目次

探索の章で扱うアルゴリズムは以下と通りです。

- 線形探索
- 二分探索
- 二分探索木
- Treeq
- 平衡木 (AVL 木、B 木、赤黒木)
- ハッシュ法

検索 (サーチ) とは、データ集合 (配列など) から、目的とする値を持った要素を探し出すことを意味する。

I.　線形探索

線形探索は、目的とする要素が見つかるまで先頭から順に要素を見ていく探索方法です。例えば、以下の配列で 0 を探すなら前から順番に、8, 4, 5, 0, 2 の順に要素を見ていきます。

8	4	5	0	2

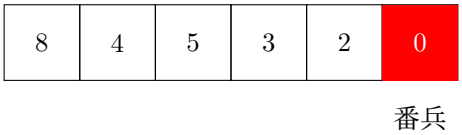
i.　線形探索の実装

コード 1　線形探索の実装

```
def linear_search(array: list[int], value: int) -> int:
    """
    線形探索をして一致するならその index を返す．一致しないときは -1 を返す
    """
    for i in range(len(array)):
        if array[i] == value:
            return i

    return -1
```

ii.　番兵

番兵法では探索するデータ集合の最後に目的とする数を追加します。最後に番兵を追加することで、より効率的に探索を行うことができます。実際には番兵を追加してもそこまで効率が良くなるわけではないですが、番兵を追加することで、ループの条件判定を省略することができます。



コード 2　番兵の実装

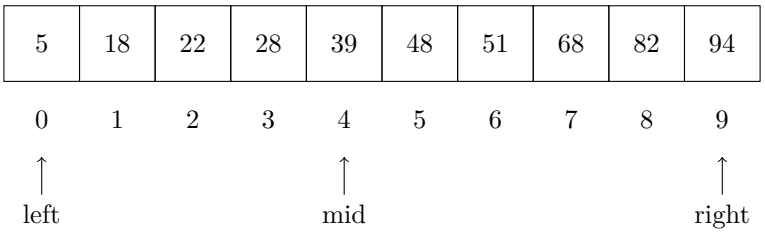
```python
def linear_search(array: list[int], value: int) -> int:
    """
    線形探索をして一致するならそのindexを返す．一致しないときは-1を返す
    """
    i = 0
    copied_array = array.copy()
    copied_array.append(value)
    while copied_array[i] != value: i += 1

    return i if i < len(array) else -1
```

II.　二分探索

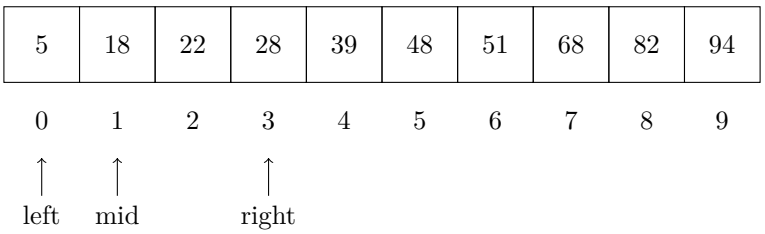
i.　配列を探索する二分探索

二分探索はソートされた配列に対して高速に探索を行うアルゴリズムの一つです。二分探索では探索する区間が条件に応じてどんどん半分になっていくため、計算量は $O(\log n)$ となります。以下の配列に対して、二分探索で 18 を探す場合を考えましょう。

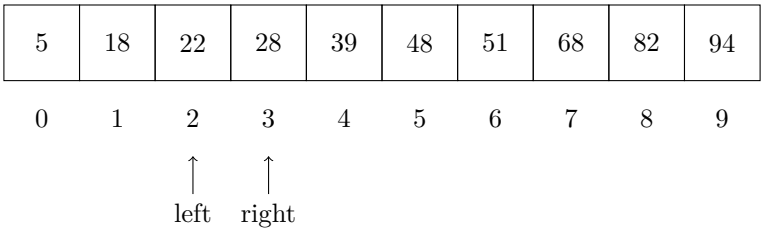
最初の区間は配列全体を取ります。0-indexed な配列を考えると、mid は 39 となり目的の 18 より大きいです。mid が目的の値より大きい場合、右の区間を狭めます。right = mid - 1 とします。



right を 3 に更新しました。mid = (0 + 3) // 2 = 1 となります。mid の値は 18 で目的の値と一致します。



次に、目的とする値を 19 として配列に存在しない場合を考えましょう。mid = 1 の値は 18 で目的の値よりも小さいので、下の図のように left を mid + 1 に更新します。mid = (2 + 3) // 2 = 2 となります。mid の値は 22 で目的の値より大きいです。よって、right を mid - 1 に更新します。すると、right = 1, left = 2 となり、left > right となるので探索を終了します。



二分探索の実装は以下の通りです。

コード 3　二分探索の実装

```
def binary_search(A: list[int], value: int) -> int:
    left, right = 0, len(A) - 1

    while left <= right:
        mid = (left + right) // 2

        if A[mid] == value:
            return mid

        if A[mid] < value:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

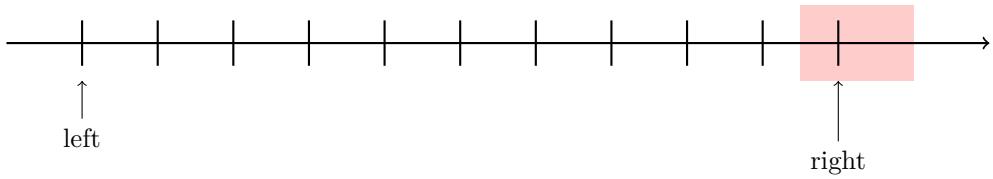
ii.　一般化した二分探索

Python の標準ライブラリである bisect の bisect_left の実装をしましょう。bisect_left はソートされた配列に対して、与えられた値以上の最小の index を返す関数です。bisect_left は二分探索を用いて実装されています。先ほどの配列から要素を探す二分探索では、要素一致するか否かを判定していましたが、今回はもっと一般化して「mid がある条件を満たすか否か」を判定して範囲を狭めていきます。このとき、left より右側の区間は条件を満たさず、right より左側の区間は条件を満たすようにします。

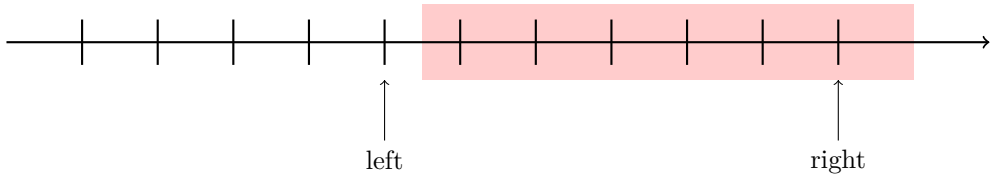
二分探索では以下の図のような条件を満たす赤い部分を条件を満たす境界ギリギリになるように更新していきます。

- left は常に条件を満たさない
- right は常に条件を満たす

という条件を満たすように処理を進めていき、最終的に left が条件を満たさない最大の index、right が条件を満たす最小の index を返します。bisect_left では left が与えられた値よりも小さく、right が与えられた値以上の最小の index を返す関数です。



初期状態



終了状態

bisect_left の実装は以下の通りです。

コード 4　bisect_left の実装

```
def bisect_left(array: list[int], key: int) -> int:
  left, right = -1, len(array)

  while right - left > 1:
      mid = (left + right) // 2

      if array[mid] < key:
```

```
            left = mid
        else:
            right = mid

    return right
```

bisect_left の実装では、左側が条件を満たさない、右側が条件を満たすといった実装になっています。これだとまだ条件次第で left が条件を満たす、right が条件を満たさないという実装もあり得ます。そこで以下ではめぐる式二分探索のさらに一般化した実装を紹介します。

iii.　めぐる式二分探索

めぐる式二分探索では、

- ng は常に条件を満たさない
- ok は常に条件を満たす

のように数直線上の右や左という概念を持たせずに実装します。めぐる式二分探索は以下のように実装されます。is_ok は条件を満たすかどうかを判定する関数ですので、条件に合わせて実装します。

コード 5　めぐる式二分探索

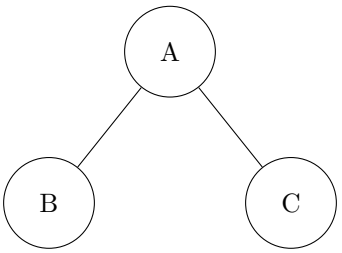
```
def binary_search(array: list[int], key: int):
    ng, ok = -1, len(array)

    while abs(ok - ng) > 1:
        mid = (ng + ok) // 2

        if is_ok(array, mid, key):
            ok = mid
        else:
            ng = mid

    return ok
```

III.　二分探索木

二分探索は探索自体は $O(\log n)$ で行うことができますが、配列がソートされているひつようがあり、データの挿入や削除がある場合は毎回ソートがあり非効率ではあります。解決策としては、**データ構造で解決**や **Treeq** などがあります。データ構造で解決する方法の一つが**二分探索木**です。

二分探索木は以下の性質を持った木構造です。

- 左子ノードは親ノードよりも小さい (または等しい) (下の図では $B \leq A$)
- 右子ノードは親ノードよりも大きい (下の図では $A < C$)



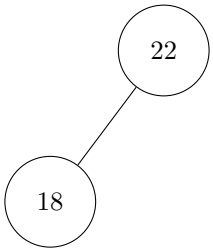
二分探索木で行う処理は以下の通りです。

- 木の作成
- 探索
- 削除

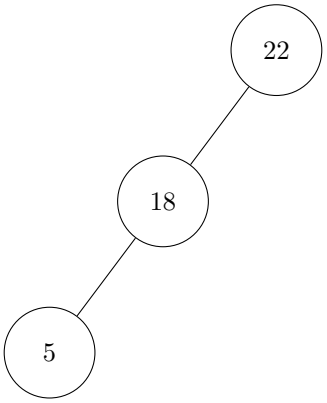
i.　二分探索木の作成

`[22,18, 5, 82, 51, 39]` の配列を二分探索木に変換してみましょう。

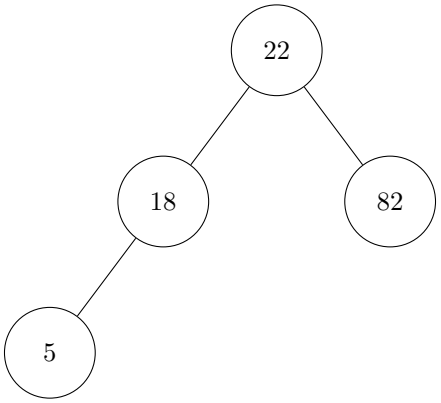
22 を挿入



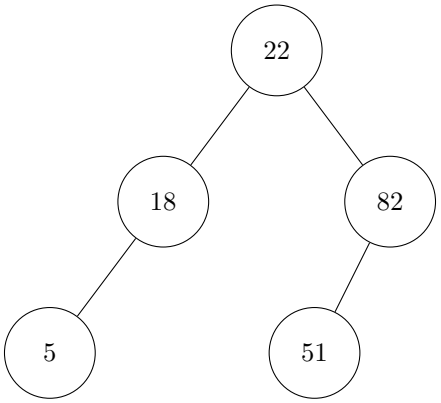
18 を挿入



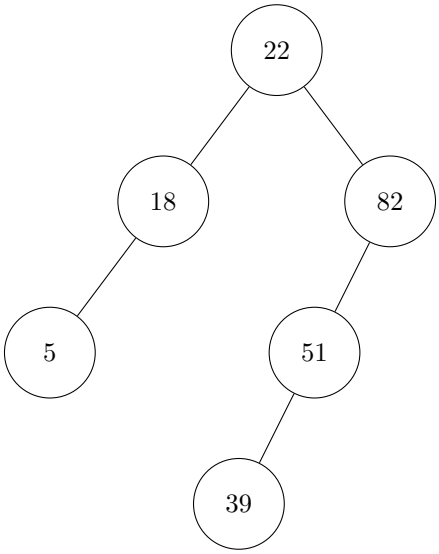
5 を挿入



82 を挿入



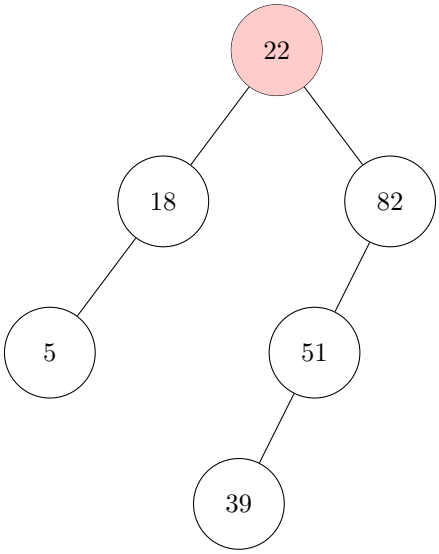
51 を挿入



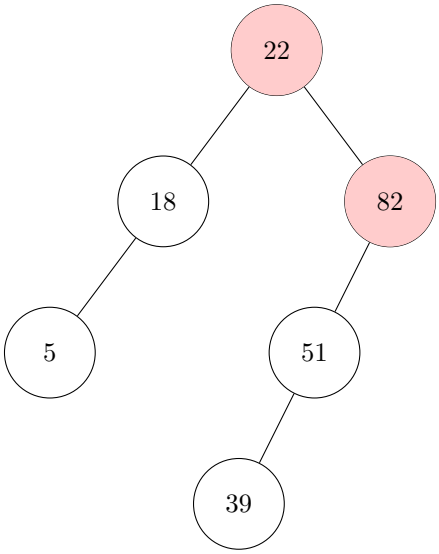
39 を挿入

ii.　二分探索木の探索

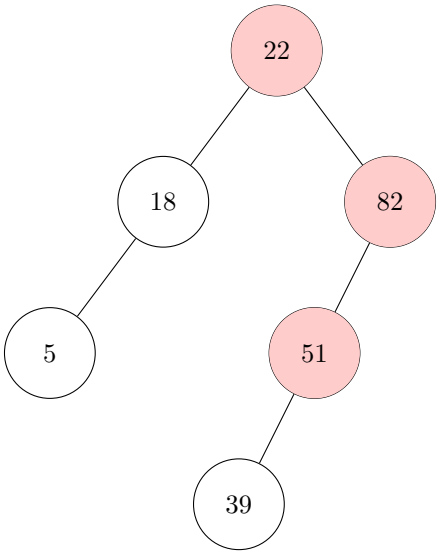
上の木を例に 51 を探してみましょう。根から順に探していきます。



51 は 22 よりも大きいので右の子ノードに進みます。



51 は 82 よりも小さいので左の子ノードに進みます。見つかりました。



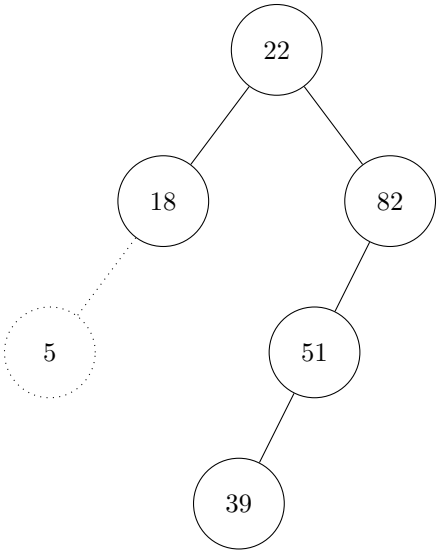
iii.　二分探索木の削除

二分探索木からノードを削除する場合は以下の 3 つのケースがあります。

- 子ノードがない場合
- 子ノードが 1 つの場合
- 子ノードが 2 つの場合

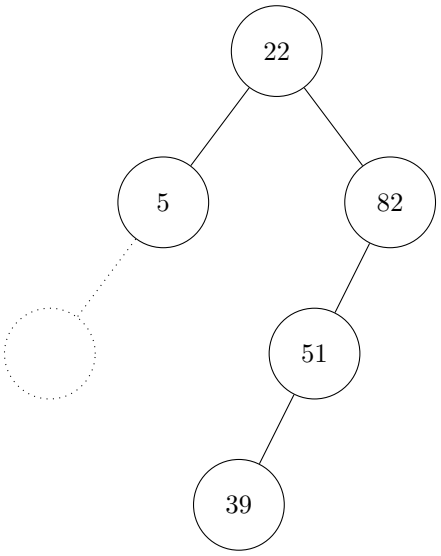
1. 子ノードがない場合

例えば 5 を削除する場合を考えます。5 は左右の子ノードがないので、他のノードに影響がないためそのまま削除します。



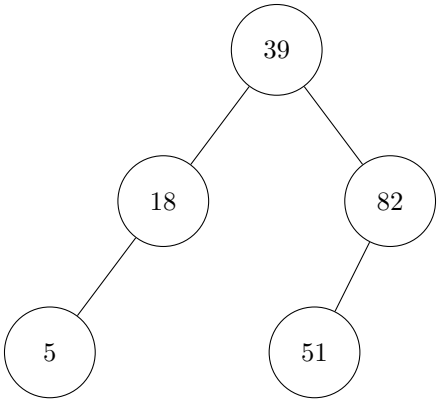
2. 子ノードが 1 つの場合

次に 18 を削除する場合を考えます。18 は左の子ノードが 5 しか持っていないので、5 を 18 の位置に移動させます。18 は親ノードから見て左の子ノードですが、右ノードの場合も同様に処理します。



3. 子ノードが 2 つの場合

最後に 22 を削除する場合を考えます。削除するノードが 2 つの子ノードを持っている場合は、削除するノードの左の子ノードの最大値か、右の子ノードの最小値を持ってきて、削除するノードに移動させます。ここでは、22 の右の子ノードの最小値 39 を持ってきて、22 の位置に移動させます。



iv. 二分木の実装

コード 6　二分木の実装

```
class Node:
  def __init__(self, data: int) -> None:
      self.data: int = data
      self.right: Node | None = None
      self.left: Node | None = None

  def __str__(self) -> str:
      return str(self.data)

class BinarySearchTree:
  def __init__(self):
      self.root = None

  def create(self, array: list[int]) -> None:
      for i in range(len(array)):
          self.insert(array[i])

  def insert(self, data: int) -> None:
      if self.root is None:
```

```
            self.root = Node(data)
        else:
            self._insert_recursively(self.root, data)

    def _insert_recursively(self, node: Node, data: int) -> None:
        if data <= node.data:
            if node.left is None:
                node.left = Node(data)
            else:
                self._insert_recursively(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
            else:
                self._insert_recursively(node.right, data)

    def search(self, value: int) -> Node | None:
        if self.root is None:
            return self.root

        return self._search_recursively(self.root, value)

    def _search_recursively(self, node: Node, value: int) -> Node | None:
        if node.data == value:
            return node

        if value < node.data:
            if node.left is None:
                return node.left
            else:
                return self._search_recursively(node.left, value)
        else:
            if node.right is None:
                return node.right
            else:
                return self._search_recursively(node.right, value)
```

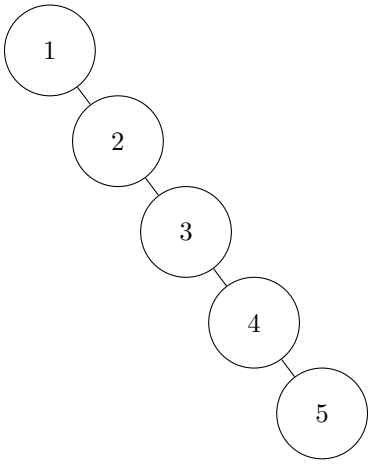
```
56
57    def delete(self, value: int) -> Node | None:
58        if self.root is None:
59            return self.root
60        else:
61            self.root = self._delete_recursively(self.root, value)
62
63    def _delete_recursively(self, node: Node, value: int) -> Node | None:
64        if value < node.data:
65            if node.left is None:
66                return None
67            else:
68                node.left = self._delete_recursively(node.left, value)
69        elif value > node.data:
70            if node.right is None:
71                return None
72            else:
73                node.right = self._delete_recursively(node.right, value)
74        else:
75            if node.left is None and node.right is None:
76                return None
77            elif node.left is None:
78                return node.right
79            elif node.right is None:
80                return node.left
81            else:
82                successor = self._successor()
83                node.data = successor.data
84                self._delete_recursively(node.right, successor.data)
85
86        return node
87
88    def _successor(self) -> Node | None:
89        if self.root is None:
90            return self.root
91
```

```
92          node = self.root
93
94          while node.left is not None:
95              node = node.left
96
97          return node
```

IV. 平衡木

i. 二分木の問題点

二分木は最悪の場合木が以下のように片方に伸びてしまった場合、計算量が $O(n)$ になってしまいます。



ii. 平衡木

平衡木は**回転**や乱択アルゴリズムを用いて木の高さを自動的に小さくする方向に調整できる木構造です。

扱う平衡木の代表的なものには以下のものがあります。

- AVL 木
- 赤黒木
- B 木
- Treap

iii. AVL 木

AVL 木とは、**Adelson-Velsky and Landis** の名前から取られた木構造で、以下の性質を持ちます。

- 任意のノードにおいて、左右の部分木の高さの差が 1 以下である

以降の考察のために **balance factor** という言葉を導入します。

balance factor

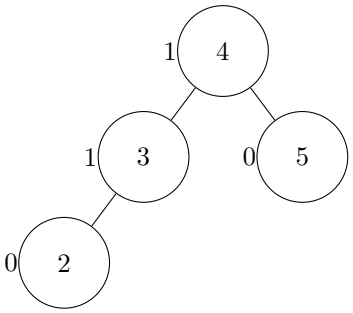
あるノードの左部分木の高さから右部分木の高さを引いた値を **balance factor** といいます。

- balance factor = 左部分木の高さ - 右部分木の高さ

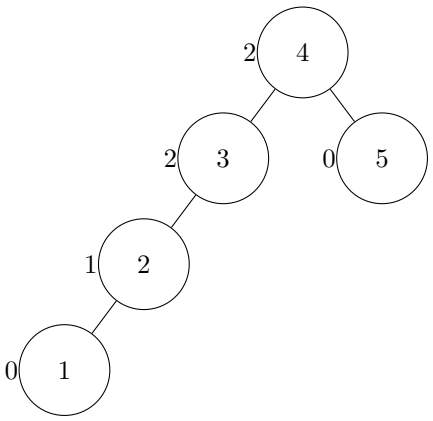
AVL 木の性質を balance factor を使って表すと以下のようになります。

- 任意のノードにおいて、balance factor は-1, 0, 1 のいずれかである

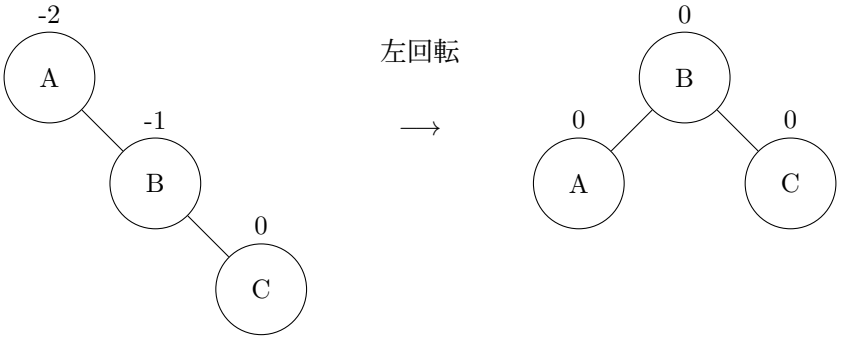
AVL 木の例を以下に示します。ノードの横に書いてあるのが balance factor です。



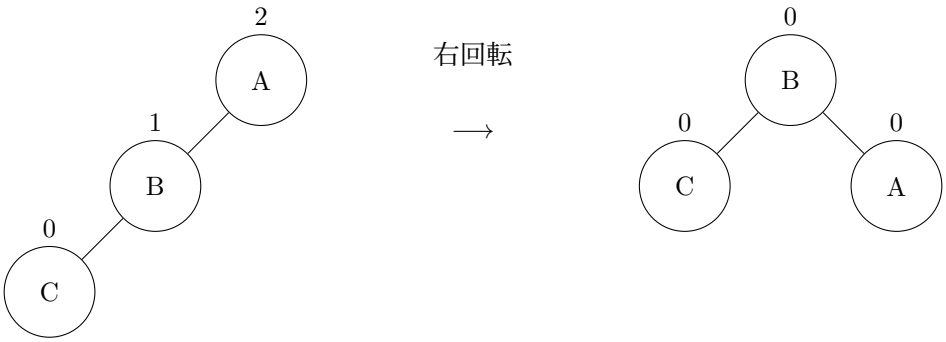
次は AVL 木の条件を満たしていない木です。



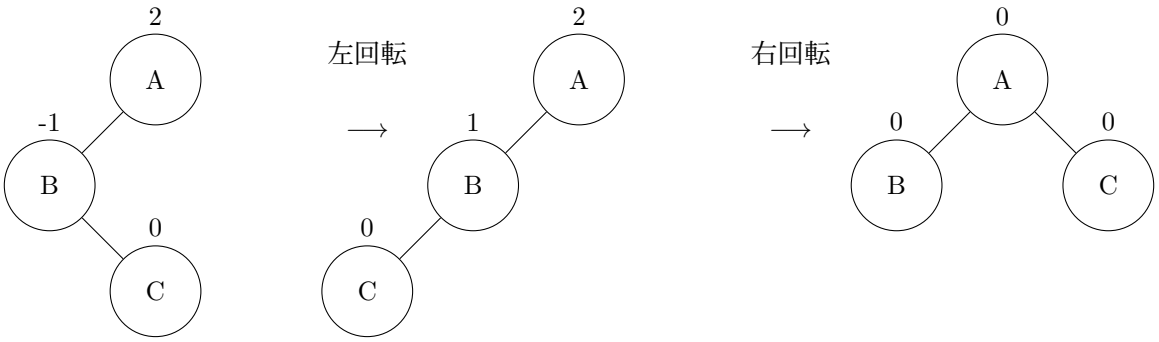
1. 左回転



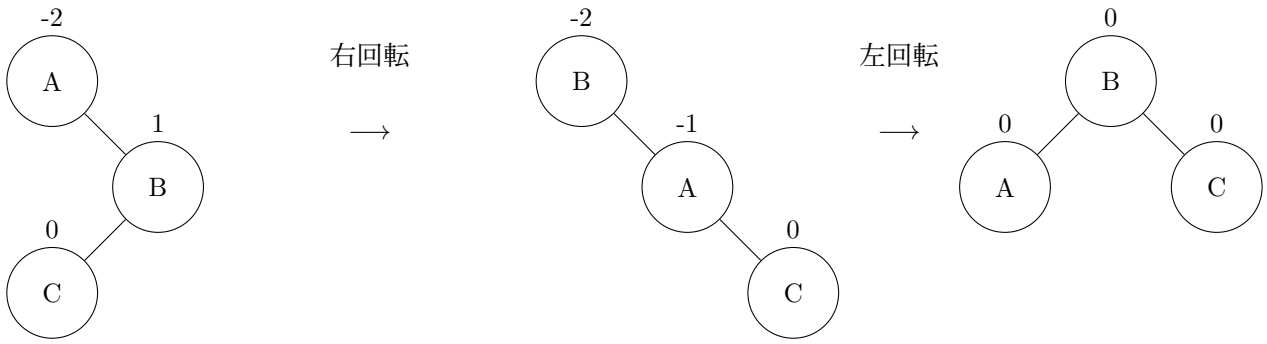
2. 右回転



3. 左右回転



4. 右左回転



コード 7　二分ヒープの実装

```
class Node:
    def \_\_init\_\_(self, value: int) -> None:
        self.data: int = value
        self.left: Node | None = None
        self.right: Node | None = None
        self.height: int = 0

    def \_\_str\_\_(self) -> str:
        return f"Node: data = {self.data} height = {self.height}"

class AVLTree:
    def \_\_init\_\_(self):
        self.root = None

    def \_height(self, node: Node | None) -> int:
        if node is None:
            return -1

        return node.height

    def \_update\_height(self, node:
     Node) -> None:
        node.height = 1 + max(self.\_height(node.left), self.\_height(
            node.right))

    def \_balance\_factor(self, node: Node) -> int:
        return self.\_height(node.left) - self.\_height(node.right)
```

```
27
28      def \_right\_rotate(self, node: Node) -> Node:
29          new\_root = node.left
30          node.left = new\_root.right
31          new\_root.right = node
32
33          self.\_update\_height(node)
34          self.\_update\_height(new\_root)
35
36          return new\_root
37
38      def \_left\_rotate(self, node: Node) -> Node:
39          new\_root = node.right
40          node.right = new\_root.left
41          new\_root.left = node
42
43          self.\_update\_height(node)
44          self.\_update\_height(new\_root)
45
46          return new\_root
47
48      def create(self, values: list[int]) -> None:
49          for value in values:
50              self.insert(value)
51
52      def insert(self, value: int) -> None:
53          if self.root is None:
54              self.root = Node(value)
55              return self.root
56          else:
57              self.root = self.\_insert\_recursively(self.root, value)
58              return self.root
59
60      def \_insert\_recursively(self, node: Node, value: int) -> Node:
61          if value <= node.data:
62              if node.left is None:
```

```
                node.left = Node(value)
            else:
                node.left = self.\_insert\_recursively(node.left, value
                    )
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                node.right = self.\_insert\_recursively(node.right,
                    value)

        self.\_update\_height(node)

        balance\_factor = self.\_balance\_factor(node)

        if balance\_factor > 1 and value < node.left.data:
            return self.\_right\_rotate(node)

        if balance\_factor > 1 and value > node.left.data:
            node.left = self.\_left\_rotate(node.left)
            return self.\_right\_rotate(node)

        if balance\_factor < -1 and value > node.right.data:
            return self.\_left\_rotate(node)

        if balance\_factor < -1 and value < node.right.data:
            node.right = self.\_right\_rotate(node.right)
            return self.\_left\_rotate(node)

        return node


    def search(self, value: int) -> Node | None:
        if self.root is None:
            return None

```

```
97            return self.\_search\_recursively(self.root, value)
98
99        def \_search\_recursively(self, node: Node, value: int) -> Node |
              None:
100           if value == node.data:
101               return node
102
103           if value < node.data:
104               if node.left is None:
105                   return None
106               else:
107                   return self.\_search\_recursively(node.left, value)
108           else:
109               if node.right is None:
110                   return None
111               else:
112                   return self.\_search\_recursively(node.right, value)
113
114
115       def delete(self, value: int) -> Node | None:
116           if self.root is None:
117               return self.root
118           else:
119               self.root = self.\_delete\_recursively(self.root, value)
120               return self.root
121
122       def \_delete\_recursively(self, node: Node, value: int) -> Node |
              None:
123           if value < node.data:
124               node.left = self.\_delete\_recursively(node.left, value)
125           elif value > node.data:
126               node.right = self.\_delete\_recursively(node.right, value)
127           else:
128               if node.left is None and node.right is None:
129                   return None
130               elif node.left is None:
```

```
131                 return node.right
132             elif node.right is None:
133                 return node.left
134             else:
135                 min\_node = self.\_find\_min(node.right)
136                 node.data = min\_node.data
137                 node.right = self.\_delete\_recursively(node.right, min
                        \_node.data)
138 
139         self\.\_update\_height(node)
140 
141         balance\_factor = self.\_balance\_factor(node)
142 
143         if balance\_factor > 1 and self.\_balance\_factor(node.left) >=
                0:
144             return self.\_right\_rotate(node)
145 
146         if balance\_factor < -1 and self.\_balance\_factor(node.right)
                <= 0:
147             return self.\_left\_rotate(node)
148 
149         if balance\_factor > 1 and self.\_balance\_factor(node.left) <
                0:
150             node.left = self.\_left\_rotate(node.left)
151             return self.\_right\_rotate(node)
152 
153         if balance\_factor < -1 and self.\_balance\_factor(node.right)
                > 0:
154             node.right = self.\_right\_rotate(node.right)
155             return self.\_left\_rotate(node)
156 
157         return node
```

iv.　B 木

AVL 木は二分木であるため、データの数が多い場合に気が高くなり検索や挿入に時間がかかってしまう問題があります。そこで枝の数が 2 本よりも多く取る B 木というデータ構造を紹介します。B 木は応用範囲が広く以下の様な様々なものに使われています。

- データベース
- ファイルシステム

B 木はノードが持てる最大の枝の本数 *order* によって定義されます。二分木ではノードは 1 つの値を持っていてその値との大小比較で左右の子ノードに振り分けていましたが、B 木ではノードが複数の値を持ち、その値の範囲で子ノードに振り分けます。order=3 の B 木の例を以下に示します。`[10, 5, 3, 2, 8, 4, 1, 6]` のデータを order=3 の B 木に入れていきます。

1.　B 木の実装

コード 8　B 木の実装

```
from bisect import bisect_left, bisect_right

class Node:
    def __init__(self):
        self.keys: list[int] = []
        self.children: list[Node] = []

class BTree:
    def __init__(self, order: int) -> None:
        self.order = order
        self.root = Node()

    def _median(self, array: list[int]) -> int:
        return array[len(array) // 2]

    def insert(self, value: int) -> None:
        node = self._insert_recursively(self.root, value)
        if node:
            new_root = Node()
```

```
20              new_root.keys = [node[0]]
21              new_root.children = [node[1], node[2]]
22              self.root = new_root
23
24      def _insert_recursively(self, node: Node, value: int):
25          # 葉ノード
26          if not node.children:
27              node.keys.append(value)
28              node.keys.sort()
29              if len(node.keys) < self.order:
30                  return None
31              else:
32                  return self._split(node)
33          else:
34              index = bisect_right(node.keys, value)
35              result = self._insert_recursively(node.children[index], value
                    )
36
37              if result is not None:
38                  median, left_node, right_node = result
39                  node.keys.insert(index, median)
40
41                  # 修正：左ノードを適切に設定し、右ノードを挿入
42                  node.children[index] = left_node  # 左の子ノードを置き換
                        える
43                  node.children.insert(index + 1, right_node)  # 右の子ノー
                        ドを挿入
44
45                  if len(node.keys) < self.order:
46                      return None
47                  else:
48                      return self._split(node)
49              return None
50
51      def _split(self, node: Node):
52          median_index = len(node.keys) // 2
```

```
53          median = node.keys[median_index]
54
55          left_node = Node()
56          right_node = Node()
57
58          left_node.keys = node.keys[:median_index]
59          right_node.keys = node.keys[median_index + 1:]
60
61          if node.children:
62              left_node.children = node.children[:median_index + 1]
63              right_node.children = node.children[median_index + 1:]
64
65          return median, left_node, right_node
66
67      def search(self, value: int) -> bool:
68          node = self.root
69          if value in node.keys:
70              return True
71          else:
72              return self._search_recursively(self.root, value)
73
74      def _search_recursively(self, node: Node, value: int) -> bool:
75          if len(node.children) == 0:
76              return False
77
78          index = bisect_left(node.keys, value)
79          next_node = node.children[index]
80
81          if value in next_node.keys:
82              return True
83          else:
84              return self._search_recursively(node.children[index], value)
85
86  b_tree = BTree(order=3)
87  b_tree.insert(5)
88  b_tree.insert(10)
```

```
89 b_tree.insert(15)
90 b_tree.insert(20)
91 b_tree.insert(30)
92
93 b_tree.insert(21)
94 b_tree.insert(22)
```

v.　赤黒木

赤黒木は AVL 木同様平衡二分探索木の１つで、以下の特徴を持っています。

- 各ノードは赤か黒の色を持つ
- 根ノードは黒である
- 赤のノードの子ノードは黒である
- あるノードからその子孫の葉ノードまでの黒の数は同じである
- 葉ノードは黒である

V.　参考

二分探索

- https://qiita.com/drken/items/97e37dd6143e33a64c8c

二分探索木

- https://www.geeksforgeeks.org/introduction-to-avl-tree/

B 木

- https://www.geeksforgeeks.org/b-tree-in-python/
- https://www.javatpoint.com/b-tree
- https://wqwq3215.medium.com/b-tree%E3%82%92%E7%90%86%E8%A7%A3%E3%81%97%E3%81%A6%E3%81%84%E3%81%8F-142f93fc3c6c

赤黒木

- https://qiita.com/kgoto/items/b15b9a494deae010d660
- http://wwwa.pikara.ne.jp/okojisan/rb-tree/index.html

EEIC2024 アルゴリズムの授業で扱った文字列照合のアルゴリズムのシケプリです。扱った内容は以下の通りです。

- 力任せ法
- KMP 法
- BM 法
- BMH 法
- ラビン・カープ法 (ローリングハッシュ)

VI.　力任せ法

力任せ法は文字通り文字列において、所望のパターンが見つかるまで前から順に比較していく方法です。以下で text と pattern の文字列が与えられたときに、text の中に pattern が含まれるかどうか考えましょう。文字列の文字を表す記法は text[i] のように 0-index で表します。

最初は text[0] と pattern[0] を比較すると一致しません。一致しないときは、text の cursor を 1 つ進めて text[1] と pattern[0] を比較します。

B	A	B	A	B	C	B	A	B	A	B	D	B

text

A	B	A	B	D								

pattern

text[1] と pattern[0] を比較すると、一致します。しかし、text[5] と pattern[4] が一致しないので、text の cursor を 1 つ進めて text[2] と pattern[0] を比較します。

B	A	B	A	B	C	B	A	B	A	B	D	B

text

	A	B	A	B	D							

pattern

これを一致していくまで続けていくと、以下のようになります。

B	A	B	A	B	C	B	A	B	A	B	D	B

text

							A	B	A	B	D	

pattern

例を通じて力任せ法では、text を 0 から text の長さ - pattern の長さまで動かすことで、pattern が text に含まれるかどうかを判定することができます。C 言語でプログラムを書いてみましょう。

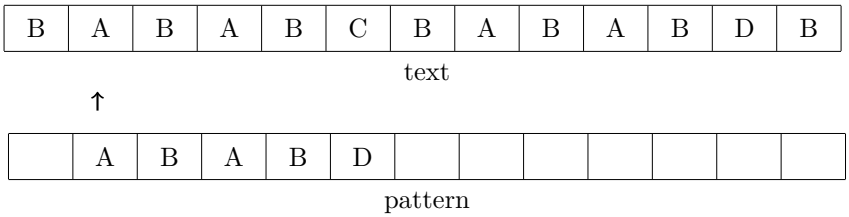
コード 9　力任せの実装

```
1 def brute_force(text: str, pattern: str) -> int:
2     for text_cursor in range(len(text) - len(pattern) + 1):
3         pattern_cursor = 0
4         moving_text_cursor = text_cursor
5         while pattern_cursor < len(pattern) and text[moving_text_cursor]
              == pattern[pattern_cursor]:
6             pattern_cursor += 1
```

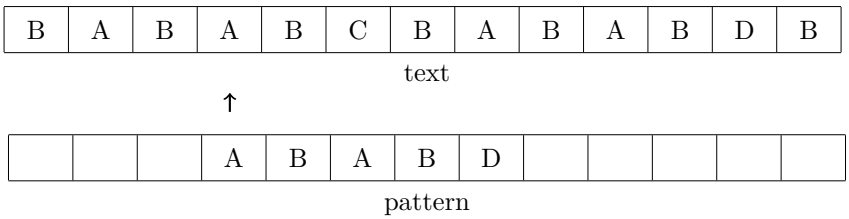
```
            moving_text_cursor += 1

        if pattern_cursor == len(pattern):
            return text_cursor

    return -1
```

VII.　KMP 法

i.　力任せ法の無駄

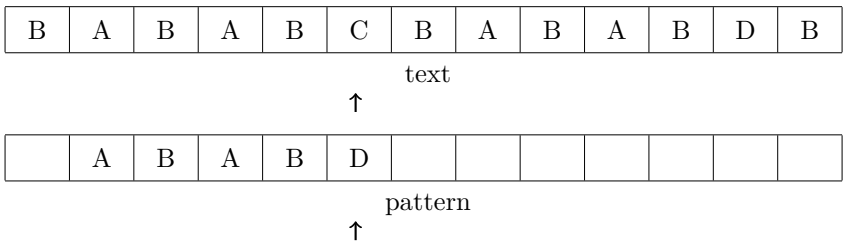


上の例では text[1] から照合を始めて text[5] で照合が失敗していることがわかります。力任せ法では次に text[2] から照合を始めることになりますが、それは text の方の cursor が一度通った場所を再度通ることになります。このように、力任せ法は text の cursor が一度通った場所を再び通ることが多いため、無駄が多いといえます。ただ、上の例では ABAB というパターンがあって、ABAB の部分が繰り返されていることがわかります。このような繰り返し部分がある場合、次に照合を始める位置をスキップすることで無駄が省けそうではないでしょうか？下の例では、text の cursor が後戻りすることなく、pattern の位置も前にスキップされています。

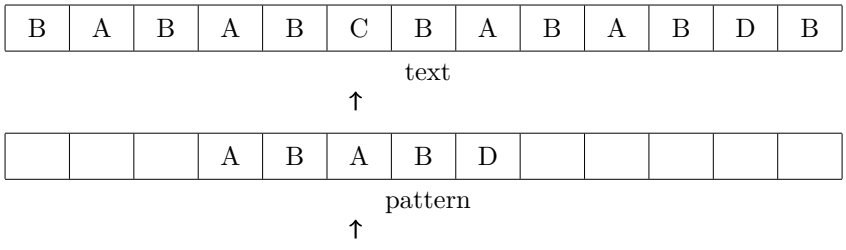


ii.　KMP 法の仕組み

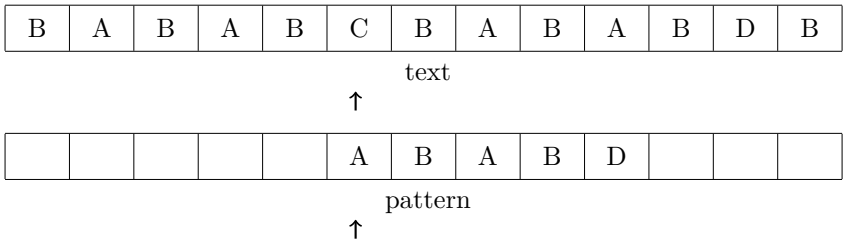
上のスキップを実現する方法の 1 つに KMP 法があります。KMP 法の流れを具体例を見て確認します。


B A B A B C B A B A B D B
text
A B A B D
pattern

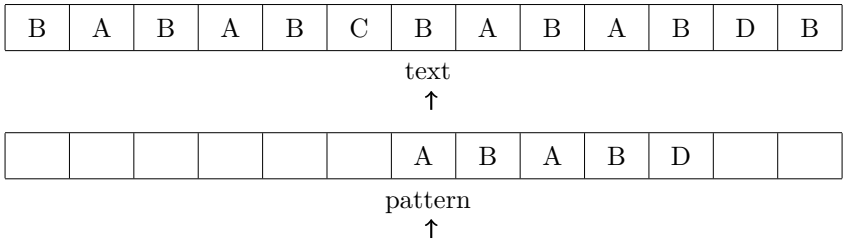
(1)


B A B A B C B A B A B D B
text
A B A B D
pattern

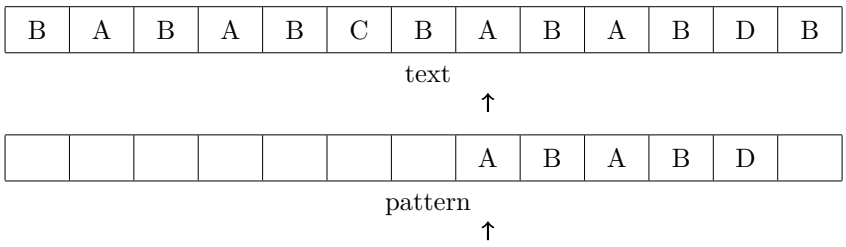
(2)


B A B A B C B A B A B D B
text
A B A B D
pattern

(3)


B A B A B C B A B A B D B
text
A B A B D
pattern

(4)


B A B A B C B A B A B D B
text
A B A B D
pattern

(5) 一致

(1) では pattern[4] で照合失敗しており、pattern[0] から pattern[3](ABAB) の前半と text[1] から text[4] の (ABAB) の後半 2 文字が一致しています。つまり、最初の 2 文字は比べなくても一致していることがわかります。そのため (2) では pattern の cursor を patten[2] にして照合を開始しています。

(2) では文字が一致しておらず、pattern をずらしても text[4], ,text[5] には一致しないので、(3) では pattern の cursor をはじめに戻しています。

(3) と (4) では文字が一致しておらず、さらに pattern もこれ以上右にシフトすることができないので、text と pattern ともに最初の位置から cursor を開始しています。

(5) では pattern が text に含まれていることがわかりました。

上の例で見た text と pattern で一致しているときに飛ばせる cursor の数は pattern によって事前に決まります。そのため、文字列照合の前に**スキップテーブル**作成を前処理として行います。

iii.　スキップテーブルの作成

スキップテーブルは以下の条件を満たします。

- スキップテーブルの配列の長さは pattern の長さ - 1
- スキップテーブルの i 番目の要素は、pattern の i 番目までの部分文字列の最大の接頭辞と接尾辞の長さ、つまり pattern[0] から pattern[i] までの文字列を見たとき、前から何文字目までが後ろから何文字目までと一致しているかを表す。
- スキップテーブルの最初の要素は 0

スキップテーブルを作成するには、pattern 同士をずらして比べていきます。その際に以下の手順で作成します。

- 文字がマッチする場合は、直近の最長部分一致の長さを記録し、そのまま照合を次の文字に進める
- 文字がマッチしない場合は、マッチが失敗した位置からパターンの中で可能な部分一致の場所に移動する (KMP 法そのもの)

B	A	B	A	B	C	B	A	B	A	B	D	B	
	B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

pattern[1] と pattern[0] から照合を開始します。pattern[1] で照合が失敗しました。pattern[0] で失敗すると pattern を動かせないので skip[1] に 0 を入れて cursor を進めます。

0	0										
0	1	2	3	4	5	6	7	8	9	10	11

skip

次は pattern[2] と pattern[0] を照合します。今回は文字が一致しているため、0 + 1 をして skip[2] に 1 を入れて cursor を進めます。また、pattern[3] と pattern[1]、pattern[4] と pattern[2] も一致しているため、skip[3] と skip[4] にはそれぞれ 2 と 3 を入れます。

B	A	B	A	B	C	B	A	B	A	B	D	B		
		B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

0	0	1	2	3							
0	1	2	3	4	5	6	7	8	9	10	11

skip

pattern[5] と pattern[3] は一致していません。マッチが失敗した位置からパターンの中で可能な部分一致の場所に移動するつまり、skip[3-1] = skip[2] の 1 にします。

B	A	B	A	B	C	B	A	B	A	B	D	B				
				B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

比較せずとも pattern[4] と pattern[0] は一致していることに注意をしてください。ただ。pattern[5] と pattern[1] で比較して失敗しているので、skip[1-1] を見て pattern を 0 から開始に移動します。

B	A	B	A	B	C	B	A	B	A	B	D	B					
					B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

下の pattern を移動できるだけ移動しても一致していないので、skip[5] には 0 を入れて再び cursor を前に移動させます。下の図のように次に移動すると上の pattern[10] までは一致していることがわかるので、skip を同様に埋めてしまいます。

B	A	B	A	B	C	B	A	B	A	B	D	B						

						B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

0	0	1	2	3	0	1	2	3	4	5	
0	1	2	3	4	5	6	7	8	9	10	11

skip

pattern[11] と pattern[5] を比較すると一致していないので、skip[5-1] = skip[4] を見ると 3 になっているので下の pattern の照合位置を 4 からにします。

B	A	B	A	B	C	B	A	B	A	B	D	B								

								B	A	B	A	B	C	B	A	B	A	B	D	B

pattern

pattern[11](D) と pattern[3](A) はまたもや一致していないので、skip[3-1] = skip[2] を見ると 1 になっているので、下の pattern の照合位置を 1 からにします。

B	A	B	A	B	C	B	A	B	A	B	D	B								

										B	A	B	A	B	C	B	A	B	A	B

pattern

またまた pattern[11] と pattern[1] は一致しておらず、もう pattern の移動する余裕はないので skip[10] には 0 を入れます。

0	0	1	2	3	0	1	2	3	4	5	0
0	1	2	3	4	5	6	7	8	9	10	11

skip

これでスキップテーブルは完成です。好きな文字列と上の例で扱った文字列を照合してみると理解が深まります。

iv.　KMP 法の実装

実装のポイントは以下の 2 つです。

- スキップテーブルの作成

- 前処理で用意したスキップテーブルを使った文字列照合

コード 10　KMP 法の実装

```
def create_table(pattern: str) -> list[int]:
    skip = [0] * (len(pattern) - 1)
    j = 0

    for i in range(1, len(pattern) - 1):
        if pattern[i] == pattern[j]:
            j += 1
            skip[i] = j
        else:
            while j > 0 and pattern[i] != pattern[j]:
                j = skip[j-1]

            # 例によっては忘れがちなので注意
            if pattern[i] == pattern[j]:
                j += 1

            skip[i] = j

    return skip

def kmp(text: str, patten: str) -> int:
    skip_table = create_table(patten)
    pattern_cursor = 0
    # len(text) - len(pattern) + 1ではないことに注意
    for text_cursor in range(len(text)):
        if text[text_cursor] == patten[pattern_cursor]:
            pattern_cursor += 1
        else:
            while pattern_cursor > 0 and text[text_cursor] != patten[
                pattern_cursor]:
                pattern_cursor = skip_table[pattern_cursor - 1]

            if text[text_cursor] == pattern[pattern_cursor]:
                pattern_cursor += 1
```

```
34
35
36          if pattern_cursor == len(patten):
37              return text_cursor - len(patten) + 1
38
39      return - 1
```

VIII. BM 法

i. BM 法の仕組み

BM 法は照合パターンの前からではなくて、後ろから照合を開始します。照合パターンの一番最後つまり D から照合を開始します。C と D は一致していないので、ずらします。

B	A	B	A	C	C	B	A	B	A	B	A	B	D	B
A	B	A	B	D										

BM 法では以下のように最初から多くスキップすることが多いです。

B	A	B	A	C	C	B	A	B	A	B	A	B	D	B
					A	B	A	B	D					

また A と D は一致していないのでスキップします。今回は A という文字で照合が失敗していますが、照合パターンにも A があるので先ほどのように一気に飛ばせません。

B	A	B	A	C	C	B	A	B	A	B	A	B	D	B
							A	B	A	B	D			

次も同様に考えると以下のようになり、一致します。

B	A	B	A	C	C	B	A	B	A	B	A	B	D	B
									A	B	A	B	D	

上の例では、照合パターンの最後の文字から照合を開始しました。その際に text で一致しない文字に応じて text の照合 cursor の位置を動かしていました。では、その動かす大きさはどのように決まるでしょうか？ KMP 法と同様にスキップテーブルを作成します。スキップテーブルを以下のポイントを意識して作成します。照合パターンの長さを l とします。

- パターンに含まれない文字とパターンの一番最後にしかない文字の移動量は l
- 一番最後以外に含まれる文字に関しては、末尾に最も近い出現位置が $i(0 \leq i < l)$ ならば、そのときの移動量は $l - i + 1$
- 移動量は text に出現する文字をすべて網羅する辞書で管理

text 照合中にどの文字で失敗したかに応じて移動量が変化するので、text に登場する文字が文字が予め分かっている必要があります。text に登場する文字に応じて US-ASCII や UTF-8 などの文字コードを使って辞書を作成します。今回は US-ASCII で実装します。

ii. パターンテーブルだけの実装の問題点

実はパターンテーブルだけでは問題が起こることがあります。プログラムの停止性に関わる問題です。例えば以下のような text と pattern を考えます。

				B	A	D	D	D	B					
				A	C	A	D	B						

マッチテーブルは以下のようです。

A	B	C	D	E
2	5	3	1	5

まず、D で失敗しているのでスキップテーブルの D を見ると 1 になっています。text の照合開始位置を一つ進めます。

				B	A	D	D	D	B					
					A	C	A	D	B					

今度も D で失敗しているので、text の照合開始位置を 1 進めましょう。今回は text の途中までは照合が成功しているので、+1 するのは照合が失敗した index であることに注意してください。

				B	A	D	D	D	B					
				A	C	A	D	B						

これは最初に見たパターンと同じになっているため、無限ループが発生しています。問題の原因は、スキップテーブルだけだと text の照合開始 index が後戻りすることがあるためです。スキップテーブルを用いることで、照合開始地点よりも前から照合してもパターンと一致することがないことが保証されているので、そもそも後戻りする意味はないです。つまり、不一致の場合に次の照合開始地点は前の照合開始地点よりも後ろになって欲しいです。

				D	B	A	B	D						
				A	C	A	B	D						

上の例では照合失敗を検知するまで 4 回比較しており、最後に照合した 1 点の index + 4 の地点から照合を開始しても問題ありません。よってスキップする大きさを決めるには、

$$skip\_size = \max(\text{比較回数、スキップテーブルの値})$$

とすれば良いです。

iii.　BM 法の実装

実装のポイントは以下の通りです。

- スキップテーブルの実装
- スキップテーブルを利用して文字列照合

コード 11　BM 法の実装

```
def bm(text: str, pattern: int) -> int:
  def create_table(pattern: str) -> list[int]:
    skip = [0] * (1 << 7)
    for i in range(1 << 7):
      last_match = -1
      for j in range(len(pattern)):
        if pattern[j] == chr(i):
          last_match = j

      if last_match == -1 or last_match == len(pattern) - 1:
        skip[i] = len(pattern)
      else:
        skip[i] = len(pattern) - last_match - 1
    return skip

  skip = create_table(pattern)

  text_cursor = len(pattern) - 1

  while text_cursor < len(text):
    moving_text_cursor = text_cursor
    pattern_cursor = len(pattern) - 1

    while text[moving_text_cursor] == pattern[pattern_cursor] and
        pattern_cursor > 0:
      moving_text_cursor -= 1
      pattern_cursor -= 1

```

```
28      if pattern_cursor == 0:
29        return text_cursor - len(pattern) + 1
30
31      # スキップテーブルを見てtextの参照開始位置を更新
32      compared_times = len(pattern) - pattern_cursor
33      skip_table_value = skip[ord(text[moving_text_cursor])]
34
35      text_cursor += max(skip_table_value, compared_times)
36
37    return -1
38
39
40  text = input()
41  pattern = input()
42
43  print(bm(text, pattern))
```

IX.　ラビン・カープ法 (ローリングハッシュ)

照合パターンとテキストのハッシュを計算し、そのハッシュが一致するか否かで文字列の一致を判定する方法を紹介します。照合パターンの長さを l とすると、ハッシュを利用することで比較の計算量を $O(l)$ から $O(1)$ へと定数時間にすることが可能です。

ハッシュは以下のように計算します。互いに素な定数 $a, h(1 < a < h)$ を用意して、照合パターン $(p_0 \cdots p_{l-1})$ に対して、

$$H(P) = (a^{l-1}p_0 + a^{l-2}p_1 + \cdots + a^0 p_{l-1}) \mod h \tag{1}$$

テキストの部分文字列 (text[0] から text[l-1]) のハッシュも同様に計算して $H(S, 0, l-1)$ とすると、H(P) と H(S, 0, l - 1) が一致するとき、text[0] から text[l-1] と照合パターンが一致すると考える。もちろんハッシュが衝突する可能性もあるので実際に一致しているか確認する必要がありますが、今回はハッシュが一致していれば文字列も一致しているとします。

しかし、このハッシュの計算をテキストの大きさだけ計算すると、$O(nl)$ の計算量になってしまいます。ここで、テキストのハッシュは $\mod h$ を取ったものであるため、

$$a^{l-1}p_0 + a^{l-2}p_1 + \cdots + a^0 p_{l-1} = H(S, 0, l-1) + Ah(\text{A は整数}) \tag{2}$$

と表せます。

これを利用すると、$H(S, 1, l)$ は前の $H(S, 0, l-1)$ を用いて以下のように表せます。

$$\begin{aligned} H(S, 1, l) &= a^{l-1}s_1 + a^{l-2}s_2 + \cdots + a^0 s_l \\ &= a(a^{l-2}s_1 + a^{l-3}s_2 + \cdots + a^1 s_{l-1}) + s_l \\ &= a(a^{l-1}s_0 + a^{l-2}s_1 + \cdots + a^0 s_{l-1}) - a^l s_0 + a^0 s_l \\ &= aH(S, 0, l-1) - a^l s_0 + a^0 s_l \end{aligned} \tag{3}$$

よって、テキストのそれぞれのハッシュは前のハッシュを利用することで、定数時間で計算できることがわかりました。これを**ローリングハッシュ**といいます。

i.　ラビン・カープ法の実装

実装のポイントは以下通りです。

- a, h を適当に決める
- 文字を数値に変換する
- テキストと照合パターンのハッシュを保持して尺取法のように計算する

コード 12　ラビン・カープ法の実装

```python
def rolling_hash(text: str, pattern: str) -> list[int]:
    a = 31
    h = 998244353

    text_length, pattern_length = len(text), len(pattern)
    text_hash = pattern_hash = 0

    # a^lを先に計算　剰余を取らないと実行時間が非常に長くなる
    a_l = 1
    for i in range(pattern_length):
        a_l = (a_l * a) % h


    # 最初のハッシュを計算する. 式(1)より
    for i in range(pattern_length):
        text_hash = (a * text_hash + ord(text[i])) % h
        pattern_hash = (a * pattern_hash + ord(pattern[i])) % h

    for i in range(text_length - pattern_length + 1):
        if pattern_hash == text_hash:
            return i

        if i < text_length - pattern_length:
            # テキストのハッシュを更新
            text_hash = (a * text_hash - a_l * ord(text[i]) + ord(text[i
                + pattern_length])) % h

            if text_hash < 0:
                text_hash += h

    return -1
```

X.　問題

問題 1 ABC 276 A - Right

英子文字からなる文字列が与えられます。最後に現れる a の index を求める問題です。存在しな

ければ、-1 を出力する問題です。力任せ法と Python の標準ライブラリを使って 2 つの解法を使って解いてみましょう。実際に AtCoder に参加するときなどは、標準ライブラリを使った解法を使うことが多いですが、勉強中は自分で実装することが大切です。

コード 13　問題 1 の解答

```
def brute_force_last_index(text: str, pattern: str) -> int:
    last_index = -1
    for text_cursor in range(len(text) - len(pattern) + 1):
        pattern_cursor = 0
        moving_text_cursor = text_cursor
        while pattern_cursor < len(pattern) and text[moving_text_cursor]
           == pattern[pattern_cursor]:
            pattern_cursor += 1
            moving_text_cursor += 1

        if pattern_cursor == len(pattern):
            last_index = text_cursor



    return last_index

text = input()
pattern = "a"

last_index = brute_force_last_index(text, pattern)

print(last_index + 1 if last_index != -1 else last_index)

# 標準ライブラリを使った解法
text = input()
pattern = "a"

last_index = text.rfind(pattern)

print(last_index + 1 if last_index != -1 else last_index)
```

問題 2 ABC 336 B - CTZ

コード 14　問題 2 の解答

```python
def cal_ctz(number: int) -> int:
    bit_size = number.bit_length()
    shift_size = 0

    while shift_size < bit_size:
        if number & (1 << shift_size):
            return shift_size
        else:
            shift_size += 1

def main():
    n = int(input())
    ctz = cal_ctz(n)

    print(ctz)

if __name__ == "__main__":
    main()
```

問題 3 yukicoder No.430 文字列検索ローリングハッシュの練習問題です。

参考

- https://blog.hamayanhamayan.com/entry/2016/10/03/083532

問題 ABC 141 E - Who Says a Pun? ローリングハッシュ + α の難しい問題です。 整数に関しては以下の内容を扱います。

- 最大公約数とユークリッドの互除法
- 拡張ユークリッドの互除法
- 素数判定
  - ナイーブな実装
  - エラトステネスの篩
- 素因数分解
  - ナイーブな実装
  - SPF を用いた実装
- 冪乗: 繰り返し二乗法
- 逆元とフェルマーの小定理

XI.　最大公約数とユークリッドの互除法

整数 $a, b(a > b)$ の最大公約数を求めるには、**ユークリッドの互除法**を使用します。

ユークリッドの互除法

$a, b(a > b)$ の最大公約数は a % b と b の最大公約数と等しい。

a, b の剰余を計算繰り返し計算していって b が 0 になった時の a が最大公約数となります。繰り返し同じ計算をするので、**再帰関数**を使って実装します。

コード 15　ユークリッドの互助法実装

```python
def gcd(a: int, b: int) -> int:
    # 終了条件
    if b == 0:
        return a

    if a < b:
        a, b = b, a

    return gcd(b, a % b)
```

最大公約数をアルゴリズムを理解するだけではなくて、図形を使って直感的に理解しましょう。参考に記したけんちょんさんの最大公約数の記事を参考にすると、最大公約数は n 次元の立方体を考えたときにそれを均等に分割できる最大の立方体の辺の長さとして考えることができます。このように考えると、最大公約数が何を意味しているのかが直感的に理解できるかもしれません。

i.　問題と解説

1.　ABC 118 C - Monsters Battle Royale

問題からはすぐには気づきにくいですが、最大公約数を利用します。n 次元の立方体を考えたときにそれを均等に分ける最大の立方体の辺の長さを求める問題と同じです。配列 A のどの要素も gcd の倍数になっているので、gcd がこれ以上分割できない最小の辺の長さとなります。どんなに操作を加えても最小の辺の長さは変わりません。このように操作を繰り返しても変わらない数のことを**不変量**といいます。与えられた配列 A の要素同士の引き算から必ず gcd を生み出せるかという疑問が生まれるかもしれませんが、ユークリッドの互除法を使うことで、gcd を求めることができます。

コラム 再帰関数

おそらく初めて再帰関数を見た方は再帰関数の動きがわかりにくいと思います。再帰関数は自分自身を呼び出す関数です。そのため、再帰関数を理解するためには、関数が呼び出された時にどのような動きをするのかを理解する必要があります。関数が呼び出されると、スタックに関数が積まれていきます。最初に呼び出した gcd は一番下に、最後に呼び出した gcd はスタックの一番上に置かれます。一番上の関数戻り値がわかったら、連鎖的にそれより下の関数の値もわかります。そして今回の場合シグネチャからもわかる通り、int のデータ型の値を返す関数です。return する箇所で再び int の値を返す関数を呼び出しているので、スタックに積まれた関数が return するまで関数が呼び出され続けます。イコールが終了条件を満たすまで連なっていると考えるとわかりやすいかもしれません。

XII.　拡張ユークリッドの互除法

ユークリッドの互除法では、最大公約数を求めることができますが、拡張ユークリッドの互除法を使うと、最大公約数だけでなく、$ax+by=\gcd(a,b)$ を満たす x,y を求めることができます。

$ax+by=gcd(a,b)$ の a,b に対してユークリッドの互除法を適用していきます。通常のユークリッドの互除法のように、$a\%b,b$ を計算していきますが、計算した結果それがれが次の一次不定方程式の係数となります。$bx_1+\{(a\%b)y_1\}=gcd(a,b)$ となります。ここで、

$$a\%b=a-(a//b)\times b$$

が成立することから、代入して

$$\begin{aligned} bx_1+ay_1-(a//b)\times by_1&=gcd(a,b)\\ ay_1+b(x_1-(a//b)y_1)&=gcd(a,b)\end{aligned}$$

$ax+by=gcd(a,b)$ と a,b について係数比較をすることで、

$$x=y_1,y=x_1-(a//b)\times y_1 \tag{4}$$

x_1,y_1 は $b,a\%b$ に対して再帰的に求めることができます。終了条件は $b=0$ のときで、このときの x が求める x となります。

具体例を見ましょう。$a=108,b=56$ つまり $108x+56y=gcd(108,56)$ です。

$$108x + 56y = gcd(108, 56)$$
$$56x_1 + 52y_1 = gcd(56, 52)$$
$$52x_2 + 4y_2 = gcd(52, 4)$$
$$4x_3 + 0y_3 = gcd(4, 0)$$

$ax+by = gcd(a,b)$ の解 (x,y) が b と $a\%b$ を係数にした一次不定方程式の解 (x_1,y_1) から求められていることを確認してください。上から順に再帰的に (b, a%b) を求めています。$b=0$ になったとき、$x_3 = 1, y_3 = 0$ になります。式 (1) より、$x_2 = y_3 = 0, y_2 = x_3 - (52//4) \times y_3 = 1$ となります。同様に、$x_1 = y_2 = 1, y_1 = x_2 - (56//54)y_2 = -1$

以上より、$x = y_1 = -1, y = x_1 - (108//56) \times y_1 = 2$ となります。

実装は以下のようになります。

コード 16　拡張ユークリッドの互助法実装

```python
def extended_gcd(a: int, b: int) -> tuple[int, int, int]:
    if b == 0:
        return a, 1, 0
    else:
        gcd, x1, y1 = extended_gcd(b, a % b)

        # 係数の更新
        x = y1
        y = x1 - (a // b) * y1

        return gcd, x, y
```

XIII.　素数判定

i.　ナイーブな実装

ナイーブな実装では以下の定理を利用して、自然数 n の素数判定を行います。

定理 1

自然数 n が $\sqrt{n}$ 以下の素数で割り切れないならば、n は素数である。

実装は以下のようになります。

コード 17　ナイーブな素数判定

```python
def is_prime(n: int) -> bool:
    # 忘れないように注意
    if n <= 1:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1

    return True
```

ナイーブな実装では、1 個の自然数が素数か判定するには $O(\sqrt{n})$ でできますが、n 個の数を判定するとなったら $O(n\sqrt{n})$ かかり少し遅いです。

ii.　エラトステネスの篩

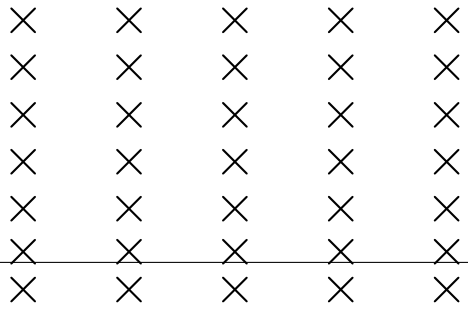
多くの自然数の素数判定を高速に行う方法の 1 つに**エラトステネスの篩**があります。求める自然数の最大値を n とすると、1 から n までの自然数の素数判定はエラトステネスの篩を使うと、$O(n \log \log n)$ の計算量で実行可能です。

エラトステネスのの篩では、判定したい自然数 n までのテーブルを作成し、2 から順に以下の操作を $\sqrt{n}$ まで繰り返します。

1. $i = 2$ からスタート
2. i がテーブルでバツがついていない場合、i を除いた i のすべての倍数を素数ではないとする
3. $i = i + 1$ ($i = \sqrt{n}$ まで 2 へ)

例として $n = 100$ の場合を見てみます。

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



i = 2 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 3 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 5 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 7 の場合

1 を除いて残った数が素数です。

iii.　エラトステネスの篩の実装

コード 18　エラトステネスの篩の実装

```
def is_prime(n: int) -> list[bool]:
    is_prime_list = [True] * (n + 1)
    is_prime_list[0] = is_prime_list[1] = False
    i = 2
    while i * i <= n:
        if is_prime_list[i]:
            for j in range(i * i,n + 1,i):
                is_prime_list[j] = False
        i += 1

    return is_prime_list
```

XIV.　素因数分解

i.　ナイーブな実装

ナイーブな素因数分解では、与えられた自然数 n を 2 から $\sqrt{n}$ まで順に割っていき、1 になれば終了します。割り切れた数を素因数としてリストに追加していきます。$\sqrt{n}$ まで続けて n が 1 でない場合は、その数も素因数としてリストに追加します。

コード 19　ナイーブな素因数分解の実装の実装

```python
def enumerate_prime_factors(n: int) -> list[int]:
    primes = []
    i = 2
    while i * i <= n:
        if n % i == 0:
            primes.append(i)
            n //= i
            while n % i == 0:
                primes.append(i)
                n //= i
        i += 1

    # 残ったnが素数の場合
    if n > 1:
        primes.append(int(n))

    return primes
```

ナイーブな実装では、自然数 n が与えられたときの計算量は $O(\sqrt{n})$ です。m 個の素因数分解をするときは計算量が $O(m\sqrt{n})$ となります。

ii.　SPF を用いた実装

SPF(Smallest Prime Factor: 自然数 n を割り切る最小の素数) を用いることで、ナイーブな実装よりも高速に素因数分解を行うことができます。SPF を用いた素因数分解はエラトステネスの篩をする際に SPF を記録することで実装できます。

コード 20　SPF を用いた素因数分解の実装

```python
def spf(n: int) -> list[int]:
        spf_table = [i for i in range(n + 1)]
        is_prime = [True] * (n + 1)

        i = 2

        while i * i <= n:
            if is_prime[i]:
                for j in range(i + i, n + 1, i):
                    is_prime[j] = False
```

```
                    if spf_table[j] == j:
                        spf_table[j] = i

            i += 1

        return spf_table

    def prime_factorization(n: int) -> list[int]:
        primes = []
        spf_table = spf(n)

        while n > 1:
            prime = spf_table[n]
            n //= prime
            primes.append(prime)

        return primes
```

iii.　互いに素

互いに素

$a, b \in \mathbb{Z}$ が互いに素であるとは、a, b の最大公約数が 1 であることをいう。

互いに素の性質をいくつか紹介します。

$a, b \in \mathbb{Z}$ の最大公約数を g とすると、

$$a = g \cdot a', b = g \cdot b' (\text{a' と b' は互いに素})$$

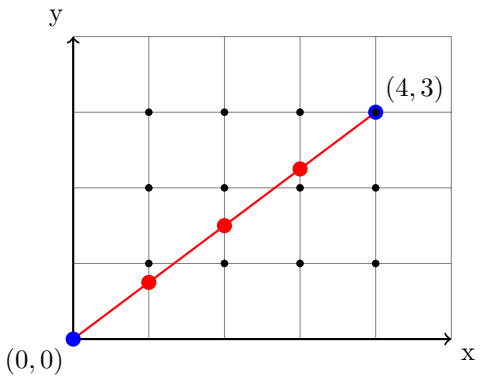
と互いに素な数と最大公約数の積で表すことができる。

この性質の応用例は 2 点間の格子点の個数です。

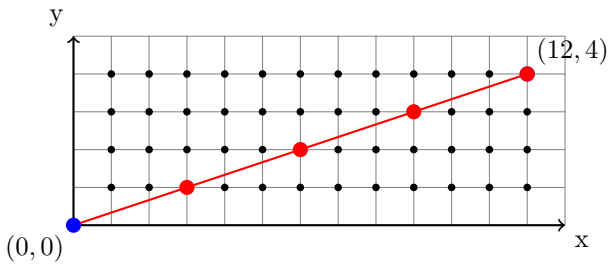
2 次元座標平面上に 2 点 $(x_1, y_1), (x_2, y_2)$ が与えられたとき、片方が原点になるように並行移動した 2 点を結ぶ直線上の格子点の個数は、2 点間の最大公約数を g とすると、$g-1$ 個です。ただし、2 点は格子点の個数に含まないとします。

2 点の位置関係が重要で片方を原点になるように並行移動しても一般性を失わないので、$x = x_2 - x_1, y = y_2 - y_1$ とします。まず、x, y が互いにその場合を考えます。互いに素のときは

$(0,0)$ と (x,y) を結ぶ線分の間には格子点は存在しません。



次に、(x,y) が互いに素ではない場合、つまり最大公約数が 1 よりも大きい場合を考えます。このとき、$x = g \cdot x', y = g \cdot y'$ と表すことができます。g が 1 よりも大きい場合は、$(0,0)$ と (x,y) を結ぶ線分は格子点を通ることがあります。この (x,y) を最大公約数 $g = 4$ で割ると、$(0,0)$ と $(3,1)$ となり、3 と 1 は互いに素です。先ほどのように互いに素の場合は格子点は存在しません。$d = 4$ というのはこの互いに素という格子点の最小単位が 4 つであることを示しています。今回の設定では、線分の端は格子点に含まれないので、格子点の個数は $4 - 1 = 3$ 個です。



実装例を以下に示します。

コード 21　2 点間の格子点の個数の実装

```
def GCD(a: int, b: int) -> int:
    if a < b:
        a, b = b, a
    if b == 0:
        return a

    return GCD(b, a % b)

def count_grid_points(x1: int, y1: int, x2: int, y2: int) -> int:
```

```
    x, y = abs(x1 - x2), abs(y1 - y2)
    return GCD(x,y) - 1
```

他にも様々な性質があります。

互いに素と一次不定方程式

$a, b \in \mathbb{Z}$ が互いに素であるとき、

$$ax + by = 1$$

を満たす $x, y \in \mathbb{Z}$ が存在する。

これは上で見た拡張ユークリッドの互除法そのものです。

$a, b \in \mathbb{Z}$ が互いに素であるとき、$b \in \mathbb{Z}$ に関して bc が a で割り切れるなら、c は a の倍数である。

証明

a, b は互いに素であるから、$ax + by = 1$ を満たす $x, y \in \mathbb{Z}$ が存在する。

$$c = c \times 1 = c(ax + by) = acx + bcy$$

acx は自明に a で割り切れる。bcy も条件より a で割り切れるから、c は a の倍数である。

合同式の割り算

$a, m \in \mathbb{Z}$ が互いに素であるとする。

$$ax \equiv by \mod m$$

であるとき、

$$x \equiv y(\mod m)$$

が成立する。

倍数の周期性

$a, m \in \mathbb{Z}$ が互いに素であるとする。

$$0, a, 2a, \cdots, (m-1)a$$

を m で割った余りを昇順に並び替えると、$0, 1, 2, \cdots, m-1$ になる。

逆元と合同方程式

$a, m \in \mathbb{Z}$ が互いに素であるとする。任意の $b \in \mathbb{Z}$ に対して、

$$ax \equiv b \mod m$$

を満たす $x \in \mathbb{Z}$ が m を法としてただ一つ在する。特に、$b = 1$ の場合を逆元という。

中国剰余定理

Euler 関数は乗法的

Euler の定理

iv.　問題 1.

素因数分解に関する問題です。

問題 約数列挙

自然数 N が与えられるので、N の約数を昇順に列挙せよ。ただし、N は $1 \leq N \leq 10^12$ を満たす。

例

入力: 12

出力: 1 2 3 4 6 12

回答:

```python
def enumerate_divisors(n: int) -> list[int]:
    divisors = set()
    for i in range(1, int(n ** 0.5) + 1):
        if n % i == 0:
            divisors.add(i)
            if i != n // i:
                divisors.add(n // i)

    return list(divisors)


def main():
    n = int(input())
    divisors = enumerate_divisors(n)

    print(*divisors)

if __name__ == "__main__":
    main()
```

問題 2.　ABC 057 C - Digits in Multiplication

約数列挙の問題です。

問題 3.　ARC 052 C - Factors of Factorial

約数の個数を求める問題です。約数の個数も素因数分解の結果から求めることを利用します。

問題 4.　ARC 026 B - 完全数

約数の総和

オイラー関数

オイラー関数とは、自然数 N が与えられたときに、$1, 2, \cdots, N$ のうち n と互いに素になる数の個数を求める関数 $\Phi(N)$ です。

回答:

```python
def enumerate_divisors(n: int) -> list[int]:
    divisors = set()
    for i in range(1, int(n ** 0.5) + 1):
        if n % i == 0:
            divisors.add(i)
            if i != n // i:
                divisors.add(n // i)

    return list(divisors)


def main():
    n = int(input())
    divisors = enumerate_divisors(n)

    print(*divisors)

if __name__ == "__main__":
    main()
```

問題 5 yukicoder No.442 和と積

2 つの整数 a, b が与えられるので、$a+b, a \times b$ の最大公約数を求める問題です。

コード 22　2 点間の格子点の個数の実装

XV.　冪乗: 繰り返し二乗法

x^n を求める際に単に $x \times x \times \cdots \times x$ と計算すると $O(n)$ の計算量がかかります。繰り返し二乗法を用いると、$O(\log n)$ で計算することができます。また、冪乗では計算結果が非常に大きくなることがあるので、計算結果を mod で割った余りを求めることがあります。剰余で答えを出すときの演

算によって処理が異なるので注意が必要です。

- 加算: 加算した後で mod を取る
- 減算: 減算した後で mod を取る
- 乗算: 乗算した後で mod を取る
- 除算: 計算途中で mod を取るときは逆元を用いる (後述)

コード 23　繰り返し二乗法の実装

```
def exp_mod(n: int, m: int, mod: int) -> int:
    if m == 0:
        return 1
    if m == 1:
        return n % mod
    ans = 1
    m1, m2 = m // 2, m % 2
    ans *= exp_mod(n, m1, mod) % mod
    ans = (ans * ans) % mod

    ans *= exp_mod(n, m2, mod)
    ans %= mod

    return ans
```

XVI.　逆元とフェルマーの小定理

除算の mod を取るとき、計算途中で mod をもとった場合と最後に mod を取った場合で結果が異なることがあります。例えば、$a = 20, b = 4, mod = 6$ で考えます。

- **計算途中で mod を取る場合:**
  1. まず、a と b に対して mod を取ります:

$$a \mod m = 20 \mod 6 = 2$$

$$b \mod m = 4 \mod 6 = 4$$

  2. 次に、得られた mod の値で除算を行い、さらに mod を取ります:

$$\left(\frac{a \mod m}{b \mod m}\right) \mod m = \left(\frac{2}{4}\right) \mod 6$$

  通常の整数除算では、これは 0（小数は考慮しない）となるため、

$$0 \mod 6 = 0$$

結果は 0 になります。

- **最後に mod を取る場合:**
  1. まず、a を b で割ります:
  $$\frac{20}{4} = 5$$
  2. 次に、その結果に対して mod を取ります:
  $$5 \mod 6 = 5$$

除算の mod を求めるときは、計算途中で mod を取ると結果が異なることがあるので、逆元を用いて計算します。

i.　フェルマーの小定理と逆元

除算における mod を考えるために**フェルマーの小定理**と**逆元**を導入します。

フェルマーの小定理

a は任意の自然数、m を素数で a, m が互いに素であるとき、以下の式が成り立つ。

$$a^{m-1} \equiv 1 \pmod{m}$$

逆元

m が素数で、a が m では割り切れない整数であるとき、以下の式を満たす x が [1, m) の範囲で一意に存在する。このような x を $\pmod{m}$ における a の逆元と呼ぶ。

$$ax \equiv 1 \pmod{m}$$

ここで、$a^{m-1} \equiv a \times a^{m-2} \equiv 1 \pmod{m}$ となるので、a^{m-2} が a の逆元であることがわかります。以上より以下のことがわかります。

a で割ることは、a の逆元をかけることと等しい

逆元を使った問題を解いてみましょう。

問題 ${}_n\mathrm{C}_k$ を 998244353 で割ったあまりを求めるプログラムを作成せよ。

回答 ${}_n\mathrm{C}_k = \frac{n!}{(n-k)!k!}$ の剰余を求めるます。 mod の世界で除算が登場しているので、逆元を用いて計算します。乗算するのは $(n-k)!, k!$ であることから、これらで割ることは mod の世界では、これらの逆元つまり、それぞれ $(n-k)!^{(DIV-2)}, k!^{(DIV-2)}$ をかけることと等しいです。

コード 24　${}_n\mathrm{C}_k$ を求める

```python
DIV = 998244353

def factorial(n: int) -> list[int]:
    memo = [1] * (n + 1)
    for i in range(1, n + 1):
        memo[i] = (memo[i-1] * i) % DIV

    return memo

def inverse(n: int) -> int:
    return pow(n, DIV - 2, DIV)

def comb(n: int, k: int, memo: list[int]) -> int:
    return (memo[n] * inverse(memo[k]) * inverse(memo[n-k])) % DIV

n, k = map(int, input().split())
memo = factorial(n)
print(comb(n, k, memo) % DIV)
```

XVII.　参考

参考になるサイトには扱ったアルゴリズムの別の見方や演習問題などがあるので、以下にリンクを示します。

最大公約数

- https://qiita.com/drken/items/0c88a37eec520f82b788

素数判定

- https://algo-method.com/tasks/318

素因数分解

- https://qiita.com/drken/items/a14e9af0ca2d857dad23#%E5%95%8F%E9%A1%8C-10-abc-150-d---semi-comm
  E7%82%B9

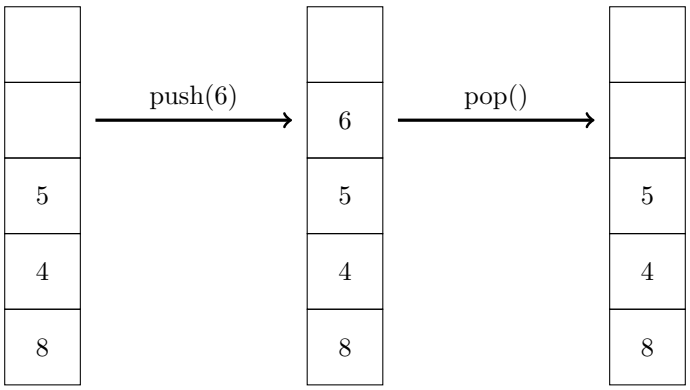
EEIC2024 アルゴリズムの授業で扱ったデータ構造のアルゴリズムのシケプリです。扱った内容は以下の通りです。

- スタック
- キュー
- 線形リスト
- ツリー
- ヒープ
- セグメント木 (発展的な内容)
- BIT(発展的な内容)

XVIII. スタック

スタックは LIFO(Last In First Out) といって、最後に入れたデータが最初に取り出されるデータ構造です。本を積み重ねるイメージで考えるとわかりやすいです。スタックには以下の操作があります。

- push: スタックにデータを追加する
- pop: スタックからデータを取り出す



スタックのイメージ

i. スタックの実装

実装のポイントは以下の 3 つです。Python の class を使って実装します。

- スタックは配列で実装する
- 要素が入る位置を示すポインタを持つ
- push と pop のメソッドを実装する

実装は以下の 26 のようになります。

コード 25　スタックの実装

```
  class Stack:
    def __init__(self, size: int) -> None:
        self.stack: list[int] = [0] * size
        self.top: int = 0

    def push(self, value: int) -> None:
        if self.top < len(self.stack):
            self.stack[self.top] = value
            self.top += 1
        else:
            print("This stack is full.")

    def pop(self) -> None:
        if self.top > 0:
            self.top -= 1
            return_value = self.stack[self.top]
            return return_value
        else:
            print("This stack is empty.")

def main():
    size = 5
    stack = Stack(size)

    stack.push(8)
    stack.push(4)
    stack.push(5)
    stack.push(6)
    return_value = stack.pop()

    print(f"returned value is {return_value}")
```

```
32
33 main()
```

Column1 collections.deque

実際にスタックを使用するときは、Python の標準ライブラリである collections の deque を使うと便利です。Python では Stack スタックやキューが明示的に用意されていないため、双方向キューである deque を使ってスタックやキューを使用します。スタックは上記のように配列で簡単に実装可能ですが、標準ライブラリの deque を使うことで、より高速にスタックを使用できます。

コード 26　スタックの実装

```
1     def main():
2     from collections import deque
3     stack = deque()
4
5     stack.append(8)
6     stack.append(4)
7     stack.append(5)
8     stack.append(6)
9
10    return_value = stack.pop()
11
12    print(f"returned value is {return_value}")
13
14 main()
```

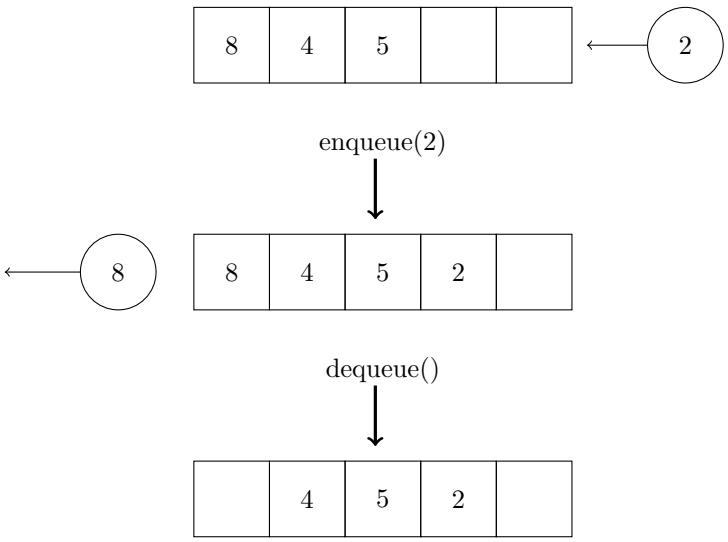
XIX. キュー

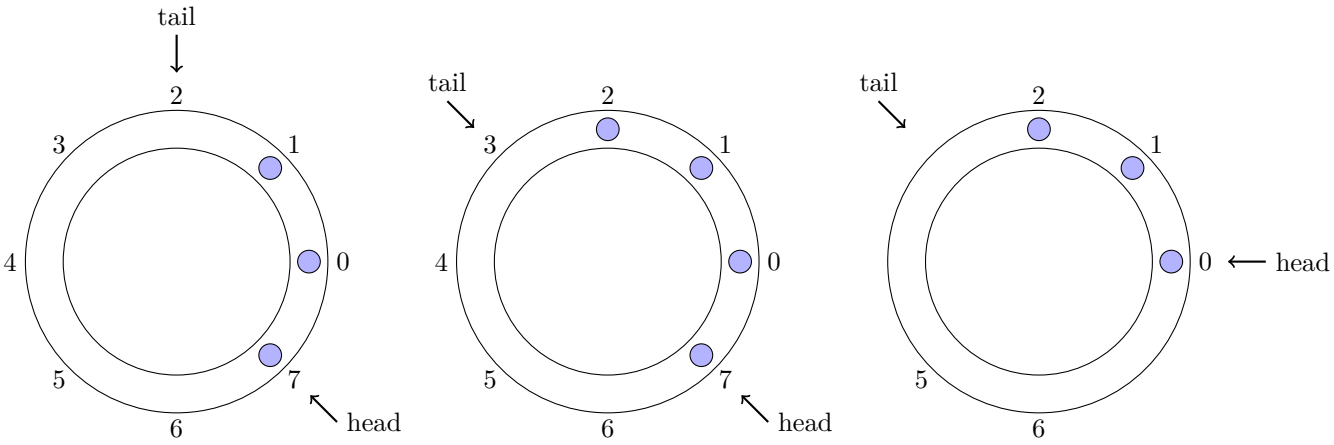
キューは FIFO(First In First Out) といって、最初に入れたデータが最初に取り出されるデータ構造です。ディズニーランドの待ち行列のようなイメージで考えるとわかりやすいです。キューには以下のような操作があります。

- enqueue: キューにデータを追加する
- dequeue: キューからデータを取り出す

先ほどのスタックと違って配列の頭からデータを取り出していることに注意してください。つまり、dequeue をするたびに取り出すデータの位置をずらす必要があります。従って、データを取り出すための先頭アドレス、データを追加するための末尾アドレスを持つ必要があります。

しかし、ここで困ったことがあります。dequeue をするたびに配列の頭が右にずれていくと、配列の先頭にデータが残っているにも関わらず、配列の末尾にデータが追加できなくなります。この問題を解決するために、リングバッファというデータ構造を使います。





enqueue()　　**dequeue()**

リングバッファの仕組みを見てみましょう。enqueue をするときは tail の位置にデータを追加し tail を 1 だけ進めます。dequeue をするときは head の位置のデータを取り出し head を 1 だけ進めます。head と tail が同じ位置になったときは、キューが空になります。実装の際は、要素の数で割った余りを使って head と tail の位置を管理します。

i.　キューの実装

キューの実装のポイントは以下の 3 つです。

- キューは配列で実装する
- 取り出す位置を示すポインタと追加する位置を示すポインタの 2 つを持つ
- リングバッファをモジュロ演算で実装する (%を使う)
- enqueue と dequeue のメソッドを実装する

コード 27　キューの実装

```
1   class Queue:
2     def __init__(self, size: int) -> None:
3         self.queue = [0] * size
4         self.size = 0
5         self.head = 0
6         self.tail = 0
7
8     def enqueue(self, value: int) -> None:
9         if not self.is_full():
10            self.size += 1
11            self.queue[self.tail] = value
```

```
12              self.tail = (self.tail + 1) % len(self.queue)
13          else:
14              print("queue is full")
15
16      def dequeue(self) -> int:
17          if not self.is_empty():
18              self.size -= 1
19              return_value = self.queue[self.head]
20              self.head = (self.head + 1) % len(self.queue)
21              print(return_value)
22          else:
23              print("queue is empty")
24
25
26      def is_empty(self):
27          return self.size == 0
28
29      def is_full(self):
30          return self.size == len(self.queue)
31
32  queue = Queue(5)
33
34  queue.enqueue(8)
35  queue.enqueue(4)
36  queue.enqueue(5)
37
38  queue.enqueue(2)
39
40  queue.dequeue()
```

Column2 collections.deque

スタックと同様にキューも実際に使用するときは、Python の標準ライブラリである collections の deque を使うと便利です。

```
1       from collections import deque
```

```
2
3  queue = deque()
4
5  queue.append(8)
6  queue.append(4)
7  queue.append(5)
8
9  queue.append(2)
10
11 return_value = queue.popleft()
12
13 print(f"returned value is {return_value}")
```

XX.　線形リスト

線形リストとは、以下の図のようにデータと次のポインタを持つデータを連結させるデータ構造です。線形リストは必要なときに必要なデータを追加していくことができるため、空間計算量的に効率がいいです。しかし、データを取り出すときは先頭から順番にたどる必要があるため、時間計算量的には効率が悪いです。

今回実装する線形リストは以下の操作ができるものです。

- 片方向の線形リスト
- 新しい要素は常に末尾に追加する
- 検索は先頭から順番に行う

双方向の線形リストや、データの追加や削除を任意の位置で行うものもありますが、今回は片方向の線形リストを実装します。

i.　線形リストの実装

先ほどのスタックやキューは配列を使って実装しましたが、線形リストはポインタ (Python ではクラスのインスタンス) を使って要素をつなげて実装します。

コード 28　線形リストの実装

```
class Node:
    def __init__(self, value) -> None:
        self.value = value
        self.next: Node = None

class LinkedList:
    def __init__(self) -> None:
        self.head = Node(None)

    def append(self, value: int) -> None:
        current_cell = self.head
```

```
12          while current_cell.next != None:
13              current_cell = current_cell.next
14
15          # 追加する数値を値にもつノード
16          new_cell = Node(value)
17          # 現在の最後のノードの次のノードに付け加える
18          current_cell.next = new_cell
19
20      def pop(self, value: int) -> Node | None:
21          """
22          valueと同じ数値を値に持つノードを取り除く
23          """
24          prev_cell = None
25          current_cell = self.head
26
27          while current_cell != None:
28              if current_cell.value == value:
29                  # 取り出すため、付け替える
30                  prev_cell.next = current_cell.next
31                  return value
32              else:
33                  prev_cell = current_cell
34                  current_cell = current_cell.next
35
36          return None
37
38  linked_list = LinkedList()
39
40  linked_list.append(10)
41  linked_list.append(535)
42  linked_list.append(51)
43  linked_list.append(40)
44  linked_list.append(40)
45  linked_list.append(1)
46
47  return_value = linked_list.pop(51)
```

```
48
49 print(return_value)
```

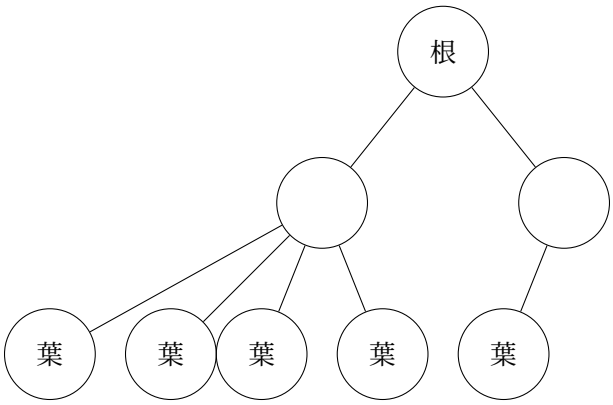
XXI.　ツリー (木構造)

木をひっくり返して根っこを上にして書いた木のような離散構造をツリーと言います。ツリーは用語が重要なため、以下の用語を覚えておくと良いです。

i.　木構造の用語

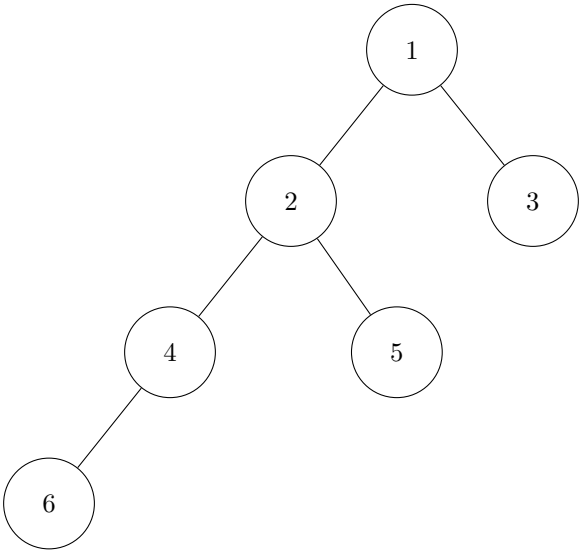
1.　根と葉

一番上のノードを**根**、一番下のノード (より下に子ノードを持っていないノード) **葉**といいます。



2.　深さと高さ

あるノードから繋がっている葉までの自分を除いたノードの数のうち最大の個数を**高さ**といいます。 葉ノード自体の高さは 0 です。また、根からあるノードまでの高さを**深さ**といいます。



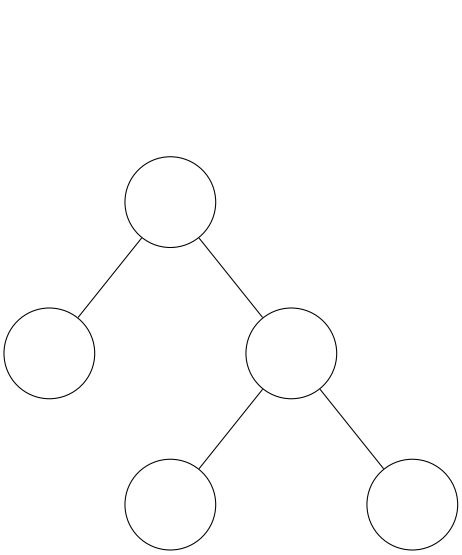
3. 二分木 (binary tree)

すべてのノードに関して、その子ノードの数が 2 以下である木を**二分木**といいます。二分木は以下のような特徴があります。上の木も二分木になっています。

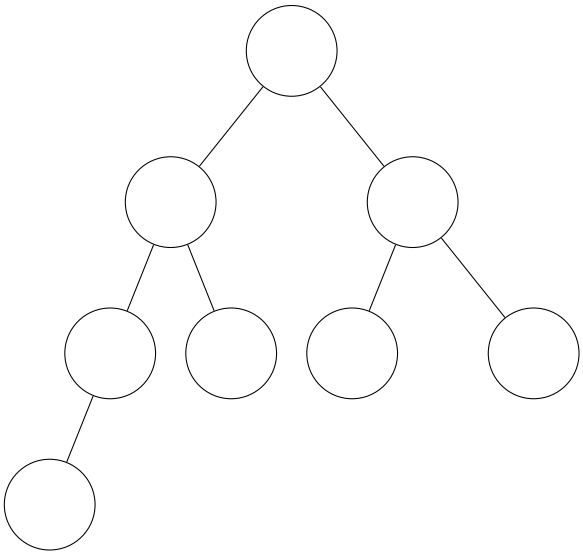
また、、完全二分木というものがあります。完全二分技は英語で表記すると full binary tree と complete binary tree があります。これらは独立した定義ですので、注意してください。違いを整理すると以下のようになります。

- full binary tree: すべてのノードが 0 個または 2 個の子ノードを持つ二分木
- complete binary tree: 根がある階層 (level) を除いて、すべての階層で埋まりうるノードがすべて埋まっているかつ、最後の階層のノードは左から右に向かって埋まっている二分木

両者の違いは最初は分かりにくいです、以下の図を見て理解を深めてください。



full binary tree の例



complete binary tree の例

XXII.　ヒープ

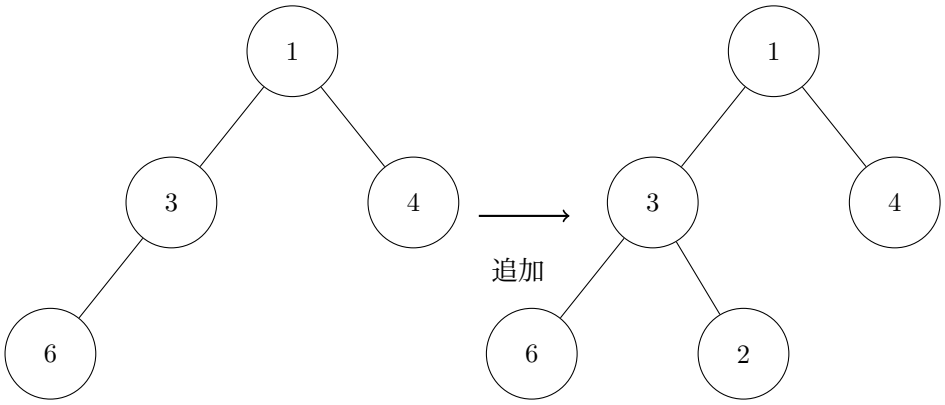
木構造を使ったデータ構造にヒープがあります。ヒープは以下の特徴があります。

- 親ノードは子ノードよりも大きい (または小さい) という性質を持つ
- 根ノードは常に最大 (または最小) となる

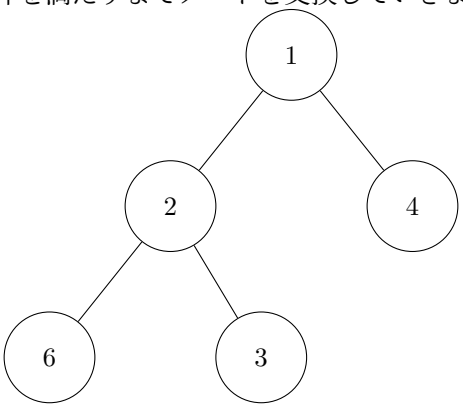
ヒープは最大値や最小値を根に持つため、最大値や最小値を取り出すのが高速です。さらに今回はヒープの中でさらに、complete binary tree になっている**二分ヒープ**というデータ構造を扱います。ヒープは complete binary tree に対して、以下の操作を行います。

- 要素の追加
- 要素の取り出し

要素の追加に関して、二分ヒープが complete binary tree になっているため、根がある階層 (level) の一番右に要素を追加してきます。追加時に、追加したノードから根までヒープの条件を満たすように再帰的にノードを交換していきます。以下は最小ヒープの例です。



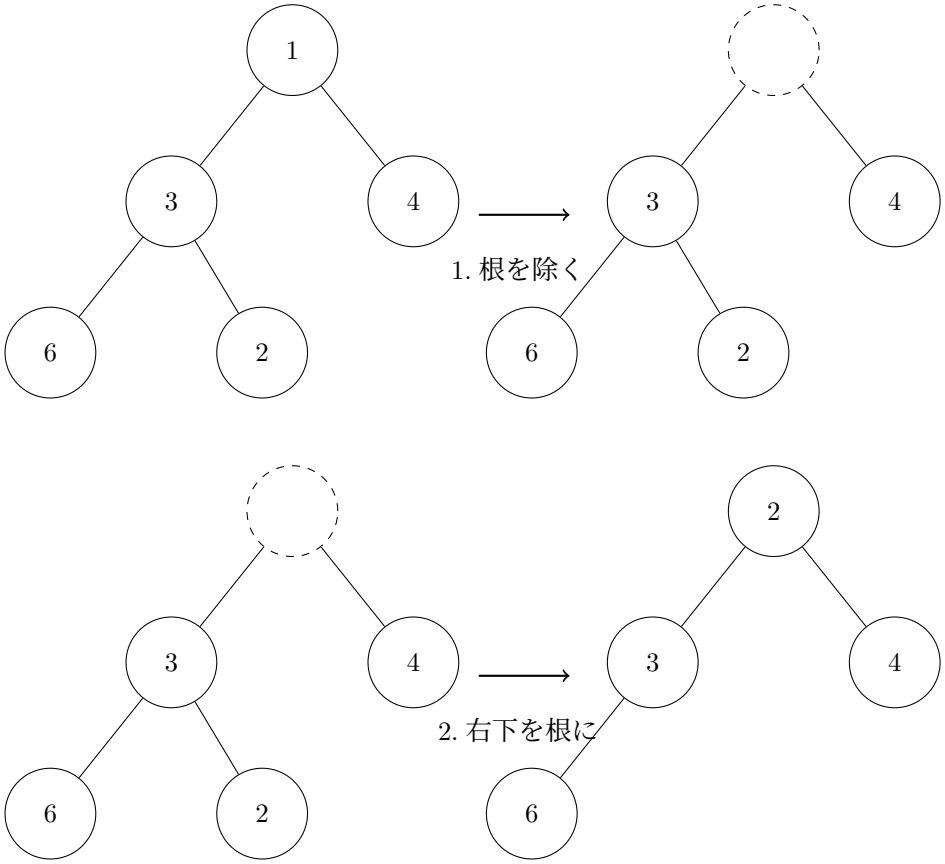
2 を追加すると、第二階層において親の値は子以下であるという性質を満たさなくなってしまいます。そこで、2 と 3 を交換します。実装としては追加したノードから根に向かって再帰的に親ノードと比較してヒープの条件を満たすまでノードを交換していきます。



二分ヒープからの削除は少し複雑です。ヒープでは必ず根から取り出すことに注意してください。以下のような操作を行います。

1. 根ノードを取り除く
2. 一番右下のノードを根に移動する
3. 根から下に向かって、子ノードと比較して、ヒープの条件を満たすように再帰的にノードを交換していく

一番右のノードを根に移動する理由は、complete binary tree を保つためです。もしも、根を取り除いた後に直接根の子ノードを根にしてしまうと、必ずしも complete binary tree にならず実装が複雑になります。



上の例では、根を取り除き一番右のノードを根に移動すると木がヒープの性質を満たしていますが、一般的にはヒープの性質を満たすまで根から下に向かってノードを再帰的に交換していきます。

二分ヒープの実装のポイントは

- 1-indexed の配列で実装する (要素数 + 1 の大きさの配列を用意する)
- 追加、削除時にヒープの条件を満たす様に再帰的に更新する

二分木は構造体を使わず配列だけで実装できる点が利点です。以下は最小ヒープの実装です。

コード 29　二分ヒープの実装

```
class Heap:
  def __init__(self, size: int) -> None:
    self.inf = 1 << 60
    self.size = size + 1
    self.heap = [self.inf] * self.size
    self.tail = 0 # 現在のヒープのデータ数

  def swap(self, index1: int, index2: int) -> None:
    self.heap[index1], self.heap[index2] = self.heap[index2], self.heap[
        index1]

  def push(self, value: int) -> None:
    if self.tail != self.size - 1:
      self.tail += 1
      self.heap[self.tail] = value
      self.check_from_leaf(self.tail)

  def check_from_leaf(self, node: int) -> None:
    # nodeが根なら再帰終了
    if node == 1:
      return

    parent = node // 2
    # ヒープの条件を満たさない
    if self.heap[parent] > self.heap[node]:
      self.swap(parent, node)
      self.check_from_leaf(parent)


  def pop(self) -> int | None:
    if self.tail != 0:
      popping_value = self.heap[1]
      # 一番右の葉ノードを根ノードに移動
      self.heap[1] = self.heap[self.tail]
```

```
      self.tail -= 1
      self.check_from_root(1)
      return popping_value
    else:
      return None

  def check_from_root(self, node: int) -> None:
    left_child, right_child = 2 * node, 2 * node + 1

    # left > rightよりこれだけで十分
    if left_child > self.tail:
      return

    # 2つの子ノードの大小で区別する
    smaller_node = larger_node = 0
    if self.heap[left_child] < self.heap[right_child]:
      smaller_node, larger_node = left_child, right_child
    else:
      smaller_node, larger_node = right_child, left_child

    # 親ノードと小さい方の子ノードを比較
    if self.heap[node] > self.heap[smaller_node]:
      self.swap(node, smaller_node)
      # 小さい方だけで十分
      self.check_from_root(smaller_node)
```

i.　問題

問題 1

ABC 141 D - Powerful Discount Tickets

とにかく一番高い商品の値段を半分の値段で買っていくのが良さそうです。上で実装した自作ヒープと Python の標準ライブラリの heapq の両方を使って解きます。

コード 30　問題 1

```
class MinHeap:
    def __init__(self, size: int) -> None:
        self.inf = 1 << 60
```

```
        self.size = size + 1
        self.heap = [self.inf] * self.size
        self.tail = 0

    def _swap(self, index1: int, index2: int) -> None:
        self.heap[index1], self.heap[index2] = self.heap[index2], self.
            heap[index1]

    def push(self, value: int) -> None:
        if self.size - 1 <= self.tail:
            raise OverflowError("the heap is full")

        self.tail += 1
        self.heap[self.tail] = value

        self._reflesh_from_leaf(self.tail)

    def _reflesh_from_leaf(self, node: int) -> None:
        if node == 1:
            return

        child = node
        parent = child // 2

        if self.heap[child] < self.heap[parent]:
            self._swap(child, parent)
            self._reflesh_from_leaf(parent)

    def pop(self) -> int | None:
        if self.tail == 0:
            return None

        popping_value = self.heap[1]
        self.heap[1] = self.heap[self.tail]
        self.heap[self.tail] = self.inf
        self.tail -= 1
```

```
39
40          self._refresh_from_root(1)
41
42          return popping_value
43
44      def _refresh_from_root(self, node: int) -> None:
45          if node > self.tail:
46              return
47          left, right = 2 * node, 2 * node + 1
48
49          if left <= self.tail and self.heap[node] > self.heap[left]:
50              self._swap(node, left)
51
52          if right <= self.tail and self.heap[node] > self.heap[right]:
53              self._swap(node, right)
54
55          self._refresh_from_root(left)
56          self._refresh_from_root(right)
57
58      def all(self) -> int:
59          sm = 0
60          for i in range(1, self.tail + 1):
61              sm += self.heap[i]
62
63          return sm
64
65  def main():
66      n, m = map(int, input().split())
67      A = list(map(int, input().split()))
68
69      heap = MinHeap(len(A))
70
71      for a in A:
72          heap.push(-a)
73
74      for _ in range(m):
```

```
75          max_price = -heap.pop()
76          discounted_price = max_price // 2
77
78          heap.push(-discounted_price)
79
80      print(-heap.all())
81
82
83  if __name__ == "__main__":
84      main()
85
86
87  # 別解 heapqを使用
88  from heapq import heapify, heappush, heappop
89
90  def main():
91      n, m = map(int, input().split())
92      A = list(map(int, input().split()))
93
94      # heapqは最小ヒープになっているため
95      for i in range(n):
96          A[i] = -A[i]
97
98      heapify(A)
99
100     for _ in range(m):
101         max_price = -heappop(A)
102         discount_price = max_price // 2
103         heappush(A, -discount_price)
104
105     print(- sum(A))
106
107 if __name__ == "__main__":
108     main()
```

XXIII.　セグメント木

セグメント木はある条件を満たす演算に関する区間の計算結果を高速に求めるデータ構造です。区間の計算結果を高速にもとめると聞いて累積和や尺取法を思い浮かべるかもしれませんが、セグメント木は扱うデータの要素が逐一変わるような状況でも区間の計算結果を高速に求めることができます。具体的な演算は、区間の最大・最小値や、合計値などがあります。

i.　数学の知識の確認

セグメント木の内容に入る前に、セグメント木を理解するのに必要な数学の知識を整理します。

1.　二項演算

自然数や文字列の集合 S の任意 2 つの元から新たに値を生成する操作のことを S **上の二項演算**といいます。特に、$x, y \in S$ に対する二項演算を $op(x, y)$ と書き、

$$op(x, y) = x * y$$

と表現することにします。

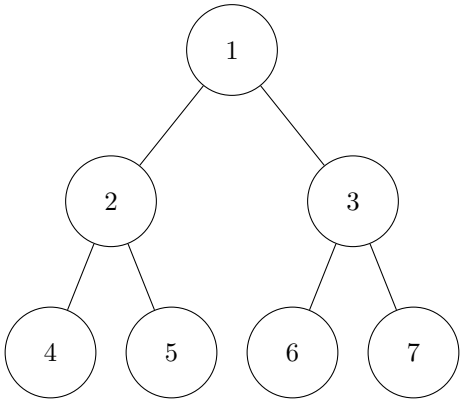
2.　閉じた演算

$x, y \in S$ に対して $op(x, y) \in S$ を満たすような演算を閉じた演算といいます。例えば、自然数の集合 S において、$op(x, y) = x + y$ は閉じた演算ですが、$op(x, y) = x - y$ は閉じた演算ではありません。

今回実装するセグメント木は、閉じた演算を扱うため、セグメント木の演算は閉じた演算になるようにします。

3.　完全二分木

セグメント木は完全二分木になっています。以下の図のように根以外のすべてのノードが 2 つの子ノードを持っていて、すべての葉の深さが同じ木になっています。



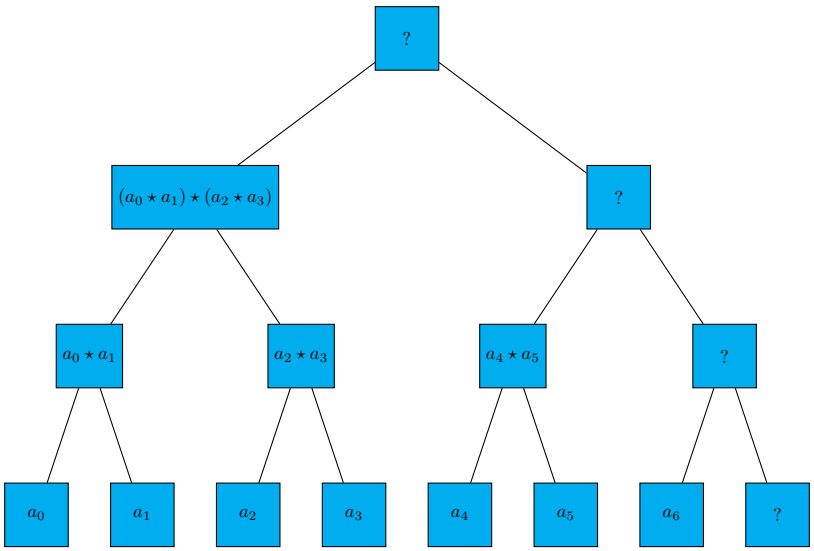
ii.　セグメント木の仕組み

セグメント木の理解に必要な知識の整理が終わったので、セグメント木の内容に入ります。

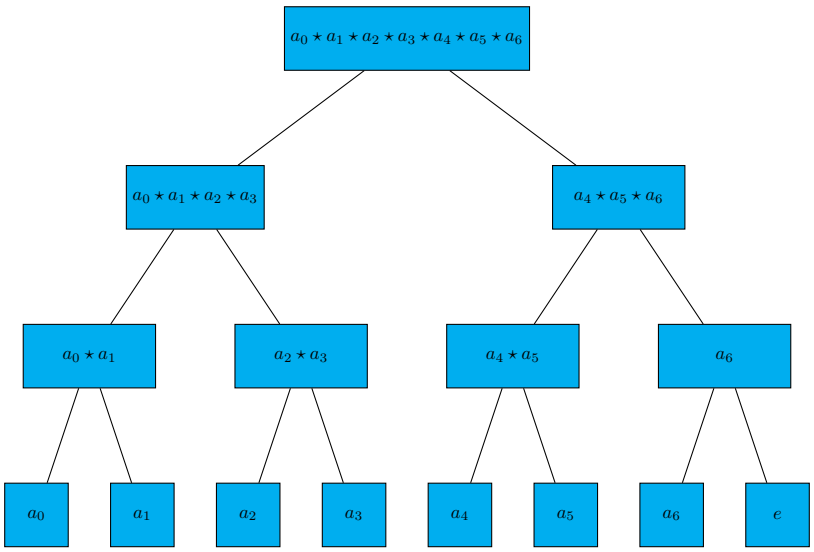
1.　セグメント木の構造

セグメント木の構造の特徴は以下の通りです。下の図は a_0からa_6 の 7 個のデータを持つセグメント木の例です。

- 葉ノードにデータを持つ
- 親ノードは 2 つの子ノードの演算結果を持つ
- $op(x, y)$ は結合法則を満たす $((x * y) * z = x * (y * z)$ が成立する$)$
- 集合 S に対する op に単位元が存在する



図には?があります。?を**全要素の総積をセグメント木の根が持つ**ように埋めていきます。下の図の e は単位元を表しています。根 $a_0 * a_1 * a_2 * a_3 * a_4 * a_5 * a_6$ は全要素の総積になっていることを考えると、その下の? は $a_4 * a_5 * a_6$ になります。同様に、そのほかの?も埋めていくと以下の図のようになります。



iii.　セグメント木の実装

セグメント木も二分木なので配列で表現することが可能です。ヒープと同様に 1-indexed で実装します。以下の操作ができるようなセグメント木を実装します。

- セグメント木の構築
- 要素の更新
- 区間の演算結果の取得

1.　セグメント木の構築

セグメント木では葉にデータが格納され、葉の数はデータの数以上でかつ 2^m で表せられる最小の数になります。2^m は 2 進数表記にすると $2^m = 1\underbrace{00\cdots0}_{m}$ となるので、2 進数表記したときに m 桁になる数と 2^m になる数であるときは葉の数を 2^m にします。つまり、データの数 n が $2^{m-1}+1 \leq n \leq 2^m$ であるとき、葉の数は 2^m になります。そして今回のセグメント木は 1-indexed の ed の木になっているため配列の 0 番目の要素は無視します。よって、配列全体の長さは $2m$ 個になります。

セグメント木では扱う演算についての単位元を用意する必要があります。例えば、区間の最大値を求めるセグメント木では、単位元は $-\infty$ になります。区間和では単位元は 0 になります。

葉をデータと単位元で埋めたら、その他のノードは 2 つの子ノードのデータを使って更新していきます。

2.　要素の更新

更新クエリに対しては、該当する葉ノードを更新した後、その親ノードを更新していきます。

3.　区間の演算結果の取得

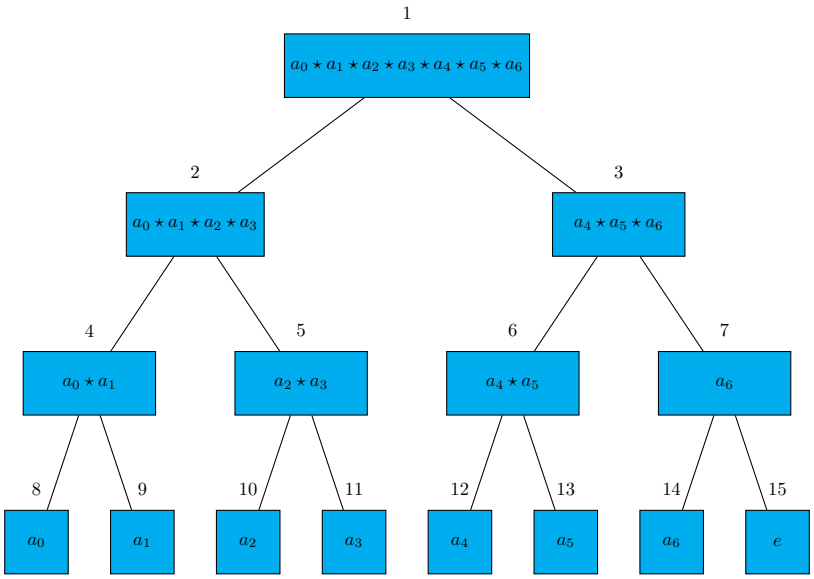
区間の演算結果の取得は少々複雑です。何個か例を挙げて説明します。右開区間 [left, right) の区間の演算結果を求めてみます。下のセグメント木を例にします。ノードの近くの数字は 1-indexed な配列でセグメント木を表現ししたときのインデックス番号です。

・$left = 0, right = 3$ のとき

求めるものは $a_0 * a_1 * a_2 = (a_0 * a_1) * a_1$ つまり 4 と 10 の演算結果になります。

・ $left = 1, right = 7$ のとき

求めるものは $a_0 * a_1 * a_2 * a_3 * a_4 * a_5 * a_6 = a_1 * (a_2 * a_3) * (a_4 * a_5) * a_6$ つまり、9 と 5 と 6 と 14 の演算結果になります。



どのノードを使うかを機械的に判断していくにはどうすればいいでしょうか。それは、$left$ と $right$ が親ノードから見て右子ノードか左子ノードにあるかを見ていけばいいです。left と right が右か左にあるかを見ていき、両者の間隔を狭めていって、left と right が同じノードになるまで続けます。

1. left を右に移動する

left が指すノードを使うか、それともその親ノードを使うかの判定を考えます。left と right の間は連続した区間であることに注意すると、left が親の左側にある場合は親ノードの値を使えばいい

ことがわかります。一方で、left が親の右側にある場合は、left が指すノードの値を使う必要があります。また 1-indexed なセグメント木では右ノードと左ノードは偶奇で判定できます。

2. right を左に移動する

right も同様に考えます。ただ、right は開区間になっているため right - 1 を使うか使わないかを考えます。right が親の左側にある時は right - 1 よりも左側の要素を使うことになりますが、それは right - 1 の親ノードが含んでいるため、right - 1 は使いません。一方、right が親の右側にある場合は right - 1 を使う必要があります。

4. 実装

以上の考察をコードにします。以下は区間の最大値を求めるセグメント木の実装例です。

コード 31　セグメント木の実装

```
# 区間の最大値を求めるセグメント木
class SegmentTree:
    def __init__(self, data_size: int, v: list[int] | None = None):
        self.data_size = data_size
        self.log = (data_size - 1).bit_length()
        self.capacity = 1 << self.log
        self.e = - (1 << 60)
        self.tree = [self.e] * (2 * self.capacity)

        # データが事前に与えられている場合
        if v is not None:
            # 葉ノードを埋める
            for i in range(self.data_size):
                self.tree[self.capacity + i] = v[i]

            # 葉ノード以外を埋める
            for i in range(self.capacity - 1, 0, -1):
                self.update(i)

    def update(self, index: int) -> None:
        self.tree[index] = max(self.tree[2 * index], self.tree[2 * index
            + 1])

    # 更新クエリ
    def set(self, index: int, value: int) -> None:
```

```
        # 葉に移動
        index += self.capacity
        # 更新
        self.tree[index] = value
        # 上のノードを更新 根(1)を更新するまで続ける
        while index > 1:
            index //= 2
            self._update(index)

    # 取得クエリ
    def prod(self, left: int, right: int) -> int:
        left_value = right_value = self.e

        # 葉に移動
        left += self.capacity
        right += self.capacity

        while left < right:
            # 範囲の左側が右のノード、本実装では奇数
            if left % 2 == 1:
                left_value = max(left_value, self.tree[left])
                left += 1

            if right & 2 == 1:
                right -= 1
                right_value = max(right_value, self.tree[right])

            # 親に移動
            left //= 2
            right //= 2

        return max(left_value, right_value)

    def all_prod(self):
        return self.tree[1]
```

XXIV. BIT

BIT(Binary Indexed Tree) はフェニック木とも呼ばれるデータ構造で、要素の値が変化する区間の**和**を高速に求めることができます。セグメント木でも区間和の計算ができますが、BIT はセグメント木よりもシンプルに実装が可能です。区間和に限定すればセグメント木は牛刀をもって鶏を割くようなものです。セグメント木はより一般的な演算に対応したデータ構造です。

BIT はセグメント木に比べてシンプルです。

i. 和

1 番目から i 番目の要素の和を高速に求めるために、以下のような操作を行います。

1. i を 2 進数表記にしたときに一番右にある 1 の要素を足していく
2. 1 の要素を足していくときに、その要素を取り除いて次の 1 の要素を足していく

例えば、$i = 7$ の場合を考えると、$7 = 111(2)$ となり、右から 1 を引いていくと、$111(2), 110(2), 100(2), 000(2)$、つまり $7, 6, 4, 0$ となります。

BIT のデータを持つ配列を bit とすると、1 から 7 番目の要素の和は、$\mathrm{bit}[0] = 0$ になっているので、

$$\mathrm{bit}[7] + \mathrm{bit}[6] + \mathrm{bit}[4] + \mathrm{bit}[0]$$

になります。

ii. 更新

要素の更新は和の計算と逆の操作を行います。具体的には、以下の操作を行います。

1. i を 2 進数表記にしたときに一番右にある 1 の要素を引いていく
2. 1 の要素を引いていくときに、その要素を取り除いて次の 1 の要素を引いていく

例えば、$A = [1, 2, 3, 4, 5, 6, 7, 8]$ という等差数列の 5 番目に 6 を足すと、$A = [1, 2, 3, 4, 11, 6, 7, 8]$ になります。このような値の更新を BIT で行ってみましょう。

まず、5 を 2 進数で表すと、101_2 となります。ここで、一番右にある 1 の桁に 1 を足す操作を順に行っていきます。$N = 8$ を超える前に操作をやめます。

具体的には、$101_2 \to 110_2 \to 1000_2$ となります。これを 10 進数に直すと、$5 \to 6 \to 8$ となります。ここで出てきた数をインデックスとして、6 を足していくことで値を更新することができます。

実際の操作は以下のようになります。

```
BIT[5] += 6
BIT[6] += 6
BIT[8] += 6
```

iii.　実装

BIT の実装例は以下の通りです。

コード 32　BIT の実装

```
class BIT:
  def __init__(self, size: int) -> None:
      self.size = size
      self.tree = [0] * (size + 1)

  def sum(self, index: int) -> int:
      res = 0
      while index > 0:
          res += self.tree[index]
          index -= index & (~index + 1)

      return res

  def update(self, index: int, value: int) -> None:
      while index < self.size:
          self.tree[index] += value
          index += index & (~index + 1)
```

XXV.　グラフの用語整理

グラフはノードとエッジからなるデータ構造です。グラフの用語を整理します。

i.　ツリー (木)

ツリーは閉路を持たない連結なグラフです。ツリーは以下の性質を持ちます。

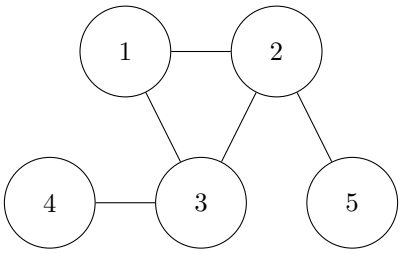
- 連結なグラフである
- 閉路を持たない

ノードの数が n であるグラフ G が木であることは、以下の条件とも同値です。

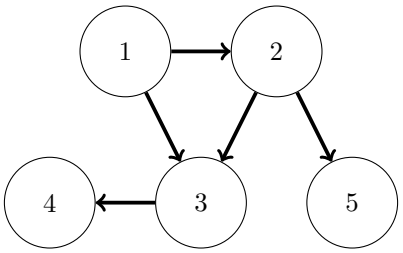
- G には閉路がなく、$n-1$ 本のエッジを持つ
- G は連結であり、$n-1$ 本のエッジを持つ
- G の任意の 2 点を結ぶ経路はただ 1 つ存在する

ii.　無向グラフと有向グラフ

無向グラフはエッジに向きがないグラフです。有向グラフはエッジに向きがあるグラフです。



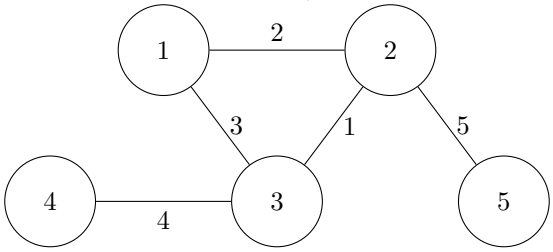
無向グラフ



有向グラフ

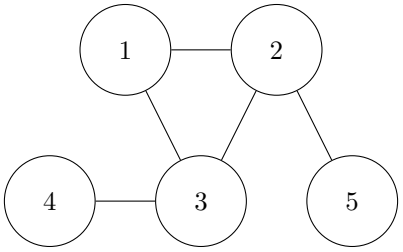
iii.　重み付きグラフ

重み付きグラフはエッジに重みがついたグラフです。重みはエッジのコストや距離を表します。



iv.　隣接行列と隣接リスト

隣接行列と隣接リストはグラフを表現するためのデータ構造です。以下のグラフを例にして、隣接行列と隣接リストを示します。



隣接行列

隣接行列はグラフのエッジを行列で表現したものです。(i, j) 成分が 1 のとき、ノード i とノード j がエッジで結ばれていることを表します。下の図では隣接行列は 1-indexed で表現しています。

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

隣接リスト

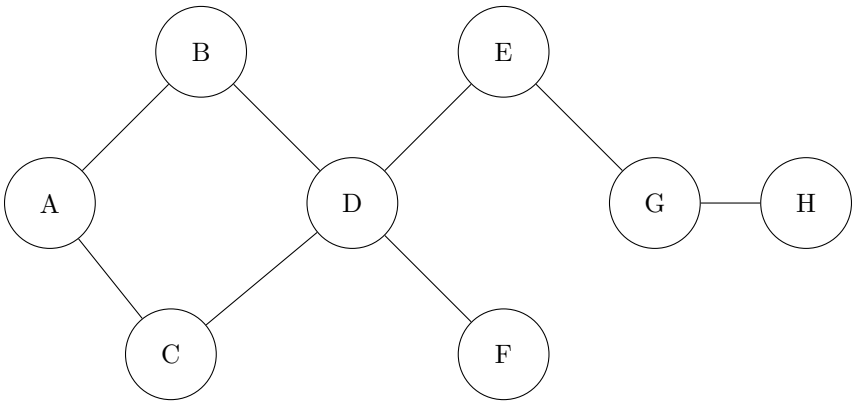
隣接リストは各ノードに隣接するノードをリストで表現したものです。

- 1: 2, 3
- 2: 1, 3, 5
- 3: 1, 2, 4
- 4: 3
- 5: 2

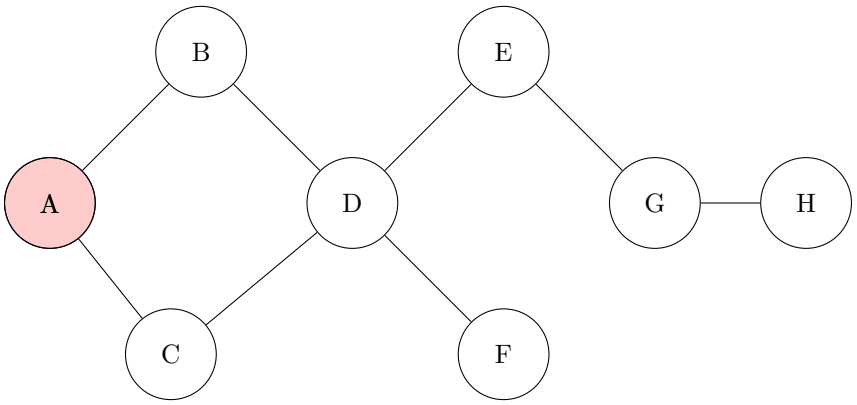
グラフの基本として、深さ優先探索 (DFS) と幅優先探索 (BFS) というグラフの探索アルゴリズムを扱います。

XXVI.　幅優先探索 (BFS)

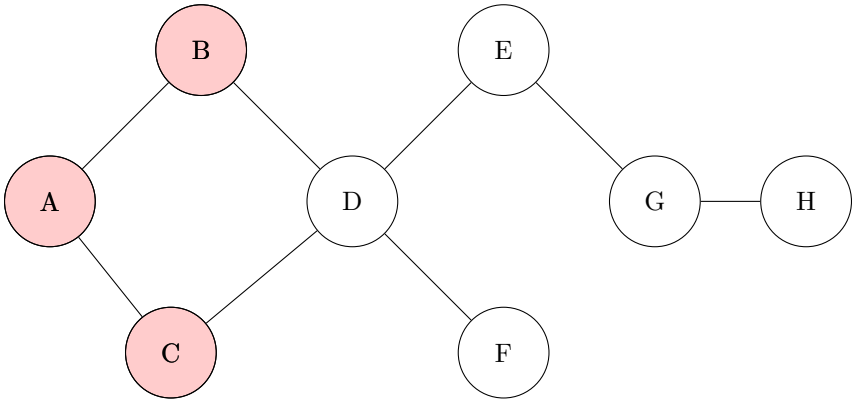
BFS は、後戻りしないように、可能性のあるルートすべてにおいて 1 ステップずつ行くアルゴリズムです。BFS の例を見てみましょう。



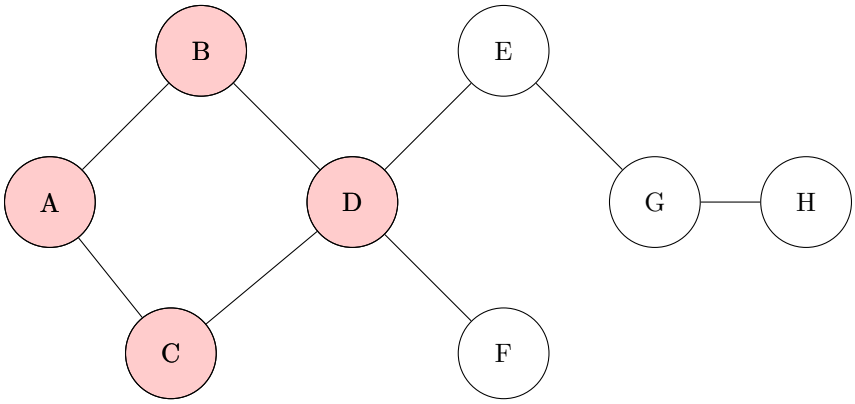
A からスタートします。



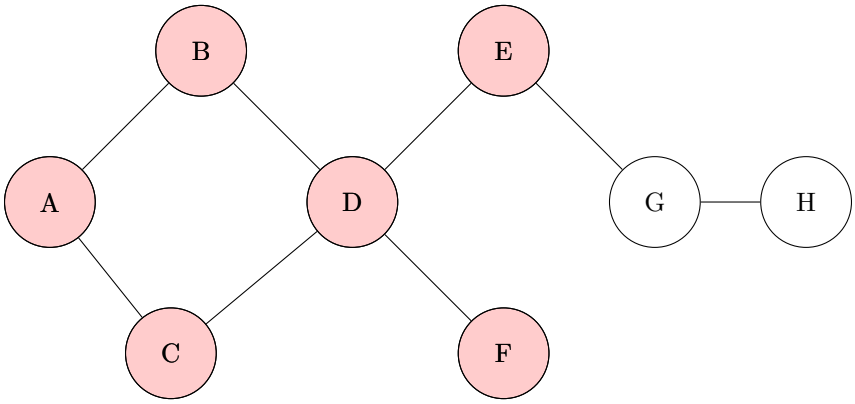
次に A と繋がっているノード B と C を探索します。



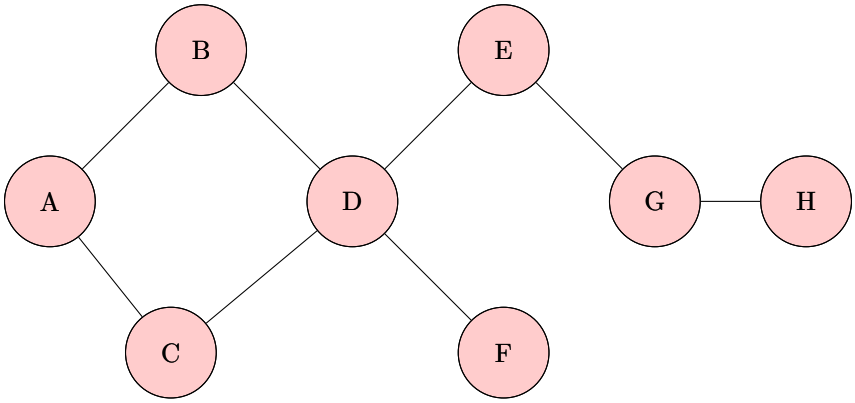
A の探索が終わったので、次に B と C の探索を行います。今回は B から探索します。B と C は同じ深さにあるので、どちらから探索しても問題ありません。B には D が繋がっているので、D を探索します。C から探索を始めようとすると、すでに D はすでに探索済みなので、探索を行いません。



次に D から探索を行います。D には E と F が繋がっているので、E と F を探索します。



最後に G と H を探索します。



これでグラフの探索が終了しました。BFS はスタート地点からの最短距離を求めることができます。

i.　BFS の実装

BFS の実装はキューを用いて行います。実装のポイントは以下の通りです。

- キューを用いて、次に探索するノードを管理する
- 探索済みのノードを管理するために、配列を用いる

隣接リストでも隣接行列でも実装できますが、隣接リストの方が実装が簡単です。また 0-indexed で実装していることに注意してください。

コード 33　深さ優先探索ヒープの実装

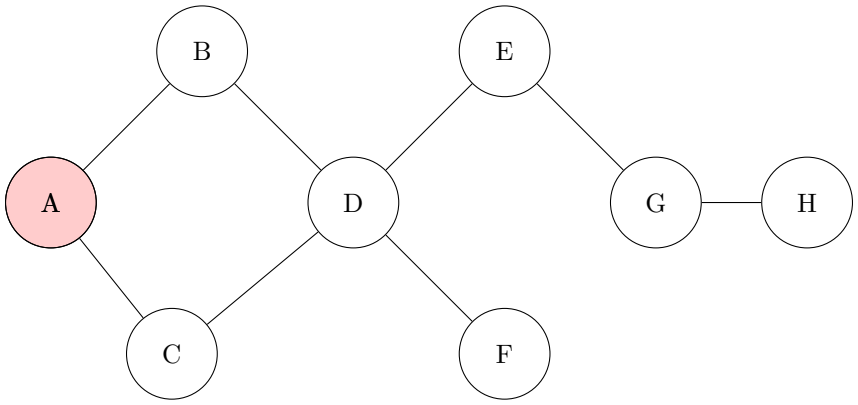
```
from collections import deque

  def bfs(graph: list[list[int]], start: int) -> list[bool]:
      visited = [False] * len(graph)
      todo = deque()

      # スタート地点で初期化
      todo.append(start)

      while todo:
          node = todo.popleft()
          visited[node] = True

          for next_node in graph[node]:
              if not visited[next_node]:
```

```
16                     todo.append(next_node)
17
18         return visited
```

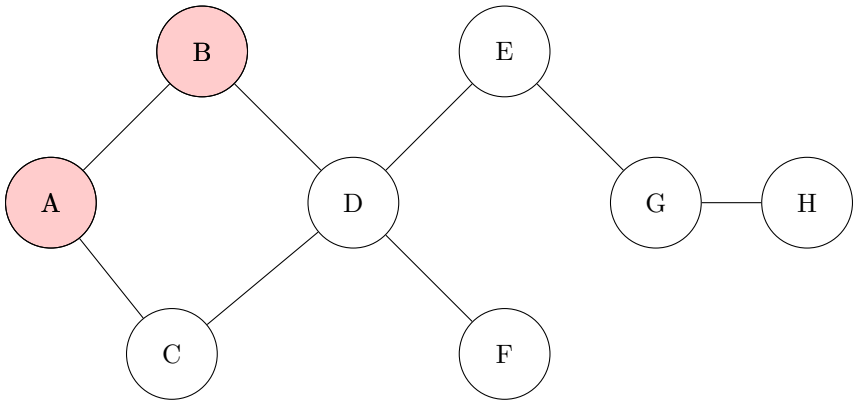
XXVII.　深さ優先探索 (DFS)

DFS は、スタート地点から次のノードに進み、進んだノードに繋がっているノードを行けなくなるまで探索するアルゴリズムです。先ほどのグラフを例にして、DFS の探索を行います。

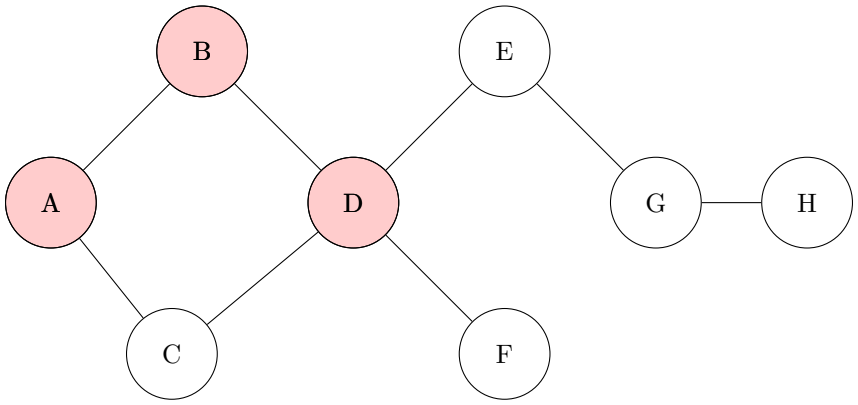
最初は A から探索を行います。



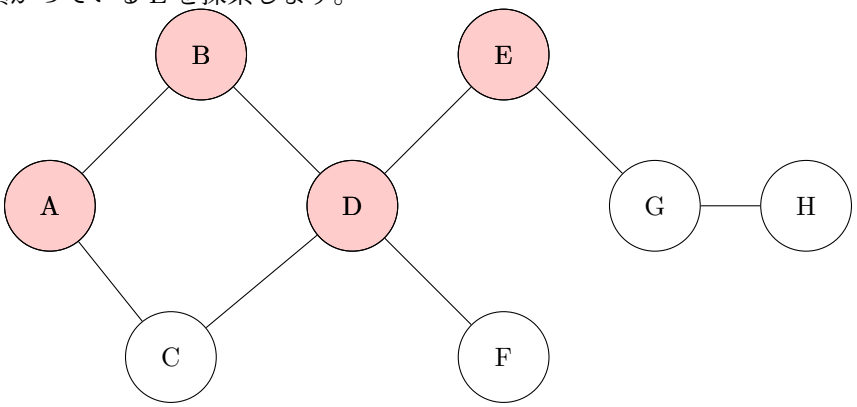
次に A と繋がっているノード B を探索します。



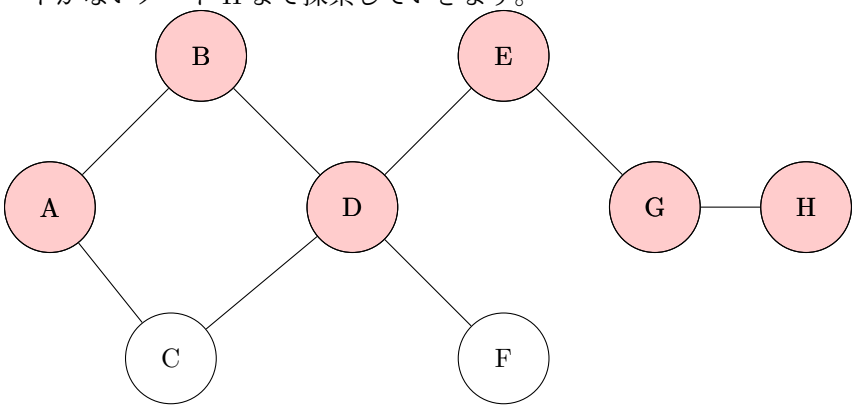
BFS では C を次に探索しますが、DFS では B に繋がっている D を探索します。



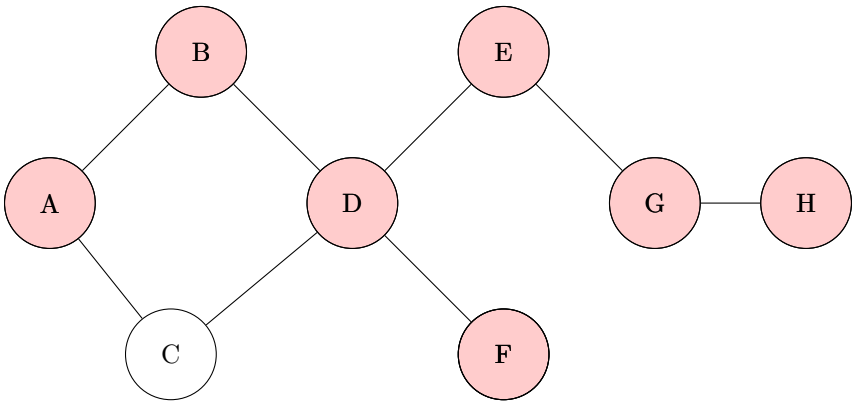
次に D に繋がっている E を探索します。



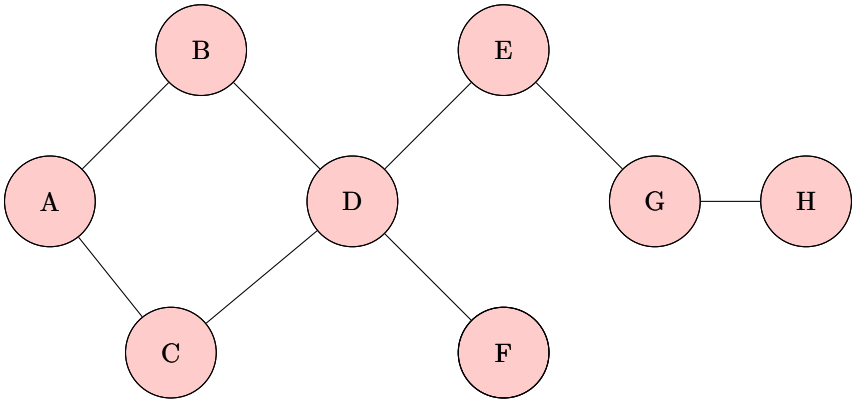
これ以上ノードがないノード H まで探索していきます。



移動する余地の残っている F を探索します。



最後にまだ探索できる A に繋がっている C の探索を行います。



これでグラフの探索が終了しました。DFS は猪突猛進な探索方法で、BFS とは異なり、最短経路を求めることができません。

i.　DFS の実装 (スタック)

DFS の実装はスタックを用いて行います。実装のポイントは以下の通りです。

- スタックを用いて、次に探索するノードを管理する
- 探索済みのノードを管理するために、配列を用いる

隣接リストでも隣接行列でも実装できますが、隣接リストの方が実装が簡単です。また 0-indexed で実装していることに注意してください。DFS との違いは、キューをスタックに変えるだけです。

コード 34　深さ優先探索ヒープの実装

```
from collections import deque

def dfs(graph: list[list[int]], start: int) -> list[bool]:
    visited = [False] * len(graph)
    todo = deque()

    # スタート地点で初期化
    todo.append(start)

    while todo:
        node = todo.pop()
        visited[node] = True

        for next_node in graph[node]:
```

```
15                 if not visited[next_node]:
16                     todo.append(next_node)
17
18         return visited
```

ii.　DFS の実装 (再帰)

DFS は再帰を用いて実装することもできます。再帰を用いると、スタックを用いた実装よりも簡潔に実装することができます。

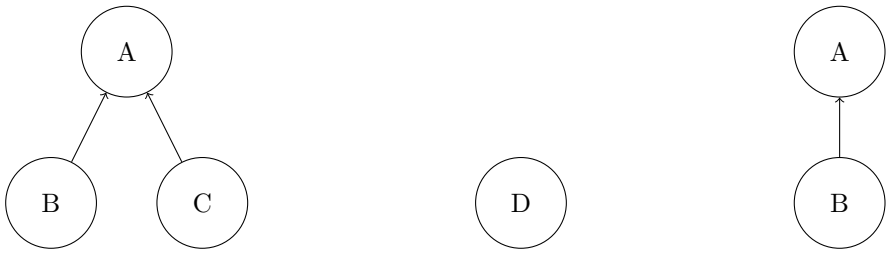
コード 35　深さ優先探索再帰の実装

```
1  def dfs(graph: list[list[int]], start: int, visited: list[bool]) -> list[
       bool]:
2      visited[start] = True
3      for next_node in graph[start]:
4          if visited[next_node]:
5              continue
6          else:
7              dfs(graph, next_node, visited)
8
9      return visited
```

XXVIII.　素集合データ構造 (Union-Find 木)

Union-Find 木はノードの集合の連結性を管理するデータ構造です。下の例では、A と D が同じノードにあるかを高速に判定したり、逆に A と D を連結したりする操作を行うことができます。Union-Find 木は以下の操作を行います。

- Union 2 つの集合を結合する
- Find: 2 つのノードが同じ集合に属しているかを判定する



コード 36　Union-Find 木の実装

```python
class UnionFind:
    def __init__(self, n: int) -> None:
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def _root(self, node: int) -> int:
        if self.parent[node] == node:
            return node
        else:
            # 経路圧縮
            self.parent[node] = self._root(node)
            return self.parent[node]

    def unite(self, x: int, y: int) -> None:
        root_x = self._root(x)
        root_y = self._root(y)

        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
```

```
21            else:
22                self.parent[root_y] = root_x
23                if self.parent[root_x] == self.parent[root_y]:
24                    self.rank[root_x] += 1
25    def is_same(self, x: int, y: int) -> bool:
26        return self.parent[x] == self.parent[y]
```

XXIX.　最短経路問題

BFS を用いた最短経路は上で紹介しましたが、今回はより効率的な最短経路問題の解法を紹介します。DFS では重さが同じのグラフでしか最短経路を求めることができませんが、ダイクストラ法を用いることで重さに異なる重み付きグラフでも最短経路を求めることができます。また、負の重みがあっても最短経路を求めることができるベルマン・フォード法も紹介します。

i.　ダイクストラ法

ダイクストラ法を理解する上で重要な重み付きグラフの性質を紹介します。

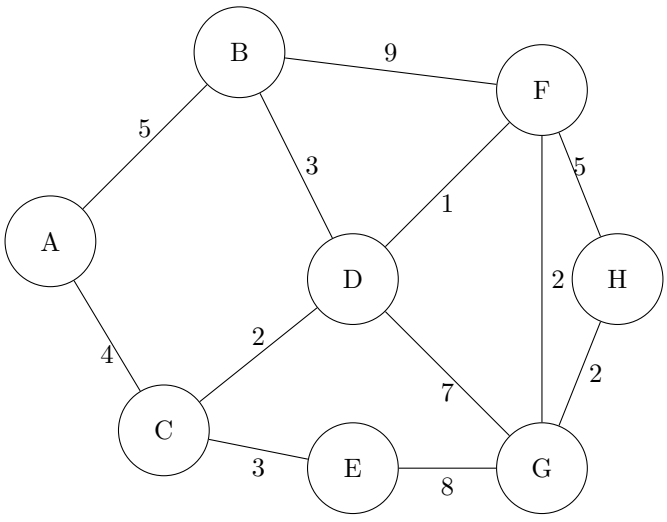
経路緩和性

最短経路の部分経路も最短経路である

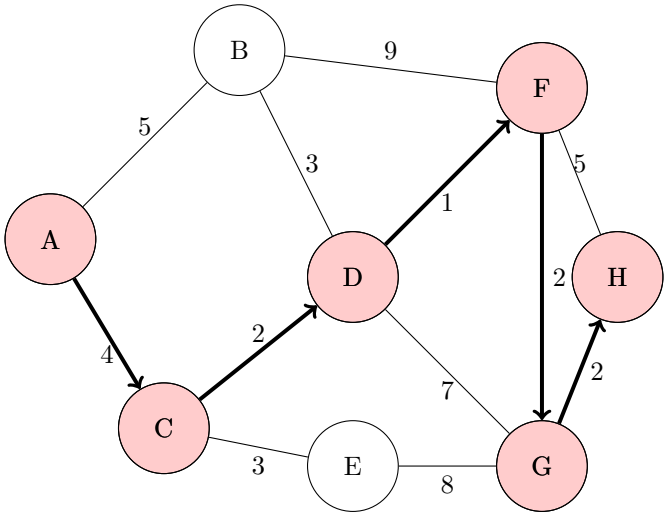
簡単な証明

P を最短経路とし、その部分経路を Q とする。もしも Q よりも短い経路 R が存在するとすると、R を使った経路の方が P よりも短い経路になるため、P は最短経路ではない。P が最短経路であるという過程に矛盾が生じるため、Q も最短経路である。

具体例を挙げて説明します。以下のグラフを考えます。



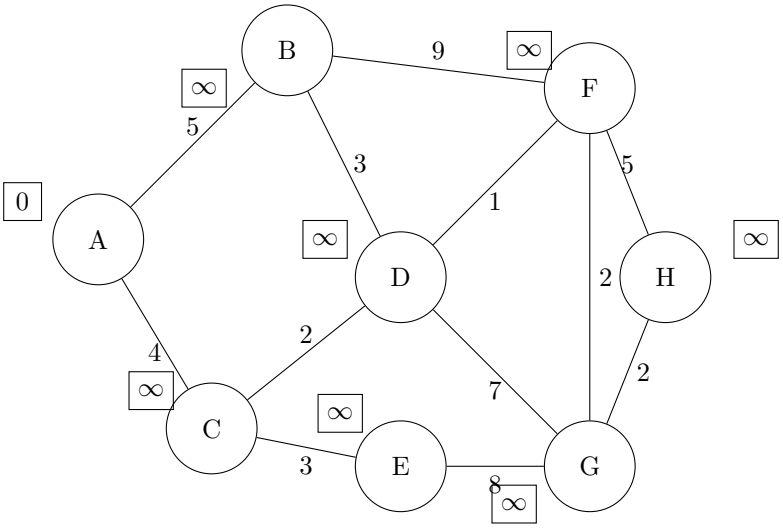
A から H までの最短経路は以下の通りです。もしもゴールが G、F、D、C いずれの場合でも、最短経路は A → C → D → F → G → H の経路になります。



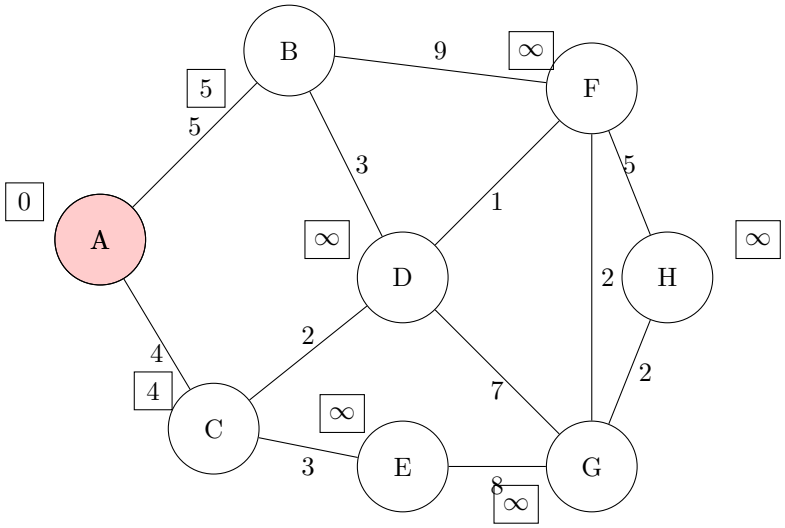
以上の性質より、あるノードまでの最短経路を考えるときはそのノードの前のノードまでの最短経路を考えればよいことがわかります。この性質を利用したのアルゴリズムがダイクストラ法です。ダイクストラ法は以下の手順で最短経路を求めることができます。

1. まだ距離が確定していないノードのうち、最も距離が短いノード x を選択する
2. ノード x に繋がっているノードの距離を更新する
3. 更新が終わるとノード x の距離を確定する
4. すべての頂点が確定するまで 1 から 3 を繰り返す

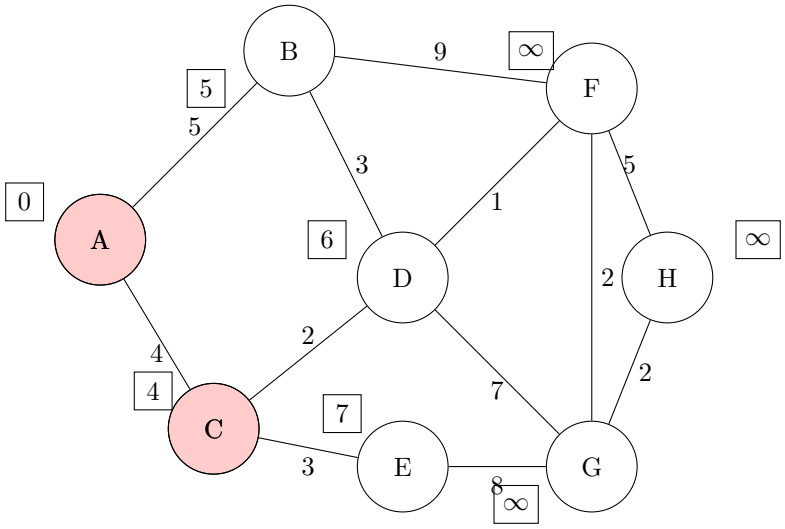
距離は ∞、スタート地点は 0 で初期化します。上のグラフを例にしてダイクストラ法の例を見てみます。最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。



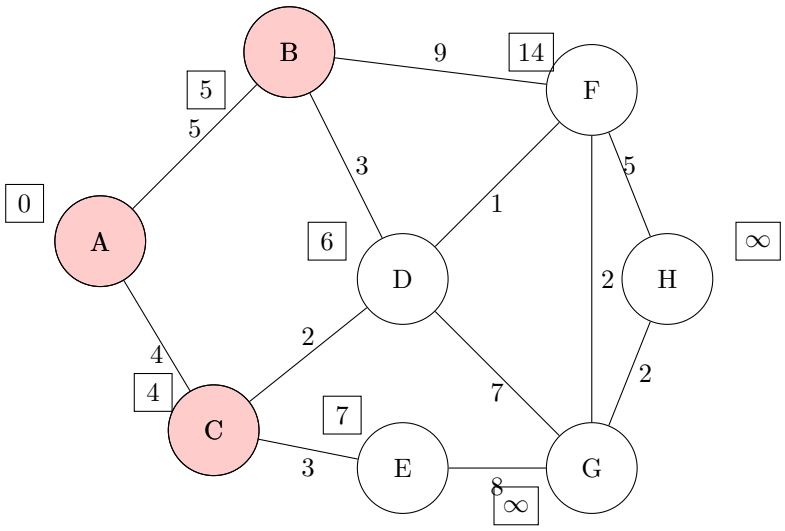
最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。更新が終わったので、A の距離を確定します。



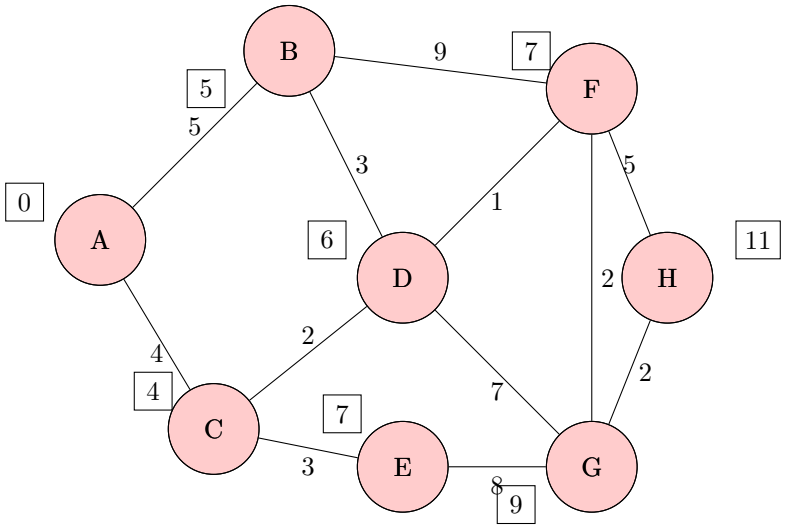
次に距離が確定していないノードで最も距離が短いノードを選択します。今回は C です。C に繋がっているノードの距離を更新します。C に繋がっているノードを更新したら、C の距離を確定します。



次に距離が確定していないノードで最も距離が短いノードを選択します。今回は B です。B に繋がっているノードの距離を更新します。D に関しては、すでにわかっている経路の方が短いので更新しません。B の距離を確定します。



以下同様に更新すると、最終的に以下のような結果になります。



1. ダイクストラ法の実装

ダイクストラ法はとてもシンプルなアルゴリズムです。ダイクストラ法の実装のポイントは以下の通りです。

- 最短距離を格納する配列を用意する
- まだ確定していないノードのうち、最も距離が短いノードを選択する。最も短いノードを $O(\log n)$ で取得するためにヒープを使う
- 選択したノードに繋がっているノードの距離を前回の距離と比較して更新する
- すべてのノードが確定するまで繰り返す

Python の標準ライブラリのヒープは tuple などを渡す index が早い要素から比較してくれるので、ダイクストラ法の実装に適しています。(移動距離、ノード) の tuple でヒープに追加することで、最短距離が短いノードを取得することができます。もちろん自分で実装したヒープを使っても問題ありません。

与えられる重み付きグラフは (終点、重み) の形式で隣接リストで与えられるとします。以下にダイクストラ法の実装を示します。

コード 37　ダイクストラ法の実装

```python
from heapq import heappop, heappush

def dijkstra(graph: list[list[int]], start: int) -> list[int]:
    done = [False] * len(graph)
    dist = [1 << 60] * len(graph)
    todo = []

    # 初期化
    dist[start] = 0
    heappush(todo, (dist[start], 0))

    while todo:
        distance, node = heappop(todo)
        if done[node]:
            continue

        # 更新
        for connected_node, weight in graph[node]:
            if distance + weight < dist[connected_node]:
                dist[connected_node] = distance + weight
                heappush(todo, (dist[connected_node], connected_node))

        done[node] = True

    return dist
```

ii.　ベルマン・フォード法

ベルマン・フォード法はダイクストラ法と異なり、負の重みを持つ辺があっても機能する単一始点全点間最短路を求めるアルゴリズムです。負の閉路の検出も可能です。ダイクストラ法とは違って最短距離の選択を行わずに、毎回すべての辺を更新します。

ベルマン・フォード法のアルゴリズムを例を使って説明します。以下のグラフを例に考えます。概要はダイクストラ法とあまり変わりません。ただし、グラフの情報が (始点、終点、重み) の list で与えられるとします。例えば、(A, B, 5) は A から B へでている重みが 5 のノードであることを示します。

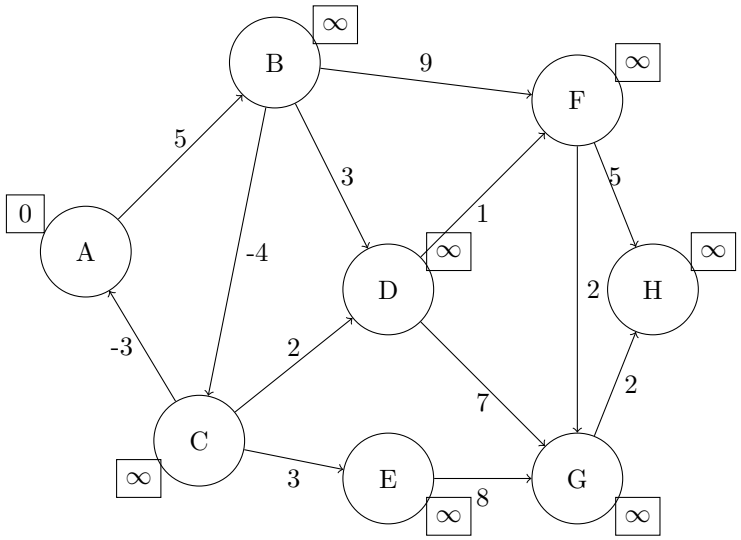
すべて列挙すると、以下の様になります。

$$\begin{aligned}\text{edges} =&(A,B,5),(B,C,-4),(C,A,-3),(C,D,2),(B,D,3),(B,F,9),(F,H,5),\\&(C,E,3),(E,G,8),(D,G,7),(D,F,1),(G,H,2),(F,G,2)\end{aligned}$$

初期化として始点の距離は 0, それ以外は ∞ とします。ベルマン・フォード法は以下のアルゴリズムに従っています。

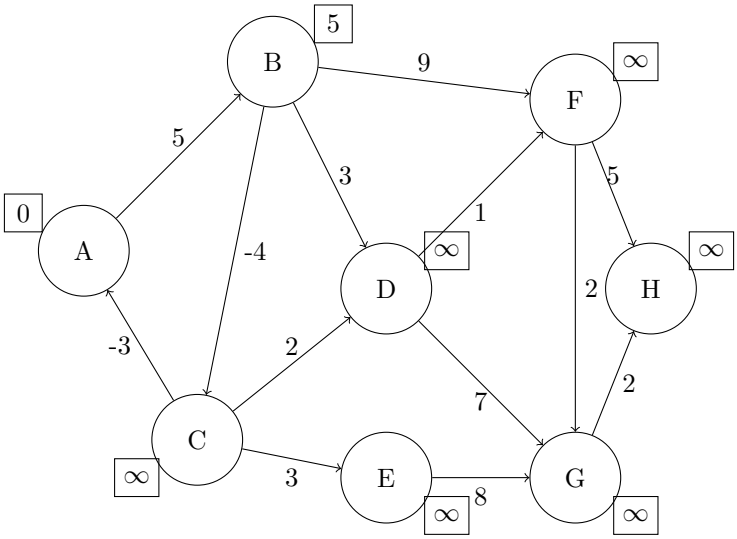
1. edges をすべて列挙し、始点から終点への重みを更新する
2. 1 を V - 1 回繰り返す (V はノードの数)

V-1 回の更新で、最短距離が確定します。もし、V 回目にも更新がある場合は負の閉路が存在することになります。最短経路を求めるのに V - 1 回の更新で十分な理由は、経路緩和性によって開始地点から終点が最も遠い場合でも、V - 1 回の更新で確定するからです。

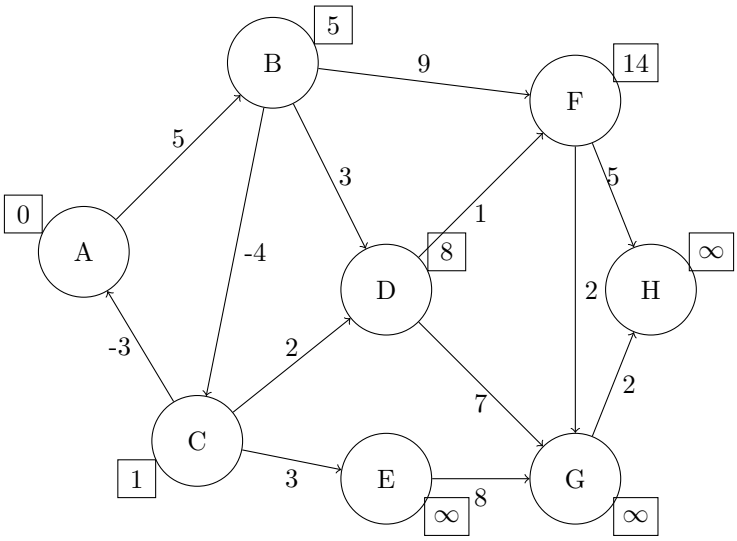


ベルマン・フォード法の流れを確認します。最初は A と繋がっている B が更新されます。他のノードも edges をすべて列挙して更新を図りますが、∞+ 有限の値と考えるため、更新されないと

みなすことができます。



2 回目の更新です。



以下のように更新を繰り返すと、最終的に以下のような結果になります。

コード 38　ベルマン・フォード法の実装

```
def bellman_ford(v: int, edges: list[tuple[int, int, int]]) -> list[int]
    | int:
  # 初期化
  dist = [1 << 60] * v
  dist[0] = 0
```

```
5
6   for _ in range(v - 1):
7       for start, end, weight in edges:
8           if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
9               dist[end] = dist[start] + weight
10
11  # 一度v-1回更新した後に負の閉路を検出
12  for start, end, weight in edges:
13      if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
14          return -1
15
16  return dist
```

iii.　SPFA(Shortest Path Faster Algorithm)

SPFA はベルマン・フォード法を高速化したアルゴリズムです。基本はベルマン・フォード法と同じですが、毎回すべての辺をチェックすること防ぐ工夫がされています。あるノード x が更新されなければ、そのノードに繋がっている他のノードの距離も更新されません。実装上は更新が必要なノードが出てきたらそれを queue に入れ、queue がからになるまで処理を続けます。

SPFA 実装の実装は以下の様になります。

コード 39　SPFA の実装

```python
from collections import deque

def spfa(v: int, edges: list[list[tuple[int, int]]]):
    inf = 1 << 60
    dist = [inf] * v
    dist[0] = 0

    node_to_check = deque()
    in_queue = [False] * v

    while node_to_check:
        current_node = node_to_check.popleft()
        in_queue[current_node] = False

        for end, weight in edges[current_node]:
            if dist[current_node] + weight < dist[end]:
                dist[end] = dist[current_node] + weight

            if not in_queue[end]:
                in_queue[end] = True
                node_to_check.append(end)

    return dist
```

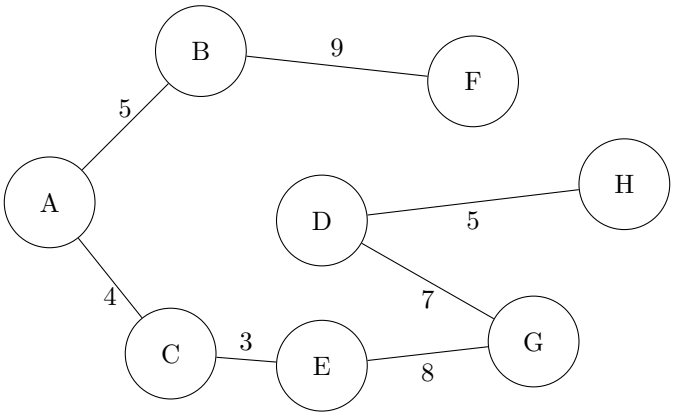
iv.　ワーシャルフロイド法

コード 40　ワーシャルフロイド法の実装

```
def warshall_floyd(n: int, dist: list[list[int]]):
  for i in range(n):
      for j in range(n):
          for k in range(n):
              dist[j][k] = min(dist[j][k], dist[j][i] + dist[i][k])
```

XXX.　最小全域木

全域木とは、すべてのノードが繋がっている木のことをいいます。また**最小全域木**とは、全域木の中で重さが最小になるもののことをいいます。

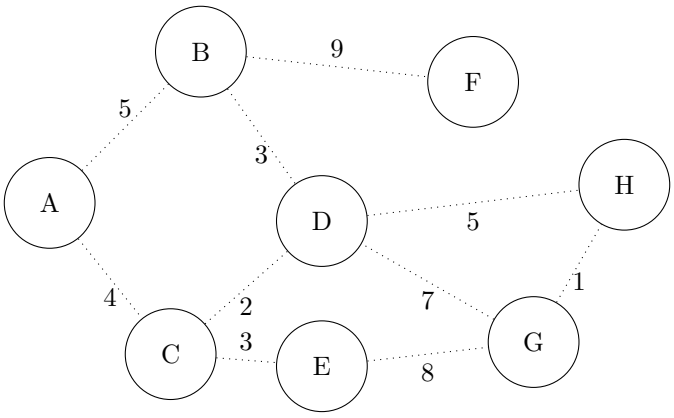


最小全域木を求めるには、辺ベースのアプローチとノードベースのアプローチの 2 種類があります。それぞれ**クラスカル法**、**プリム法**と呼ばれています。

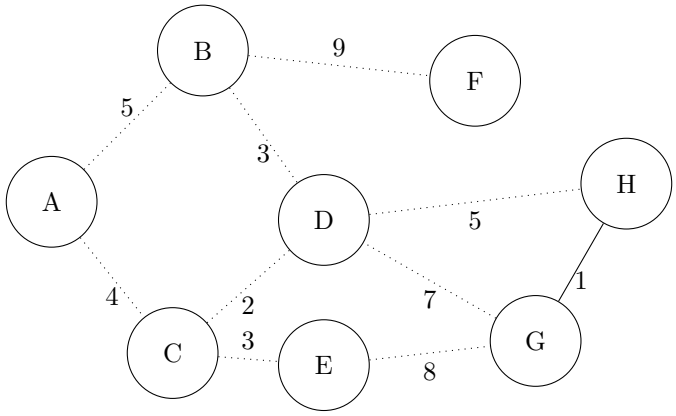
i.　クラスカル法

クラスカル法は、存在する辺を重さが小さい順に並べて入れていき、閉路ができないことが確認できた場合は追加し、すべての辺をチェックし終えたら終了するアルゴリズムです。以下のアルゴリズムに従っています。

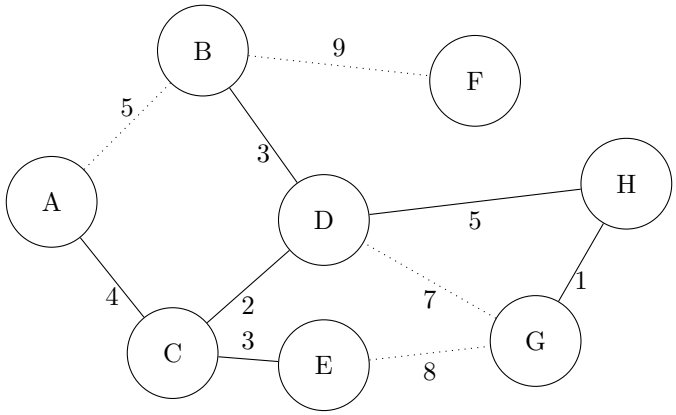
1. すべての辺を重さの小さい順にソート
2. 重さの最も小さい辺を選ぶ
3. 今までに選んだ辺から構成される木に 2 で選んだ辺を追加した時に、閉路が生まれないことを確認する。閉路が新しくできないならこの辺を追加する
4. すべての辺をチェックし終えるまで 2 から 3 を繰り返す



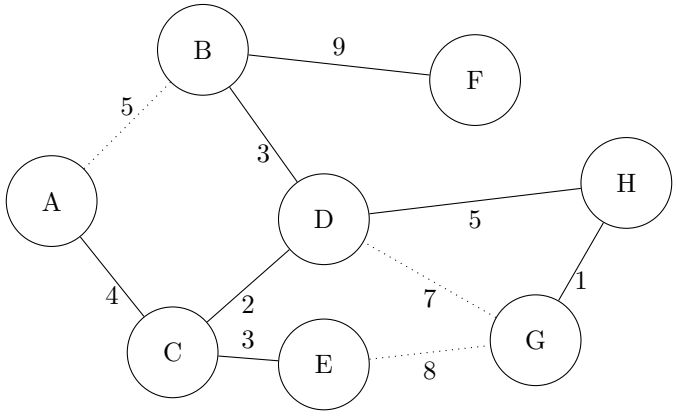
クラスカル法の例を見てみましょう。まずは、すべての辺の中で最も重さが小さい辺を選んでそれを木に追加したときに、閉路ができないのでその辺を追加します。



同様に、CD、CE、BD、AC、DH を順に追加しても閉路はできないので、追加します。



しかし、AB、DG、EG を繋げると閉路ができてしまうので追加しません。最後に BF を追加するとすべての辺をチェックし終えたので以下のような最小全域木ができあがります。



クラスカル法の実装上のポイントは以下の 2 つです。

- すべての辺を重さの小さい順にソートする
- 辺を追加したときに閉路かどうかを判定する

最初のポイントは、辺を重さの小さい順にソートすることで、簡単に実装できます。2 つ目のポイントは、BFS や DFS を使っても実装できますが、効率が良くありません。そこで、Union-Find 木を使って実装することが一般的です。

ある辺をグラフに木に挿入しようとしたとき、辺を作る 2 点が同じ集合に属している場合に辺を繋げると閉路ができてしまいます。つまり、Union-Find を使ってふたつのノードが同じ集合に属しているかを判断します。

コード 41　クラスカル法の実装

```python
class UnionFind:
    def __init__(self, n: int) -> None:
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def _root(self, node: int) -> int:
        if self.parent[node] == node:
            return node
        else:
            # 経路圧縮
            self.parent[node] = self._root(node)
            return self.parent[node]

    def unite(self, x: int, y: int) -> None:
        root_x = self._root(x)
        root_y = self._root(y)

        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                if self.parent[root_x] == self.parent[root_y]:
                    self.rank[root_x] += 1
    def is_same(self, x: int, y: int) -> bool:
```

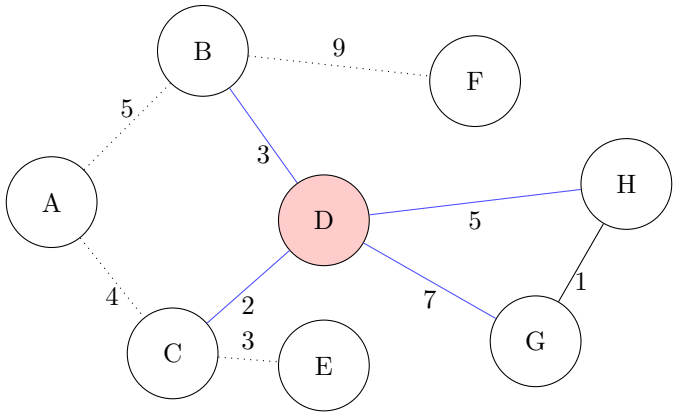
```
26          return self.parent[x] == self.parent[y]
27
28  def kruskal(v: int, edges: list[tuple[int, int, int]]) -> list[list[int,
        int]]:
29      """
30      args:
31          v: node size
32          edges: (start, end, weight)
33      """
34      sorted_edge_costs = []
35      for edge in edges:
36          sorted_edge_costs.append([edges[2], edges[0], edges[1]])
37
38      sorted_edge_costs.sort()
39
40      uf_tree = UnionFind(v)
41
42      minimum_spanning_tree = []
43
44      for weight, start, end in sorted_edge_costs:
45          if not uf_tree.is_same(start, end):
46              uf_tree.unite(start, end)
47              minimum_spanning_tree.append([start, end])
48
49      return minimum_spanning_tree
```

ii.　プリム法

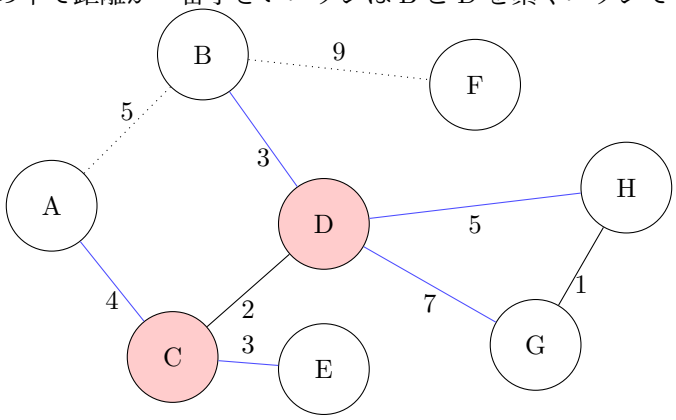
プリム法とは、すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、すべてのノードがつながったら終了するアルゴリズムです。プリム法は以下のアルゴリズムに従っています。

1. 任意のノードを選び、訪問済みにする
2. そのノードに繋がっているすべての辺を最小全域木の候補の辺として追加する
3. 最小全域木の候補の辺の中から、接続先のノードが未訪問である最短の距離の辺を選ぶ
4. 選んだ辺を最小全域木に入れ、その接続先のノードを訪問済にする。
5. 4 で新しく訪問したノードから、さらにその先に繋がっている辺のうち、接続先のノードが未訪問のすべての辺を最小全域木の候補に追加する
6. すべてのノードが訪問済になるまで 2 から 4 を繰り返す

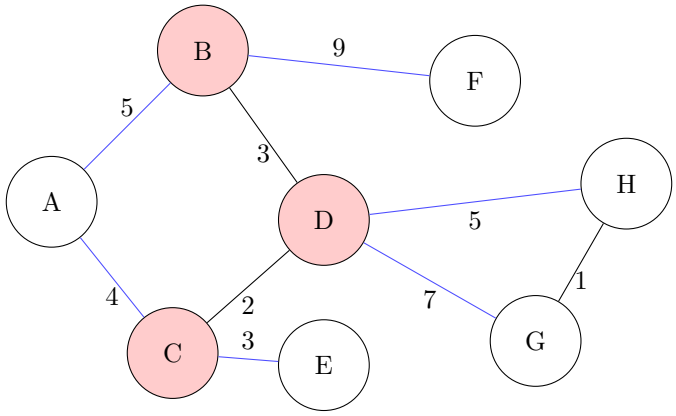
プリム法の具体例を見てみましょう。まずは、任意にノードを選び、訪問済にします。ここでは D を選びます。D から繋がっているエッジを青色で示します。その中で最も距離が短いエッジは C と D を繋ぐエッジです。



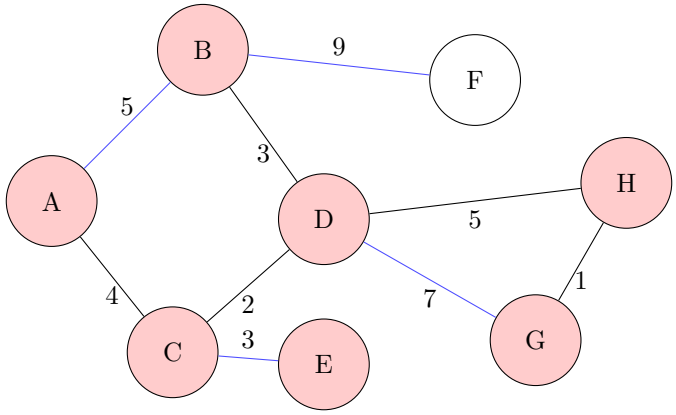
C は未訪問なので、C を訪問済にして、C から繋がっているエッジを最小全域木の候補に追加します。青色のエッジの中で距離が一番小さいエッジは B と D を繋ぐエッジです。



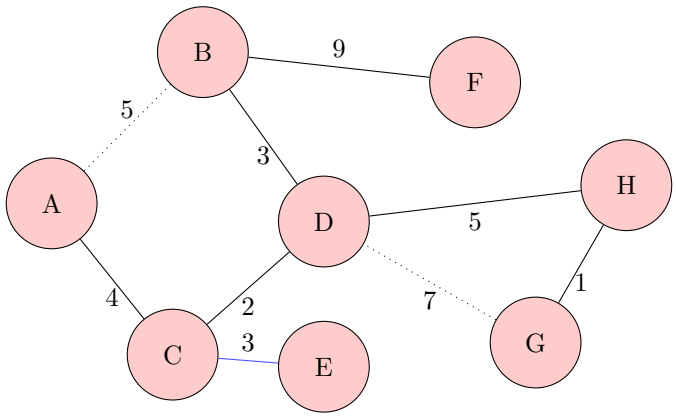
B は未訪問なので、B を訪問済にして、B から繋がっているエッジを最小全域木の候補に追加します。青色のエッジの中で最も短いエッジは C と E を繋ぐエッジです。



同様にして、A と C を結ぶエッジ、D と H を結ぶエッジ、G と H を結ぶエッジを追加します。



残りは、A と B を結ぶエッジ、D と G を結ぶエッジがありますが、A と B を結ぶエッジと D と G を結ぶエッジを追加すると閉路ができてしまうので追加しません。最後に D と G を結ぶエッジを追加すると、すべてのノードがつながったので終了です。以下のような最小全域木ができあがります。



プリム法の実装例を以下に示します。

コード 42　プリム法の実装

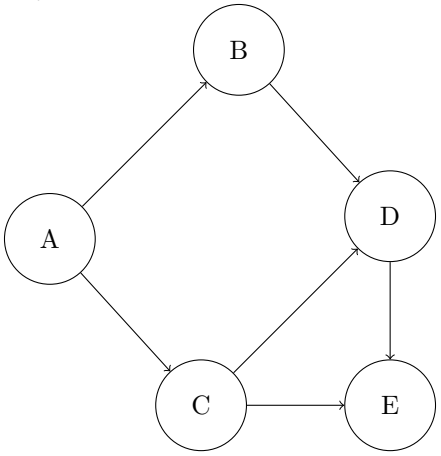
```
import heapq

def prim(v: int, edges: list[list[int, int, int]]):
    edges_from = [[] for _ in range(v)]

    for start, end, weight in edges:
        edges_from[start].append([weight, start, end])

    edge_heap = []
    minimum_spanning_tree = []
    included = [False] * v

    # nodeをひとつ選ぶ この実装では0を選ぶ
    included[0] = True
    # node0につながる辺をすべてヒープに入れる
    for edge in edges_from[0]:
        heapq.heappush(edge_heap, edge)

    while edge_heap:
        weight, start, end = heapq.heappop(edge_heap)
        if not included[end]:
            included[end] = True
            minimum_spanning_tree.append([start, end])

```

```
25              for edge in edges_from[end]:
26                  if not included[end]:
27                      heapq.heappush(edge_heap, edge)
28
29      return minimum_spanning_tree
```

XXXI.　トポロジカルソート

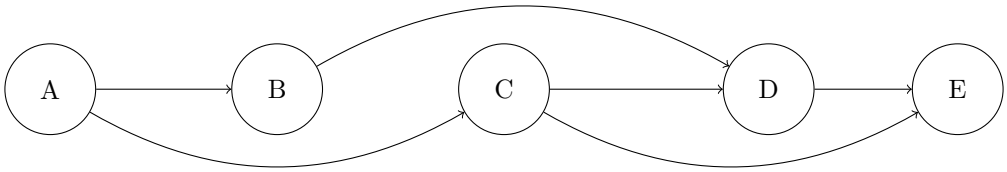
DAG という性質を持ったグラフをソートするアルゴリズムであるトポロジカルソートを扱います。**DAG**(Directed Acyclic Graph) とは、その名の通り有向グラフで閉路を持たないグラフのことを指します。

以下に DAG の例を示します。有向グラフで閉路がないため、どこのノードから辿っても元の位置に戻ってくることはできません。



DAG の例

トポロジカルソートはすべての辺が同じ方向を向くようにノードをソートするアルゴリズムです。上の DAG をトポロジカルソートをすると、以下のようになります。有向辺がすべて右向きになっていることがわかります。注意点として、トポロジカルソートの結果は一意ではありません。



有向グラフの性質をより理解するために**次数、入次数、出次数**という用語を導入します。次数とは、ノードにつながっている辺の数を指します。入次数とは、ノードに入ってくる辺の数を指し、出次数とは、ノードから出ていく辺の数を指します。次数 = 入次数 + 出次数 です。DAG の性質として、必ず入次数 0 のノードが存在します。

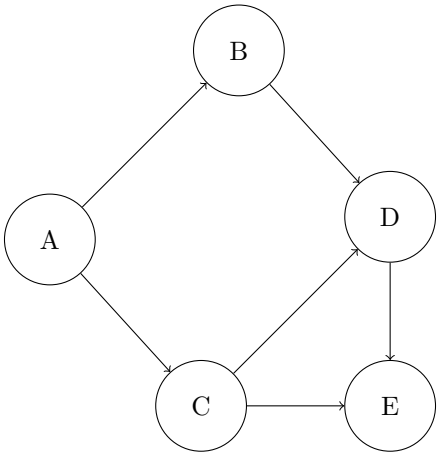
トポロジカルソートの代表的なアルゴリズムとして、Kahn のアルゴリズムと Tarjan のアルゴリズムを紹介します。

i. Kahn のアルゴリズム

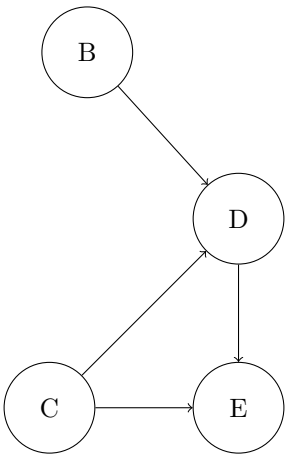
Kahn のアルゴリズムは、トポロジカルソートを行うアルゴリズムの一つです。アルゴリズムの流れは以下の通りです。

1. 入次数が 0 のノードをグラフから取り除き、ソート済配列に追加する
2. 1 を入次数 0 のノードがなくなるまで繰り返す

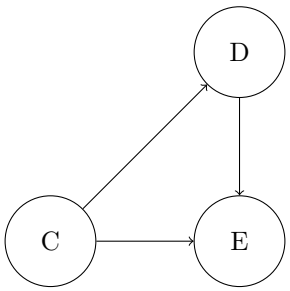
Kahn のアルゴリズムの例を以下に示します。



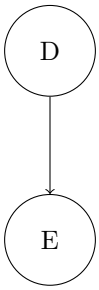
入次数 0 のノード A を取り除き、A に繋がっているノードの次数を更新します。



入次数 0 のノード B を取り除きます。



入次数 0 のノード C を取り除きます。最後にノード D と E を順に取り出せばトポロジカルソートは完成です。



Kahn のアルゴリズムの実装例を以下に示します。

コード 43　Kahn のアルゴリズムの実装

```
from collections import deque

def kahn_topological_sort(v: int, e: int, edges: list[list[int, int]]) ->
    list[int]:
    indeg = [0] * v
    outedge = [[] for _ in range(v)]

    for start, end in edges:
        indeg[end] += 1
        outedge[start].append[end]

    sorted_g = [i for i in range(v) if indeg[i] == 0]
    deq = deque(sorted_g)

    while deq:
        node = deq.popleft()
        for connected_node in outedge[node]:
```

```
            e -= 1
            indeg[connected_node] -= 1
            if indeg[connected_node] == 0:
                sorted_g.append(connected_node)
                deq.append(connected_node)

    if e != 0:
        return None

    return sorted_g
```

ii.　Tarjan のアルゴリズム

Tarjan のアルゴリズムは、深さ優先探索を用いてトポロジカルソートを行うアルゴリズムです。アルゴリズムの流れは以下の通りです。

1. 未訪問のノードを訪問済にする
2. そのノードに繋がっているノードを再帰的に訪問する
3. そのノードのすべての子ノードを訪問し終えたら、そのノードをソート済配列に追加する

Tarjan のアルゴリズムの実装例を以下に示します。

コード 44　Tarjan のアルゴリズムの実装

```
def tarjan_topological_sort(v: int, edges: list[list[int, int]]) -> list[
    int]:
    def dfs(node: int):
        visited[node] = True
        for connected_node in outedge[node]:
            if not visited[connected_node]:
                dfs(connected_node)
        sorted_g.append(node)

    outedge = [[] for _ in range(v)]
    visited = [False] * v
    sorted_g = []

    for start, end in edges:
        outedge[start].append(end)
```

```
15
16      for i in range(v):
17          if not visited[i]:
18              dfs(i)
19
20      return sorted_g[::-1]
```

XXXII.　最大流問題

i.　最大流問題と貪欲法の限界

ii.　フォード・ファルカーソン法

XXXIII.　最小費用流問題

XXXIV.　二部グラフ

XXXV.　参考

ベルマンフォード法

- https://qiita.com/ko-ya346/items/359a3e03c5e20b04c573

トポロジカルソート