なんとなく分かる データ構造とアルゴリズム

for 2007 2008 2009 オブプロ

松澤　芳昭

参考文献

- Javaで学ぶデータ構造とアルゴリズム
  Robert Lafore 岩谷宏訳 ソフトバンク1999

データ構造とアルゴリズムとは

- コンピュータは1度に一つのことしかできない
  - メモリへの書き込み，読み込み（一つだけ）
  - 計算（一つだけ）
- 手順を上記の機能に分解できないとコンピュータは問題を解けない
- CSは短くて長い歴史を持つ．これまで人類が様々な問題を解いて獲得した知恵の集大成．
- そのまま応用することもできるし，その考え方は新しい解法を創造するときの足がかりとなる→学習することに価値がある→プロになる人はデータ構造とプログラミング論をとろう

データ構造とアルゴリズム概観

- 0．データ構造とアルゴリズムとは
- 1．配列
- 2．検索とオーダー
- 3．並び替えアルゴリズム
- 4．ハッシュテーブル
- 5．スタックとキュー

1．配列

- 大きさが決まっている入れ物
- ArrayListの中身は配列

作成方法の違い

```
ArrayList<String> list = new ArrayList<String>();
String[] array = new String[10];
```

配列とリストの違い

追加

```
list.add(“おはよう”);
array[0] = "おはよう"; //何番目に追加するか，計算しなければならない
```

削除

```
list.remove(0);
array[0] = null; //つめなければならない
```

取得

```
list.get(0);
array[0];
```

大きさの取得

```
list.size();
array.length;
```

2. 検索とオーダー

- 検索の代表的なアルゴリズムについて説明できる
- アルゴリズムの性能について比較検討し、説明できる

効率が悪いとどうなるか？

- 1万人の社員がいる会社があります。
- ログインするために社員番号でパスワードを検索します。
- 1人の検索に0.1秒かかると、全員がログインし終わるまでに何分かかりますか？

こたえ:
1分で600人だから
10000/600=16.66分　　→大変だ！

検索とオーダー

- リニアサーチ
- バイナリサーチ

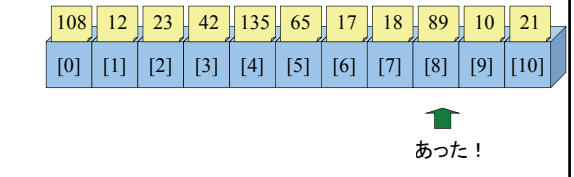
検索のアルゴリズム

- 番号を検索することを考え
  - 要素の先頭から順番に比較し、見つかるまで続ける

リニアサーチ

- 最初から順番にに探す方法を「リニアサーチ」という.

89を探す場合



効率は？

- 番号が1万件あった場合
  - 0を検索→1回の比較(if文)で見つかる
  - 9999を検索→1万回の比較で見つかる

1個あたり平均5000回の比較で見つかる
よって、比較が実行される回数は
全部で5000回×10000個=50,000,000
(5千万回)

よって、n個の検索で必要な比較の数は
n/2 × n = n^2/2

10万回検索をしたら…

- 10万要素を10万回検索した場合は、
  100000 × 100000 /2 = 5,000,000,000
  (50億回)
  の比較が必要
- 理論上は5秒で検索が終わりますが、実際にやってみると、もっとかかります

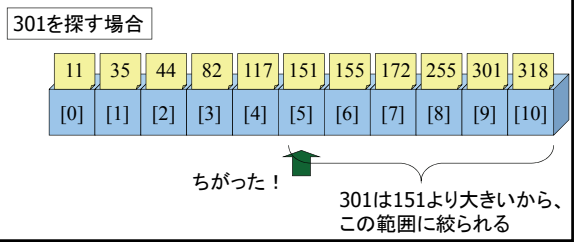
バイナリサーチ

- バイナリサーチ
  - 配列の中身がソート済み（順番に並んでいる）の場合に有効
    - 今回はfor文で順番に追加されているのでソート済みになっています

バイナリサーチの考え方(1)

①まず、真ん中を探します。

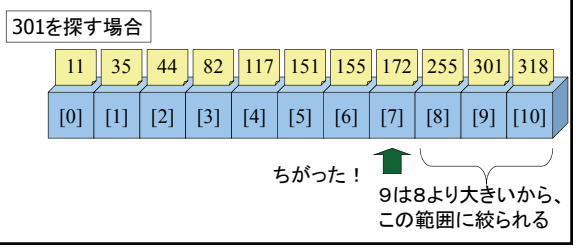
②違った場合も、探す範囲は半分に絞られます。
（数が少なかったら右半分、逆なら左半分にあります。）



バイナリサーチの考え方(2)

③次に、その範囲の中のまた真ん中を探します。

④それを見つかるまで繰り返します。



②バイナリサーチの効率

- 例えば、100個の要素の中から検索することを考えてみましょう。

リニアサーチの場合
比較の回数は　－最小の場合で1回
　　－最大の場合で100回
　　－平均50回

バイナリサーチの場合
比較の回数は　－最小の場合で1回
　　－最大の場合で7回
　　－平均7回以下

バイナリサーチの効率（2）

要素数	比較最大回数	
	リニアサーチ	バイナリサーチ
10	10	2
100	100	7
1000	1000	10
10000	10000	14
100000	100000	17
1000000	1000000	20
10000000	10000000	24

バイナリサーチの効率（3）

バイナリサーチの比較回数	検索できる範囲
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

2^比較回数
ようするに
「2の巾乗」
が検索できる範囲
になる。

2の巾乗を逆計算する

- 問題：100をバイナリサーチすると、最大何回の比較が必要か。計算で求めよ

答え　$\log_2(100) = 6.644$
切り上げて7回

検索の効率を比較する

- Nつの要素の中から検索するときの効率は、以下のように計算されます。

リニアサーチの場合
比較の回数は　－最小の場合で1回
－最大の場合でN回
－平均N/2回
→ Nに比例

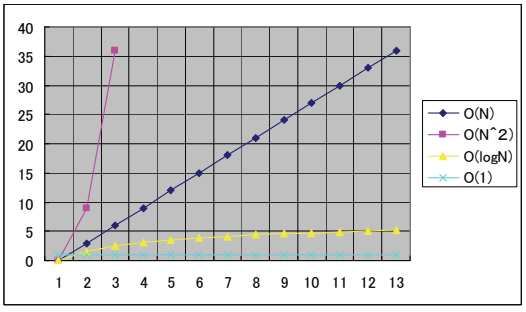
バイナリサーチの場合
比較の回数は　－最小の場合で1回
－最大の場合でlog2(N)回
－平均log2(N)回以下
→ Log2(N)に比例

効率を比べる一般的な記法

- BigO記法
  1. BigO記法は、その名の通り、大文字のOを使いますが、order(次数)の意味です。

	bigO記法
Nに比例	O(N)
Log2(N)に比例	O(LogN)
Nの2乗に比例	O(N^2)

BigOのグラフ化



並び替えアルゴリズムの効率

- バブルソート
- 選択ソート
- 挿入ソート

並び替えのアルゴリズム

- バイナリサーチを使うには、ソート済み（順番に並んでいる状態）である必要がありました
- ソート（並べ替え）をするアルゴリズムを考えましょう

簡単なソートをやります

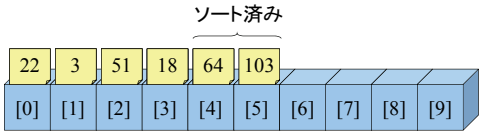
- ソートは重要で時間のかかる処理なので、これまでにも多くのコンピュータ科学者が研究に取り組み、いくつかの高度な方法も発明されています
- 今回はその中で比較的簡単な3つの方法を紹介します
  - バブルソート
  - 選択ソート
  - 挿入ソート

これらのテクニックは素朴で、遅いアルゴリズムですが、やってみる価値はあります
単に分かりやすいだけでなく、データの並び方や数によって、これらの方法のほうが速い場合もあるからです

バブルソート

- アルゴリズム
  - 2つの番号を比較する
  - 左側の方の番号が大きければ番号を入れ替える。
  - 右へ一つ移動する
  - ソートの終ってない部分(毎回小さくなる)に対して上のステップを繰り返す
    - ※配列の一番左の箱の中の番号が最小になるように並べ替える場合です

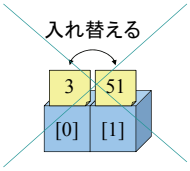
バブルソート



とやっていくと、最後にソートされます

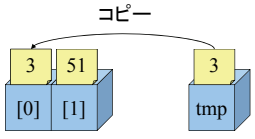
①コンピュータでの「入れ替え」(Swap)

- 前ページの例では、「入れ替え」を一回の操作で行っているように見えます．しかし，コンピュータはそのような操作はできません



コンピュータでの「入れ替え」(Swap)やり方

- 一時的な変数を用意します



考えてみよう

- バブルソートの効率を考えなさい

特に
– ①比較の回数
– ②入れ替えの回数
を考えなさい

バブルソートの効率

- 比較の回数

	比較の回数
2個の要素の時	1 回
3個の要素の時	2 ＋ 1 回
4個の要素の時	3 ＋ 2 ＋ 1 回
N個の要素の時	N + N-1 + … + 2 + 1 = N(N-1)/2 回

バブルソートの効率

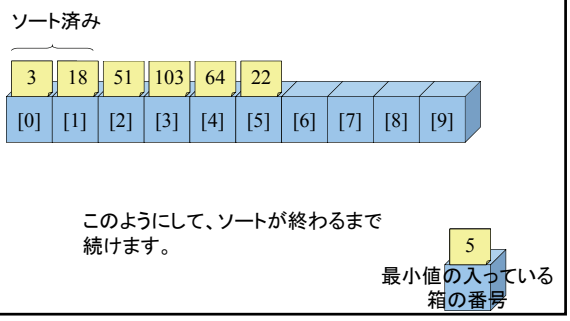
- 入れ替えの回数
  確率的には、比較したうちの半分は入れ替えになります。
- よって…

N(N-1)/4回

選択ソート

- アルゴリズム
  – 全番号から、一番小さい番号を探す。
  – 見つかったものと、一番左の番号を入れ替える。
  – ソートの終ってない部分(毎回小さくなる)に対して上のステップを繰り返す。

選択ソート



考えてみよう

- 選択ソートの効率を考えなさい

特に
– ①比較の回数
– ②入れ替えの回数
を考えなさい

選択ソートの効率

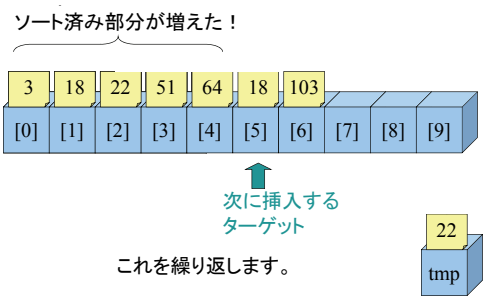
- 比較の回数

	比較の回数
2個の要素の時	1 回
3個の要素の時	2 + 1 回
4個の要素の時	3 + 2 + 1 回
N個の要素の時	N + N-1 + … + 2 + 1 = N(N-1)/2 回

挿入ソート

- アルゴリズム
  - 途中までソートが終わっているとします（そのほうが理解しやすいので）
  - ソートが終わっていない中で、一番左にある要素をソート済みの要素の中で、あるべき位置に挿入します
  - ソートの終ってない部分(毎回小さくなる)に対して上のステップを繰り返します

挿入ソート



これを繰り返します。

考えてみよう

- 挿入ソートの効率を考えなさい

特に
- ①比較の回数
- ②入れ替えの回数

を考えなさい

挿入ソートの効率

- 比較の回数

	比較の回数
2個の要素の時	1 回
3個の要素の時	2 + 1 回
4個の要素の時	3 + 2 + 1 回
N個の要素の時	N + N-1 + … + 2 + 1 = N(N-1)/2 回

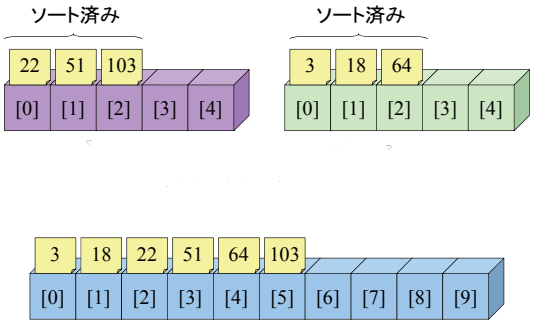
場合によって効率が変わる！

- 挿入ソートは、ほとんどソートされているときには、とても効率的
- 逆に、逆順にソートされている場合、最大の比較回数と移動回数が実行されてしまい、バブルソートと同じ遅さになってしまう

マージソート

- アルゴリズム
  - 要素が一つになるまで配列を2分割し続ける
  - 隣り合う要素同士を比較して大きいほうが右になるように分割された2つの配列をマージ（統合）します

マージソート



マージソートの効率

N	作業スペースへのコピー回数	コピーの合計回数	比較回数(最大:最小)
2	2	4	(1:1)
4	8	16	(5:4)
8	24	48	(17:12)
16	64	128	(49:32)

Javaのライブラリを使ってソート

```
ArrayList<String> list = new ArrayList<String>();

....
....

Collections.sort(list);
```

ハッシュテーブルとは

- 検索と効率
- ハッシュテーブル概要
- keyとvalue

検索と効率

- リニアサーチ
  - O(N)
- バイナリサーチ
  - O(logN)
- ハッシュテーブル
  - O(1)

ハッシュテーブル概要

- 検索する対象を数字にして（インデックスをつける）を対応する番号の配列にいれておけば、直接検索できる
  - 木村を1001に変換して，配列の1001番地に入れておく

0	
1	
2	
3	

……

配列番号	内容
1000	
1001	木村
1002	
1003	

13.1.3 keyとvalue

- key
  - 検索するためのキー
- value
  - 格納されている値

key	value
木村	30点
稲垣	80点
香取	95点
中居	55点

ハッシュテーブルの仕組み

- 名前をキーに検索できるようにする
- ハッシュ値を求める
- 番地の衝突を回避する
- ハッシュテーブルの利点･欠点

ハッシュ値を求める

- ①名前を数字に変換する
- ②配列の番号と対応させる
- ③小さい配列で済むようにするには

①名前を数字に変換する

- インデックシング
  - 文字として扱っている間は、コンピュータでは検索できないので、何らかの数字に変換する

例えば、こんなやり方があります（詳しくは専門書を参照）

変換表

木	98
村	103
拓	67

木村拓　→　9810367

②配列の番号と対応させる

- 例えば、「木村拓」を9810367という数字に変換したら対応する配列の番地に挿入する
  - こうしておけば、1回で検索できる

1	
2	
3	
4	

……

9810366	
9810367	木村拓
9810368	
9810369	

しかし、そんなに大きな配列を作るのは無駄

③小さい配列で済むようにするには

- 例えば1000要素の配列で済ませることを考える
  - 1000で割ったあまりを求める

木村拓 → 9810367 → 367

1	
2	
3	
4	

.....

366	
367	木村拓
368	
369	

.....

番地の衝突を回避する

- ①番地の衝突
- ②衝突回避(1):空き番地法
- ③空き番地法の問題点
- ④衝突回避(2):分離連鎖法

①番地の衝突

- 違うオブジェクトが同じハッシュ値になる可能性がある

木村拓 → 31256448 → 448
山下智久 → 587648448 → 448

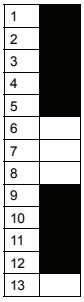
②衝突回避(1):空き番地法

- もし計算された番地にすでにデータが格納されていた場合次の番地にデータを格納する
  - 次も格納されていたらさらに次の番地に格納しようとする
- 検索するときにはハッシュ値を求めた後、その番地から順に配列の中身をリニアサーチで調べる

③空き番地法の問題点

- クラスター化
  - 集中して埋まってしまう個所ができてしまう

リニアサーチが長くなって効率が落ちる

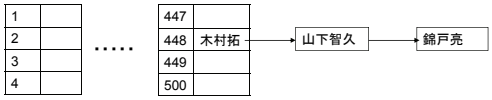


クラスター化を防ぐ

- 平方探索
  - 重複した時に、次の番地ではなく1,4,9,16こ先と少し離れた場所に置く
  - 第2種クラスター化
- ダブルハッシュ
  - キーの値によって次の番地を決める
- 分離連鎖法
  - 次に説明する方法

④衝突回避(2):分離連鎖法

- 衝突したら、その番地からさらに連結リストを使ってデータを格納する
- 検索するときにはハッシュ値を求めた後、その番地から順に連結リストを使って調べる



ハッシュテーブルの利点･欠点

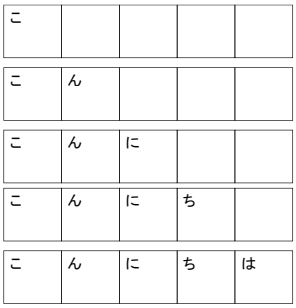
- 利点
  - 検索が速い → O(1)
  - 名前でも速く検索できる
    - バイナリサーチでも工夫すればできる
- 欠点
  - あるキーによる検索しかできない
    - 名前をキーにしたら、idでの検索はリニアサーチ
  - データの保持に内部的には配列を使用するので、事前に用意した配列が満杯に近づいてくると検索、挿入の効率が格段に落ちる
  - 保持するデータを昇順や降順にソートして取り出すといった機能はない

スタックとキュー

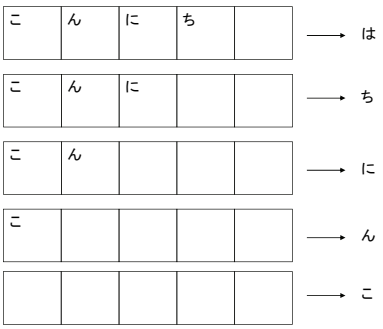
- 2種類のデータ収納パターン
  - スタック
    - Last In First Out(LIFO)
    - 後入れ先出し方式
  - キュー
    - First In First Out(FIFO)
    - 先入れ先出し方式

後入れ先だし方式（スタック）

- 挿入（例：こんにちは）



スタック，取り出し



スタックのその他の使い道

- 算術式のパース(解析)
- XML,HTMLなどの入れ子構造の解析
- 食堂のトレー置き場
- etc

先入れ先出し方式（キュー）

- 挿入（例：こんにちは）

こ				
こ	ん			
こ	ん	に		
こ	ん	に	ち	
こ	ん	に	ち	は

キュー，取り出し

ん	に	ち	は		→ こ
に	ち	は			→ ん
ち	は				→ に
は					→ ち
					→ は

キューのその他の使い道

- 銀行の待ち行列
- プリンタの待ち行列
- コンピュータのプロセスの待ち行列
  （プライオリティーキュー）
- etc…

おまけ：算術式のパース

- 問題：
  3 + 4 * 5
  (4 + 5) * (8 - 4)
  などをコンピュータはどうやって計算するか？

答え

- コンピュータはそのまま計算できない。
  →特別な記法に変換する必要がある。

- 例)
  3 + 4 * 5　　→　　3 4 5 * +

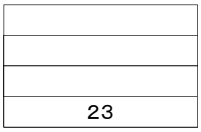
中置記法　　後置記法
(逆ポーランド記法)

後置記法で書かれた式を計算する

- ルール
  - 左から一文字ずつ読む
  - 数字がきたらスタックにつむ
  - 演算子がきたらスタックの上から2つの数字を取り出して演算する
    演算したら結果をスタックの上に積む
  - これを式の最後まで繰り返す

後置記法で書かれた式を計算する

計算対象
3 4 5 * +
↑



スタック

中置記法と後置記法

中置記法	後置記法
A+B-C	AB+C-
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C-D)	AB+CD-*
((A+B)*C)-D	AB+C*D-

後置記法と日本語

- 3 + 4 * 5
  3 4 5 * +
  →3に4と5をかけたものを足す
- 5 * 8 + 7 * 3
  5 8 * 7 3 * +
  →5と8をかけたものと7と3をかけたものを足す

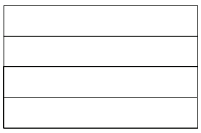
- 日本語と語順が同じ。
  →日本語とコンピュータは相性がいい。

中置記法を後置記法に変換する

- ルール
  - 左から一文字ずつ読む
  - 数字がきたら、結果に書き出す
  - かっこがきたら
    - 開きかっこの場合→スタックに積む
    - 閉じかっこの場合→開きかっこを見つけるまで、スタックから演算子を取り出して、結果に書き出す
  - かっこ以外の演算子がきたら
    - スタックが空なら→スタックに積む
    - スタックを覗いて、スタック演算子よりも優先順位の低い演算子だったら、結果に書き出す。そうでなければ、スタックに積む
  - 式を最後まで読み取ったら、スタックに入っているものをすべて出力する

中置記法を後置記法に変換する

対象
3 + 4 * 5



出力
3 4 5 * +

スタック