Universidade de Aveiro

Merkle-Hellman cryptosystem

Tiago Portugal 103931

David Cobbilac 104789

Afonso Azevedo 104272

Algoritmos e estruturas de dados - 40437 1º Semestre DETI

09/01/2021

Introdução

O sistema de enciptação Merkel-Hellman knapsack foi um dos primeiros sistemas de encriptação de chave publica a ser inventados sendo criado por Ralph Merkle and Martin Hellman em 1978.
No entanto apesar do metodo ser mais simples que as normas establecidas anteriormente, foram rapidamente encontradas falhas no algoritmo.

O metodo de encriptação deste algoritmo é assimétrico pelo que gera duas chaves, uma **publica** e outra **privada**.

Ex:

> O Tiburcio e Gertruncia querem partilhar um documento confidencial e para isso vão usar o sistema de encriptação Merkle and Hellman.
>
> O sistema de encriptação vai gerar a cada um, um par de chaves, uma privada que não devem partilhar e outra publica para proceder á encriptação.
> O documento vai ser enviado da Gertruncia para o Tiburcio, então para isso , primeiro a Gertruncia terá que pedir a chave publica do Tiburcio.
>
> Após receber a chave a Gertruncia terá que fazer uma copia do ficheiro e encriptar essa copia através do algoritmo de Merkel-Hellman. Esta cópia é necessária pois após ser feita a encriptação com a chave publica do Tiburcio só este a pode desencriptar o documento com a sua chave privada.
>
> No final ela terá de enviar o documento ao Tiburcio que á chegada, ele usará a sua chave privada para proceder á desencriptação.

Encriptação de uma palavra de n bits

Neste trabalho vamos explorar uma verão simplificada deste algoritmo onde o problema de knapsack será substituido por um problema de somas de um subconjunto.

Geração de chaves

A chave privada consiste em 3 elementos:

Super sequencia gerada aleatóriamente.

$$W = \{w_1, w_2, w_3, \ \dots \ , w_n\}$$

$$\text{onde}$$

$$w_{i+1} > \sum_{i=0}^{n} w_i$$

Constante *m* aleatória

$$m > \sum_{i=0}^{n} wi$$

Constante *a* aleatória coprima de *m*

$$mdc(m, a) = 1$$

A chave publica **P** que consiste num array de inteiros

$$P = \{p_1, p_2, p_3, \ \dots \ , p_n\}$$

$$\text{que obtemos ordenando}$$

$$W = \{w_1, w_2, w_3, \ \dots \ , w_n\}$$

$$\text{onde}$$

$$w_i' = a * w_i \bmod m$$

Encriptação

Dado um array **B** de **n** bits o resultado da encriptação será um numero inteiro **C'** tal que:

$$C' = \sum_{i=0}^{n} b[i] * p[i]$$

Desencriptação

Numa situação normal, tendo acesso á chave privada usariamos uma algoritmo euclidiano em com um algoritmo ambisioso "greedy" para eficientemente resolvermos um problema de subsomas para chegar aos dados originais.

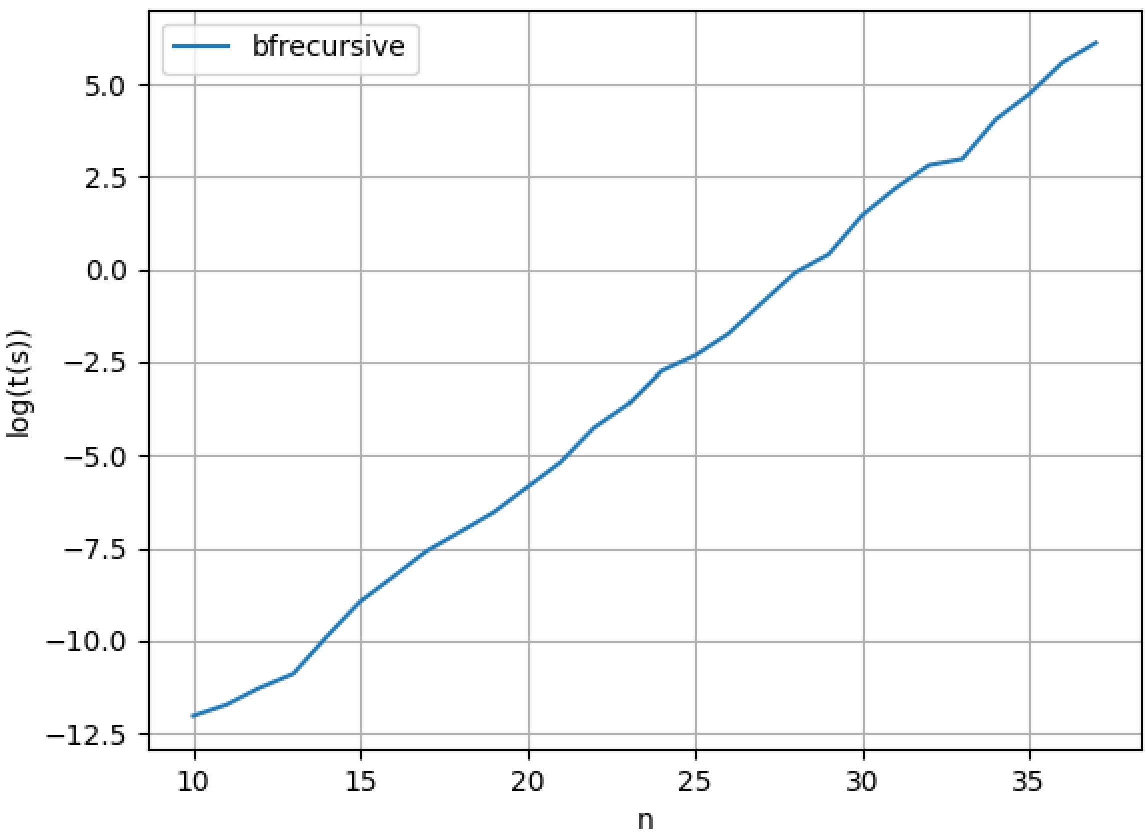
No entanto neste projeto não nos é dada á chave privada, sendo o nosso objétivo criar e explorar as possiveis formas de apartir da chave publica chegar á privada documentando e estudando as complexidades computacionais (temporais e espaciais) dos diferentes metodos.

Brute_Force

Este implementação consiste em percorrer todas as somas possiveis iterativamente até encontarmos a soma correta.
Dada uma chave publica de **n** elementos o conjunto de subsomas possiveis a percorrer será **2^n**, pelo que ,teoricamente, a **Big O notation será 2^n*n** sendo o ultimo n relativo ao ciclo for usado para confirmar a soma.

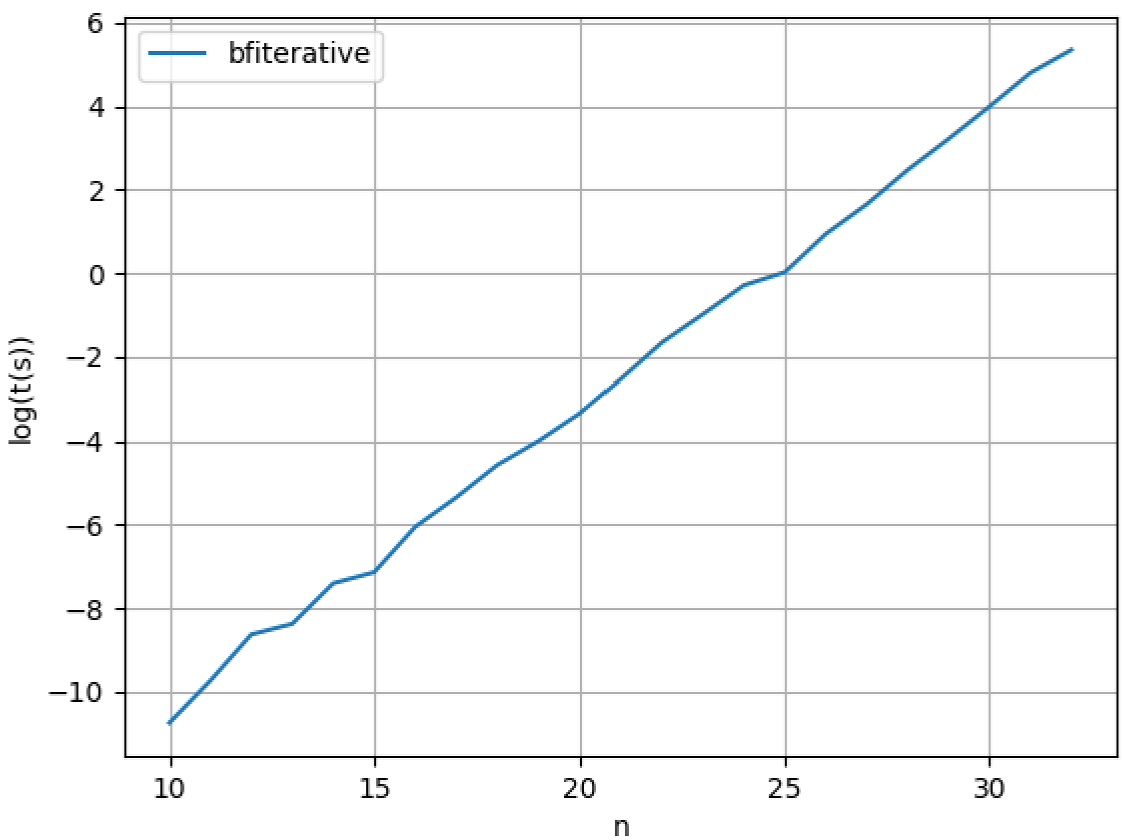
```
int brute_force_recursive(int n, integer_t p[n], integer_t desired_sum, int
idx, integer_t partial_sum, int b[n]){
// verifica valida cada bit dos dados encriptados comforme a soma parcial atá
chegar á soma desejada
        int i;
        if (partial_sum == desired_sum){
                for (i = idx; i < n; i++){
                        b[i] = 0; // mete o resto da chave a zero
                }
                return 1;
        }

        if (idx == n || partial_sum > desired_sum){
                return 0;
        }
```

```
        b[idx] = 0;
        int r = brute_force_recursive(n, p, desired_sum, idx + 1, partial_sum
/* + p[idx]*b[idx] */, b);

        if (r != 0){
                return r;
        }

        b[idx] = 1;
        r = brute_force_recursive(n, p, desired_sum, idx + 1, partial_sum +
p[idx] /* + p[idx]*b[idx] */, b) != 0;
        return r;
}
```

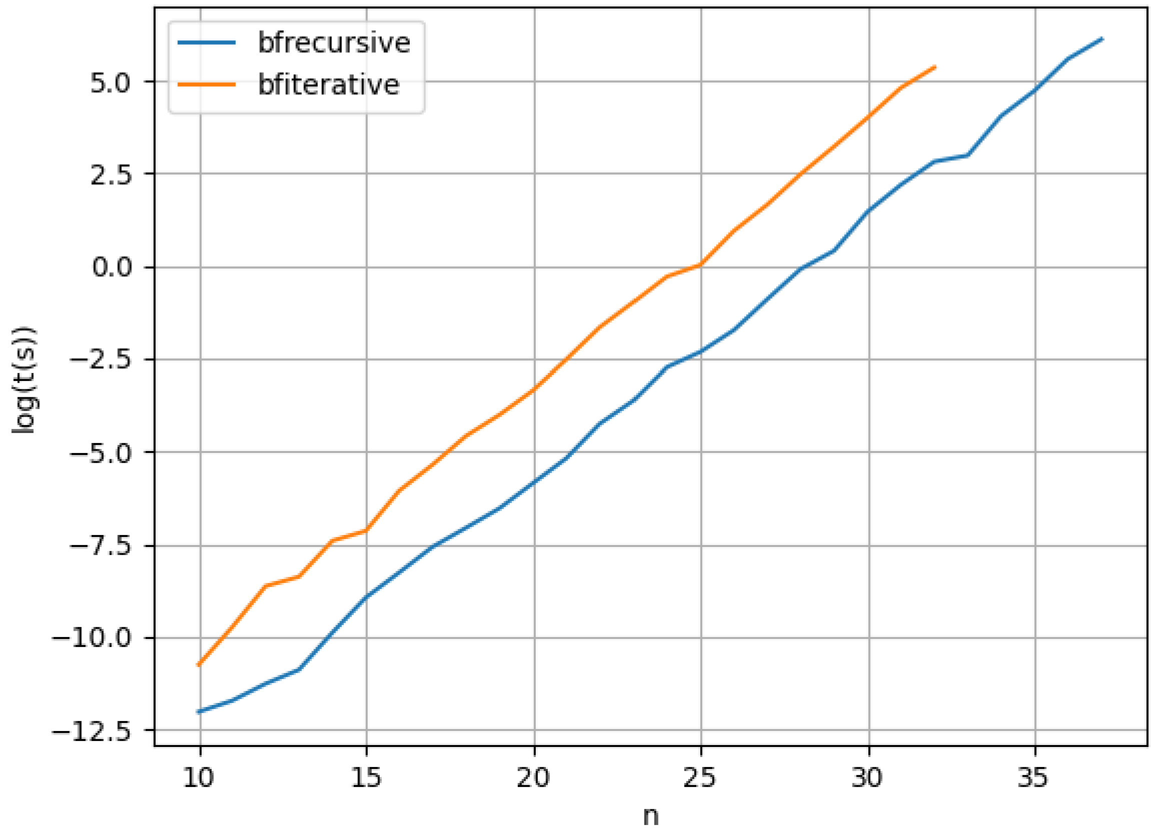



```
int brute_force_iterative(int n,integer_t p[n],integer_t desired_sum,int
b[n]){

        long long int total_sums = 1 << n;
        long long int i,j,sum;

        for(i = 0; i<total_sums;i++){
```

```
                sum = 0;
                for(j = 0; j<n ;j++){
                        if (i & (1 << j)){
                                b[j] = 1;
                                sum += p[j];
                        }
                }

        if(sum == desired_sum){
                return 0;
        }

        memset(b, 0, n*sizeof(int));

        }

        return 1;
}
```



Implementamos este metodo de duas formas, **recursivamente e iterativamente**, quando aplicado em geral o **metodo recursivo** quando verificando exprimentalmente é ligeiramente mais eficiente do que o método recursivo.



Horowitz and Sahni

Baseada no metodo desenvolvido por Ellis Horowitz e Sartaj Sahni para o problema de subsomas, esta implementação ao custo da memória acaba por ser mais efeciente que o **brute force** em termos temporais.

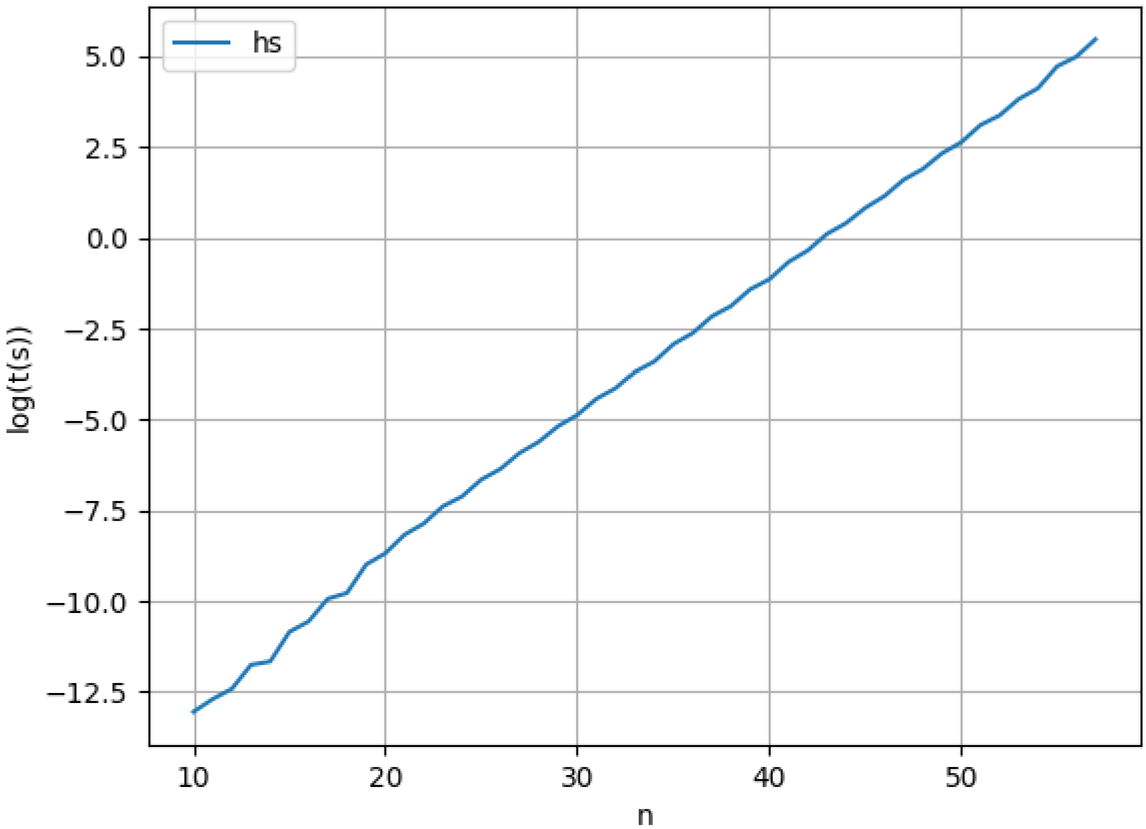
Aqui diminuimos o numero de somas a ser calculadas para **2*2^(n/2)**, dividindo o array de forma relativamente parcial em duas partes, calculando as suas subsomas. Apartir das subsomas por um **metodo de ponteiros ascendente e descendente** sumando os dois valores dos ponteiros até chegarmos a soma desejada (ponteiro ascendente → 1º array, ponteiro descendente → 2º array), com o valor desses ponteiros aplicamos o anterior método **brute force** (recursivo) com o **array de subsomas, subarray e a subsoma** chegando assim á mensagem original.

```
void hs(int n,integer_t p[n],integer_t desired_sum,int b[n]){

        int n1 = n/2;
        int n2 = n - n1;
        integer_t t=1ULL<<n1;
        integer_t* lower = (integer_t*)malloc(sizeof(integer_t)*t);
```

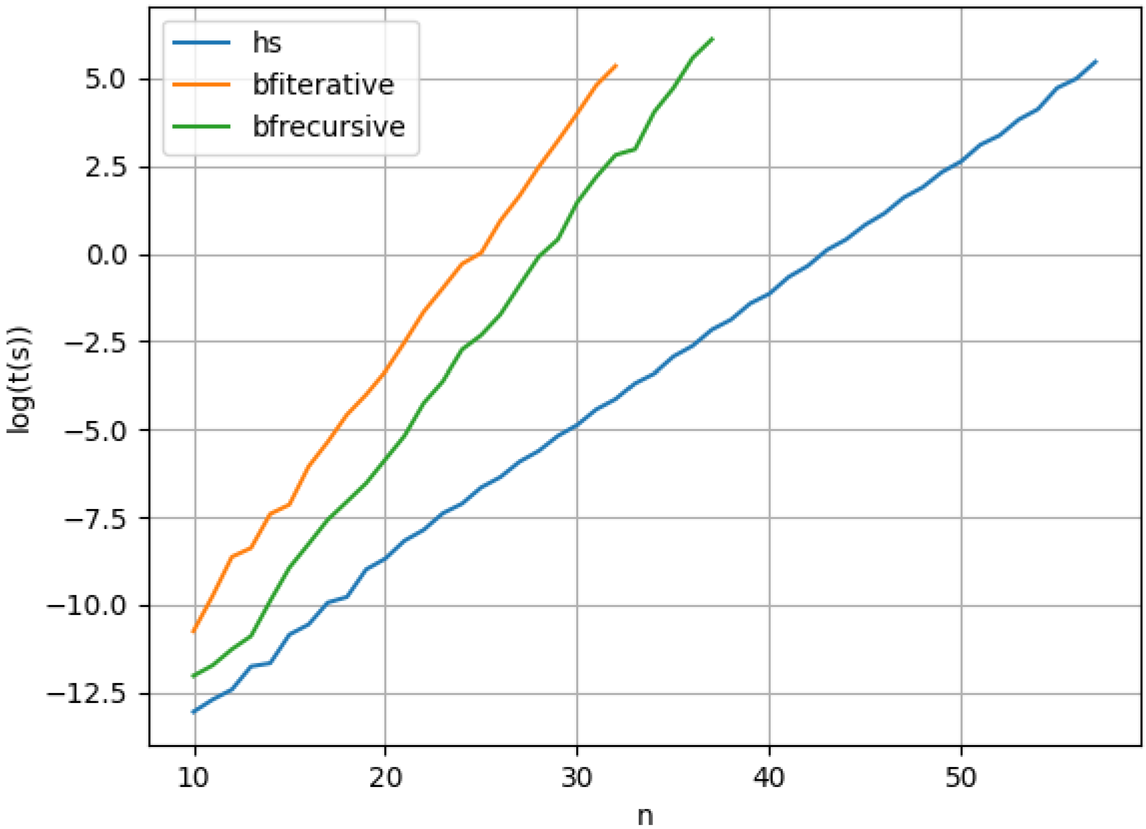
```
        t=1ULL<<n2;
        integer_t* upper = (integer_t*)malloc(sizeof(integer_t)*t);
        integer_t i = 0;
        integer_t j = (1ULL << n2) -1;

        make_sums(n1,p,lower);
        make_sums(n2,p+n1,upper); // função auxiliar no fim do documento

        while(1){
                if(upper[j]+lower[i] == desired_sum){
                        break;
                } else if (upper[j]+lower[i] < desired_sum){
                        i++;
                        continue;
                }
                j--;
        }

        brute_force_recursive(n1,p,lower[i],0,0,b);
        brute_force_recursive(n2,p+n1,upper[j],0,0,b+n1);
        free(lower);
        free(upper);
}
```

Em termos de complexidade computacional temporal este algorimo terá **Big O notation** de **4*2^(n/2)*n/2** (4 → 2*O(n) de brute force) , ignorando as constantes , **2^(n/2)*n/2**, que se confirma exprimentalmente.

E em termos de complexidade computacional espacial este algoritmo terá **Big O notation** de **(2^n/2 *2)**.



hs
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
log(t(s))
10
20
30
40
50
n



Schroeppel and Shamir

Este método é basedo num melhoramente feito por Schroeppel and Shamir que visa melhorar a complexidade espacial do metodo anterior, calculando as subsomas dos arrays **lower** and **upper** iterativamente.

Inicialmente para calcularmos as somas iterativamente,vamos dividir as chave publica em 4 e calculamos as subsomas desses arrays (lwa,lwb;upa,upb), sendo que apartir de agora em vez de termos 2 arrays de 2^(n/2)*sizeof(integer_t) bytes temos 4 de apenas 2^(n/4)*sizeof(integer_t).

```
int n1 = n/2;
int n1a = n1/2;
int n1b = n1 - n1a;
int n2 = n - n1;
int n2a = n2/2;
int n2b = n2 - n2a;

integer_t *lwa = (integer_t*)malloc(sizeof(integer_t)*(1LL<<n1a));
integer_t *lwb = (integer_t*)malloc(sizeof(integer_t)*(1LL<<n1b));
integer_t *upa = (integer_t*)malloc(sizeof(integer_t)*(1LL<<n2a));
```

```c
integer_t *upb = (integer_t*)malloc(sizeof(integer_t)*(1LL<<n2b));

make_sums(n1a,p,lwa);
make_sums(n1b,p+n1a,lwb);
make_sums(n2a,p+(n1a+n1b),upa);
make_sums(n2b,p+(n1a+n1b+n2a),upb);
```

Para calcular e parcialmente amazernar as somas podemos ultilizar duas estruturas **heaps ou priority queues**.
Em que para gerar as somas do **lower** e **upper** no metodo **hs**, iteramos os indices de **lwa** e **upa** por **lwb upb** para uma **min heap** e uma **max heap** ou uma **priority queue crescente** e uma **priority queue decrescente**, respétivamente.
Começamos por preencher estas estruturas iterando pelo primeiro indice de **ia** e o ultimo de **ja** todo os arrays **b**.

```c
for(int ia=0;ia<(1LL << n1a);ia++){
        pair_sum pair = {ia, 0, lwa[ia]+lwb[0]};
        add(lower_heap, pair);
}

for(int ja=0;ja<(1LL<<n2a);ja++){
        pair_sum pair = {ja, (1LL<<n2b)-1, upa[ja]+upb[(1LL<<n2b)-1]};
        add_max(upper_heap, pair);
}
```

Depois á medida que damos poll estraimos novos pares **(ia,ib)** para calcular a soma e depois aumentando o **ib** caso a soma seja menor ,ou novos pares **(ja,jb)** diminuimos o **jb** caso a soma seja maior.
Cada vez que estraimos um par substituimos pelo seguinte até que se encontre a **desired_sum** ou acabem-se os pares.
Isto é algo parecido ao algoritmo dos dois ponteiros em **hs** mas aqui é dinamico.

```c
while(lower_heap->size != 0 && upper_heap->size != 0){

        i = peek(lower_heap);

        j = peek_max(upper_heap);

        sum = i.sum+j.sum;

        if (sum == desired_sum) break;

        if (sum < desired_sum){
```

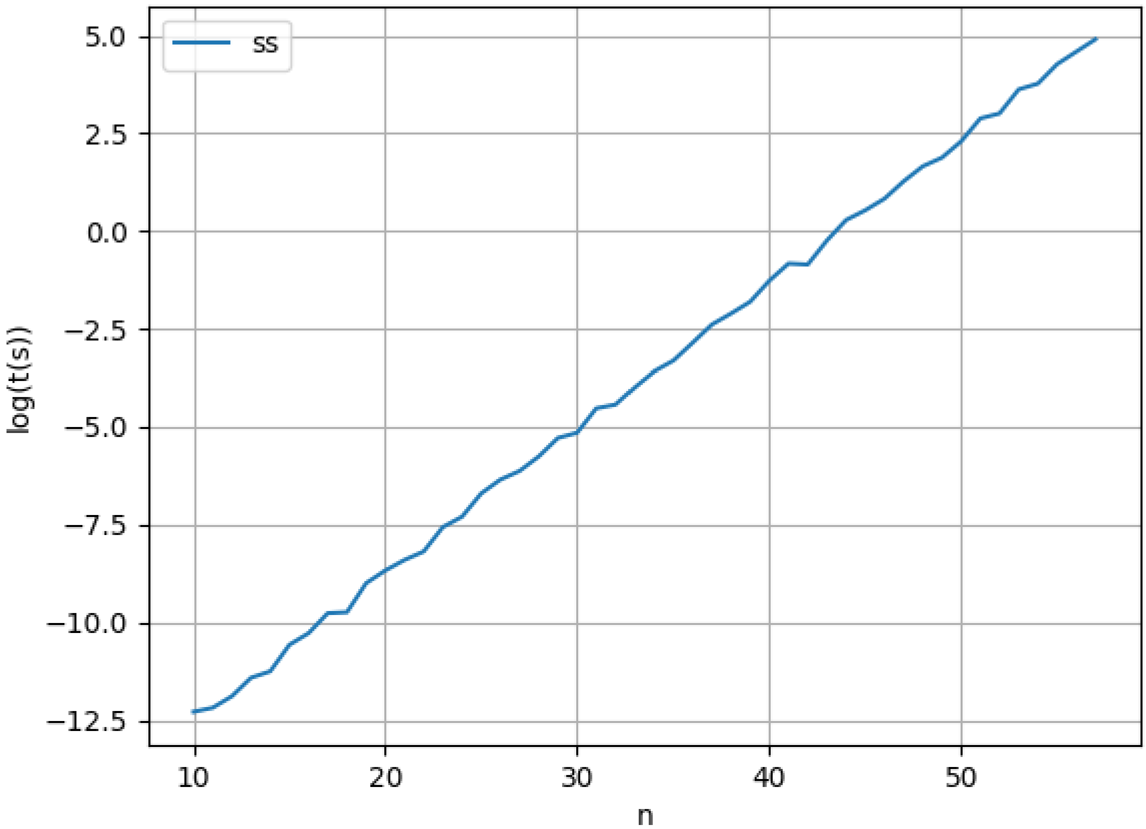
```
                poll(lower_heap);
                if (i.b+1 < (1LL<<n1b)){
                        pair_sum pair = {i.a, i.b+1, lwa[i.a]+lwb[i.b+1]};
                        add(lower_heap, pair);
                }
        }

        if (sum > desired_sum){
                poll_max(upper_heap);
                if ((j.b-1) != -1){
                        pair_sum pair = {j.a, j.b-1, upa[j.a]+upb[j.b-1]};
                        add_max(upper_heap, pair);
                }
        }
}
```

Inicialmente tentamos por aplicar o metodo através de **heaps** no entanto não após de encontrar as subsomas certas não encontramos nenhuma maneira viavel de encontrar os indices referentes á soma em **lwa,lwb,upa,upb** pelo que semalhaça das **heaps com as priority queue** fizemos uma simples transformação, como é evidenciado pelo códico de origem ("if it works, dont touch it!!!") passando o array de ints das **heaps** para um array de uma estura chamada **pair_sum** que amazerna os **indices e a soma**, soma apartir da qual operamos a **"heap"**.
Além disso consultando o que pensamos ser o estudo original conduzido por [Schroeppel and Shamir](), também são ultilizadas as **priority queues**.

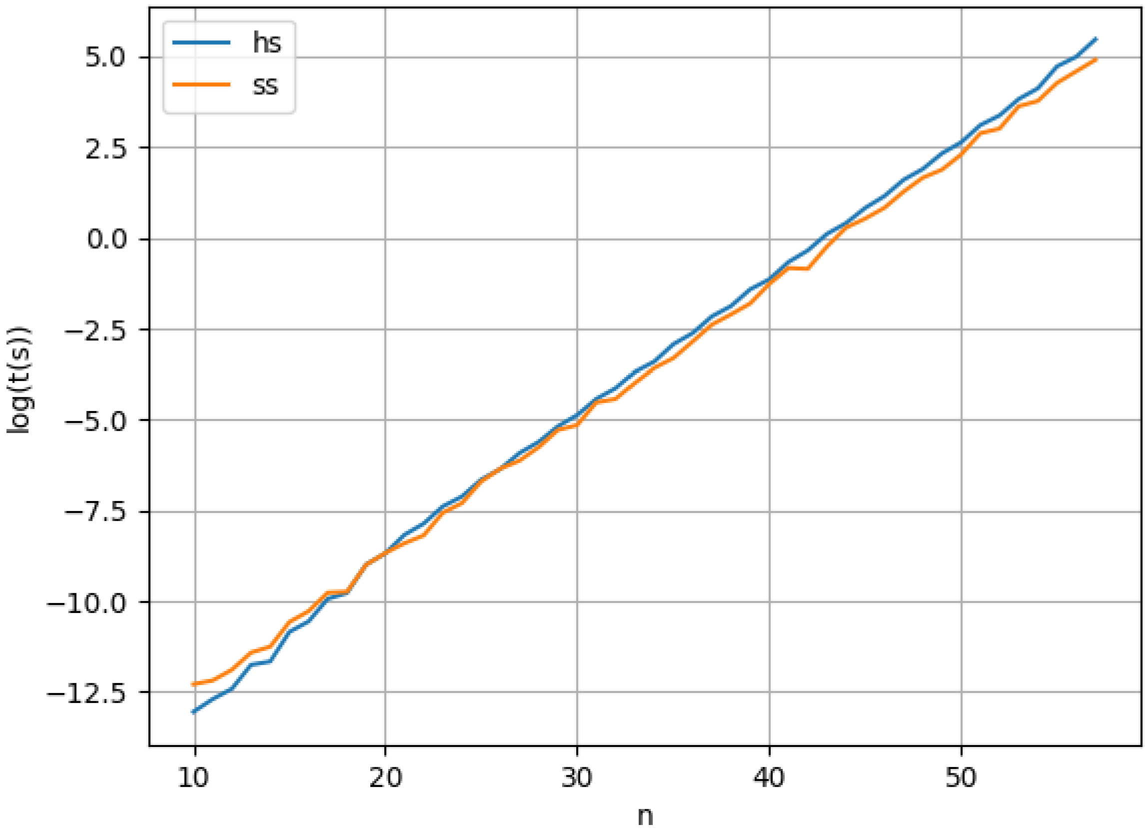
Depois assim com no **hs** recorremos ao brute_force_recursive para fazer os resto do trabalho.

```
brute_force_recursive(n1a,p,lwa[i.a],0,0,b);
brute_force_recursive(n1b,p+n1a,lwb[i.b],0,0,b+n1a);
brute_force_recursive(n2a,p+(n1a+n1b),upa[j.a],0,0,b+(n1a+n1b));
brute_force_recursive(n2a,p+(n1a+n1b+n2a),upb[j.b],0,0,b+(n1a+n1b+n2a));
```

O desempenho deste metodo segue a seguinte equação log temporal



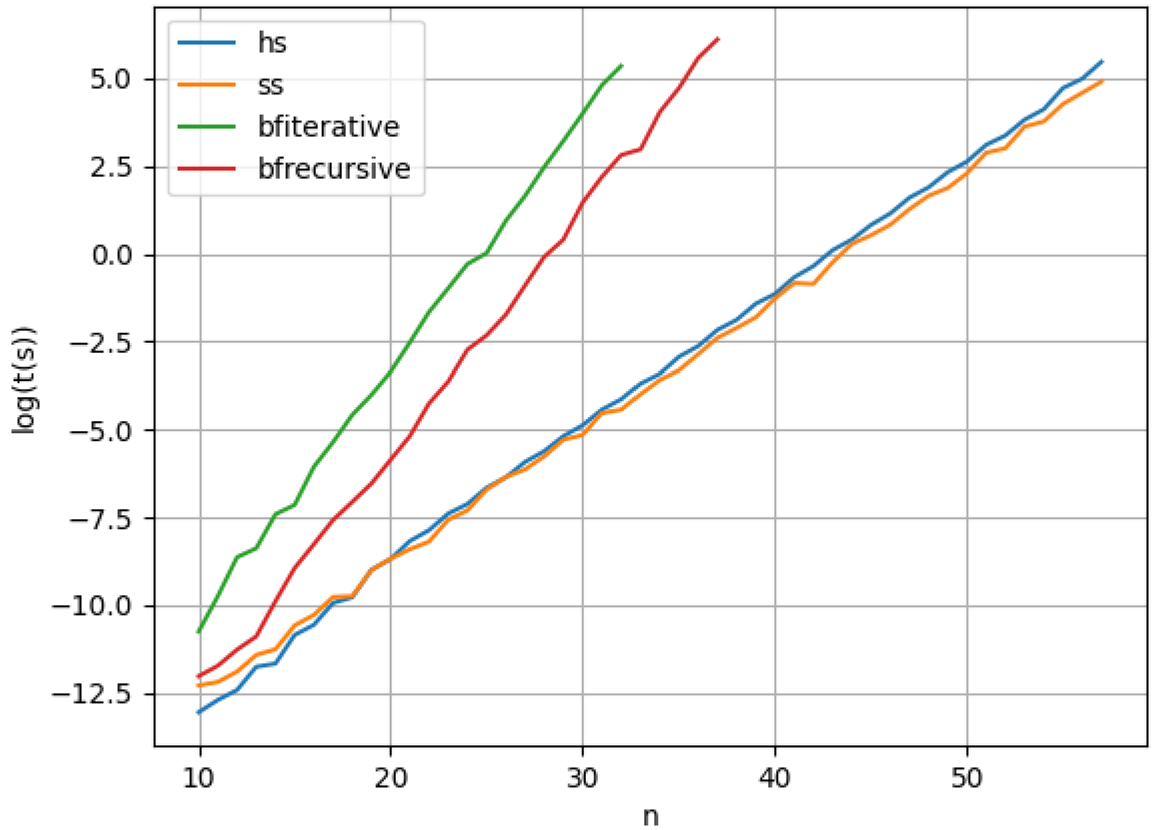
A o desenpenho deste algoritmo apesar de ter como objétivo melhorar apenas a complexidade espacial chega a superar ligeiramente em complexidade temporal o metodo **hs**



No entanto a verdadeira vantangem deste métedo é que pode ir a valores de **n** muito maiores devido ás limitações de memória do metodo de **Horowitz and Sahni**.

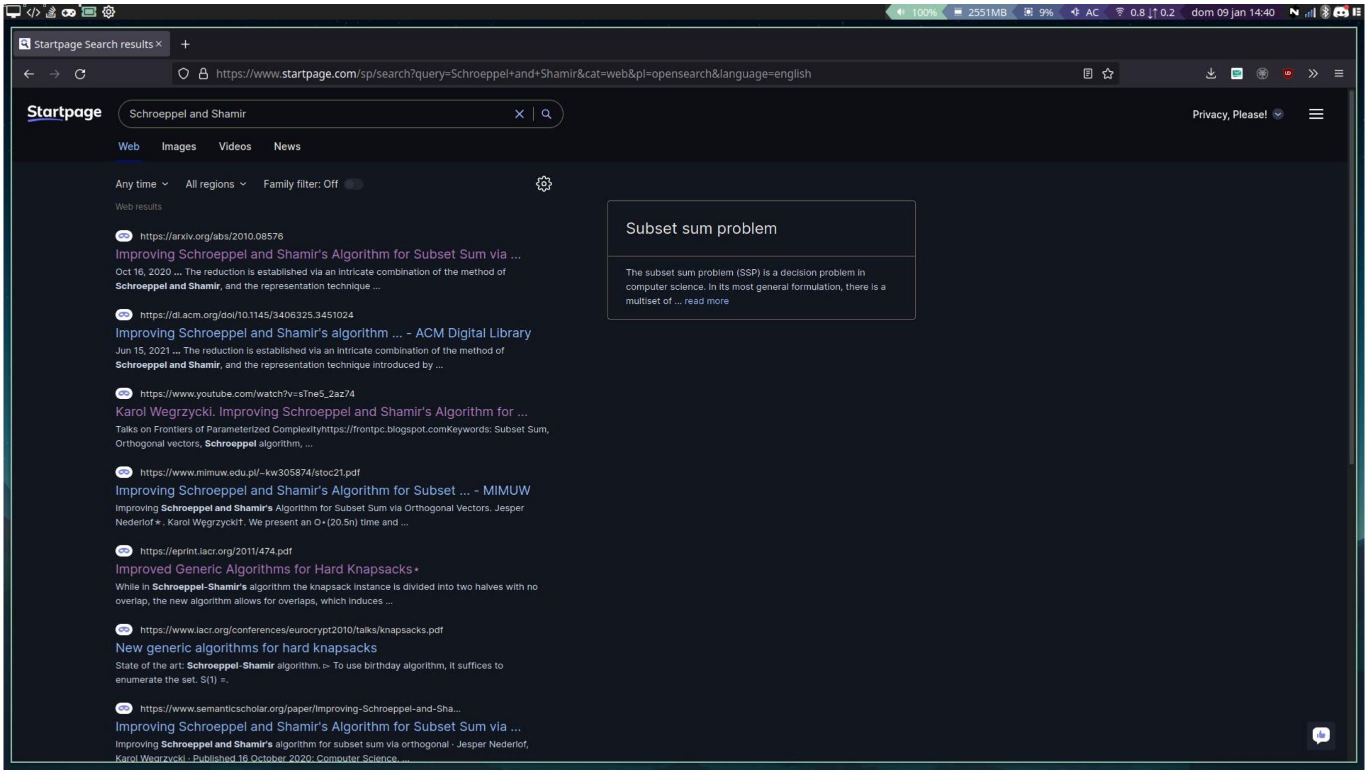
Conclusão

Penso que neste estudo ficou claro que a metedologia com que abordamos um problema computacionalmente complexo pode afetar drasticamente o desenpenho da solução.




```
bfrecursive -> m=0.6899559192527079 b = -19.447056305602256
bfiterative -> m=0.7193673730963013 b = -17.64530981437745
hs -> m=0.38767328582093785 b = -16.60737278635058
ss -> m=0.3711995416075664 b = -16.191157005468522
```

Além disso ao estudar algoritmos relativamente antigos (mais velhos que todos neste grupo), apercebemo-nos que apesar diferentes linguagens e frameworks que vão aparecendo ao longo do tempo que revolucionam ou revolucionaram o mundo nas mais diferentes áreas do desenvolvimento de software, algoritmos quando de qualidade são tão relevantes como sempre foram, podemos ver isso na pesquisa que fizemos a tentar encontrar pistas nomeadamente para o ultimo algoritmo onde a maioria dos resultados referiam-se não ao metodo em si, mas sim a um melhoramento relativamente recente.



Todo o códico e documentação no desenvolvimento deste trabalho está no seguinte repositório no github.

[Merkle_and_Hellman_decryptosystem]

Bibliografia

A Hybrid Recursive Multi-Way Number Partitioning Algorithm - https://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.208.2132

Schroeppel and Shamir reshearch - https://apps.dtic.mil/sti/pdfs/ADA080385.pdf