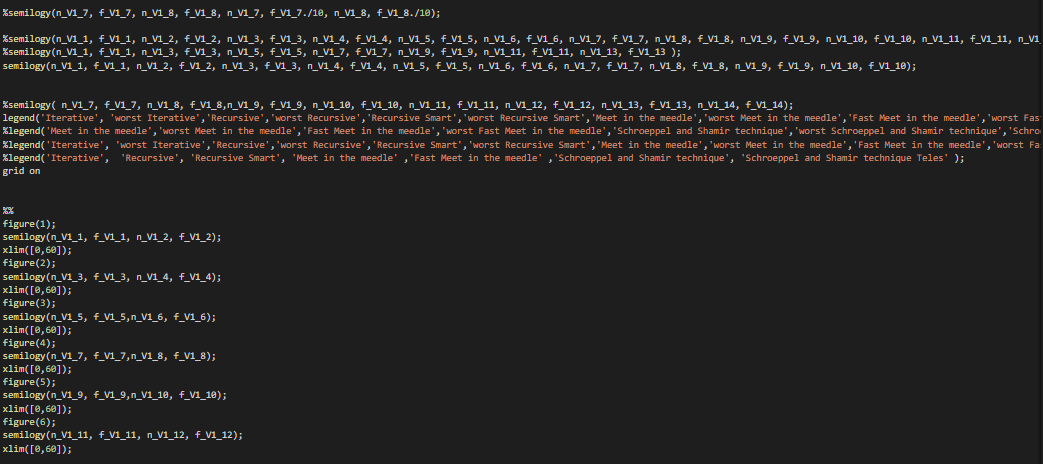
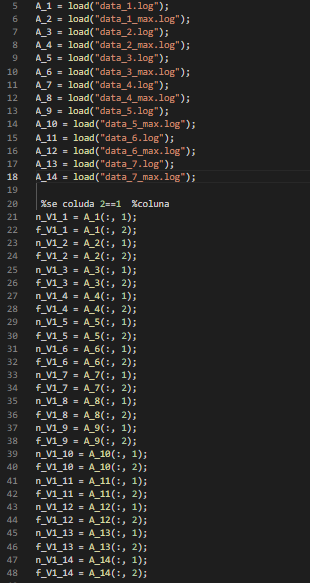
*Chart, line chart

Description automatically generated*

Código utilizado

 **

Código Matlab

**

Alguns Resultados Obtidos