TR - 02

Paula Rehbein
Matrícula: 202411179
3° semestre
Turma 30

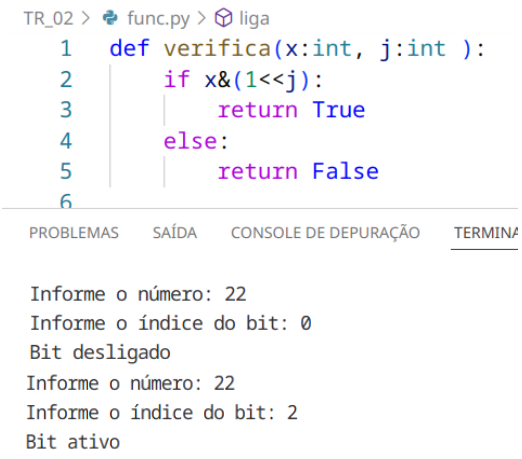
Bitmask ou Pesquisa de Máscara de Bits

1 Funcionamento passo a passo

É uma técnica que pode ser usada para testar todas as possibilidades de uma situação. A Bitmask usa a representação binária de um número, onde os bits são indexados iniciando em 0 e da direita para a esquerda. Podemos usar essa representação para se referir a acontecimentos, sendo cada 0 e 1 um acontecimento, com 0 como não ocorrido e 1 para ocorrido.

Por exemplo, temos uma lista de compras com 5 produtos e queremos verificar se todos os produtos foram comprados. Se o produto do índice 1 e o produto do índice 4 não foram comprados a representação seria 01101 = 18 (em decimal), agora se apenas os produtos do índice 2 e do índice 0 foram comprados a representação seria 00101= 5 (decimal). Ou se todos tivesse sido adquiridos a representação seria 11111 = 31.

Inicialmente, o que significa `(1<<j)` usado em todas as equações? Esse código é um shift left, ou seja, ele move o 1 para a esquerda *j* vezes no número binário, o que é equivalente a 2^*j*.[1] Isso vai ser usado para comparar casos isolados, pois precisamos fazer operações lógicas com um bit em determinada posição, e para isso ser possível precisamos de um número binário com o bit, na mesma posição, ativo. Vejamos as aplicações para entender:



- Verificação: verificar se algo aconteceu olhando o seu bit correspondente. Seguindo o exemplo da lista de compras temos os itens: sabonete (índice 0), farofa (índice 1), café (índice 2), erva-mate (índice 3) e pão (índice 4), sendo cada item um número 0 ou 1. Digamos que a lista de itens comprados esteja dessa forma 10110 = 22 e queremos verificar se foram comprados café e sabonete, para isso verificamos o valor de seu índice (representado por j), na lista (representado por x) `x&(1<<j)` ou seja, verificamos se o elemento da posição j está ativo em x.
Representando no código ao lado.
Podemos ver no código que o sabonete (índice 0) não foi comprado pois seu bit está desligado. Diferente do café (índice 2) que foi comprado, bit ligado.
Como conseguimos verificar isso? Bom, o número binário da lista x (no exemplo é 10110) e o número 1<<j (ou seja, 2^j), que no caso do café é 00100, sofrem a operação lógica *AND*, o que retorna 0 (*false*) se o bit estiver desligado (0 *and* 1 = 0), e 1 (*true*) se o índice estiver ligado (1 *and* 1 = 1).

10110 (lista atual)
and 00100 (café, valor = 4)
00100 (não retornou 0, assim é *true*, o café está na lista)

- Ligar Bit: Agora digamos que em nossa lista de compras (sabonete, farofa, café, erva-mate e pão) representada por 10110, queremos ligar o bit do sabonete (índice 0), pois ele foi comprado. Para isso

[1] Disponível em: < Operador de deslocamento para a esquerda em C++>

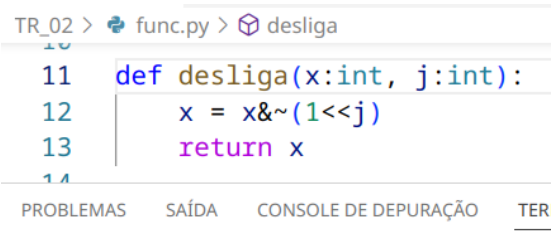
aplicamos a operação `x |=(1<<j);`, ou seja, aplicamos a operação lógica *or* com o número 1 ao índice que queremos ligar, o que resulta em 1 (1 *or* 0 = 1):


```
Informe o número: 22
Informe o índice do bit: 0
Número atualizado: 23.
```

No nosso caso, retornou o valor atualizado da lista (23 = 10111), pois tendo 10110 = 22, ao ligarmos o valor do índice 0 resulta em 10111 = 23 (2^0 = 1 e 22 + 1 = 23).

10110 (lista inicial, valor = 22)
or 00001 (sabonete, valor = 1)
10111 (lista final, retornou 23 pois é o resultado de *or* entre 1 e 22).

- Desligar Bit: Agora na nossa lista (sabonete, farofa, café, erva-mate e pão), atualmente 10111=23, percebemos um erro, não foi comprado pão (índice 4), ou seja, temos que desligar seu bit correspondente. Isso pode ser feito com `x = x&~(1<<j);`,o que é nada mais que *x and* negação de 1 na posição *j*, o que retorna *false* (~1=0, assim, ~1 *and* 1 = 0), vejamos:

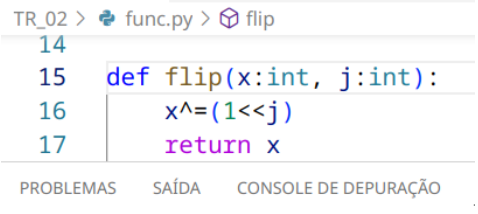



```
Informe o número: 23
Informe o índice do bit: 4
Número atualizado: 7.
```

Como desligamos o bit da posição 4, o retorno é 7 pois 2^4 = 16 (o valor do pão na lista), e 23 - 16 = 7, ou seja, 00111 = 7 é a representação da lista de itens comprados sem o pão. Ex.:
~(10000) = 01111

10111 (lista inicial, valor = 23)
and 01111 (lista sem pão = 15)
00111 (lista final, retornou 7).

- Operação Flip: essa operação é usada para ligar um bit desligado e desligar um bit ligado, ela faz isso aplicando a operação lógica *xor* ao índice que escolhemos: `x = x^(1<<j);`. Por exemplo, temos nossa lista atual (sabonete, farofa, café, erva-mate e pão) representada por 00111 = 7, e acabamos de ir no mercado novamente e compramos a erva-mate (índice 3), veja o código:




```
Informe o número: 7
Informe o índice do bit: 3
Número atualizado: 15.
```

O retorno é 15, pois, a lista (00111 = 7), mais o elemento erva-mate (2^3 = 8), resulta em 01111 = 15 (7 + 8 = 15). Ex.:

00111 (lista inicial, valor = 7)
xor 01000 (valor da erva-mate = 8)
01111 (lista final, retornou 15).

Seguindo o exemplo, chegamos em casa e percebemos que o café comprado é muito pequeno, e precisamos comprar mais, para isso queremos desligar seu bit correspondente na lista, para indicar que não foi comprado, assim segue o código:

```
TR_02 > func.py > flip
14
15  def flip(x:int, j:int):
16      x^=(1<<j)
17      return x
PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO

Informe o número: 15
Informe o índice do bit: 2
Número atualizado: 11.
```

Podemos perceber que o retorno foi 11 pois a lista (01111 = 15) sem o café (índice 2, tendo valor $2^2 = 4$) resulta em 15 - 4 = 11, sendo a lista final 01011.Ex.:

```
  01111   (lista inicial, valor = 15)
xor 00100  (valor do café = 4)
  01011  (lista final, retornou 11).
```

2 Complexidade (melhor, médio e pior caso) de BitMask DP (Programação dinâmica com BitMask)

Antes de mostrar a complexidade precisamos ver o caso do caixeiro viajante, que é um exemplo da aplicação da máscara de bits:

Imagine que um caixeiro viajante precisa visitar várias cidades diferentes. Ele começa em uma cidade qualquer e precisa visitar todas as outras exatamente uma vez e depois voltar para casa — tudo isso gastando o menor tempo ou custo possível.

O plano do caixeiro (como pensamos o algoritmo) O caixeiro pensa assim:

1. Preciso lembrar por onde já passei para não visitar a mesma cidade duas vezes.
2. Preciso comparar todas as rotas possíveis para achar a mais curta.
3. Mas há muitas rotas! Então ele pensa: "E se eu for guardando os melhores caminhos aos poucos?"

Como ele organiza sua memória (bitmask). Agora, ele decide criar uma tabela de memória, onde ele guarda:

- ➢ Quais cidades ele já visitou (ele faz isso usando uma máscara de bits).
  - Por exemplo, se ele visitou as cidades 0, 2 e 3 de 4 cidades, ele anota isso como 1101 (em binário).
- ➢ Em que cidade ele está agora.
- ➢ E o menor custo para chegar nessa situação.

Considerando o caso do caixeiro viajante vamos a complexidade de implementação da BitMask:

2.1 Complexidade de tempo de melhor caso considerando o caso do caixeiro viajante:

Segundo MOHAN (2025)[2] melhor caso ocorre quando a poda (cortando caminho) precoce reduz cálculos desnecessários, como:

- Usando memorização eficiente para evitar recomputação de estado.
- Ignorar transições quando um caminho ideal é encontrado precocemente.

Nesse caso a complexidade pode ser dita $O(2^n)$.

2.2 Complexidade média considerando o caso do caixeiro viajante:

De acordo com MOHAN(2025), para uma implementação bem otimizada com poda e armazenamento cache, o desempenho prático fica próximo de $O(2^n * n)$, mas em alguns cenários pode estar mais próximo de $O(1,5^n)$ devido a poda de estados redundantes.

[2] Disponível em:<BitMask DP>.

2.3 Complexidade de tempo no pior caso considerando o caso do caixeiro viajante:

Segundo MOHAN(2025), se for processado todos os subconjuntos possíveis e fazendo a transação entre os estados a pior complexidade é $O(2^n * n)$. Isso pois existem 2^n subconjuntos possíveis (pois cada elemento pode ser incluído ou não), e para cada subconjunto, é inteirado sobre os n elementos para decidir as transições.

3 Casos de uso e aplicações práticas do BitMask

3.1 Casos de uso

- ➢ Uma forma de uso complexa de busca de máscara de bits é em trajetórias na migração de Banco de Dados, onde o Bit Mask Search consegue compactar os dados melhorando a eficiência de algoritmos de mineração (GEETHA, RAMARAJ, 2013).
- ➢ Segundo Ortiz (2018), o BitMask pode ser usado em Estrutura-Empacotamento para Eficiência de Rede onde pode ser aplicado o mascaramento de bits para economizar espaço e melhorar eficiência de rede “empacotando” os sinalizadores dos pacotes[3].
- ➢ O uso da BitMask pode otimizar o uso de memória em sistemas de recursos limitados (KODICHATH, 2020)[4]

3.2 Aplicação prática

- Combinação de conjuntos: formar um conjunto que respeite critérios. Ex.:

```
'''2. Problema de mochila (versão pequena)
Você tem itens com certos pesos, e uma mochila com capacidade limitada.
Use bitmask para testar todas as combinações possíveis de itens e encontrar o maior valor
total sem ultrapassar o peso máximo.'''

v = []
resp = 0
maior = []
quantidade_itens = int(input("Quantidade de itens: "))
peso_maximo = float(input("Peso máximo em Kg que a mochila aguenta: "))

for i in range(quantidade_itens):
   v.append(float(input(f"Peso do item {i+1} em Kg: ")))

for mask in range((1 << quantidade_itens)):
   soma = 0
   itens = []
   for i in range(quantidade_itens):
       if(mask &(1<<i)):
           soma+=v[i]
           itens.append(i+1)

   if  soma <= peso_maximo:
       if len(maior)<len(itens):
           maior = itens

print("Quantidade de itens máximos que cabem na mochila: ",len(maior))
print("Itens", maior)
```

[3] Disponível em: <Afinal, máscaras de bits não são tão esotéricas e impraticáveis...>.
[4] Disponível em: <Usos reais de operadores bit a bit>

4 Comparações com algoritmos semelhantes

A semelhança desses algoritmos ao Pesquisa de Máscara de Bits é que ambos usam representação binário para melhorar desempenho. A seguir os algoritmos semelhantes:

- Bit Stream Mask Search (BSMS): Algoritmo usado em mineração de conjunto de itens frequentes. Ele transforma um arquivo de entrada em dados numéricos, que em seguida é compactado em um array para processamento posterior. Essa abordagem aumenta a eficiência geral de outros algoritmos de mineração, como o apriori, em termos de complexidade temporal e espacial (RAMARAJ, VENKATESAN, 2009). A diferença entre ele e o Bit Mask Search é que ele usa técnicas de BM Search para uso específico.
- N-MostMiner e Top-K-Miner: Algoritmo que visa aumentar a eficiência de mineração de conjuntos de itens frequentes utilizando representação de vetores de bits (BASHIR, JAN, BAIG, 2009). Sua diferença com o Bit Mask Search é que ele usa representação de vetores de bits, enquanto o Bit Mask Search se limita a elementos. N-MostMiner e Top-K-Miner usam técnicas de BitMask porem não são o mesmo algoritmo.

5 Links complementares

Bitmask Search ou Pesquisa de Máscara de Bits:

- Bitmasks: Uma maneira muito esotérica (e impraticável) de gerenciar booleanos
- Afinal, máscaras de bits não são tão esotéricas e impraticáveis...
- Bit Mask Search Algorithm for Trajectory Database Mining

Vídeos:

- Bitmask em C++
- Mascara de Bits C/C++

Binary Tree Sort

1 Funcionamento passo a passo

A Binary Tree Sort é uma estrutura de dados em árvore, onde cada nó possui, no máximo, dois nós filhos, que são chamados de filho esquerdo e filho direito. Existem vários tipos de árvores binárias. Pode ser representada em matriz ou em lista encadeada.

A Binary Tree Sort (árvore de ordenação binária) é um algoritmo de classificação que cria uma Binary Tree Search (árvore de pesquisa binária) e a partir dos elementos de entrada, ele percorre a árvore para que os elementos já sejam classificados quando adicionados.

1.1 Algoritmo:

```
1. class NoDaArvore:
2.     def __init__(self, item=0):
3.         self.valor = item
4.         self.esquerda, self.direita = None, None
5.
6. raiz = NoDaArvore()
7. raiz = None
8.
9. def inserir(valor):
```

```
10.    global raiz
11.    raiz = inserirGalho(raiz, valor)
12.
13. def inserirGalho(raiz, valor):
14.    if raiz == None:
15.        raiz = NoDaArvore(valor)
16.        return raiz
17.    if valor < raiz.valor:
18.        raiz.esquerda = inserirGalho(raiz.esquerda, valor)
19.    elif valor > raiz.valor:
20.        raiz.direita = inserirGalho(raiz.direita, valor)
21.    return raiz
22.
23.
24. def mostraEmOrdem(raiz):
25.    if raiz != None:
26.        mostraEmOrdem(raiz.esquerda)
27.        print(raiz.valor, end=" ")
28.        mostraEmOrdem(raiz.direita)
29.
30. def arvore(arr):
31.    for i in range(len(arr)):
32.        inserir(arr[i])
33.
34.
35. arr = [5, 4, 7, 2, 11]
36. arvore(arr)
37. print(mostraEmOrdem(raiz))
```

Explicando cada parte do código: vamos por partes, acompanhando o fluxo do código.

- Linhas 35 e 36:

```
arr = [5, 4, 7, 2, 11]
arvore(arr)
```

Nessas duas linhas é definido um array de números inteiros e chamada a função *arvore* passando o array como argumento.

- Linha 30:

Função arvore recebe um array como argumento e percorre cada elemento desse array passando-os como argumento na função inserir:

```
def arvore(arr):
    for i in range(len(arr)):
        inserir(arr[i])
```

- Linha 9:

A função inserir recebe o valor, chama a variável global *raiz* e passa *raiz* e a variável *valor* como argumentos na função inserirGalho:

```
def inserir(valor):
    global raiz
```

```
raiz = inserirGalho(raiz, valor)
```

- Linha 13:

inserirGalho recebe os parâmetros raiz e valor. Ela verifica se *raiz* guarda algum dado ou não (ou seja, nulo), e caso for nulo, *raiz* recebe uma nova instância da classe *NoDaArvore*, guardando o dado da variável *valor* e recebendo um galho direito e esquerdo:

```
class NoDaArvore:
    def __init__(self, item=0):
        self.valor = item
        self.esquerda, self.direita = None, None

def inserirGalho(raiz, valor):
    if raiz == None:
        raiz = NoDaArvore(valor)
        return raiz
```

Mas caso o dado da *raiz* não for nulo (o que significa que *raiz* já tem galho direito e esquerdo e guarda algum dado), e for maior que o dado da variável *valor*, então o galho esquerdo de *raiz* recebe uma nova instância da classe *NoDaArvore*, guardando o dado da variável *valor* e recebendo um galho direito e esquerdo. Porém caso o dado da *raiz* não for maior que o dado da variável *valor*, então o galho direito de *raiz* recebe uma nova instância da classe *NoDaArvore*, guardando o dado da variável *valor* e recebendo um galho direito e esquerdo:

```
class NoDaArvore:
    def __init__(self, item=0):
        self.valor = item
        self.esquerda, self.direita = None, None
def inserirGalho(raiz, valor):
    if raiz == None:
        raiz = NoDaArvore(valor)
        return raiz
    if valor < raiz.valor:
        raiz.esquerda = inserirGalho(raiz.esquerda, valor)
    elif valor > raiz.valor:
        raiz.direita = inserirGalho(raiz.direita, valor)
    return raiz
```

É nessa parte do código que a ordenação da árvore binária acontece, pois os dados menores que o dado anterior já armazenado na árvore são armazenados nos galhos à esquerda dos nós, enquanto os dados com um valor maior que os dados já presentes na árvore são armazenados nos galhos à direita dos nós. Levando a próxima etapa do algoritmo.

- Linha 28:

A função *mostraEmOrdem* é a função que percorre a árvore binária ordenada e mostra no console os valores ordenados. Ela recebe a *raiz* da árvore, e se a raiz não for nula a função se chama recursivamente passando o valor dos galhos da raiz, sendo o primeiro o à esquerda (que contém os valores menores) e depois o à direita (que contém os valores maiores). Ou seja, *mostraEmOrdem* se chama recursivamente até chegar na folha mais à esquerda da árvore binária, que terá o galho esquerdo

nulo, fazendo com que comece a ser mostrado no console os valores das folhas, subindo para o galho e indo para a direita, até chegar ao fim na folha mais à direita da árvore.

```
def mostraEmOrdem(raiz):
   if raiz != None:
       mostraEmOrdem(raiz.esquerda)
       print(raiz.valor, end=" ")
       mostraEmOrdem(raiz.direita)

mostraEmOrdem(raiz)
```

Saída no console:

```
2
4
5
7
11
```

2 Complexidade (melhor, médio e pior caso)

2.1 Pior caso:

Segundo Baeldung(2024)[5], a complexidade de tempo de Binary Tree Sort(BTS) é formada por:

- *n,* que é o tamanho do array na entrada do algoritmo.
- complexidade de inserção de um novo nó: que no caso médio é *O(log n)* e *O(n)* no pior caso.
- complexidade de tempo de construção da BTS, considerando o tempo de inserção: no caso médio é *O(n log n)* e $O(n^2)$ no pior caso.
- complexidade de travessia pela BTS que é *O(n)*.

Sendo assim, a pior complexidade de tempo para classificar uma árvore binária desbalanceada usando o BTS é $O(n^2)$.

2.2 Médio caso:

De acordo com o site GeeksforGeeeks[6] a complexidade de tempo em um caso médio, onde os dados são aleatórios, é *O(n log n)*, pois adicionar um item a uma BTS leva *O(log n)*, assim adicionar *n* itens leva *O(n log n)*.

2.3 Melhor caso:

A complexidade de tempo no melhor caso, que é quando há balanceamento da árvore, é *O(n log n)*, podendo tornar até o pior caso — que é a inserção de itens ordenados, que normalmente acarretam em uma árvore degenerada para algum lado — em *O(n log n)* (Wikipedia, 2025)[7].

[5] Disponível em: < Classificando os elementos em uma árvore binária >.
[6] Disponível em: < Classificação em árvore>.
[7] Disponível em: < Classificação de árvores>.

3 Casos de uso e aplicações práticas

3.1 Casos de uso

- Indexação em Bancos de Dados: as Árvores de Ordenação Binária (BSTs) são utilizadas para manter os dados ordenados, facilitando operações de busca, inserção e exclusão eficientes em bancos de dados.
- Binary Sort Tree possui diversas aplicações em ciências da computação pois permite a busca e classificação eficiente dos elementos[8].

4 Comparações com algoritmos semelhantes

- Algoritmo de Busca em Profundidade (DFS): de acordo com PARMAR (2025) é um algoritmo organizado em estrutura de árvore, que ao percorrer os dados, começa com um nó raiz e visita todos os nós mais profundos de um ramo antes de retroceder e ir para o próximo ramo.
- Algoritmo de Busca em Largura (BFS): PARMAR (2025), afirma que esse algoritmo, também organizado em estrutura de árvore, percorre seus dados nível a nível, ou seja, percorre todos os elementos de um nível antes de ir para o próximo nível mais abaixo.

Ambos algoritmos são muito semelhantes ao BST, pois os dois têm estrutura de árvore binária, tendo a diferença na forma que percorrem os dados.

[8] Disponível em: < O que é: Binary Search Tree >.

Referências

Creative Commons Attribution Share Alike 4.0 International. Enumeração de submáscara. 06 de junho de 2022. Disponível em: < Enumeração de submáscara >. Acesso em: 22 de maio de 2025.

MOHAN, Divya. KAPOOR, Puneet. Projeto de Análise de Algoritmo. 2025. Disponível em: < Projeto de Análise de Algoritmo >. Acesso em: 22 de maio de 2025.

MELO, Lawrence. Bitmask. 2019. Disponível em: < Bitmask >. Acesso em: 22 de maio de 2025.

RAMARAJ, E. VENKATESAN, N. Bit Stream Mask-Search Algorithm in Frequent Itemset Mining. European Journal of Scientific Research. Vol.27 No.2 (2009), pp.286-297. Disponível em: < Bit Stream Mask-Search Algorithm in Frequent Itemset Mining >. Acesso em: 23 de maio de 2025.

BASHIR, Shariq. JAN, Zahoor. BAIG, Abdul. Fast Algorithms for Mining Interesting Frequent Itemsets without Minimum Support. [v1] Ter, 21 de abril de 2009. Disponível em: < Fast Algorithms for Mining Interesting Frequent Itemsets without Minimum Support >. Acesso em: 23 de maio de 2025.

GEETHA, P. RAMARAJ, E. Bit Mask Search Algorithm for Trajectory Database Mining. International Journal of Computer Applications (0975 – 8887), 2013. Disponível em: < Bit Mask Search Algorithm for Trajectory Database Mining >. Acesso em: 23 de maio de 2025.

ORTIZ, Basti. Afinal, máscaras de bits não são tão esotéricas e impraticáveis… 2024. Disponível em: <Afinal, máscaras de bits não são tão esotéricas e impraticáveis...>. Acesso em: 23 de maio de 2025.

KODICHATH, Sreedev. Usos reais de operadores bit a bit. 2020. Disponível em: <Usos reais de operadores bit a bit>Acesso em: 23 de maio de 2025.

BEALDUNG. Classificando os elementos em uma árvore binária. 2024. Disponível em: < Classificando os elementos em uma árvore binária >Acesso em: 24 de maio de 2025.

GEEKSFORGEEKS. Classificação em árvore. 2023. Disponível em: < Classificação em árvore > Acesso em: 24 de maio de 2025.

WIKIPEDIA. Classificação de árvores. 2025. Disponível em: < Classificação de árvores>. Acesso em: 24 de maio de 2025.

BEALDUNG. Aplicações de Árvores Binárias. 2022. Disponível em: < Aplicações de Árvores Binárias >. Acessado em 5 de junho de 2025.

NAPOLEON. O que é: Binary Search Tree. 2025. Disponível em: < Binary Search Tree > Acessado em 5 de junho de 2025.

PARMAR, Anand K. 4 Tipos de Algoritmos de Percurso de Árvore. Disponível em: < 4 Tipos de Algoritmos de Percurso de Árvore >. Acessado em 5 de junho de 2025.