Analysis of Sorting Algorithms

Holger Findling

6/11/2013

Contents

Algorithmics 3
N^2 Sorting Algorithms 5
General Support Functions 5
Bubble Sort 6
Best Case Analysis 6
Worst Case Analysis 7
Performance 8
Selection Sort 9
Best Case Analysis 9
Worst Case Analysis 9
Performance 11
Recursive Selection Sort 12
Best Case Analysis 12
Worst Case Analysis 13
Performance 14
Insertion Sort 15
Best Case Analysis 15
Worst Case Analysis 15
Performance 17
N^2 Sorting Algorithm Comparison 18
Ascending Order 18
Descending Order 19
Random Order 20
Divide and Conquer 21
Solving Recursions using the Expansion Method 21
Quick Sort 24
Best Case Analysis 24
Worst Case Analysis 25
Performance 25
Partition Algorithm for Quick Sort 27
Mergesort 28
Best Case Analysis 28
Worst Case Analysis 28
Performance 29
Merge 30
Heap Sort 31

Algorithmics

Table 1 displays the relationship between N and functions F (n) = an + b, F (n) = n lg (n), F (n) = $an^2 + bn + c$, and F (n) = $an^3 + bn^2 + cn + d$.

N represents the number of integers to sort; *i.e.*, the size of the integer array. The tabularized results indicate the number of lines of code to be executed per function.

Figure 1 graphically displays the data shown in Table 1. The horizontal axis displays N and the vertical axis represents lines of code count (LOC). It should be noted that the vertical axis is configured as a function of log, base 2. Here we can see the drastic differences in LOC requirements for N^3 and N^2 type algorithms.

Table 1 N versus LOC

N	F(n) = an + b	F(n) = n lg(n)	F(n) = n*n	F(n) = n*n*n
10000	11000	132877.1238	100011000	1.33112E+12
20000	21000	285754.2476	400021000	9.26144E+12
30000	31000	446180.2464	900031000	2.9792E+13
40000	41000	611508.4952	1600041000	6.89227E+13
50000	51000	780482.0237	2500051000	1.32654E+14
60000	61000	952360.4928	3600061000	2.26985E+14
70000	71000	1126654.711	4900071000	3.57916E+14
80000	81000	1303016.99	6400081000	5.31448E+14
90000	91000	1481187.364	8100091000	7.53579E+14
100000	101000	1660964.047	10000101000	1.03031E+15

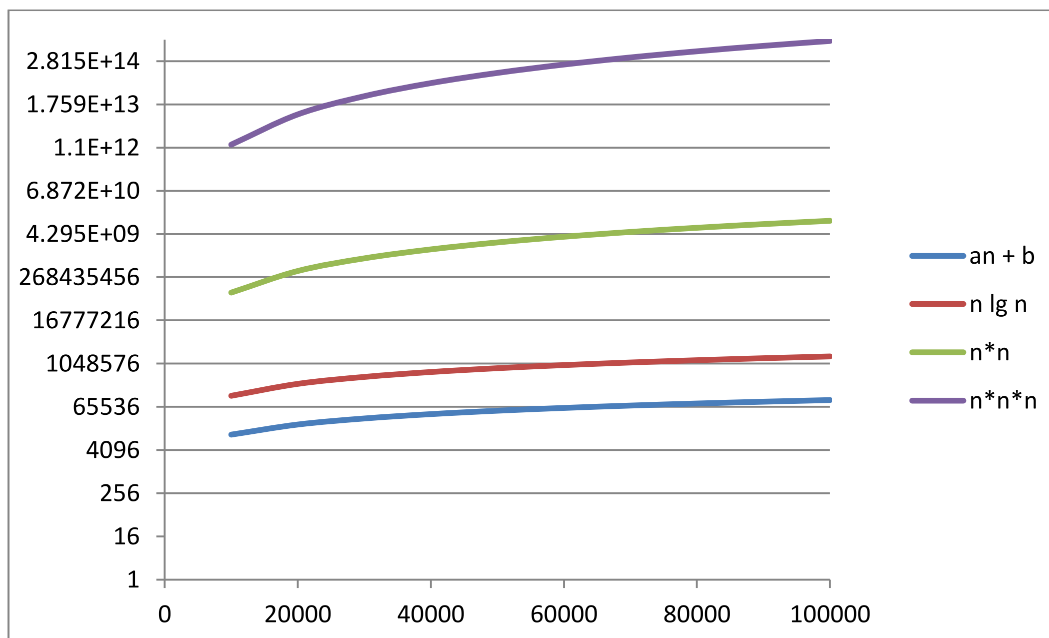


Figure 1 n versus n lg(n), n*n, n*n*n

When testing an algorithm one typically measures running times, which is not always easily compared to lines of code executed. However, the expected behavior should be the same regardless. The computer used for testing the sorting algorithms in this paper uses a dual core CPU, Intel, with a 2 GHz clock. The operating system is MS Windows 7, and 95 processes are running in the background. The average CPU usage varies between zero and four percent.

Establishing running times is an important aspect of algorithm development since it also aids in determining system throughput.

The results of a simple performance test, executing C++ code is shown below for two different processors. It is best to run a performance test directly on the target platform instead of trying to interpolate the performance.

Acer Inc.
AMD Turion ™ 64 X2 Mobile, Technology TL-50
1.61 GHz, 896 MB of RAM
Physical Address Extension

Start Time = Sat Dec 02 11:09:17.437
Stop Time = Sat Dec 02 11:09:19.93
Total Running time (ms) = 1656 per 780,060,004 lines of code
2.122914 nanoseconds per line of C++ code

Intel R
Celeron ® CPU
1.79 GHz, 256 MB of RAM

Start Time = Sat Dec 02 11:27:15.156
Stop Time = Sat Dec 02 11:27:17.15
Total Running time (ms) = 1859 per 780,060,004 lines of code
2.383150 nanoseconds per line of C++ code

N^2 Sorting Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort

General Support Functions

Swap

```
1    void CSort::Swap (int *pI, int *pN)
2    {
3        int temp = *pN;
5        *pN = *pI;
6        *pI = temp;
7        return;
9    }
```

Bubble Sort

```
1    void CSort::BubbleSort (int *pArray, long n)
2    {
3        int *pI
4        int *pL;
5        int *pN = pArray;
6        int *pEnd = pArray + n - 1;

7        for (pL = pArray; pL < pEnd; pL++) {
8            for (pI = pArray; pI < pEnd; pI++) {
9                pN = pI + 1;
10               if (*pI > *pN) {
11                   Swap(pI, pN);
12               }
13       }}

14       return;
15   }
```

Best Case Analysis

Line 3, 5 – Declarations are resolved at compile time, no computational cost incurred.

Line 5, 6 – 1 LOC each

Line 7 – 1 LOC, and 2 LOC each time the loop is repeated

Line 8 – 1 LOC, and 2 LOC each time the loop is repeated

Line 9 – 1 LOC each time the inside loop is executed

Line 10 – 1 LOC and the conditional statement is considered false

$$T(n) = \sum_{k=1}^{n} \left(3 + \sum_{i=1}^{n} 4 \right) + 3$$

$$T(n) = \sum_{k=1}^{n} (4n + 3) + 3$$

$$T(n) = n(4n + 3) + 3$$

$$T(n) = 4n^2 + 3n + 3$$

$$\mathbf{T(n) \in \Omega(n^2)}$$

Worst Case Analysis

Line 5, 6 – 1 LOC each

Line 7 – 1 LOC and 2 LOC each time the loop is repeated

Line 8 – 1 LOC and 2 LOC each time the loop is repeated

Line 9 – 1 LOC each time the inside loop is executed

Line 10 – 1 LOC and the conditional statement is considered false

Line 11 – 4 LOC

$$T(n) = \sum_{k=1}^{n} \left(3 + \sum_{i=1}^{n} 7\right) + 3$$

$$T(n) = \sum_{k=1}^{n} (7n + 3) + 3$$

$$T(n) = n(7n + 3) + 3$$

$$T(n) = 7n^2 + 3n + 3$$

$$\mathbf{T(n)\ \varepsilon\ O(n^2)}$$

Performance

$4n^2 + 3n + 3 <= T(n) <= 7n^2 + 3n + 3$

$T(n) \varepsilon \Theta (n^2)$

Table 2 Bubble Sort, N versus Time

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	481	1062	983
20000	1854	4138	3904
30000	4174	9317	8750
40000	7431	16575	15852
50000	11505	26082	24301
60000	16568	37267	35300
70000	22567	50693	47631
80000	29496	66261	62233
90000	37397	84099	78772
100000	46519	103694	98430

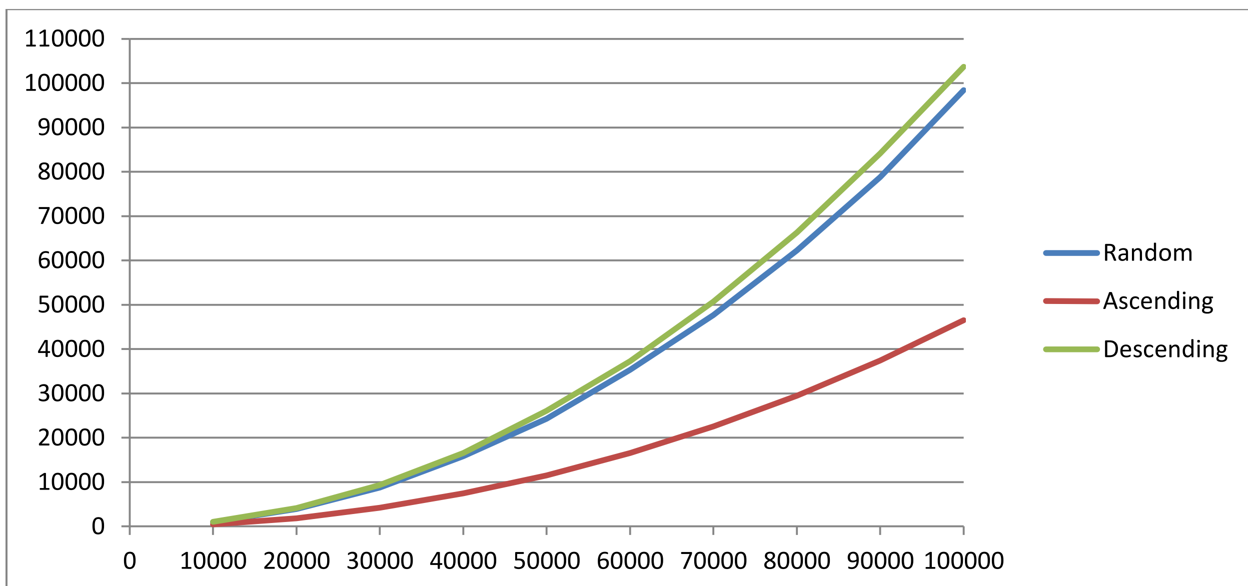


Figure 2 Bubble Sort Performance

Selection Sort

```
1    void CSort::SelectionSort(int *pArray, long n)
2    {
3        for (int *pLast = pArray + n - 1; pLast > pArray; pLast--) {
4            int *pMaxValue = pArray;

5            for (int *pIndex = pArray + 1; pIndex <= pLast; pIndex++) {
6                if (*pIndex > *pMaxValue) {
7                    pMaxValue = pIndex;
8                }
9            }
10           if (*pMaxValue > *pLast) {
11               Swap(pMaxValue, pLast);
12           }
13       }

14       return;
15   }
```

Best Case Analysis

Line 3 – 1 LOC and 2 LOC each time the loop is repeated

Line 4 – 1 LOC each time the outside loop is executed

Line 5 – 1 LOC and 2 LOC each time the loop is repeated

Line 6 – 1 LOC and the conditional statement is considered false

Line 10 – 1 LOC, the conditional statement is considered false

$$T(n) = \sum_{k=1}^{n} \left(5 + \sum_{i=1}^{n} 3\right) + 1$$

$$T(n) = \sum_{k=1}^{n} (3n + 5) + 1$$

$$T(n) = n(3n + 5) + 1$$

$$T(n) = 3n^2 + 5n + 1$$

$$\mathbf{T(n)\ \varepsilon\ \Omega(n^2)}$$

Worst Case Analysis

Line 3 – 1 LOC and 2 LOC each time the loop is repeated

Line 4 – 1 LOC each time the outside loop is executed

Line 5 – 1 LOC and 2 LOC each time the loop is repeated

Line 6 – 1 LOC and the conditional statement is considered true

Line 7 – 1 LOC

Line 10 – 4 LOC, the conditional statement is considered true

$$T(n) = \sum_{k=1}^{n} \left(8 + \sum_{i=1}^{n} 4\right) + 1$$

$$T(n) = \sum_{k=1}^{n} (4n + 8) + 1$$

$$T(n) = n(4n + 8) + 1$$

$$T(n) = 4n^2 + 8n + 1$$

$$\mathbf{T(n)\ \varepsilon\ O(n^2)}$$

Performance

$$3n^2 + 5n + 1 <= T(n) <= 4n^2 + 8n + 1$$

$$\mathbf{T(n) \ \varepsilon \ \Theta \ (n^2)}$$

Table 3 Selection Sort, N versus Time

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	241	221	203
20000	1969	884	809
30000	2180	1999	1809
40000	3888	3544	3200
50000	6084	5534	5025
60000	8727	7980	7260
70000	11895	10850	9836
80000	15582	14221	12841
90000	19644	17989	16272
100000	24239	22189	20138

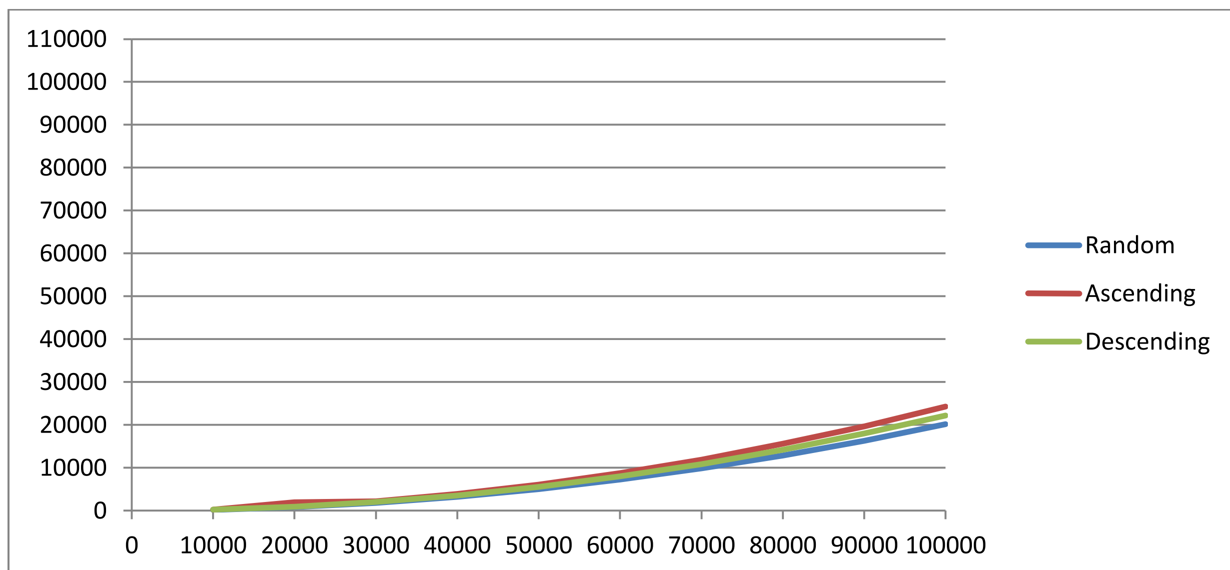


Figure 3 Selection Sort Performance

Recursive Selection Sort

```
1   void CSort::SelectionSort_R (int *pArray, long n)          T(n)
2   {
3         int *i, temp;
4         int *k = pArray + n - 1;                                1
5         int *key = pArray;                                      1

6         if (k == pArray)                                        1
7               return;

8         for (i = pArray; i <= k; i++) {                         2
9               if (*i > *key)                                    1
10                    key = i;                                    1
11        }

12        Swap(key, k);                                           3
13        SelectionSort_R (pArray, n-1);                          T (n-1)
14        return;
15  }
```

Best Case Analysis

Line 4, 5 – 1 LOC each

Line 6 – 1 LOC, it terminates the recursion

Line 8 – 1 LOC, and 2 LOC each time the loop is repeated

Line 9 – 1 LOC and the conditional statement is considered false

Line 12 – 3 LOC

Line 13 – Recursive Function Call, the array is decremented each time. T (n-1)

$$T(n) = T(n-1) + \sum_{i=1}^{n} 3 \quad + 6$$

$$T(n) = T(n-1) + 3n + 6 \quad \text{which is equivalent to } \sum_{i=0}^{n-1} (3n + 6)$$

$$T(n) \simeq 3n^2 + 6n$$

$$\mathbf{T(n)\ \varepsilon\ O(n^2)}$$

Worst Case Analysis

Line 4, 5 – 1 LOC each

Line 6 – 1 LOC, it terminates the recursion

Line 8 – 1 LOC, and 2 LOC each time the loop is repeated

Line 9 – 1 LOC and the conditional statement is considered true

Line 10 – 1 LOC

Line 12 – 3 LOC

Line 13 – Recursive Function Call, the array is decremented each time. T (n-1)

$$T(n) = T(n-1) + \sum_{i=1}^{n} 4 + 6$$

$$T(n) = T(n-1) + 4n + 6 \quad \text{which is equivalent to } \sum_{i=0}^{n-1} (4n+6)$$

$$T(n) \simeq 4n^2 + 6n$$

$$\mathbf{T(n) \in O(n^2)}$$

Performance

$3n^2 + 6n <= T(n) <= 4n^2 + 6n$

$T(n) \varepsilon \Theta (n^2)$

Table 4 Selection Sort Recursive, N versus Time

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	243	222	211
20000	1969	890	824
30000	2381	2011	1890
40000	3880	3563	3249
50000	6073	5555	5081
60000	8747	8031	7307
70000	11909	10922	9944
80000	15556	14229	12986
90000	19659	18070	16470
100000	24278	22228	20298

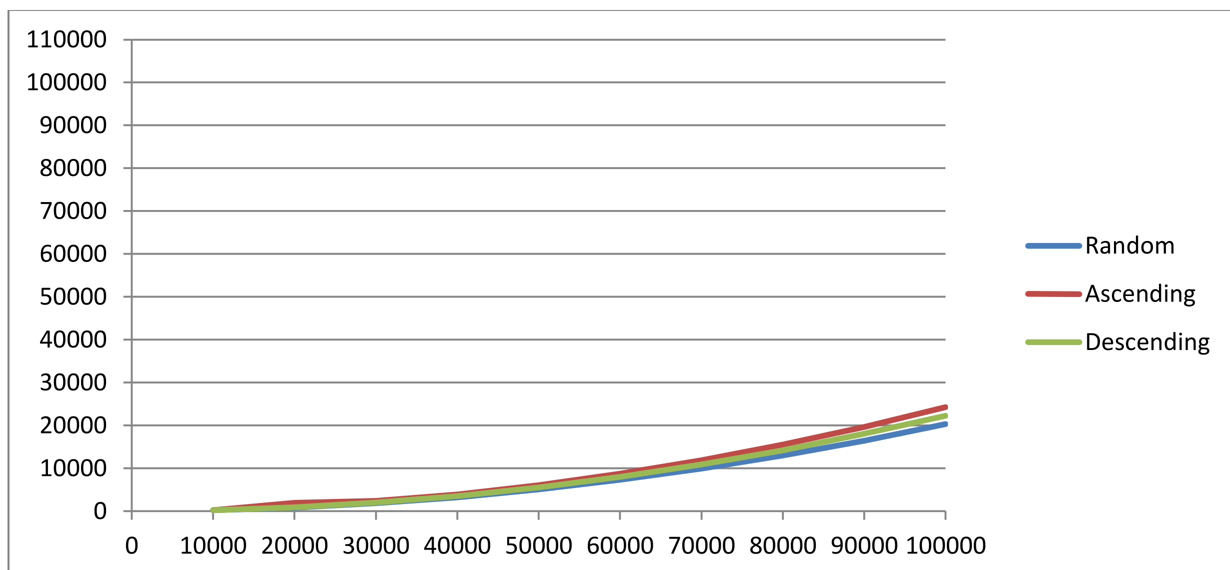


Figure 4 Selection Sort Recursive Performance

Insertion Sort

```
1     void CSort::InsertionSort (int *pArray, long n)
2     {
3         int *pSecond = pArray + 1;
4         int *pLast = pArray + n - 1;

5         while (pSecond <= pLast) {
6             int key = *pSecond;
7             int *pFirst = pSecond - 1;

8             while ((pFirst >= pArray) && (*pFirst > key)) {
9                 *(pFirst+1) = *pFirst;
10                pFirst--;
11            }

12            *(pFirst+1) = key;
13            pSecond++;
14        }

15        return;
16    }
```

Best Case Analysis

Line 3, 4 – 1 LOC each

Line 5 – 1 LOC each time the loop is executed

Line 6, 7, 12, 13 – 4 LOC each time the outside loop is executed

Line 8 – 2 LOC each time the loop is executed, the loop condition is not satisfied

$$T(n) = \sum_{i=1}^{n} 5 \quad + 2$$

$$T(n) = 5n + 2$$

$$\mathbf{T(n) \ \varepsilon \ \Omega(n)}$$

Worst Case Analysis

Line 3, 4 – 1 LOC each

Line 5 – 1 LOC each time the loop is executed

Line 6, 7, 12, 13 – 4 LOC each time the outside loop is executed

Line 8 – 2 LOC each time the loop is executed

Line 9, 10 – 2 LOC each time the inside loop is executed

$$T(n) = \sum_{i=1}^{n} \left(3 + \sum_{k=1}^{n} 4\right) + 2$$

$$T(n) = \sum_{i=1}^{n} (3 + 4n) + 2$$

$$T(n) = n(3 + 4n) + 2$$

$$T(n) = 4n^2 + 3n + 2$$

$\mathbf{T(n)\ \varepsilon\ O(n^2)}$

Performance

$5n + 2 <= T(n) <= 4n^2 + 3n + 2$

$T(n) \in O(n^2)$

Table 5 Insertion Sort, N versus Time

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	0	260	135
20000	0	1058	524
30000	1	2376	1176
40000	0	4217	2113
50000	1	6597	3290
60000	1	9465	4725
70000	0	12910	6456
80000	1	16953	8427
90000	1	21627	10675
100000	1	26382	13219

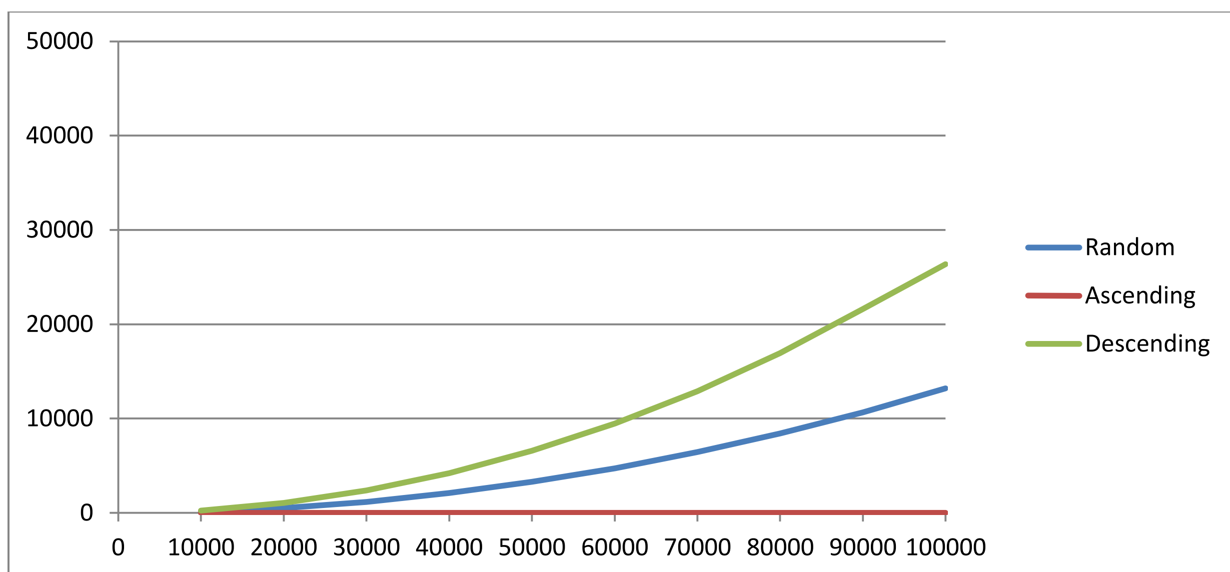


Figure 5 Insertion Sort Performance

N^2 Sorting Algorithm Comparison

- Bubble Sort
- Selection Sort
- Insertion Sort

Ascending Order

Table 6 Ascending Order

N	Bubble Sort (ms)	Selection Sort (ms)	Selection Sort R (ms)	Insertion Sort (ms)
10000	481	241	243	0
20000	1854	1969	1969	0
30000	4174	2180	2381	0
40000	7431	3888	3880	0
50000	11505	6084	6073	0
60000	16568	8727	8747	0
70000	22567	11895	11909	0
80000	29496	15582	15556	0
90000	37397	19644	19659	0
100000	46519	24239	24278	0

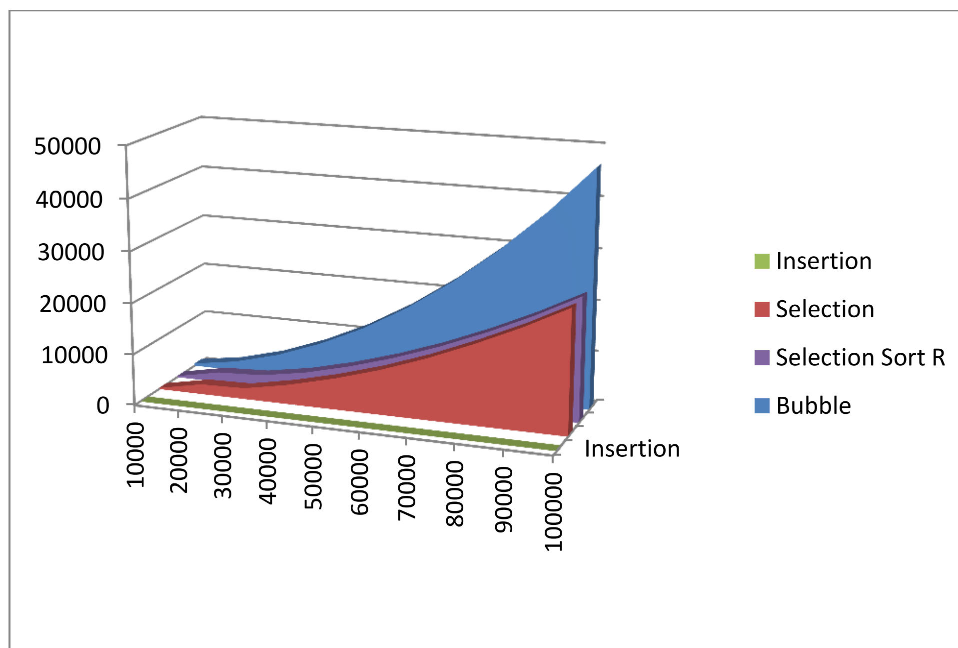


Figure 6 Ascending Order

Descending Order

Table 7 Descending Order

N	Bubble Sort (ms)	Selection Sort (ms)	Selection Sort R (ms)	Insertion Sort (ms)
10000	1062	221	222	260
20000	4138	884	890	1058
30000	9317	1999	2011	2376
40000	16575	3544	3563	4217
50000	26082	5534	5555	6597
60000	37267	7980	8031	9465
70000	50693	10850	10922	12910
80000	66261	14221	14229	16953
90000	84099	17989	18070	21627
100000	103694	22189	22228	26382

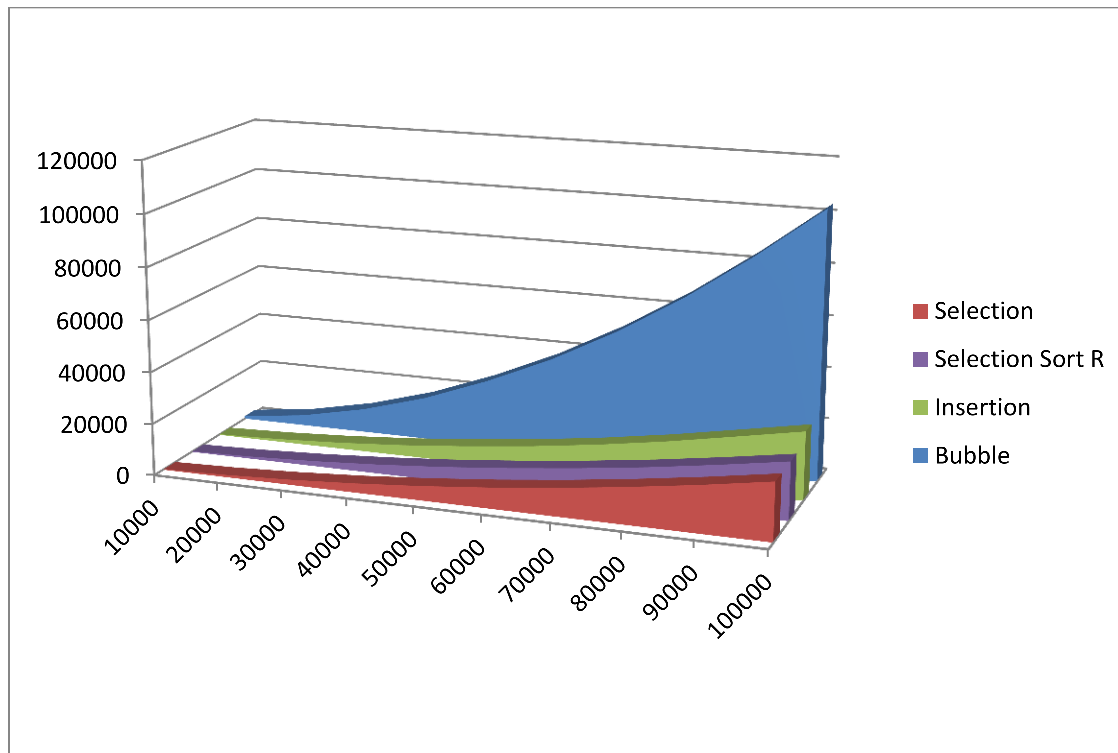


Figure 7 Descending Order

Random Order

Table 8 Random Order

N	Bubble Sort (ms)	Selection Sort (ms)	Selection Sort R (ms)	Insertion Sort (ms)
10000	983	203	211	135
20000	3904	809	824	524
30000	8750	1809	1890	1176
40000	15852	3200	3249	2113
50000	24301	5025	5081	3290
60000	35300	7260	7307	4725
70000	47631	9836	9944	6456
80000	62233	12841	12986	8427
90000	78772	16272	16470	10675
100000	98430	20138	20298	13219

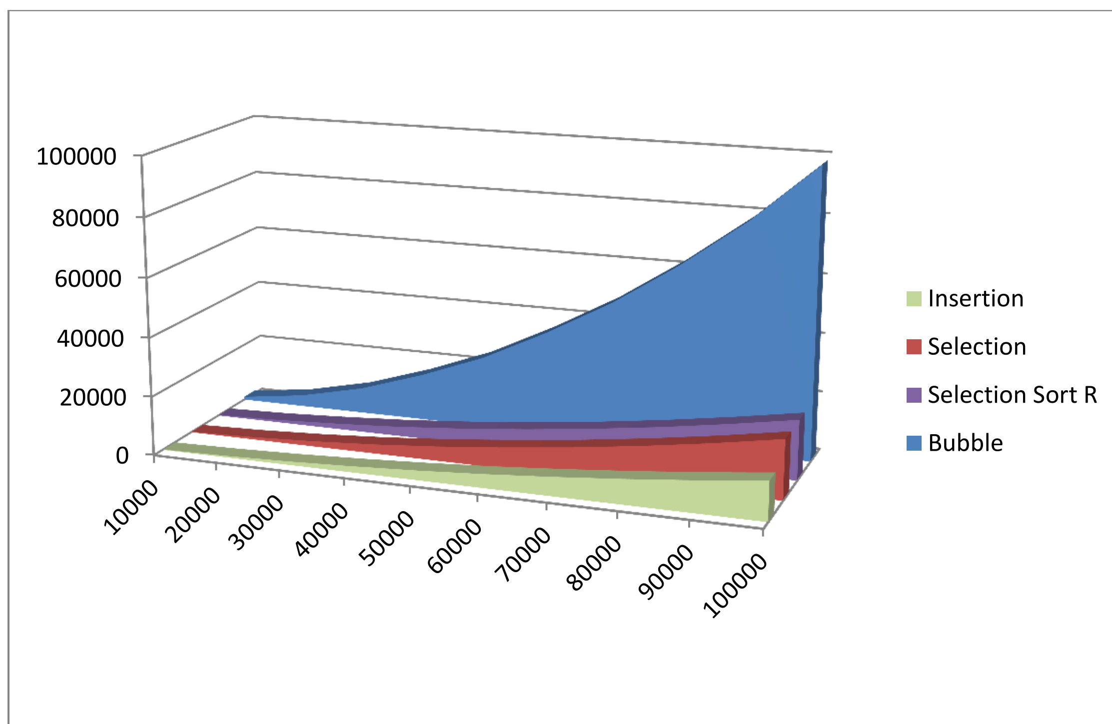


Figure 8 Random Order

Divide and Conquer

Solving Recursions using the Expansion Method

Let's take a look at three general divide and conquer recurrence equations.

Solve the recursion: $T(n) = 2\,T(n / 2) + n$ where $T(1) = 1$

Let's inspect the recursion and calculate the first six terms.

$T(1) = 1$

$T(2) = 2\,T(2 / 2) + 2 = 2$

$T(4) = 2\,T(4 / 2) + 4 = 12$

...

Table 9 Divide and Conquer

N	1	2	4	8	16	32
T(n)	1	4	12	32	80	192

Solution by Expansion method

$$T(n) = n + 2\,T(n / 2)$$

Expand the recurrence equation until a pattern is detected.

$$T(n) = n + 2\,[\,n / 2 + 2T(n / 4)\,]$$

$$T(n) = n + 2\,[\,n / 2 + 2\,[\,n / 4 + 2\,T(n / 8)\,]]$$

$$T(n) = n + n + n + 2^3\,T(n / 2^3)$$

$$T(n) = 3\,n + 2^3\,T(n / 2^3)$$

$$T(n) = 4\,n + 2^4\,T(n / 2^4)$$

$$T(n) = 5\,n + 2^5\,T(n / 2^5)$$

...

$$\mathbf{T(n) = p\,n + 2^p\,T(n / 2^p)} \qquad \textbf{Equation \#1}$$

Expand the recursion until $T(n / 2^p) = T(1)$. Solve for p.

$n / 2^p = 1$

$2^p = n$

$\log 2^p = \log n$

$p \log 2 = \log n$

$p = \log n / \log 2$

$p = \lg (n)$ **Equation #2**

Substitute Equ. #2 into Equ. #1

$T (n) = n \lg (n) + 2^{\lg (n)} T (1)$

$T (n) = n \lg (n) + 2^{\lg (n)}$

$O (n) = n \lg (n)$

Verify:

$T (32) = 32 \lg (32) + 2^{\lg (32)}$

$T (32) = 32 (5) + 32 = 192.$

Solve the recursion: $T (n) = 2 T (n / 2) + an$

$T (n) = an + 2 T (n / 2)$

$T (n) = an + 2 [an / 2 + 2T (n / 4)]$

$T (n) = an + 2 [an / 2 + 2 [an / 4 + 2 T (n / 8)]]$

$T (n) = an + an + an + 2^3 T (n / 2^3)$

$T (n) = pan + 2^p T (n / 2^p)$ where $p = \lg (n)$

$T (n) = an \lg (n) + 2^{\lg (n)} T (1)$

$O (n) = an \lg (n)$

Solve the recursion: $T (n) = 2 T (n / 2) + an + b$

$T(n) = b + an + 2\,T(n/2)$

$T(n) = b + an + 2\,[b + an/2 + 2T(n/4)]$

$T(n) = b + an + 2\,[b + an/2 + 2\,[b + an/4 + 2\,T(n/8)]]$

$T(n) = b + an + 2b + an + 4b + an + 2^3\,T(n/2^3)$

$T(n) = 3an + b + 2b + 4b + 2^3\,T(n/2^3)$

$T(n) = 4an + b + 2b + 4b + 8\,b + 2^4\,T(n/2^4)$

$T(n) = pan + 2^{p-1}\,b + 2^p\,T(n/2^p)$

Stop the expansion when $(n/2^p) = 1$, then $p = \lg(n)$

$T(n) = an\lg(n) + 2^{\lg(n)} + \underline{(2^{p-1} + 2^{p-2} + \ldots + 2 + 1)\,b}$

Geometric Series

We know that the term *an lg (n)* is dominant over the term $2^{\lg(n)}$; however we need to evaluate the geometric series.

Solve the Geometric Series:

$$\sum_{k=1}^{p-1} b_k\,2^k = b/(1-r) - (b\,2^p)/(1-r)$$

r = 2, therefore

$$\sum_{k=1}^{p-1} b_k\,2^k = b/(1-2) - (b\,2^p)/(1-2)$$

$$\sum_{k=1}^{p-1} b^k\,2^k = b\,(2^p - 1)$$ **But** $2^p = n$

$$\sum_{k=1}^{p-1} b^k\,2^k = b\,(n - 1)$$

Substitute:

$T(n) = an\lg(n) + 2^{\lg(n)} + (2^{p-1} + 2^{p-2} + \ldots + 2 + 1)\,b$

$T(n) = an\lg(n) + 2^{\lg(n)} + (n-1)\,b$

O (n) = n lg (n)

Quick Sort

The Quick Sort algorithm is comprised of two components, partitioning the array and then processing each partition separately.

```
void QuickSort (double *pA, long A0, long An)
{
    long q;
    if (A0 < An) {                                   1
        q = Partition (pA, A0, An);                  Theta (n)
        QuickSort (pA, A0, q);                       T (n) = T (n/2)
        QuickSort (pA, q + 1, An);                   T (n) = T (n/2)
    }
    return;
}
```

T (n) = T (n/2) + T (n/2) + Theta (n) + 1

T (n) = 2 T (n/2) + Theta (n) + 1

Best Case Analysis

The best case occurs when the content of the array is in random order.

The partition routine produces two equal sub-arrays each time.

$T(n) = T(n/2) + T(n/2) + \Theta(n)$

$T(n) = 2T(n/2) + n + c$ --- Same format as the general case.

$T(n) = 2T(n/2) + a\,n + b$ --- General Case

Therefore:

$T(n) = n \lg(n)$

$T(n) \in \Omega(n \lg(n))$

Worst Case Analysis

The worst case occurs when the array is already sorted.

The partition routine produces one region with n – 1 elements and the other region with only 1 element.

$$T(n) = T(n-1) + \Theta(n) + c$$

$$T(n) = \sum_{k=1}^{n} \Theta(n) + c \qquad \text{(Partition n times)}$$

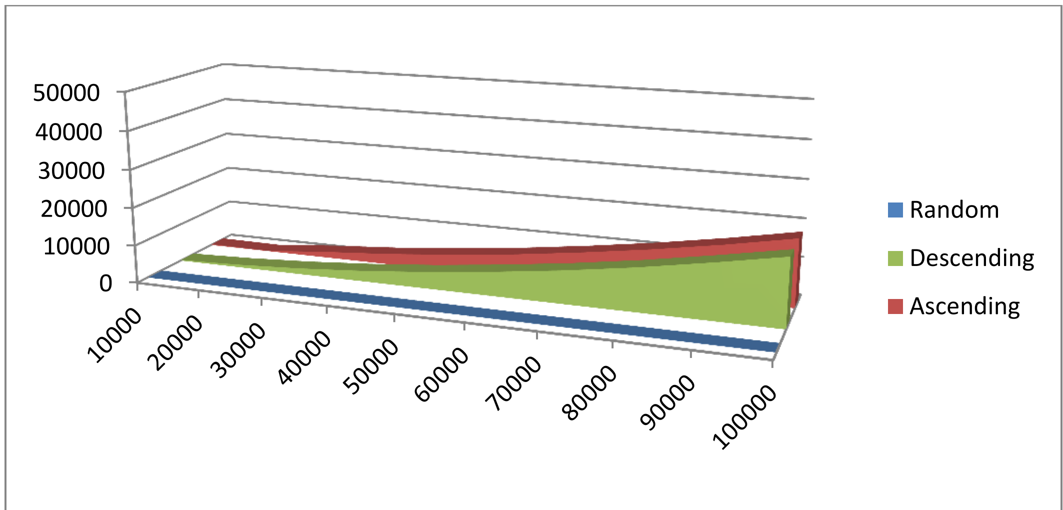
$$T(n) = O(n^2 + c\,n)$$

$$\mathbf{T(n) = O(n^2)}$$

Performance

$$\mathbf{\Omega(n \lg(n)) <= T(n) <= O(n^2)}$$

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	196	180	5
20000	72	714	4
30000	1610	1604	7
40000	2853	2847	9
50000	4483	4449	12
60000	6425	6443	16
70000	8780	8726	17
80000	11424	11408	20
90000	14447	14502	23
100000	17838	17827	25


50000
40000
30000
20000
10000
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Random
Descending
Ascending

Partition Algorithm for Quick Sort

```
long Partition (double *pA, long A0, long An)
{
    long i = A0  - 1;                   1
    long j = An + 1;                    1
    double temp;
    double x = pA[A0];                  1

    while (TRUE) {                      This loop will be performed at most n times
        while (TRUE)  {                 This loop will be performed less than n times
            if (j != 0)   --j;          2
            else
                break;                  1
            if (pA[j] <= x)
                break;                  2
        }

        while (TRUE)  {                 This loop will be performed less than n times
            if (i != An)
                i++;                    2
            else
                break;                  1

            if (pA[i] >= x)
                break;                  2
        }

        if (i < j)  {                   1
            Swap();                     3
        }
        else
            return j;                   1
    }
}
```

T (n) $\simeq$ 15 n + 3

T (n) ε Theta (n)

Mergesort

The Mergesort algorithm was developed by John von Neumann and is one of the first sorting algorithms used on a computer. Mergesort is comprised of two components, partitioning the array into halves, and then merging the sorted halves into one array.

```
1   void CSort::MergeSort(int* pArray, int low, int high)        T(n)
    {
2       int mid;
3       if (low < high) {                                         1
4           mid = ((high + low) / 2);                             1
5           MergeSort (pArray, low, mid);                         T (n/2)
6           MergeSort (pArray, mid+1, high);                      T (n/2)
7           Merge (pArray, low, mid, high);                       Theta (n)
        }
        return;
    }
```

Best Case Analysis

$T(n) = T(n/2) + T(n/2) + \Theta(n) + 2$

$T(n) = 2T(n/2) + \Theta(n) + 2$ --- Same format as the general case.

$T(n) = 2T(n/2) + n + c$ --- General case.

Therefore:

$T(n) = n \lg(n)$

$\mathbf{T(n) \in \Omega(n \lg(n))}$

Worst Case Analysis

$T(n) = T(n/2) + T(n/2) + \Theta(n) + 2$

$T(n) = 2T(n/2) + \Theta(n) + 2$ --- Same format as the general case.

$\mathbf{T(n) \in O(n \lg(n))}$

Performance

$$\Omega \ (n \lg (n)) <= T(n) <= O(n^2)$$

Therefore

$$T(n) \in \Theta \ (n \lg (n))$$

Table 10 Mergesort, N versus Time

N	Ascending (ms)	Descending (ms)	Random (ms)
10000	46	46	49
20000	175	179	178
30000	390	383	393
40000	666	673	669
50000	1035	1023	1036
60000	1467	1457	1476
70000	1987	2008	1992
80000	2554	2550	2563
90000	3215	3225	3231
100000	3990	3964	3989

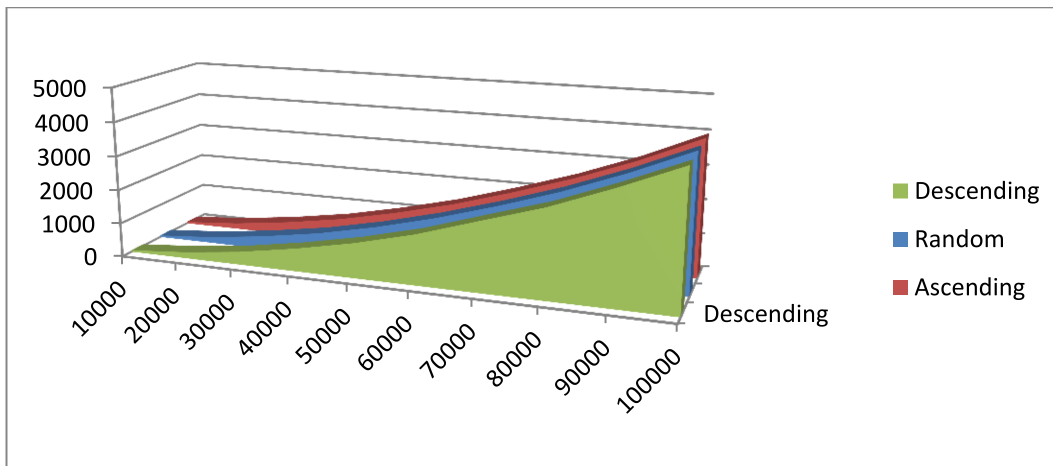


Figure 9 Mergesort Performance

Merge

```
void CSort::Merge (int* pArray, int low, int mid, int high)
{
    int i = low;
    int j = mid + 1;
    int k = 0;
    int* pB = new int[high+1];

    while (i <= mid && j <= high) {
        if (pArray[i] <= pArray[j])
            pB[k++] = pArray[i++];
        else
            pB[k++] = pArray[j++];
    }

    while (i <= mid) {
        pB[k++] = pArray[i++];
    }

    while (j <= high) {
        pB[k++] = pArray[j++];
    }

    k--;
    while (k >= 0) {
        pArray[low + k] = pB[k];
        k--;
    }

    delete pB;
}
```

Heap Sort