アルゴリズムステージヒント

Hanoi

Knight

Packjack

Search

Sort

NQueen

追加箇所赤字にしてあるはずです

Hanoi

まず、ハノイの塔とは何か説明します。大きさの異なる円盤を多きいものから順に重ねた塔です。この塔をa,b,c,の3箇所あるスペースの内、塔が置いてあるaからbへ移動させることを考えます。ただし、塔、円盤の移動の仕方にルールがあります。

1.移動の際動かせる円盤は1枚ずつであり、上から取り、別の塔、スペースの上に重ねることしかできない

2.円盤を置くことができるのはa,b,cの3つのスペースのみ

3.絶対に小さな円盤は大きな円盤の下に置くことはできない

以上のルールを守り円盤を移動させます。

アルゴリズムのヒント

ルールに従って円盤を塔をbへ移動させたいならば、一番下の円盤をbに移動させる必要があります。そして、bに一番下の円盤を移動させるにはそれ以外の円盤らをcへ移しておく必要があります。そして、bに一番下の円盤が移せたなら、cに積んである等をbへ移動させればよいのです。n枚の円盤が塔になっていれば、

上からn-1枚をcへ移動させる

最も下の円盤をbへ移動させる

cのn-1枚をbへ移動させる

これらの処理を考えるとわかりやすいでしょう。

つまり

n枚の円盤を移動させるメソッド()

n-1枚の円盤を移動させるメソッド()

下の円盤を一枚移動させるメソッド()

n-1枚の円盤を移動させるメソッド()

end

また、n,n-1枚の円盤を動かすことは本質的には同じ処理であることから、引数にn-1を取らせれば、これらは再帰的に記述することができます。また必要な情報はスタート位置、塔の退避場所、ゴールと円盤の数です。

n枚の円盤を移動させるメソッド(スタート,ゴール,退避,数)

のようになることが分かってきます。

最初のメソッドでは引数をどのようにすれば良いでしょうか。下の円盤を動かしたいのですから、スタートはそのままに、n-1枚の円盤のゴールは退避場所になるでしょう。

そして、下の一枚をゴールへ動かします。では次のメソッドはどうなるでしょうか。今、n-1枚の塔は退避場所にあります。これをスタートとして、ゴールは最初に設定したゴールです。そして、もとのスタート位置から塔はなくなります。これで作るべきメソッドはわかるはずです。あとは終了条件を考えましょう。

operator(hanoi)

hanoiSub(hanoi,1,b,c,n)

end

のような構造を考えましょう。

hanoi.move(from,to)でfromからtoへ円盤を動かします。

Knight

ナイトツアー問題とは普通は8\*8の64マスすべてを通るようにチェスのナイトを動かすことができるかという問題です。チェスのナイトの動きは8種類ありますね。(十字方向に2マス進み進行方向に対して左右いずれか1マス進みます。4\*2=8種類)

これをバックトラックアルゴリズムを使って解きます。

バックトラックアルゴリズムの直感的な説明としては、いけるとこまでいってダメになったら戻って別の手を打つというものだと思っていいでしょう。

ナイトをnステップ動かせるかどうかを判定するメソッドを考えましょう。このメソッドを考えるとバックトラックを使うタイミングが分かってくるはずです。これを手作業でやる時のことを考えると、

まず、ナイトが動かせるマスがあるかどうか探す。

あれば動かす。

次にナイトを動かす前に、動かせるマスがあるかどうか判定する。

動かせるなら動かす。

動かせなければバックトラックする。

これを繰り返す。

そしてこの4行そのものがナイトツアー可能かどうかを判定するアルゴリズムであると分かると思います。また、繰り返しであるから、再帰を使えば簡単に書けそうです。

再帰を使ったメソッドの大枠を考えましょう。

ナイトをnステップ動かせるかどうか判定するメソッド()

ifナイトが次どこかのマスに動かせる

ナイトを動かす()

ifナイトをn+1ステップ動かせるかどうか判定 then

return true

else

バックトラックする

end

end

return false

end

また、このメソッドは0ステップ目から始まることを考えると、総マス数-1ステップまで到達すれば終了してよいはずです。また移動候補は8通りあるので、どのマスに対しても到達可能マスが見つかるまで8回繰り返して判定すればよいはずです。

tour(horse)

tourSub(horse,0)

end

の構造で考えましょう。horse.canMove(i)でiのパターンの移動ができるか判定します。

horse.move(i)でiのパターンの移動をします。horse.backtrack()でバックトラックします。

Packjack

ナップサック問題を考えましょう。ナップサック問題とは容量の決まったナップサックにものを詰め込むことを考えます。ただし、詰め込むものにはサイズと価値がきまっており、容量を超えないように価値を最大にすることを考えます。(アイテム数+1)\*(ナップサック容量+1)の配列を考えるとわかりやすいでしょう。

行にアイテムの種類、列に容量が0,1,2,...とならんだ2次元配列を考えます。

1番目のアイテムをナップサックに入れます(1行目)。このとき一番目のアイテムのサイズ以上の列に1番目のアイテムの価値を入れます。入らない列には0を入れておきます。

次に2番目のアイテムを入れます。このとき、入れないという選択肢がありますが、それは2番目のアイテムと1番目のアイテムのサイズの和が列の容量を超えてしまうときです。

また、1,2両方は入らないが、2を入れたほうが価値が大きくなる場合、1の価値を削除して2の価値に上書きしましょう。あくまで行うのはこれまでの価値と次のアイテムの価値との加算と比較です。これを繰り返すと行列の右下に最大の価値が導出されているはずです。また、工夫すればどのアイテムをナップサックに入れたか求めることができます。

knapsack(packjack)

を考えます。

packjack.getItemNum()でナップサックの容量を得られます。

アイテム名はitemName[]で容量と価値はitems[][]です。

packjack.pickItem(itemName[])でアイテムを取り出せます。

Search

ソートされた要素列から指定された値を取り出したいときには要素を探索する必要があります。2分探索を考えましょう。2分探索は3点に着目して要素を探します。左、右、中の3点を考えます。まず、左に要素列の最小値をあて、右に最大値、中に(右の要素番号-左の要素番号)/2の値をあてます。取り出したい目標値と中の値を比較します。小さいもの順で並んでいるとするならば、目標値の方が大きければ、中の要素番号を左に更新、代入して、中の番号をを先と同様に導出します。目標値の方が小さければ、中の要素番号を右に更新して、中は同様に導出します。これを目標値がヒットするまで続けます。

例を見ましょう。(0,1,2,3,4,5,6,7,8,9)ここから8を探します。

まず、左は0,右は9,中は4です。

4と8を比較します。8の方が大きいので左を4,右を9,中を6とします。

6と8を比較します。同様に,左を6右を9中を7とします。

7と8を比較します。同様に、左を7右を9中を8とします。ここで8にヒットしたので終了します。このように解の候補を絞り込むことができます。ポイントは既に要素列が整列されていることにあります。

メソッドの大枠を考えましょう.

while

中の番号を決める

中の番号の要素と目標値との大小を比較する

if

同じであれば終了する

目標値が小さいとき中の番号-1を新しい右の番号にする

目標値が大きいとき中の番号+1を新しい左の番号にする

end

end

終了条件は目標が見つかったときまたは右と左の番号が一致したときである。

メソッドの引数はboxs配列,targetです。boxs.checkElement(a,b)で要素番号a,b値の大小を比較できます。

boxs.setResult(a)で最終結果を入れられます。

Sort

ソートアルゴリズム、並び替えを考えましょう。

バブルソートを考えます。今、横に一列になって要素が並んでいることを想像してください。(2,1,0,4,3)これらを左から順に小さいものから並べること考えます。

隣り合う要素の大小を比較します。まず、1,2を1,2に入れ替えます。(1,2,0,4,3)

次に、2,0を比較して入れ替えます。(1,0,2,4,3)

次に、2,4を比較します。入れ替えは起こりません(1,0,2,4,3)

次に、4,3を比較して入れ替えます。(1,0,2,3,4)

最終要素の比較が完了したらここで初めに戻ります。最も大きい要素が最右に来たはずです。

また、1,0を比較して入れ替えます。(0,1,2,3,4)

ここでは完了したので終了しますが、本当ならばまだ処理は続きます。例えば、

(4,3,2,1,0)の場合であれば、0を最も左に持ってくるには4周入れ替えを行う必要があります。つまり要素数-1回繰り返しを行う必要があります。

メソッドの大枠を考えると

while 要素数-1回

while要素数-1-周回数

隣り合う要素を比較して入れ替える

end

end

添え字は周回数と要素番号を持つでしょう。

使う配列はarrayです。

選択ソートを考えます。選択ソートは最小値、最大値を要素列から見つけ出しその要素と先頭要素の値を交換してソートを行うアルゴリズムです。

例を見てみましょう。

(2,1,0,4,3)を選択ソートで小さいものから順に並べます。最小値を見つけるアルゴリズムは知っていると思います。第一要素を仮の最小値として、他の要素と比較し、より小さいものがあればその最小値を更新し比較を続けるアルゴリズムです。それによって最小値は0と分かったので2,0を交換します。(0,1,2,4,3)

2番目の要素以降での最小値を探します。ここでは1です。ただし、それ以降に1より小さな値はないので入れ替えは起こりません。2も同様です。最後に3を見つけたとき4,3を交換し、比較できる要素がなくなり終了します。

メソッドの大枠を考えます。

while

仮の最小値を決める

while

最小値を見つける

end

入れ替える

end

仮の最小値を決めるときに要素番号を1つずつ増やす必要があります。

使う配列はarrayです。

NQueen

Nクイーン問題を考えましょう。n\*nのチェス盤にクイーンの駒を互いに取り合わないような配置は可能かどうかを解く問題です。チェスのクイーンは縦横斜め方向へ任意のマス数進むことができますね。互いに取り合わないというのは、コマ同士が同一直線状、対角線上にないように配置することと同じです。これを解くアルゴリズムを考えましょう。

またこれをn駒目を置くアルゴリズムとしてとらえて、再帰的に記述することができます。

大枠を考えましょう。

n駒目を置くメソッド()

候補地を決める

駒を置く

n+1駒目を置くメソッド()

end

解が見つかったとき、queen.putQueen(ans[][][],ans[][][])でオブジェを動かしましょう。

Knight2

貪欲法で解けることが知られています。ワーンスドロフの規則に従うとうまくいくことが知られています。

あるマスに飛び移ったとき、そのマスからさらに飛び移ることのできるすべてのマスを拾う。そして、それぞれのマスからさらに何個のマスへ飛び移れるかをカウントして、最少の飛び方しかできないマスに実際に飛び移る。

貪欲法はお金を支払う問題にたとえられ、555円を最少の硬貨数で支払うときに大きな硬貨から順に取り出すとわかりやすいですね。これと同様に飛び方が一番少ないマスから順に飛んでいく点が貪欲法であります。ただし、この規則で必ず解けることは証明されておらず有名な未解決問題とされています。最少の飛び方を探すメソッドgetNextJumps(knight)を考えましょう。最小値を探す問題と同じです。knight.move(i)とcanMove(i)を使ってナイトを動かして、次のマスへ進めるかどうかを判定、カウントしましょう。戻るときはknight.backtrack()です。