ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.1.
Assistances and comments will be acknowledged.

8.9 Inversions

We have already defined inversions in 8.1 *Sorting algorithms* as a measure for determining the *unsortedness* of a list. We will investigate a few further properties of inversions.

8.9.1 Basic properties

A pair of entries (a_j, a_k) in a list where

$j < k$ but $a_j > a_k$, that the number of such pairs is $\binom{n}{2} = \frac{n(n-1)}{2}$ and that the expected number of

inversions in a randomly generated list should be approximately $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \Theta\left(n^2\right)$.

8.9.2 Average Range of Inversions

Now, what is the probability that a randomly generated list will have close to $n(n-1)/4$ inversions? For example, is it possible that a randomly generated list of size 1000 will have significantly fewer than a quarter of a million inversions? Unfortunately, as n becomes large, it becomes more-and-more likely. The standard deviation of the expected number of inversions in a randomly generated list grows at a rate of $\sqrt{\frac{1}{72}n(n-1)(2n+5)} = \Theta(n^{3/2})$ which slower than the expected number of inversions; therefore you are essentially guaranteed that a randomly generated list will have relatively close to the expected number of inversions. If you were to plot

$$\frac{n(n-1)}{4} \pm 1.96 \cdot \sqrt{\frac{n(n-1)(2n+5)}{72}},$$

you would see the *95 % confidence interval* for any value of n. This is shown in Figure 1.

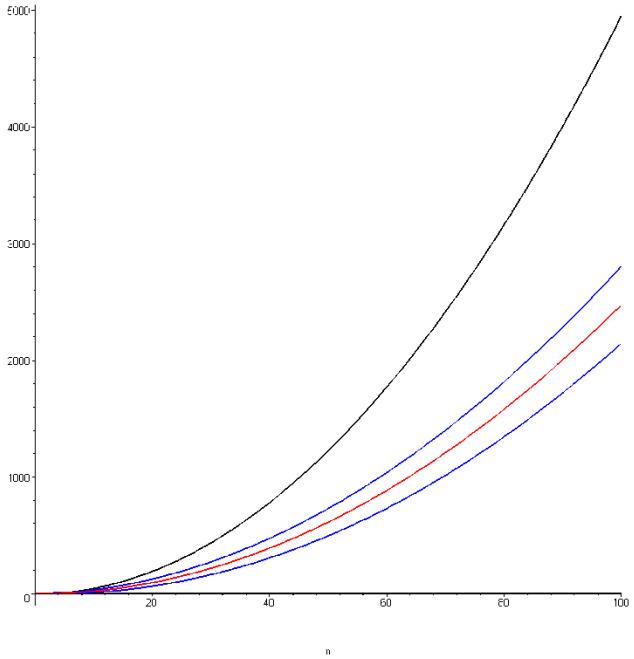

Figure 1. A 95 % confidence interval for the number of inversions in a randomly generated list.

A 95 % confidence interval says that 19 times out of 20, a result will appear in that interval. For example, in our previous example, with had a list size of $n = 20$ and therefore the confidence interval would be 95 ± 30.2 inversions. If increased the size of the array to 1000, we would expect $249\,750 \pm 10\,338$ inversions. Consequently, 19 times out of 20, a randomly generated list of size 1000 would have between 239412 and 260088 inversions. A 99 % confidence interval would be

$$\frac{n(n-1)}{4} \pm 2.575 \cdot \sqrt{\frac{n(n-1)(2n+5)}{72}},$$

so even then, 99 times out of 100, the number of inversions would not go outside the range [236168, 263331]. To learn more about confidence intervals, see Wikipedia.

8.9.3 Counting inversions

Suppose we want to count the number of inversions in a list. We will look at two algorithms: the naïve algorithm that runs in $\Theta(n^2)$ time with $\Theta(1)$ memory, and another that runs in $\Theta(n \ln(n))$ time but with $\Theta(n)$ additional memory.

8.9.3.1 Naïve implementation

In order to count the number of inversions, we could run a simple algorithm by comparing all possible pairs, as in the program:

```
template <typename Type>
int inversions( Type *array, int n ) const {
    int count = 0;
    for ( int i = 0; i < (n - 1); ++i ) {
        for ( int j = (i + 1); j < n; ++j ) {
            if ( array[i] > array[j] ) {
                ++count;
            }
        }
    }
    return count;
}
```

The run-time of this algorithm is clearly $\Theta(n^2)$. Can we do better? To consider an answer, let's look at merge sort.

8.9.3.2 Algorithm based on merge sort

In the merge sort algorithm, we split the list into two, sort each list recursively, and then merge the resulting sorted lists. For example, take the array

14	15	5	19	9	2	0	10	3	6	17	13	4	8	18	12

which has 58 inversions—very close to the expected number of inversions in a randomly generated list.

Going through the process of merge sort, we would split this array into two, and sort the two lists:

0	2	5	9	10	14	15	19

3	4	6	8	12	13	17	18

Suppose in the process of generating these two lists, we determined that there are 19 and 9 inversions in these sub-lists, respectively.

In merging these two arrays, we note that when we finally insert 3 into the merged list:

0	2	5	9	10	14	15	19

3	4	6	8	12	13	17	18

0	2	3													

then 3 must have formed an inversion with 5, 9, 10, 14, 15 and 19 in the original list.

Similarly, once we merge 4,

0	2	5	9	10	14	15	19

3	4	6	8	12	13	17	18

0	2	3	4												

we again note that 4 must have also formed another six inversions with the same entries in the original list. Next, we merge 5, and then 6:

0	2	5	9	10	14	15	19

3	4	6	8	12	13	17	18

0	2	3	4	5	6										

Again, 6 must have formed an inversion with 9, 10, 14, 15 and 19. Following the merging of these two lists, we count 6 + 6 + 5 + 5 + 3 + 3 + 1 + 1 = 30 inversions. Thus, we have a total of 19 + 9 + 30 = 58 inversions.

The cost of such an algorithm, however, is $\Theta(n)$ additional memory, whereas the original implementation ran with $\Theta(1)$ additional memory—a few local variables.

8.9.3.3 Implementation

The implementation is also quite similar to that of merge sort:

```
template <typename Type>
int inversions( Type *array, int n ) {
    if ( n <= 1 ) {
        return 0;
    }

    nl = n/2;
    nr = n - n/2;

    Type larray[nl];
    Type rarray[nr];

    // split the array into two halves
    split( array, n, larray, nl, rarray, nr );

    // recursively call inversions on both halves
    int count = inversions( larray, nl ) + inversions( rarray, nr );

    // merge returns the number of inversions as a result of merging
    return count + merge( array, n, larray, nl, rarray, nr );
}
```

together with the two helper functions

```
template <typename Type>
void split( Type *array, int n, Type *larray, int nl, Type *rarray, int nr ) {
    assert( nl + nr == n );
    for ( int i = 0; i < nl; ++i ) {
        larray[i] = array[i];
    }

    for ( int i = 0; i < nr; ++i ) {
        rarray[i] = array[nl + i];
    }
}

template <typename Type>
int merge( Type *array, int n, Type *larray, int nl, Type *rarray, int nr ) {
    int il = 0, ir = 0, ia = 0, count = 0;
    while ( il < nl && ir < nr ) {
        if ( larray[il] <= rarray[ir] ) {
            array[i] = larray[il];
            ++il;
        } else {
            array[i] = rarray[ir];
            ++ir;
            count += nl - il;   // we moved the entry in right array past anything
        }                       // not yet merged from the left array
        ++i;
    }
    for ( ; il < nl; ++il, ++i ) array[i] = larray[il];
    for ( ; ir < nr; ++ir, ++i ) array[i] = rarray[ir];
    return count;
}
```