提出日：2024/7/17

プログラミング演習　第13回演習レポート

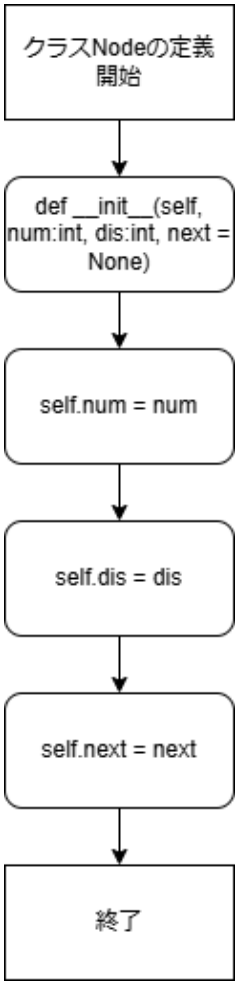
担当教員：杉本　千佳先生
所属：理工学部　数物・電子情報系学科
電子情報システムEP
学年・クラス：２年　Fe1
学籍番号：2364092
氏名：熊田　真歩

（１）課題番号、課題名：第 12 回基本課題 1、「全探索
基本課題２「線形探索」
第 13 回基本課題 1、「二分探索」

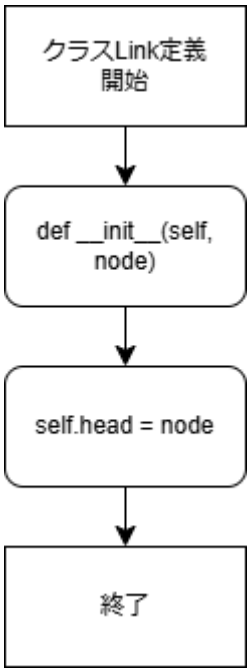
(2)プログラムのフローチャート

・第 12 回基本課題 1「全探索」

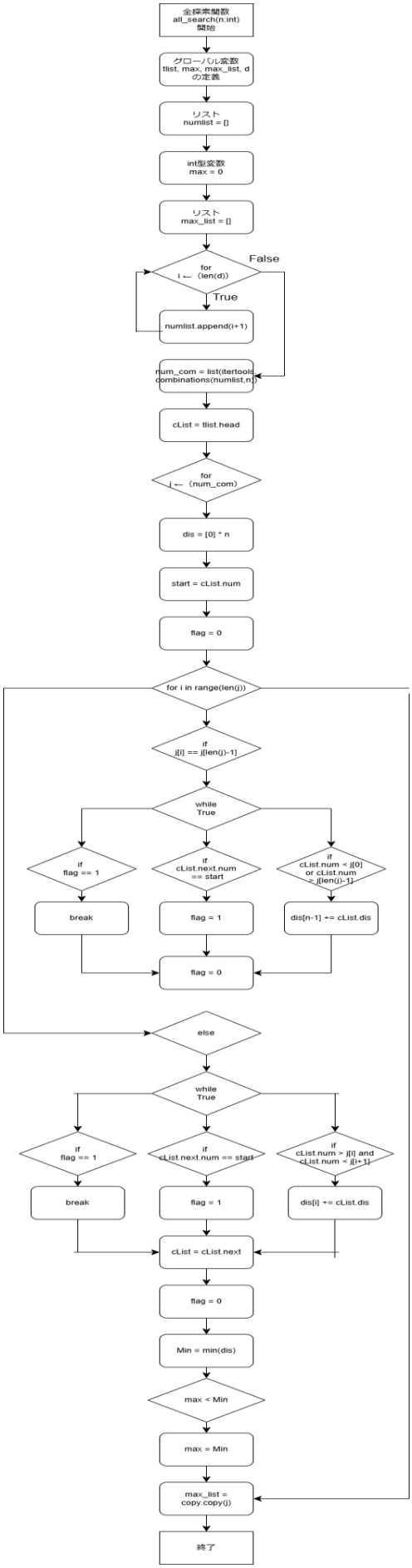
・クラス Node について



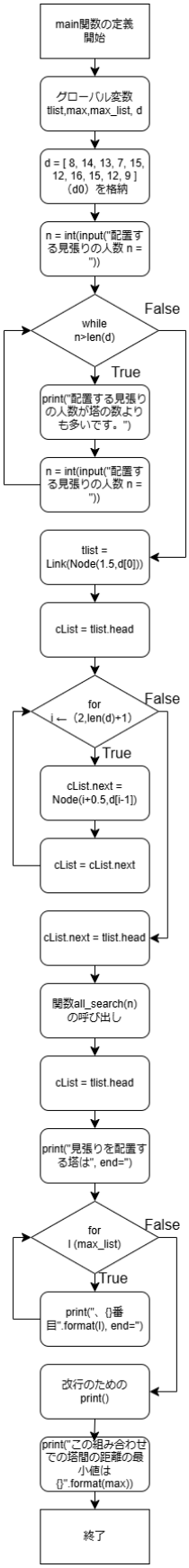
・クラス Link について



・全探索関数について

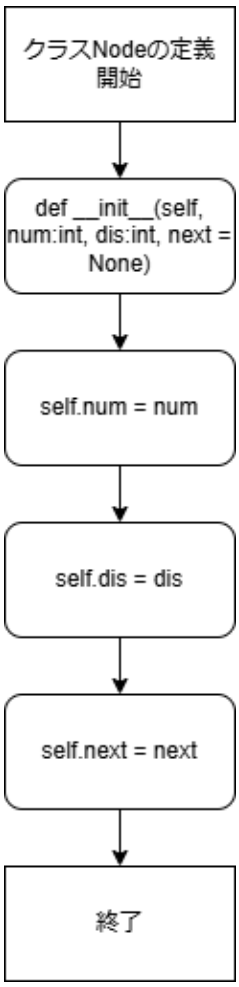

全探索関数
all_search(n:int)
開始
グローバル変数
tlist, max, max_list, d
の定義
リスト
numlist = []
int型変数
max = 0
リスト
max_list = []
for
i ← (len(d))
False
True
numlist.append(i+1)
num_com = list(itertools.
combinations(numlist,n))
cList = tlist.head
for
j ← (num_com)
dis = [0] * n
start = cList.num
flag = 0
for i in range(len(j))
if
j[i] == j[len(j)-1]
while
True
if
flag == 1
break
if
cList.next.num
== start
flag = 1
if
cList.num < j[0]
or cList.num
> j[len(j)-1]
dis[n-1] += cList.dis
flag = 0
else
while
True
if
flag == 1
break
if
cList.next.num == start
flag = 1
if
cList.num > j[i] and
cList.num < j[i+1]
dis[i] += cList.dis
cList = cList.next
flag = 0
Min = min(dis)
max < Min
max = Min
max_list =
copy.copy(j)
終了

・main 関数について

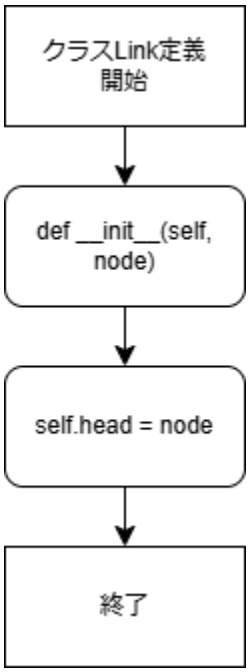

main関数の定義
開始
グローバル変数
tlist,max,max_list, d
d = [8, 14, 13, 7, 15, 12, 16, 15, 12, 9]
（d0）を格納
n = int(input("配置する見張りの人数 n = "))
while
n>len(d)
False
True
print("配置する見張りの人数が塔の数よりも多いです。")
n = int(input("配置する見張りの人数 n = "))
tlist = Link(Node(1.5,d[0]))
cList = tlist.head
for
i ← (2,len(d)+1)
False
True
cList.next = Node(i+0.5,d[i-1])
cList = cList.next
cList.next = tlist.head
関数all_search(n)の呼び出し
cList = tlist.head
print("見張りを配置する塔は", end="")
for
l (max_list)
False
True
print(", {}番目".format(l), end="")
改行のための
print()
print("この組み合わせでの塔間の距離の最小値は{}".format(max))
終了

・第 12 回基本課題 2「線形探索」

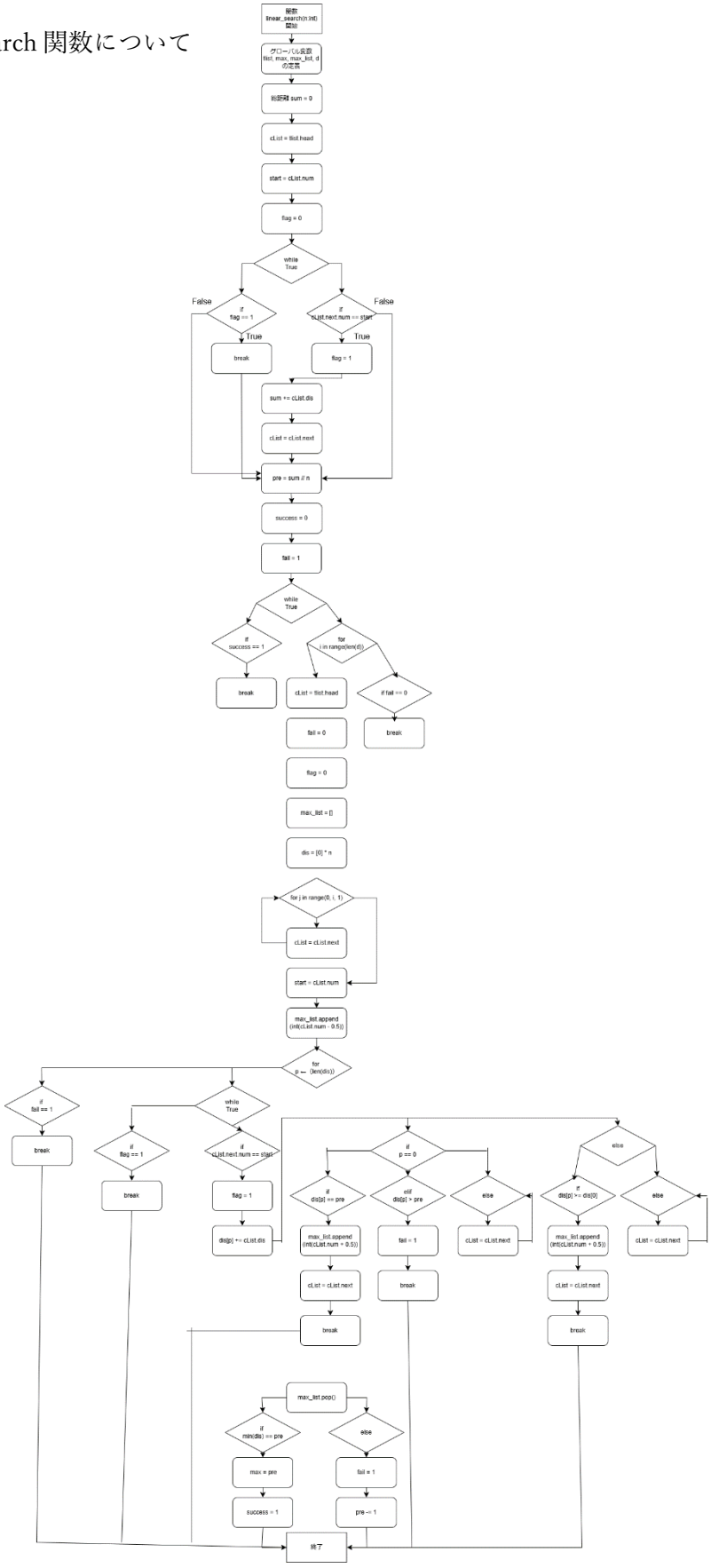
・クラス Node について



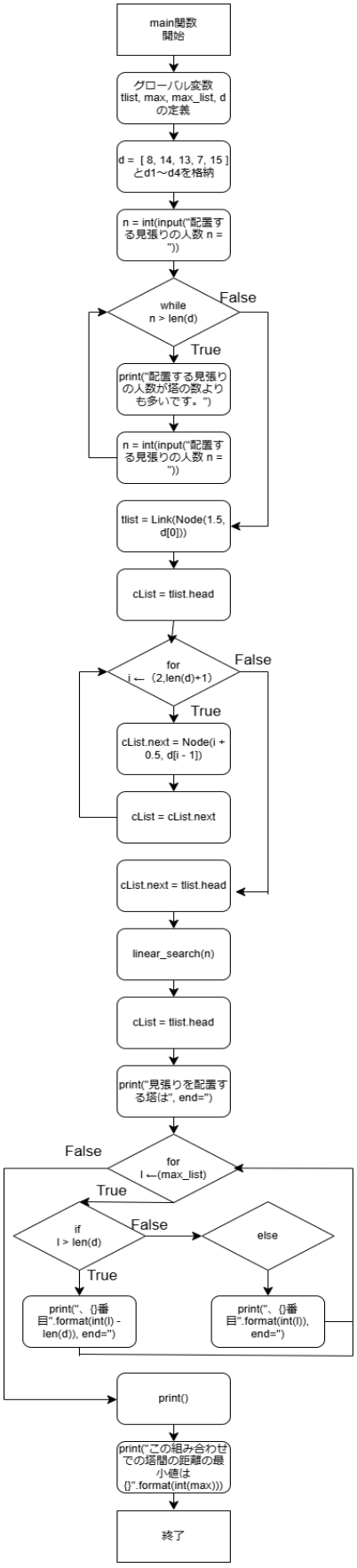
・クラス Likedlist について



・関数 liner__search 関数について

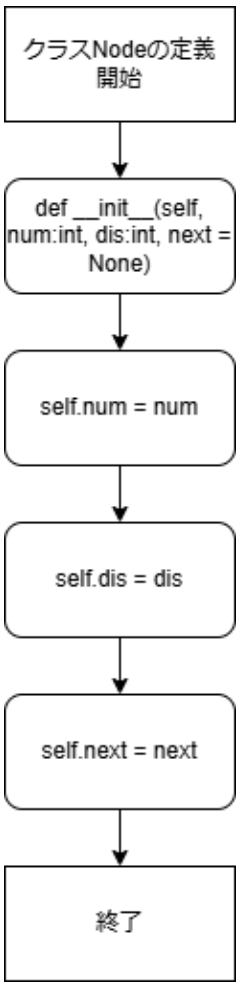


・main 関数について

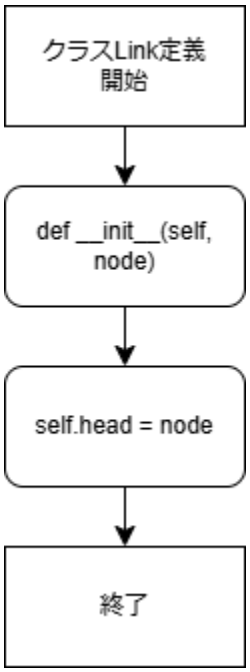

main関数
開始
グローバル変数
tlist, max, max_list, d
の定義
d = [8, 14, 13, 7, 15]
とd1～d4を格納
n = int(input("配置する見張りの人数 n = "))
while
n > len(d)
False
True
print("配置する見張りの人数が塔の数よりも多いです。")
n = int(input("配置する見張りの人数 n = "))
tlist = Link(Node(1.5, d[0]))
cList = tlist.head
for
i ← (2,len(d)+1)
False
True
cList.next = Node(i + 0.5, d[i - 1])
cList = cList.next
cList.next = tlist.head
linear_search(n)
cList = tlist.head
print("見張りを配置する塔は", end="")
for
l ←(max_list)
False
True
if
l > len(d)
False
else
True
print("、{}番目".format(int(l) - len(d)), end="")
print("、{}番目".format(int(l)), end="")
print()
print("この組み合わせでの塔間の距離の最小値は{}".format(int(max)))
終了

・第 13 回基本課題1「2 分探索」

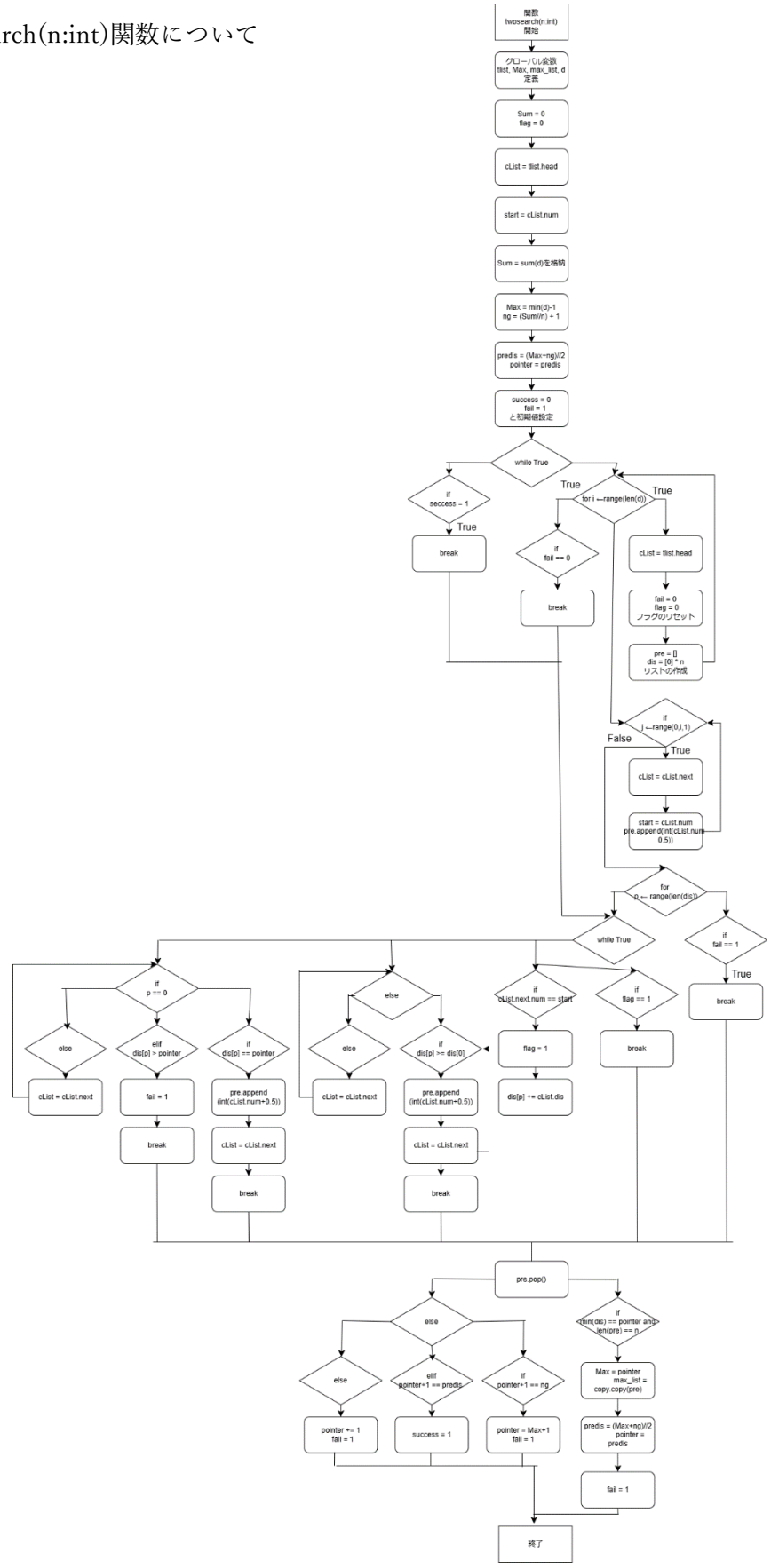
・クラス Node について



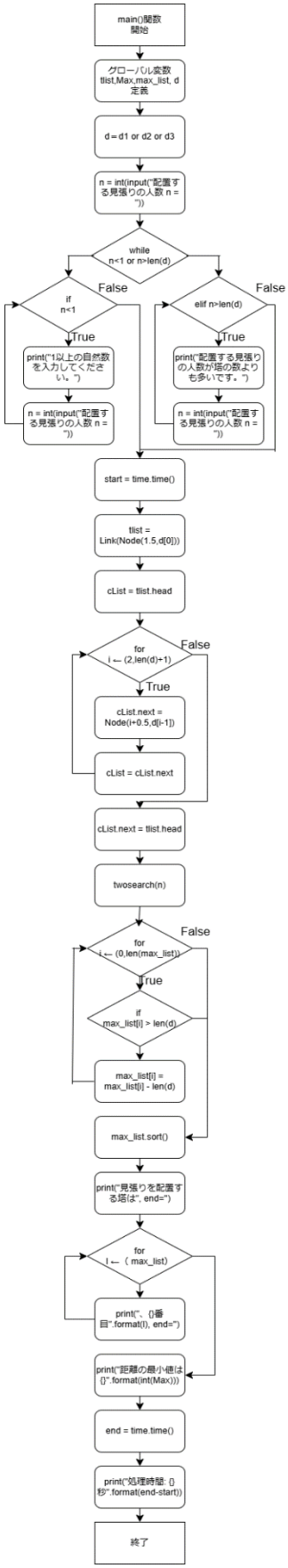
・クラス Likedlist について



・twosearch(n:int)関数について



・main 関数について


main()関数
開始
グローバル変数
tlist,Max,max_list, d
定義
d＝d1 or d2 or d3
n = int(input("配置する見張りの人数 n = "))
while
n<1 or n>len(d)
if
n<1
False
True
print("1以上の自然数を入力してください。")
n = int(input("配置する見張りの人数 n = "))
elif n>len(d)
False
True
print("配置する見張りの人数が場の数よりも多いです。")
n = int(input("配置する見張りの人数 n = "))
start = time.time()
tlist =
Link(Node(1.5,d[0]))
cList = tlist.head
for
i ← (2,len(d)+1)
False
True
cList.next =
Node(i+0.5,d[i-1])
cList = cList.next
cList.next = tlist.head
twosearch(n)
for
i ← (0,len(max_list))
False
True
if
max_list[i] > len(d)
max_list[i] =
max_list[i] - len(d)
max_list.sort()
print("見張りを配置する場は", end="")
for
i ← (max_list)
print("、{}番目".format(i), end="")
print("距離の最小値は{}".format(int(Max)))
end = time.time()
print("処理時間: {}秒".format(end-start))
終了

(3)アルゴリズムが「正しいこと」である説明

(3) －1. 第 12 回基本課題１「全探索」

初めに、全ての入力に対してプログラムは停止するということを説明する。基本課題 1 においては配置する見張り人の人数を int 型変数として入力する。ここで見張り人の数は、自然数であるという前提条件のもとにプログラムを作成した。この時、見張り人の数が塔の数よりも多い時、プログラムはエラーメッセージを表示する。そして再び見張り人の数を聞き、結果の出力後、プログラムを終了するという事が実装されている。無論、見張り人の数が塔の数よりも少ない時は結果出力後、プログラムは停止する。すなわち、見張り人が自然数人である時は全ての入力にたいしてプログラムは停止する。

次にすべての入力に対して「正しい答え」を出力することについて説明する。塔の数よりも見張り人の数が多い時はプログラムが成立しないため、エラーメッセージを表示する。これは正しい答えであると言える。また、見張り人の数が塔の数よりも少ない場合は正常にプログラムを実行する。すなわち、データ d0 および d1 に対して見張りの人数を入力した時、見張りを配置する塔の番号と塔間の距離の最小値を出力する。 この結果はいずれの場合も手計算で算出した結果と一致した。すなわち、プログラムは全ての入力に対して「正しい答え」を出力する。

（３）－２．第 12 回基本課題２「線形探索」

第 12 回基本課題 1 と同様に、初めに、全ての入力に対してプログラムは停止するということを説明する。基本課題 1 においては配置する見張り人の人数を int 型変数として入力する。ここで見張り人の数は、自然数であるという前提条件のもとにプログラムを作成した。この時、見張り人の数が塔の数よりも多い時、プログラムはエラーメッセージを表示する。そして再び見張り人の数を聞き、結果の出力後、プログラムを終了するという事が実装されている。無論、見張り人の数が塔の数よりも少ない時は結果出力後、プログラムは停止する。すなわち、見張り人が自然数人である時は全ての入力にたいしてプログラムは停止する。

次にすべての入力に対して「正しい答え」を出力することについて説明する。塔の数よりも見張り人の数が多い時はプログラムが成立しないため、エラーメッセージを表示する。これは正しい答えであると言える。また、見張り人の数が塔の数よりも少ない場合は正常にプログラムを実行する。すなわち、データ d1、d2、d3 および d4 に対して見張りの人数を入力した時、見張りを配置する塔の番号と塔間の距離の最小値を出力する。この結果はいずれの場合も手計算で算出した結果と一致した。すなわち、プログラムは全ての入力に対して「正しい答え」を出力する。

(3) －3. 第 13 回基本課題 1「2 分探索」

第 12 回基本課題 1 および 2 と同様に、初めに、全ての入力に対してプログラムは停止するということを説明する。基本課題 1 においては配置する見張り人の人数を int 型変数として入力する。ここで見張り人の数は、自然数であるという前提条件のもとにプログラムを作成した。この時、見張り人の数が塔の数よりも多い時、プログラムはエラーメッセージを

表示する。そして再び見張り人の数を聞き、結果の出力後、プログラムを終了するという事が実装されている。無論、見張り人の数が塔の数よりも少ない時は結果出力後、プログラムは停止する。すなわち、見張り人が自然数人である時は全ての入力にたいしてプログラムは停止する。

次にすべての入力に対して「正しい答え」を出力することについて説明する。塔の数よりも見張り人の数が多い時はプログラムが成立しないため、エラーメッセージを表示する。これは正しい答えであると言える。また、見張り人の数が塔の数よりも少ない場合は正常にプログラムを実行する。すなわち、データ d2、d3 および d4 に対して見張りの人数を入力した時、見張りを配置する塔の番号と塔間の距離の最小値を出力する。この結果はいずれの場合も手計算で算出した結果と一致した。すなわち、プログラムは全ての入力に対して「正しい答え」を出力する。

つまり、3 つの探索方法のいずれの場合も説明方法は同様であることがわかる。

（４）ソース・プログラムの説明

（４）－1. 第 12 回基本課題 1「全探索」

```
import itertools, copy    #使用するツールの導入

class Node:               #クラス Node の定義
    def __init__(self, num:int, dis:int, next = None): #クラスの定義
        self.num = num    #num をクラス変数に適応
        self.dis = dis    #dis をクラス変数に適応
        self.next = next  #next をクラス変数に適応

class Link:               #クラス Link の定義
    def __init__(self, node):  #クラスの定義
        self.head = node       #head をクラス変数に適応

def all_search(n:int):         #全探索関数の定義
    global tlist, max, max_list, d  #4 つのグローバル変数を設定
    numlist = []                    #空リストを作成
    max = 0                         #変数 max の初期値を 0 で定義
    max_list = []                   #最大リストも初期では空リストとして定義
    for i in range(len(d)):         #d の長さ分つまり塔の数分だけ探索を行う
        numlist.append(i+1)         #numlist に i＋1の値を加えていく
    num_com = list(itertools.combinations(numlist,n)) #新たなリストを作成。
```

```
    cList = tlist.head                                  #cList にタワーの先頭を
                                                          加える
    for j in num_com:                                   #探索のリストそれぞれに対
                                                          して
        dis = [0] * n                                   #0 のみのリストをn個作成
                                                          したものを変数 dis に代入
        start = cList.num                               #探索の開始点の設定
        flag = 0                                       #ループのためにフラグを作成
        for i in range(len(j)):                         #全ての塔に対して
            if j[i] == j[len(j)-1]:                     #現在の棟が最後である時
                while True:                           #以下の条件が正しいときまでループ
                    if flag == 1:                       #フラグが1の時
                        break                           #ループ終了
                    if cList.next.num == start:      #開始点が次の値と等しい時
                        flag = 1                        #フラグを1とする
                    if cList.num < j[0] or cList.num > j[len(j)-1]: #この条
                                                                      件の時
                        dis[n-1] += cList.dis               #最後の距離を加算
                    cList = cList.next                  #探索リストの次の数を格納
                flag = 0                                #フラグを初期化
            else:                                       #上の条件以外の時
                while True:                           #以下の条件が正しい時ループ
                    if flag == 1:                       #フラグに1が立っている時
                        break                           #ループ終了
                    if cList.next.num == start:      #現在の塔が初期であるとき
                        flag = 1                        #フラグに1をたてる
                    if cList.num > j[i] and cList.num < j[i+1]:  #現在の塔
                                                          が最初でも最後でもない時
                        dis[i] += cList.dis             #数値をインクリメント
                    cList = cList.next                  #現在のリストを次に移す
                flag = 0                                #フラグを初期化
        Min = min(dis)                                  #最小距離を取得
        if max < Min:                                #最小距離が最大距離よりも大きい時
            max = Min                               #最小距離の値を最大距離の値に代入
            max_list = copy.copy(j)                     #最大の最小距離を持つノードの
                                                          リストを更新
```

```
def main():                          #main 関数の定義
    global tlist,max,max_list, d     #全探索関数と同様の変数をグローバル変数として定義
    d = [ 8, 14, 13, 7, 15, 12, 16, 15, 12, 9 ]  #d0 または d1 を適応させる
    n = int(input("配置する見張りの人数 n = "))   #配置する見張り人の数を変数 n に代入
    while n>len(d):                  #見張り人の数が等の数よりも多い時
        print("配置する見張りの人数が塔の数よりも多いです。") #エラーメッセージを出力
        n = int(input("配置する見張りの人数 n = "))       #再び見張り人の数を聞く
    tlist = Link(Node(1.5,d[0]))     #リンクリストのヘッドノートの作成
    cList = tlist.head               #リスト、リンクの頭から開始
    for i in range(2,len(d)+1):      #塔の数＋1 回だけ処理を繰り返す
        cList.next = Node(i+0.5,d[i-1])  #次のノードの作成
        cList = cList.next           #現在の塔を次へ移す
    cList.next = tlist.head          #循環リストの作成
    all_search(n)                    #最適な配置を探索
    cList = tlist.head               #リスト、リンクの初期値をリセット
    print("見張りを配置する塔は", end='')  #見張りを配置する等を出力
    for l in max_list:
        print("、{}番目".format(l), end='')  #見張りを置く塔の出力
    print()                          #改行
    print("この組み合わせでの塔間の距離の最小値は {}".format(max)) #最小距離を出力

main()
```

(4)－2. 第 12 回基本課題 2「線形探索」

```
class Node:#クラス Node の定義
    def __init__(self, num:int, dis:int, next=None):#クラスの定義
        self.num = num    #num をクラス変数に適応
        self.dis = dis    #dis をクラス変数に適応
        self.next = next  #next をクラス変数に適応
```

```
class Link:                #クラス Link の定義
    def __init__(self, node):  #クラスの定義
        self.head = node       #head をクラス変数に適応
def linear_search(n:int):      #線形探索関数の定義
    global tlist, max, max_list, d  #4 つのグローバル変数を設定
    sum = 0                    #総距離の初期化
    cList = tlist.head         #リスト、リンクを頭からスタート
    start = cList.num          #スタートノード番号の格納
    flag = 0                   #while ループを扱うフラグの初期値を 0
    while True:                #条件が正しい時ループ
        if flag == 1:          #フラグに 1 が立っている時
            break              #ループ終了
        if cList.next.num == start:  #現在の塔が頭である時
            flag = 1                 #フラグに 1 を立てる
        sum += cList.dis             #総距離に現在の距離を加算
        cList = cList.next           #扱う塔の対象を次へと移す
    pre = sum // n                   #平均距離の計算
    success = 0                      #成功した場合のフラグを設定
    fail = 1                         #失敗した場合のフラグを設定
    while True:                      #条件が正しい時のループ
        if success == 1:             #成功フラグが 1 の時
            break                    #ループ終了
        for i in range(len(d)):      #塔の数だけループ
            if fail == 0:            #失敗フラグが 0 の時
                break                #ループ終了
            cList = tlist.head       #リストリンクの頭から再スタート
            fail = 0                 #失敗フラグの初期化
            flag = 0                 #フラグを値を初期化
            max_list = []            #最適な配置リストを初期化
            dis = [0] * n            #距離リストの初期設定
            for j in range(0, i, 1):
                cList = cList.next   #塔の対象を次へと移す
            start = cList.num        #ノードの頭の番号を格納
            max_list.append(int(cList.num - 0.5))
```

```
        for p in range(len(dis)): #dis の数だけ繰り返し
            if fail == 1:         #失敗フラグの値が 1 の時
                break             #ループ終了
            while True:           #条件が正しい時ループ
                if flag == 1:     #フラグに 1 が立っている時
                    break         #ループ終了
                if cList.next.num == start: #扱っている塔が初めの塔の時
                    flag = 1                 #フラグに 1 を立てる
                dis[p] += cList.dis          #距離を変数に加算
                if p == 0:                   #算出した距離が 0 である時
                    if dis[p] == pre:        #ここに平均距離を格納
                        max_list.append(int(cList.num + 0.5))
                        cList = cList.next   #対象を次の塔へ移す
                        break                #ループを抜ける
                    elif dis[p] > pre:       #距離が全体の平均値よりも長い時
                        fail = 1             #失敗フラグに 1 を立てる
                        break                #ループ終了
                    else:                    #上のふたつの条件以外の時
                        cList = cList.next   #対象の塔を次のものに移す
                else:                        #算出した距離が 0 ではない時
                    if dis[p] >= dis[0]: #距離が初期値よりも長くなっている時
                        max_list.append(int(cList.num + 0.5))
                        cList = cList.next       #対象の塔を次のものに移す
                        break                    #ループ終了
                    else:                    #上の条件以外の時
                        cList = cList.next   #対象の塔を次の塔に移す
    max_list.pop()                           #最後の要素を削除
    if min(dis) == pre:                      #最小距離が平均値と等しい時
        max = pre                #最大値を平均値に格納
        success = 1              #成功フラグに 1 をたてる
    else:                        #上記条件以外の時
        fail = 1                 #失敗フラグをたてる
        pre -= 1                 #平均値を−1 する
```

```
def main():                          #main 関数の定義
    global tlist, max, max_list, d  #4 つのグローバル変数の定義
    d =  [ 8, 14, 13, 7, 15 ]       #d1 を適応している
    n = int(input("配置する見張りの人数 n = "))    #配置する見張り人の数を変数 n
                                                    に代入
    while n > len(d):                          #見張り人の数が等の数よりも多い時
        print("配置する見張りの人数が塔の数よりも多いです。")  #エラーメッセージ
                                                            を出力
        n = int(input("配置する見張りの人数 n = "))     #再び見張り人の数を聞く

    tlist = Link(Node(1.5, d[0]))            #リンクリストのヘッドノートの作成
    cList = tlist.head                       #リスト、リンクの頭から開始
    for i in range(2, len(d) + 1):           #塔の数＋1 回だけ処理を繰り返す
        cList.next = Node(i + 0.5, d[i - 1]) #次のノードの作成
        cList = cList.next                   #現在の塔を次へ移す
    cList.next = tlist.head                  # 循環リストの作成

    linear_search(n)                         #線形探索関数の呼び出し
    cList = tlist.head                       #リスト、リンクの初期値をリセット
    print("見張りを配置する塔は", end='') #見張りを配置する等を出力
    for l in max_list:
        if l > len(d):                       #l が全体の塔の数よりも大きい時
            print("、{}番目".format(int(l) - len(d)), end='') #l から塔の数を
                                                            引いた数を出力
        else:                                #上記条件以外の時
            print("、{}番目".format(int(l)), end='') #l 番を結果として出力
    print()                                              #改行
    print("この組み合わせでの塔間の距離の最小値は {}".format(int(max)))
                                                        #距離の最小値を出力

main()                                                  #main 関数の実行
```

・第 13 回基本課題 1「2 分探索」

```
import itertools, copy, time    #用いるツールの導入
class Node:                     #クラス、Node の定義
    def __init__(self, num:int, dis:int, next = None): #定義文
        self.num = num          #変数 num の定義
        self.dis = dis          #変数 dis の定義
        self.next = next        #変数 next の定義

class Link:                     #クラス、Link の定義
    def __init__(self, node):   #定義文
        self.head = node        #変数 node の定義

def twosearch(n:int):           #二分探索関数定義
    global tlist, Max, max_list, d   #グローバル変数の定義
    Sum = 0                     #総距離の初期値を 0 とする
    cList = tlist.head          #最初の塔から探索を開始
    start = cList.num           #変数 strat を初期の塔に設定
    flag = 0                    #初期値でフラグは立てない。
    Sum = sum(d)                #総距離を格納
    Max = min(d)-1              #変数 Max に塔の最小距離－1 の値を格納
    ng = (Sum//n) + 1           #平均値＋1
    predis = (Max+ng)//2        #左の値を格納しておくことで処理に用いる
    pointer = predis            #ポインタを設定
    success = 0                 #成功フラグを設定、たてはしない
    fail = 1                    #失敗フラグを作成し、たてる
    while True:                 #条件式が正しい限り繰り返し
        if success == 1:        #成功フラグがたっているとき
            break               #ループ終了
        for i in range(len(d)): #塔の数だけ処理を繰り返す
            if fail == 0:       #失敗フラグが立っていないとき
                break           #ループ終了
            cList = tlist.head  #塔をはじめのものに再設定
            fail = 0            #失敗フラグ回収
            flag = 0            #フラグ回収
            pre = []            #リスト pre の作成
            dis = [0] * n       #リスト[0]を個作成
```

```
            for j in range(0,i,1): #範囲の中での繰り返し
                cList = cList.next #探索を行う塔を次の塔にうつす
            start = cList.num      #対象の塔を再設定
            pre.append(int(cList.num-0.5))  #リストに結果を踏まえ追加
            for p in range(len(dis)):       #塔それぞれに対して
                if fail == 1:               #失敗フラグが立っている時
                    break                   #ループ終了
                while True:         #条件が合うまでループを繰り返すことで探索を行う
                    if flag == 1:           #フラグが立っている時
                        break               #ループ終了
                    if cList.next.num == start: #次の塔が開始位置である時
                        flag = 1                #フラグをたてる
                    dis[p] += cList.dis         #見張りの塔の距離を更新
                    if p == 0:                  #見張りの位置が最初の場所である時
                       if dis[p] == pointer #更新した見張りの塔が適切であった時
                            pre.append(int(cList.num+0.5)) #距離を更新
                            cList = cList.next             #探索の対象の塔を
                                                             次の塔に移す
                            break                          #ループ終了
                        elif dis[p] > pointer:             #更新した塔が適切
                                                             でなかった時
                            fail = 1                       #失敗フラグをたてる
                            break                          #ループ終了
                        else:                              #そのほかの時
                            cList = cList.next             #探索の対象の塔を
                                                             次の塔に移す
                    else:                                  #見張りの位置が最初
                                                             の場所では無い時
                        if dis[p] >= dis[0]:        #見張りの現在の距
                                                      離が初期距離よりも小さい時
                            pre.append(int(cList.num+0.5)) #この塔番号をリスト
                                                             に追加
                            cList = cList.next             #探索の対象の塔を
                                                               次の塔に移す
                            break                          #ループ終了
                        else:                              #上記条件以外の時
```

```
                        cList = cList.next              #探索の対象の塔を
                                                          次の塔に移す
        pre.pop()                                       #pre リストから最後
                                                          の塔を削除
        if min(dis) == pointer and len(pre) == n:        #見張り人が n 人配
置されているかつ見張り人が担当する塔の距離の最小値が目標距離である時
            Max = pointer                       #Max を更新
            max_list = copy.copy(pre)           #条件を満たす塔の配置の記憶
            predis = (Max+ng)//2                #次の目標値の設定
            pointer = predis                    #新しい目標を設定
            fail = 1                            #失敗フラグを立てる
        else:                                 #見張り人の配置が条件を満たしていないとき
            if pointer+1 == ng:                 #探索の範囲設定
                pointer = Max+1                 #探索範囲の拡張
                fail = 1                        #失敗フラグが立っている時
            elif pointer+1 == predis:           #探索が終了できる時
                success = 1                     #成功フラグが立っている時
            else:                               #その他の時
                pointer += 1                    #次の探索ステップへすすむ
                fail = 1                        #失敗フラグをたてる

def main():                           #main 関数の定義
    global tlist,Max,max_list, d    #グローバル変数の設定
    d = [ 8, 14, 13, 7, 15, 12, 16, 15, 12, 9, 20, 22, 15, 12, 11, 25,
13, 7, 10, 6, 11, 11, 5, 8, 16, 7, 11, 18, 12, 9]  #d2
    n = int(input("配置する見張りの人数 n = "))          #見張り人の数を入力
    while n<1 or n>len(d):                      #見張り人の数が適切では無い時
        if n<1:                                 #見張り人が 0 人以下の時
            print("1 以上の自然数を入力してください。")      #エラーメッセージの出力
            n = int(input("配置する見張りの人数 n = "))     #もう一度見張
                                                          り人の数を聞く
        elif n>len(d):                          #見張りの人数が塔の数よりも多い時
            print("配置する見張りの人数が塔の数よりも多いです。")  #エラーメッセージ
                                                          の出力
            n = int(input("配置する見張りの人数 n = "))          #もう一度見張
                                                          り人の数を聞く
```

```
    start = time.time()                                  #二分探索に要する時間計測
    tlist = Link(Node(1.5,d[0]))                         #リンクリストの先頭ノードを作成
    cList = tlist.head                                    #現在のノード、cList をリ
                                                          ンクリストの先頭ノードに設定
    for i in range(2,len(d)+1):                          #距離リスト d の長さだけ繰り返し、
                                                          してリンクリストに追加
        cList.next = Node(i+0.5,d[i-1])                   #ノードの生成
        cList = cList.next                                #次の塔にうつす
    cList.next = tlist.head                               #循環リストの作成
    twosearch(n)                                          #二分探索関数の呼び出し
    for i in range(len(max_list)):                       #maxlist の長さ分だけ処理の繰り返し
        if max_list[i] > len(d):                         #maxlist の要素が塔の数より大きい場合
            max_list[i] = max_list[i] - len(d)            #その要素の修正
    max_list.sort                                         #要素を小さい順に並び替える
    print("見張りを配置する塔は", end='')                  #見張りを配置する塔の出力
    for l in max_list:                                    #maxlist の中身
        print("、{}番目".format(l), end='')                #見張りの番号を出力
    print()                                               #改行
    print("この組み合わせでの塔間の距離の最小値は {}".format(int(Max)))
                                                                  #最小距離の出力
    print()                                                       #改行
    end = time.time()                                     #時間測定終了
    print("処理時間: {}秒".format(end-start))              #要した時間の出力
```

（５）考察

⑸-1. 線形探索について

線形探索について、第 12 回 基本課題 1.「全探索」（ ex12_1.py ）と基本課題 2.「線形探索」（ ex12_2.py ）を以下の 2 つの条件でそれぞれ実行した。

・データは d2（ 塔の数 30 ）

・n = 8（ 8 人の見張りを置く ）

実行結果は全探索で以下のようになり、実行時間は 8 分 12 秒であった。

また、線形探索の実行結果は以下のようになり、その実行時間は 2 秒であった。

ここで結果を踏まえ、「全探索」（ex12_1.py）と「線形探索」（ex12_2.py）の 2 つのプログラムの処理の差についてアルゴリズムを比較して考察する。

初めに、全探索の場合と線形探索の場合における実行時間について考察する。基本課題 1 の全探索はその名前の通り、可能な組み合わせをすべて試すことで探索を行う。そのため、実行結果も数の小さい順に出力されている。組み合わせは 30 塔に 8 人を配置するため 30__C__8 の組み合わせがあり、計算量が非常に多くなると考えた。しかし、全探索ではすべての可能性の探索を行うため丁寧で確実な探索が可能となる。一方で、線形探索は 30 個の塔を探索していくため、線形探索の計算量は O(n)であることがわかる。よって、全探索は線形探索よりも計算量が多く、実行時間がより長いことが分かる。すなわち、全探索は時間をかけて丁寧に探索したいときに適している。線形探索はより素早く探索したいときに適している。

(5)-2. 2 分探索について

2 分探索について、第 12 回 基本課題 2.「線形探索」（ex12_2.py）と第 13 回 基本課題 1.「2 分探索」（ex13_1.py）を以下の条件で実行しなさい。

・データは d3（ 塔の数 150 ）および d4（ 塔の数が 150 で塔間隔が広い ）

・上記 d3 と d4 それぞれで n = 10（ 10 人の見張りを置く ）と n = 100（ 100 人の見張りを置く ）

初めに線形探索の実行結果をそれぞれ下に示す。

d3 の時（実行時間計 10 秒）

d4 の時（実行時間計 1 分 57 秒）

次に 2 分探索の時の実行結果をそれぞれ下に示す。

d3 の時（実行時間計 6 秒）

d4 の時（実行時間計 3 分 35 秒）

上記の 4 つのパターンについて、「線形探索」(ex12_2.py) と「 2 分探索」(ex13_1.py) の 2 つのプログラムの処理の差について
アルゴリズムを比較して考察した結果を d3 と d4 のデータの差異および n の差異を具体的に挙げながら比較して述べなさい。
まず、d3 と d4 のデータの差異について考察する。塔の間隔が狭い d3 においては線形探索も 2 分探索もどちらも比較的短い時間で処理をすることができた。しかし時間差では 2 分探索の方は実行時間が短く、より優れていると言える。塔間隔の距離が長い d4 では、実行時間は線形探索、2 分探索共に比較的長くなっている。ここで効率の良いとされる 2 分探索の方は実行時間が長い。これは二分探索の不得意分野であると言える。
次に n の差異の影響について考察する。n=10 の時のように見張りの数が少ない時、比較的実行時間は短い。どちらかと言えば 2 分探索の方が効率的であることが分かる。また、n = 100 のように見張り人の数が多い時は上に示したように計算量は線形探索で O(n)、2 分探索で O(nlogn)より、二分探索の方が適していると言える。

まとめとして、線形探索においては上記で述べた通り、順番にひとつずつの塔を探索するため、その計算量は O(n)となる。ここで 2 分探索は探索範囲を 2 分割し、探索を行うアルゴリズムであるため、その計算量は O(nlogn)となる。したがって、2 探索は最も効率の高いアルゴリズムであることがわかる。しかしながら、塔間隔の広い d4 のケースを見ると 2 分探索の実行時間は決して短くなく、線形探索の方が適していると言える。つまり、本プログラムでは塔間隔などその他の条件によって、どの探索方法が最適なのかは異なる。

次に、本プログラムで採用したアルゴリズムに関して、改善の余地はあったかという点について考察する。本プログラムでは見張り人の人数を手入力で入力することでプログラムが作動した。ここで、その入力した見張り人の人数が負の数や少数値であった時に第 12 回のプログラムではエラーが発生してしまうという作りになっていた。これは汎用性が低く改善するべきアルゴリズムであった。そのため、第 13 回の方ではその点を改善し、より汎用性の高いプログラムとした。