■バブルソート

```
a = [80, 50, 56, 30, 51, 70]
N = len(a)

for i in range(N - 1):
    for j in range(N - 1, i, -1):
        if a[j] < a[j-1]:
            a[j], a[j-1] = a[j-1], a[j]
print(a)
```

`for i in range(N - 1):` この外側のループは、リスト全体を N-1 回スキャンします。
`for j in range(N - 1, i, -1):` 内側のループは、リストの末尾から現在の i の位置まで逆方向に要素を比較します。

`if a[j] < a[j-1]:` 隣接する要素を比較し、右の要素が左の要素より小さい場合はその位置を交換します。
`a[j], a[j-1] = a[j-1], a[j]` この操作で要素の位置を交換します。
このプロセスを繰り返すことで、リストの最小の要素が順に前方に移動し（"バブルアップ"）、リストが昇順に並び替えられます。

■2 分探索法

```
a = [2, 3, 7, 11, 31, 50, 55, 70, 77, 80]
N = len(a)    # データ数
keydata = 55
flag = 0
low, high = 0 , N-1
while low <= high:
    mid = (low + high) // 2
    if a[mid] == keydata:
        print('{:d} は {:d} 番目にありました'.format(a[mid], mid))
        flag = 1
        break
    if a[mid] < keydata:
        low = mid + 1
    else:
        high = mid - 1
if flag != 1:
    print('見つかりませんでした')
```

このコードは、二分探索法を使って、ソートされた配列 a の中から特定の値 keydata を探索するプログラムです。このアルゴリズムは、探索範囲を段階的に狭めながら目的の値を効率的に見つけ出します。コードの各部分の意味は以下の通りです：

初期設定：

a は探索するソートされた配列

N は配列 a の長さ（要素数）

keydata は探索したい値

flag は、値が見つかったかどうかを示すフラグです（初期値は 0）。

探索範囲の初期化：

・low, high は探索範囲の下限と上限を示します（初めは配列の全範囲）。

二分探索のループ：

`while low <= high:` 探索範囲が存在する間、ループを続けます。

`mid = (low + high) // 2:` 探索範囲の中間点を計算します。

`if a[mid] == keydata:` 中間点の値が目的の値と一致した場合、位置を出力し、flag を 1 に設定して

ループを終了します。
`if a[mid] < keydata:` 中間点の値が目的の値より小さい場合、探索範囲の下限を中間点の次の要素に更新します。
`else:` 中間点の値が目的の値より大きい場合、探索範囲の上限を中間点の前の要素に更新します。
探索結果:

`if flag != 1:` もし flag が 1 でない場合、つまり目的の値が見つからなかった場合、「見つかりませんでした」と出力します。

■ハッシュ

ハッシュは、データを効率的に格納、検索、比較するための技術です。具体的には、ある入力（たとえば文字列やファイル）に対して、固定長の一意的な値（ハッシュ値）を生成する関数（ハッシュ関数）を使います。ハッシュの主な用途と特徴は以下の通りです：

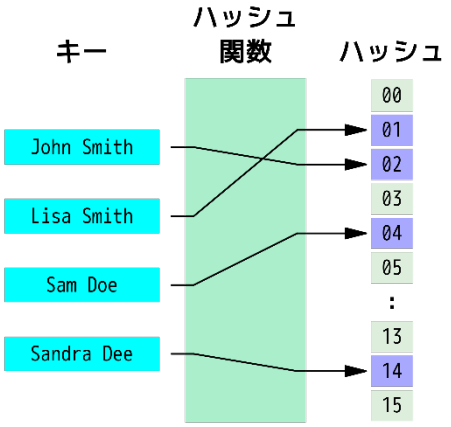
データの整理と高速アクセス：ハッシュテーブルは、ハッシュ値をキーとしてデータを格納します。ハッシュ値を使用することで、データの検索、挿入、削除を高速に行うことができます。

データの一意性の確認：ファイルやデータのハッシュ値は、その内容の「指紋」として機能します。異なるデータからは通常、異なるハッシュ値が生成されるため、データの整合性の確認に用いられます。

セキュリティ：ハッシュは暗号化技術にも応用されます。例えば、パスワードの格納には、元のパスワードから生成されたハッシュ値が使われることが多いです。これにより、元のパスワードを直接保存するリスクを避けることができます。

ハッシュ衝突：理論上は、異なるデータから同じハッシュ値が生成される場合があります（ハッシュ衝突）。良いハッシュ関数は、この衝突の可能性を極力低く保つように設計されています。

ハッシュ関数の例としては、MD5、SHA-1、SHA-256 などがありますが、用途に応じて適切なハッシュ関数を選ぶ必要があります。例えば、セキュリティが重要な場面では、MD5 や SHA-1 よりも強力な SHA-256 が推奨されます。




```
class Person:
    def __init__(self, name, tel):
        self.name = name
        self.tel = tel

def hash(s):    # ハッシュ関数
    n = len(s)
    return ((ord(s[0]) - ord('A')
            + (ord(s[n//2 - 1]) - ord('A')) * 26
            + (ord(s[n - 2]) - ord('A')) * 26 * 26) % ModSize)

TableSize = 1000
ModSize = 1000
table = [Person('', '') for i in range(TableSize)]  # データ・テーブル

while (data := input('名前,電話番号？')) != '/':
    sdata = data.split(',')
    name = sdata[0]
    tel = sdata[1]
    n = hash(name)
    table[n] = Person(name, tel)

while (data := input('検索するデータ？')) != '/':
    n = hash(data)
    print('{:d} {:s} {:s}'.format(n, table[n].name, table[n].tel))
```

◆ハッシュ関数の部分の働き

この hash 関数は、与えられた文字列 s に対してハッシュ値を計算するカスタムハッシュ関数です。この関数の動作は次のようになります：

1. 文字列 s の長さ n を取得します。

2. 文字列の最初の文字（s[0]）、中央の文字（s[n//2 - 1]）、および最後から 2 番目の文字（s[n - 2]）の ASCII 値を取得します。

3. 各文字の ASCII 値から、大文字の'A'の ASCII 値を引きます。これにより、大文字アルファベットの文字を 0 から 25 の範囲の数値に変換します。

4. これらの数値に対して、特定の重み付けを行います。最初の文字はそのまま、中央の文字には 26 を乗算し、最後から 2 番目の文字には 26 の 2 乗を乗算します。これにより、文字列の異なる部分が異なる影響をハッシュ値に与えるようにしています。

5. 最後に、これらの値の合計をあるモジュラス ModSize で割った余りをハッシュ値として返します。ModSize はおそらくハッシュテーブルのサイズであり、ハッシュ値を特定の範囲内に制限するために使われます。

このハッシュ関数は、文字列の特定の文字を使ってハッシュ値を計算するという点でユニークです。ただし、実際のアプリケーションではこの方法は限られた分布とハッシュ衝突のリスクを伴うため、通常はより複雑なハッシュ関数が使用されます。特に、セキュリティが重要な用途では、暗号学的に強力なハッシュ関数が必要です。

```
hash(A1, A2･･･An)=(A1+ An/2×26+An-1×262) mod 1000
```

たとえば、

```
hash(“SUZUKI”)=((‘S’-‘A’)+(‘Z’-‘A’)×26+(‘K’-‘A’)×262)%1000
              =(18+650+6760)%1000
              =428
```

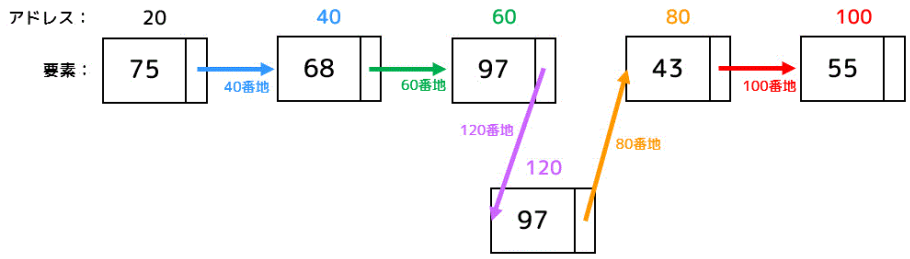
クイックソート

```
def quick(a, left, right):
    if left < right:
        s = a[(left + right) // 2]    # 中央の値を軸にする
        i = left - 1          # 軸より小さいグループと
        j = right + 1         # 大きいグループに分ける
        while True:
            i += 1
            while a[i] < s:
                i += 1
            j -= 1
            while a[j] > s:
                j -= 1
            if i >= j:
                break
            a[i], a[j] = a[j], a[i]

        quick(a, left, i - 1)   # 左部分列に対する再帰呼び出し
        quick(a, j + 1, right)  # 右部分列に対する再帰呼び出し

a = [41, 24, 76, 11, 45, 64, 21, 69, 19, 36]
N = len(a)
quick(a, 0, N - 1)

print(a)
```

リスト　データ構造

```
class Person:
    def __init__(self, name, age, next):
        self.name = name
        self.age = age
        self.next = next

    def disp(self):
        print('name={:s},age={:d}'.format(self.name, self.age))

def ins(key, name, age):
    global tail
    for i in range(tail):
        if a[i].name == key:
            a[tail] = Person(name, age, a[i].next)
            a[i].next = tail
            tail += 1

Max = 20
a = [Person('', 0, 0) for i in range(Max)]
a[0] = Person('taro', 18, 1)
a[1] = Person('jiro', 17, 2)
a[2] = Person('kotaro', 19, -1)
tail = 3  # リストの末尾

ins('taro', 'gaku', 16)
ins('kotaro', 'ken', 21)

p = 0
while p != -1:
    a[p].disp()
    p = a[p].next
```

リストへのデータ挿入




```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def disp(self):
        print('name={:s},age={:d}'.format(self.name, self.age))

a = [Person('taro', 18), Person('jiro', 17), Person('kotaro', 19)]
b = Person('gaku', 16)
keyname = 'jiro'
flag = 0
for i in range(len(a)):
    if a[i].name == keyname:
        a.insert(i, b)  # keyname の前に挿入
        flag = 1
        break
if (flag == 0):  # keyname がなければ末尾に追加
    a.append(b)

for ai in a:
    ai.disp()
```

リストからの削除

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def disp(self):
        print('name={:s},age={:d}'.format(self.name, self.age))

a = [Person('taro', 18), Person('jiro', 17), Person('kotaro', 19)]
keyname = 'jiro'
for i in range(len(a)):
    if a[i].name == keyname:
        del a[i]
        break

for ai in a:
    ai.disp()
```

スタック

```
def push(n):    # スタックにデータを積む手続き
    global sp
    if sp < MaxSize:
        stack[sp] = n
        sp += 1
        return 0
    else:
        return -1   # スタックが一杯のとき

def pop():          # スタックからデータを取り出す手続き
    global sp
    if sp > 0:
        sp -= 1
        return 0, stack[sp]
    else:
        return -1, 0 # スタックが空のとき

MaxSize = 100         # スタック・サイズ
stack = [0 for i in range(MaxSize)]   # スタック
sp = 0                # スタック・ポインタ

while (c := input('i:push, o:pop, e:end ?')) != 'e':
    if c == 'i':
        data = input('data?')
        ret = push(data)
        if ret == -1:
            print('スタックが一杯です')
    if c == 'o':
        ret, n=pop()
        if ret == -1:
            print('スタックは空です')
        else:
            print('pop --> {:s}'.format(n))
```

ハノイ

```
def hanoi(n, a, b, c):      # 再帰手続
    if n > 0:
        hanoi(n - 1, a, c, b)
        move(n, a, b)
        hanoi(n - 1, c, b, a)

def move(n, s, d):          # 円盤の移動シミュレーション
    pie[sp[d]][d] = pie[sp[s] - 1][s]   # s->d へ円盤の移動
    sp[d] += 1                          # スタック・ポインタの更新
    sp[s] -= 1
    for i in range(N - 1, -1, -1):
        result = ''
        for j in range(3):
            if i < sp[j]:
                result += '{:8d}'.format(pie[i][j])
            else:
                result += '        '
        print(result)
    print('       a       b       c')
    print('{:d} 番の円盤を {:s}-->{:s} に移す'.
          format(n, chr(ord('a') + s),chr(ord('a') + d)))

pie = [[0] * 3 for i in range(20)]  # 20:円盤の最大枚数, 3:棒の数
sp = [0, 0, 0]                      # スタック・ポインタ

N = 4
for i in range(N):                  # 棒 a に円盤を積む
    pie[i][0] = N - i
sp[0], sp[1], sp[2] = N, 0, 0       # スタック・ポインタの初期設定

hanoi(N, 0, 1, 2)
```

キュー

```
def queuein(n):     # キューにデータを入れる手続き
    global head, tail
    if (tail + 1) % MaxSize != head:
        queue[tail] = n
        tail += 1
        tail %= MaxSize
        return 0
    else:
        return -1   # キューが一杯のとき

def queueout():     # キューからデータを取り出す手続き
    global head, tail
    if tail != head:
        n = queue[head]
        head += 1
        head %= MaxSize
        return 0, n
    else:
        return -1, 0  # キューが空のとき

MaxSize = 100         # キュー･サイズ
queue = [0 for i in range(MaxSize)]   # キュー
head = tail = 0       # 先頭データ,終端データへのポインタ

while (c := input('i:queuein, o:queueout, e:end ?')) != 'e':
    if c == 'i':
        data = input('data?')
        ret = queuein(data)
        if ret == -1:
            print('待ち行列が一杯です')
    if c == 'o':
        ret, n = queueout()
        if ret == -1:
            print('待ち行列は空です')
        else:
            print('queue data --> {:s}'.format(n))
```

キュー　head tail

```
def queuein(n):     # キューにデータを入れる手続き
    global head, tail
    if (tail + 1) % MaxSize != head:
        queue[tail] = n
        tail += 1
        tail %= MaxSize
        return 0
    else:
        return -1   # キューが一杯のとき

def queueout():     # キューからデータを取り出す手続き
    global head, tail
    if tail != head:
        n = queue[head]
        head += 1
        head %= MaxSize
        return 0, n
    else:
        return -1, 0 # キューが空のとき

def disp():         # 待ち行列の内容を表示する手続き
    global head, tail
    i = head
    result = ''
    while i != tail:
        result += '{:s},'.format(queue[i])
        i += 1
        i = i % MaxSize
    print('all queue data --> {:s}'.format(result))

MaxSize = 100         # キュー・サイズ
queue = [0 for i in range(MaxSize)]   # キュー
head = tail = 0       # 先頭データ,終端データへのポインタ
    ......
```

```
    ......

while (c := input('i:queuein, o:queueout, l:list, e:end ?')) != 'e':
    if c == 'i':
        data = input('data?')
        ret = queuein(data)
        if ret == -1:
            print('待ち行列が一杯です')
    if c == 'o':
        ret, n = queueout()
        if ret == -1:
            print('待ち行列は空です')
        else:
            print('queue data --> {:s}'.format(n))
    if c == 'l':
        disp()
```

tree 「表現」

```
class Person:
    def __init__(self, left, name, right):
        self.left = left
        self.name = name
        self.right = right

nil = -1
a = [Person(  1, 'Machilda',   2), Person(  3, 'Candy',   4),
     Person(  5, 'Rolla'   , nil), Person(nil, 'Ann'  , nil),
     Person(  6, 'Emy'     ,   7), Person(nil, 'Nancy', nil),
     Person(nil, 'Eluza'   , nil), Person(nil, 'Lisa' , nil)]

keyname = 'Ann'
p = 0
while p != nil:
    if keyname == a[p].name:
        print('{:s}は見つかりました'.format(keyname))
        break
    elif keyname < a[p].name:
        p = a[p].left     # 左部分木へ移動
    else:
        p = a[p].right    # 右部分木へ移動
if p == nil:
    print('{:s}は見つかりませんでした'.format(keyname))
```

データの追加

```
class Person:
    def __init__(self, left, name, right):
        self.left = left
        self.name = name
        self.right = right

nil = -1
a = [Person(  1, 'Machilda',   2), Person(  3, 'Candy',   4),
     Person(  5, 'Rolla'   , nil), Person(nil, 'Ann'  , nil),
     Person(  6, 'Emy'     ,   7), Person(nil, 'Nancy', nil),
     Person(nil, 'Eluza'   , nil), Person(nil, 'Lisa' , nil)]

keyname = 'Patie'
p = 0             # 木のサーチ
while p != nil:
    old = p
    if keyname <= a[p].name:
        p = a[p].left
    else:
        p = a[p].right

a.append(Person(nil, keyname,nil))   # 新しいノードの接続

if keyname <= a[old].name:
    a[old].left = len(a)-1
else:
    a[old].right = len(a)-1

for ai in a:
    print('{:2d},{:10s},{:2d}'.format(ai.left, ai.name, ai.right))
```