Comparing Performance of Insertion Sort and Selection Sort

Nitin Verma
mathsanew.com

November 22, 2021

Does *Insertion Sort* always outperform *Selection Sort*? Or vice versa? If not always, under what conditions one would become faster than the other? In this article, we will observe the factors affecting performance of these two sorting algorithms, and see examples where one of them outperforms the other.

An implementation of the two sorting algorithms in C has been provided below. The input array $a[\,]$ of n elements of type *object* (an structure) needs to be sorted in ascending order, using the comparison function $less\_than()$. We will be denoting any subarray of $a[\,]$ from some index x to some index y ($y \geq x$) as $a[x : y]$.

```
typedef struct object
{
  unsigned int key;
  short        field1;
  char         field2[2];
  int          more_fields[2];
} object;

#define less_than(o1, o2) (o1.key < o2.key)
```

Copyright © 2021 Nitin Verma. All rights reserved.

```
void insertion_sort(object a[], int n)
{
  int    i, j;
  object t;

  i = 1;

  /* loop-invariants (specified for case n > 0):
       P: a[0:i-1] contains same elements as the initial a[0:i-1]
       Q: a[0:i-1] is sorted */

  while(i < n)
  {
    /* create hole (position that can be overwritten) at index i */
    t = a[i];
    j = i;

    /* loop-invariants:
         R: the "hole" is at index j
         S: t is less than each element in a[j+1:i] */

    while(j >= 1 && less_than(t, a[j-1]))
    {
      a[j] = a[j-1];
      j = j-1;
    }

    /* copy t to the hole at index j */
    a[j] = t;

    i++;
  }

  /* P and Q hold with (i = n) */
}
```

```
void selection_sort(object a[], int n)
{
  int    i, j, min;
  object t;

  i = 0;

  /* loop-invariants:
       P: a[0:i-1] contains the i smallest elements of initial a[]
       Q: a[0:i-1] is sorted */

  while(i < n-1)
  {
    min = i;
    j = i+1;

    /* loop-invariant R: a[min] is minimum in a[i:j-1] */

    while(j < n)
    {
      if(less_than(a[j], a[min]))
        min = j;

      j = j+1;
    }

    /* R holds with (j = n) */

    if(min != i)
    {
      /* swap a[i] and a[min] */
      t = a[i];
      a[i] = a[min];
      a[min] = t;
    }

    i++;
  }

  /* P and Q hold with (i = n-1) */
  /* Now a[n-1] must be among the largest, so a[0:n-1] is sorted */
}
```

Counting the Operations

We will simply use "transfer" to refer to the copying of an *object* from one location to another, and "comparison" to refer to a *less_than*() call.

In *insertion_sort*(), the outer loop runs $(n-1)$ times. Since the inner loop can run anywhere between 0 to i times, for our analysis we assume it to run fi times for some f in interval $[0, 1]$. Each iteration of the outer loop will do $(fi+2)$ transfers. Due to the inner loop, there will be $(fi+1)$ comparisons. So the total number of transfers and comparisons in the method will be respectively:

$$\begin{aligned} T_I &= \sum_{i=1}^{n-1}(fi+2) = \frac{fn(n-1)}{2} + 2(n-1) \\ C_I &= \sum_{i=1}^{n-1}(fi+1) = \frac{fn(n-1)}{2} + (n-1) \end{aligned}$$

For the case when the input array is already sorted, the inner loop will run 0 times, giving $f = 0$, and so $T_I = 2(n-1)$, $C_I = (n-1)$. Here the method has a linear time complexity in n. But when the input array is in decreasing order, the inner loop will run i times, giving $f = 1$, and so $T_I = (n-1)(n+4)/2$, $C_I = (n-1)(n+2)/2$. In this case, the method has a quadratic time complexity in n.

For our analysis below, we will assume $f = 1/2$ (the element $a[i]$ inserted in the middle of the sorted subarray), and so:

$$\begin{aligned} T_I &= \frac{(n-1)(n+8)}{4} \\ C_I &= \frac{(n-1)(n+4)}{4} \end{aligned} \tag{1}$$

In *selection_sort*() too, the outer loop runs $(n-1)$ times. But the inner loop always runs $(n-i-1)$ times. Each iteration of the outer loop will do $(n-i-1)$ comparisons (due to the inner loop) and 3 transfers (for swapping $a[i]$ and $a[min]$). So the total number of transfers and comparisons in the method will be respectively:

$$\begin{aligned} T_S &= 3(n-1) \\ C_S &= \sum_{i=0}^{n-2}(n-i-1) \\ &= \sum_{k=1}^{n-1} k \quad \{\text{using } k = n-i-1\} \\ &= \frac{n(n-1)}{2} \end{aligned} \tag{2}$$

Performance Comparison

The outer loop of both the methods runs $(n-1)$ times, and their inner loop runs in total almost $(n-1)(n+4)/4$ and $n(n-1)/2$ times (using C_I and C_S from (1) and (2)). If the work done by an iteration of the inner loop were to be roughly the same for both methods (and similarly for the outer loop excluding the inner loop's work), we could say that *insertion_sort*(), having lesser iterations count, should outperform *selection_sort*().

But the factors affecting their *relative* performance can be many, in addition to the visible difference in their set of operations per iteration: compiler optimizations, hardware performance factors related to branches and locality of reference, and so on. (For example, in each iteration of the outer loop, *selection_sort*() accesses the complete subarray $a[i : n-1]$, but *insertion_sort*() accesses only a part of the subarray $a[0 : i]$; this gives the former a lower locality of reference.) In fact, for any algorithm in general, performance related conclusion should be made very carefully with the underlying conditions specified, and often there is no simple answer.

Here we will focus on two factors which may become significant for the performance of any comparison based sorting algorithm: the cost of array elements' transfer and the cost of comparing them. The transfer cost becomes significant, for example, when the underlying storage has slow read/write speed, or when each array element has large size. The comparison cost becomes significant, for example, when the keys are of a complex type, or the comparison involves many steps instead of a simple '<' operator.

We note from (1) and (2) that the number of transfers in *selection_sort*() is always linear (in n), but it is quadratic for *insertion_sort*() (assuming $f = 1/2$). This much difference is not present in the growth of their number of comparisons; it is quadratic for both. So, as the transfer cost gets sufficiently high, this difference between T_I and T_S will start becoming visible in the total costs, and *selection_sort*() will start outperforming *insertion_sort*(). As mentioned earlier, one way the transfer cost would increase is by increasing the size of each array element.

For example, with a randomly generated array $a[\,]$ of $n = 30000$ elements, I find that *selection_sort*() starts to outperform *insertion_sort*() when *object*'s *more_fields*[ ] array has 7 or more elements (resulting in larger sized *object*). These observations are for a particular system and will vary across hardware and compiler options.

We also know from (1) and (2) that the number of comparisons required by *selection_sort*() is almost double than that of *insertion_sort*() (assuming

$f = 1/2$). So an increase in comparison cost should increase the total cost of *selection_sort*() more than that of *insertion_sort*(). While keeping the *more_fields*[ ] array at 7 elements, if the comparison function is now modified as below, I notice *insertion_sort*() outperforming *selection_sort*() (again, this is system specific). Note that this function is equivalent to the earlier one.

```
#define less_than(o1, o2) \
  ((o1.key/10)*10 + o1.key%10 < (o2.key/10)*10 + o2.key%10)
```

■