Sorting Methods for Small Arrays

Renato F. Werneck
rwerneck@inf.puc-rio.br

Celso C. Ribeiro
celso@inf.puc-rio.br

PUC-Rio.Inf.MCC 23/00
July 2000

Abstract We present and compare four efficient quadratic, comparison-based algorithms for small arrays and (for three of them) almost sorted inputs. In addition to the well-known insertion sort and selection sort, we analyze 2-insertion sort (a variation of insertion sort) and stacksort (based on selection sort). We show that the new algorithms perform fewer comparisons on average than their original versions. The theoretical analysis is confirmed by experimental data, which include promising results concerning the use of 2-insertion sort in conjunction with quicksort.

Keywords: Sorting; Algorithms; Insertion Sort; 2-Insertion Sort; Stacksort.

Resumo Esta monografia apresenta e compara quatro algoritmos quadráticos baseados em comparações que são eficientes para o tratamento de vetores pequenos e (no caso de três dos métodos) para entradas quase ordenadas. Além dos conhecidos *insertion sort* e *selection sort*, também são analisados os algoritmos *2-insertion sort* (uma variação do *insertion sort*) e *stacksort* (baseado no *selection sort*). Demonstra-se que os novos algoritmos propostos executam menos comparações em média que as respectivas versões originais. A análise teórica é confirmada por dados experimentais, que incluem resultados promissores sobre o uso do algoritmo *2-insertion sort* com o *quicksort*.

Palavras-chave: Ordenação; Algoritmos; Insertion Sort; 2-Insertion Sort; Stacksort.

1 Introduction

The most generic sorting algorithms are those based solely on comparisons between the elements, without assuming anything about how the keys are represented or about the distribution of the input data. The simplest algorithms of that kind are quadratic: this includes selection sort, insertion sort and bubble sort, to mention a few. Most elaborate algorithms, such as mergesort and heapsort, have $O(n \log n)$ worst case running time, which is optimal.

Optimal algorithms are undoubtedly the best choice for sorting large arrays, but they are often slower than quadratic algorithms for small instances. This is so because the constants and lower-order terms hidden in the O notation are usually higher for more complex algorithms. This fact has already been noticed and sometimes is used to speed up the optimal algorithms. Sedgewick [2], for instance, suggests that insertion sort is used to process the smaller subproblems generated by quicksort.

This work analyzes and compares four of the fastest quadratic comparison-based sorting algorithms. Section 2 briefly discusses the performance of two very well-known algorithms, selection sort and insertion sort. Then we present two variations of these algorithms: stacksort (section 3) and 2-insertion sort (section 4). Section 5 summarizes the theoretical results and compares the algorithms. In section 6, we use experimental data (including tests with quicksort) to extend the comparative evaluation. Finally, section 7 presents concluding remarks about this work and suggests some extensions.

Notation and Conventions Let $v[1], v[2], \ldots, v[n]$ be the input array to be sorted in nondecreasing order. Position 0 is reserved for a sentinel, whenever needed.

The algorithms are evaluated primarily by the number of comparisons (C) they perform. Since for most algorithms this value depends not only on the number of elements (n), but also on their original order, we evaluate the best (C_b), average (C_a) and worst (C_w) cases whenever necessary.

2 Standard Algorithms

2.1 Selection Sort

Selection sort is one of the simplest sorting algorithms. In every iteration i, a linear search determines the position of the largest element of the unsorted part of the array, which is swapped with the current element in the position it should occupy in the sorted array, $v[n-i+1]$. Figure 1 shows the pseudocode for the algorithm.

Unlike other sorting algorithms, the number of comparisons made by selection sort does not depend on the original ordering of the input. There are $n-1$ iterations (the last element is determined automatically). At iteration i, all but one of the elements of the unsorted part of the array are compared to another element (the current largest in this iteration). Thus, the total number of comparisons (C) is:

$$C = \sum_{i=1}^{n-1} (n-i) = \frac{n^2}{2} - \frac{n}{2}. \tag{1}$$

2.2 Insertion Sort

Another efficient algorithm for small instances is insertion sort. It also consists of $n-1$ iterations. In the beginning of iteration i ($1 \leq i \leq n-1$), the first i elements are already sorted and one has to find the correct position for the element currently in $v[i+1]$. Since we are interested in small instances, we consider the implementation of insertion sort using right-to-left linear search in each iteration. For large instances, binary search would be a better choice. The easiest way to guarantee

```
procedure selection(left, right) {
   for i = left to right − 1 do {
       maxpos ← i;
       maxval ← v[i];
       for j = i + 1 to right do {
           if (v[j] ≥ maxval) then {
               maxpos ← j;
               maxval ← v[maxpos];
           }
       }
       v[maxpos] ← v[i];
       v[i] ← maxval;
   }
}
```

Figure 1: Selection Sort

that the linear search will never go beyond the limits of the array is to insert a *sentinel* in position 0, i.e., we must choose $v[0]$ such that $v[0] \leq v[j], \forall j \in \{1 \ldots n\}$. Figure 2 presents the pseudocode for insertion sort.

```
procedure insertion(left,right) {
   for i = left + 1 to right do {
       tmp ← v[i];
       j ← i − 1;
       while (v[j] > tmp) do {
           v[j + 1] ← v[j];
           j ← j − 1;
       }
       v[j + 1] ← tmp;
   }
}
```

Figure 2: Insertion Sort

In the best case (sorted inputs), each iteration requires only one comparison to determine that $v[i+1]$ is already in its correct position. Summing over all iterations, we have:

$$C_b = \sum_{i=1}^{n-1} 1 = n - 1.$$

For reversely sorted input arrays, $i + 1$ comparisons will be necessary in iteration i, since the new element will always be inserted into the first position of the sorted portion of the array (right after the sentinel). Considering the entire execution of the algorithm, the total number of comparisons in the worst case is:

$$C_w = \sum_{i=1}^{n-1} (i + 1) = \frac{n^2}{2} + \frac{n}{2} - 1.$$

For random inputs, the expected number of comparisons in iteration i is the arithmetic mean of the best and the worst cases: (i+2)/2. New elements are expected to be inserted exactly in the middle of the sorted part of the array. Thus, the average number of comparisons performed by insertion sort is:

$$C_a = \sum_{i=1}^{n-1} \frac{s+2}{2} = \frac{n^2}{4} + \frac{3n}{4} - 1. \tag{2}$$

3 Stacksort

Stacksort is a selection-based algorithm, in the sense that in each iteration it selects an element to be inserted in its final position. The algorithm is very similar to selection sort: in the first iteration, the largest element is determined and exchanged with the last element of the array. In the second iteration, the same is done to the second largest element. In general, the i-th largest element is determined in the i-th iteration,as in selection sort. However, unlike selection sort, stacksort is capable of using information gathered in previous iterations to speed up later ones.

3.1 Inefficiency of Selection Sort

Consider the execution of selection sort. In every iteration, we scan the array from left to right to determine the position of the largest element not yet processed. This is done by keeping only the position of the *local maximum*, the largest element of the already scanned portion of the array. Let $maxpos$ be such position. We start by setting $maxpos \leftarrow 1$. During the scan, whenever we find an element $v[j]$ greater than $maxpos$, we make it the new local maximum ($maxpos \leftarrow j$).

This strategy has a major source of inefficiency. Suppose the largest element found in iteration i is in position p. Then, by the time iteration $i+1$ reaches position p, it will have performed exactly the same comparisons as iteration i, since elements in positions 1 to $p-1$ remain unchanged between them. Moreover, the same local maxima will be found and all of them, with the possible exception of one (the last), will eventually be discarded, as in the previous iterations.

3.2 The Algorithm

The motivation for stacksort is to avoid such redundant, useless comparisons. It does so by keeping in a stack the positions of all local maxima found since the begining of the algorithm (and not yet inserted into their final positions).

We start the algorithm with "1" in the stack, since $v[1]$ is the first local maximum. The element at the top of the stack is exactly the current value of $maxpos$. In the first iteration, whenever we find an element $v[p]$ greater than the $v[maxpos]$, we push p onto the stack (thus making $maxpos \leftarrow p$). By the end of this iteration, the stack will contain all local maxima found. Furthermore, the value at the top ($maxpos$) will be the position of the largest element in the array. As in selection sort, we swap $v[n]$ (the element in the last position) and $v[maxpos]$, and then remove $maxpos$ from the stack.

So far, the comparisons required by stacksort are exactly the ones that would be performed by selection sort for the same input. The difference between the algorithms appears in the iterations that follow. Selection sort would start scanning from position 1 again. Stacksort, on the other hand, can use the information gathered in previous iterations to start in a further position.

Consider the scenario discussed for selection sort: assume that the largest element found in a given iteration i is in position p. In stacksort, this means that, after scanning the array, p became the position at the top of the stack. Let q be the element (position) at the top after removal of p. We know that q was inserted into the stack before p.Since both p and q are positions of local maxima, $q < p$ and $v[q] \leq v[p]$.

Since q is a local maximum, there is no point in searching for the maximum in positions 1 to $q-1$, because they cannot contain elements greater than $v[q]$. Furthermore, since q became the element at the top of the stack immediately after the removal of p, we can guarantee that no element between positions q and p is greater than $v[q]$. If there were such element, it would have been pushed onto the stack after q. Therefore, we can also skip positions $q+1$ to $p-1$.

This is the reason why stacksort can be more efficient than selection sort. In general, there is no need to scan the entire unsorted portion of the array in every iteration. Instead of starting the search for the maximum in the first position, as in selection sort, we can start in p, the position of the largest element found in the previous iteration.

Notice that, even though stacksort does not avoid scans, it makes them shorter. A scanning procedure similar to the one described for the first iteration must be carried out in the others. In a given iteration i, the first candidate (*maxpos*) is the element at the top of the stack (q). Whenever an element greater than $v[maxpos]$ is found, we push it onto the stack and update *maxpos*. When we reach the end of the unsorted part of the array, we swap $v[maxpos]$ and $v[n-s+1]$ and pop the first element of the stack.

Implementation Issues One aspect that must be addressed when implementing stacksort is the fact that whenever the maximum of a given iteration i happens to be in the first position of the array, the stack will be empty in the beginning of the next iteration. In this case, we cannot get the first value of *maxpos* from the stack.

One way to deal with this would be to test if the stack is empty in the beginning of every iteration and, if so, set $maxpos \leftarrow 1$ and start scanning from the second position. Note that only one extra test will be required in each of the $n-1$ iterations. This is far better than the $O(n^2)$ extra tests required in versions of insertion sort that do not resort to sentinels.

But there is a better solution. If we can guaratee that $v[0]$ is not larger than any element in positions 1 to n, it can be used as a sentinel. Before starting the main loop of the algorithm, we insert 0 into the stack, thus making $v[0]$ the first local maximum. This will guarantee that the stack will never be empty. Although $v[0]$ may be at the top of the stack in some iteration (even more than one), it will be immediately replaced by $v[1]$, since $v[1]$ will always be a local maximum (by definition, it is not smaller than the sentinel).

The pseudocode for stacksort presented in figure 3 uses the sentinel. Note that, since it is a generalized version of the algorithm that sorts the array from positions *left* to *right*, the sentinel must be $v[left-1]$. One must also be aware that *maxpos* is used in a slightly different way than we have discussed. It does not contain a copy of the element at the top of the stack, but must be interpreted as a complement to the stack. Whenever *maxpos* is updated during the scan, its old value is pushed onto the stack; whenever a position is removed from the stack, it is held in *maxpos*.

3.3 Analysis

The first iteration requires a full scan of the array. We begin with the sentinel as the local maximum and update the value *maxpos* whenever larger elements are found. Since every element must be compared to some local maximum, the first iteration requires exactly n comparisons.

The number of comparisons performed in any subsequent iteration i depends on the position of the maximum found in iteration $i-1$. Let such element be $b(i-1)$. Every element from position $b(i-1)$ to $n-i+1$ (the last one of the unsorted part of the array) must be compared to the local maximum. Hence, each iteration requires $n-i+1-b(i-1)+1$ comparisons. Summing over the remaining iterations (2 to $n-1$), we have:

$$C = n + \sum_{i=2}^{n-1}[n-i-b(i-1)+2] = \frac{n^2}{2} + \frac{3n}{2} - 3 - \sum_{s=1}^{n-2} b(i). \tag{3}$$

The best case of stacksort occurs when the input data is already sorted. In this case, $b(i) = n-i+1$

```
procedure stacksort(left,right) {
    stack.push(left−1);
    lastpos ← left;
    for i = right downto left + 1 do {
        maxpos ← stack.pop();
        maxval ← v[maxpos];
        for j = lastpos to i do {
            if (v[j] ≥ maxval) then {
                stack.push(maxpos);
                maxpos ← j;
                maxval ← v[maxpos];
            }
        }
        v[maxpos] ← v[i];
        v[i] ← maxval;
        lastpos ← maxpos;
    }
}
```

Figure 3: Stacksort

and

$$C_b = \frac{n^2}{2} + \frac{3n}{2} - 3 - \sum_{i=1}^{n-2}(n-i+1) = n. \tag{4}$$

Note that comparisons between elements of the array take place only in the first iteration. In the following ones, a single comparison between array indices (not considered in equation 3) is enough to determine that the element is already in its final position.

We now consider the average case of the algorithm. For random input data, the maximum is expected to be exactly in the middle of the unsorted part of the array. Hence, the expected value of $b(i)$ is $\frac{1}{2}(n-i)+1$ (this expression takes into account the fact that the first element is in $v[1]$, not $v[0]$) and the average number of comparisons performed by stacksort is

$$C_a = \frac{n^2}{2} + \frac{3n}{2} - 3 - \sum_{i=1}^{n-2}\left(\frac{n-i}{2}+1\right) = \frac{n^2}{4} + \frac{3n}{4} - \frac{1}{2}. \tag{5}$$

In the worst case, the algorithm behaves as selection sort. The entire unsorted part of the array must be scanned in every iteration, which means that $b(i) = 1$ and

$$C_w = \frac{n^2}{2} + \frac{3n}{2} - 3 - \sum_{i=1}^{n-2} 1 = \frac{n^2}{2} + \frac{n}{2} - 1. \tag{6}$$

We note that the worst case of the algorithm does *not* correspond to reversely sorted input arrays. In this case, each of the first $\lfloor n/2 \rfloor$ iterations reduces the portion of the array that needs to be scanned by two units and the remaining iterations are executed in constant time. The actual worst case of the algorithm occurs when the first element of the input is the maximum and the rest of the array is already sorted.

The Stack Comparing the results obtained for stacksort (equations 4, 5 and 6) to selection sort (equation 1), we notice that stacksort has better performance for virtually every instance. However, it relies on extra memory for the stack, whose size depends on the maximum number of local maxima found during the execution of the algorithm. It can be as low as one (for "almost sorted" inputs, the worst case of the algorithm), but as high as n (for sorted arrays). Not surprisingly, better running times require larger stacks.

Knuth [1] shows that the expected number of local maxima in random arrays with k elements is $H(k) = 1 + 1/2 + 1/3 + \ldots + 1/k$. Therefore, in any given iteration i of the algorithm, the expected size of the stack will be $H(n-i+1) = O(\log n)$, which is much smaller then the $O(n)$ worst case.

An interesting characteristic of stacksort is that it requires only $\Theta(n)$ stack operations, regardless of the input array or the size of the stack. As figure 3 shows, only one *pop* operation is performed in each of the $n-1$ iterations. Since the stack starts and ends the execution with only one element (the sentinel), the number of *push* operations is also limited to $n-1$ (or to n, considering the initial insertion of the sentinel).

4 2-Insertion Sort

2-insertion sort is a simple extension of insertion sort: instead of inserting one element at a time in its final position, 2-insertion sort inserts two elements. In each iteration i, the first two elements of the unsorted part of the array (e_i and $e_i + 1$) are compared; let *max* be the larger of them and let *min* be the smaller (note that only one comparison is needed to determine which one is *max* and which is *min*).

We scan the array from right to left, i.e., from $e_i - 1$ to 0 (as in insertion sort, we assume that there is a sentinel in $v[0]$ such that $v[0] \leq v[j], \forall j \in \{1, 2, \ldots n\}$). For each position j in this interval, we first compare $v[j]$ with *max*. If $v[j]$ is greater, there is no need to compare it with *min*, since by definition $min \leq max$. All we have to do is to transfer the element from position j to $j+2$ (i.e., $v[j+2] \leftarrow v[j]$) in order to open some room for both *max* and *min* at the sorted part of the array.

There will be some index j for which $v[j] \leq max$ (in the worst case, it will be 0, the sentinel). When this happens, we simply set $v[j+2] \leftarrow max$. Then we procede to the search for the correct position of *min* exactly as in the original insertion sort, but starting at the j-th position of the array.

Since in each iteration of 2-insertion sort two elements are inserted, only $\lfloor n/2 \rfloor$ iterations are necessary to sort the array. When the number of elements is even, the algorithms begins with $e_1 = 1$; when it is odd, it begins with $e_1 = 2$. This guarantees that there will be two elements to process in every iteration, including the last one.

4.1 Analysis

We already know that 2-insertion sort performs $\lfloor n/2 \rfloor$ iterations. In order to determine its average case behavior, we must calculate the number of comparisons comparisons performed in each of these iterations.

In a given iteration i, elements in positions e_i and $e_i + 1$ are processed. The value of e_i depends on the parity of n: $e_i = 2i - 1 + n\backslash 2, \forall s \geq 1$ ($n\backslash 2$ denotes the remainder of the division of n by 2). One comparison determines *min* and *max*. Let r_i be the position with the highest index in the sorted part of the array that contains an element not greater than *min*. Clearly, either $r_i = e_i - 1$ or $v[r_i] \leq min < v[r_i + 1]$. Each element from $e_i - 1$ down to r_i will be compared to *min*, *max* or both. More precisely, one of them ($v[j]$) will be compared to both *max* and *min*; the other $e_i - r_i - 1$ elements will be used in only one comparison, either with *max* (for those greater than $v[j]$) or with *min* (for those smaller than $v[j]$). Thus, the overall number of comparisons in iteration i (c_i) is

$$c_i = 1 + (e_i - r_i - 1) + 2 = e_i - r_i + 2. \tag{7}$$

Summing over all iterations, we can determine the number of comparisons performed along the entire execution of the algorithm:

```
procedure insertion2 (left, right) {
   for i = left + (right − left + 1) mod 2 to right do {
      if (v[i] ≤ v[i + 1]) then {min ← v[i]; max ← v[i + 1]};
      else {max ← v[i]; min ← v[i + 1]};
      j ← i − 1;
      while (v[j] > max) do {
         v[j + 2] ← v[j];
         j ← j − 1;
      }
      v[j + 2] ← max;
      while (v[j] > min) do {
         v[j + 1] ← v[j];
         j ← j − 1;
      }
      v[j + 1] ← min;
   }
}
```

Figure 4: 2-Insertion Sort

$$C = \sum_{i=1}^{\lfloor n/2 \rfloor} (e_i - r_i + 2). \tag{8}$$

Since $e_i = 2i - 1 + n\backslash 2$, C can be rewritten as

$$C = \sum_{i=1}^{\lfloor n/2 \rfloor} (2i - 1 + n\backslash 2 - r_i + 2) = \sum_{i=1}^{\lfloor n/2 \rfloor} (2i - r_i + 1) + \left\lfloor \frac{n}{2} \right\rfloor (n\backslash 2). \tag{9}$$

Equations 8 and 9 show that the actual performance of 2-insertion sort depends on the value of r_i, which determines the best, the average, and the worst cases of the algorithm. Given those expressions, we proceed to the analysis of the best, worst and average cases of the algorithm.

The best case of 2-insertion sort occurs when the array is already sorted. In terms of our notation, $r_i = e_i - 1$ in every iteration of the algorithm. From equation 8, the number of comparisons performed is:

$$C_b = \sum_{i=1}^{\lfloor n/2 \rfloor} [e_i - (e_i - 1) + 2] = 3 \left\lfloor \frac{n}{2} \right\rfloor.$$

The worst case of 2-insertion sort is also similar to that of insertion sort: an array sorted in reverse (decreasing) order. This means that $r_i = 0$ in every iteration i of the algorithm. From equation 9,

$$C_w = \sum_{i=1}^{\lfloor n/2 \rfloor} (2i + 1) + \left\lfloor \frac{n}{2} \right\rfloor (n\backslash 2) = \frac{n^2}{4} + n - \frac{5(n\backslash 2)}{4}.$$

We now examine the average case of the algorithm, which depends on the expected value of r_i (denoted by $\overline{r}_i$) for any given iteration i. Denote $v[e_i]$ and $v[e_{i+1}]$ (the first two elements of the unsorted part of the array in iteration i) by a and b, respectively. For r_i to be k (any value in the interval $[0, e_i - 1]$), one of $\{a, b\}$ must be inserted into the k-th position and the other into some position l such that $l \geq k$. By finding the probability that r_i is exactly k, for every possible k, we will be able to determine $\overline{r}_i$.

Including the sentinel, there are e_i elements in the sorted part of the array in the beginning of iteration i. This means that there are exactly e_i slots where a or b can be inserted: between $v[0]$ and $v[1]$, $v[1]$ and $v[2]$, and so on; the last slot is right after $v[e_i - 1]$. Elements a and b can be inserted in a single slot or in two different slots. Hence, the number of possible configurations (c) after the insertion of a and b is

$$c = (e_i) + \binom{e_i}{2} = e_i + \frac{e_i(e_i - 1)}{2} = \frac{e_i^2 + e_i}{2} \tag{10}$$

Note that r_i will be k when one of the slots is the k-th and the other is greater than (or equal to) k. Using equation 10 and the fact that there are $e_i - k$ configurations with this property for any given $k \in \{0 \ldots e_i - 1\}$, we can determine $p_i(k)$, the probability that r_i is exactly k:

$$p_i(k) = \frac{e_i - k}{(e_i^2 + e_i)/2}$$

The expected value of r_i follows immediately from this result:

$$\overline{r}_i = \sum_{k=0}^{e_i - 1} k p_i(k) = \frac{e_i - 1}{3}.$$

Finally, using this value in equation 9, we can determine de average number of comparisons performed by 2-insertion sort:

$$\begin{aligned}
C_a &= \sum_{i=1}^{\lfloor n/2 \rfloor} \left(2i - \frac{e_i - 1}{3} + 1\right) + \left\lfloor \frac{n}{2} \right\rfloor (n \backslash 2) = \\
&= \sum_{i=1}^{\lfloor n/2 \rfloor} \left(2i - \frac{2i - 2 + n\backslash 2}{3} + 1\right) + \left\lfloor \frac{n}{2} \right\rfloor (n \backslash 2) = \\
&= \frac{n^2}{6} + \frac{7n}{6} - \frac{4(n\backslash 2)}{3}
\end{aligned} \tag{11}$$

5 Comparative Analysis of the Algorithms

5.1 Comparisons

Table 1 summarizes the results obtained in previous sections. Selection sort is the only algorithm whose performance (number of comparisons) is completely independent of the input array. All the other algorithms are quadratic on average, but linear in the worst case. For random inputs and large n, selection sort performs at least twice the number of comparisons required by the other algorithms.

Stacksort and insertion sort perform roughly the same number of comparisons for random inputs (they actually differ by a small, constant factor). Both are outperformed by 2-insertion sort, which asymptotically needs only two thirds of the comparisons they require. When it comes to the worst case, 2-insertion sort is potentially twice as fast as the others for large instances. The only inputs for which 2-insertion sort does not require fewer comparisons than insertion sort or stacksort are small, random ones (with $n \leq 4$ or $n = 6$) and sorted arrays (in this case, 2-insertion sort requires approximately 50% more comparisons).

Algorithm	best	average	worst
Selection Sort	$\frac{n^2}{2} - \frac{n}{2}$	$\frac{n^2}{2} - \frac{n}{2}$	$\frac{n^2}{2} - \frac{n}{2}$
Stacksort	n	$\frac{n^2}{4} + \frac{3n}{4} - \frac{1}{2}$	$\frac{n^2}{2} + \frac{n}{2} - 1$
Insertion Sort	$n - 1$	$\frac{n^2}{4} + \frac{3n}{4} - 1$	$\frac{n^2}{2} + \frac{n}{2} - 1$
2-Insertion Sort	$3 \lfloor \frac{n}{2} \rfloor$	$\frac{n^2}{6} + \frac{7n}{6} - \frac{4(n \backslash 2)}{3}$	$\frac{n^2}{4} + n - \frac{5(n \backslash 2)}{4}$

Table 1: Number of comparisons

5.2 Data Movements

For insertion sort and 2-insertion sort, a comparison is generally followed by a data movement. Hence, both algorithms move $O(n^2)$ elements per execution on average. Selection sort and stacksort, on the other hand, swap only one pair of elements in each iteration, regardless of the input array. As a result, they move only $O(n)$ array elements during the execution, even in the worst case.

However, if we also consider updates to the value of *maxpos* (the position of the largest element found in a given iteration), selection sort can perform up to $O(n^2)$ data movements. This happens for sorted inputs, for instance, since the number of attributions is equal to the number of local maxima found in each iteration i. For random inputs, the expected number of local maxima is $H(n - s + 1) = O(\log n)$. Summing over all iterations, this results in $O(n \log n)$ assignments to *maxpos* during the entire execution of selection sort for random inputs.

Compare this with stacksort. The pseudocode in figure 3 makes it clear that attributions to *maxpos* are always related to a stack operation (*pop* or *push*). As we have seen in section 3.3, there will be only $O(n)$ stack operations for any input array, which means that stacksort performs no more than a linear number of attributions to *maxpos*. This is asymptotically better than the $O(n \log n)$ average case of selection sort, let alone its $O(n^2)$ worst case.

5.3 Memory Requirements

Selection sort, insertion sort and 2-insertion sort are in-place algorithms, in the sense that they need only a fixed amount of extra memory regardless of the value of n. Selection sort and insertion sort need only memory enough to store one additional element; 2-insertion sort needs two extra elements.

For stacksort, on the other hand, memory usage is a major issue. It needs not only memory enough to hold an extra array element, but also up to $n + 1$ memory positions for the stack. For random inputs, however, $O(\log n)$ memory positions should suffice, as we have seen in section 3.3. It is worth mentioning that the maximum amount of memory allocated for the stack depends only on the number of elements, and not on the size of each individual one. (This is not always true. The amount of extra memory required by mergesort, for instance, is proportional not only to n, but also to the element size.)

5.4 Almost Sorted Inputs

All the algorithms mentioned, with the exception of selection sort, are very efficient for arrays divided into sections with size at most k (a constant) and organized such that every element of a given section is not smaller than any element in previous sections.

Note that for insertion sort, 2-insertion sort and stacksort, the last element of a section acts as a sentinel for the following one (this is not true for selection sort). Hence, sorting the entire array can be seen as sorting $O(n)$ subarrays of size at most k. Each section requires $O(k^2)$ comparisons, but, since k is constant, so is k^2. Therefore, any of the three algorithms can sort these arrays in linear time.

Almost sorted arrays are particularly important in optimized implementations of quicksort. The algorithm is very efficient asymptotically, but simpler, quadratic algorithms achieve better running times for very small instances. Since quicksort has a recursive nature, it can be made to process only subproblems with size greater than a given parameter (*cutoff*), ignoring smaller ones. Although the resulting array will not be completely sorted, it will be divided into sections of size at most *cutoff* such that no element of a given section is smaller than any other element in previous sections. Therefore, it can be sorted in linear time by insertion sort, 2-insertion sort or stacksort.

Note that stacksort must be used with care for almost sorted inputs. The expected size of the stack will no longer be $O(\log n)$, but $O(n)$, since at least one element of each of the $O(n)$ sections is a local maximum of the entire array.

6 Experimental Results

This section presents some experimental results that show the actual behavior of the algorithms discussed. We compare the algorithms when processing random, small ($n \leq 80$) arrays (subsection 6.2) and when used as subroutines of quicksort (6.3).

6.1 Methodology

6.1.1 Implementation and Testing Environment

The algorithms were implemented as C++ templates and compiled with `GCC` with the `-O3` (full optimization) flag. In order to make the codes as efficient as possible, pointer arithmetic was used to access the elements of the arrays (and the stack in stacksort). The tests were performed on an AMD K6-2 with 128 MB RAM running at 350 MHz under Red Hat Linux 6.1.

We compare the algorithms in terms of their running times. Operation counting could also be used, but there would be certain restrictions. Stacksort, for instance, is the only one that requires stack operations. There is no straightforward way to compare them with the operations of the insertion-based algorithms. On the other hand, actual running times provide a common ground for comparing the algorithms. All times reported are CPU times, which lead to more accurate results than real time.

One must be aware, however, that any analysis based on running times leads to results that are significantly dependent on both the implementation of the algorithms and the arquitecture in which they are tested. The relative performance of the programs is determined by the relative costs of the basic operations they perform (comparisons, data movements, stack operations, etc.). They are determined not only by problem-specific factors such as the size of the elements, but also by computer-architecture factors, such as number of registers and cache size. Furthermore, since we are dealing with very similar algorithms, register allocation strategies (in other words, the compiler) may have a decisive influence over their relative running times.

Most of the instances tested had running times too small to be measured with acceptable precision. In those cases, we considered the time necessary to sort a large set of random arrays with the same number of elements (tipically filling up most — but not all, to avoid I/O operations — of the available memory). The arrays were created in advance, already with the sentinels placed in $v[0]$. To minimize cache effects between executions, the order in which the arrays were processed was always a random permutation of the order in which they were created. A different random seed was used for each execution of the program, so neither the arrays nor the permutations were the same across different executions.

6.2 Small Instances

Figure 5 shows the performance of the quadratic algorithms for random, small arrays ($3 \leq n \leq 80$) of 32-bit integer keys. The results for an optimized version of "pure" quicksort are also reported. Each point in the graphic represents the average time necessary to sort at least 2 million arrays, divided into 10 equal-sized groups, each corresponding to one execution of the program. The number of arrays per execution was determined so as to fill up approximately 64 MB of memory; therefore, more arrays were tested for smaller values of n (for $n = 3$, there were almost 2.8 million arrays per execution).

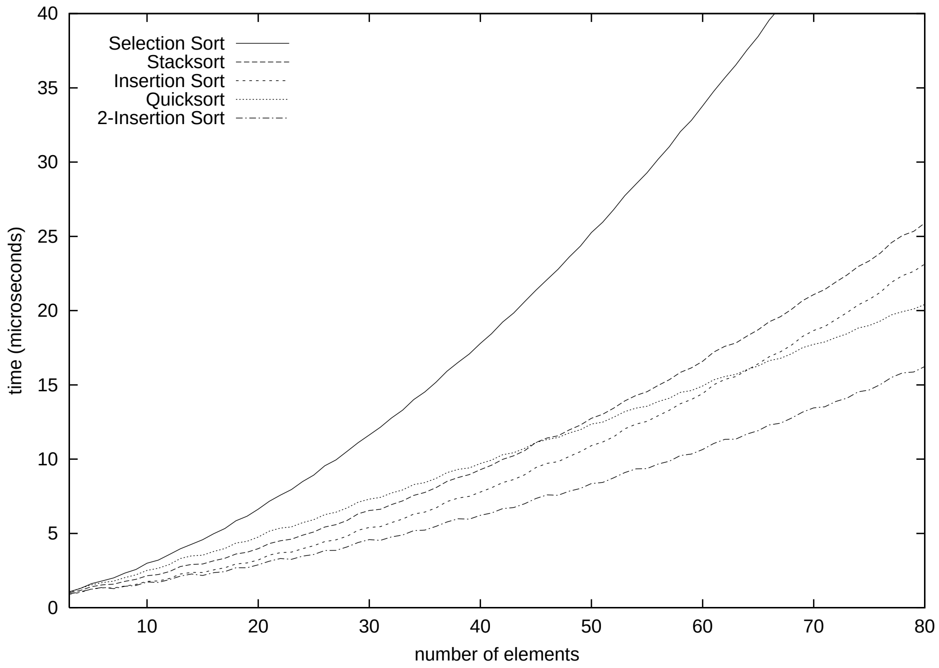

Figure 5: Running times for small instances

First of all, it is interesting to notice that the algorithms in fact are better than quicksort for small instances. This particular version of quicksort uses the median-of-three pivot selection scheme suggested by Sedgewick [2], which makes the algorithm faster for large instances but slows it down for small ones.

The relative performance of insertion sort and 2-insertion sort follows what was predicted. For really small instances ($n \leq 4$ and $n = 6$), insertion sort is the best choice. For larger instances, 2-insertion sort becomes the faster of the algorithms.

When it comes to stacksort, equations 5 and 8 show that it requires roughly the same number of comparisons in the average case as insertion sort. However, according to figure 5, the actual performance of stacksort is not as good as that of insertion sort for small values of n. The code for stacksort is more complex, specially considering that it needs to perform stack operations. But since there is only a linear number of such operations, they tend to become less important as n increases. In fact, stacksort becomes faster than insertion sort for large values of n (in our implementation, this happens for $n > 400$, approximately). Stacksort can also become a serious contender when comparisons are computationally much cheaper than assignments. This happens, for instance, when the elements to be sorted are huge records with small integer keys.

6.3 Sorting Large Arrays with Quicksort

In this section, we analyze how the quadratic algorithms can be used in conjunction with quicksort. We tested different values of *cutoff* (defined in section 5.4) for quicksort with three of the quadratic algorithms (insertion sort, 2-insertion sort and stacksort). The results were compared to each other and to "pure" quicksort. We omit the analysis of selection sort because both our theoretical and experimental results have made it clear that we cannot expect this algorithm to outperform the others (preliminary tests have confirmed this in pratice).

The general implementation of quicksort is exactly the same in every case. The median-of-three pivot selection scheme was used in all variations. For the sake of efficiency and to guarantee that only $O(\log n)$ extra memory is required by quicksort, we opted for its iterative implementation, which explicitly uses a stack to keep the subproblems that must be sorted. Unlike the tests described in the previous section, where a sentinel was inserted in $v[0]$ in advance, we made quicksort select the smaller of the first *cutoff* elements to be the sentinel before using one of the quadratic algorithms to sort v from position 2 to position n. Since determining the sentinel requires only $\Theta(\mathit{cutoff})$ time, this operation has only marginal influence over the running time of the entire algorithm.

As we mentioned in section 5.4, the three quadratic algorithms are well suited to almost sorted inputs. Therefore, we could have used "interrupted" quicksort and make only one call to a quadratic algorithm to finish the sorting process. However, our tests have shown that running these algorithms several times (whenever quicksort has to sort a subproblem with fewer than than *cutoff* elements) is a faster alternative in most modern architectures, since it makes better use of cache memory. For stacksort, this approach has the additional feature of requiring constant ($O(\mathit{cutoff})$) amount of extra memory, instead of $\Theta(n)$. In order to keep the running times as small as possible, the quadratic algorithms were implemented as macros, thus avoiding expensive function calls.

Figure 6 shows the running times of quicksort when used alongside with each of the three quadratic algorithms with *cutoffs* ranging from 4 to 64. For all curves, the value corresponding to *cutoff* zero actually represents the running time of "pure" quicksort. Each point represented is actually the average of 1,000 executions of the algorithm (10 time measurements of groups of 100 executions). The arrays consisted of 100,000 integer keys uniformly distributed in the interval $[1 \dots 10^6]$, with repetitions allowed.

The graphic shows that any of the algorithms can be useful to speed up quicksort. The best results were achieved by 2-insertion sort, which was up to 19% faster than "pure" quicksort. Despite being marginally slower than insertion sort for small *cutoffs*, for values larger than 20 2-insertion sort is the best choice. These results are compatible with those shown in figure 5, since 20 is only an upper bound on the number of elements in the subarrays actually sorted by the quadratic algorithms. It is important to notice, however, that the precise point in which 2-insertion sort supercedes insertion sort may vary according to the implementation of both algorithms and quicksort itself.

Also implementation-dependent is the optimal *cutoff*. There are some general principles, however. Firstly, faster algorithms usually achieve their optima with higher *cutoffs*. This means that less work can be done by quicksort and more by the simpler, quadratic algorithm. In our implementation, the optimal values were aproximately 12, 22 and 36 for stacksort, insertion sort and 2-insertion sort, respectively. Secondly, faster algorithms are less sensitive to the exact value of the *cutoff*. Figure 6 clearly shows that, after the optimum, running times for 2-insertion sort grow in a slower fashion than for insertion sort. This would be irrelevant if we knew in advance the optimal *cutoff* for every instance the algorithm could possibly have to solve. However, this is not feasible, specially when we expect the code for the algorithm to be compiled in more than one architecture or to deal with different element types. In those cases, 2-insertion sort tends to be more flexible than insertion sort.

In our architecture, the best result achieved by quicksort with 2-insertion sort is a little more than 1% faster than the best of quicksort with insertion sort. Moreover, 2-insertion sort is the better option for a small range of *cutoffs*: any value from 20 to 57 makes this algorithm faster than insertion sort with its optimal *cutoff*.

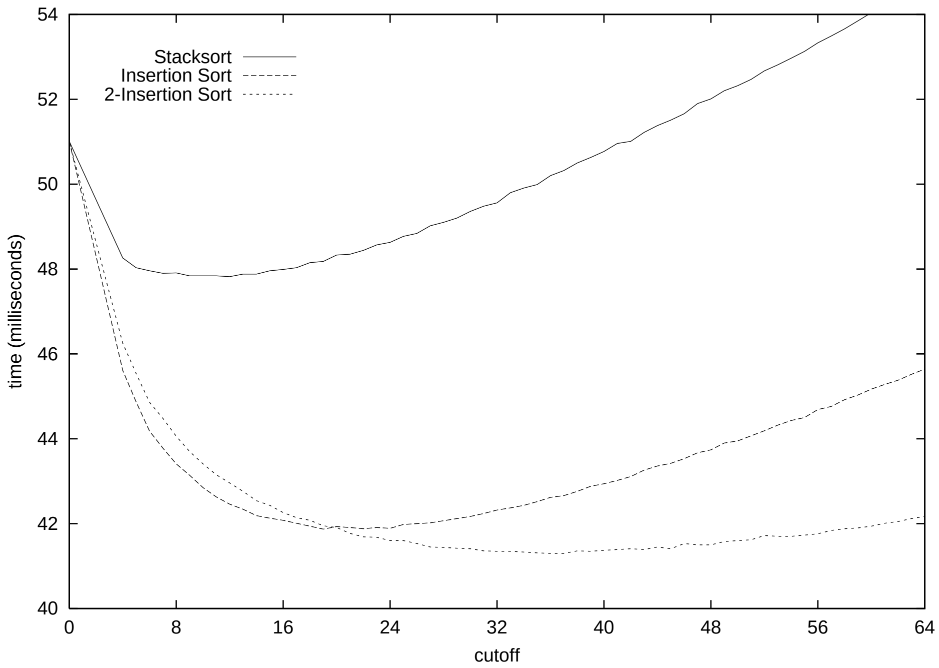

Figure 6: Performance of quicksort with three different algorithms for $n = 100,000$ (time for cutoff 0 represents "pure" quicksort on all curves)

A final, important remark concerning the use of any quadratic algorithm with quicksort must be made. Quadratic algorithms accelerate quicksort not because they execute fewer comparisons (this only happens for very small *cutoffs*), but because their basic operations are simpler. Therefore, one cannot expect to achieve performance gains as expressive as those obtained for integer arrays when sorting more complex elements (such as strings, for instance).

7 Extensions and Related Work

Insertion sort and selection sort have been extensively analyzed before [1, 2]. To our knowledge, there is no reference in the literature to an algorithm similar to 2-insertion sort. Stacksort was also developed independently, but we found out that it is very similar to an algorithm suggested by Knuth in [1] (section 5.2.3, exercise 8). Knuth's algorithm also "remembers" previously found local maxima. However, instead of using a stack, it uses an auxiliary array that contains, for each element p, the position of the last local maximum found before p became a local maximum. This approach has the disadvantage that the arrays need initialization, which can be relatively costly for small instances. Moreover, the algorithm requires $\Theta(n)$ extra memory, while stacksort can work with only $O(\log n)$ on average.

Other Algorithms Comparison-based algorithms are abundant in the literature. We selected algorithms that seem to have competitive performance in practice. Early experimental evaluation showed that bubble sort and odd-even sort, for instance, are much slower than the algorithms we tested, specially due to the amount of data movement they require. Insertion sort with binary search was also implemented, but it only outperformed linear insertion sort for relatively large arrays (with more than 100 elements). Other algorithms, not necessarily quadratic, could be tested. Shellsort and bitonic sort, for instance, may be good alternatives.

One interesting family of algorithms that should be further analized is that of generalized versions of 2-insertion sort. We use the generic demomination of k-insertion sort for these algorithms, where k is any constant greater than 1. In each iteration, the k first elements of the unsorted part of the array are sorted (using the original insertion sort, for instance) and then inserted in the appropriate positions. Analyzing the expressions that describe the complexities of k-insertion sort for various k values, we observe that, while the constant multiplying the n^2 term gets smaller as k increases, the coefficient of the linear term (n) gets larger. This means that larger k values may lead to asymptotically better algorithms, but they are not necessarily faster for small instances (which is what we are interested in). In fact, preliminary tests with 3-insertion sort have shown that, although it outperformed insertion sort, it was not faster than 2-insertion sort for small values of n.

Other Applications The algorithms we suggested were analyzed only when used to sort small arrays and subproblems of quicksort. There are, however, other situations in which they could be useful. Sorting small portions of the input in bottom-up mergesort is a simple one. Another interesting application would be to use stacksort or (specially) 2-insertion sort in shellsort, which originally works by applying insertion sort several times.

Experimental Analysis Another possible extension of this work involves more comprehensive experimental testing. Other computer architectures, programming languages and compilers could be used. One could also test the behavior of the algorithms for data structures with more complex comparison functions. Finally, testing the algorithms with both singly and doubly linked lists may be interesting. They can deal with both data structures — the only difference is that the insertion-based algorithms must use left-to-right (instead of right-to-left) linear search for singly linked lists.

References

[1] D. E. Knuth. *The Art of Computer Programming, Volume 3 / Sorting and Searching.* Addison-Wesley, second edition, 1997.

[2] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.