A glance at *big-O* notation.

There are precise way of talking about the approximate properties of programs. We are going to use one, called *big-O* notation.

If we write that the execution time of program *P* is $O\left(N^2\right)$ we mean "in the worst case, its execution time is roughly proportional to N^2, given a large enough problem".

Usually, worst-case behaviour is much easier to work with: what a program does given the most fiendish problem (of size *N*, or whatever) that there can be.

Sometimes, programs have different behaviour on large and small problems.

So: in the worst case, and given a large enough problem.

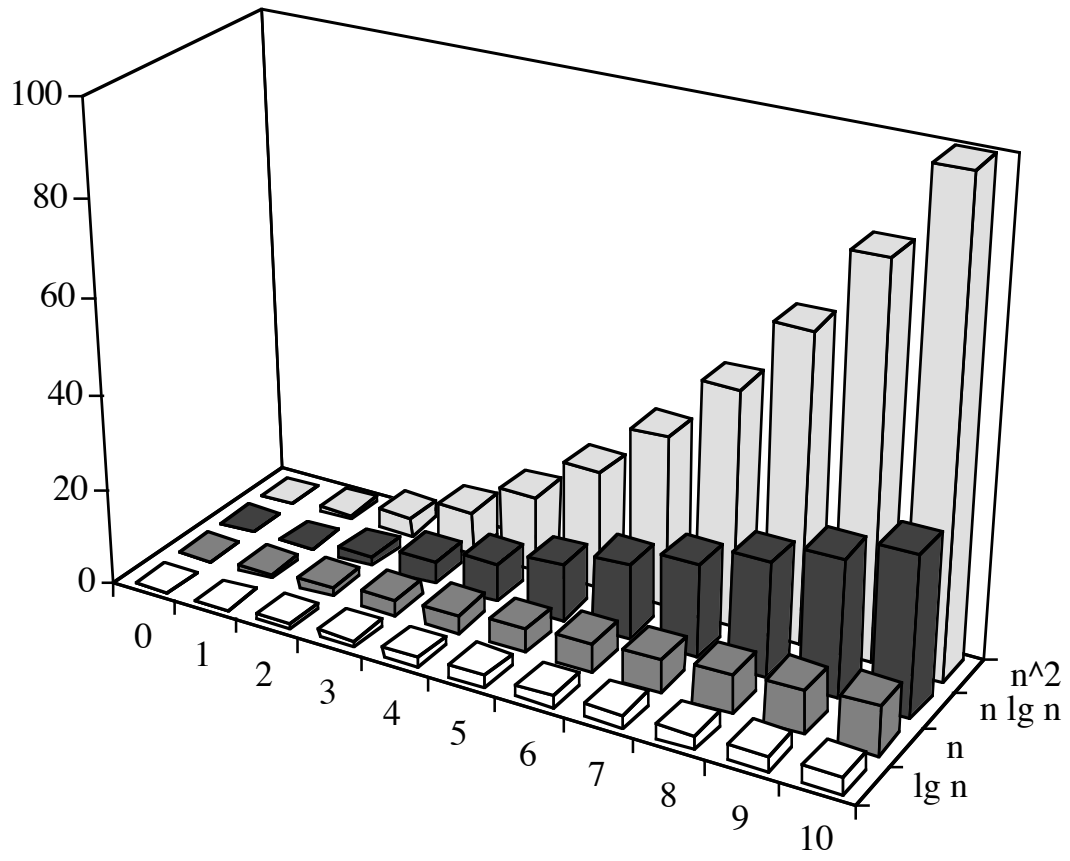
> *Later in the course I shall be more precise about the meaning of big-O notation; for the moment just treat it as a convenient shorthand.*

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

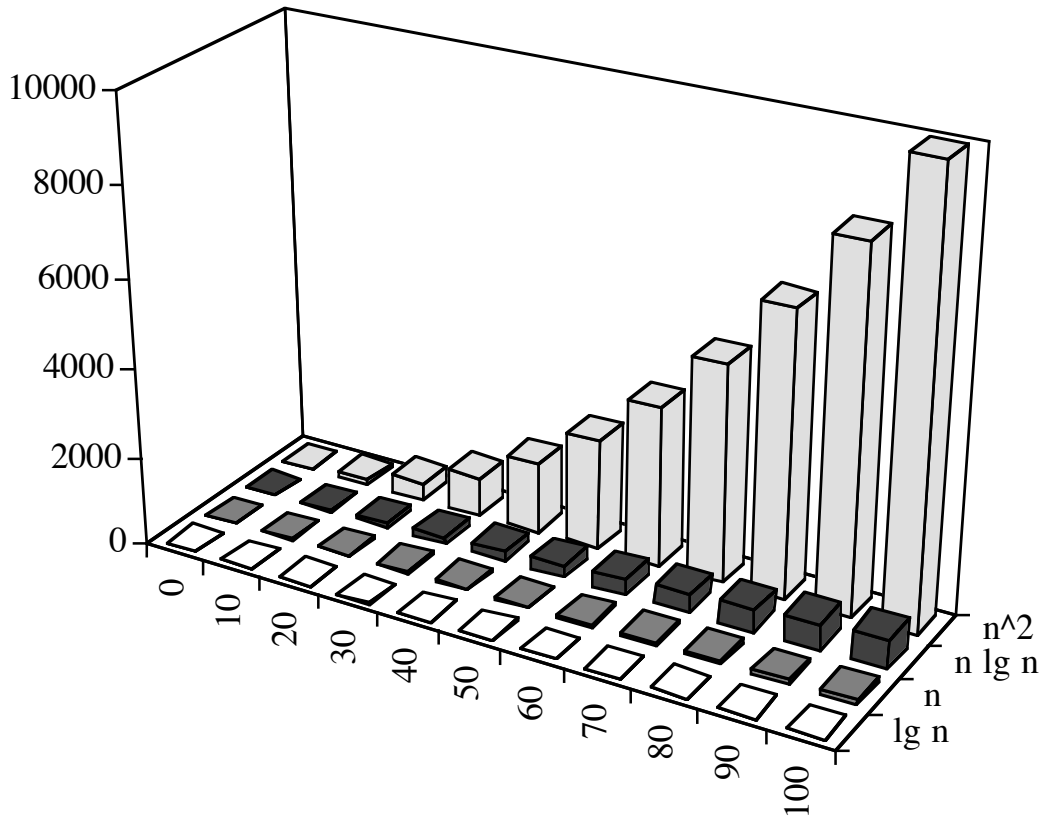
We have already seen algorithms for the same problem which have $O(N)$ and $O\left(N^2\right)$ execution times.

It is possible to be better than $O(N)$; it's possible to be better than $O(N)$ and worse than $O\left(N^2\right)$. Many algorithms have $O(\lg N)$ or $O(N \lg N)$ execution times

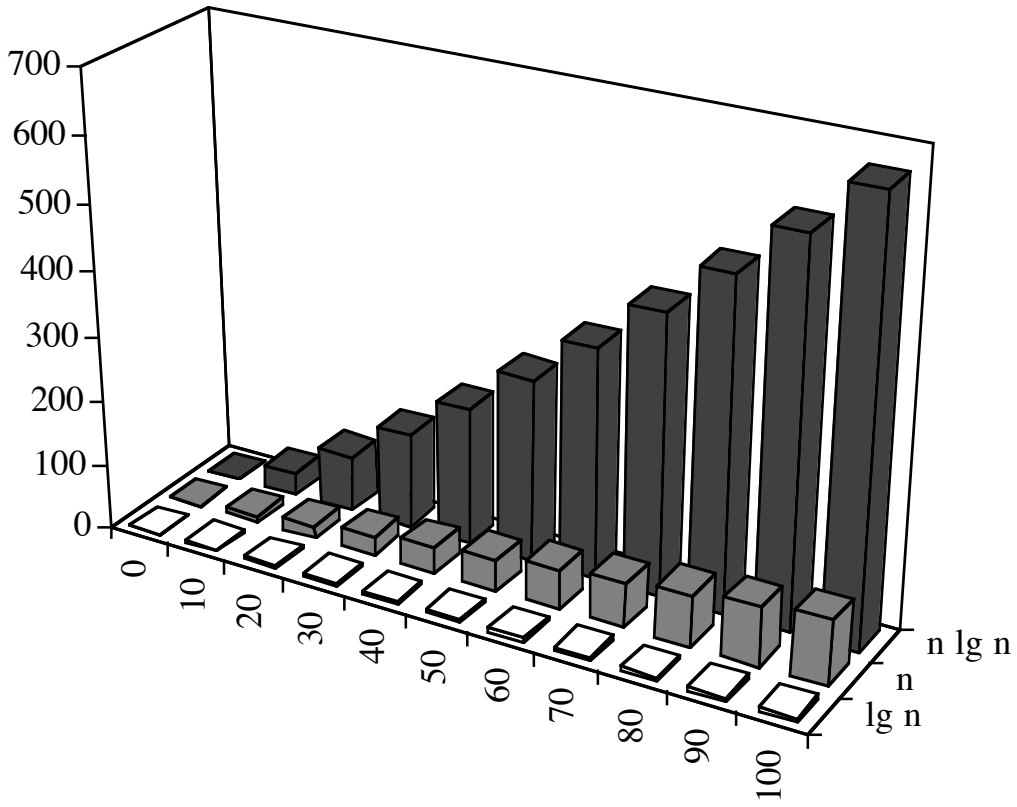
$\lg N$ *is* $\log_2 N$

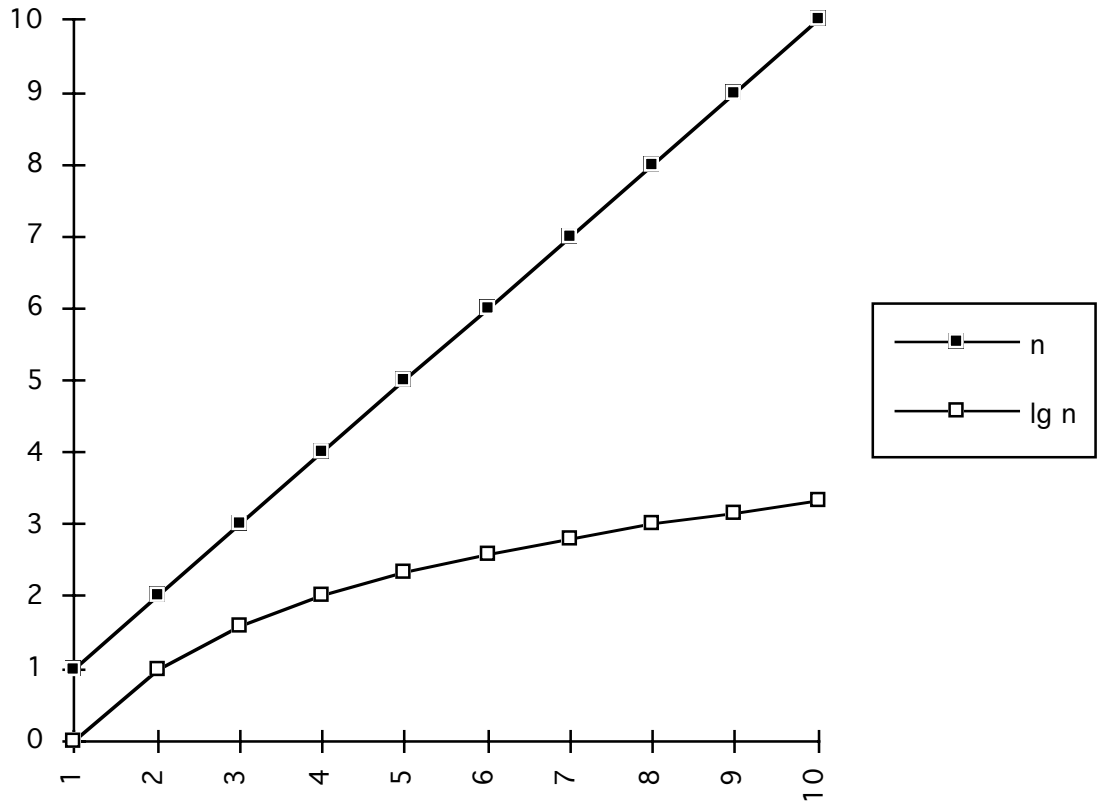


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON


10000
8000
6000
4000
2000
0
0
10
20
30
40
50
60
70
80
90
100
n^2
n lg n
n
lg n

If we leave out the worst offender, we can see how the other three compare:





$\lg N$ *grows so slowly that* $N \lg N$ *looks almost linear.*

We shall see that execution times for obvious sorting algorithms are $O(N^2)$*, clever ones are* $O(N \lg N)$.

We shall see that execution times for obvious searching algorithms are $O(N)$*, clever ones are* $O(\lg N)$ ***or better***.

If you think all this is impossibly detailed and nit-picking, ***you are studying the wrong subject****.!!*

Sorting.

This is a classical computer science problem, with all kinds of practical applications.

It's important because you can

- merge sorted arrays in $O(N)$ time;
- search a sorted array in $O(\lg N)$ time;
- find the median of a sorted array in $O(1)$ (constant: *better* than logarithmic) time;
- eliminate duplicates in a sorted array in $O(N)$ time;
- ... and so on.

Sorting is a 'high-level primitive' in lots of program design work: lots of solutions involve sorting the data at some stage or other.

Specifying a sorting algorithm

The problem is to take in a (possibly disordered) sequence of values and to output an ordered sequence.

We can sort anything on which we define an order: names, numbers, bus routes, football teams, pop groups ... Just define the ordering.

As an example we are going to take sequences of integers. The obvious ordering is then either (<) or (>), but we shall use (≤), because we don't mind if our input sequences contain repetitions.

> *technical language: a sequence ordered by (<) is in* ascending order*; (>) is* descending order*; my chosen ordering (≤) is* non-descending *(think about it!) and then, of course, (≥) is* non-ascending order.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

The specification says: take a possibly disordered sequence and produce an ordered sequence. What we mean is, for example:

Input	Output
[4, 3, 1, 7]	[1, 3, 4, 7]
[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
[1, 2, 3, 1, 2, 3]	[1, 1, 2, 2, 3, 3]

But a *strict* reading of the specification reveals a flaw: our program could behave like this:

Input	Output
[4, 3, 1, 7]	[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
[1, 2, 3, 1, 2, 3]	[1, 2, 3, 4, 5]

This is a famous flaw: the specification doesn't relate the output to the input.

To be more precise:

start with an array A;

produce A', such that (i) A' is ordered by ($\leq$);
(ii) A' is a permutation of A.

technical language: 'permutation'. Look it up in a dictionary: it sort of means 're-arrangement', 'shuffle'.

This is what 'ordered by ($\leq$)' means:

$$|B| = n \rightarrow \forall i, j \left(0 \leq i < j < n \rightarrow B_i \leq B_j \right)$$

(read as: if B is a sequence of length n, then whenever i comes before j and both are indices within the bounds of B, B_i can be put before B_j in the ($\leq$) ordering.)

Notice the technical language: 'indices', 'ordering', 'bounds'. Here, the operation |...| means 'length of'.

Is an empty sequence (a sequence of length 0) sorted? Well *certainly it is*. The definition of 'ordered'

$$\forall i, j\left(0 \le i < j < n \rightarrow B_i \le B_j\right)$$

is satisfied if n is zero, simply because there are no counter examples – we can't find i and j such that $0 \le i < j < 0$.

> *If there are no counter-examples it must be true, mustn't it?*
>
> *But there are no counter-examples (I hope) to the remark "all the circus elephants in this room are drunk". So it must be true, mustn't it?*
> *Can I shake your faith in what you read, so that you* ***challenge*** *it?*

For just the same reason a single-element sequence is sorted – we can't find i and j such that $0 \le i < j < 1$.

By the time we get to two-element sequences the content of the sequences begin to matter: $i = 0$ and $j = 1$ satisfies $0 \le i < j < 2$, and we know that we must have $B_0 \le B_1$.

It's more difficult to say what a permutation of a sequence is. If we write $Freq(B, x)$ to mean 'the number of times the value x occurs in sequence B' then 'C is a permutation of B' can be written as

$$\forall i \big(Freq(B, i) = Freq(C, i) \big)$$

(in words: every integer i must occur exactly as many times in B as it does in C.)

That condition is impossible to check in finite time, because there are infinitely many integers. It won't do. Oh dear.

> *Questions like: "can you calculate the answer in finite time?" matter greatly to computer scientists. They are at the root of the subject of this course and other courses. Since I'm not going to prove formally that my programs satisfy this condition, it doesn't matter that I can't check it, but I'm going to proceed as if it did matter.*

We really only need to check the integers which actually occur in B and C:

$$|B| = n \rightarrow$$

$$\forall i \left(0 \le i < n \rightarrow \begin{pmatrix} Freq(B, B_i) = Freq(C, B_i) \wedge \\ Freq(B, C_i) = Freq(C, C_i) \end{pmatrix} \right)$$

(in words: every value B_i must occur exactly as many times in B as it does in C, and vice-versa for C_i.)

We don't need to say that the length of sequence B is the same as the length of sequence C – it's an implicit consequence of the definition.

Can you spot what goes wrong if we only check the B_i frequencies and ignore the C_is? That kind of 'logical debugging' is what computer scientists must be able to do.

Can you spot what's wrong with this definition of permutation? It isn't simply that it misses a 'vice-versa' condition – it's wronger than that.

$$|B| = n \rightarrow \forall i \left(0 \le i < n \rightarrow \exists j \left(0 \le j < n \rightarrow B_i = C_j \right) \right)$$

The specification of a sorting algorithm, for the purposes of this discussion, is that it starts with an array A of length n, and it produces A' which is a permutation of A and is ordered by ($\leq$).

Further we need to say:

A sequence C of length n is ordered if

$$\forall i, j \left(0 \leq i < j < n \rightarrow C_i \leq C_j \right)$$

C is a permutation of B (and B is therefore a permutation of C) if

$$\forall i \left(0 \leq i < n \rightarrow \begin{pmatrix} Freq(B, B_i) = Freq(C, B_i) \wedge \\ Freq(B, C_i) = Freq(C, C_i) \end{pmatrix} \right)$$

We shan't often bother with specifications so difficult as the specification of a permutation. It is included here just to show that it is possible to be precise, if you are prepared to make the effort.

this is by no means the only, nor even the best, definition of what it means for B to be a permutation of C.

If I had said that the output should be in ascending (<) order, I would have written an *unsatisfiable* specification. There is no algorithm which will sort the sequence $[1,2,1,2]$ into ascending order!

> *How would I have to restrict the definition of the problem to allow the specification to use (<) order rather than (≤) order?*.

Selection sort: a quadratic – $O(N^2)$ – sorting algorithm.

i find the smallest thing in $A[0..n-1]$ and exchange it with $A[0]$;

ii then find the smallest thing in $A[1..n-1]$ and exchange it with $A[1]$;

iii then find the smallest thing in $A[2..n-1]$ and put it in $A[2]$;

... and so on, until all you have left is $A[n-1..n-1]$, (which is already sorted) or $A[n..n-1]$ (ditto).

I shan't write the whole algorithm – that's for you to do in the lab.

To find the smallest thing in $A[i..n-1]$:

```
min = A[i]; // min value
minp = i;   // position at which min value occurs
for (j=i+1; j<n; j++)
  if (A[j]<min) { min=A[j]; minp=j; }
```

This program will finish with a copy of ***a*** smallest element

> *not necessarily **the** smallest element*

in *min*, and its index in *minp*. It doesn't change *i* or $A[i]$.

Next, put $A[i]$ in $A[minp]$, and $A[minp]$ in $A[i]$. We have a copy of $A[minp]$ in *min*:

```
A[minp]=A[i]; A[i]=min;
```

Now to find the smallest thing in $A[i..n-1]$, the first program does $n-i-1$ comparisons $A[j] < min$ in every case.

> *and about $n-i$ comparisons $j < n$, as part of the operation of the* for.

The number of assignments it does depends on the result of those comparisons: in the worst case it does $n - i + 1$ assignments.

> *hard to say how many it does 'on average'. Stick to the worst case.*

The number of comparisons is clearly approximately proportional to $n - i$, in every case. The number of assignments is clearly proportional to $n - i$, in the worst case.

So the execution time of this part of the program could be said to be $O(n - i)$.

In step (i), to find the smallest thing in $A[0..n-1]$, it does work proportional to n; in step (ii), to find the smallest thing in $A[1..n-1]$, it does work proportional to $n-1$, ... and so on.

Therefore selection sort takes $O(n^2)$ time.

The program doesn't use any extra arrays, and it only uses four variables (i, j, *min*, *minp*).

Therefore selection sort uses $O(1)$ space.

Bubble-sort: another quadratic sorting algorithm.

Almost the same idea as selection sort: find the smallest thing in $A[0..n-1]$ and permute the array so that it appears in $A[0]$; then find the smallest thing in $A[1..n-1]$ and permute the array so that it appears in $A[1]$; ... and so on.

To find the smallest thing in $A[i..n-1]$ and permute the array so that it appears in $A[i]$:

```
for (j=n-1; j>i; j--)
  if (A[j]<A[j-1]) {
    temp=A[j]; A[j]=A[j-1]; A[j-1]=temp;
  }
```

In time, this is an $O(n^2)$ algorithm; in space it is $O(1)$.

You can make the argument yourself.

> *you may be tested on the argument. Discuss it with your tutor and all your friends.*

Some teachers prefer bubble sort to selection sort, because it is possible to 'stop early'.

If the array is already sorted, the algorithm will make no exchanges. We detect that with a variant the standard '∃' search trick:

```
changed = false;
for (j=n-1; j>i; j--)
  if (A[j]<A[j-1]) {
    temp=A[j]; A[j]=A[j-1]; A[j-1]=temp;
    changed = true;
  }
```

> *the trick is based on the fact that the trivial case of ∃i... evaluates to / describes 'true'. Think of the empty set; think of drunken circus elephants.*

If the program gets to the end of a step without making any changes, then no more steps are needed.

> *But in the worst case it does more work than selection sort. You should be able to make an argument to support this.*
>
> *In the average case it does more work than selection sort. You should be able to verify this with random data in the lab.*

Insertion sort: the best little $O(N^2)$ algorithm.

i $A[0..-1]$ is already sorted;

ii $A[0..0]$ is already sorted;

iii to sort $A[0..1]$, given that $A[0..0]$ is already sorted, either leave it alone (because $A[0] \leq A[1]$) or *insert* $A[1]$ into $A[0..0]$ before $A[0]$ (because $A[1] \leq A[0]$);

iv to sort $A[0..2]$, given that $A[0..1]$ is already sorted, either leave it alone (because $A[1] \leq A[2]$) or *insert* $A[2]$ before $A[0]$ (because $A[2] \leq A[0]$) or between $A[0]$ and $A[1]$ (because $A[0] \leq A[2] \leq A[1]$);

...

ii+i to sort $A[0..i]$, given that $A[0..i-1]$ is already sorted, either leave it alone (because $A[i-1] \le A[i]$) or <u>*insert*</u> $A[i]$ before $A[0]$ (because $A[i] \le A[0]$) or between $A[j-1]$ and $A[j]$ (because $A[j-1] \le A[i] \le A[j]$);

...

So the problem is to find a position j such that $0 \le j \le i \wedge (j = 0 \vee A[j-1] \le A[i]) \wedge A[i] \le A[j]$.

notice that if $j = i$ we don't have to move anything. Sneaky!

The analysis above shows that we begin our work at step (iii), with $i = 1$, and we increase i at every step.

Curse the Java designers, once again, for using the equals sign to mean 'becomes'. In normal mathematical notation it means 'equal to'.

```
for (int i=1; i<n; i++) ...
```

On each execution of the loop body, $A[i] \leq A[i]$, so if we start with `j=i`; we have established the first and third parts of the condition; it remains to reduce *j* until the middle part is satisfied:

```
for (int i=1; i<n; i++) {
  for (int j=i; j!=0 && A[j-1]>A[i]; j--) ;
  ...
}
```

> *The semicolon on the second line of this program is **not a mistake** and it is **not unnecessary**.*

Having found *j*, we can move $A[j..i-1]$ up by one position, taking care to make a copy of $A[i]$ first:

```
for (int i=1; i<n; i++) {
  for (int j=i; j!=0 && A[j-1]>A[i]; j--) ;
  Value tmp = A[i];
  for (int k=i; k>j; k--) A[k]=A[k-1];
  A[j] = tmp;
}
```

> *Variables i, j and k have to be* `int`*s, but the elements of A can be any type – not necessarily* `int`.

Each of the inner *for*s in this program does (worst case) work proportional to i.

> *There is an argument (e.g. Weiss p226) that on average j is about half i; that means the program is on average twice as fast as in the worst case.*

Those linear inner loops are put together in such a way (2+3+...) as to make a triangle of execution times, so in execution time this is another $O(N^2)$ sorting algorithm.

It uses three variables, so in space it's another $O(1)$ algorithm.

Speeding up insertion sort.

The two inner loops of the program above run through just exactly the same values (from i down to the value of j such that $j = 0 \vee A[j-1] \leq A[i]$). So it's possible to combine them:

```
for (int i=1; i<n; i++) {
  Value tmp = A[i];
  for (int j=i; j!=0 && A[j-1]>A[i]; j--)
    A[j]=A[j-1];
  A[j] = tmp;
}
```

and then it's possible to avoid recalculation of $A[i]$:

```
for (int i=1; i<n; i++) {
  Value tmp = A[i];
  for (int j=i; j!=0 && A[j-1]>tmp; j--)
    A[j]=A[j-1];
  A[j] = tmp;
}
```

See Weiss, p225.

Why is insertion sort the best little algorithm?

1 It uses no more space than the other simple sorts.

2 Fancier algorithms use more space.

3 In the best case, it is $O(N)$ (if the array is sorted, then the inner loop never does anything).

4 On average it moves about half as many things as bubble sort, each about twice as effectively.

Check points 3 and 4 in the lab.

Be prepared to rehearse arguments in support of each of these points in a test. Talk to your tutors, and your friends.

Can selection sort be faster than insertion sort?

Selection sort does $O(N)$ exchanges, and on average about twice as many comparisons as insertion sort; its best case is very much the same as its worst.

Selection sort does *fewer exchanges* than insertion sort, at the cost of some extra comparisons.

If the cost of comparisons dominates, insertion sort is a winner.

If the cost of exchanges dominates (e.g. in sorting collections of large data with small keys), selection sort can be a winner.