アルゴリズム入門　＃12

地引　昌弘

2024.12.26

はじめに

これまで、ループ処理と再帰処理を題材に、アルゴリズムの性能を規定する計算量について学んで来ました。今回は、そこで得られた知見を利用し、様々な問題を効率良く計算できる動的計画法について説明します。動的計画法は、ある問題に対して、その問題を直接解く代わりに小さい問題から順に解いて行き、その都度記録した結果を用いて最終的に必要な解を求める手法です。

1　前回の演習問題の解説

1.1　演習 11-1a ～ 11-1h: — 様々なメソッドの計算量

各計算量を一覧しておきます:

$$\text{a: } O(1),\ \text{b: } O(n),\ \text{c: } O(n^2),\ \text{d: } O(n),\ \text{e: } O(1),\ \text{f: } O(\log n),\ \text{g: } O(3^n),\ \text{h: } O(n!)$$

以下、演習 11-1f と演習 11-1h について、少し説明します。

演習 11-1f

演習 11-1f では、指数 n を 2 で割った余りに応じて n の値を変更し、その後再帰呼び出しをしています。このような場合に再帰呼び出しの段数を見積もるには、各再帰呼び出し毎に、指数 n がどのくらい小さくなるかを考えます (これは、#11 で取り上げた、引き算により最大公約数を求める計算回数の見積もり等と同じ考え方です)。ここでは、指数 n を二進数として扱ってみましょう。

このアルゴリズムでは、$n \,\%\, 2 = 0$ の場合に指数 n を 2 で割っています (`near1pow3(n)` → `near1pow3(n/2)`)。二進数を 2 で割ることは、1 桁右へ (小さい方へ) ずらすことに該当し、この時、割る前の 1 桁目の値と剰余は等しくなります。例えば、二進数 $1001_{(2)}$ (十進数 $9_{(10)}$) に対する各回の計算は、次のようになります (以下では、十進数における $_{(10)}$ を省略):

- 1 回目の割り算: $1001_{(2)} \div 2 = 100_{(2)} = 4 \cdots 1$ (剰余)
- 2 回目の割り算: $100_{(2)} \div 2 = 10_{(2)} = 2 \cdots 0$ (剰余)
- 3 回目の割り算: $10_{(2)} \div 2 = 1_{(2)} = 1 \cdots 0$ (剰余)
- 4 回目の割り算: $1_{(2)} \div 2 = 0_{(2)} = 0 \cdots 1$ (剰余)

これより、$n \,\%\, 2 = 0$ となり `near1pow3(n/2)` が呼び出される回数は、指数 n を二進数で表わした場合の “0” の個数と等しく、高々$\log n$ 回であることが分かります。

次に、$n \,\%\, 2 > 0$ の場合は指数 n を 1 だけ減らしています (`near1pow3(n)` → `near1pow3(n-1)`)。この時、指数 n はほとんど減りませんが、次回の計算 (再帰呼び出し先での計算) では、n が奇数から偶数に変わるため、必ず $n \,\%\, 2 = 0$ となります。つまり、$n \,\%\, 2 > 0$ となる回数は、$n \,\%\, 2 = 0$ となる回数と高々同じというわけです。これより、`near1pow3(n-1)` が呼び出される回数は、高々$\log n$ 回だと言えます。

以上より、`near1pow3(n/2)` が呼び出される回数も、`near1pow3(n-1)` が呼び出される回数も高々$\log n$ 回なので、その計算量は $O(\log n)$ と見積もることができます。

演習 11-1h

演習 11-1h は、少し注意が必要です (これまでのような、指数ではありません)。このプログラムでは、順列の計算を perm1 関数で行なっています (perm 関数は、順列の計算に使う 1 〜 n までの数字を入れた配列を用意しているだけです)。perm1 関数では、まず n 個の数値を調べて順列の 1 桁目を決めた後、最初の再帰で $n-1$ 個の分岐が発生します。次に、2 段目の再帰にて順列の 2 桁目が決まった後は、各 $n-1$ 個の再帰呼び出しそれぞれにつき、$n-2$ 個の分岐が発生します。順列の最後の桁が決まるまで同様に考えると、その計算量は $n \times (n-1) \times \cdots \times 1$ より $O(n!)$ となります。一般に、$n \to \infty$ の時、$a^n \ll n!$ となるため (詳細は、#6 脚注 1 を参照)、$O(n!)$ はあらゆるアルゴリズムの中で最も多い計算量の一つとなります (言い換えれば、最も遅いアルゴリズムの一つ)。

参考までに、答えを求めるために $O(n!)$ の計算量が必要な問題として、「巡回セールスマン問題」があります。これは、都市の集合と各 2 都市間の移動コスト (例えば距離) が与えられた時、最小の移動コストで全ての都市をちょうど 1 回ずつ訪れて出発地へ戻る経路を求める問題です。セールスマンが、最安の運賃で各顧客を 1 度ずつ訪問し、本社へ戻る道筋を求める問題と同じだとして、このような題名を付けられています。

1.2 演習 11-2 — ユークリッドの互除法の計算量

引き算の代わりに剰余演算を用いるユークリッドの互除法により、最大公約数を求める Python プログラムを示します:

```
def gcd_div(x, y):
  while True:
    if x > y:
      x = x % y
      if x == 0: return(y)
    else:
      y = y % x
      if y == 0: return(x)
  return
```

この計算量については、前回説明した引き算により最大公約数を求めるアルゴリズムと同様な考え方で見積もることができます。引き算アルゴリズムによる計算量の見積もりでは、毎回の引き算により x, y がどの程度小さくなるかに着目しました。ユークリッドの互除法では、x, y がその剰余に置き換わる時点で、どの程度小さくなるかに着目します。

まずは、$a = b \cdot q + r_1$ なる関係式を考えます。ユークリッドの互除法により、これは $b = r_1 \cdot q' + r_2$ という関係式に変わります (b を r_1 で割った余りが r_2 なので、当然 $r_1 > r_2$ であることに注意して下さい)。同様に、その次の関係式は $r_1 = r_2 \cdot q'' + r_3$ となります。これを一般化し、ユークリッドの互除法による関係式を、以下のように定義します。

$$r_i = r_{i+1} \cdot q_i + r_{i+2}, \quad r_i > r_{i+1}$$

ここで、r_i と r_{i+2} のより詳細な大小関係を考えます。

i). $r_i/2 \geq r_{i+1}$ の場合:
$r_i > r_{i+1},\ r_{i+1} > r_{i+2}$ より、$r_i/2 \geq r_{i+1} > r_{i+2}$ が成り立つ。

ii). $r_i/2 < r_{i+1}$ の場合:
この不等式を $r_i - 2r_{i+1} < 0$ と変形し、両辺に $r_i\ (> 0)$ を加えると、$2r_i - 2r_{i+1} < r_i$ より $r_i - r_{i+1} < r_i/2$ となる。この関係を図 1 に示す。これより、$r_i = r_{i+1} \cdot q_i + r_{i+2}$ を満たす q_i は 1 以外存在しない。よって、$r_i = r_{i+1} + r_{i+2}$ より、$r_{i+2} = r_i - r_{i+1} < r_i/2$ が成り立つ。

以上より、ユークリッドの互除法では、必ず $r_i/2 > r_{i+2}$ が成り立ちます。添字番号に注意すると、2 回の計算毎に半減して行くことが分かります。つまり、高々$2\log n$ 回の計算で剰余は 0 になるわけです。

従って、ユークリッドの互除法により最大公約数を求めるアルゴリズムの計算量は、$O(\log n)$ となります。ところで、引き算を用いたアルゴリズムでは、2 数の組み合わせに、最善の場合やバランスの良い場合 (効率的に計算できる場合)、あるいは最悪な場合などが存在し、その都度計算量に大きな変化がありました。しかし、剰余演算を用いたアルゴリズ

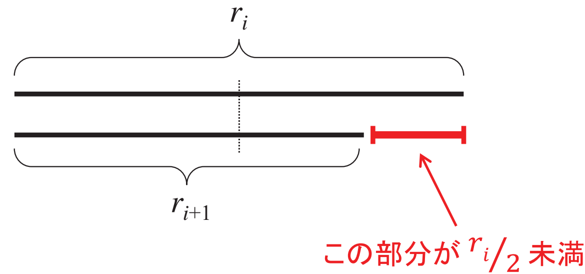

図 1: ユークリッドの互除法における各項の大小関係

ムでは、上で述べたように、2 数の組み合わせに特に条件があるわけではありません。つまり、どのような 2 数の組み合わせにおいても効率的な計算を行なえるため、最大公約数を求めるには、引き算を用いるアルゴリズムより、剰余を用いるアルゴリズムの方が効率が良いと言えます[1]。

最後に、繰り返しの制御構造を含むアルゴリズムは、再帰処理 · ループ処理の二つに大きく分類できます。再帰処理は、アルゴリズムを見通し良く記述できますが、再帰呼び出しが指数的に増加する場合は、対象データの渡し方などにそれに見合った配慮がないと (要は処理の軽減がないと)、計算量も指数的に増えてしまいます。再帰処理の一般的な計算量は、c: 関数の計算量, d: 分岐数, g: 世代数 とすると、$O\left(\sum_g (c \cdot d^g)\right)$ になります。これより、各世代の総計算量を $c \cdot d^g = f(g)$ とした時、$g \to \infty \Rightarrow f(g) \to 0$ にできるか、ということです。この場合、再帰処理全体の計算量は、"各世代毎の計算量 × 世代数" $= O(f(g) \cdot g) =$ "緩やかな増加" となります (**ここでは世代数を基本に考えましたが、世代数 ↔ データ数等の置き換えに慣れること!!**)。これに対しループ処理は、(パスカルの三角形のような) 非効率なアルゴリズムでも、通常は計算量の増加を多項式で抑えられるため (n 回のループが i 個重なったとしても $O(n^i)$)、再帰処理より速い場合があります。

2 動的計画法

2.1 動的計画法とは

フィボナッチ数を求める場合、再帰的定義に沿ってそのまま計算してしまうと計算量が指数的に増大し、効率的な計算はできませんでした。そのため、フィボナッチ数の計算では、ループ処理を利用して効率的に計算する方法を検討しました。ここで、両者の本質的な違いを改めて確認してみましょう。

フィボナッチ数の定義:

$$fib(n) = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ fib(n-1) + fib(n-2) & (\text{otherwise}) \end{cases}$$

```
# 再帰的定義によるプログラム
def fib(n):
  if   n < 1:  return(0)     # 不等号により、n に負数を渡された場合まで対応
  elif n == 1: return(1)
  else:        return(fib(n-1) + fib(n-2))

# ループ処理を用いたプログラム
def fib_loop(n):
  x0 = 1; x1 = 0             # x1 が fib(n) になるよう x0 の初期値を調整していることに注意
  for i in range(n):
    t = x0 + x1
    x0 = x1
    x1 = t
  return(x1)
```

[1]余談ですが、最初にユークリッドが考案した方法は、引き算を使う方法だったという説があります。

どちらのプログラムもフィボナッチ数の定義に従った足し算をしていますが、両者が $fib(n-1)$ と $fib(n-2)$ を用意する方法をあらためて見比べてみると、次に述べる本質的な違いのあることが分かります。つまり、再帰的定義によるプログラムでは、毎回 $fib(n-1)$ と $fib(n-2)$ を計算しているのに対し、ループ処理を用いたプログラムでは、前回の $fib(n-1)$ と $fib(n-2)$ を覚えておき計算していない、という点です。

これをもう少し汎用化し、(前回紹介した組合せ数の計算で「パスカルの三角形」を用いる意義として触れたように) フィボナッチ数の計算に際し、配列 `fib[i]` を用意して計算した値はそこに蓄えておく、という方法を考えることができます。例えば、配列を用いて最大 30 番目までのフィボナッチ数を順番に計算する Python プログラム (の一部) は、次のようになります:

```
fib = ita.array.make1d(31); fib[0] = 0; fib[1] = 1
for i in range(2, 31): fib[i] = fib[i-1] + fib[i-2]
```

このコードは、パスカルの三角形と同様、同じプログラム内で様々なフィボナッチ数を計算する場合に、該当する配列の要素を参照するだけで済むので、遥かに効率化できます。

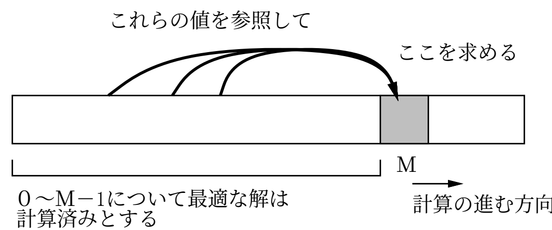

図 2: 動的計画法の考え方

一般に、関数を計算する際に一度計算した結果を引数と一緒に覚えておき、同じ引数に対しては覚えておいた値を返すようにすることを**メモ化** (Memoization) と呼びます。この手法では、図 2 のような「値 $0 \sim M-1$ までの解が分かっていれば、値 M についての解を求められる」問題に対し、配列を用いて 0 から順に解を記録して行くことで値 M の解を求めます。このような、ある問題に対してその問題を直接解く代わりに小さい問題から順に解いて行き、その都度記録した結果を用いて最終的に必要な結果を求める手法のことを、**動的計画法** (Dynamic Programming) と呼びます[2]。動的計画法の狙いは、再帰処理の良い部分 (手続きを簡潔に表現できる) とループ所の良い部分 (得られた結果を再利用できる) を組み合わせることで、問題の効率的な解決を図ることです。他の方法では計算量が多過ぎて扱えない問題が、動的計画法によって効率良く扱える場合は数多くあります[3]。

2.2 部屋割り問題 (1 次元配列を利用する例)

動的計画法の適用例として、次のような問題を考えてみます。

> 合宿で 1 泊料金が「1 人部屋: 5,000 円、3 人部屋: 12,000 円、7 人部屋: 20,000 円」というホテルに泊まることになった[4]。合計宿泊人数 n 人に対し、最も安い宿泊金額の総計を求めよ。

この問題では、7 人部屋が非常に割安なので、7 人より少ない人数で泊まっても 7 人部屋を選んだ方が良い場合があり、最適な割り当てを求めるのはそう簡単ではありません。この問題に限って言えば、できるだけ多くの人数を 7 人部屋に割り当て、残った 1〜6 人については、可能性のある全ての割り当てを検討すればできそうに見えます (これは素直な考え方ですね)。しかし、部屋の種類が増えると、それに応じて場合分けも多くなり、見通しの良い手順を作ることは困難になります (素直な考え方による手順が実は複雑になってしまう例は、次回にお見せします)。

そこで、動的計画法の適用を試みてみましょう。まずは、再帰的な関係を見出します。n 人を最安の部屋割りにした料金は、下記のうち最も小さい金額と考えることができます:

[2] 因みに、「動的」という文字が入っていますが、何か特別に動的な特徴を見い出している or 利用しているわけではありません。この手法を考案した人がそういう名前を付けた、というだけです。

[3] とは言え、動的計画法は強力な手法ですが、1) 状況に応じてメモ化されたデータを使い分ける、2) 同、後戻りが発生するような問題では、うまく適用できない場合があります。 → **この特徴は重要なのでよく覚えておいて下さい。**

[4] 参加者は全員同性とします (まあ、お約束ということで)。各部屋には収容人数より少ない人数で泊まっても構いません。また、どの部屋も数は十分あるものとします。

- “1 人部屋に 1 人を割り当てた 5,000 円” + “残りの $n-1$ 人を最安の部屋割りにした料金”
- “3 人部屋に 3 人を割り当てた 12,000 円” + “残りの $n-3$ 人を最安の部屋割りにした料金”
- “7 人部屋に 7 人を割り当てた 20,000 円” + “残りの $n-7$ 人を最安の部屋割りにした料金”

一部屋に 2 人の割り当てや 5 人の割り当てを考えない理由は、各部屋に最大人数まで割り当てた方が効率的だから (つまり、一番安くできるから) です。これをもとに、形式的な再帰的定義を考えます。人数 n に対して最も安い値段を計算する関数 $room_price$ は、次のように定義されます:

$$room_price(n) = min \begin{cases} room_price(n-1) + 5000 \\ room_price(n-3) + 12000 \\ room_price(n-7) + 20000 \end{cases}$$

min と言う関数は聞いたことがないと思いますが、“右側の選択肢のうち一番小さい値を取る” という意味だと解釈して下さい。$room_price(n)$ は、$n \leq 0$ の時は 0 とします (これ以上少ない宿泊者は存在しないので、宿泊費も 0 円という意味)。

以上を参考に、$room_price(n)$ を計算する時は、1) 既に $room_price(n-1)$ ほか二つの計算を終えており、2) 各結果が記録されているはずなのでそれらを利用する、という方針に沿って Python のプログラムを作成してみましょう。

対応関係 (漸化式 · 再帰呼び出し · 配列 · 配列への処理) を理解すること

まずは、各人数毎の宿泊費を保持する配列 `room_price` を用意します (i 人が宿泊した際の宿泊費は `room_price[i]` に入ります)。これが、関数 $room_price$ に該当します。その後、1 〜 n までの `i` について、上記三つの場合における値を順番に比較し、最小値を配列 `room_price` に記録して行きます。この動作が、関数 $room_price$ を再帰的に呼び出して行くことに該当します。例えば、再帰的に $room_price(i-3)$ を呼び出す代わりに、既に結果が記録されている `room_price[i-3]` を参照するというわけです。

```
!pip install ita                       # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def room(n):
  room_price = ita.array.make1d(n+8)    # 配列のサイズに注意
  for i in range(len(room_price)): room_price[i] = 0

  # 過去の計算結果を利用し、人数が増える度にどの部屋割りが最安かを調べて行く。
  for i in range(1, n+1):
    min = room_price[i-1] + 5000
    if min > room_price[i-3] + 12000: min = room_price[i-3] + 12000
    if min > room_price[i-7] + 20000: min = room_price[i-7] + 20000
    room_price[i] = min
  return(room_price[n])
```

これを見ると、`i` が小さい時に、`room_price[-1]` とか `room_price[-2]` を参照してしまうようなコードになっていますね。実は、Python では配列の添字がマイナスの場合、ぐるっと回って「末尾から何番目」を参照します。そのため、配列のサイズを余分に取っておき、最初は全部 0 を入れるようにしています (これにより、`room_price[-1]` などは全て末尾の方にある 0 が表示される)。

では動かしてみましょう:

```
>>> room(10)
32000
>>> room(11)
37000
>>> room(12)
40000
```

なるほど、12 人だと 7 人部屋二つの方が安いようですね。

ところで実用を考えると、宿泊費だけではなく、何人部屋を幾つ使っているかも知りたいですよね? それには、人数が増えるにつれ (つまり、**動的計画を進めるにつれ**)、選択した部屋も記録して行く必要があります。これを記録するために、次の関数 $room_sel(n)$ を定義します:

$$room_sel(n) = \begin{cases} 1 & (\text{if } room_price(n-1) + 5000 \text{ is the smallest}) \\ 3 & (\text{if } room_price(n-3) + 12000 \text{ is the smallest}) \\ 7 & (\text{if } room_price(n-7) + 20000 \text{ is the smallest}) \end{cases}$$

この関数は、上で定義した関数 $room_price(n)$ が、三つの選択肢から最安としてどの部屋を選んだかを返します。例えば、“3 人部屋に 3 人を割り当てた 12,000 円”+“残りの $n-3$ 人を最安の部屋割りにした料金” が最安だった場合は、3 を返します。これは、宿泊者が i 人の時点で選択した部屋を意味することに注意して下さい (最安部屋割りの再帰的関係をもう一度確認しましょう)。先ほどと同様、プログラムでは、各人数毎に選んだ部屋のリストを保持する配列 `room_sel` を用意します (宿泊者が i 人になった時、選んだ部屋が `room_sel[i]` に入るわけです)。これが、関数 $room_sel$ に該当します。選んだ部屋のリストを得るには、例えば `room_sel[12]` が 7 だった場合、その次は `room_sel[5]` を調べます。これは、下記を意味します (‡1)。

- 宿泊者が 12 人の時、`room_sel[12]` を調べてみると、“7 人部屋に 7 人を割り当てた 20,000 円”+“残りの $n-7$ 人を最安の部屋割りにした料金” が最安であり、7 人部屋を選択していた (言い換えれば、最後に選択した部屋は 7 人部屋だった)。
- 残りは $12-7=5$ 人なので、次は `room_sel[5]` に記録されている部屋を調べればよい (そこに入っている部屋が最安の部屋として記録されているはず)。

以下、順次 `room_sel` を「逆向きに」辿って行きます。このような情報のことを「トレース バック情報」と呼ぶことがあります。トレース バック情報の意義は、どのような処理の経緯を経て、その結果に至ったかの記録です。この例では、上記 ‡1 で述べたような方法で記録を取りましたが、正しく記録を辿れるのであれば、これ以外の方法でも構いません。では、先の `room` 関数を改造してトレース バック情報を記録し、金額に続いて部屋のリストを (一つの配列として) 並べて返すようにしてみましょう:

```
def room_1(n):
  room_price = ita.array.make1d(n+8)
  for i in range(len(room_price)): room_price[i] = 0
  room_sel = ita.array.make1d(n+1)
  for i in range(len(room_sel)): room_sel[i] = 0

  # 過去の計算結果を利用し、人数が増える度にどの部屋割りが最安かを調べて行く。
  for i in range(1, n+1):
    min = room_price[i-1] + 5000; s = 1
    if min > room_price[i-3] + 12000: min = room_price[i-3] + 12000; s = 3
    if min > room_price[i-7] + 20000: min = room_price[i-7] + 20000; s = 7
    room_price[i] = min
    room_sel[i] = s          # i 人が宿泊する時に、最後に選択した部屋 (の人数) を記録して行く。

  # 最安の宿泊費および選択した部屋を逆向きに辿り、配列に入れて行く (整形した表示にするため)
  a = [room_price[n]]        # 初項は最安の宿泊費 (x=[y] →要素 1 個の配列を作成)
  i = len(room_sel) - 1
  while i > 0:
    a.insert(len(a), room_sel[i])
    i = i - room_sel[i]      # トレース バック
  print(a)
```

これを動かすと、今度は選択された部屋の一覧も表示されます:

```
>>> room_1(10)
[32000, 3, 7]
>>> room_1(11)
[37000, 1, 3, 7]
>>> room_1(12)
[40000, 7, 7]
```

「部屋割り問題」と同様、「釣り銭問題」も動的計画法が使える典型的な問題です。例えば、米国ではコインの額面が「1¢」「 5¢」「 10¢」「 25¢」の 4 種類あります。ここで、ある金額 (¢) を示された場合、その金額と等しい最低枚数のコインを決めるのは少々面倒ですね。これも、次のように考えると動的計画法で解けます (但し、$coins(0) = 0$ と定義します):

$$coins(n) = min \begin{cases} coins(n-1)+1 & (n \geq 1) \\ coins(n-5)+1 & (n \geq 5) \\ coins(n-10)+1 & (n \geq 10) \\ coins(n-25)+1 & (n \geq 25) \end{cases}$$

2.3 経路数問題 (2 次元配列を利用する例)

ここまでは 1 次元の配列を用いる動的計画法でしたが、問題の構造によっては、2 次元以上の配列を用いることになります。その簡単な例として、経路数を数える問題を考えてみましょう。

問題: 図 3 の m×n セルにある S の位置から、「右方向と下方向にだけ」移動して G の位置へ至る経路は、合わせて何通りあるか。左側 (障害物なし) と右側 (網掛けのセルは通れない) のそれぞれについて答えよ。また、「斜め右下」への移動が許される場合はどうか。

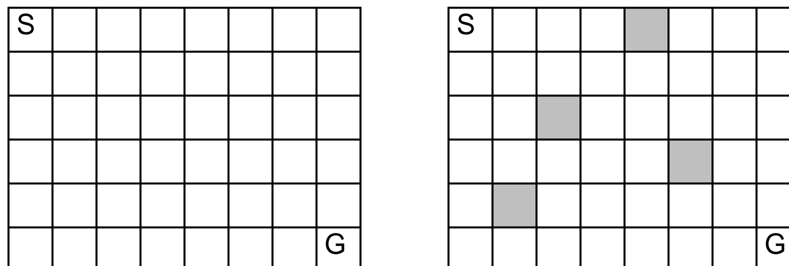

図 3: ゴールまでの経路の数は?

まずは、斜めの移動がない場合です。この場合、あるセル X に至るには、上のセルから来るか、左のセルから来るかのどちらかです。そして、両者は同じ経路ではありません。よって、セル X に至る経路が何通りあるかは、「(X の) 上のセルに至る経路が何通りあるか」と「(X の) 左のセルに至る経路が何通りあるか」を足したものになります。各セルに至る経路数は、図 4 (次ページ) のように、左上端から順番に数えて埋めていけば計算できますね。最左列 · 最上行については、1 通りしか行き方がないのは明らかなので、最初から埋めておくことができます。また、途中に障害物がある場合は、そこへ行けないので、そのセルに至る経路数は 0 として扱います。

参考までに、図 4 の左側については、次のように考えることができます。例えば、左上端のセルから右下端のセルへ至る経路は、縦 7 列および横 5 行のセルによる組み合わせなので、${}_{12}C_5$ or ${}_{12}C_7 = 792$ となります。実に数学的で美しい解法ですが (私もこのような解法は大好きです)、障害物のある場合はその都度個別に考える必要があり、また斜めを許す場合も面倒です。数学では、このような条件が入る問題はあまり好まないでしょう[5]。これに対し、動的計画法で解くことは、手で埋めて行く作業と似ていますね。実は、数学的アルゴリズムと人間の (知的) 作業とは、対応関係を作ることができます。この辺りを深く研究されたのが、クルト・ゲーデル (Kurt Gödel) やアラン・チューリング (Alan Turing)

[5] この辺りの事情に興味を抱いた人は、是非、四色問題についての書籍を読んでみて下さい。面白い本が幾つかあります。

といった名高い研究者達です。これらの研究を通じて、“公理に基づく演繹では真偽を判断できない問題が存在する”、あるいは “与えられたプログラムが有限時間に停止するかどうかを判断できるプログラムは存在しない” (後者は #1 で紹介しましたね) といった知見が導かれました[6]。

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792

1	1	1	1		0	0	0
1	2	3	4	4	4	4	4
1	3		4	8	12	16	20
1	4	4	8	16		16	36
1		4	12	28	28	44	80
1	1	5	17	45	73	117	197

図 4: ゴールまでの経路の数を求める

さて、余談はこのくらいにしておき、経路数を求める Python プログラムを見てみましょう。まずは障害物のない場合です:

```
import pprint

def paths1():
  size_i = 6; size_j = 8
  a = ita.array.make2d(size_i, size_j)
  for i in range(size_i): a[i][0] = 1
  for j in range(size_j): a[0][j] = 1

  # 上セルおよび左セルに至る経路数を用いて、各セルに至る経路数を調べて行く。
  for i in range(1, size_i):
    for j in range(1, size_j):
      a[i][j] = a[i-1][j] + a[i][j-1]
  pprint.pprint(a, width=40)
  return(a[size_i-1][size_j-1])
```

このプログラムでは、各セルへ至る経路数を 2 次元配列 `a` に保持します。但し、2 次元配列 `a[i][j]` は、i 行 · j 列のセルに対応することに注意しましょう (座標系とは違います/以前にも述べましたが、混乱しやすいのでしっかりと確認して下さい)。今回は、最終的な経路数を整理された形で確認したいため、`pprint.pprint` 関数を利用しました。「最左列 · 最上行」にあるセルついては、そこへ至る経路が「上から · 左から」しかないので、2 次元配列の初期値として 1 を入れておきます。実行結果は下記の通り:

[6] 以下、興味がある人のために少し補足しておきます。#7 (14 ページ以降) では、1 次元セル オートマトンのあるパターンが、実はプログラミング言語と同等であるという話題を紹介しました。この時、計算可能なアルゴリズムであれば必ずプログラミングできる簡潔なシステムとして、チューリング機械をお見せしましたが、これはチューリングが考案したアイデアです。また、ゲーデルが示した「計算可能な全ての関数は、極めて簡潔な原始基本関数 (定数関数 · 後者関数 · 射影関数) とその合成および原始帰納法により定義できる」という原始帰納的関数の考え方についても触れました (実は後世、これ以外の再帰的関数が発見されたのですが)。これにより、関数が計算可能であれば、簡潔な関数および手順の組み合わせとして表現できるため、必ずプログラムとして記述できることが分かります (ここまでが、#7 で取り上げた内容でした)。但し、もう少し厳密に言うと、これは “原始帰納的に計算可能” $\Rightarrow$ “プログラムとして計算可能” を意味しており、その逆 “プログラムとして計算可能” $\Rightarrow$ “原始帰納的に計算可能” が言えたわけではありません (先ほど述べたように、原始帰納的に定義できない計算可能な関数が存在します)。では、「プログラムとして計算できない関数」とは一体何でしょうか。“関数の計算は必ず何らかの手順に従う” かつ “プログラムは手順を記述したもの” なので、プログラムとして計算できない関数 f とは、1) “手順を記述できない” $\Rightarrow$ “手順が存在しない” $\Rightarrow$ “関数 f は計算できない”, 2) “結果が分からない” $\Rightarrow$ “関数 f の計算が終わらない” $\Rightarrow$ “事実上、関数 f は計算できない” のどちらかになるはずです。つまり、プログラムとして計算できない関数とは、そもそも計算できない関数だというわけです。これより、現在では、“関数が計算可能” であることと “プログラムとして計算可能” であることは同値である (つまり、プログラムとして計算できるもの、言い換えれば、チューリング機械上で計算できるものを、計算できるものと定義しよう)、と提唱されるようになりました (提唱であって、定理や公理ではありません…ちょっと気持ち悪いですか?)。

```
>>> paths1()
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 2, 3, 4, 5, 6, 7, 8],
 [1, 3, 6, 10, 15, 21, 28, 36],
 [1, 4, 10, 20, 35, 56, 84, 120],
 [1, 5, 15, 35, 70, 126, 210, 330],
 [1, 6, 21, 56, 126, 252, 462, 792]]
792
>>>
```

障害物がある場合は少々面倒ですが、例えば障害物があるセルには事前に -1 を入れておき、経路数が負数ならば、そのセルは使えないセルとして値を更新せず、かつそのセルの値も他へ加えない、と考えることができます (配列 a の作成では内包表記を使いました):

```
def paths2():
  size_i = 6; size_j = 8
  a = [[0 for j in range(size_j)] for i in range(size_i)]
  a[0][4] = a[2][2] = a[3][5] = a[4][1] = -1     # まず最初に障害物のあるセルへ -1 を入れる
  for i in range(size_i):                        # 最左列の初期化
    if a[i][0] < 0: break                        # 障害物があったら、それより下へは行けない
    else: a[i][0] = 1
  for j in range(size_j):                        # 最上行の初期化
    if a[0][j] < 0: break                        # 障害物があったら、それより右へは行けない
    else: a[0][j] = 1

  # 上セルおよび左セルに至る経路数を用いて、各セルに至る経路数を調べて行く
  for i in range(1, size_i):
    for j in range(1, size_j):
      if a[i][j] < 0: continue                   # 障害物のセルは何もしない
      if a[i-1][j] > 0: a[i][j] += a[i-1][j]     # 上セルが障害物かを調べてから経路数を追加
      if a[i][j-1] > 0: a[i][j] += a[i][j-1]     # 左セルが障害物かを調べてから経路数を追加
  pprint.pprint(a, width=40)
  return(a[size_i-1][size_j-1])
```

2 次元配列の初期化では、障害物があるセルから先 (そのセルから右側 · 下側にある各セル) の経路数を 0 にしておく必要があるので、最初は全て 0 を入れています (今回は内包表記を使ってみました)。次に、障害があるセルへ -1 を入れます。その後、最左列 · 最上行のセルへ 1 を入れますが、-1 のセルを発見したら `break` 命令によりそこで止める (以後の経路数は更新しない) ようにしています。また、各セル毎の経路計算 (動的計画の本体) では、障害物があるセルに出会うと、`continue` 命令でそのセルの計算を取り止めています[7]。こちらも確認のため実行結果を示しておきます:

```
>>> paths2()
[[1, 1, 1, 1, -1, 0, 0, 0],
 [1, 2, 3, 4, 4, 4, 4, 4],
 [1, 3, -1, 4, 8, 12, 16, 20],
 [1, 4, 4, 8, 16, -1, 16, 36],
 [1, -1, 4, 12, 28, 28, 44, 80],
 [1, 1, 5, 17, 45, 73, 117, 197]]
197
>>>
```

[7] `break` は、そのループ回における以後の処理を取り止め、“ループを終了する” 命令です。`continue` も、そのループ回における以後の処理を取り止めますが、ループ自体は終了せず “次のループ回へ移る” 命令です。これらに対し、`pass` は、何もせず “単に次の処理へ移る” 命令です。各者の違いをよく理解しておきましょう。

最後に、経路数の数え上げより少し複雑な問題として、障害物のあるセルを避けながら、できるだけ大きい正方形を配置する問題を考えてみましょう (図 5)。正方形の最大辺数 (1 辺のセル数) と、その大きさの正方形を配置できる場所を求めます。

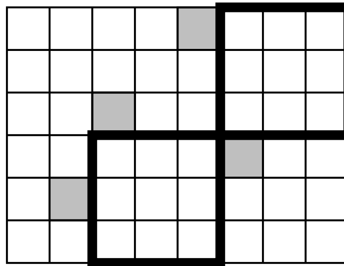

図 5: 最大正方形の配置を求める

この問題は、一見、どこから手を付ければ良いのか扱いに困りますが、例えば、「各セルに、自身が右下隅となる正方形の最大サイズを入れる」問題と定義し直せば、動的計画法を利用できます (図 6)。

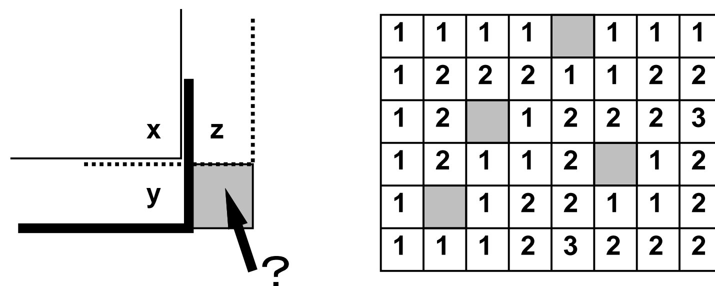

図 6: 最大正方形の動的計画法

この動的計画法の方針は、次のように考えることができます。

- セルに障害物がない場合、そのセルへは少なくとも「1」を入れることができる。
- セルに障害物がなく、かつ x · y · z のどれかに障害物があれば、そのセルへは明らかに「1」しか入れられない。
- セルに障害物がなく、かつ x · y · z のいずれにも障害物はない場合は?

まずは、図 6 の右側を参考に、その規則を考えてみて下さい。

∗ これまで様々なアルゴリズム技法について学んで来ましたが、上の問い掛けは、「そもそもどのような手順に直せば効率的に問題が解けるか」を考えることに該当し、アルゴリズム技法云々より前に (要は一番最初に) 考えるべきことです。ここで、うまい手順を考え付くことができれば、後は最適なアルゴリズム技法を適用することで、さらに効率化が図れます[8]。

∗ またその逆として、プログラムを見せられた時、「問題を効率的に解くために、どのような手順に直されたのか」を理解することも、同じく重要です。

是非、これらを見通せるようになって下さい。

[8]これに対し、アルゴリズム技法だけを駆使して、あまりうまくない手順を効率化することは、なかなか難しいものです (例えば、単純選択法をマージ ソートより速くすることは、極めて困難です)。

規則性を発見できましたか? 今回の規則は、x·y·z の 3 セルに入っている各値の最小値より、1 だけ大きい値が該当します。その理由を以下に示しておきます。

略証:

各セルに入る値を「自身が右下隅となる正方形の最大サイズ (一辺の長さ)」とする (‡2)。あるセル i に入る値を L_i とし、セル i の左上 · 左 · 上にある 3 セルをセル $x,\ y,\ z$ (図 6 左, セル $i =$ 灰色セル)、これらのセルに入る値を $L_x,\ L_y,\ L_z$ とする。また、$L_x,\ L_y,\ L_z$ のうち、最小の値を L_a、それ以外の値を L_b とする。各 L は整数なので、$L_b \geq L_a + 1$ あるいは $L_b = L_a$ となる。ここで、$L_a,\ L_b$ に該当するセルをセル $a,\ b$ とする (例えば、$L_a = L_y,\ L_b = \{L_x, L_z\}$ であれば、セル y がセル a、セル $x,\ z$ がセル b となる)。

1. $L_b \geq L_a + 1$ の場合:

まずは、L_i と L_b との関係について考える。セル i はセル $x,\ y,\ z$ と隣接しているので、セル i を右下隅とする最大正方形が、セル $x,\ y,\ z$ を右下隅とする各最大正方形のどれかより大きいならば、それは辺が 1 セル分だけ大きい正方形となる。つまり、L_i と $L_x,\ L_y,\ L_z$ の差は、それぞれ高々一つである (‡3)。ここで、セル i を右下隅とする最大正方形が、セル b を右下隅とする各最大正方形より大きいと仮定する。この時、$L_i = L_b + 1$ となり (1 セル分だけ大きいので +1)、$L_b \geq L_a + 1$ より $L_i \geq L_a + 2$ となるが、これは ‡3 に矛盾する。よって、セル i を右下隅とする最大正方形が、セル b を右下隅とする各最大正方形より大きくなることはない。

次に、L_i と L_a との関係について考える。以下では、セル y をセル a とする ($x,\ z$ の場合も同様な議論ができる)。$L_b(= L_x, L_z) - 1 \geq L_a(= L_y)$ なので (各 L は整数 $\rightarrow L_x, L_z$ は少なくとも L_a 以上)、‡2 よりセル x, z を右下隅とするサイズ L_a の正方形を作ることができる。つまり、セル i の上 · 左 · 左上にある 3 セルをそれぞれ右下隅とするサイズ L_a の正方形を三つ作ることができる。よって、セル i を右下隅とするサイズ $L_a + 1$ の正方形を作ることができる。

2. $L_b = L_a$ の場合:

(この場合は素直に) セル i の上 · 左 · 左上にある 3 セルをそれぞれ右下隅とするサイズ L_a の正方形を三つ作ることができる。よって、セル i を右下隅とするサイズ $L_a + 1$ の正方形を作ることができる。

発見した規則による Python のプログラムを以下に示します。障害物のあるセルには「−1」を入れてあります:

```
def squares():
  size_i = 6; size_j = 8

  # セルに障害物がない場合、少なくとも「1」を入れることができる。
  a = [[1 for j in range(size_j)] for i in range(size_i)]
  a[0][4] = a[2][2] = a[3][5] = a[4][1] = -1              # 障害物のあるセル

  # x/y/z の値を用いて、各セルが右下隅となる正方形の最大サイズを入れて行く
  for i in range(1, size_i):
    for j in range(1, size_j):
      if a[i][j] < 0:                                     # 障害物のあるセルは除外
        continue
      elif min([a[i-1][j-1], a[i-1][j], a[i][j-1]]) < 0:  # x/y/z のどれかが障害物の場合
        a[i][j] = 1
      else:                                               # x/y/z のどれも障害物でない場合
        a[i][j] = min([a[i-1][j-1], a[i-1][j], a[i][j-1]]) + 1
  pprint.pprint(a, width=40)
```

上のプログラムでは、配列の最大と最小を計算してくれる max 関数と min 関数を使っています。

- 配列が変数 a に入っている場合: max(a), min(a)
- 配列を直接指定している場合: max([, ... ,]), min([, ... ,])

以下に実行例を示します:

```
>>> squares()
[[1, 1, 1, 1, -1, 1, 1, 1],
 [1, 2, 2, 2, 1, 1, 2, 2],
 [1, 2, -1, 1, 2, 2, 2, 3],
 [1, 2, 1, 1, 2, -1, 1, 2],
 [1, -1, 1, 2, 2, 1, 1, 2],
 [1, 1, 1, 2, 3, 2, 2, 2]]
>>>
```

演習 12-1 部屋割り問題を動的計画法で解くプログラムを作成しなさい。作成後は、「13 人部屋: 33,000 円」「17 人部屋: 40,000 円」の選択肢を追加し、最適な部屋割り (要はトレース バック) および宿泊費を求めなさい。

演習 12-2 釣り銭問題を動的計画法で解くプログラムを作成し、適当な金額 (¢) を与えた場合、その金額と等しいコインの最小枚数およびその組み合わせを求めなさい。

演習 12-3 動的計画法を用いて、図 3 の m×n セルにある S の位置から、「右 · 下 · 斜め右下」の 3 方向へ移動しながら G の位置へ至る経路数を求めなさい。