アルゴリズム　30－7
（1992. 11. 20）

各種ソーティングアルゴリズムの実際的評価

浅野哲夫

大阪電気通信大学工学部

寝屋川市初町１８－８

あらまし

本文では、従来から提案されているソーティングアルゴリズムを実験的に評価した結果について報告する。実験的評価においては入力の乱数列をどのようにして生成するかが重要である。本文では、ヒストグラムを大雑把に指定した上で、さらに単調列に分解したときの列の個数を指定して乱数列を生成する効率のよい方法についても報告する。

和文キーワード　ソーティング・アルゴリズム、実験的評価、乱数生成

Practical Evaluation of Sorting Algorithms

Tetsuo Asano

Osaka Electro-Communication University

18-8 Hatsu-cho, Neyagawa

Abstract

This paper presents the results of experimental evaluation of existing sorting algorithms that have been proposed so far. The most important in such experimental evaluation is how to generate input random sequences. In this paper we present an efficient algorithm for geneating random sequences of specified histogram and specified number of monotone subsequences.

英文 key words　Sorting Algorithms, Experimental Evaluation, Random Number Generation

1 まえがき

与えられたデータの列をある順序にしたがって並べ直すソーティングは、最も基本的でしかも応用範囲の広い問題として、計算機の発明以来多くの研究者によって研究されてきた。一説によると、商用プログラムの実に2割以上がソーティング処理に費やされていると言われている[3]。関心が高い分、各種ソーティングアルゴリズムの効率の解析も詳しく行われている。

本文では、ソーティングアルゴリズムを実際的な面から評価するために行った計算機実験の結果について報告する。実際的評価という意味においては実際の場面で生じるデータをテストデータとして用いるべきであるとする立場も考えられるが、そのようなデータの性質を一般的に記述することは困難である。そこで、本文では、最初にデータの頻度分布（ヒストグラム）を指定した上で乱数を生成する方法について説明する。さらに、入力列の整列性を表現するパラメータとして、本文では、入力列を単調（増加あるいは減少）部分列に分解したときの部分列の個数（攪乱要因に対応）を用いるが、任意の乱数列をそのような部分列の個数が指定された値になるように整形する効率のよい方法についても提案する。

2 テスト用入力乱数列の性質

ソーティングアルゴリズムを評価する上で入力として用いる乱数列は十分一般的なものでなければならない。例えば、UNIX等で用意されている一様乱数を用いた場合は、バケット法が威力を発揮するであろう。そこで、本文では度数分布が一様でない乱数を用いることにする。また、ソーティング法の中には入力がソートされた状態に近い程処理時間が短くて済むものがある。どれほどソートされた状態に近いかを表すパラメータとしてインバージョン (inversion) の個数をあげることができる。たとえば昇順にソートする場合を考えると、入力列 $(a_0, a_1, \cdots, a_{n-1})$ に対するインバージョンとは、$a_i > a_j$, $i < j$であるようなインデックスのペア (i, j) のことである。もちろん、既にソート済みの列にはインバージョンは存在しない。逆に、逆順にソートされた列の場合は、あらゆるペアがインバージョンであるので、それらの個数は $n(n-1)/2$ になる。したがって、一般により多くのインバージョンを含む列の方がソートされた状態から遠いと考えてもよさそうである。インバージョンについてはKnuth[3]によって詳しく調べられているが、任意の乱数列を並び替えてインバージョンの個数が入力で任意に指定した値になるような方法は筆者の知る限りでは存在しないようである。実際、適当な数を与えて、それだけのインバージョンをもつ列が存在するかどうかを判定すること自体容易ではないように思われる。

上記の理由から、本文では与えられた数列がどの程度非整列であるかを示すのに二村ら[1]によって導入された攪乱要因 (disordering factor) を用いる。これは、両隣の要素よりも大きい要素（攪乱要素）の個数として定義される。定義より、長さ n の数列の攪乱要因 d は、0以上 $(n-1)/2$ 以下であり、$0 \leq d \leq (n-1)/2$ を満たすいかなる整数 d に対しても、攪乱要因が d に等しい数列が存在する。ただし、数列の要素はすべて異なるものと仮定する。本文では任意に与えられた長さ n の乱数列を攪乱要因 d の列に変形する $O(n \log n)$ 時間のアルゴリズムを示す。

3 ヒストグラムの指定

ここでは、入力データの度数分布（ヒストグラム）が予め指定された形になるように入力となる乱数列を生成する方法について述べる。まず、簡単のため、ヒストグラムは多角形として指定されるものとする。すなわち、ヒストグラムを指定するのに直線だけを用いる。2次曲線も使えるようにすることもできるが、ここでは述べない。

ここでの目的は、$[0, 1]$ 区間の一様乱数発生ルーティンを用いて、ヒストグラムが指定されたものに近くなるような乱数の変換を考えることである。すなわち、一様乱数 rに対して関数 $g(r)$ によって $r' = g(r)$ という値を求めると、その度数分布（ヒストグラム）がちょうど入力で指定されたものになるような関数 $g(r)$ を求めたい。このような関数 $g(r)$ は、実際には別の関数 $f(r)$ の逆関数として定義される。

さて、ヒストグラムを表す多角形の各頂点から垂線を下ろすと、この多角形を垂直な2辺をもつ台形に分割することが出来る。そのような台形の上辺の左右の頂点の座標を $A(x_0, y_0)$ および $B(x_1, y_1)$ とし、線分 ABの傾きを $h = (y_1 - y_0)/(x_1 - x_0)$ とする。いま、$x_0 \leq x < x + \triangle x \leq x_1$ なる微小区間 $[x, x + \triangle x]$ を考えると、この微小区間における四角形の面積は関数の値の差 $f(x + \triangle x) - f(x)$ に等しくなければならない。したがって、図1より次式が得られる。

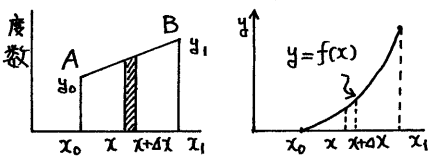


図1． 乱数変換の図式

$f(x+\triangle x)-f(x)=$

$\frac{1}{2}(hx-hx_0+y_0+h(x+\triangle x)-hx_0+y_0).$

したがって、以下のようにして関数 $f(x)$ の形を計算することができる。

$$f'(x)=\lim_{\triangle x\to 0}\frac{f(x+\triangle x)-f(x)}{\triangle x}=hx-hx_0+y_0.$$

$$f(x)=\frac{1}{2}hx^2+(y_0-hx_0)x+c.$$

上では暗黙のうちに傾き h が 0 でないことを仮定していたが、$h=0$ の場合は、関数 $f(x)$ が 1 次関数になる。すなわち、

$f(x)=y_0x+c.$

である。ただし、上式において c は定数である。

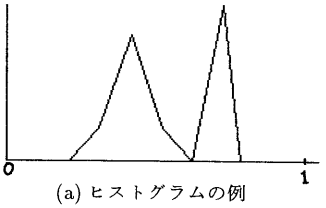
このようにしてヒストグラムを表わす各辺に対して 1 次関数または 2 次関数を計算して、全体としてそれらが連続になるように定数項 c を決定すればよい。

以上より、ヒストグラムを表わす点列 $(p_0,p_1,\ldots,p_{n-1})$ が与えられたとき、求めるべき関数 $g()$ の逆関数は、以下の手順で点と辺の交互列 $(q_0,e_0,q_1,e_1,\ldots,e_{m-2},q_{m-1})$ として計算できる。

[ヒストグラムの変換]

$q_0=p_0$; j = 0;
for(i=1; i<n; i=i+1){
　if($p_{i-1}p_i$が $y=0$ なる水平線分){
　　j = j + 1; $q_j=p_i$;
　　e_{j-1}は q_{j-1}と q_jを結ぶ水平線分
　} else if($p_{i-1}p_i$が $y>0$ なる水平線分){
　　e_jは $y=y(p_i)x+c$ なる線分。ただし、$y(p_i)$ は点 p_iの y座標値を表わす。
　　e_jが点 q_jを通るように c の値を定める。
　　さらに得られる式で $y=x(p_i)$ と置いて yの値 y_rを求める。
　　j = j + 1; $q_j=(x(p_i),y_r)$.
　} else if($p_{i-1}p_i$が傾き h の線分){
　　e_jは $y=\frac{1}{2}hx^2+(y(p_i)-hx(p_i))x+c$ なる辺。
　　e_jが点 q_jを通るように c の値を定める。
　　さらに得られた式で $x=x(p_i)$ と置いて yの値 y_rを求める。
　　j = j + 1; $q_j=(x(p_i),y_r)$.
　}
}

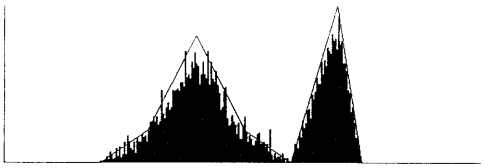
図 2 (a)のようにヒストグラムを指定したとき、関数 $f(x)$ は図 2 (b) のような曲線になる。図 2 (c) は、UNIX の乱数ルーティン random() を用いて一様乱数を生成したときのヒストグラムを示している。この乱数列を図 2 (b) の関数の逆関数で変換して得られた乱数列のヒストグラムを示したのが図 2 (d) である。



(a) ヒストグラムの例

(b) 変換用の関数

(c) 一様乱数のヒストグラム



(d) 変換後の乱数列のヒストグラム

図 2．　指定したヒストグラムをもつ乱数列の生成

4 攪乱要因の指定

ここでは、長さ n の乱数列を、入力で任意に指定された攪乱要因をもつ乱数列に変換する効率のよい方法について考える。先にも定義したように、数列 $(a_0, a_1, \ldots, a_{n-1})$ の攪乱要因 d とは、両隣の要素より大きい要素、すなわち、$a_i > \max(a_{i-1}, a_{i+1})$ なる要素 a_i の個数である。最大値が列の端にある場合、定義より、この最大値は攪乱要因に貢献しない。インバージョンの場合は、既にソート済みの数列に対してのみ値が0となったが、攪乱要因の場合は局所最小値が存在する場合でも値が0になる。

本文では、長さ n の数列を指定された攪乱要因をもつ数列に変換する $O(n \log n)$ 時間のアルゴリズムを提案する。以下ではこのようなアルゴリズムを整形アルゴリズムと呼ぶ。

4.1 整形アルゴリズム

いま、1次元配列 a[ ] に乱数が格納されているものとする。ここでの目的は、この乱数列を攪乱要因が整数 d であるような乱数列に整形（あるいは変形）することである。このアルゴリズムの基本的な考え方は、乱数列から最大値を取り除いた後、残りの乱数列をランダムに2分割して、この最大値の左右に配置するとともに、攪乱要因をこれらの部分列にランダムに分配するというものである。一般的に、部分列 a[left] - a[right] が d 個の攪乱要因を含むように変形する場合を考えよう。

まず、部分列の最大要素を求め、これを列から取り除いておくものとする。残りの部分列をランダムに2分割したときの左部分列の長さを n_l とし、右部分列の長さを n_r とする。また、左右の部分列の攪乱要因をそれぞれ d_l および d_r とする。いま、上で求めた最大要素の左に攪乱要因が d_l で要素数 n_l の部分列を配置し、その右に攪乱要因が d_r で要素数 n_r の部分列を配置するものとする。このとき、左右の部分列が空でなければ、その最大要素は攪乱要素となるから、$d_l + d_r = d - 1$ かつ $n_l + n_r = n - 1$ となるように、ランダムに d_l, d_r, n_l, n_r の値を定めればよい。左部分列が空の場合は上記の最大要素を左端に配置することになり、この要素は攪乱要素となりえない。したがって、攪乱要因はそっくり右部分列に引き継がれることになる。すなわち、$d_r = d,\ n_r = n - 1$ である。右部分列が空の場合も同様である。

攪乱要素とは両隣の要素より大きい要素のことであったから、攪乱要因を d とすると、全体の要素数 n は $2d + 1$ 以上でなければならない。したがって、全体の要素数 n が $2d + 1$ に等しい場合（これを余裕のない場合と呼ぶ）は、左右の部分列はいずれも空でないので、$d_l + d_r = d - 1$ になるように d_l と d_r を定めなければならない。この場合、さらに左右の部分列にはやはり余裕がないから、その長さ n_l と n_r は、$n_l = 2 * d_l + 1$ および $n_r = 2 * d_r + 1$ として決まってしまう。

一方、余裕がある場合、すなわち、全体の要素数 n が $2 * d + 1$ よりも大きい場合は、左右の部分列の長さをランダムに選ぶことができる。左部分列の長さ n_l は、$0 \le n_l \le n - 1$ の範囲で任意に選んでよい。n_l の値が決まれば、$n_r = n - 1 - n_l$ として n_r の値が決まる。

4.2 データ構造

上記のアルゴリズムを単純に実現すると実行時間は $O(n^2)$ 必要になるが、区間木と対称ヒープのデータ構造を用いると、実行時間を $O(n \log n)$ に削減することができる。最初に次のように再帰的に定義される区間木 $T(0, n-1)$ を用意する。

[区間木 $T(0, n-1)$ の定義]

(1) $T(0, n-1)$ の根は節点 $v[0, n-1]$ である。

(2) 節点 $v[i, j]$ は、$i < j$ のとき、それぞれ $v[i, m]$, $v[m+1, j]$ として特徴づけられる左右の子節点をもつ。$i = j$ のときは、葉節点である。

(3) 各節点 $v[i, j]$ に区間 $[i, j]$ を対応させ、さらに二つの値を蓄える。一つは、配列 a[ ] の第 i 要素から第 j 要素までの中の最大の要素のインデックス $\max[i, j]$ であり、もう一つは区間 $i, j]$ に存在する要素の個数 $s[i, j]$（当然、最初は $s[i, j] = j - i + 1$）である。

(4) この他に、配列の各要素 $a[i]$ に対して、節点 $v[i, i]$ へのポインタ $p[i]$ も用意しておく。

図3に簡単な例を示す。

上記の区間木 $T(0, n-1)$ が $O(n)$ の時間と記憶領域で構成できることは明らかであろう。この区間木を用いると、$[0, n-1]$ の任意の部分区間 $[i, j]$ における最大値は $O(\log n)$ 時間で求めることができる。これは、どのような区間も

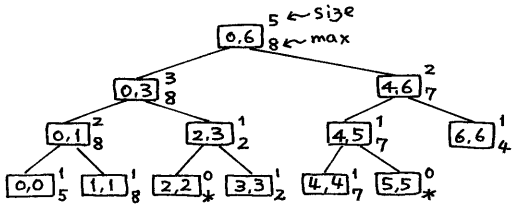


図3. (5, 8, *, 2, 7, *, 4) に対する表現.
ただし、* は要素が存在しないことを示す。

$O(\log n)$ 個の節点に対応する区間の和として表現されるからである。

以上の準備の下に、区間 $[l,r]$ の要素を並び替えて攪乱要因が指定値 d になるようにする手続き arrange(l,r,d) を説明しよう。この手続きでは、区間 $[l,r]$ における最大要素 $a[m]$ を求め、これを配列から取り除く。このとき、$i \leq m \leq j$ なるすべての節点 $v[i,j]$ について、その最大値 $\max[i,j]$ とサイズ $s[i,j]$ を変更しなければならない。これらの操作は節点 $v[m,m]$ から根まで辿ることにより実行できるから、$O(\log n)$ 時間以内に終えることができる。

さて、最大値を選んだ後、乱数によって左右の部分列の長さ n_l と n_r を定め、区間 $[l,r]$ に残っている要素のうち左から n_l 個の要素を取り出して左部分列を構成し、残りの要素でサイズ n_r の右部分列を構成する。この操作を効率よく実行するためのデータ構造として、二村らによって提案された対称ヒープを利用する。

数列に対応する（減少）対称ヒープとは次の3つの性質を有する木構造である。

(1) どの節点も高々2個の子節点しかもたない。
(2) 親節点の値は子節点の値以上である。
(3) 中間順に木を辿ると元の数列が得られる。

数列を対称ヒープを用いて表現すると、ちょうど2個の子節点をもつ節点が攪乱要素に対応している。図4は，数列 $(10,1,5,3,6,2,7,3,15,4,6)$ に対する（減少）対称ヒープである。

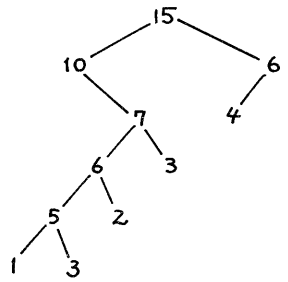


図4. 数列 (10,1,5,3,6,2,7,3,15,4,6)に対する（減少）対称ヒープ.

アルゴリズム は，以下の通りである。

(1) 注目区間 $[l,r]$ の最大値 $a[m]$ を求めて，この最大値をもつ節点 v を作る。
(2) 乱数によって攪乱要因 d を左右の部分列に振り分ける。
(3) 乱数によって左右の部分列のサイズ n_l および n_r を定め，区間 $[l,r]$ の左から n_l 個の要素を取り出して，左部分列を表す区間 $[l,k]$ と残りの区間 $[k+1,r]$ それぞれについて，対応する節点の集合を求める。
(4) 再帰的に左右の部分列を上で指定した通りに変形し，それら根を上記の節点 v の左右の子節点とする。もちろん，一方の部分列が空ならば，対応する子節点も存在しない。

注目区間を表す節点の集合を子となる部分列に渡していくことにより，上記の操作をすべて1回あたり $O(\log n)$ 時間で実行することができる。再帰呼び出しの部分を除くと，操作の回数は最大値を選ぶ回数に比例するから $O(n)$ である。したがって，全体の計算時間は $O(n\log n)$ であることがわかる。

5 実験結果

上に述べた乱数列生成法を用いて各種ソーティングアルゴリズムの評価を行った。乱数データとしてはとりあえず2種類を生成した。一方は（図2に示した）度数分布が2つの山をもつものであり，他方は一様乱数である。データ数 n は，$20,40,80,160,320,640,1280,2560$ と変化させ，それぞれのサイズ n について，攪乱要因 d を $0.1*n, 0.2*n, 0.3*n, 0.4*n$ の4通りとした。

実験の対象としたソーティング・アルゴリズムは以下の9つである。

(1) 単純なバブルソート。
(2) 単純選択法（列の最大値を列の右端の要素と交換する）。
(3) シェルソート法。
(4) ヒープソート法。
(5) クイックソート法。
(6) バケットソート法。
(7) マージソート法。
(8) 単調列マージ法（入力列を単調部分列に分割した後，部分列をマージする）。
(9) 対称整列法（二村らの対称ヒープを用いた方法 [2]）。

実験では入力列のサイズ n と攪乱要因 d のペア (n,d) に対して乱数の初期値を5通りに変えて乱数列を生成し，各ソーティング・アルゴリズムにおいて，データへのアクセス回数（参照回数と書き込み回数）を求めた。図5に図2 (a) の度数分布をもつ乱数列に対する実験結果を，図6に一様分布の乱数列に対する実験結果を示す。いずれのグラフにも4種類の折れ線グラフが示されているが，これらは上から順に，最大参照回数，最小参照回数，最大書き込み回数，および最小書き込み回数をプロットしたものである。実験の結果から，バケット法と対称整列法がよい結果をあげているようであるが，これらの方法ではデータを蓄える配列以外にも多くの記憶領域が必要になるのが欠点であろう。

6 結論

本文では各種ソーティング・アルゴリズムを実験的に評価するためのテストデータ生成法について述べるとともに，ワークステーションを用いた実験結果について述べた。計算機実験に予想外に手間取ったため，実験結果に対する検討を十分に行うことができなかった。また，バケット法やマージソート法にはまだまだ改良の余地があるものと思われる。今後は，最適なバケットサイズの決定方法等についても実験に基づいて考察を行いたいと考えている。

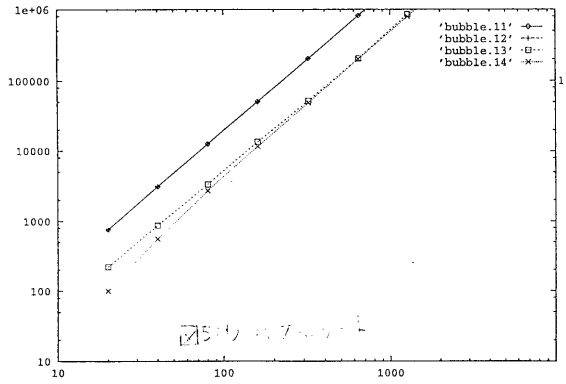
謝辞 本研究を進めるに当たってカナダ・クィーンズ大学の Prof. David Rappaport 氏に様々な助言を頂いた。ここに改めて感謝の意を表したい。

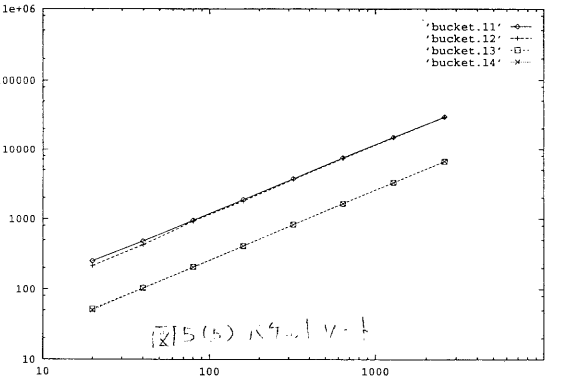
参考文献

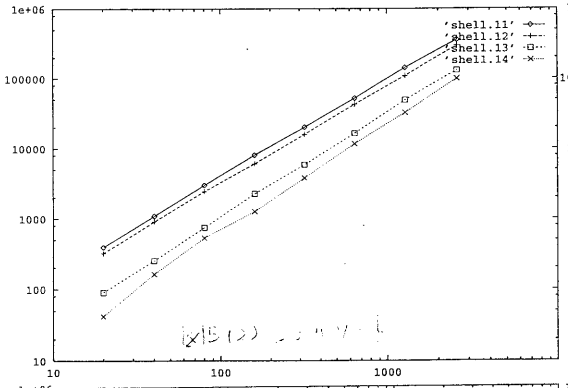
[1] 二村、筧、二村：”対称ヒープの実現とその応用”、情報処理学会アルゴリズム研究会資料 28-7，1992.7.18。

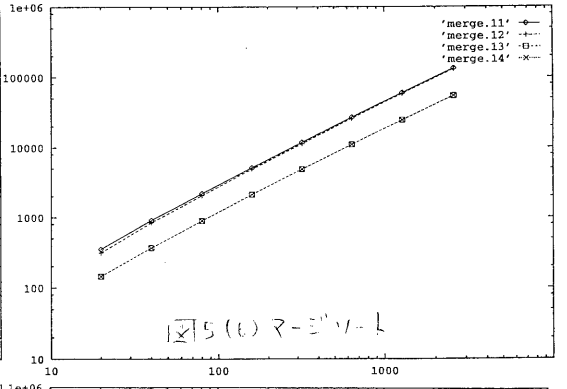
[2] 二村、二村，筧：”対称整列法:攪乱要因 d に対し $O(n \log d)$ 時間の整列法”、情報処理学会アルゴリズム研究会資料 28-8，1992.7.18。

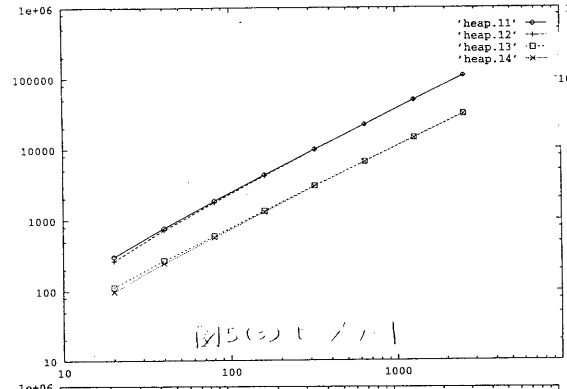
[3] Knuth, D.E.: "The Art of Computer Programming, Vol.3: Sorting and Searching", Addison-Wesley, U.S.A., 1973.

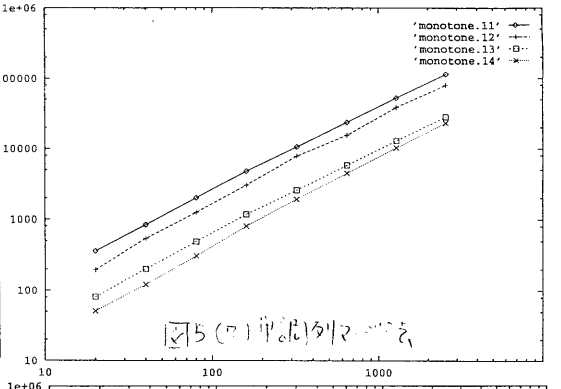

'bubble.11'
'bubble.12'
'bubble.13'
'bubble.14'

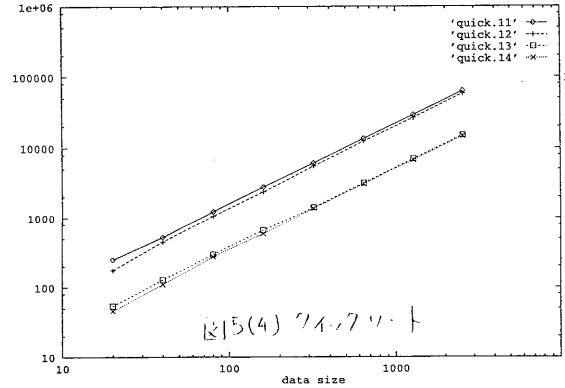

'bucket.11'
'bucket.12'
'bucket.13'
'bucket.14'

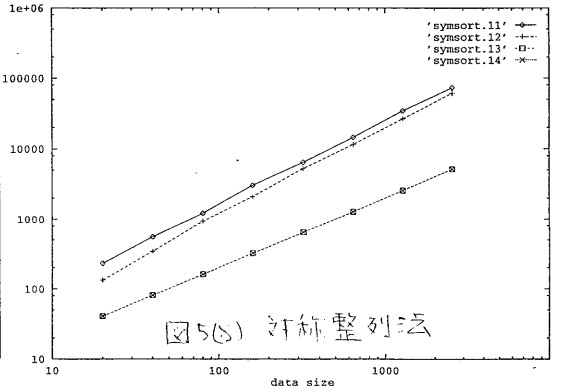

'shell.11'
'shell.12'
'shell.13'
'shell.14'

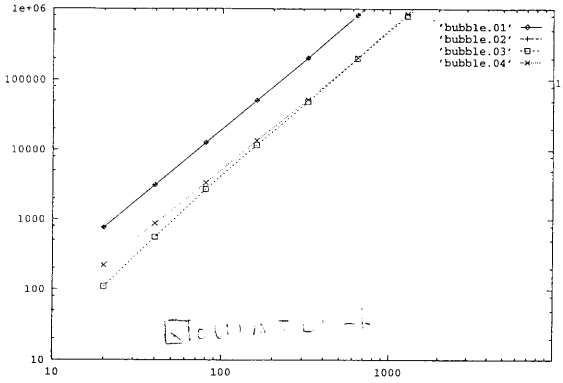

'merge.11'
'merge.12'
'merge.13'
'merge.14'

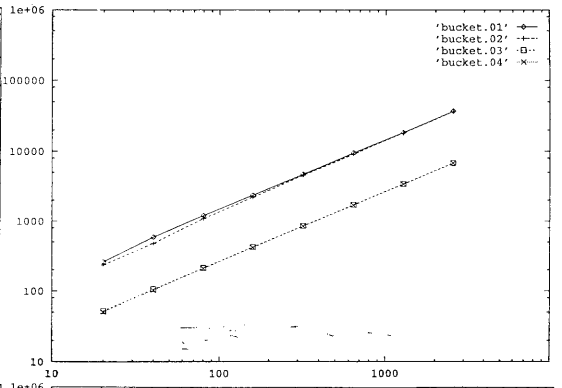

'heap.11'
'heap.12'
'heap.13'
'heap.14'

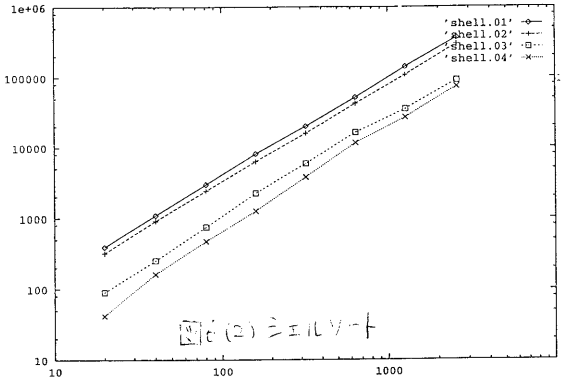

'monotone.11'
'monotone.12'
'monotone.13'
'monotone.14'

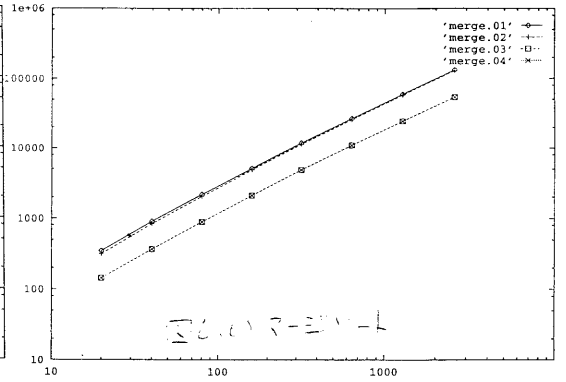

'quick.11'
'quick.12'
'quick.13'
'quick.14'
data size

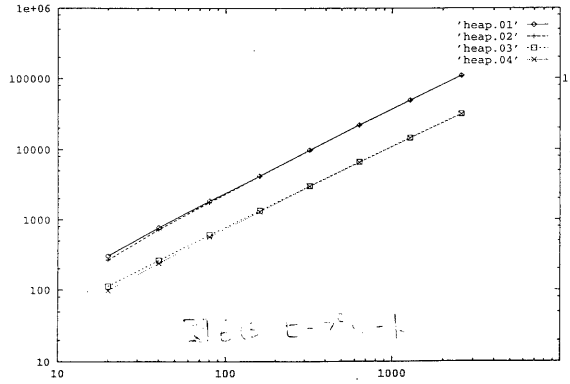

'symsort.11'
'symsort.12'
'symsort.13'
'symsort.14'
data size

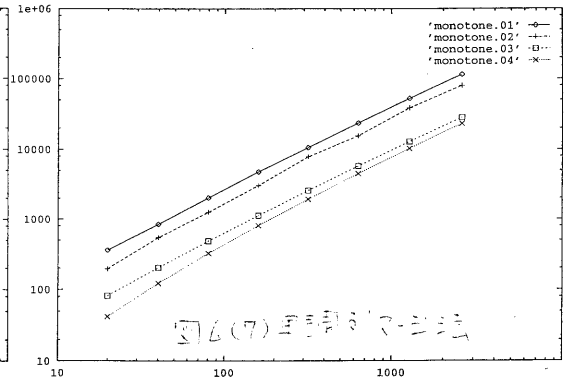

'bubble.01'
'bubble.02'
'bubble.03'
'bubble.04'
1e+06
100000
10000
1000
100
10
10
100
1000

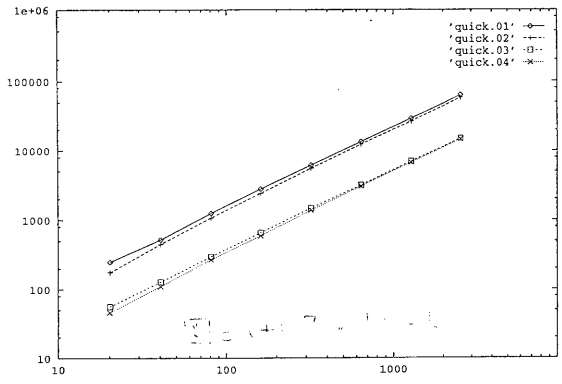

'bucket.01'
'bucket.02'
'bucket.03'
'bucket.04'
1e+06
100000
10000
1000
100
10
10
100
1000

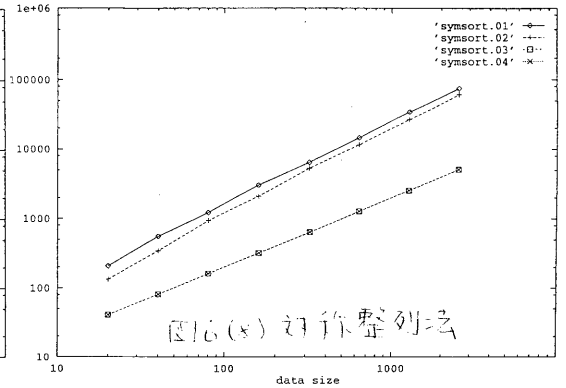

'shell.01'
'shell.02'
'shell.03'
'shell.04'
1e+06
100000
10000
1000
100
10
10
100
1000


'merge.01'
'merge.02'
'merge.03'
'merge.04'
1e+06
100000
10000
1000
100
10
10
100
1000


'heap.01'
'heap.02'
'heap.03'
'heap.04'
1e+06
100000
10000
1000
100
10
10
100
1000


'monotone.01'
'monotone.02'
'monotone.03'
'monotone.04'
1e+06
100000
10000
1000
100
10
10
100
1000


'quick.01'
'quick.02'
'quick.03'
'quick.04'
1e+06
100000
10000
1000
100
10
10
100
1000


'symsort.01'
'symsort.02'
'symsort.03'
'symsort.04'
1e+06
100000
10000
1000
100
10
10
100
1000
data size