AN EMPIRICAL STUDY OF COMBSORT AND WAYS TO IMPROVE IT

By

YUH-CHING SU

Bachelor of Business Administration
Tatung Institute of Technology
Taipei, Taiwan, R.O.C
1980
Master of Business Administration
Oklahoma City University
Oklahoma City, Oklahoma
1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1993

AN EMPIRICAL STUDY OF COMBSORT AND WAYS TO IMPROVE IT

Thesis Approved:

J. Chandler

Thesis Adviser

Blayne E. Mayfield

[illegible]

Thomas C. Collins

Dean of the Graduate College

PREFACE

This thesis studies Combsort and improves it. Like Shellsort, the average case of Combsort is very hard to analyze mathematically. Nevertheless, we attempt to estimate the average complexity of our improved versions of Combsort through the approximation of their best cases plus the overhead of the final passes according to our empirical results. We also compare this sorting method with Shellsort, Heapsort, and Quicksort to see where it stands as a new entrant to the sorting family. We also try our best to improve all these sorting methods investigated to avoid the danger of one program being more optimized than the others. I deeply thank God that He is the source of all wisdom and He is my motivation. "The plans of the heart belong to man, but the answer of the tongue is from the Lord." [Proverbs 16: 1] Above all the people I know, my parents' patience and support even up to this moment of my life, enable me to endure and persevere. Their gracious love is beyond my comprehension.

A word of thanks to my thesis adviser, Dr. J. P. Chandler, whose expertise in the area of my study and experiences accumulated from many devotional years of teaching and advising all contributed to the completion of this thesis. I also like to thank the other committee members Dr. Hedrick and Dr. Mayfield for their willingness to help which made me feel I was not alone in my work on the many cold, dark nights.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Sorting algorithms are so fundamental that they existed long before computers were invented. In the late 1950's, large high-speed computers were developed and one of the jobs that they can achieve much more efficiently than human beings can do is sorting. those simple, intuitive ways of sorting which had been used manually were transformed into sorting algorithms. An example is Insertion Sort which is a typical way that bridge players sort the cards in their hands. However, all those simple methods seem not good enough for sorting large amount of data. They all require $O(N^2)$ of steps to sort in the average case. Thus more powerful methods were developed in turn. They all perform significantly better than those elementary sorting methods. As the use of computers become more popular, the need for a faster algorithm become even more apparent. When a sorting job is implemented in a computer, people generally take advantage of the high-speed RAM as much as possible, that is, put as much as possible of the data in the main memory and perform sorting there. If all the data elements can be put in the RAM at once, we call it internal sorting, otherwise it is named external sorting which needs to consider the sequential nature of slower secondary storage like magnetic disk or tape. In this study, besides the Combsort we are going to look into, three other popular internal sorting algorithms -- Quicksort, Heapsort, and Shellsort -- will also be compared. All these methods bear some similar characteristics. They are good, in-place or almost in-place (Quicksort), and require no predetermined knowledge about the data. Their performances are superior to those found in the elementary methods, especially toward large sizes of data. In-place means that they require no extra storage. Quicksort requires a small stack for its recursive function calls. That is what we mean by "almost" in-place. And unlike some other special purpose sorting algorithms that require the user to know about the distributions of data before sorting is activated, they will all work well with random data.

Every program code in the text is written in Pascal due to its English-like syntax, thus reducing the ambiguity in understanding the demonstrated algorithm. However, the timing information is derived from the equivalent C program since C has a built-in library function to do the job easily. Another difference between the two is that I have tried to optimize the C code as much as possible toward every sorting method I compared so that I can tell whether one algorithm is faster than another (at least in C), but in Pascal, my purpose is to make the demonstration of algorithm as clear as possible. For example, I will use a global array data[ ] in every Pascal code although I know it is not a good programming practice in most cases, but I will not use "pointer" though I will show that the use of pointer will significantly speed up some algorithms. Throughout the text, the changes needed to speed up the codes are shown inside a set of double quotation marks "". Since a Pascal statement ends with a semicolon ';', the reader should not be confused with the punctuation immediately after the revised Pascal code. We will try to improve every sorting algorithm as much as possible and explain why and how we might achieve our goal. This sometimes is a long process and needs several steps to make it clear. Therefore, we may have several versions of codes concerning each algorithm. As soon as we find we cannot improve one method anymore, we will name that program by its key word with a suffix 'su' (speed-up). For example, a revised version of Heapsort will be named Heapsu. Sometimes there will be numbers after the 'su' suffix. If it is a '2', it is a two-way method (sorting from both ends). There may be more than one final version of an algorithm. We will use an alphabetic character (i.e. a, b) after the number to differentiate them. However, we realize that 'speed' is not necessarily the dominant factor to advocate an algorithm. The simplicity and ease of programming sometimes is more important. In particular, we will apply Insertion Sort to speed up the Two-way Combsort and call it Combsu2. We also make an effort in the end of the speed-up process to remove the Insertion part with another version of Combsu2 called Combsu2ni (ni stands for 'no insertion'). If this still is not clear to the reader, I shall apologize for the ambiguity of this naming convention.

Combsort was introduced by Stephen Lacey and Richard Box [15] in 1991. It is almost as simple as those elementary sorting methods, i.e., Insertion Sort, Selection Sort, and Bubble Sort, yet it is supposed to be comparable in speed to Shellsort (both use diminishing gap sizes), which, according to some authors, is the method of choice for many sorting applications (say less than a few thousand elements). The sorting behavior of Combsort resembles that of Shellsort in many ways – both use diminishing increment sequences. The optimal shrink factor suggested by the authors of Combsort [15] is 1.3, but the shrink factors of the popular implementations of Shellsort range from 2 to 3. The final pass of Shellsort is Insertion Sort but the last few passes of Combsort use the technique found in improving the original Bubble Sort by setting a dirty pass flag to signal whether the sorting job is done. The Pascal code of Combsort is shown in Program A. Program B is supposedly an improved version of Combsort called Combsort11 that resets gap 9 or 10 to 11 [15].

```
{ Assume there is a global array data[] }
{ lo, hi are lower and upper bounds of the array indices respectively }
procedure comb(lo,hi: integer);
var i,j,top,gap,v: integer;
var flag: boolean;
begin
        gap:=hi-lo+1;
        while (gap>1) or (flag=true) do
        begin
                gap:=trunc(gap/1.3);
                if gap=0 then  gap:=1;
                flag:=false;
                top:=hi-gap;
                for i:=lo to top do
                begin
                        j:=i+gap;
                        if data[i]>data[j] then
                        begin
                                flag:=true; v:=data[i]; data[i]:=data[j]; data[j]:=v
                        end
                end
        end
end;
```

Figure 1. Program A (Combsort), by Lacey & Box

```
procedure comb11(lo,hi: integer);
var i,j,top,gap,v: integer;
var flag: boolean;
begin
        gap:=hi-lo+1;
        while (gap>1) or (flag=true) do
        begin
                gap:=trunc(gap/1.3);
                case gap of
                        0:  gap:=1;
                        9:  gap:=11;
                        10:  gap:=11
                end;
                flag:=false;
                top:=hi-gap;
                for i:=lo to top do
                begin
                        j:=i+gap;
                        if data[i]>data[j] then
                        begin
                                flag:=true; v:=data[i]; data[i]:=data[j]; data[j]:=v
                        end
                end
        end
end;
```

Figure 2. Program B (Combsort11), by Lacey & Box

Like Shellsort [14], the time complexity of the running time for Combsort is difficult to analyze, thus the study of this paper will be done empirically with respect to the following aspects:

- What's the difference in performance between Combsort and Shellsort with different sizes of data?
- What might be the time complexity of Combsort for very large lists? (i.e. N^2, N*lgN, etc.)
- What are some ways to improve Combsort?

1. Can the program code be tuned up?
2. Is the Shrink Factor 1.3 the best?
3. What is the optimal initial gap size?
4. If Combsort 11, by resetting gap size 9 or 10 to 11, would improve the performance, are there any other gap sizes which could be reset to achieve better results?
5. Insertion Sort is known to be a good sorting method for very small or nearly sorted lists and it is used to improve Quicksort in practice by simply ignoring small partitions where Quicksort does poorly and use Insertion Sort to finish it up [22]. What is the optimal cutoff point for Combsort if Insertion Sort can indeed help Combsort toward the end of sorting process?
6. Are there any other alternatives which might work better than Insertion Sort to conclude the sorting for Combsort?

- Compare the final version of Combsort with the most popular methods like Quicksort, Heapsort, and Shellsort define its scope of implementations (find out where Combsort is more applicable than the rest). These comparisons will be made on different versions of each sorting algorithm, including the best implementations I could find plus careful examination of the code and make my best effort to improve them to avoid the danger that one program may be more "optimized" than the others. There are two ways to do these comparisons. First, just let the programs run on the same machine and record the time (benchmark tests). Second, analyze the program codes and decide what are the dominant factors to run them (complexity analysis). Generally speaking, the number of comparisons, the number of swaps and perhaps the number of data movements are the three major time-consuming processes for most sorting algorithms. Among them, the comparisons count is the dominant factor for all sorting algorithms we are going to look into. Although counting the comparisons alone is good enough to get a big picture concerning the performance of

these sorting algorithms, we will also do a little benchmark test to check whether our inference by the complexity analysis is reasonable. If not, figure out why!
The input data is derived from two sources:

1. The built-in pseudorandom number generator from Borland C++ that will generate integers 0-32767 will be used to test smaller sizes of data (less than 5000) while avoiding too many repetitions of keys (more than 1%). The timing information for testing small lists is done on a single user PC (A Dell 386sx at 16 MHz) to allow consistency of the environment; for larger sizes of data, the best way I could come up with is using the UNIX system command "time" to record only the user time subtracting the time used to generate the random numbers in a separate run..

2. A user-derived pseudorandom number generator that has a very long cycle and will not have any repetitions of keys on the maximum sizes of data (15 millions) we will use to test on the UNIX machine (Sequent). The asymptotic analyses are mainly based on the larger sizes of data we test here.

CHAPTER II

TUNING UP THE CODE

Before we get into the analysis of the efficiency of the sorting algorithms we will examine, let's describe some basic operations which contribute to almost all the time required to perform these sorting methods. Since we are dealing with only internal sorting in this paper, we assume an array contains all data elements to be sorted.

- Comparison - Check the order between two array elements or one array element and a fixed value. This is the dominant factor concerning the performance of sorting.
- Swap - If two elements are out of order after being compared, they need to exchange their positions (transposition). This 'swap' is usually done in a three-step operation. First, save the value of either one of the two elements to a temporary variable; second, move the other element to the position of the element being saved; third, store the variable containing the first element into the other element's position.
- Move - The second step in the 'swap' operation described above.
- Save - The first (save) and the third (store) steps in a 'swap' operation performs almost identical. We simply use the term 'save' to describe both.
- Pass - A sorting algorithm sorts a list by iterating many times through loops. Each iteration on a loop is referred to as a pass. The overhead involved in going through a pass is to initialize the loop.

From the definitions, we know that one 'swap' equals one 'move' plus two 'saves'. Some algorithms (i.e. Shellsort, Insertion Sort) do not swap two out-of-order elements right away, rather they defer this swap operation until it is clear that they are not going to be compared in the same pass again. That way we can eliminate some 'save' operations and take advantage of the efficiency of comparing against a fixed value. These arguments shall be clear when one look at the program codes for these sorting algorithms.

There are several things which will help improve the performance of Combsort and/or make it look neater in Program C (Figure 3). C code equivalence of the Pascal code (if they are significantly different) will be denoted in the parentheses throughout this paper.

1. Remove the first line inside the for loop "j:=i+gap" and insert "j" into the test statement and increment j like i. Thus every "j:=i+gap" will be replaced by "j:=j+1" except the first one, which will be replaced by "j:=gap+1".

2. If "1" is done, the test statement "i<top" can then be substituted with "j<=hi" and the variable "top" and the line "top:=size-gap" can be discarded. Better even, we can start comparison from the end to the beginning of the array as that of Insertion Sort because comparing with a small number is a little faster than comparing with a large number [14].

3. The explicit type conversion using casting in the first line inside the do-while loop is unnecessary in C since the implicit type conversion rule follows the same track at least as efficiently as the explicit ones.

4. The dirty pass flag can be got rid of by doing one test outside the inner loop which I referred to as "smart dirty pass flag." Notice the order of swap is important because otherwise we can not prove that it should work. This same technique can be applied to improve Bubblesort, too. A proof is included in the following.

Proof: One property of Bubblesort is that at each pass it will knock out one element which will not be compared at subsequent passes. Since we sort the list from top to bottom, the smallest element must reside in the first position (lo) of the list after the first pass when the gap becomes 1 (like Bubblesort). It is obvious that at the end of each following pass if any swap occurs (even when equal keys are present), the buffer variable "v" is greater than the smallest element data[lo] for we set v to the larger element of the last two elements being compared.

5. Since the outer loop test is only necessary for the last few passes when the gap length diminishes to 1, we can utilize more flexible "goto" statements for this loop control instead of a combination test "until gap=1 and v=data[lo];." We are also able to get rid of "if gap=0 then gap:=1;" on each pass of the outer loop. The use of "goto" here seems to be desirable, for otherwise, we would virtually replace those two goto statements with "repeat ....until gap=1 and v=data[lo];" on every pass of outer loop. The reader might be amused to consider how to implement a more efficient no-goto version here than the one suggested in the comments of Program C.

The revised version of code is shown in Figure 3 (Program C).

```
{ the changes needed for a no-goto version is indicated in the comments }
{ starting with an action word - add, delete, or eliminate and ending with "here" }
procedure comb1(lo,hi: integer);
label loop, out;  { delete this line here }
var i, j, gap, v: integer;
begin
        gap:=trunc((hi-lo+1)/1.3);
{       add "repeat" here }
loop:   j:=hi;  { eliminate "loop:" here }
        for i:=j-gap downto lo do
        begin
                if data[i]>data[j] then
                begin   { watch the order of swap }
                        v:=data[i]; data[i]:=data[j]; data[j]:=v
                end;
                j:=j-1
        end;
        if gap>1 then gap:=trunc(gap/1.3)
        else if v<>data[lo] then v:=data[lo] { smart flag }
        else goto out;   { delete this line here }
        goto loop;    { delete this line here }
        { add: "until gap=1 and v=data[lo]" here }
out: end;  { eliminate "out:"  here }
```

Figure 3. Program C

Program C (Figure 3) is a tuned-up version of Program A (Figure 1), which means gaining speed without the cost of losing the simplicity of the code. Furthermore, if we add one test statement "if (gap=9) or (gap=10) then gap:=11;" right before the for-loop, the code will be an improved version of Combsort11. Obviously, these improvements will not reduce the complexity of Combsort or Combsort11; they will however, speed it up some due to the elimination of unnecessary statements like "top:=size-gap", "flag:=true", "flag:=false", etc.. These represent some good speed-ups independent of the languages being used. When a more flexible (lower level) language like C is used, the improvement can be as much as 100% in speed, because the use of a faster operation (incrementation j++) in place of a slower one (addition j=i+gap), the use of pointers (we will explain this later), and putting the most frequently used variables (such as i, j) into machine registers. See the counter part of the C program code in Appendix B and Table I in the following for its timing results. (The smaller sizes don't have noticeable differences because they are too fast to be recorded precisely.)

TABLE I

AVERAGE RUNNING TIMES FOR
PROGRAMS A AND C
(IN SECONDS)

Prog.\Size	500	1000	2000	3000	4000	5000	10000
Prog. A	0.0549	0.1648	0.4395	0.6044	0.9340	1.15385	2.5824
Prog. C	0.0549	0.1099	0.2198	0.3846	0.4396	0.6044	1.3187

It can be seen in Table II that the best Shrink Factor is 1.3 through our empirical results; Figure 4 is its graph representation. We test on sizes of 500, 1000, 2000, 3000. From Table II, the Shrink Factor 1.3 is a clear winner. Then we test on a finer scale: from 1.24 to 1.35 for twelve different Shrink Factors at those four different sizes. The winner lies between 1.27 and 1.32 inclusively. The performance differences in this range are also very small - see Table III and Figure 5. It is safe to say that a Shrink Factor close to 1.3 should be good enough.

TABLE II

PERFORMANCE MEASURE FOR PROGRAM C AT VARIOUS SHRINK FACTORS - COARSER VIEW (NO. OF COMPARISONS)

SF\Size	500	1000	2000	3000
1.1	18283	43308	100374	162404
1.2	11146	26304	61712	97639
1.3	9420	23109	51779	81854
1.4	13413	28014	93997	199191
1.5	19238	74657	295088	932726

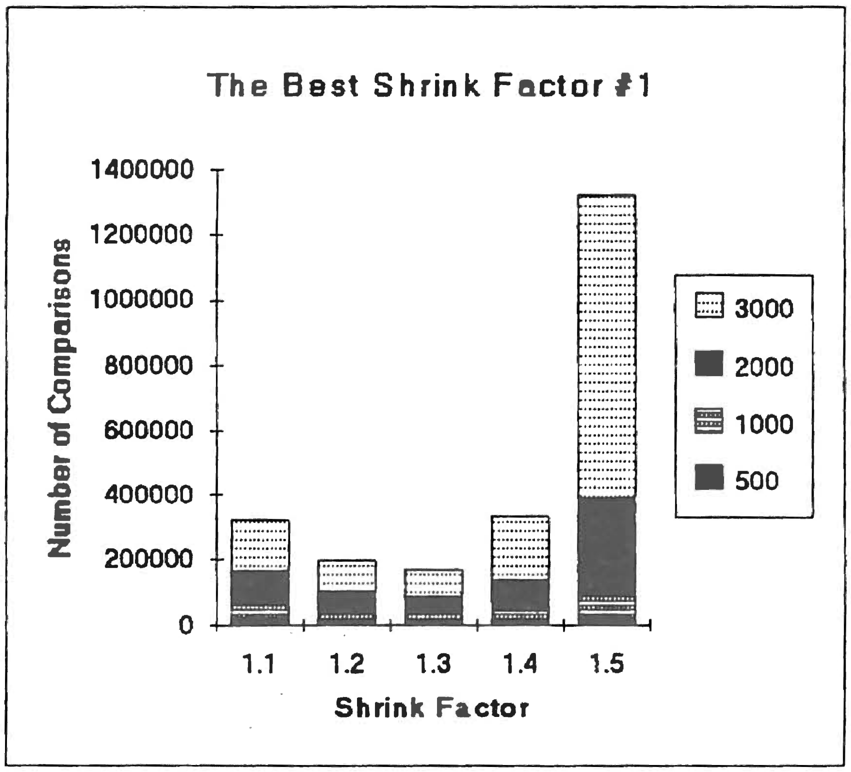

Figure. 4 Bar Chart for Table II

TABLE III

PERFORMANCE MEASURE FOR PROGRAM C AT VARIOUS SHRINK FACTORS - FINER VIEW (NO. OF COMPARISONS)

SF\Size	500	1000	2000	3000
1.24	10224	23810	54947	87800
1.26	10077	23206	52369	85143
1.27	9650	23054	51062	84063
1.28	9614	23181	51522	84460
1.29	9773	21696	50759	82720
1.30	9420	23109	51779	81854
1.31	9278	22914	51399	83359
1.32	9520	22621	47402	83671
1.33	8869	22605	50580	86345
1.34	9852	32592	51758	80615
1.35	10157	22883	55125	90174

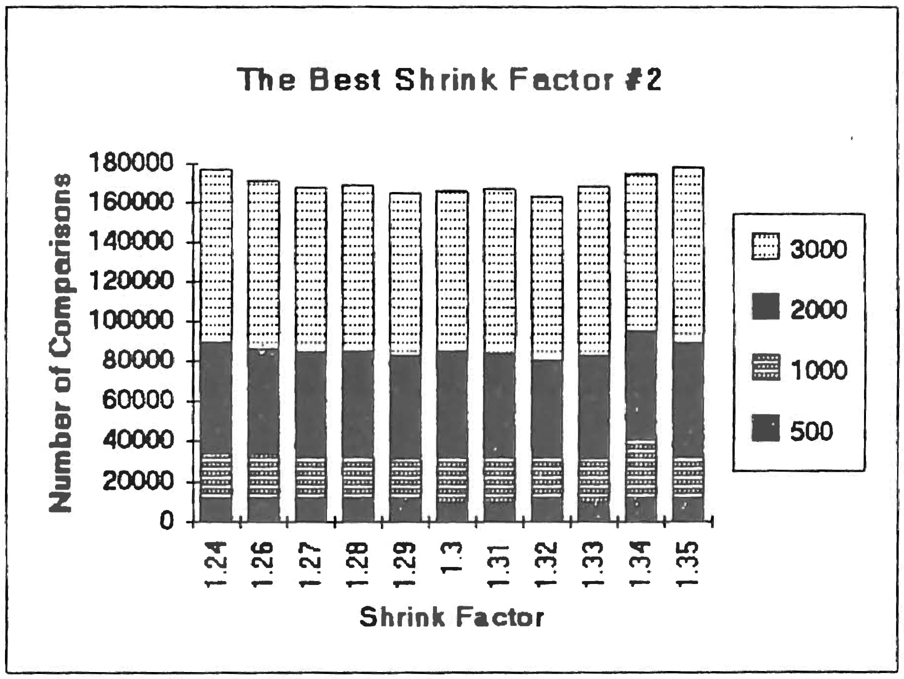

Figure 5. Bar Chart for Table III

The initial gap N/1.3 is too large because for a random list the average distance an element will move after sorting is about size/3. Therefore, size/3 as the initial gap might be a good estimate and this presumption will be verified later. Simply replace the first line in Program III with "gap:=(hi-lo+2) div 3;", and we are done with resetting the initial gap size. Note that we use "hi-lo+2" instead of "hi-lo+1" so that a test statement "if gap=0 then gap:=1;" can be saved for size=2. Though the optimal initial gap size is hard to decide, it doesn't make much difference in between size/2 and size/3.5. The improvement is marginal over the original one. See Table IV and Figure 6. There the X-axis is the Initial Gap Factor (IGF) and size/IGF equals the initial gap size.

TABLE IV

PERFORMANCE MEASURE FOR PROGRAM C
AT VARIOUS INITIAL GAP FACTORS
(NO. OF COMPARISONS)

INP\Size	500	1000	2000	3000
1.3	9420	23109	51779	81854
1.7	9255	24295	51945	80609
2	9551	21370	50509	75452
2.5	10461	20506	49974	81833
3	9264	21490	45957	78508
3.5	8420	20999	52356	73427
4	11329	30141	45669	81391

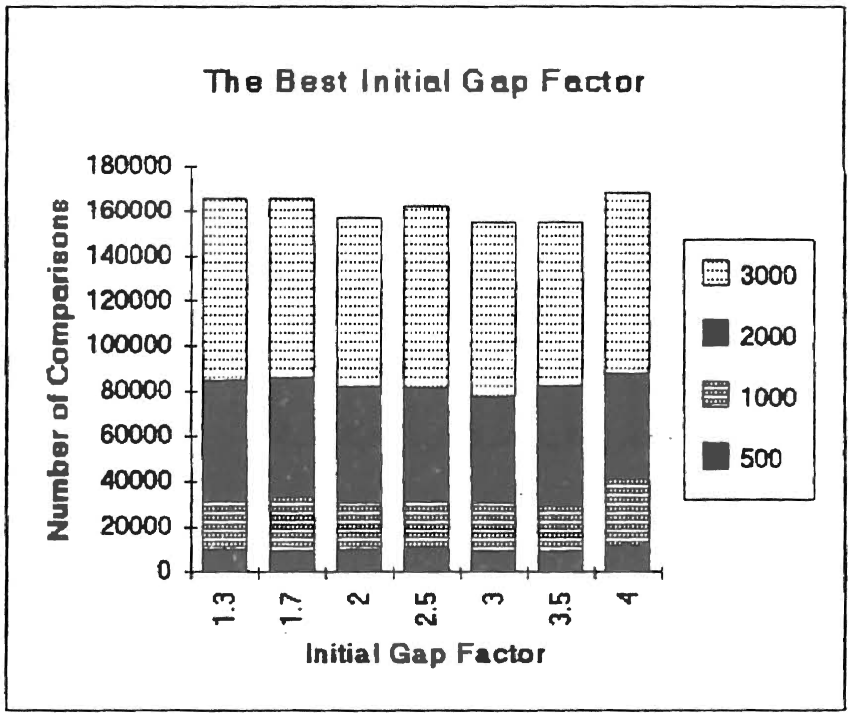

Figure 6 Bar Chart for Table IV

A desirable way of making an efficient increment sequence for Shellsort is to minimize the occurrence of one increment being a perfect divisor of another (non-relatively prime) because one famous theorem about Shellsort is: "If a k-ordered list is h-sorted, it remains k-ordered." [14] Although this theorem does not apply to Combsort exactly (Property 4.1a, Chapter IV), it still has some impact on Combsort (Property 4.2, Chapter IV). Combsort11 resets the gap sizes 10 and 9 to 11 and gains a little on the average (not always) - See Table V. It is hard to believe at the first glance that Combsort11 would be a better performer than the original Combsort, because after 11, the gap sequence is 8-6-4-3-2-1 which breaks the rule of "relatively prime" found in Shellsort. Now the possible sequences after gap<30 are the following:

Sequence	#1.	29-22-16-12-	9	-6-4-3-2-1
Sequence	#2.	28-21-16-		
Sequence	#3.	27-20-15-	11	-8-6-4-3-2-1
Sequence	#4.	26-20-		
Sequence	#5.	25-19-14-	10	-7-5-3-2-1
Sequence	#6.	24-18-13-	10	
Sequence	#7.	23-17-13-		

Notice that all these sequences bear non-relative-primeness in some way. But how much does it matter? The idea of getting a relatively prime gap sequence is to avoid redundant comparisons. The major departure of Combsort from Shellsort, as we mentioned earlier, is that the innermost loop of Shellsort is replaced by a simple comparison. Therefore, the redundant comparisons for Combsort could only happen in the direction of sorting while Shellsort inserts each item backwards in a loop and may encounter redundant comparisons in both directions. For example, if we want to sort a list "1-3-2" into ascending order, and the gap sequence for both Combsort and Shellsort is "2-1", the first pass is to compare 1 and 2, and no swaps occur. If the sorting direction is from left to right, Combsort would not have a redundant comparison at gap=1 but Shellsort would have compared 1 and 2 again after swapping 2 and 3. If sorting from right to left, both would have a redundant comparison (1 and 2) after swapping 2 and 3. Another notable difference is that Combsort uses a much smaller Shrink Factor than Shellsort does (1.3 vs. 2 to 3) and, according to some of my test results, the number of transpositions for Combsort is about 20% less than that of Shellsort. I also did some tests on Shellsort, and my empirical results shows that the number of elements travel more than seven gaps far in one pass is negligibly small - see Table V.

TABLE V

MOVING DISTANCE FREQUENCY TESTS FOR
ONE SHELLSORT (SHELLSU15)
(OCCURRENCES)

Size	1500	15000	150000	1500000
move=0	9806	141798	1856260	22931916
move=1	8842	125422	1641921	20355701
move=2	1475	20615	267300	3265928
move=3	774	11141	144371	1814454
move=4	393	5921	79667	1005222
move=5	227	3267	41774	526345
move=6	122	1537	20193	254789
move=7	49	669	9129	114363
move=8	17	281	3807	45630
move=9	9	107	1390	16269
move=10	0	36	450	5127
move=11	0	14	106	1487
move>11	0	2	37	449

It is reasonable to think, since Combsort has a smaller number of transpositions, that most elements should travel less than seven (say five or six) gaps in one pass.

Accordingly, the only times that the relative-primeness would have some good impact are when a gap is half of any of the previous gaps or is one-third of any of the previous gaps. I call the former kind of situation "immediately non-prime" (INP) and the latter "next to INP" (NINP). In relation to Table V, a redundant comparison in the INP situation is that when move=1 and in the NINP case when move=2. If the behavior of moving frequency for Combsort is similar to that of Shellsort (future work), the INP should have about six times more frequent occurrence than the NINP (see Table V). For those sequences stated above, the immediate-non-primeness (or INP) they inherited are:

For 9:	12-6-3, 4-2
For 10:	14-7 (sometimes), 10-5
For 11:	8-4-2, 6-3

Two things concerning INP are: INP counts and the magnitude of the size when an INP occurs. It is straightforward to see that the greater the INP counts the worse the performance will be. The reason why the latter would affect the performance is also easy to explain. Every time an INP happens on a certain gap, it could cost that gap size of inversions if later on it needs to be taken care of by a smaller gap. This inference can explain why resetting 10 at Sequence #5 to 11 could gain good speed since it could avoid the INP of 14-7 but little or no improvement at Sequence #6 or #7 since it would introduce an additional INP count. Sequence #6 in particular is the worst case scenario for Combsort11 because we will also get three NINPs (24-8, 18-6, and 6-2) in addition to one INP. Our test results support these arguments. However, resetting 9 to 11 for Sequence #1 shouldn't be a good idea since it will create the INP of 22-11 and does not reduce the INP counts. Nevertheless our empirical results contradict this inference. See Table VI and Figure 7 in the following. Note the five sets of data we chose represents Sequences #1, #2, #5, #6, and #7 respectively around 3000. Sequences #3 and #4 won't introduce any changes for Combsort11 from Combsort.

TABLE VI

PERFORMANCE MEASURE FOR COMBSORT AND COMBSORT11 AT VARIOUS GAP SEQUENCES (NO. OF COMPARISONS)

Size (Seq.)	2800 (#1)	2730 (#2)	3100 (#5)	3050 (#6)	3000 (#7)
Comb	74400	72065	83296	76468	78508
Comb11	68793	67891	77402	77681	74904

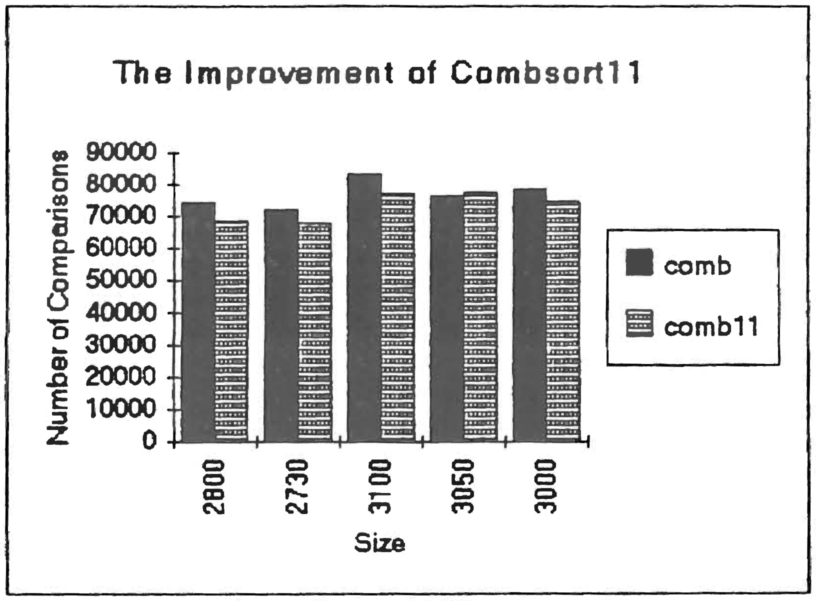

Figure 7 Bar Chart for Table VI

Why has resetting 9 or 10 to 11 almost always improved the performance of Combsort regardless of the argument we just made? If we look at the lists when the gap size reduces to 1, we will find very few inversions left, but they might still need quite a few passes to complete because they degrade to Bubblesort which might fix only one inversion at each pass (see Chapter IV for the worst case discussion). Resetting 9 to 11 will delay the degradation process by exchanging gap 9 with two gaps 11 and 8. In other words, we spend one more pass before it degrades to Bubblesort, and that's where the performance gain comes from. Although resetting 10 to 11 will also delay the degradation process by one pass, the performance gain will be offset by the increasing INP counts and the magnitude of INP as the previous arguments stated. So a better way of resetting gap size is that after the gap size reduces to 9, simply decrement the gap size by one on each pass. The empirical results are indicated in Table VII and Figure 9. An improved version of Combsort (Combsu) is in Figure 8.

```
procedure combsu(lo,hi: integer);
label loop, out;
var i,j,gap,v: integer;
begin
        gap:=(hi-lo+2) div 3;  { +2 is necessary for size=2 to work }
loop:   j:=hi;     { OT. we will need one additional if here  }
        for i:=j-gap downto lo do
        begin
                if data[i]>data[j] then
                begin
                        v:=data[i]; data[i]:=data[j]; data[j]:=v
                end;
                j:=j-1
        end;
        if gap>10 then gap:=trunc(gap/1.3)
        else if gap>1 then gap:=gap-1
        else if v<>data[hi] then v:=data[hi] {smart flag }
        else goto out;
        goto loop;
out: end;
```

Figure 8. Program D (Combsu)

TABLE VII

PERFORMANCE MEASURE FOR PROGRAM B AND D AT VARIOUS SEQUENCES (NO. OF COMPARISONS)

Prog.\Size	2800 (#1)	2730 (#2)	2650 (#3)	2600 (#4)	3100 (#5)	3050 (#6)	3000 (#7)
Comb11	68793	67891	63265	63890	77402	77681	74904
Combsu	68238	66258	64055	63106	76160	74935	74602

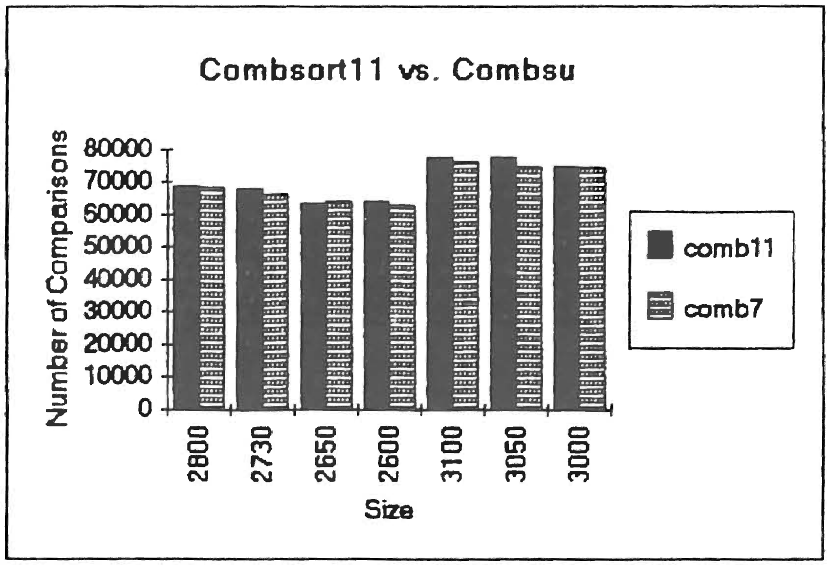

Figure 9. Bar Chart for Table VII

Though the improvement in comparisons of Combsu over Combsort11 is only marginal, it introduces some good speed-ups (about 20%) by reducing the overhead and makes the program look neater. Some more speed-up can be achieved by introducing Insertion Sort toward the end of Combsort when it becomes as inefficient as Bubblesort. This improvement is significant although it almost doubles the size of code. The performance gain is about 15% to 20%. See Table VIII. The motivations to use Insertion Sort are threefold:

1. Insertion Sort is good at almost sorted lists where Combsort does poorly and degrades to Bubblesort when the gap size is one.

2. There is no way to eliminate all INPs discussed above when the gap size is less than 10 without jeopardizing the degradation process of the Shrink Factor becoming one. For instance, we can reset 6 to 5 on 15-11-8-6-4-3-2 and produce the sequence 15-11-8-5-3-2 which eliminates all INPs. Nevertheless, by doing that we will speed up the degradation process of the Shrink Factor by one pass.

3. When Insertion Sort is used, the dirty pass flag can be completely got rid of, the test and reset statement "if gap=0 gap:=1" or "if gap>0 ...." is no longer needed, and the need of two "goto" statements (for efficiency) to replace the compound test outside the loop is nonexistent. One important thing about applying the Insertion Sort is when to use it; that is, what is the best cutoff point for Combsu to use Insertion Sort. The results in Table VIII and Figure 10 show that a cutoff point around 6 is generally a good performer. The number of moves becomes more significant when the cutoff point for Insertion Sort is larger. Program E (Figure 11) is the resulting code. Note that the Insertion Sort we use there has exploited Property 4.5 (in Chapter IV) to get rid of the need for a sentinel key.

TABLE VIII

PERFORMANCE MEASURE FOR COMBSU AT VARIOUS CUTOFF POINTS FOR INSERTION (NO. OF COMPARISONS)

Cut\Size	500	1000	2000	3000
0	8807	20583	47152	74602
4	7186	17038	38602	63002
5	6919	16880	38594	62620
6	6907	16835	38287	62563
7	6993	16694	38224	62299
8	6961	16713	39176	62281

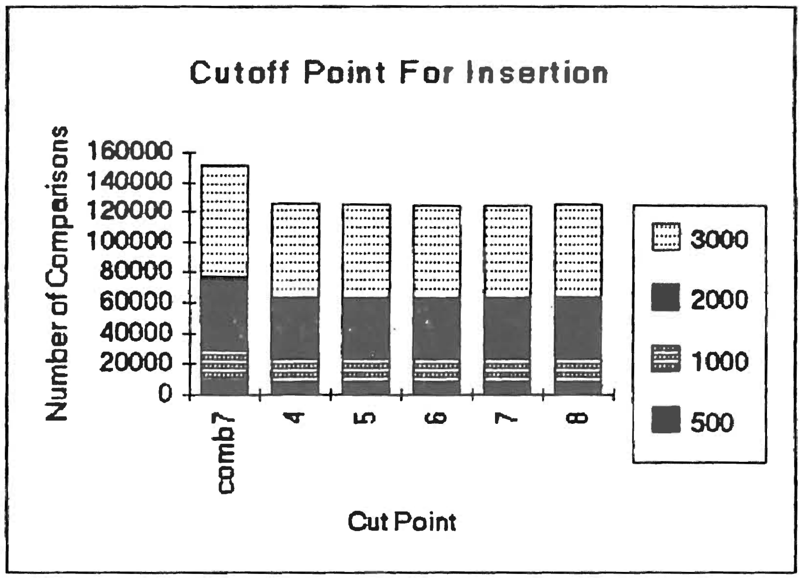

Figure 10. Bar Chart for Table VIII

```
procedure combins(lo,hi: integer);
var i,j,gap,v: integer;

procedure insertion(lo,hi: integer);
var i,j,v: integer;
begin
        if hi-lo>6 then j:=lo+6   { 6 is the cut off gap size }
        else j:=hi;
        for i:=j-1 downto lo
        do begin   { send the sentinel here }
                if data[i]>data[j] then
                begin   v:=data[i]; data[i]:=data[j]; data[j]:=v          end;
                j:=i
        end;
        for i:=lo+2 to hi do
        begin
                j:=i; v:=data[i];
                while data[j-1]>v do
                begin
                        data[j]:=data[j-1];
                        j:=j-1
                end;
                data[j]:=v
        end
end;
begin
        gap:=(hi-lo) div 3;
        while gap>6 do
        begin
                j:=hi;
                for i:=hi-gap downto lo do
                begin
                        if data[i]>data[j] then
                        begin   v:=data[i]; data[i]:=data[j]; data[j]:=v          end;
                        j:=j-1;
                end;
                gap:=trunc(gap/1.3);
        end;
        insertion(lo,hi)
end;
```

Figure 11. Program E (Combins7)

After applying Insertion Sort to our Combsort, we need to re-plot all the optimal values discussed so far - Shrink Factor, Initial Gap Size, and the resetting of certain gap sizes to reduce the INPs. However, everything remains the same after some experiments similar to what we have done to this point. It is easy to explain why the first two factors do not change since they are more or less independent of the Insertion phase - they do not interfere with the work which Insertion Sort is going to do. But what about the last factor? Of course we don't need to reset 9 or 10 to 11 as Combsort11 did or change the gap sequence like Combsu since we have cut it off by the Insertion Sort at a gap size of 6. Yet we can still think about reducing the INPs at some larger gap sizes. It would be trivial to reset a couple of gap sizes, because the algorithmic improvement of resetting gap sizes would be only marginal, but the overhead involved and the complication of the programming code would lessen its significance. Nonetheless, I tried to eliminate the INPs by incrementing or decrementing every even number gap size, hoping there would be some performance gain. The programming is easy: simply add one line of code "gap:=gap+(gap+1) mod 2;" or "gap:=gap-(gap+1) mod 2;" after "gap:=trunc(gap/1.3);". The incrementing version performs poorer but the decrementing one performs a little better (about 3%) than the original one. See Table IX and Figure 12.

TABLE IX

PERFORMANCE MEASURE FOR PROGRAM E AND
ITS INCREMENT AND DECREMENT VERSION
BY RESETTING EVEN GAPS TO ODD
(NO. OF COMPARISONS)

Prog.\Size	500	1000	2000	3000
Combins7	6907	16835	38287	62563
even+1	7208	17321	40519	63953
even-1	6885	15791	37389	60078

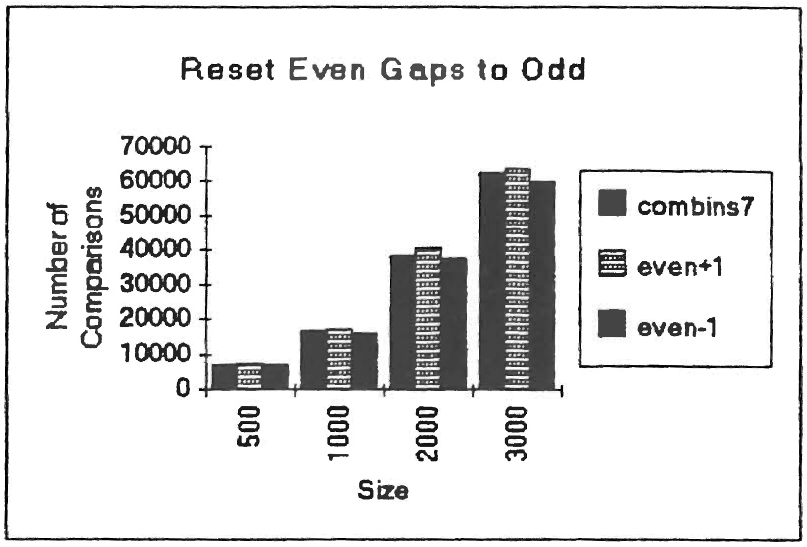

Figure 12. Bar Chart for Table IX

Here comes the prompt again! "Do we need to re-plot those optimal values again since we decremented the even gaps?" The answer is a resounding "yes". Actually, we already had a clue that we might want to increase the Shrink Factor a little bit when decrementing each even gap, which has an effect of slightly increasing the Shrink Factor and performs better. After a series of tests like we did previously, I found the optimal Shrink Factor to lie around 1.4 which is a significant improvement (about 12%); the best cutoff point for the Insertion Sort is 3 which has marginal effect (1%); and the IGF remains pretty much the same. See Table X and Figure 13. A program for this implementation is too trivial to be included here since we only need to change 1.3 to 1.4 and 6 to 3 in addition to the one added line (reset even to odd) mentioned earlier for Program E.

TABLE X

PERFORMANCE MEASURE FOR PROGRAM E AT TWO SHRINK FACTORS - 1.3 AND 1.4 (NO. OF COMPARISONS)

SF\Size	500	1000	2000	3000
1.3	6885	15791	37389	60078
1.4	6192	14329	32787	53392

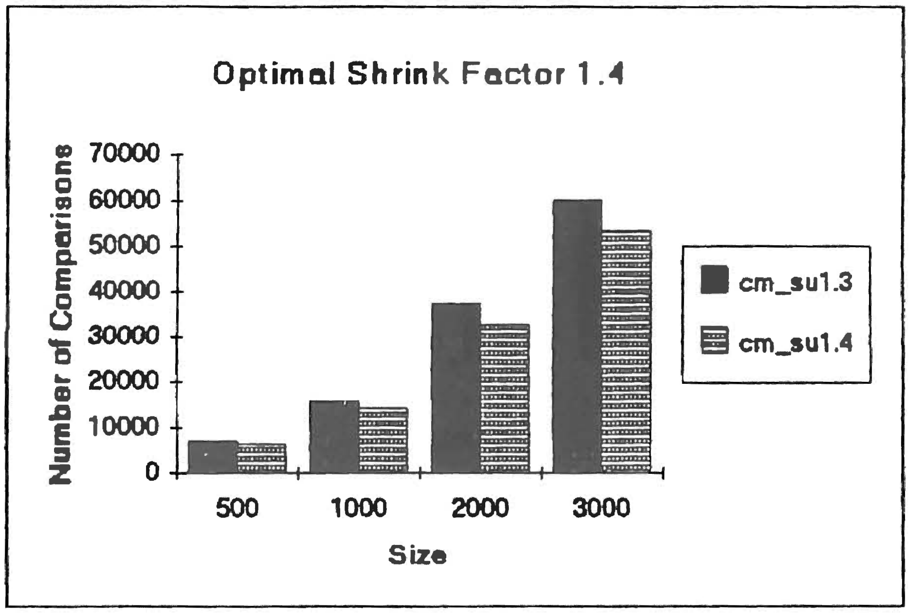

Figure 13. Bar Chart for Table X

Although Lacey & Box [15] referred to Combsort as a variation of Bubblesort by allowing distant comparisons along the list, algorithmically speaking it is a lot more like Shellsort. No doubt their idea of Combsort must have originated from Bubblesort. In fact, what is common for Combsort and Bubblesort is that they do not defer the transposition like Shellsort once an out-of-order is located. One improved version of

Bubblesort is called Cocktail Shaker Sort in which alternate passes go in opposite directions, so that any element that needs to traverse a long way to its final position toward either end of the list has a chance to reach its final destiny faster. That sparked my last attempt to improve Combsort for the time being. After a series of tests on different configurations at the same set of data we used before, I found that the optimal IGF remains at 3, the Shrink Factor increases to 1.44, and the cut-off point for Insertion Sort decreases to 3. These changes really make sense since the Two-way Combsort did a better job in sorting; it allows a little bigger jump on the gaps used and wouldn't need the Insertion Sort until the last moment when it degenerates into the notorious Bubblesort. Program VI is my implementation for it. A sentinel for the insertion part can be adjusted according to the cutoff point used.

```
procedure combins2(lo,hi: integer);
label: out;
var i,j,gap,v: integer;

begin
        gap:=(hi-lo) div 3;
        i:=lo;
        repeat
                for j=lo+gap to hi do
                begin
                        if data[i]>data[j] then
                        begin
                                v:=data[i]; data[i]:=data[j]; data[j]:=v
                        end
                        i:=i+1
                end;
                if (gap<=3) then goto out;
                gap:=trunc(gap/1.44);
                gap:=gap-(gap+1) mod 2;
                j:=hi;
                for i:=hi-gap downto lo do
                begin
                        if data[i]>data[j] then
                        begin
                                v:=data[i]; data[i]:=data[j]; data[j]:=v
                        end;
                        j:=j-1
                end;
                gap:=trunc(gap/1.44);
                gap:=gap-(gap+1) mod 2
        until gap<2;
out:  insertion(lo,hi)
end;
```

Figure 14. Program F (Two-way Combsort)

According to my sampling, this revision did help reduce the number of comparisons except for the smallest data we tested at 500 - see Table XI. The improvement seems to increase as the size of data gets larger. We will verify that in Chapter IV. However, the number of transpositions (swaps) also increased for this Two-way Combsort (as it is called from now on) - see Table XI.

TABLE XI

PERFORMANCE MEASURE FOR PROGRAM D AND PROGRAM F AT FOUR VARIOUS SIZES (NO. OF COMPARISONS)

Prog.\Size	500	1000	2000	3000
Combsu	6192	14329	32787	53392
Combsu2	6294	14291	32577	52726

TABLE XII

PERFORMANCE MEASURE FOR PROGRAM D AND PROGRAM F AT FOUR VARIOUS SIZES (NO. OF SWAPS)

Prog.\Size	500	1000	2000	3000
Combsu	2405	5398	12204	19884
Combsu2	2734	5989	13740	22202

These tests are by no means complete, especially toward large size of lists, but they all agree with our inferences. We'll come back in Chapter IV and see a more complete test for Combsort at sizes up to 15 million.

CHAPTER III

COMPETITIONS

In this chapter, we are going to look into some popular methods of internal sorting and compare them with the refined version of Combsort and see where we stand in performance as a new entrant to the sorting family. A word of caution before our comparison analysis is that different methods will bear different overheads other than the comparisons count, although it still dominates. Thus we will do some benchmark tests at the end of this chapter and verify whether our results need to be adjusted accordingly.

The Sibling - Shellsort

Shellsort is an improved version of Insertion Sort which allows exchanges of elements that are not adjacent. The significant factor in Shellsort is the selection of a good increment sequence. Two popular sequences presented here are Hibbard's (Program G) and Knuth's sequences (Program VIII). The first line of code in these two programs, the "repeat until loop" (for loop), is to determine a good starting gap. The rest of the code is very similar to Combsort. The only significant difference is that the innermost loop in Shellsort is replaced by a single comparison (if statement) and a possible "swap" in Combsort. There are other increment sequences which would lead to a more efficient sort. But it is difficult to beat the following program by more than 20%. And the conjecture for the complexity of Shellsort for a large size of data is either $N*(\log_2 N)^2$ or N^k where $1<k<2$. For example, two conjectures about the sequence Knuth suggested are $N^{5/4}$ and $N*(\log_2 N)^2$ in the average case [21]. The worst cases for both sequences are the same $(N^{3/2})$ and can be proved. Robert Sedgewick has improved

Shellsort by several good increment sequences which lower both the upper bound and apparently the average asymptotic complexity, but none of these improvements he suggested produce an O(N* $\log_2$ N) complexity like that of Quicksort (average case) or Heapsort. The authors of Combsort claimed that Combsort has (N*$\log_2$ N) complexity. If that is true, Combsort will run faster than any version of Shellsort available now, for a fairly large N. We will investigate this in Chapter IV. The best known sequence which Sedgewick found was in the form of $(9*4^i-9*2^i+1) \cup (4^i-3*2^i+1)$. I simply used a calculator and initialized it up to the maximum size I am going to test in this paper, in the beginning of the program in Program IX. Notice that the counterpart C program in Appendix II wraps around the innermost loop with an if statement, so that the unconditional save can be avoided [2] and the first "j>=h" is also eliminated.

```
procedure shellh(lo,hi: integer);
var i,j,gap,v: integer;
begin
        gap:=1;
        repeat gap:=2*gap until gap>(hi-lo)/4;
        gap:=gap-1;
        repeat
                gap:=gap div 2;
                for i:=gap+1 to hi do
                begin
                        v:=data[i]; j:=i;
                        while j>gap and data[j-gap]>v do
                        begin
                                data[j]:=data[j-gap];  j:=j-gap
                        end;
                        data[j]:=v
                end
        until gap=1;
end;
```

Figure 15.   Program G by T. N. Hibbard

```
procedure shellk(lo,hi: integer);
var i,j,gap,v: integer;
begin
    gap:=1;
    repeat gap:=3*gap+1 until gap>hi;
    repeat
        gap:=gap div 3;
        for i:=gap+1 to hi do
        begin
            v:=data[i]; j:=i;
            while j>gap and data[j-gap]>v do
            begin
                data[j]:=data[j-gap]; j:=j-gap;
            end;
            data[j]:=v
        end
    until gap=1;
end;
```

Figure 16. Program H by D. E. Knuth

```
procedure shells(lo,hi: integer);
var i,j,k,gap,v: integer;
var ary:  array[1..21] of integer;
begin
        ary[1]:=1; ary[2]:=5; ary[3]:=19; ary[4]:=41; ary[5]:=109; ary[6]:=209;
        ary[7]:=505; ary[8]:=929; ary[9]:=2161; ary[10]:=3905; ary[11]:=8749;
        ary[12]:=16001; ary[13]:=36449; ary[14]:=64769; ary[15]:=146305;
        ary[16]:=260609; ary[17]:=587521; ary[18]:=1045505; ary[19]:=2354689;
        ary[20]:=4188161; ary[21]:=9427969;
        k:=20;  gap:=(hi-lo) div 3 * 2;
        while ary[k]>gap do  k:=k-1;
        repeat
                gap:=ary[k];  k:=k-1;
                for i:=gap+1 to hi do
                begin
                        v:=data[i]; j:=i-gap;
                        while (j>=lo) and (data[j]>v) do
                        begin
                                data[j+gap]:=data[j];
                                j:=j-gap
                        end;
                        data[j+gap]:=v
                end
        until gap=1
end;
```

Figure 17. Program I by R. Sedgewick

In Table XIII and Figure 18, we compare the result from Table X (Combins) with these three versions of Shellsort since we have yet to find out whether the Two-way Combsu is practical. This version of Combsort seems to fall only behind Sedgewick's sequence and takes second place by the comparisons count.

TABLE XIII

PERFORMANCE MEASURE FOR COMBINS AND SOME FAMOUS SHELLSORT IMPLEMENTATIONS (NO. OF COMPARISONS)

Prog.\Size	500	1000	2000	3000
cm_su1.4	6192	14329	32787	53392
shellh	6289	14901	34643	56531
shellk	5917	14388	33180	54998
shells	6100	13847	31524	50395

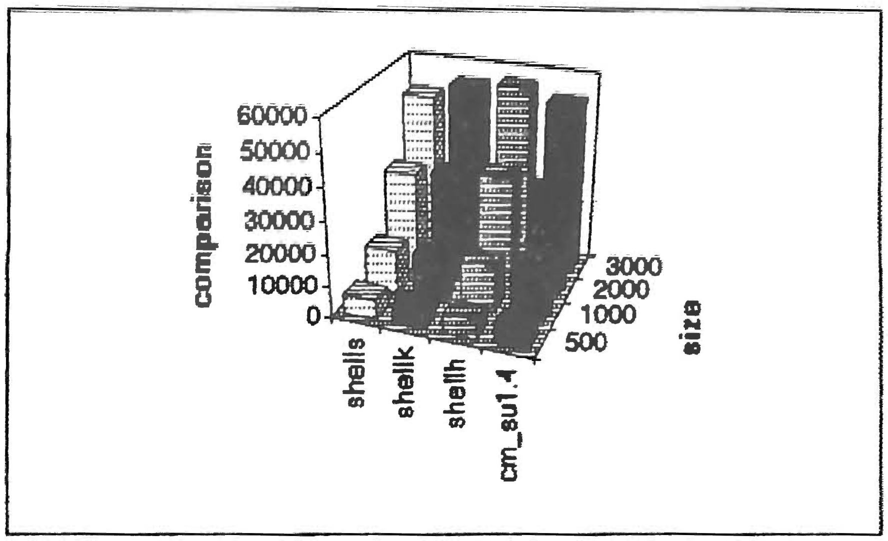

Figure 18. 3-D Bar Chart for Table XIII

The Elite - Quicksort

Since 1960 when C. A. R. Hoare invented Quicksort, it has become the most popular general-purpose sorting algorithm. In many cases, it is imbedded in the library routines of computer languages, i.e. C language. Through the years, research efforts on sorting algorithms have been put more on Quicksort than on any of the other sorting methods. To its credit Quicksort is the fastest general purpose, internal sorting algorithm so far (in the average case), and it is not difficult to implement. Its behavior is subject to mathematical analysis and precise performance statements can be made without much argument. The basic algorithm for Quicksort is "divide-and-conquer". It works by dividing a list into two parts, then sorting the parts independently, iterating the divide-and-conquer process until the list is sorted. Ideally, we like to see two equal-length partitions every time so that the list will take $\lceil \log_2 N \rceil$ passes to finish all partitions. However, if we always divide a list into two very unequal parts, one part with only one element, the number of passes Quicksort will take is N. If this kind of partitions persist throughout all passes, it is the worst case for Quicksort. The efficiency of this sorting method heavily relies on how we do our partitions. For a random list, since the elements are equally likely distributed, the possibility of the worst case happening is very unlikely no matter which element we choose as the pivot for partition. The easiest pick for the pivot is from either end of the list because we always know their array indices; but for already sorted or nearly sorted lists, which do happen in practice frequently, these partition methods end up with their bad or worst cases (calling itself for N times and only knocking off one element for each call). To cope with this disturbing feature of the last implementations, we can pick the middle element as its pivot and the previous worst or bad cases would turn around to be the best or good cases. However, concatenating two sorted sublists of equal or nearly equal lengths into a large list is likely to make this middle-pick implementation perform poorly. Accordingly, the undesirable features of Quicksort are that its worst case takes $O(N^2)$ operations, and it is tricky to program correctly, especially when one attempts to improve an implementation. The performance measure by comparisons-count for Quicksort is usually underestimated since it has a very short innermost loop and the comparison is done against a fixed value. But the overhead of the recursive function call involved seems to balance it out some if one doesn't make an effort to remove the recursion, which is usually done in practice. The probability of the worst case behavior occurring in Quicksort can be reduced significantly by applying some developed techniques, i.e. Median-of-Three (proposed by Singleton [14]), choosing the median of the three elements from the top, the bottom, and the middle of the list as the pivot element; the cost is to lengthen its code. Program X is this popular implementation of Quicksort using Median-of-Three as the pivot for partition, and ending each partition when the partition size is less than 10; then apply Insertion Sort to the whole list after Quicksort is done. The Insertion Sort is omitted here since it is the same as the one used for Combsort earlier. This Median-of-Three approach helps Quicksort in two ways: it reduces the worst case probability (more detailed discussions later) and makes the program run faster by about 5% [21]. The Insertion Sort help Quicksort save many function calls to small partitions and slightly reduces the number of comparisons if it is properly implemented. The reduction in the running time is about 20% [21].

```
procedure quicksort(lo,hi: integer);
var i,j,mid,v,t: integer;
label start;
begin
        if hi-lo>9 then
        begin
                mid:=lo+(hi-lo) div 2;
                if data[lo]>data[mid] then
                begin t:=data[lo]; data[lo]:=data[mid]; data[mid]:=t end;
                if data[lo]>data[hi] then
                begin t:=data[lo]; data[lo]:=data[hi]; data[hi]:=t end;
                if data[mid]>data[hi] then
                begin t:=data[mid]; data[mid]:=data[hi]; data[hi]:=t end;
                v:=data[mid];
                i:=lo; j:=hi;
                goto start;
                repeat
                        t:=data[i]; data[i]:=data[j]; data[j]:=t;
start:                  repeat i:=i+1  until data[i]>=v;
                        repeat j:=j-1  until v>=data[j]
                until i>j;
                quicksort(lo,j);
                quicksort(i,hi)
        end
end;
```

Figure 19. Program J Quicksort with Median-of-Three and Cutoff Point for the Insertion Sort at 10

As mentioned earlier, there have been quite a few research efforts put into the improvement of Quicksort. In the beginning, most of these efforts focused on how to make a "quicker" sort. Many ideas have been suggested and tried, but there is no convincing improvement except for the above implementation because the algorithm is so well balanced that the effect of speeding up one part can be more than offset by the side effect in another part of the program. For example, some people [17] tried to use mean instead of median from a small sample as the pivot for partition. Yet this depends heavily on the distribution of the data, which means the worst case (or bad cases) are much more likely to happen than the median method if it is not uniformly distributed. Of course, we can always take more items to gain better approximation of the median at the cost of some overhead regarding the selection of the median from a bigger sample and of lengthening the code. For example, median-of-five should be a better guess of the median than that of median-of-three, but the selection of the former takes 6 comparisons rather than 3, and these comparisons are done between two array elements rather than one against a fixed value. Thus further choosing more samples to estimate the median is probably not a good idea. After all, most people would probably not be willing to go for the lengthy program like median-of-five which gains little in average performance. After devoting some good amount of time in improving Quicksort, Robert Sedgewick [21] sighed, "It is tempting to try to improve Quicksort: a faster sorting algorithm is computer science's 'better mousetrap.' That is why later on, after realizing the difficulties in speeding it up, most of these research efforts have focused on how to reduce the probability of the worst case scenario happening. Again, none of these improvements seem to have won enough converts to affect the above implementation being taken to be an optimized version of Quicksort. Besides taking the median from a small sample, Hoare [10] suggested calling on a random number generator to get the pivot element for partition. That may be a safe choice to avoid the worst case in practical problems [21], though for a random list, this approach will neither reduce the worst case probability nor the number of comparisons over choosing any fixed partition element. Compared with the Median-of-Three approach which eliminates about 1/7 [14] of the number of comparisons, the random approach appears to be significantly slower. Therefore, it never gained any popularity, because few people would be willing to sacrifice such a loss in performance in order to avoid the very unlikely bad cases that they thought might never happen to them. However, the existence of the bad cases if not the worst case is still quite real for the Median-of-Three implementation. One good example would be cacatenating a list in ascending order to a descending one at about equal length (i.e. 10-8-

6-4-2-1-3-5-7-9). That is to say, the bad cases for the last implementation do happen in practice. In fact, it is possible to completely get rid of the worst case and derive a version of Quicksort with an O(N*lgN) complexity. Because theoretically, we can revise Quicksort algorithm to find a median in linear time (about (2+2ln2)N comparisons) [21], and use a true median (no guessing) for each one of the lgN partitions and derive a version of Quicksort with about (2+2ln2)NlgN comparisons. However, it is not practical to use this approach because of its extremely complex algorithm and this true-median Quicksort will not beat Heapsort or other O(NlgN) sorting methods. Therefore, I propose a small revision to the last partition scheme which I call "Median-of-Four". Here is my plan:

1. Take four elements from the beginning, the end, one third from the beginning, and one third from the end of the list.
2. After four compare-exchanges, we have the smallest one in the beginning and the largest one in the end of the list.
3. Start our partition-exchange process for Quicksort from the second one and the second last one.

The advantages of the above algorithm over Median-of-Three are:

1. The probability of the worst case is reduced in "most" (explained later) cases; 3 out of 4 rather than 2 out of 3 must be among the extreme values of the keys. A general term for calculating the worst case probability is $2*C_{N-k}^{m-k}/C_N^m$ where N is the size of population, m is the size of sample, and k is the number of elements required to be among the extreme values of keys. The leading term (constant 2) is for both the largest and smallest possible choices. For example, picking a key from a random list for use in partitioning (m=1, k=1), the worst case probability is $2*\ C_{N-1}^{1-1}/C_N^1$ = 2/N (the worst key could be either the smallest or the largest). Now for Median-of-Three (m=3, k=2) the worst case probability is $2*C_{N-2}^{3-2}/C_N^3$ = 2*(N-2) /(N*(N-

1)*(N-2)/(2*3)) = 12/(N*(N-1)). The worst case probability for our method Median-of-Four (if we agree in most cases 3 out of 4 must be among the extreme values of keys) evaluates to 2*C_{N-3}^{4-3}/C_N^4 =48/(N*(N-1)*(N-2)) which even beats that of Median-of-Five at 2*C_{N-3}^{5-3}/C_N^5=120/(N*(N-1)*(N-2)).

2. The worst case performs twice as fast; it will knock out two elements on each pass rather than one.

3. The average number of comparisons will be reduced by about 6% (see Table XXIII) for relatively large sizes because this approach yields a better estimate of the median. It is as though we apply median-of-five (four plus the average of the medians) at only one additional comparison rather than 3 more comparisons. Thus it will beat the complicated programming effort required for Median-of-Five.

For certain peculiar distributions of data, however, the above advantages except #2 may not be so obvious. One may even argue that the possibility of a worst case scenario is even twice as much as that of the Median-of-Three because 2 out of 4 being among the extreme values of keys might suffice to cause the occurrence of the worst case on certain distributions of data. This is a legitimate concern because 2*C_{N-2}^{4-2}/C_N^4 = 2*12/(N*(N-1)) =2* (2*C_{N-2}^{3-2}/C_N^3). Notice that the probability of two out of four elements being picked to be among the two largest keys equates the probability of two out of three being picked to be among the two extreme values (largest or smallest) of keys. However, the phrase 'among the two extreme values' is not precise. A more accurate argument should be: 2 out 4 must be among either the two extreme "large" values or the two extreme "small" values of keys. Adding the conjunctive verb 'either or' to the statement will cut down its probability some. For example, picking four elements from a list 1-2-10-11-20-21-30-31-40-41-99-100 will fall into its worst pick scenario for our Median-of-Four method as long as 99 and 100 are present in our four-element picks, but it is not necessarily true if our picks simply include 1 and 2. In fact, besides picking (1, 2, 99, 100), there are only two other picks (1, 2, 10, 11) and (1, 2, 11, 20) that will form the worst cases. Therefore, the worst case probability of Median-of-Four for this twelve-item list at this pass is only about 0.4% ($2/C^{4}_{12}$) more than that of Median-of-Three. Moreover, this slim chance must happen consistently at the four (not only three) sampling points throughout all passes. On the other hand, picking four elements from a list 1-2-52-53-61-62-71-72-81-82-99-100 will become its worst pick scenario if 1 and 2 are included in the picks, but it is not necessarily true if 99 and 100 are present in this four-element picks. It would be very difficult in practice to find a situation where the likelihood of the worst case of my approach should be almost twice as great as that of Median-of-Three.

One more thing we can try to improve the last implementation of Quicksort. Is Insertion Sort really the best of all for a very short list? Or is there room to improve Insertion Sort at all? There is no known method which will beat Insertion Sort for a very short list (say less than 10); Selection Sort comes closest to it. There are a couple of ways to improve Insertion Sort: List Insertion, Two-way Insertion, etc. Yet none of these improvements are suitable for a very small list size like the cutoff point (<20) for Quicksort. It is true that using a sentinel key will eliminate the boundary check for the innermost loop and speed things up some. Yet sometimes it might be difficult to choose a sentinel key. In the implementation of serving as a finish-up sorting method the sentinel key can be easily obtained by running one pass of the Exchange Sort (Bubblesort) for the cutoff size at one end of the list. For example, if the cutoff point is 10, we know the size of the largest possible partition is 10. Because the smallest element in the list must fall among the leftmost partition, we can be assured that by running one pass of Bubblesort from the 10th element to the 1st element, the smallest element must be in the first position of the array. Here I propose an easy way to set the sentinel and to help sort at the same time. We must imbed it in the Quicksort and let it run at the end of every partition. Since it will help sort, resulting in twice as large a cutoff point (20) at about the same number of comparisons. Program K demonstrates the unique way of setting the sentinel for the Insertion phase for Median-of-Four partition method as we proposed. See that this new partition method requires virtually one additional compare-exchange statement compared to Median-of-Three, yet it will harvest as much as 5% (Table XIV) performance gain and reduce the worst case likelihood to a negligible level. Also note that the test statement for the innermost loop of the Insertion Sort has been tuned so that "data[j-1]>v" can be changed to "data[j]>v" due to the larger cutoff point (see Program K). The reason for this is simple. Compared with the popular version of Insertion Sort, we replaced the statement "data[j]:=data[j-1];" with "data[j+1]:=data[j];" inside the inner loop in which they make no difference in running time. However, outside the inner loop, we replaced "j:=i;" and "data[j]:=v;" with "j:=i-1;" and "data[j+1]:=v;" where some overheads are involved. The loss in "j:=i-1;" is offset by the gain in the first "if data[j]>v". If one element does not go through the loop, then we will waste a little bit with this 'tuned-up' at the end of the loop ("data[j+1]:=v;" instead of "data[j]:=v;"). If an element goes through once and exits the loop, then we neither gain nor lose anything (savings in "data[j]>v " trade off with loss in "data[j+1]:=v;"). If an element goes through the loop more than once, then we will gain as many more passes through the loop as we save by replacing 'data[j-1]>v' with 'data[j]>v'. I think this small tune-up ought to be the way to program Insertion Sort. Except for very small size (say less than 10) or a nearly sorted list, we should always gain from this revision to Insertion Sort. Program XII is a partial code for Median-of-Five Quicksort implementation; compare with Program XI and see the complications it introduces.

```
{ Median-of-Four partition method - advocated by Su }
procedure quicksort(lo,hi: integer);
var i,j,v,t: integer;
label start;
begin
        if hi-lo>19 then
        begin
                i:=lo+(hi-lo) div 3;
                j:=i+(i-lo);
                if data[lo]>data[j] then
                begin t:=data[lo]; data[lo]:=data[j]; data[j]:=t end;
                if data[i]>data[hi] then
                begin t:=data[i]; data[i]:=data[hi]; data[hi]:=t end;
                if data[lo]>data[i] then
                begin t:=data[lo]; data[lo]:=data[i]; data[i]:=t end;
                if data[j]>data[hi] then
                begin t:=data[j]; data[j]:=data[hi]; data[hi]:=t end;
                v:=data[i]/2+(data[j]+1)/2;      { to avoid overflow }
                i:=lo; j:=hi;
                goto start;
                repeat
                        t:=data[i]; data[i]:=data[j]; data[j]:=t;
start:                  repeat i:=i+1  until data[i]>=v;
                        repeat j:=j-1  until v>=data[j]
                until i>j;
                quicksort(lo,j);
                quicksort(i,hi)
        end
        else begin              { folding compare starts }
                j:=hi;
                repeat
                        i:=lo;
                        repeat
                                if data[i]>data[j] then
                                begin
                                        v:=data[i]; data[i]:=data[j]; data[j]:=v
                                end;
                                i:=i+1; j:=j-1
                        until i>j
                until lo>j;
                for i:=lo+2 to hi do            { Insertion starts }
                begin
                        j:=i-1; v:=data[i];
                        while data[j]>v do   { small tuned up here }
                        begin
                                data[j+1]:=data[j];
```

```
                    j:=j-1
                end;
                data[j+1]:=v
            end
        end
end;
```

Figure 20. Program K - Quicksu (Median-of-Four Partition) with Folding Compare to Set Sentinel

```
{ Median-of-Five partition - about 3% slower than Prog. XI }
procedure quicksort(lo,hi: integer);
var i,j,v,t,m: integer;
label start;
begin
        if hi-lo>19 then
        begin
                i:=lo+(hi-lo) div 4;
                m=i+(i-lo);  j:=hi-(i-lo);
                if data[lo]>data[j] then
                begin t:=data[lo]; data[lo]:=data[j]; data[j]:=t end;
                if data[i]>data[hi] then
                begin t:=data[i]; data[i]:=data[hi]; data[hi]:=t end;
                if data[lo]>data[i] then
                begin t:=data[lo]; data[lo]:=data[i]; data[i]:=t end;
                if data[j]>data[hi] then
                begin t:=data[j]; data[j]:=data[hi]; data[hi]:=t end;
                if data[i]>data[m] then
                begin t:=data[i]; data[i]:=data[m]; data[m]=t; end
                else if data[m]>data[j] then
                begin t:=data[j]; data[j]:=data[m]; data[m]=t; end;
                if data[lo]>data[m] then
                begin t:=data[lo]; data[lo]:=data[m]; data[m]=t; end
                else if data[m]>data[hi] then
                begin t:=data[m]; data[m]:=data[hi]; data[hi]=t; end;
                v:=data[m];
                i:=lo; j:=hi;
                goto start;
        {same as Program XI from now on }
```

Figure 21. Program L - Median-of-Five partition (partial code)

Table XIV and Figure 22 are the results of our comparisons counts. The fastest version of Combsort only barely catches original Quicksort (taking the middle key as the pivot) at size 500 and will be out of competition when the size gets larger. The Median-of-Three with a cut-off point for the Insertion at 10 (Quick3_10) is a significant improvement over the original Quicksort and is on a par with the same partition method but using our Insertion strategy (Quick3_20). This shows that our Insertion Strategy does not yield a smaller comparisons count, but will save some overhead in function calls and perform slightly faster. Median-of-Five (Quick5_20) is poorer for small sizes because of its overhead but has a little edge over the above two only at the largest size we tested. The last one (Quicksu) seems to be the best. It takes the four sampling elements from the beginning, the end, one third from the beginning, and one third from the end of each list, making the bad cases unlikely to happen and reducing the number of comparisons by 2%, and this degree of improvement seems to go up as the size gets larger. We shall see this later. We will also have a benchmark test at the end of this chapter.

TABLE XIV

PERFORMANCE MEASURE FOR COMBINS AND VARIOUS VERSIONS OF QUICKSORT (NO. OF COMPARISONS)

Prog.\Size	500	1000	2000	3000
Combsu1.4	6192	14329	32787	53392
Quick	6151	13563	29990	47760
Quick3_10	4677	10780	24503	37128
Quick3_20	4623	10709	24476	37105
Quick5_20	4727	10701	23429	36696
Quick_su	4687	10530	23351	36528

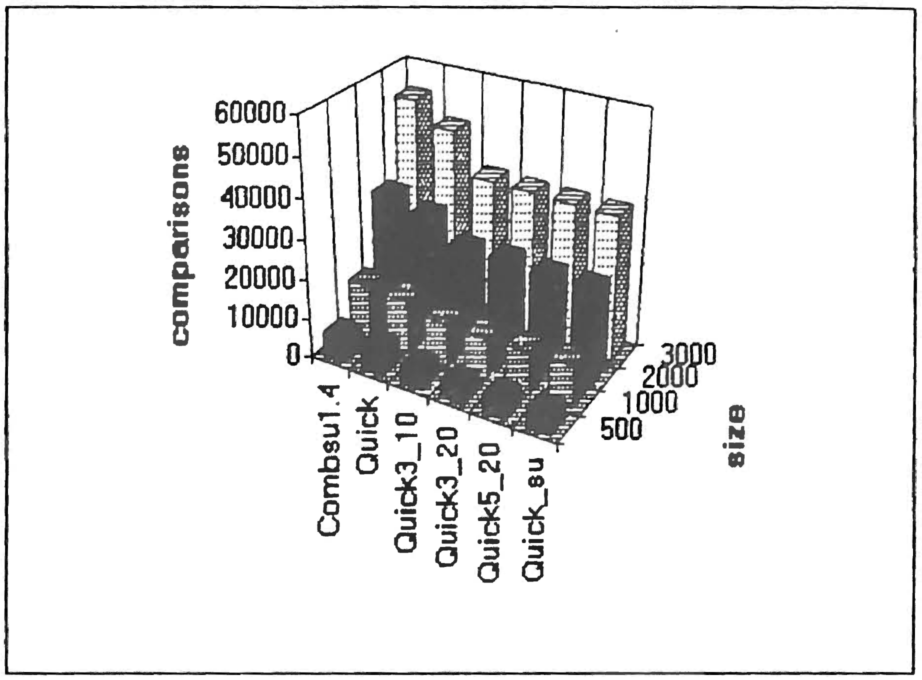

Figure 22. 3-D Bar Chart for Table XIV

The Elegance - Heapsort

Heapsort is another method which has N*lgN complexity like Quicksort and it is a guaranteed N*lgN in the worst case, not just on the average. Yet its innermost loop is quite a bit longer than that of Quicksort and this crude version of Heapsort is considered to be twice as slow as Quicksort on the average by many people [14] [21]. Note that it is desirable to use a bottom-up method to construct the heap (linear-time) because most of the heaps processed are small (about half of them are of size one and need no comparisons). The program code for this crude version of Heapsort is shown in the following:

```
{ Heapsort (crude version) sorts elements in the global array "data" with indices }
{ between lo and hi (both inclusive) using bottom-up method to construct the heap. }
procedure downheap(k,h_idx: integer);
label 0;
var i,j,v: integer;
begin
        v:=data[k];
        while k<=h_idx div 2 do
        begin
                j:=k+k;
                if j<h_idx then if data[j]<data[j+1] then  j:=j+1;
                if v>=data[j] then  goto 0;
                data[k]:=data[j];  k:=j
        end;
0:      data[k]:=v
end;

procedure heapsort(lo,hi: integer);
var k,t,h_idx: integer;

begin
        for k:=hi div 2 downto 1 do downheap(k,hi);
        h_idx:=hi;
        repeat
                t:=data[1]; data[1]:=data[h_idx]; data[h_idx]:=t;
                h_idx:=h_idx-1; downheap(1,h_idx)
        until h_idx<=1
end;
```

Figure 23.   Program M - Heapsort

Comparing the innermost loop of Heapsort with that of Quicksort, it is not hard to convince one that Heapsort actually runs twice as slow as Quicksort. However, as R. W. Floyd suggested, during the sorting heap phase most of the keys tend to be quite small, so it is possible to move one comparison out of the main loop and virtually cut the number of comparisons in half [14]. The following code (Program N) is an implementation taking advantage of that property with two major enhancements I found to speed it up. I present a full implementation here because the main procedure calls another procedure (compare2) in addition to the one that does the sorting, and it uses one extra space to achieve efficiency (we will explain this later). All other programs we have presented so far simply call its sorting procedure and do not need an extra space. The improvements from Heapsort to Heapsu (Figure 24 vs. Figure 23) are many:

1. Use "repeat-until" rather than a "while" loop and save 1.5N of 'j<h_idx' test statements in procedure downheap(), because we call this procedure N/2 times for constructing the heap and N times for sorting from the heap. We must make sure that the smallest size we will ever get is 4, so that its parent node data[j div 2] in the heap exists (j>=2, for we don't use data[1] for our sort-heap phase) throughout the entire sorting phase. That's why our loop control statement is "while h_idx>4" and we need a call to "compare2(2,3)" after the call to heapsort() to complete the sorting.
2. "if j<h_idx and data[j] <data[j+1]" is simplified to "if data[j]<data[j+1]" inside the loop. The reason is that as long as the last item is not the largest, it doesn't matter whether it is compared or not concerning sorting from a heap. A proof of this is included in Appendix A. Thus the call to compare2() prior to the one to heapsu() is to eliminate the possibility of the last item being the largest.
3. Use only one variable and save one statement: "j:=j*2;" (j<<=1; recommended in C) substitutes "j:=k+k;" and "k:=j;".
4. The use of a bit shift operation in C is more efficient than multiplying or dividing by two.
5. Utilizing a register variable j when possible (in C for example) gives a noticeable speed-up since j is referred to about $7*N*\log_2 N$ times within this loop.
6. The extra space saves about 2*N 'save' (t:=data[1] and data[h_idx]:=t) operations. Comparing the 'sort from the heap' loop in procedure heapsort() between Figure 24 and that of Figure 23, one will notice that we don't need the temporary variable 't' for swapping the first and last element in the heap because we just put the largest

element in the heap (data[1]) into the extra space (data[h_idx+1], h_idx=size initially) we reserved after the end of the heap construction, so only three rather than five statements are in the loop. What this extra space has achieved is that it enables us to replace a 'swap' operation with a 'move' operation. The sorted segment runs from 2 to N+1 in the array rather than 1 to N.

```
program heapsu;
{ A heapsort program implementation with R.W. Floyd's idea to move }
{ one compare test out of the inner loop enhanced with some of Su's ideas }

const
        max = 15000;

type
        list = array[1..max] of integer;

var
        data: list;
        i: integer;

{Heapsort sorts elements in the global array "data" with indices between }
{ lo and hi (both inclusive) using bottom-up method to construct the heap.}

procedure compare2(i,j: integer);
var v: integer;
begin
        if data[i]>data[j] then
        begin
                v:=data[i]; data[i]:=data[j]; data[j]:=v
        end
end;

procedure downheap(i,v,h_idx: integer);
var j: integer;
begin
        j:=i;
        repeat
                if data[j]<data[j+1] then  j:=j+1;
                data[j div 2]:=data[j];
                j:=j*2
        until j>=h_idx;
        j:=j div 2;
        if v>data[j] then
        begin
                j:=j div 2;
                while (v>data[j div 2]) and (j>=i) do
                begin
                        data[j]:=data[j div 2]; j:=j div 2
                end
        end;
        data[j]:=v
end;
```

```
procedure heapsort(lo,hi: integer);
var h_idx: integer;

begin
        { constructing the heap }
        for h_idx:=(hi-1) div 2 downto lo do downheap(h_idx*2,data[h_idx],hi);
        h_idx:=hi+1;  { extra space at top of the list }
        while h_idx>4 do    { sorting from the heap }
        begin
                data[h_idx]:=data[1]; h_idx:=h_idx-1;
                downheap(2,data[h_idx],h_idx-1)
        end;
        data[h_idx]:=data[1];  h_idx:=h_idx-1
end;

begin { Random is a user defined pseudorandom number generator }
        for i:=1 to max do data[i]:=Random(30000);
        if max>2 then
        begin
                compare2(1,max); heapsort(1,max); compare2(2,3)
        end
        else  { for size<=2 }
        begin
                if max=2 then  compare2(2,3)
                else data[2]:=data[1]  { size=1 }
        end;
end.
```

Figure 24.   Program N - Heapsu

Table XV shows that Heapsu should have gained about 40% in comparisons count. Comparing the result with Quicksort (Table XIV), it surpasses all versions of Quicksort in that category.

TABLE XV

PERFORMANCE MEASURE FOR THE TWO VERSIONS OF HEAPSORT (NO. OF COMPARISONS)

Prog.\Size	500	1000	2000	3000
Heap	7443	16843	37698	60234
Heapsu	4703	10427	22842	36105
% gained	36.81%	38.09%	39.41%	40.06%

Since we are comparing four different algorithms, we should do some benchmark tests to see what other overheads might have affected the performance of each algorithm. In practice, the running time is the most important factor for people to choose one method over another. But because it is platform-dependent (both hardware and software) and case sensitive, a seemingly irrelevant change in the code may affect the speed without the programmer's knowing it; thus it is dangerous to rely too much on the timing information to evaluate the efficiency of an algorithm. As an example, the programs I tested would run a little faster if the line "randomize();" is present in the programs than if it is not. The following benchmark tests are obtained from my Dell 386sx 16 Mhz PC running at 8 Mhz (to magnify the differences) using Borland C++. I have tried my best to optimize the performance for each individual algorithm. We have two sets of tests. The first one includes the winner of each algorithm using array indices. The second set uses pointers and includes the Quicksort with the middle element for the partition pivot, see Table XVI & XVII, Figures 25 & 26. The C programs for set 2 are included in Appendix B.

TABLE XVI

AVERAGE RUNNING TIMES (IN SECONDS)
FOR THE VARIOUS METHODS
(NO POINTERS)

Prog.\Size	1000	2000	5000	10000
Combsu	0.275	0.549	1.704	3.736
Shells	0.22	0.549	1.594	3.516
Heapsu	0.22	0.495	1.315	2.805
Quicksu	0.11	0.275	0.77	1.705

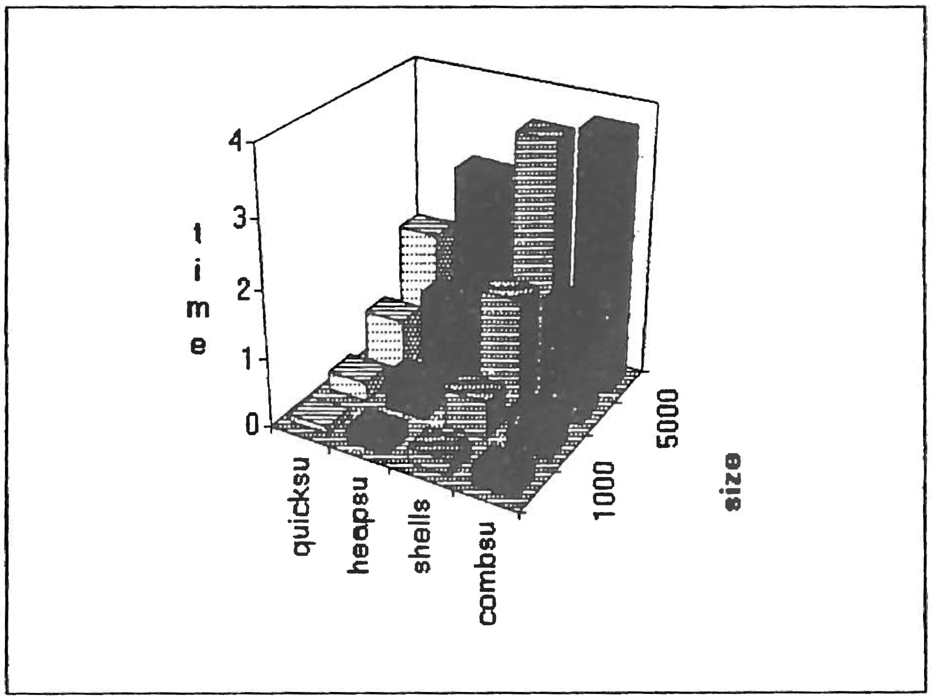

Figure 25. 3-D Chart for Table XVI

TABLE XVII

AVERAGE RUNNING TIMES (IN SECONDS)
FOR VARIOUS SORTING METHODS
(POINTERS ALLOWED)

Prog.\Size	1000	2000	5000	10000
Heapsu	0.22	0.495	1.315	2.805
Shells	0.165	0.44	1.155	2.585
Combsu	0.11	0.275	0.855	1.868
Quick	0.165	0.275	0.77	1.54
Quicksu	0.055	0.165	0.439	0.99

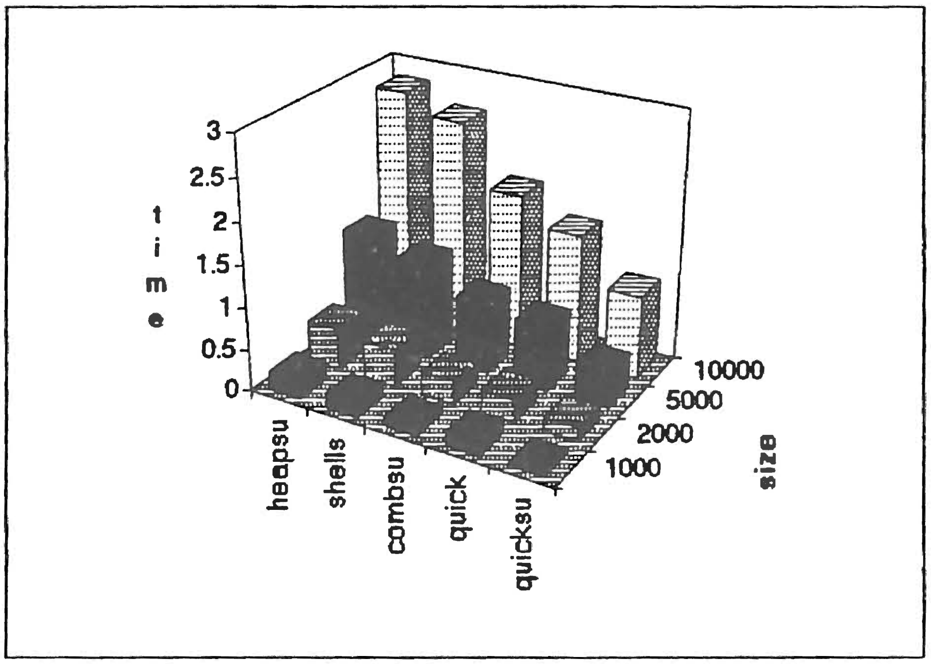

Figure 26. 3-D Bar Chart for Table XVII

The result from Table XVI and Figure 25 should be no surprise at all. The rankings for all of them remain intact from our comparison counts analysis; Quicksort seems to lead the pack even more but Heapsort seems to be caught up some by those behind it.

This difference only reflects what we have observed concerning the overhead involved in addition to the comparisons counts. However, when pointer references are used, Combsu2 passes Shells and Heapsu which becomes the loser, trailing the original Quicksort, but does not reduce the gap from Quicksu. It takes a moment of thinking to justify our observation. When the pointer, in C for example, is used instead of the array, it offers one advantage: direct references rather than indirect references. Do not confuse this with the terms used in assembly. What we mean by these two terms will be explained in the following. Consider that every time we use data[i] in C, we mean *(data+i). We call this an 'indirect' reference because we refer to an address required the addition of two terms. As an alternative, we can make a pointer variable j=data+i, then refer to *j. We call this a 'direct' reference because we need to refer to only one term. Accordingly, both Quicksort and Combsort take full advantage of this and run a lot faster than the non-pointer versions. Shellsort takes some advantage of it and runs a little faster than the non-pointer versions. In contrast, Heapsort cannot take advantage of this at all since it gains no speed for these pointer-versions over the array versions. Clearly, Heapsort requires the array indices to be calculated in multiplication or division, but we cannot do such operations in pointers. It also requires the index to start from 1, but we cannot make a pointer point to 1, either. If we insist on using pointers for Heapsort, we still need to add an extra integer index to the pointer and manipulate the index all the time. We will explain this pointer operation in more details in Chapter VI.

CHAPTER IV

SOME PROPERTIES OF COMBSORT AND ITS PERFORMANCE ESTIMATION

Before we get into this chapter, let's define some terms which will be used later.

1. k-ordered: If we extract all elements at distances of multiples of k in a list, we will form a sorted list, no matter which element we start with. A list of that kind is called k-ordered list.

2. Combsort and Shellsort (transitive verbs): when we use these two terms, followed by 'at gap=k', we mean that we sort a list according to what Combsort or Shellsort will do in a particular pass at gap=k (1<=k<N).

Combsort is so similar to Shellsort that we should mention the differences between the two. Property 4.1 & Corollary 4.1a are some examples.

Property 4.1: If a list is Combsorted at gap k, it may not be k-ordered throughout the list.

Proof: If a list with five elements 8-6-3-1-2 is Combsorted at gap 2, it becomes 3-1-2-6-8. 3 and 2 are still out of order at gap 2.

This is different from Shellsort because after a list is Shellsorted at gap k, it is a k-ordered list.

Corollary 4.1a: If a list is Combsorted at gap k, and then it is Combsorted at gap h, still it may not be k ordered.

Proof. A list with six elements 3-5-6-2-4-1 is Combsorted at gap 2, then it is Combsorted at gap 1. The resulting list is 2-3-1-4-5-6. 2 and 1 are not ordered at gap 2.

This corollary does not hold for Shellsorting. A well-known property of Shellsort is: if a list is k-ordered, and then is h-sorted, it remains k-ordered. The above properties can make one tell Combsort from Shellsort. The next two properties of Combsort will give us some clues why it might be a good alternative to Shellsort.

Property 4.2: If a list is k-ordered, then is Combsorted at gap h, it remains k-ordered.

Proof: Surprisingly, the proof of this is easier than the one found in Shellsort, though it is lengthy. See Appendix A.

Let's visualize this property by an example. If we let a 5-ordered list 3-7-9-4-1-5-8-10-6-2 be Combsorted at gap 1-9 respectively. The resulting lists all remain 5-ordered and are shown in the following:

start:	3	7	9	4	1	5	8	10	6	2
gap=1:	3	7	4	1	5	8	9	6	2	10
gap=2:	3	4	1	5	8	7	6	2	9	10
gap=3:	3	1	5	4	7	6	2	10	9	8
gap=4:	1	5	8	4	3	2	9	10	6	7
gap=5:	3	7	9	4	1	5	8	10	6	2
gap=6:	3	7	6	2	1	5	8	10	9	4
gap=7:	3	6	2	4	1	5	8	10	7	9
gap=8:	3	2	9	4	1	5	8	10	6	7
gap=9	2	7	9	4	1	5	8	10	6	3

This property of Combsort indicates that Combsort will take advantage of the sortedness of a list, like Shellsort. In 1984, Hong-lee Yu [30] experimented with a new variation of Shellsort which exactly is Combsort with a Shrink Factor of 2.0 and using Hibbard's sequence with a cutoff point at 1 for Insertion Sort in our terms. The result was a major disappointment; it fell short of all versions of Shellsort he compared even when the size of list was about a moderate 100 or more. However, he noticed the following property.

Property 4.3: Every pass of Combsort where gap>1 takes only O(N) comparisons.

Proof: Since we start from 1 to N-gap, each with a single comparison, we will end up with exactly 'N-gap' number of comparisons for the pass. See also Property 4.7.

If we can expect that the degree of sortedness is so high before gap=1 that the Insertion phase will only take O(N) comparisons, then we will have a version of Combsort with the complexity of O(N*lgN) which will beat any version of Shellsort which requires $O(N^{y/x})$ or $O(N*\log^2 N)$ asymptotically. However, a Shrink Factor of 2.0 is much too large according to our test results.

Property 4.4: For a list of size N, if we Combsort the list by decrementing the gap from N-1 to 1, the list will be sorted.

Proof: Pratt [19] found a unique way of doing Shellsorting where the innermost loop is replaced by a single comparison which is similar to that of Combsort. If by taking all increments $h=2^p*3^q$ as the gap sequence for Pratt's Shellsort to work, and from Property 4.2 we know that adding more Combsort passes will not mess up the k-ordered characteristics. Because Pratt's sequence is a subset of {N-1, N-2, ... 1}, the list will be sorted using more passes in addition to Pratt's sequence. The number of comparisons of this version of Combsort is N*(N-1)/2 like Bubblesort with many fewer transpositions. The program code is one of the simplest in sorting.

Property 4.5: When a list is Combsorted at gap k in ascending (descending) order, the largest (smallest) element can be found among the last (first) k elements in the list.

Proof: If X_i is the largest element in the list and the list is Combsorted at gap k, it will go as far as it could until i+k>size. Now if X_i is not among the last k elements, then i+k<=size which is a contradiction. Reversing the direction of sorting follows the same argument accordingly. We have used this property and the next Corollary to substitute the sentinel for Insertion Sort in Chapter III. As an example, if we Combsort a list 5-3-6-2-4-1 at gap=2, the resulting list is 5-2-4-1-6-3. The largest element 6 can be found among the last 2 elements (6 and 3).

Corollary 4.5a: For the Two-way Combsort with a Shrink Factor less than 2, except for the first pass, any other pass Combsorts the list at gap k from one end, and the largest and the smallest elements are among the last k and the first k elements respectively.

Proof: If we Combsort a list from one end at gap k, we know that k elements on the other end will have one of the extreme elements, from Property 4.5. If the gap size for the last pass is h, where $k<h<2*k$, we know the other extreme element on this end was among the h elements of this end before we started this pass. Since $h<2*k$, we know after the first k comparisons we shall cover all h elements and bring the extreme one to the first h elements. Following the previous example, if we continue to Combsort that list at gap=1 from right to left, the resulting list is 1-5-2-4-3-6. This is agreeable to our argument.

Perhaps the most significant property is Property 4.6 which could be a stepping stone to the fitted asymptote for Combsort. Let's be optimistic for a moment! What is the best case for Combsort? Clearly, before we get into gap 1, if the list is sorted already, then the last pass is just to go through the list and make sure the list is sorted. That is our best case scenario which will be the stepping stone for our average behavior analysis.

Property 4.6: The best case in comparisons for Combsort is about $N*\log_s N-(N-1)/(s-1)$, where N is the size of the list and s is the Shrink Factor.

Proof: Let J be the number of passes for Combsort and j be the ordinal numbers of J, $1<=j<=J$, and h_j be the gap size at pass j, then $J=\lfloor \log_s N \rfloor$, $h_j \cong N/s^j$ and the number of comparisons for pass j is $N-h_j=N*(1-1/s^j)$. The total number of comparisons for all passes is $\sum_{j=1}^{J} N*(1-1/s^j) = N*(\sum_{j=1}^{J} 1 - \sum_{j=1}^{J} 1/s^j) = N*(J - \sum_{j=1}^{J} (1/s * 1/s^{j-1}))$

$= N*(J-1/s*(1-1/s^J)/(1-1/s) = N*(J-1/s*(1-1/s^J)/(s-1)/s)$

$= N*(J-1/(s-1)+(1/s^J)/(s-1)) \cong N*\log_s N-(N-1)/(s-1).$

(because $J=\lfloor \log_s N \rfloor$, $s^{\log sN}=N$)

For example, if we let s=2 (too big, we know), the result can be simplified to $N*\log_2 N-N+1=N*(\log_2 N-1)+1$. If N=2048, the result is 20481.

Property 4.7: The worst case for Combsort/Combsort11 is $O(N^2)$.

Proof: See next Chapter.

As we have stated earlier, the samples we tested were too small to represent their complexities. Now let's take a glimpse at larger sizes. In Table XVIII, we can see that the Shrink Factor 1.4 is no longer optimal for Combsu; 1.32 is optimal instead. Yet the optimal Shrink Factor remains about the same for the two-way version - 1.43 vs. 1.44. That means our propaganda for the Two-way Combsort is valid. In fact, the 1.44 version still ran a wee bit faster than the 1.43 version due to the passes it saved, according to some timing tests on the machine I used.

TABLE XVIII

PERFORMANCE MEASURE FOR THE VARIOUS VERSIONS OF COMBSORT WE TESTED TOWARD VERY LARGE SIZES (NO. OF COMPARISONS)

Prog.\Size	150	1500	15000	150000	1500000	15000000
comb11	2071	34052	505069	6400088	77500086	9.25E+08
combins	1519	27093	401264	5466929	66461371	7.97E+08
combsu1.4	1364	23812	351036	5250646	1.31E+08	3.18E+09
combsu1.32	1332	25387	379996	5009725	62619923	7.6E+08
combsu21.43	1375	23501	330424	4306174	53984809	6.30E+08

Table XVIII also reveals that the number of comparisons for Combsu has decreased about 6% from original Combsort with a cutoff point at 6 for the Insertion Sort (combins). This 6% is about lg1.32/lg1.3. That is to say, we do not add more work to the last pass of Combsu than to the last pass of Combins. Nevertheless, lg1.43/lg1.32 represents about 29% increase where Combsu2 only improves about 14% from Combsu at the last three sizes. This shows us that we do put more work to the last pass of Combsu2 than to the last pass of Combsu; but the savings from the fewer passes (O(N) for each pass) of the bigger Shrink Factor (1.43 vs. 1.32) outweighs the extra work that the last pass of Combsu2 requires. Anyway, all these versions except Combsu at 1.4 seemed to perform consistently well over various data sizes we tested. We could try to do the complexity studies on one and get the pictures for all. From Property 4.6 we can get a pretty accurate estimate on the number of comparisons for any version of Combsort before the Insertion Phase if the cutoff point for the Insertion is 0. Now the cutoff point for Combsu2 is 3, so we save about 3N from the best case; and the Shrink Factor is 1.43, so we will have $\lceil \log_{1.43}N \rceil \cong 1.94*N*\log_2 N$ passes, each pass requires 'N-gap-1' number of comparisons. The gaps through all passes being a geometric series at a factor of 1.43 from N can be summed up to '(N-1)/(1.43-1)'. Therefore, we can be assured that its complexity before the Insertion is about $1.94*N*\log_2 N-(N-1)/(1.43-1)-3*N$. If the Insertion phase will take O(N) complexity, we can get a complexity at '$1.94*N*\log_2 N+O(N)$' which should be a good starting point for the average-case complexity analysis. We will leave a more detailed analysis for future works. Table XIX is the performance in comparisons count for the previous version of Combsu2. Another version of Combsu2 which will be discussed later is also included.

TABLE XIX

PERFORMANCE MEASURE FOR THE TWO VERSIONS OF TWO-WAY COMBSORT (NO. OF COMPARISONS)

Size	Combsu2 SF=1.43	Combsu2d SF=1.45
150	1375	1374
300	3283	3335
450	5477	5449
600	7620	7809
750	10127	10088
900	12630	12654
1500	23501	23132
3000	52505	52233
4500	84225	83246
6000	116966	116205
7500	150549	151301
9000	185885	184472
15000	330424	332305
30000	720628	713915
45000	1139503	1136537
60000	1561337	1547692
75000	2013173	1992330
90000	2448843	2421730
150000	4306174	4281711
300000	9256621	9167972
450000	14376770	14248039
600000	19684930	19368330
750000	24889850	24739313
900000	30473709	30295575
1500000	53984809	52477143
4000000	152743968	151617368
15000000	629665022	623785820

Figure 27 is the trend for the Two-way Combsort using the least squares method to calculate a straight line that best fits our data when plotted on a log-log graph. The fit formula can be derived as $5.86 * N^{1.12964}$; but its goodness of fit is not convincing.

Table XX includes the last six points being projected and their residuals (actual minus fit) as well as their relative residuals (in percentage).

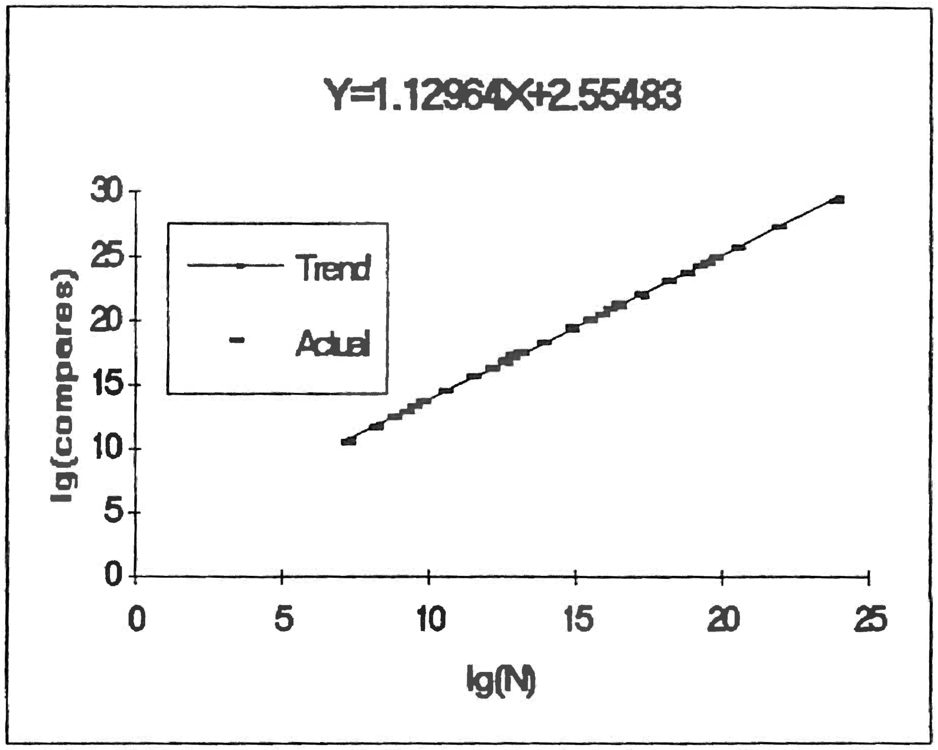

Figure 27. Exponential Fit for Combsu2 (Two-way Combsort)

TABLE XX

PARTIAL FIT RESULTS FOR FIGURE 27
(NO. OF COMPARISONS)

Size	600000	750000	900000	1500000	4000000	15000000
Actual	19684930	24889850	30473709	53984809	1.53E+08	6.3E+08
Fit	19784002	25455851	31277638	55698472	1.69E+08	7.51E+08
Residual	-99071.757	-566001	-803930	-1713663	-1.6E+07	-1.2E+08
Rel. Res.	-1	-2	-3	-3	-9	-16

Now let's do the same fit at N*lgN. The result is a much better fit. See Figure 28 and Table XXI. This fit formula is calculated to 1.993423*Nlg(N)-5.49875*N.

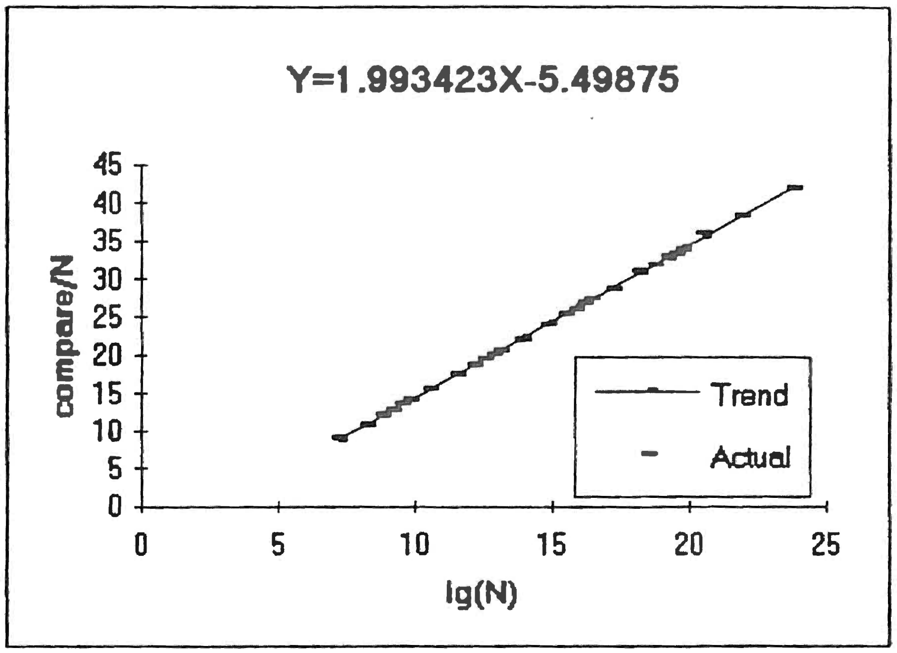

Figure 28. N*lg(N) Fit for Combsu2 (Two-way Combsort)

TABLE XXI

PARTIAL FIT RESULTS FOR FIGURE 28
(NO. OF COMPARISONS)

Size	600000	750000	900000	1500000	4000000	15000000
Actual	19684930	24889850	30473709	53984809	1.53E+08	6.3E+08
Fit	19658526	25054461	30537259	53099057	1.53E+08	6.3E+08
Residual	26404	-164612	-63550	885752	-136589	-655654
Rel. Res.(%)	0	-1	0	2	0	0

It is amazing to see that a simple method like Combsort, after some thoughtful refinements, should outperform a well-developed method like Shellsort. Although Sedgewick's method is a close competitor, its strange sequence makes it more complicated to program and less appealing. Nevertheless, before we claim victories for Combsort, let's stand behind Shellsort and learn from our opponent – Combsort. Besides the innermost loop, there is one subtle difference between the two: Shellsort takes a unique path for its diminishng sequence while Combsort utilizes multiple sequences according to the size of data and a Shrink Factor. Why not try some Shrink Factors for Shellsort and see how it is doing? To my surprise, it was so easy to beat all known Shellsort's sequences at some noticeable margin except Sedgewick's by a Shrink Factor ranging from 1.7 to 2.3 excluding those being very close to 2.0 which exactly was what the original Shellsort used as a Shrink Factor. "Why such a simple idea hasn't been discovered for such a long time (although later on, I realized Gonnet [8] had suggested a Shrink Factor of 2.2 for Shellsort not long ago)," I pondered. Then I reasoned, "because they were too preoccupied by the original version of Shellsort, they simply tried to avoid the redundant comparisons by introducing the relative primeness to its sequence; and the easiest way to achieve this goal is simply miss one to the common factor 2 (Hibbard's) or 3 (Knuth's)." Actually, while maintaining the properties of being relatively prime, we can miss a little more (say 10%, 20%, etc.) to the common factor of 2. This turns out to be a better way of making the gap sequence for Shellsort. My empirical result shows that 2.2 indeed is not bad except it has some bad cases which could be 15% slower than some other Shrink Factors while in other cases it performs quite well. One example of the bad cases for 2.2 is: 200-90-40-18-8-3-1 as its gap sequence where 40, 18, and 8 have 200, 90, and 40 in their ways 5 gap-length away. In other words, it violates the golden rule of Shellsort where relative primeness is the most important factor to observe. Accordingly, some redundant comparisons are inevitable in such a case. In fact, I found that any Shrink Factor taking the form of $(2*n+1)/n$, $n \in \{2,3,4,5\}$ or $(2*n-1)/n$, $n \in \{4,5,...9\}$, was excellent in its good cases. The first group seems to have fewer comparisons but more transpositions, and the overall performance is just a little better than those of the second group. In particular, both n=2 (SF=2.5) or n=4 (SF=2.25) hardly had any bad cases. The former does the best job at sizes<200,000 but gradually yield to the performance of n=3, n=4, or n=5. The latter seems to be an overall champion especially toward very large lists. In fact, Sedgewick's sophisticated sequence is in line with our approach by alternating between about 2.25 and 1.78 except for very small gaps. No wonder it is effective! After numerous tests, I concluded that a combination of n=2 and n=4 would be the best policy. Furthermore, the sequences ending up with 4-1 perform just a little better than those of 3-1, 2-1, or 5-1 (resetting 2 to 1). Therefore, the unique sequence approach which Hibbard first suggested followed by Knuth and Sedgewick to improve the original Shellsort was actually not a bad idea at all. I propose the following Shellsort program (Shellsu) which is the best among all I have tested. And coding this program is easy - see Program O. A comparison with Sedgewick's sequence (Shells) and Gonnet's SF 2.2 (Shellg) is also included in Table XXII.

```
{ a Shellsort sequence using 2.5 & 2.25 as its Shrink Factors }
{ 1,4,11,28,71,178,401,903,2032,4573,10290,23153,52095,.... }
procedure shellsu(lo,hi: integer);
label out;
var i,j,gap,v: integer;
begin
        gap:=178;  { if gap<=178, SF=2.5 }
        i:=(hi-lo) div 4;
        while (i>gap) do gap:=gap*2+gap div 4+1;
        repeat
                for i:=gap+1 to hi do
                begin
                        v:=data[i]; j:=i;
                        while data[j-gap]>v do
                        begin
                                data[j]:=data[j-gap]; j:=j-gap;
                                if j<=gap then goto out
                        end;
out:                    data[j]:=v
                end;
                if (gap>200) then gap:=gap*4 div 9 { no floating point cal. }
                else gap:=gap*2 div 5;
        until gap=0
end;
```

Figure 29. Program O (Shellsu)

TABLE XXII

PERFORMANCE MEASURE FOR VARIOUS VERSIONS OF SHELLSORT WE TESTED (NO. OF COMPARISONS)

Data Size	Shells	Shellg	Shellsu
150	1304	1385	1298
300	3373	3368	3104
450	5329	5400	5091
600	7454	7396	7219
750	9668	9906	9625
900	12065	12296	11955
1500	22176	22507	21777
3000	50721	50390	49131
4500	80960	79089	78744
6000	112028	110623	109685
7500	146271	147569	141613
9000	177369	181865	175369
15000	319931	320084	310993
30000	700975	797956	680821
45000	1096030	1110792	1074445
60000	1511528	1518825	1476409
75000	1932948	1977032	1889952
90000	2365586	2441920	2312121
150000	4153947	4123815	4070271
300000	8920114	11546618	8721866
450000	13907834	13989369	13577030
600000	18963992	18788515	18618873
750000	24172838	23824145	23718173
900000	29516989	29204217	28901484
1500000	51194031	51430313	50363190
4000000	148316266	146104807	145379527
15000000	613729850	612867414	601180634

"Iron sharpens iron, so one man sharpens another." [Proverbs 27:17] Since we let Shellsort learn from Combsort by applying a Shrink Factor type of gap sequence, we may also improve Combsort by utilizing the unique path gap sequences as people do in Shellsort. After many days of trying, I found the way we did to avoid the common factor of 2 by decrementing each even gap by 1 is no longer necessary since we are looking into a unique path with very few INPs in it. The best Shrink Factor is 1.45 for this version of Combsort.

```
{a sequence of 4-5-7-11-17-26-38-56-82-119-173-251-364...}
procedure combins2(lo,hi: integer);
label: loop, out;
var i,j,gap,v: integer;

begin
        i:=(hi-lo) div 3;
        gap:=26;
        while i>gap do gap:=trunc(gap*1.45)+1;
loop:   i:=lo;
        for j=lo+gap to hi do
        begin
                if data[i]>data[j] then
                begin
                        v:=data[i]; data[i]:=data[j]; data[j]:=v
                end
                i:=i+1
        end;
        gap:=trunc(gap/1.45);
        if gap<5 then
        begin
                gap:=gap+1; { 4->5, 3->4, 2->3 }
                if gap=3 then goto out
        end;
        j:=hi;
        for i:=hi-gap downto lo do
        begin
                if data[i]>data[j] then
                begin
                        v:=data[i]; data[i]:=data[j]; data[j]:=v
                end;
                j:=j-1
        end;
        gap:=trunc(gap/1.45);
        if gap<5 then
        begin
```

```
                gap:=gap+1; { 4->5, 3->4, 2->3 }
                if gap=3 then goto out
        end;
        goto loop;
out:    insertion(lo,hi)
end;
```

Figure 30. Program P (Combsu2ins w/ Unique Path)

The savings in comparisons count from this version of Combsort (Combsu2d) over the previous version (Combsu2) is a minor one - see Table XIX. Adding the overhead of setting the unique path makes this improvement negligible.

Table XXIII is a benchmark test for the Two-way Combsort, Shellsu, Shells, Qui_3_10 (Quicksort with Median-of-the-three and cutoff at 10 for Insertion Sort), Quicksu, Heap, and Heapsu. Note that Heapsu runs faster than Quick_3_10 at smaller sizes but is outperformed at larger sizes. That is because of the trick we did to Heapsort by adding one extra space to save 2N 'saves'. That effect remains a constant factor to N while the number of moves grow in proportion to N*lg(N).

TABLE XXIII

AVERAGE RUNNING TIMES FROM THE BEST VERSIONS
OF VARIOUS SORTING METHODS WE TESTED
(NO POINTER MANIPULATIONS)
(IN SECONDS)

Prog./Size	15000	150000	1500000	15000000
combsu2	1.1	16.7	212.2	2567.0
shellsu	1.0	14.9	190.1	2356.5
shells	1.3	18.4	233.8	2865.6
quick_3_10	0.9	13.8	166.1	1606.2
quicksu	0.8	10.6	151.9	1521.5
heap	1.6	21.0	265.7	3192.2
heapsu	0.9	13.3	168.7	2078.3

All these timing values are slightly overestimated because they all include the overheads of keeping counters for comparisons and transpositions. Moreover, they didn't make use of pointers to speed up the programs - for Quicksort and Combsort especially. If they did, Combsu2d would have caught Heapsu and become the runner-up of this pack since Heapsort will not take advantage of pointer manipulations as we have discussed in Chapter III - see Table XXIV.

TABLE XXIV

AVERAGE RUNNING TIMES FROM THE BEST VERSIONS
OF VARIOUS SORTING METHODS WE TESTED
(POINTER MANIPULATIONS)
(IN SECONDS)

Prog./Size	15000	150000	1500000	15000000
Combsu2	0.8	12.4	163.9	1916.7
Shellsu	0.9	13.7	175.7	2158.6
Quicksu	0.6	8.3	98.4	1143.8
Heapsu	0.9	13.3	168.7	2078.3

Since we have improved all four Sorting Algorithms significantly, let's also take a look at what we have reaped. Table XXV records the comparison counts and their relative positions in proportion to N*lg(N).

TABLE XXV

AN OVERVIEW OF OUR IMPROVEMENTS
THE ORIGINAL AND THE IMPROVED
ALSO N*LG(N) DIAGNOSTICS
(NO. OF COMPARISONS)
(NO. OF N*LG(N))

Size	150	1500	15000	150000	1500000	15000000
NlgN	1084	15826	208090	2579190	30774797	357576887
Heap	1708	27146	370272	4701098	57019438	669417154
comp./NlgN	1.58	1.72	1.78	1.82	1.85	1.87
Heapsu	1147	16555	215180	2651230	31513434	366972662
comp./NlgN	1.06	1.05	1.03	1.03	1.02	1.03
Quick	1633	23161	285474	3586822	41573618	483622739
comp./NlgN	1.51	1.46	1.37	1.39	1.35	1.35
Qui3_10	1184	17856	231272	2939692	35034723	411848883
comp./NlgN	1.09	1.13	1.11	1.14	1.14	1.15
Quicksu	1097	17215	227530	2766166	33325584	389345426
comp./NlgN	1.01	1.09	1.09	1.07	1.08	1.09
Shellsu	1298	21777	310993	4070271	50363190	601180634
comp./NlgN	1.20	1.38	1.49	1.58	1.64	1.68
Comb11	2071	34052	505069	6400088	77500086	925000111
comp./NlgN	1.91	2.15	2.43	2.48	2.52	2.59
Comb2d	1374	23132	332305	4281711	52477143	623785820
comp./NlgN	1.27	1.46	1.60	1.66	1.71	1.74

CHAPTER V

SCOPE OF IMPLEMENTATIONS

While I was rejoicing at the big improvement for Combsort, I also marveled at the huge chunk of code that I have added to it. I asked myself, "Who in the world is going to use this monster!" Compared to the fastest version of Shellsort I developed, I don't see why one would rather use the messy code like Two-way Combsort with Insertion as its finishing up partner. Therefore, as far as the scope of implementations is concerned, I would have said, "The real strength of this sorting algorithm is its balance between ease of programming and its level of performance." had I had not found a nearly worst case scenario taking a form of quadratic function (see Table XXVI) like those elementary sorting methods. We know the worst case for most versions of Shellsort is much better than $O(N^2)$.

TABLE XXVI

PERFORMANCE MEASURE IN NEARLY WORST CASE FOR COMBSORT11 (NO. OF COMPARISONS)

Data size	100	200	400	800	1600
Comparisons	2779	13514	62168	284235	1304321

Table XXVI is the result of Combsort11 for sorting a trouble-making list. Here is how I construct a trouble maker for Combsort. Combsort only fixes the inversions in the data one gap at a time, and each element has only one chance to jump backwards on each pass. Thus there are not many passes (about $2.5*\log_2 N$) before the gap becomes 1. Assume that we are sorting a list into ascending order. Now a relatively small item situated in a position near the end of the list. Coincidentally, for the first few passes it does not have a chance to move, which means all the elements in front of it with a multiple of gaps in distance from it are smaller than this element. At a later stage of sorting, because the gaps are relatively small, although it will probably be carried backward one gap on each following pass, it still is some distance away from its sorted position before gap=1. When the gap becomes 1, it will take as many passes as the number of inversions this element possesses. For example, with a list of 100 integers (1-100) we could construct a bad case for Combsort by putting 16 in position 100. Since Combsort uses a gap sequence of 76-58-44-33-25-19-14-11-8-6-4-3-2-1-1..., we could fill up the positions in 24 (gap=76), 42 (gap=58), 56, 12 (gap=44), 67, 34, 1 (gap=33), 75, 50, 25 (gap=25) with numbers smaller than 16. Then for each position we have filled up with these small numbers, we need to consider combinations of those gaps (76,58,44,33,and 25) which may interfere with our plan to avoid having 16 swapped with any of them. Therefore, we also need to fill up positions 31 (44+25), 23 (44+33), 17 (58+25), 9 (58+33), and 6 (44+25+25) with numbers smaller than 16. Therefore, after we fill up those 15 positions with integers of 1 to 15 in any order, we could fill up the rest of the unoccupied positions(84 in total) with 17 to 100 in any order. If we try to sort this list with Combsort, the last item 16 will not change its position in the first five passes. From then on until gap becomes 1 it will move backward every time for a distance of each specific gap in each pass (19,14,11,8,6,4,3,2). It then will reach position 33 which is 17 positions away from its final destiny and will require 17 additional passes to get there. For a random list of this size, the average number of passes for gap=1 is two. We call this a "black sheep effect" - one bad member makes the whole group suffer. The C program code for constructing such a hostile list is in Appendix C.

To prevent such a "black sheep" effect, one can use the Insertion Sort or use our Two-way Combsort. There is no need for both to exist just for preventing that one "black sheep." It is preferable to use the "Two-way" method to the addition of Insertion Sort because the former does not add any more complication to the code, simply repetitions of code. Not only is it easier to program, it will perform faster. In fact, it is going to perform just about 8% slower than Combsu2d (having the Insertion Sort) after one small refinement (otherwise, the margin will be about 15%) by introducing a boundary check to save many unnecessary comparisons when gap becomes 1 - see Program Q.

```
procedure combsu2ni(lo,hi: integer);
label: loop, out;
var i,j,idx,gap,v: integer; { idx is for boundary check }

begin
        i:=(hi-lo+2) div 3;  { initial gap }
loop:   i:=lo;
        for j=lo+gap to hi do
        begin
                if data[i]>data[j] then
                begin
                        v:=data[i]; data[i]:=data[j]; data[j]:=v; idx:=i
                end
                i:=i+1
        end;
        if gap>10 then  gap:=trunc(gap/1.42) { for most passes }
        else if gap>1 then gap:=gap-1 { change the course of SF }
        else if lo<idx then
        begin  hi:=idx; idx:=hi  end { new high index }
        else goto out  { break out of the loop }
        j:=hi;
        for i:=hi-gap downto lo do
        begin
                if data[i]>data[j] then
                begin
                        v:=data[i]; data[i]:=data[j]; data[j]:=v; idx=j
                end;
                j:=j-1
        end;
        if gap>10 then  gap:=trunc(gap/1.42)
        else if gap>1 then gap:=gap-1
        else if hi>idx then
        begin  lo:=idx; idx:=lo  end { new low index }
        else goto out
        goto loop;
out: end;
```

Figure 31. Program Q (Combsu2ni)

Notice that the optimal Shrink Factor for the above Program is 1.42. Interestingly enough, after examining the fit result from my test data (average of 10 sets), it represents a flatter curve than that of any other versions of Combsort or Shellsort in relation to the one term "N*lg(N);" it is about 2.0*N*lg(N). See the following two charts and compare with Table XXV. Here we used the least squares method again but only fit the second half of the actual data. A comparison table (average and standard deviation) with Shellsu is included in Appendix D.

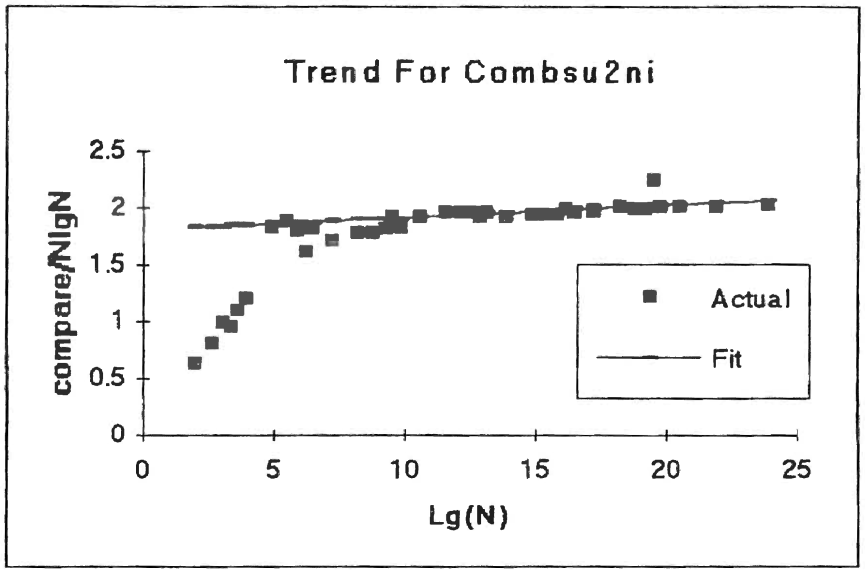

Figure 32. N*lg(N) Diagnostic Fit Chart for Combsu2ni

One may still concern the fact that the data are rising a little at large N. Perhaps the complexity is worse than NlgN. Let's use the same method of plotting but with Nlg^2N diagnostics. The result is not very good at all, see Figure 34. Finally, we compare the trend of Combsu2ni with those of Quicksort and Heapsort (with known NlgN complexity). We also let Shellsu participate in this NlgN probing chart. In contrast, our result is acceptable, see Table XXVII and Figure 35.

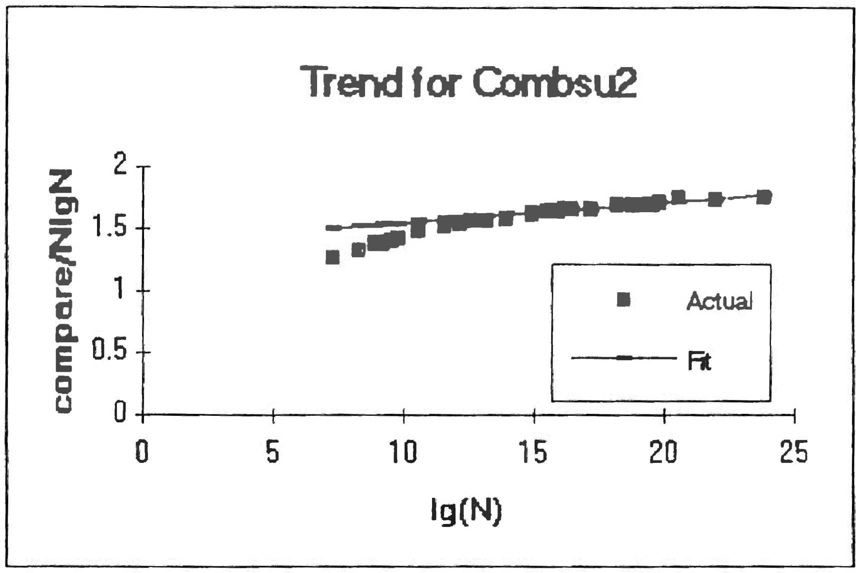

Figure 33. N*lg(N) Diagnostic Fit Chart for Combsu2

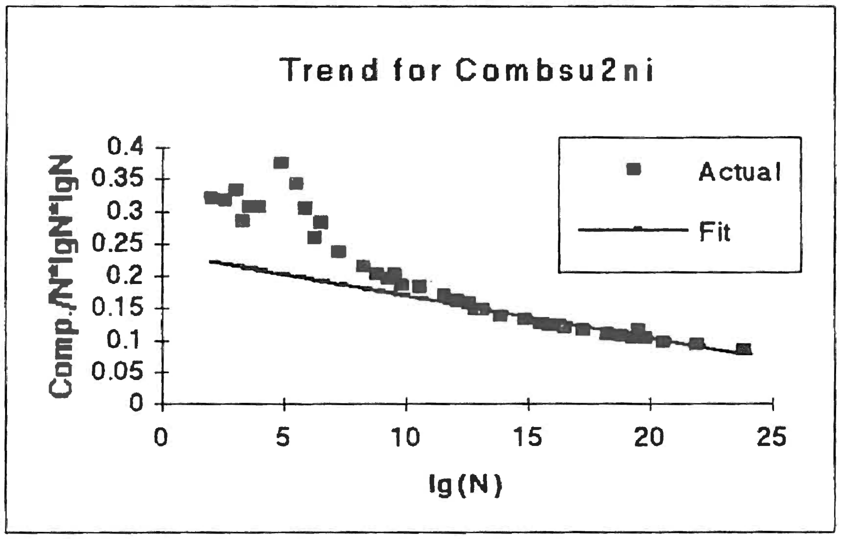

Figure 34. N*lg^2(N) Diagnostic Fit Chart for Combsu2

TABLE XXVII

N*LG(N) PROBING FIT CHART FOR COMBSU2NI
AND SHELLSU15 IN CONTRAST TO
HEAPSORT AND QUICKSORT
(COMPARISONS/N*LG(N))

Prog./Size	150	1500	15000	150000	1500000	15000000
Combsu2ni	1.71	1.92	1.93	1.98	2.01	2.03
Shellsu	1.20	1.38	1.49	1.58	1.64	1.68
Heap	1.58	1.72	1.78	1.82	1.85	1.87
Quick	1.51	1.46	1.37	1.39	1.35	1.35

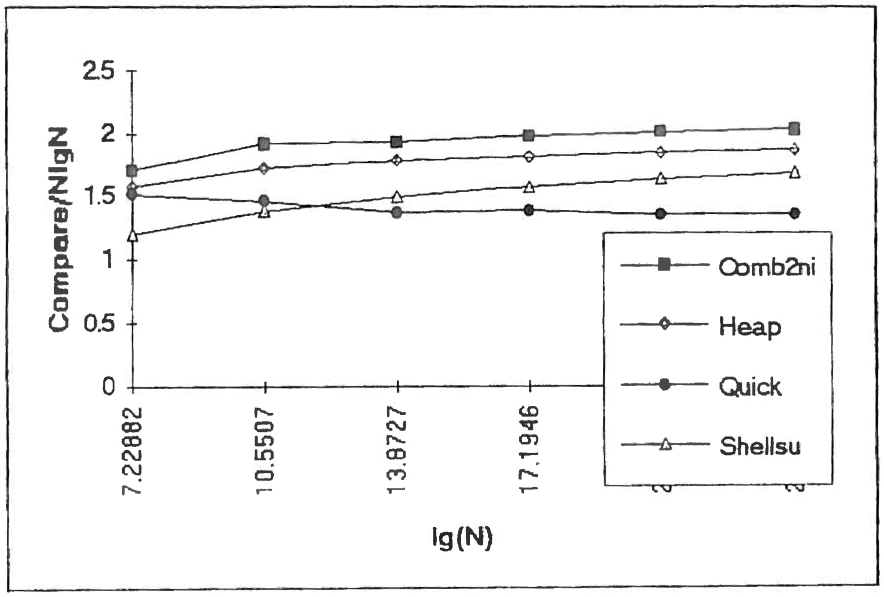

Figure 35. Line Chart for Table XXVII

Although I did encounter once a bad case for one version of my Combsort and subsequently found the way to construct a bad list for Combsort, I also rigorously tested Combsort11 with random data: 2000 lists for very small sizes, 50 for very large sizes, and 200 for the rest, hoping to find some very bad cases but could not find a single one. I perceive that the incidence of the bad cases is small. When the Two-way Combsort is used as in Program Q, it is logical to think that the probability of bad cases is negligibly low. So the bottom line of my suggestions about the scope of implementations concerning Combsort is this: Use Combsu (Program D) when one is in a hurry to write a sorting routine, but revise it later (Program Q) before it is heavily used. The performance of Program Q is about as fast as the best version of Shellsort we found.

CHAPTER VI

CONCLUSIONS AND FUTURE WORKS

Although our thesis topic is Combsort, there are quite a few insights for making other sorting methods run faster. Perhaps the most notable one is that we proved that we could simplify the innermost loop of Heapsort. A very useful partition scheme "Median-of-Four" for Quicksort not only reduces the worst case probability but also gains some speed in the average case. We found some good ways to get good increment sequences for Shellsort that beat most other known sequences easily by some noticeable margin. We even improved Insertion Sort by finding an easy way to set the sentinel and help sort at the same time. We discovered that we could get rid of the dirty pass flag for Bubblesort when we tried to improve Combsort.

We improved the running speed of Combsort by as much as 130%. We were able to construct a nearly worst case scenario for Combsort and Combsort11 which took $O(N^2)$ to sort. We also proposed Two-way Combsort to cope with this kind of bad cases. The Two-way Combsort we have developed is an efficient, general-purpose internal sorting algorithm. There are several properties found and proved for Combsort. For data with any partial order, it will do better according to the authors of Combsort [15] and some of my test results. We also have shown a version of Shellsort which performs better than any other versions of its kind. Because of the relatively large Shrink Factor it uses (2.25 to 2.5), it may even beat Quicksort for a list with a high degree of sortedness. Heapsort is more or less indifferent to the distribution of the data. Quicksort is notorious in its worst case behavior. Therefore, we should think a little more positively for the efficiencies of Combsort and Shellsort in practice. As they both use a gap sequence, parallelism will come more naturally. It might be a whole new ball game for them to compete with Quicksort in parallelism.

When using pointers in C, we have shown that some algorithms (Combsort and Quicksort) have significant speed-ups, but other algorithms like Heapsort do not run any faster. Is this a language-dependent feature or a portable feature? We know that down at a lower level like assembly code, there is no such thing as "pointers". But what was the pointer that did the magic work for our programs in C? It takes some moments of thinking to answer that question. Let's review some characteristics of pointers in C.

1. The relationship between pointers and arrays is so strong that no operations peformed by the array indices cannot be achieved by the pointer manipulations.
2. Pointer increment is scaled by the objects this pointer points to. For example, if we increment a pointer point to a long integer, the pointer will advance by the size of a long integer (probably 4 bytes or 8 bytes).
3. An array name is a pointer expression.
4. A reference to an array is converted by the compiler to a pointer to the beginning of the array.

The fundamental difference between the array and the pointer is that a pointer is a variable but an array name is a constant. The only arithmetic operation allowed for pointers are:

- Adding or substracting a pointer and an integer.
- Substracting or comparing two pointers.

From characteristic #4 above, we can see that a reference to an array element a[i] for example, is actually done by a pointer pointing to the beginning of the array, then add 'i' to that pointer. Looking at the innermost loops of Quicksort and Combsu, we increment or decrement the array indices to get the next pair of elements for comparing. It is fruitful to use pointers in those cases because we can save as many '+' operations as these a[i] are referenced and perform incrementation or decrementation directly on the pointers. The operation performed on the variable in the innermost loop of Shellsort is an addition or substraction of the magnitude of the gap (gap>=1). We gain nothing for using pointers in this case because we will still need to add or substract the gap after we reference to the address pointed to by the pointer. However, the two variables in the middle loop of Shellsort can be done with incrementation or decrementation (see Program O in Appendix B) and that's where the modest savings come from (not the inner loop). The operations on the variables of both the inner and outer loop of Heapsort involve illegal operations (mutiplication and division) for the pointers. Therefore we are bound to use another variable for these operations if we insist on using pointers, and perform exactly in the same fashion as how the array indices work. Evidently there will be no gain but pain (poor clarity) for this type of pointer implementation. In short, we can utilize the pointer instead of the array to speed up if the only operations on the array indices are incrementation and/or decrementation. That's the reason why the pointer version of Combsu will run as fast as Heapsu and lag behind Quicksu by only about 60% on random lists. Further, both pointers and array indices can be put into the machine registers (by declaration); otherwise, they will be treated as automatic variables and put in RAM. We are not clear how exactly pointers are implemented in C in terms of assembly or machine language level. One thing is for sure, the array implementation in C is quite different from the array in the assembly implementation, because there is no such thing as pointers in assembly and a reference to a C array is converted to a pointer by the compiler before using it. Therefore, the advantage of using pointers is probably language-dependent. If a program is coded properly in assembly or compiled by an 'optimized' compiler, the non-pointer versions in this paper will probably approach the pointer versions. In these two cases, array indices probably will be held in index registers at all times; our non-optimized array references probably result in indices being stored in RAM, causing extra memory references and loss of speed. To verify this argument we compared some of our results with Knuth's [14] rusults. His MIX (an assembly language) running time of Heapsort is 2.15 times of that of Qui_3_10. Our non-pointer versions of these two programs give a ratio of 1.99 (an acceptable difference due to different machines used) while the pointer version reveals a ratio of 2.64 (probably more difference than one would expect due to different machines used).

There are several suggestions for future research on the sorting methods we have investigated.

1. Can we construct a bad case list for Combsu2ni?
2. What is the minimal number of passes (the best sequence) for Combsort to sort a list without the need to call Insertion Sort or utilizing some form of dirty pass flag to signal whether the sorting is done?
3. Can we find good fits for Combsu2ni or Shellsu on their asymptotic averages?
4. Table V shows the moving distance frequency for one version of Shellsort. Could we take advantage of the fact that most elements in the list will not go very far and design an even more efficient Shellsort? What about Combsort?
5. We did a lot of code twiddling to make the programs run faster besides those algorithmic improvements. What result would we get if we use some "optimized" compilers? It would be an interesting aspect to probe how optimized our compiler is by testing our tuned-up versions of programs.

A SELECTED BIBLIOGRAPHY

[1] A. Aho, J. Hopcroft and J. D. Ullman. Data Structures and Algorithms. Addison-Wesley, Reading, MA (1983).

[2] Bauer, L. B. "An Empirical Study of Shellsort." Unpublished M. S. Thesis, Oklahoma State University, 1980.

[3] Boothroyd, J. "Shellsort: Algorithm 201." Communications of the ACM, 6 (1963), 445.

[4] Brown, M. R. "Implementation and analysis of binomial queue algorithms," SIAM Journal of Computing, 7, 3, (August, 1978).

[5] C. R. Cook and D. J. Kim, "Best Sorting Algorithm for Nearly Sorted Lists." Communications of the ACM, 23 11 (1980), 620-624.

[6] Hannu Erkio, "The Worst Case Permutation for Median-of-Three Quicksort," The Computer Journal, 27 3 (1984), 276-277.

[7] J. Esakow and T. Weiss, Data Structures - An Advanced Approach Using C. Prentice Hall, Englewood Cliffs, NJ (1989).

[8] G. Gonnet, Handbook of Algorithms and Data Structures. Addison-Wesley, Reading, MA (1984).

[9] P. Helman and R. Veroff, Walls and Mirrors - Intermediate Problem Solving and Data Structures. Benjamin-Cummings, Menlo Park, CA (1988).

[10] C. A. R. Hoare, Algorithm 64: Quicksort. Communications of the ACM, 4, 7, 321 (July 1961).

[11] E. Horowitz and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Rockville, MD (1977).

[12] Hibbard, Thomas N. "An Empirical Study of the Minimal Storage Sorting." Communications of the ACM, 3 (1960), 206-213.

[13] J. Incerpi and R. Sedgewick, Improved Upper Bounds on Shellsort, Journal of Computer and System Sciences, 31 (2) 210 224, (1985).

[14] Knuth, D. E. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, MA (1973).

[15] S. Lacey and R. Box, "A Fast, Easy Sort." Byte, April 1991, 315-320.

[16] K. Melhorn, Data Structures and Algorithms 1: Sorting and Searching. Springer. New York (1984).

[17] Dalia Motzkin, "Meansort," Communications of the ACM, 26 4 (1983), 250-251; "More about Meansort," Communications of the ACM, 27 7 (1984), 719-722.

[18] J. Ian Munro and Venkatesh Raman, "Sorting with Minimum Data Movement," Journal of Algorithms, 13, 374-393 (1992).

[19] V. Pratt, Shellsort and Sorting Networks. Garland Publishing, New York (1979). (Originally presented as the author's Ph. D. thesis, Stanford University, 1971).

[20] R. Sedgewick, "A New Upper Bound for Shellsort", Journal of Algorithms 2 (1986) 159-173.

[21] R. Sedgewick, Algorithms. Addison-Wesley, Reading, MA (1988).

[22] R. Sedgewick, "Implementing Quicksort Programs," Communications of the ACM 21 10 (1978) 847-856.

[23] D. L. Shell, "A High-Speed Sorting Procedure". Communications of the ACM 2 (7) 30-32, (1959).

[24] B. Singh and T. L. Naps, Introduction to Data Structures. West Publishing Co, St Paul, MN (1985).

[25] H. F. Smith, Data Structures - Form and Function. Harcourt Brace Jovanovich, New York (1987).

[26] D. Stubbs and N. Webre, Data Structures with Abstract Data Types and Pascal. Brooks/Cole, Monterey, CA (1989).

[27] R. L. Wainwright, "A Class of Sorting Algorithms Based on Quicksort," Communications of the ACM 28 4 (1985), 396-402.

[28] M. A. Weiss, "Empirical Study of the Expected Running Time of Shellsort." The Computer Journal 34 1 88-91 (1991)
[29] M. A. Weiss and R. Sedgewick, Journal of Algorithms 11 242-251 (1990).

[30] H. L. Yu "Investigations of Shellsort." Unpublished M. S. Thesis, Oklahoma State University, 1984.

APPENDIX A

PROOFS THAT THE COMBINED TEST IN THE INNERMOST LOOP OF HEAPSORT CAN BE SIMPLIFIED TO A SINGLE TEST, AND OF PROPERTY 4.1

Proof: The test for "j<h_idx" (h_idx is the heap size) can be got rid of in the inner loop of procedure downheap() in Program M. The same procedure in Program N is its revision.

The reason we need to make sure that j<size (heap size) is that we don't want "if data[j]<data[j+1]", otherwise. Now if the heap size at any moment is an odd number, we really don't need to check j<h_idx because after "j:=j*2", j is an even number and is less than the heap size (for the loop control statement is "j<=size/2"). Now assume that our 'data size' is an even number (1000 as an example).

1. Our first pick is correct; we start building the heap bottom-up from j:=(size-1)/2 (499 for our example) instead of j:=size/2 (500); thus we always keep j smaller than size. Inside the loop we let j:=j*2 (j=998 now), then j<=size-2 (998) and j+1<=size-1 (999). Since size is an even number, only the last item (j=1000) may not be in the heap after the heap construction phase, and since we have managed to avoid the situation where the last item is the largest one (we let data[1000]<=data[1] with a call to compare2(1,size) before constructing the heap), the largest item must be on top of the heap after the heap is built, and it is the first pick of our sorting phase.

2. Our second pick is correct; we insert the last item (data[1000]) back into the heap from the top (j=1) and form an odd-size heap since the heap size decreases by one (h_idx=999) and becomes an odd number. Therefore, the second pick is also correct.

3. Our second insertion (insert data[999]) will form a heap; the second insertion must form a heap (heap size=998) if the item (data[999]) being inserted is not in the path of the last item (j=998) in the heap. (In other words, we don't want the situation where coincidentally, the item being inserted (data[999]) is the largest one in the heap and is inserted into the last position (j=998).) Even if this item is inserted into the last position, we know from the previous discussion that it is all right as long as the last item is not the largest in the heap. We happened to have that since the previous step was from an odd-size heap (j=999, data[j] was not left out of the heap).

4. The rest of the picks and insertions are correct; accordingly, we alternate the heap size even and odd (step 1 and 2) throughout the sorting phase and get the sorting job done.

5. It works for an odd-size heap to begin with; just relax the first step of the above and an identical proof holds when the list size is an odd number.

Property 4.2 If a list is k-ordered, and then Combsorted with gap h, it remains k-ordered.

Proof: Let X_i be any element in the list, assume h+1<=i<=size-k-h. Being k-ordered, $X_i<=X_{i+k}$, we need to prove that after one pass of Combsort with gap h, $X_i<=X_{i+k}$ is still valid. When the list is Combsorted with gap h, the element in slot 'i' can only be one of the three elements (X_{i-h}, X_i, X_{i+h}), and one of the three (X_{i+k-h}, X_{i+k}, X_{i+k+h}) will reside in slot 'i+k'. There are nine possible combinations in a 3x3 permutation. We can quickly eliminate three combinations - (X_{i-h}, X_{i-h+k}), (X_i, X_{i+k}), and (X_{i+h}, X_{i+h+k}) - by the definition of being k-ordered. There are six cases left to be considered:

Case 1: (X_{i-h}, X_{i+k}) - Since $X_{i-h}<=X_{i-h+k}$ (k-ordered) and $X_{i-h+k}<=X_{i+k}$ (otherwise they must have been swapped after being Combsorted with gap h), $X_{i-h}<=X_{i+k}$ is true.

Case 2: (X_{i-h}, X_{i+k+h}) - Since $X_{i-h}<=X_{i+h}$ (otherwise they must have been swapped after being Combsorted with gap h) and $X_{i+h}<=X_{i+h+k}$ (k-ordered), $X_{i-h}<=X_{i+k+h}$ is true.

Case 3: (X_i, X_{i+k-h}) - Since $X_i<=X_{i+k}$ (k-ordered) and $X_{i+k}<=X_{i+k-h}$ (that's why they have been swapped at gap h), $X_i<=X_{i+k-h}$ is true.

Case 4: (X_i, X_{i+k+h}) - Since $X_i<=X_{i+h}$ (otherwise they must have been swapped after being Combsorted with gap h) and $X_{i+h}<=X_{i+h+k}$ (k-ordered), $X_i<=X_{i+k+h}$ is true.

Case 5: (X_{i+h}, X_{i+k-h}) - Since $X_{i+h}<=X_i$ (that's why they have been swapped at gap h) and $X_{i+k}<=X_{i+k-h}$ (that's why they have been swapped at gap h); we know that $X_i<=X_{i+k}$ (k-ordered), so $X_{i+h}<=X_{i+k-h}$ is true.

Case 6: (X_{i+h}, X_{i+k}) - Since $X_{i+h}<=X_i$ (that's why they have been swapped at gap h) and $X_i<=X_{i+k}$ (k-ordered), $X_{i+h}<=X_{i+k-h}$ is true.

We have exhausted all cases for a general assumption of i (h+1<=i<=size-k-h). For a special 'i' (say i<=h or i>size-k-h), we just need to select partial cases from our proof in the above, and the proof will still be valid.

APPENDIX B

SELECTED C PROGRAM CODES FOR THIS PAPER

```
/*  Corresponds to Program A (Figure 1) in the text */
/* Assume a global array data[] and global variable "size" */
void comb()
{
int switches, i, j, top, gap;
int hold;
        gap=size;
        do {
                gap=(int)((float)gap/SHRINKFACTOR);
                if (gap==0)  gap=1;
                switches=0;     // dirty pass flag
                top = size - gap+1;
                for (i=0;i<top;i++)
                {
                        j=i+gap;
                        /*  swap  */
                        if (data[i]>data[j])   {
                                hold=data[i];
                                data[i]=data[j];
                                data[j]=hold;
                                ++switches;
                        }
                }
        }  while (switches || (gap>1));
}
```

```
/*  Corresponds to Program B (Figure 2) in the text */
void comb11()
{
int switches, i, j, top, gap;
int hold;
        gap=size;
        do {
                gap=(int)((float)gap/SHRINKFACTOR);
                switch(gap)
                {
                        case 0: gap=1;    // bubble sort
                                break;
                        case 9:
                        case 10:        gap=11;
                                break;
                        default:        break;
                }
                switches=0;    // dirty pass flag
                top = size - gap;
                for (i=0;i<top;i++)
                {
                        j=i+gap;
                        /*  swap  */
                        if (data[i]>data[j])  {
                                hold=data[i];
                                data[i]=data[j];
                                data[j]=hold;
                                ++switches;
                        }
                }
        } while (switches || (gap>1));
}
```

```
/*  Corresponds to Program C (Figure 3) in the text */
#define SHRINKFACTOR 1.3   /* optimal Shrink Factor */

/* use pointer and get rid of the dirty pass flag */
/* in calling procedure, to call this procedure might look like: */
/* comb_sort(data, data+size-1);  (data[ ] need not to be global) */
void comb_sort(lo, hi)
int *lo, *hi; /* pointer to the first and last items of array */
{
register int *i, *j;  /* if pointer can be put in registers */
int  hold, gap;
        gap=(hi-lo+1)/SHRINKFACTOR;  /* better initial gap */
        do {
                for (j=hi,i=j-gap;i>=lo;--i,--j)
                {
                        if (*i>*j)   { /* watch the order of swap */
                                hold=*i;  /* if j first, we might */
                                *i=*j; /* waste a pass when */
                                *j=hold;  /* only the first two swap */
                        }
                }
                if (gap>1) gap/=SHRINKFACTOR; /* for most passes */
                else  /* gap=1, don't change the gap size */
                if (hold!=*hi) hold=*hi; /* smart dirty pass flag */
                else break;  /* break out of the loop */
        }  while (1);  /* infinite loop until a "break" or "goto" */
}
```

```
/*  Corresponds to Program D (Figure 8) in the text */
#define SHRINKFACTOR 1.3   /* optimal Shrink Factor */

void comb_sort(lo, hi)
int *lo, *hi; /* pointer to the first and last items of array */
{
register int *i, *j;  /* if pointer can be put in registers */
int  hold, gap;
        gap=(hi-lo+2)/3;  /* better initial gap, +2 for size<3 */
        do {
                for (j=hi,i=j-gap;i>=lo;--i,--j)
                {
                        if (*i>*j)   { /* watch the order of swap */
                                hold=*i;  /* if j first, we might */
                                *i=*j; /* waste a pass when */
                                *j=hold;  /* only the first two swap */
                        }
                }
                if (gap>9) gap/=SHRINKFACTOR; /* for most passes */
                else  /* change the course of Shrink Factor by */
                        if (gap>1) --gap;  /* simply decrementing the gap */
                        else  /* gap=1, don't change the gap size */
                                if (hold!=*hi) hold=*hi; /* smart dirty pass flag */
                                else break;  /* break out of the loop */
        }  while (1);  /* infinite loop until a "break" or "goto" */
}
```

```
/*  Corresponds to Program E (Figure 11) in the text */
#define SHRINKFACTOR 1.3   /* optimal Shrink Factor */

void comb_insert(lo, hi)
int *lo, *hi; /* pointer to the first and last items of array */
{
void insertion(int *, int *);
register int *i, *j;  /* if pointer can be put in registers */
int  hold, gap;
        gap=(hi-lo+2)/3;  /* better initial gap, +2 for size<3 */
        do {
                for (j=hi,i=j-gap;i>=lo;–i,–j)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                        }
                }
                gap/=SHRINKFACTOR;
        }  while (gap>6);  /* 6 is cut of point for Insertion Sort */
        insertion(lo, hi);
}
void insertion(lo, hi)
int *lo, *hi;
{
register int *i, *j;
int v;
        /* guaranteed sentinel */
        for (j=hi, i=j-1; j>lo; i–, j–)   {
                if (*i>*j)  {
                        v=*j;
                        *j=*i;
                        *i=v;
                }
        }
        for (i=lo+1; i<=hi; i++) {
                v=*i;
                j=i;
                while (*(j-1)>v)  {
                        *j=*(j-1);
                        j–;
                }
                *j=v;
        }
}
```

```
/*  Correspond to Program F (Figure 14) in the text */
#define SHRINKFACTOR 1.44   /* optimal Shrink Factor */

void comb_insert2(lo, hi)
int *lo, *hi; /* pointer to the first and last items of array */
{
void insertion(int *, int *);
register int *i, *j;  /* if pointer can be put in registers */
int  hold, gap;
        gap=(hi-lo+2)/3;  /* better initial gap, +2 for size<3 */
        do {
                for (i=lo, j=i+gap;j<=hi; ++i, ++j)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                        }
                }
                if (gap<=3)  break;
                gap/=SHRINKFACTOR;
                (gap+=(gap&1))--;  /* decrements even gap */
                for (j=hi,i=j-gap;i>=lo;--i,--j)
                {
                        if (*i>*j)  {
                        hold=*i;
                        *i=*j;
                        *j=hold;
                }
                gap/=SHRINKFACTOR;
                (gap+=(gap&1))--;  /* decrements even gap */
        }  while (gap>3);  /* 3 is cut of point for Insertion Sort */
        insertion(lo, hi);
}
```

```
/*  Corresponds to Program G (Figure 15) in the text */
void shellh(size)
int size;
{
int i,j,h;
int v;
        for (h=1; h<=size/8; h<<=1);
        h--;
        do {
                for (i=h; i<size; i++)   {
                        j=i;
                        v=data[i];
                        while (j>=h && data[j-h]>v)   {
                                data[j]=data[j-h];
                                j-=h;
                        }
                        data[j]=v;
                }
                h>>=1;
        } while (h>0);
}


/*  Correspond to Program H (Figure 16) in the text */
void shellk(size)
int size;
{
int  i, j, h;
int  hold;
        for (h=1; h<=size/9; h=3*h+1);
        do  {
                for (i=h; i<size; i++)
                {
                        j=i;
                        hold=data[j];
                        while (j>=h && data[j-h]>hold)   {
                                data[j]=data[j-h];
                                j-=h;
                        }
                        data[j]=hold;
                }
                h/=3;
        } while (h>0);
}
```

```
/*  Correspond to Program I (Figure 17) in the text */
/* the array start with 0 for the size=1, otherwise, cannot set the init. gap */
/* Baur's idea to avoid unconditional save is implemented here */
unsigned ary[21]={1,5,19,41,109,209,505,929,2161,3905,8749,16001,
36449,64769,146305,260609,587521,1045505,2354689,4188161,9427969};
void shells(size)
int size;
{
register int  i,j;
int  k, h, hold;
        for (k=20,h=size/3*2+1; ary[k]>h; k--); /* +1 for small */
        /* the optimal initial gap is about 1/3 size to 2/3 size */
        do  {
                h=ary[k--];
                for (i=h; i<size; i++)   {
                        j=i-h;
                        hold=data[i];
                        if (data[j]>hold)  {
                                data[i]=data[j];
                                while (j>=h && data[j-h]>hold)   {
                                        data[j]=data[j-h];
                                        j-=h;
                                }
                                data[j]=hold;
                        }
                }
        }  while (h>1);
}
```

```
/*  Corresponds to NO Program in the text but is compared in the charts and tables */
/* Quicksort with median element as pivot - so that sentinel can be eliminated */
/* This program does not appear in the text, but is used to compare for its running time */
void quick_sort(low_ptr, high_ptr)
int *low_ptr, *high_ptr;
{
int *pivot_ptr;
extern int *partition(int *, int *);

        if (high_ptr>low_ptr)  {
                pivot_ptr = partition(low_ptr, high_ptr);
                quick_sort(low_ptr, pivot_ptr-1);
                quick_sort(pivot_ptr, high_ptr);
        }
}
int *partition(low_ptr, high_ptr)
register int *low_ptr, *high_ptr;
{
register int  pivot=*(low_ptr+(high_ptr-low_ptr)/2), temp;
/* upon termination, left<pivot<right */
        while (low_ptr<=high_ptr)  {
                while (*low_ptr<pivot)  low_ptr++;
                while (*high_ptr>pivot)  high_ptr--;
                if (low_ptr<=high_ptr)  {
                        temp=*low_ptr;
                        *low_ptr=*high_ptr;
                        *high_ptr=temp;
                        low_ptr++; high_ptr--;
                }
        }
        return low_ptr;
}
```

```
/*  Corresponds to Program J (Figure 19) in the text */
void quick_sort(lo, hi)
int  *lo, *hi;
{
register *i, *j, pivot;
int  temp, *m;
          if (hi-lo>9)  {
                    m=lo+(hi-lo)/2;
                    if (*lo>*hi) { pivot=*lo; *lo=*hi; *hi=pivot; }
                    pivot=*m;
                    if (*lo>pivot) { *m=*lo; *lo=pivot; pivot=*m; }
                    else if (pivot>*hi) { *m=*hi; *hi=pivot; pivot=*m; }
                    i=lo+1; j=hi-1; goto start;
                    while (i<=j) {
                              temp=*i;
                              *i=*j;
                              *j=temp;
                              i++; j--;
start:                        while (*i<pivot)  i++;
                              while (*j>pivot)  j--;
                    }
                    quick_sort(lo, j);
                    quick_sort(i, hi);
          }
}
```

```
/*  Corresponds to Program K (Figure 20) in the text */
void quick_sort(lo, hi)
int  *lo, *hi;
{
register int  *i, *j, pivot;
int  temp;
          if (hi-18>lo)  {
                    temp=(hi-lo)/3;
                    i=lo+temp; j=hi-temp;
                    if (*lo>*j) { pivot=*lo; *lo=*j; *j=pivot; }
                    if (*i>*hi) { pivot=*i; *i=*hi; *hi=pivot; }
                    if (*lo>*i) { pivot=*lo; *lo=*i; *i=pivot; }
                    if (*j>*hi) { pivot=*j; *j=*hi; *hi=pivot; }
                    pivot=(*i>>1)+((*j+1)>>1);
                    i=lo+1; j=hi-1;
                    goto start;
                    while (i<=j) {
                              temp=*i;
                              *i=*j;
                              *j=temp;
                              i++; j--;
start:                        while (*i<pivot)  i++;
                              while (*j>pivot)  j--;
                    }
                    quick_sort(lo, j);
                    quick_sort(i, hi);
          }
          else  {
                    j=hi;
                    do {
                              for (i=lo; i<j; i++, j--)
                                        if (*i>*j)  { pivot=*i; *i=*j; *j=pivot; }
                    } while (lo<j);
                    for (i=lo+2; i<=hi; i++)  {
                              j=i;  v=*i;
                              while (*(j-1)>v)  { *j=*(j-1);  j--; }
                              *j=v;
                    }
          }
}
```

```
/*  Corresponds to Program L (Figure 21) in the text */
void quick_sort(lo, hi)
int  *lo, *hi;
{
int  temp;
register int  *i, *j, pivot;
int  *m;
        if (hi-18>lo)  {
                temp=(hi-lo)/4;
                i=lo+temp; j=hi-temp; m=i+temp;
                if (*lo>*hi) { pivot=*lo; *lo=*hi; *hi=pivot; }
                if (*i>*j) { pivot=*i; *i=*j; *j=pivot; }
                if (*i>*m) { pivot=*i; *i=*m; *m=pivot; }
                else if (*m>*j) { pivot=*j; *j=*m; *m=pivot; }
                else pivot=*m;
                if (*lo>pivot) { *m=*lo; *lo=pivot; pivot=*m; }
                else if (pivot>*hi) { *m=*hi; *hi=pivot; pivot=*m; }
                i=lo+1; j=hi-1; goto start;
        SAME AS ABOVE PROGRAM FROM NOW ON ........
}
```

```
/*  Corresponds to Program M (Figure 23) in the text */
/* heapsort from p.152-156 of Algorithms 2nd. Ed. by Robert Sedgewick - assume
size>=3 */
void heapsort(size)
int  size;
{
int  k;
int  t;
void downheap(int);
        h_idx=size;
        for (k=h_idx/2;k>=1;k--)  downheap(k);
        do  {
                t=data[1]; data[1]=data[h_idx];
                data[h_idx--]=t;
                downheap(1);
        }  while (h_idx>1);
}
void downheap(k)
int  k;
{
int  j, v;
        v=data[k];
        while (k<=h_idx/2)  {
                j=k+k;
                if (j<h_idx && data[j]<data[j+1])    j++;
                if (v>=data[j])   break;
                data[k]=data[j];  k=j;
        }
        data[k]=v;
}
```

```
/* Corresponds to Program N (Figure 24) in the text */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int size;
int *data;
void main(int argc, char *argv[])
{
extern void heapsort();
extern void compare2(int, int);
clock_t start, end;
        if (argc !=2)  { puts("require one and only one argument!");    exit(1); }
        size=atoi(argv[1]);
        data = new int[size+2];
        randomize();
        for (int i=1; i<=size; i++)  {  *(data+i) = rand();  }
        start=clock();
        if (size>2) { compare2(1,size); heapsort(); compare2(2,3);  }
        else  { data[++size]=data[1]; if (size==2) compare2(2,3);  }
        end=clock();
        delete data;
        printf("\n%f\n",  (end-start)/CLK_TCK);
}
/* assume for size >2 */
void heapsort()
{
void downheap(int, int, int);
register int h_idx=size-1>>1;
        for (;h_idx>=1;h_idx--) {  downheap(h_idx<<1,data[h_idx],size); }
        h_idx=++size;    // for extra space
        while (h_idx>4)  {      // require cutoff point at h_idx>4
                data[h_idx--]=data[1];
                downheap(2,data[h_idx],h_idx-1);
        }
        data[h_idx--]=data[1];
}
void downheap(int i,int v,int h_idx)
{
register int j=i;
        do {
                if (data[j]<data[j+1])  j++;
                data[j>>1]=data[j];
                j<<=1;
        }  while (j<h_idx);
        j>>=1;
        if (v>data[j])  {
                j>>=1;
                while (v>data[j>>1] && j>=i)  {  data[j]=data[j>>1];  j>>=1; }
        }
        data[j]=v;
}
void compare2(int i, int j)
{
        if (data[i]>data[j]) { int t=data[i]; data[i]=data[j]; data[j]=t; }
}
```

```
/*  Corresponds to Program O (Figure 29) in the text */
void shell(lo, hi)
int  *lo, *hi;
{
register  int *i, *j;
int  v, h, *bound;
        h=178;
        v=(hi-lo+3)/4;
        while (v>h)  h=(h<<1)+(h>>2)+1;
        do {
                bound=lo+h;
                for (i=lo,j=bound; j<=hi; i++,j++)  {
                        if (*i>*j)  {
                                v=*j; *j=*i;
                                if (i<bound || *(i-h)<=v)  *i=v;
                                else  {
                                        do {
                                                *i=*(i-h);
                                                i-=h;
                                        }  while (i>=bound && *(i-h)>v);
                                        *i=v;
                                        i=j-h;
                                }
                        }
                }
                if (h>225) h=(h<<2)/9;
                else {
                        if (h==1) break;
                        h=(h<<1)/5;
                }
        }  while (1);
}
```

```
/* Corresponds to Program P (Figure 30) in the text */
void comb(lo, hi)
int *lo, *hi;
{
void insertion(int *, int *);
int gap, hold;
register int *i,*j;
        gap=26;  hold=size/3;
        while (gap<hold)  (gap*=SHRINKFACTOR)++;
        do {
                for (i=lo,j=lo+gap;j<=hi;i++,j++)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                        }
                }
                gap/=SHRINKFACTOR;
                if (gap<5) {
                        gap++;
                        if (gap==3) break;
                }
                for (j=hi,i=j-gap;i>=lo;i--,j--)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                        }
                }
                gap/=SHRINKFACTOR;
                if (gap<5) {
                        gap++;
                        if (gap==3) break;
                }
        } while (1);
        if (hi>lo) insertion(lo,hi);
}
```

```
/*  Corresponds to Program Q (Figure 33) in the text */
/* no resetting even to odd(SF=1.42), better fit for N*Lg(N) (1 item) */
/* use idx to check the unsorted part for gap=1, SF=1.443 */
/* get rid of insertion and dirty pass flag, SF=1.43 */
/* from comb_su2, but change 13-9-7-5-4-3-2 or 11-7-5-4-3-2, SF=1.443 */
#define SHRINKFACTOR  1.42;
void comb(int *lo, int *hi)
{
void insertion(int *, int *);
int gap, hold;
register int *i,*j,*idx;
        gap=(hi-lo+2)/3;  // make sure size=2 works
        do {
                for (i=lo,j=lo+gap;j<=hi;i++,j++)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                                idx=i;
                        }
                }
                if (gap>10) gap/=SHRINKFACTOR;
                else {
                        if (gap>1) gap--;
                        else {
                                if (lo<idx) { hi=idx; idx=hi; }
                                else break;
                        }
                }
                for (j=hi,i=j-gap;i>=lo;i--,j--)
                {
                        if (*i>*j)  {
                                hold=*i;
                                *i=*j;
                                *j=hold;
                                idx=j;
                        }
                }
                if (gap>10) gap/=SHRINKFACTOR;
                else {
                        if (gap>1) gap--;
                        else {
                                if (idx<hi) { lo=idx; idx=lo; } /* might waste a pass */
                                else break;
                        }
                }
        }  while (1);
}
```

APPENDIX C

NEARLY WORST CASE CONSTRUCTION FOR COMBSORT

```
/* Major tasks performed by this procedure - T# will be referenced  */
/* T1. Decides the cut_point_gap_size for the black sheep starts to move */
/* T2. Calls chkcls() to ensure the black sheep will not move before CPGS */
/* T3. Places the rest of list in reverse order into each empty position */

void anticomb(size)
int size;  /* size of data */
{
extern void chkclr(int, int, int);
int j, pos;
int cut_point_gap_size;  /* when should the black sheep move */

  pos=2;         /* T1 */
  j=size/10;
  while (j>0) { pos+=2; j/=10; }
  cut_point_gap_size=size/pos;

  pos=size-1;  /* change of pos could help optimize the worst case */
  d=1;         /* The first data value is 1 */

  chkclr(size,pos,cut_point_gap_size);  /* T2 */

  list[pos]=d++;  /* place the black sheep at pos */
  for (j=size-1;j>=0;j--)         /* T3 */
    if (list[j]==0) list[j]=d++;  /* Change to in order speed up little */
}

/* Major tasks performed by this procedure - T# will be referenced        */
/* T1. Checks and fills the empty positions in front with small numbers   */
/*     that are smaller than the black sheep will be                    */
/* T2. Recursively calles itself after filling each empty position to   */
/*     assure that there will be no interference to overtake the black    */
/*     sheep before the cut_point_gap_size                         */

void chkclr(size,pos,g) /* size: size of data, g: current gap size  */
int size, pos, g;       /* pos: position to check for clearance */
{
int j, gap;
  gap=size/SHRINKFACTOR;  /* do it according to what Combsort would do */
  while (gap>g) {  /* because Combsort uses diminishing gap sizes */
    if (pos>=gap) j=pos-gap;    /* T1 */
    else j=pos;
    while (j>=gap) j-=gap;    /* decides the starting position to check */
    while (j<pos)  {      /* doing in other direction will not work */
        if (list[j]==0) list[j]=d++;  /* fills the small number if empty */
```

```
          chkclr(size,j,gap);   /* T2 */
          j+=gap;
     }
     gap/=SHRINKFACTOR;      /* same as Combsort */
   }
}
```

APPENDIX D

AVERAGE AT VARIOUS SIZES AND STANDARD ERRORS FOR PROGRAM O AND PROGRM Q

Compare	Combsu2ni		Shellsu	
Size	Average	Stdev	Average	Stdev
4	5.1	0.99	5.9	1.447
6	12.7	1.77	12.7	2.050
8	23.9	1.91	20.5	3.371
10	31.6	1.90	29.8	3.270
15	70.4	2.07	55.6	4.893
30	270.8	1.40	149.3	14.186
60	636.0	1.76	396.3	19.394
90	1064.4	17.76	658.6	23.356
150	1858.9	34.44	1298.1	31.125
300	4377.3	73.62	3104.2	54.865
450	7098.0	82.26	5091.2	108.243
600	10068.8	98.19	7219.3	78.341
750	13732.4	238.95	9624.8	170.382
900	16136.0	228.32	11954.8	195.985
1500	30461.9	597.57	21776.6	194.274
3000	67665.8	575.71	49130.6	252.535
4500	106979.2	788.04	78744.2	564.170
6000	147917.8	978.90	109684.7	606.621
7500	185705.3	2320.72	141612.9	864.660
9000	231960.2	604.52	175368.7	476.571
15000	402111.9	4291.11	310992.6	954.500
30000	867518.2	9773.39	680821.0	1576.385
45000	1354186.7	14451.72	1074444.6	3210.411
60000	1858278.3	17159.71	1476409.2	1955.129
75000	2433401.9	20469.57	1889951.7	4284.169
90000	2915081.3	29701.21	2312121.4	3143.516
150000	5112774.4	44353.33	4070271.1	4752.074
300000	11016525.2	253727.60	8721865.6	9841.806
450000	16887703.1	89997.36	13577030.4	11384.870
600000	22925621.8	131973.30	18618873.0	21068.360
750000	32807208.4	753596.00	23718172.5	23998.290
900000	35803641.9	172998.80	28901483.9	29416.670
1500000	61976199.2	697842.90	50363189.8	40982.830
4000000	176853509.8	1499164.00	145379526.6	52833.320
15000000	727616368.8	3811196.00	601180633.8	307314.700

VITA 2

Yuh-Ching Su

Candidate for the Degree of

Master of Science

Thesis: AN EMPIRICAL STUDY OF COMBSORT AND WAYS TO IMPROVE IT

Major Field: Computer Science

Biographical:

Personal Data: Born in Taiwan, R.O.C., June 19, 1956, The son of Mr. and Mrs. Ying-yuan Su.

Education: Received Bachelor degree in Business Administration from Tatung Institute of Technology, Taiwan, R.O.C. in May, 1980; Received Master of Business Administration degree from Oklahoma City University in May,1986; Completed requirements for the Master of Science degree at Oklahoma State University in July, 1993.

Professional Experience: Teaching Assistant, Oklahoma State University, Department of Computing and Information Sciences, Stillwater, Oklahoma, June, 1988 to August, 1988.
Network Administrator, Oklahoma State University, College of Arts and Sciences Extension, Stillwater, Oklahoma, June 1988 to January, 1991.