Part. 3　再帰アルゴリズム

問題を解くための手順を「アルゴリズム」という。ここでは再帰を使った代表的なアルゴリズムをいくつか学ぶ。

目次

3-1. アルゴリズムと計算量（お話）

・アルゴリズム

アルゴリズムとは問題を解くための手順のことである。複数桁の計算を行うときに使う「筆算」は人間が使うアルゴリズムの一種だし、「右手を壁につけて歩き続ければ迷路を脱出できる」というのも迷路を解くためのアルゴリズムの１つである。

コンピュータにアルゴリズムを指示し、実行させるのがプログラムである。アルゴリズムによって速いもの、遅いもの、メモリをたくさん使うもの、使わないものなど良し悪しがある。コンピュータは一秒間に何億回という計算速度と、何億文字という記憶領域を持っているものの、下手なアルゴリズムを使うとすぐにその能力の限界に達してしまう。

・計算量

アルゴリズムの速さを表す１つの尺度が計算量である。計算量は、入力の長さ n に対して、どれだけの計算時間がかかるかを O 記号を使って表す。n に比例する時間がかかるなら O(n)、n の二乗に比例するなら O(n^2)、n の階乗に比例するなら O(n!)といった具合である。直感的には、「プログラムの中で一番多く実行される文が何回実行されるか」を考えるといい。

（１）O(n)アルゴリズムの例

・n 個の数 x[0], x[1], ..., x[n-1]のなかに、ある数 a が含まれているか調べる。

```
for(int i=0; i<n; i++){
  if(x[i]==a){
    return 1;
  }
}
```

この if 文は n 回実行されるので、これは O(n)アルゴリズムである。

・ある数 a の n 乗を計算する。

```
float x = 1;
for(int i=0; i<n; i++){
  x *= a;
}
```

これも x*=a;が n 回実行されるので、O(n)である。

なお、O(2n)とか O(10n)みたいな比例係数は考えないで、こういうのも全て n に比例しているので O(n)と書く。

（２）O(n^2)アルゴリズムの例

・n 個の数 x[0], x[1], ..., x[n-1]を小さい順に並び替える。

```
for(int i=0; i<n; i++){
    x[i]から x[n-1]までで一番小さい数を探して、x[i]と入れ替える
}
```

for 文の内側の、一番小さい数を探す、という作業にも n に比例する時間がかかるので、結局 n^2 に比例する時間がかかることになる。

（３）O(n!)アルゴリズムの例

・1, 2, ..., n を並び替えてできる n 桁の整数で、ある数 a で割り切れるものはいくつあるか。

1, 2, ..., n の並び替えは n!通りあるので、再帰で全部調べれば n!の計算が必要である。

・n 個の町をまわって、出発地点に戻ってくる道のなかで、最も移動距離の短い町をまわる順番を探す。

これも町 1, 2, ..., n をまわる場合の数は 1 から n までの並び替えの場合の数と同じなので、n!の計算が必要である。

（４）O(log2(n))アルゴリズムの例

log2(x)とは、2 の log2(x)乗が x になるような数である。log2(2)=1, log2(4)=2, log2(64)=6 である。n が 1000000 くらいになっても log2(1000000)は 20 くらいなので、非常に高速なアルゴリズムといえる。O(logn)と略記することも多い。

・ある数 a の n 乗を計算する。

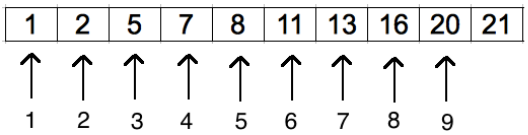
a を n 回掛け算するのでは O(n)かかってしまうのだが、実は $a^{2x} = a^x * a^x$ という性質を使うと、例えば a^8 を求めるには a^4 が分かればよく、a^4 は a^2、a^2 は a が分かればいいので、a^2, a^4, a^8 の３回の計算で a が求まることが分かる。これは a が 1024 でも、2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 と 10 回の計算で求めることができる。これは O(logn)アルゴリズムである。

3-2. 二分法

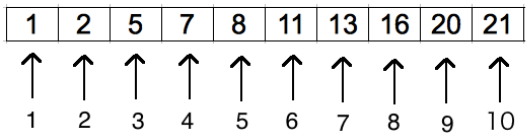
小さい順に並んだ n 個の数列の中に、ある数があるかどうか調べる O(logn)アルゴリズムが二分法である。数列を左から順番に調べていくと、最悪 n 回、平均 n/2 回比較を行わないといけないので、O(n)かかってしまう。ここで、数列が小さい順ということを利用して、「数列のまんなかを調べて、目的の数よりも小さければ、次に右側の領域を、大きければ左側の領域を調べる」という操作を再帰的に繰り返すことで O(logn)まで高速化することができる。

数列 | 1 | 2 | 5 | 7 | 8 | 11 | 13 | 16 | 20 | 21 | のなかに、20 があるか調べる手順と、10 があるか調べる手順について、O(n)アルゴリズムと二分法を比較してみよう。

・左から順に調べていく O(n)アルゴリズム

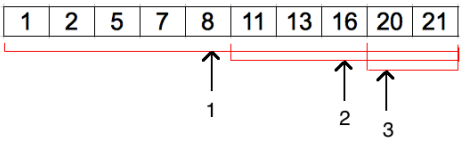


20を見つけたところで終了



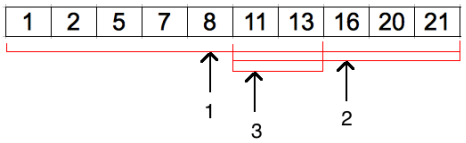
10は見つからなかった

・二分法 O(logn)



1. x[0]からx[9]までのどこかにある
2. x[5]からx[0]までのどこかにある
3. x[8]からx[9]までのどこかにある

20を見つけたので終了



1. x[0]からx[9]までのどこかにある
2. x[5]からx[0]までのどこかにある
3. x[5]からx[6]までのどこかにある

どこにもないので終了

二分法では、調べる領域が n 個→n/2 個→n/4 個… と減っていくので、n が 1024 でも 10 回の比較で探索を終わらせることができる。従って、このアルゴリズムの計算量は O(logn)である。

（１）O(n)アルゴリズムの実装

まずは左から調べる O(n)アルゴリズムを実装してみよう。

```
int[] a;
int N = 10000;

int lsearch(int t){
  for(int i=0; i<N; i++){
    if(a[i]==t){
      return i;
    }
  }
  return -1;
}

void setup(){
  a = new int[N];
  a[0] = 0;
  for(int i=1; i<N; i++){
    a[i] = a[i-1] + (int)random(1, 3.0);
  }
  noLoop();
}

void draw(){
  int t = (int)random(0, 10000);
  int res = lsearch(t);
  if(res!=-1) println(t + "  " + res + "  " + a[res]);
  else println(t);
}
```

グローバル配列 a が調べたい数列、N が数列の数の個数を表す。setup ブロックの中では、a をランダムな、小さい順に並んだ数列にしている。

a[i] = a[i-1] + (int)random(1, 3.0); と書くことで、必ず配列の右側が左の数より大きくなるようになっている。

命令 int lsearch(int t)は、a の中に t があるかどうか調べる命令である。for 文で a を左から順に調べていき、t が見つかったら何番目に見つかったかを返し、最後まで見つからなかったら-1 を返す。

draw ブロックでは、まず調べたいターゲットの数 t を乱数で生成し、lsearch を呼び出す。見つかっていたら、t と lsearch の戻り値と、その場所の a の数を表示する。実装が正しければ、t と a[res]は一致するはずだ。もし見つかってなかったら、t だけを表示している。

（２）これに、二分法命令 int bsearch(int l, int r, int t)を加えよう。この命令は、a[l], a[l+1], ..., a[r-2], a[r-1]のなかに、t が含まれているかを調べる命令である。探索領域が a[r]まででなく、a[r-1]までであるところがコツである。次の方針で実装する。

(i) r-l が 1 以下のとき、探索領域が残り 1 つ以下になっている。a[l]が t と等しければ、見つかっているので l を返す。等しくなかったら、見つからなかったということなので、-1 を返す。
(ii) a[l]から a[r-1]までのなかの、まんなかの数と t を比較したい。まず、変数 mid を用意して、ここに(l+r)/2 を入れる。
(iii) a[mid]が t より大きければ、t は a[l]から a[mid-1]までの間にあるはずなので、bsearch(l, mid-1, t)を再帰呼び出しする。
そうでなければ、t は a[mid]から a[r-1]までにあるので、bsearch(mid, r, t)を呼び出す。
(iv) 呼び出した結果を return する。

先ほどの lsearch 命令と新しく作った bsearch 命令を両方実行して、結果がちゃんと一致するかどうか確認せよ。

（３）lsearch と bsearch の速度比較をしたい。N=1000000 （百万）として、さらに draw ブロックの中に for 文を書いて、lsearch を 10000 回実行するようにせよ。lsearch の計算量は O(n)なので、10000*1000000=100 億回の計算がかかるはずだが、いったい何秒で計算が終わるだろうか？

次に bsearch も同様に 10000 回実行するようにせよ。log 百万は 20 くらいなので、20 万回の計算で済むはずだが、実際は何秒で計算が終わるだろうか？

（４）二分法を使うと、ルートを求める計算が高速でできる。次のような再帰命令 float root(float l, float r, float t)を書け。

・この命令は、l 以上 r 未満の小数 x のなかで、x*x = t をみたす x を返す

・さっきと同じように、x = (l+r)/2 として、x*x > t なら左半分の領域で、そうでなければ右半分の領域で再帰呼び出しする。

・r と l の差が 0.00001 未満になったら、そのときの r を正解として return する。

最初から使える命令 sqrt(x)の結果と比較して答え合わせせよ。

ちなみに、float ではなく double を使うとさらに精度を上げることができる。

（５）二分法を使って $x^3 + 2x^2 + 3x + 4 = 5$ を解け。（0.275682…になるはず）

※このように二分法は方程式を解くのに便利だが、単調増加か単調減少（グラフが右上がりか右下がり）でないと使えないので注意。

3-3. メモ化

Part2 の「数学の問題を解く」のところで、フィボナッチ数列を求める再帰命令を書いたが、それに fib(44)を求めさせてみよう。かなり時間がかかるのが分かるはずだ。

これは、fib(44)を求めるために fib(43)と fib(42)が呼び出され、fib(43)のために fib(42)と fib(41)が…と再帰が深くなるごとに 2 倍 2 倍に呼び出される命令が増えていくからである。しかし、よく考えてみると、fib(42)は fib(44)の再帰の時と fib(43)の再帰の時の 2 回呼び出されているし、もっと低い fib(10)とかはもう信じられないくらいの回数呼び出されているはずである。そして、fib(10)は何回実行しても結果は変わらないのだから、毎回そこからさらに再帰をたどるのは無駄である。

ここで出てくるのが「メモ化」のアイデアだ。一度計算した再帰命令の結果を配列に保存しておいて、すでに計算した再帰命令だったら、その保存しておいた結果から値を取り出すのである。こうすることで、計算量を劇的に速くすることができる。

（1）フィボナッチ数列のメモ化

Part2 のフィボナッチ数列をメモ化する。グローバル配列 a[i]に fib(i)の結果を記録する。a の中身ははじめは全て-1 にしておいて、ある fib(i)についてその結果がわかったら、a[i]にその値を保存する。fib(j)が呼ばれた時に a[j]が-1 でなかったら、fib(j)はすでに計算されているということなので、すぐに a[j]を return する。

```
int[] a;

int fib(int n){
  if(a[n]!=-1) return a[n];
  int ans = fib(n-1) + fib(n-2);
  a[n] = ans;
  return ans;
}
```

fib(44)を計算する速度を、メモ化してない時とメモ化した時で比較せよ。メモ化すると fib(10000)とかも一瞬で片付けることができるが、int の値の上限を突破してしまうので正解は出ない。

ちなみに、メモ化した時の fib(n)の計算量は O(n)である（なぜか？）。メモ化していないときは O(1.5^n)から O(2^n)くらいだと思う。

（２）Part2-2-(4)をメモ化せよ。引数が２つあるので、２次元配列でメモ化することになる。

（３）2009 年の情報オリンピック予選５番を解け（むずいです）
http://www.ioi-jp.org/joi/2009/2010-yo-prob_and_sol/index.html

3-4. ソート

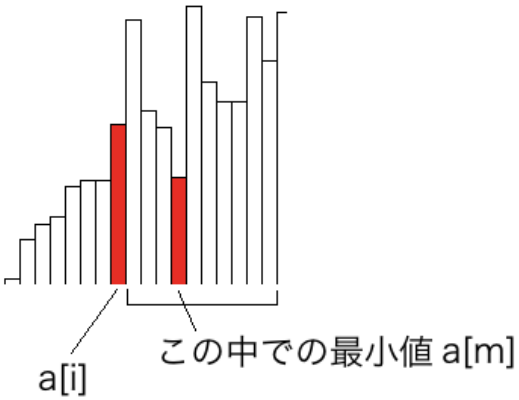
ばらばらに並んだ数を、小さい順とか大きい順とかに並び替えるアルゴリズムを、ソートという。

（１）選択ソート

グローバル配列 a, 変数 N を用意して、a に並び替えたい数列、N に数列の要素の個数を記録する。setup メソッドの中で、a[0]から a[N-1]までにランダムな小数を代入しておく。次のアルゴリズムを実装せよ。

(i) i が 0 から N-1 まで繰り返す

1. a[i]から a[N-1]までで一番小さい数を探す。それを a[m]とする。
2. a[i]と a[m]を入れ替える。



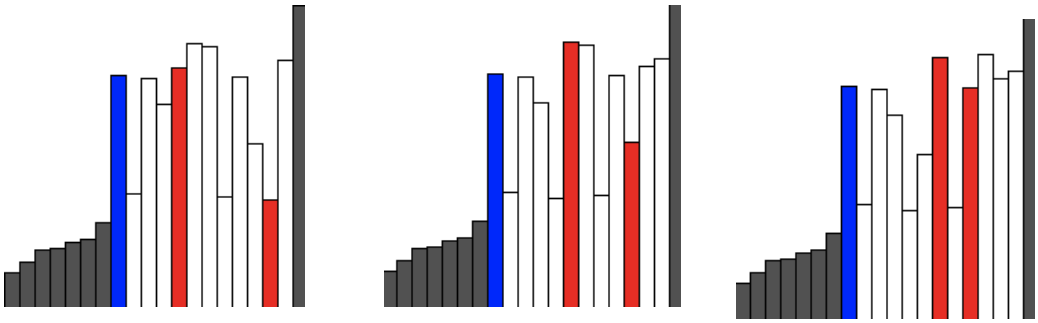
選択ソートの計算量は O(n^2)である。

（２）クイックソート

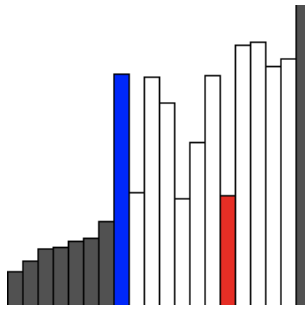
次の再帰命令 void qsort(int l, int r)について考える。これは a[l]から a[r-1]までをソートする命令である。

下の図で、灰色でない部分がこの a[l]から a[r-1]までとしよう。

(i) a[l]をピボットと呼ぶことにする（下の図の青色）
(ii) a[l+1]から a[r-1]までを左から順に見ていき、最初にピボットを超えたところを il とする。
(iii) 今度は右から順に見ていき、最初にピボットを下回ったところを ir とする。
(iv) il<ir なら、両者を入れ替える。（下の図の赤色ふたつ）



(v) il>ir なら、a[l+1]から a[ir]までは全てピボット以下、a[il]から a[r-1]までは全てピボット以上になっているはずである。ここで、ピボット a[l]と a[ir]を入れ替えると（下の図の青と赤）、a[l]から a[ir-1]までは全てピボット以下、a[il]から a[r-1]までは全てピボット以上となる。



(vi) qsort(l, ir)と qsort(il, r)を再帰呼び出しする。

再帰ごとに領域はだいたい半分ごとになっていき、一回の走査では O(r-l)回かかるので、最良の場合で O(nlogn)回の計算量となる。