今日からソートを3回。
アルゴリズム9個を教えます。

[整列とは？]
あの人の本では600個書いてある。
あらゆる処理で速いアルゴリズムというのは存在しない。
あらゆる処理の基本になるので3回で集中して勉強しましょう。

[同じものを抽出することをマージというのか。]

[日本人全員の名前をリストアップしたファイルの話]
ある名前の数を調べる訳ではない。
1つの名前だったら1回走査すればいい。
全ての名前のそれぞれの出現回数を調べなければならない。
もしそれぞれの名前で走査したら、漸近計算量はn^2になってしまう。
だからまずソートしなければならない。

[unixのsortコマンドはマージソートです]
デフォルトでは文字コードでソートするので、数字としてソートしたい場合は-nオプションをつける。
デフォルトでは昇順なので、降順にするには-r(reverse)オプションをつける。
[uniqは当然1回なめるだけだから、漸近計算量はnですね。]

3回のソートに関する講義概要

基数整列法は、漸近計算量nで恐ろしいほど速いんだけど、実際に作ると遅いから使われない。

キーの中身まで入る？？？？？？？？？？

クイックソートは、

普通にバラバラならnlognで動く。

まったく逆順に整列されていたりすると、n^2になる？

ヒープソートとマージソートは最悪計算量との差がない？

前提条件

マージソートはある程度大きなものも扱えるけど、今回教える8つのアルゴリズムは、ある程度小さなデータを扱う。

ハードディスクにはみ出すようなデータは結構工夫しないと速くならないけど、それは教えない。

ハードディスクの読み取り方式に依存する(局所アクセス)。

マージソートは結構相性がいい。

今日やる4つ目はn^1.5とかいう変なやつです。

メモリの使い方の種類

読み込んだデータの配列の中で入れ替える(その場ソート)

配列のコピーが必要

マージソートがこれ

安定性(stable) - ちょっと役が微妙なんだけど

キーの値が同じレコードの元の順序が、保存されるものを、stableなsortという。

入出力は

sort(int a[], int N){}

に固定しましょう。中身だけ変える。

また、昇順に固定しましょう。

1からNにしたい。

0はとっておきたい。

a[0]に特殊な値(番兵)を入れておくことによって、あるところに来たらとまるとかさせたい？

[16:04]ここから板書
今日やること
- バブル整列法
- 選択整列法
- 挿入整列法
- シェルソート法

バブル整列法

これは実習で作りましたよね。
でも計算量を出すところはやった方が良いのか。
隣同士比べて、小さいものを左に。
1回やると一番大きいのは一番右に行く
だから、探索する要素を1つずつ減らしていく。

```
bubble ( int a[], int N ) {
    // 最初はNこ見て、1つずつ減らしていって、最後は2個見る。
    // これは外側か
    // iの中身がNからだんだん1つずつ減らしていって、2になったらforを抜けるという意味か。
    for ( i = N ~ 2 )
        // jの中身を2からだんだん増やしていって、iになったらforを抜ける。
        // i-1回回っている。
        // 中身はコンスタントタイム1でいいよね。
        // 代入して入れ替えるのに時間がかかるけど
        for ( j = 2 ~ i )
            if ( a[j-1] > a[i]
                a[j-1] <=> a[j]
}
```

シェイカー法
ほぼ全部揃っているんだけど、1個だけ違うという場合、
最初が逆の場合は1回で終わるけど、
最後が逆の場合は、flagを立てて途中で終わらせるやり方だと、N回になる。
これを速くするためにはどうすればいい？
ひっくり返せばいい。
バブルソートを上からとしたからと両方から交互にやる。
ほとんどの場合は計算時間にほとんど差は出ない。

その場整列です。
安定かどうかは授業の最後の方で訊くよ。

① バブル整列法

$$T(N) = \sum_{i=N}^{2} i \quad - 1$$

$$= \sum_{i=2}^{N} i \quad - 1$$

$$= \sum_{i=1}^{N-1} i$$

$$= \frac{1}{2} N(N-1)$$

$$= \Theta(N^2)$$

$$= O(N^2)$$ ← flagを立てて途中で終わるようにすると

選択整列法(selection sort)

一番小さいやつを見つけて、左に置く。
次に小さいやつを見つけて、左に置く。
[これは数の分なめなくちゃいけないから完全にn^2に一致するな。]
[どうやって消したものを見ないようにするのかの実装も大事そう]
[これは配列のコピーが必要なのか]

実際はちょっと工夫すればその場ソートにできるので、その場ソートで実装される。

見つけた一番小さい数と一番左を交換する
[これだと、バラバラに消えないから、見ないようにするのも、単純にインデックスを増やせばいいだけだから楽になるね。]

実装方法によってstableかstableじゃないかは違ってくる。

```
selection ( int a[], int N ) {
    // どこまで整列が終わっているかを規定する
    // 一番外側のiは、探す配列の最初のインデックスを指定する。
    // 最初は1番目から探すけど、
    for ( i = 1 ~ N-1 )
        min = i
        // 最初の値を一番小さいと仮定してminに入れる。minの中身と配列の整列されていない要素を比較していくから、比較する対象は1つずつ減っていく。
        for ( j = i+1 ~ N )
            if ( a[j] < a[min] )
                a[min] <=> a[i]
}
```

真ん中は $j = i+1 \sim N$ だから、

$N - (i+1) + 1 = N - i$ 回回る。

$$\sum_{i=1}^{N-1} (N-i)$$

$$= \sum_{i=1}^{N-1} i \quad \because \text{逆転させても一緒}$$

$$= \frac{1}{2}N(N-1)$$

$$= \Theta(N^2)$$

$$\neq O(N^2) \longleftarrow \text{チェックしようがないらしい}$$

挿入整列法(insertion sort)

トランプのカードを思い出して下さい。

並んでないやつを取り出して、並んでそうなところに挿入しますよね。
今は直感的に分かるように配列でやるけど、配列の場合ずらさないといけないので遅くなる。

ずらすところをちょっとだけ速くするためにはどうすれば良いか。

挿入対象を記憶しておいて、
大きい方から探していく。
挿入対象よりも大きいものを1つずつ右にずらしていく。
挿入対象より小さいものを見つけたら挿入する。

```
insertion ( int a[], int N ) {
    // 1つ目はソート済みだと考えて、2つ目から取り出す。
    for ( i = 2 ~ N )
        V = a[i] // 挿入対象を記憶する
        j = i
        // 挿入対象(V)より、穴が空いている要素の1つ左が大きいなら、a[j-1]を右にずらさなくちゃいけない。
        while ( a[j-1] > V )
            a[j] = a[j-1]
            j--
}
```

このままだと Segmantation fault する。

一番左にあった数より、小さい数を持っていると、 while (a[j-1] > V) の条件がいつまでもviolateされない。
a[0]に番兵(すごく小さい数)を入れると大丈夫。

番兵を入れるのが気持ち悪ければwhileの条件式を複雑にすればいい

while (a[j-1] && j >= 1)
でも番兵の方が少し速くなる。
しかし漸近計算量としては変わらないので、条件式を複雑にした方が論理としてはシンプルかな〜

③ 挿入整列法

$$T(N) = \sum_{i=2}^{N} i$$

$$= \sum_{i=1}^{N} i - (1) \quad \longleftarrow \ i=1 \text{ のときの } i$$

$$= \Theta(N^2)$$

$$= O(N^2) \quad \longleftarrow$$ 整列されたデータは while 文が1回で止まるので $N \times 1 = N$ になる。

10	9	8	7	6	5	4	3	2	1

$$\sum_{i=2}^{N} i = \sum_{i=1}^{N} i - 1$$

$$= \sum_{i=1}^{10} i - 1$$

$$= 55 - 1$$

$= 54$ 代入

シェルソート法(Shell sort) <= Shellさんという人の名前

挿入ソートの拡張

並んでいるやつはすごく速い

バブルソートみたいにチェックするんじゃなくて、そのまんまで並んでいるやつは速くなる。

挿入ソートの高速化

ポイント1 遅い理由

挿入候補として小さい値が出てくると遅くなる(もの凄い数のずらしが起こるから)

ポイント2 速いとき

挿入候補に大きい値ばっかり出てくる(もともとあるていど整列されているということでは？)

おおよそ小さい順に並んでいる場合はすごく速い。

おおよそ並んでいる状態にする前処理をすれば、n^2より速くなるんじゃいか。

じゃあどうやって前処理をするのか。

クイックソートでも使う方法。データを減らすと急に速くなる。

オーダーn^2のアルゴリズムには特徴がある。データを半分にすると2倍速くなるんじゃない。n^2だから4倍速くなる。

単純に半分にしてソートしたら、小から大、小から大になる。

分け方を工夫する。

一つ飛びで一つのセットにする。(偶数項の配列と奇数項の配列にわけるとか)

そうするとおおよそ全体的に左が小さくなる。

これをh(h=いくつ飛びでやったか)整列という。

ここでは2整列？

h=2整列と呼ぶの？

もっと分割量を上げるともっと早くなるんじゃないの？

h=3整列にしてみる。

これを極端に増やせばいい。

h=2整列

2 * (N/2)^2 = N^2 / 2

h=3整列

3 * (N/3)^2 = N^2 / 3

h=(9/N)整列 <= だいたいこのくらいにすると良いと

N/9 * (N/(N/9))^2 = 9N

なんと線形オーダー！

でもこれは前処理。

[前処理した後の挿入ソートの漸近計算量はどうやって計算するんだ？]

[前処理は1回だけじゃないのか]

h整列のhをだんだん小さくして、前処理を重ねていくのか。

そして最後にh=1で完璧に並べると。

そうすると、線形オーダーの処理を何回も重ねることになる。

ただ、1つずつ減らすと段数がNになるので、

Nオーダーの処理をN階繰り返すので、N^2になってしまう。

そうすると、1/3ずつへらしていくとか、半分ずつ減らしていくとか。

半分ずつ減らした場合、段数はlogNになる。

そうすると漸近計算量はNlogNになる。

ただ残念ながらこれが答えじゃない。

初めは線形で動くけど、hを小さくしていくと時間がかかったりするので、
NlogNより大きくて、N^1.5より小さくなる。

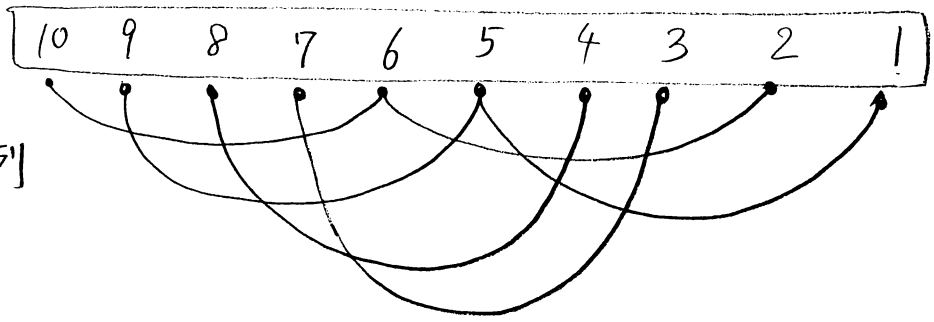
正確な漸近計算量は証明されていない。

クヌースの本だと、
h_i = 3h_{i-1} + 1
h_1 = 1
から出てくる数列の逆順をhにすると、ちょうど線形オーダーになるらしいとのこと。

実際に最悪計算量のデータを動かしてみよう
[10,9,8,7,6,5,4,3,2,1]
挿入ソートでやると、
index2からスタートして、

④ Shell sort

h=4 整列



10	6	2
2	6	10

最悪 5代入

9	5	1
1	5	9

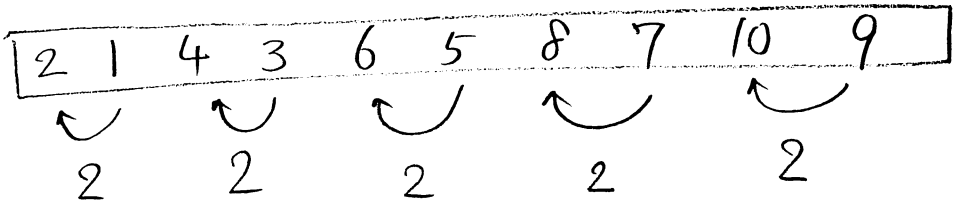
5代入

8	4
4	8

最悪 2代入

7	3
3	7

2代入



24代入

今までの4つのソート法がstableかどうかを考えましょう。

bubble sort

stable

同じだったら変えないってやれば、安定ですね。

selection sort

その場整列で実装するとstableじゃない

配列コピーで実装するとstable

insertion sort

stable

あくまでも左から行くので先にあるやつは先に左に入るからね。

Shell sort

stableじゃない

飛び飛びに並び替えているんだからぐちゃぐちゃですよね。