Introduction to Algorithms 1998

Short name I2A

Lectures by Richard Bornat

Labs by Colin Blackett, Richard Bornat and Peter O'Hearn

all lectures in PP1 (Tuesdays 10 till 12)

(with a couple of breaks for the sake of humanity ...)

labs Tuesdays 2-6 in ITL.

Book of the course will be "Data Structures and Problem Solving using Java" by Weiss, published by Addison-Wesley; it isn't quite available at the time of writing.

To see a world in a grain of sand,
And heaven in a wild flower,
Hold infinity in the palm of your hand,
And eternity in an hour.

William Blake: Auguries of Innocence

He that would do good to another man must do it in Minute
Particulars,
General Good is the plea of the scoundrel, hypocrite and flatterer,
For Art and Science cannot exist but in minutely organised
Particulars.

William Blake: Jerusalem

He who shall teach the child to doubt,
The rotting grave shall ne'er get out.

William Blake: Auguries of Innocence

To follow this course:

- you should know how to program (and/or be willing to learn quickly);
- you should know some algebra and some arithmetic (and/or be willing to learn quickly);
- you should know some predicate calculus (and/or be willing to learn quickly).

You should already understand:

- that nobody can teach you anything, but that you can learn if you are shown what you need to learn;
- that technical language conveys complicated ideas concisely, and sometimes is the only language which can convey those ideas at all;
- that asking questions is a sign of intelligence, not a sign of ignorance;
- that if you think hard enough about a topic (recursion, induction, logic, arithmetic, ...), and talk about it enough, the ideas will come to you;
- that you came to University to do difficult things (if you had wanted to do easy things, you could have stayed at home and read Ladybird books).

This course will be challenging. Challenging courses are what university study is all about.

Introduction to *Algorithms*?

‘Old Algorithm’ (Al-Khowezmi) invented long- (addition, subtraction, multiplication, division).

> *The numeral system used in Europe is still called ‘Arabic’ to distinguish it from the earlier ‘Roman’ stick-counting system.*
>
> **Numeral** *system, not* **number** *system. There’s a difference, and it’s important.*

Hence ‘a procedure which you can follow without necessarily understanding it’;

hence ‘a computer program for solving a problem’;

> *because computers can’t understand* ***anything***

hence ‘the design of a computer program for solving a particular problem in general’.

So our *algorithms* are actually *programs*, well-known and well-understood solutions to well-known problems.

Correctness and Efficiency

Our two concerns will be *correctness* and *efficiency* – roughly “what a program does” and “how easily it does it”.

In looking at correctness we shall be (semi-) formal when *specifying* what a program should do, and informal when *proving* that it does it.

In looking at efficiency we shall use arithmetic and algebra, very simply.

Our programs will be small. We shall look at the properties of

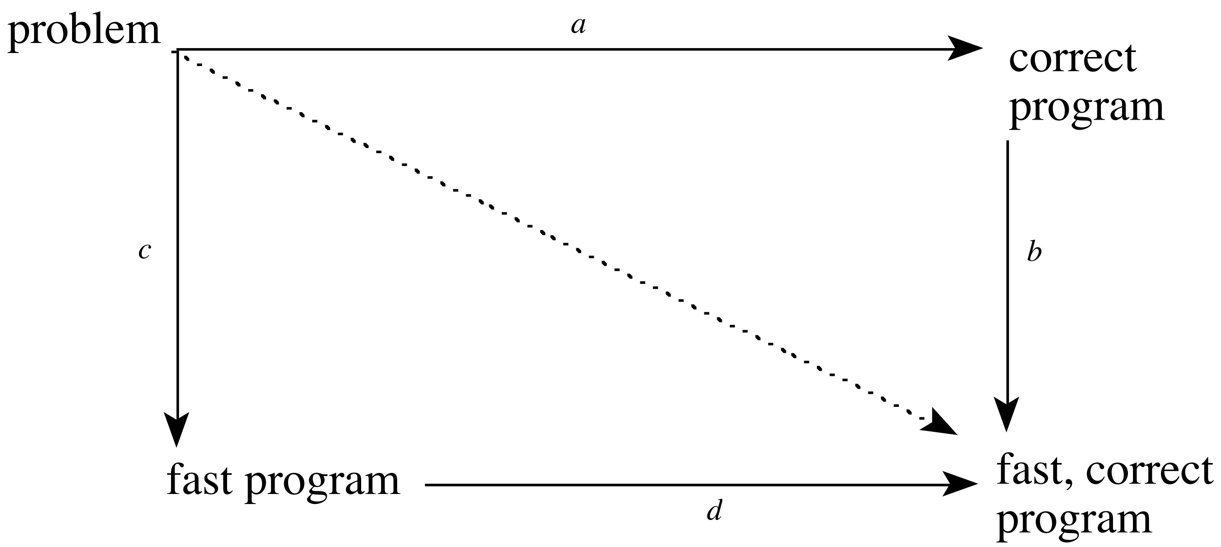
- assignment instructions;
- sequences of instructions;
- repetition (*for* and *while*)

plus, occasionally, recursion.

For the first half of the course, at least, we shall concentrate on programs that manipulate arrays.

Correctness matters more than efficiency

The famous Landin diagram shows various ways of making a fast correct program.



problem

correct program

fast program

fast, correct program

Computer scientists claim that route *ab* is easier/better/more effective than *cd*.

An example. Moving a section of an array.

I suppose that I have an array which looks like this:

✔	▲	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆

and I want to shift some of the elements (for example, all those from the one which contains ♣ up to the one which contains ➳) rightwards one position.

I might draw the effect on the array so:

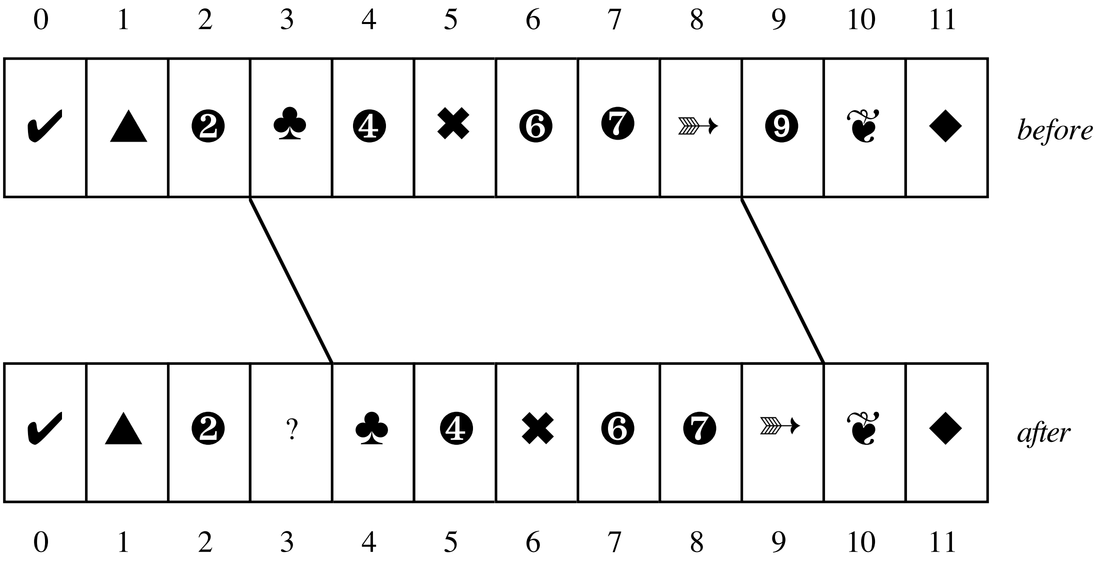
				♣	❹	✖	❻	❼	➳		

That picture doesn't say enough, because it doesn't talk about the rest of the array.

As well as moving those elements, I should like to leave the rest of the array unchanged. The effect I want is:

✔	▲	❷	?	♣	❹	✖	❻	❼	➳	❦	◆

Here’s a different version of the specification:



0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆	*before*
✔	▲	❷	?	♣	❹	✖	❻	❼	➳	❦	◆	*after*
0	1	2	3	4	5	6	7	8	9	10	11	

Numbering the elements will help to make it a little more abstract.

The picture *says nothing* about what happens outside the world of array elements 0 to 11.

> *that's a general problem with specifications, whether pictures, predicate calculus, whatever.*

The picture describes how things finish up (the 'after' picture) in terms of how they start out (the 'before' picture).

In technical terms, a *post-condition* and a *pre-condition*.

Specifying the array shifter in predicate calculus

I am going to write a single formula, combining pre- and post-condition for the time being.

We deal with the pre/post distinction by describing the ‘pre’ array as A, and the ‘post’ array as A'.

$$\begin{array}{l}
A'[0]=A[0]\wedge A'[1]=A[1]\wedge A'[2]=A[2]\wedge \\
A'[4]=A[3]\wedge A'[5]=A[4]\wedge A'[6]=A[5]\wedge \\
A'[7]=A[6]\wedge A'[8]=A[7]\wedge A'[9]=A[8]\wedge \\
A'[10]=A[10]\wedge A'[11]=A[11]
\end{array}$$

In specifications, the analogue of repetition is quantification.

We have two standard kinds of remark:

- $\forall j(i \le j < k \rightarrow P(A[j]))$ – read as ‘all the $A[j]$s such that $i \le j < k$ have property P’, or ‘ all the $A[j]$s in the range $i..k-1$ have property P’, or ‘all the elements $A[i], A[i+1], \ldots, A[k-1]$ share property P’;
- $\exists j(i \le j < k \wedge P(A[j]))$ – read as ‘at least one of the $A[j]$s such that $i \le j < k$ has property P’, or ‘ at least one of the $A[j]$s in the range $i..k-1$ has property P’, or ‘at least one of the elements $A[i], A[i+1], \ldots, A[k-1]$ has property P’.

The $\rightarrow$ in the $\forall$ remark, and the $\wedge$ in the $\exists$ remark, are not accidental. Review your knowledge of the predicate calculus if you don’t understand the distinction.

$$\forall i \begin{pmatrix} (0 \leq i < 3 \rightarrow A'[i] = A[i]) \wedge \\ (4 \leq i < 10 \rightarrow A'[i] = A[i-1]) \wedge \\ (10 \leq i < 12 \rightarrow A'[i] = A[i]) \end{pmatrix}$$

Technically this is a remark about every possible integer value of *i* (it says $\forall i$), which says something specific about values 0, 1, 2, 4, ..., 9, 10 and 11 and manages to say nothing specific about values less than 0, greater than 11 or exactly 3.

It *doesn't* say 'and that's all there is to say': there might be more.

the picture specification had the same problem ...

You should be well aware that there are two obvious guesses for the array shifting program

The guesses are:

i $A[4] = A[3]; A[5] = A[4]; \ldots\ A[9] = A[8]$

ii $A[9] = A[8]; A[8] = A[7]; \ldots\ A[4] = A[3]$

(i) <u>***doesn’t work***</u> but (ii) does, and the difference has to do with the properties of the assignment instruction.

Definition of the assignment instruction

The instruction $x = E$, pronounced x ***becomes*** E,

> *curse the inventors of C for using the symbol '=' to mean 'becomes', and curse the inventors of Java for copying them.*

has a simple mechanical reading: calculate the value of E and then put a copy in variable x.

It follows that the instruction $x = y$, where x and y are variables, makes x and y the same. So does $y = x$.

One effect of the assignment instruction $x = E$ is to *obliterate* the previous value of x – so $x = y$ obliterates x, and $y = x$ obliterates y.

> *We can't obliterate values, only copies of values.*

$x = E$ obliterates a single copy of a value – the copy stored in x – and $x = y$ makes a new copy of a value – the copy stored in y – as well as obliterating the copy which used to be stored in x.

> *these fine distinctions matter: there is a real difference between a copy of a book and the book itself.*

So $A[4] = A[3]$ makes a copy of the value in $A[3]$ and obliterates the copy of the value in $A[4]$, and if we need that value and have no other copies we are in trouble ...

That's what's wrong with program (i): $A[4] = A[3]$; $A[5] = A[4]$;K $A[9] = A[8]$. It obliterates each value $A[4], A[5],$K $, A[9]$ in turn, finally replacing them *all* with a copy of what was originally in $A[3]$. Its effect can be pictured:

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆	*before*

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	♣	✖	❻	❼	➳	❾	❦	◆	*after A* [4]=*A*[3]

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	♣	♣	❻	❼	➳	❾	❦	◆	*after A* [5]=*A*[4]

...

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	♣	♣	♣	♣	♣	♣	❦	◆	*finally*

By contrast, program (ii) obliterates only copies of values that aren’t needed.

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	➋	♣	➍	✖	➏	➐	➳	➳	❦	◆	*after A* [9]=*A*[8]

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	➋	♣	➍	✖	➏	➐	➐	➳	❦	◆	*after A* [8]=*A*[7]

.. and so on, until

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	➋	♣	♣	➍	✖	➏	➐	➳	❦	◆	*finally*

Note that the fourth position is unchanged from its initial value. Should we add that to the specification? I think not.

We would almost certainly write our solutions as repetitions:

i' `for (i=4; i<10; i++) A[i]=A[i-1];`

ii' `for (i=9; i>3; i--) A[i]=A[i-1];`

Unfortunately, program (i') – although it is wrong – looks as if it comes straight from the predicate calculus specification

$$\forall i \begin{pmatrix} (0 \le i < 3 \to A'[i] = A[i]) \wedge \\ (4 \le i < 10 \to A'[i] = A[i-1]) \wedge \\ (10 \le i < 12 \to A'[i] = A[i]) \end{pmatrix}$$

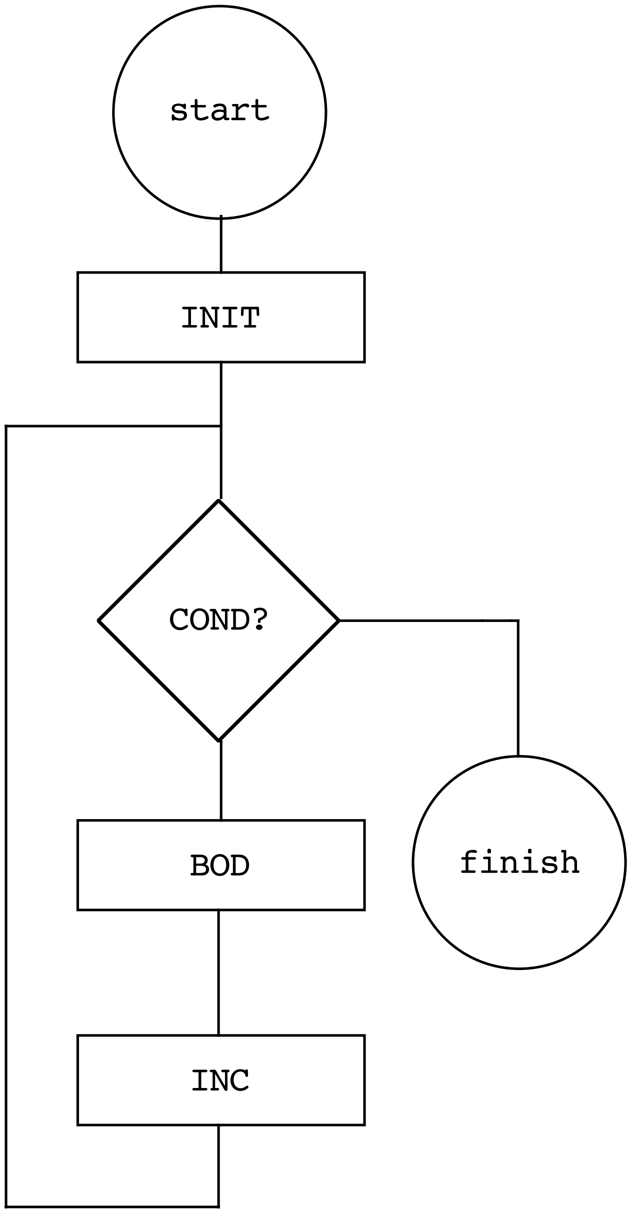
whereas program (ii') – which is correct – isn't such an obvious translation.

Definition of the *for* instruction

The *for* instruction

```
for (INIT; COND; INC) BOD
```

has a simple mechanical definition, which is easily given as a flowchart:



Every *for* is equivalent to the sequence `<INIT, COND, <BOD, INC, COND>*>`.

or to `<INIT, <COND,BOD, INC>*, COND>`

In the case of program (ii'):

```
i=9; if (i>3) { A[i]=A[i-1]; i--; ... }
```

from which we can deduce

informally, knowing that 9>3

that there will be a first execution of BOD, and that it will be equivalent to `A[9]=A[9-1]`.

In the case of program (i'):

```
i=4; if (i<10) { A[i]=A[i-1]; i++; ... }
```

and it's clear that the first instruction will be `A[4]=A[4-1]`.

It's fairly easy to see, from the mechanical definition, that the two programs (i) and (ii) are equivalent to (i') and (ii').

is it so easy? Can you do the calculations? How would you lay out your working?

The ‘cost’ of executing a program

Suppose that in order to execute a program you had to rent a computer.

You would have to pay for the time you used, and the memory space you used.

The same if you have to buy a computer.

We begin with the simplest example: the cost, in time and space, of the assignment instruction `A[i]=A[i-1]`.

Knowing the address of array `A`, and the address of variable `i`, what does a machine have to do to carry out the assignment?

```
MOVE i, r1          // copy i to reg 1
SUB %1, r1          // value of i-1 in reg 1
ADD %A, r1          // address of A[i-1] in reg 1
MOVE i, r2          // copy i to reg 2
ADD %A, r2          // address of A[i] in reg 1
MOVE (r1), (r2)     // copy from A[i-1] to A[i]
```

That takes three moves, two adds, one subtract.

A more efficient version:

```
MOVE i, r1            // copy i to reg 1
ADD %A, r1            // address of A[i] in reg 1
MOVE r1, r2           // address of A[i] in reg 2
SUB %1, r1            // value of A[i-1] in reg 1
MOVE (r1), (r2)       // copy from A[i-1] to A[i]
```

> *using autodecrement or autoincrement wouldn’t help, because it would cause an extra subtraction or addition.*

Modern computers are designed so that ***arithmetic and store access are constant-time operations***.

> *even quite recently that wasn’t the case. Many desktop machines, even ten years ago, didn’t have constant-time multiplication or division.*

From all this we can deduce that the time and space used by `A[i]=A[i-1]` ***doesn’t depend on the value of*** **i** ***or on any of the values stored in array*** **A**. It takes some constant *Ta* time and uses no additional space at all.

So the cost of executing program (ii) – $A[9]=A[8]; A[8]=A[7];$ K $A[4]=A[3]$ – is just $6 \times Ta1$ in time, where *Ta1* is the cost of each of the assignment instructions, and nothing at all in space.

> Ta1 *will be smaller than* Ta. *Can you see why?*

The cost of executing program (ii') –

```
for (i=9; i>3; i--) A[i]=A[i-1];
```

is a little more because it uses a *for* instruction and a more complicated assignment.

It consists of the cost of executing six assignment instructions from the body (each costing *Ta* in time and nothing in space), plus the cost of the initial assignment `i=9`, plus the cost of the initial test `i>3`, plus the cost of six increments `i--` and six more tests `i>3`, plus the cost of six or seven jumps.

> *Can you see where the jumps come from? Look at the flowchart.*

Program (ii') also uses more space than program (ii), because it needs the variable *i*.

Over all, program (ii') will probably take about two or three times as long to execute as program (ii), and uses one extra memory location.

> *I'm neglecting the cost of the memory required to hold the program: program (ii') will probably be a good deal smaller program (ii). But that's nitpicking: neither of them uses very much space.*

But program (ii) is a particular solution to a specific problem. Program (ii') points to a general solution to a general problem.

I want to write – and cost – a program which moves a segment $A[m \ldots n-1]$ of an array rightwards.

> m *and* n *are the* ***<u>parameters</u>*** *of the problem, defining its size.*

iii `for (i=n; i>m; i--) A[i]=A[i-1];`

When $m \geq n$ this does a fixed amount of work (one assignment, one test, one jump); when $m < n$ it does that same fixed amount of work, plus exactly $m - n$ executions each of the body, the increment, the test, and a jump.

Ignoring the fixed work, which is small, this program’s costs are proportional to $m - n$ in time, and are one variable in space.

> *You have just seen your first ‘linear’ cost program.*

There isn’t a faster program for this problem,

> *you might like to persuade yourself of that fact*
>
> *you should be aware that your argument might depend on the kind of machine you imagine ...*

which is to say that the cost of shifting things around in an array is ‘linear’ – that is, proportional to the number of things we are shifting around.

A different array-shifting problem

I want to move all the elements which are in *even-numbered positions* to the right-hand end of the array, preserving their order. I don't care what happens to the odd-numbered positions.

Here's a picture of the specification for the particular case of a twelve-element array:

0	1	2	3	4	5	6	7	8	9	10	11	
✔	▲	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆	*before*

?	?	?	?	?	?	✔	❷	❹	❻	➳	❦	*after*
0	1	2	3	4	5	6	7	8	9	10	11	

I'm first going to write the program just using the picture, without writing the predicate calculus specification.

> *Don't suppose from this or any other example that I think that pictures are better or worse than predicate calculus. It's just that in this case, writing the predicate calculus gives the game away!*

The program will work left-to-right.

First I want my program to move the '✔' value right one position, to lie next to the '❷' value.

Then I want it to shift the '✔,❷' segment right one position, to lie next to the '❹' value.

Then I want to move '✔,❷,❹' right one position, to lie next to the '❻' value. And so on.

My program does its work on an array of size *n*, which can be either even or odd:

```
iv  for (j=0,k=1; k<n; j++,k+=2) {
      for (i=k; i>j; i--) A[i]=A[i-1];
    }
```

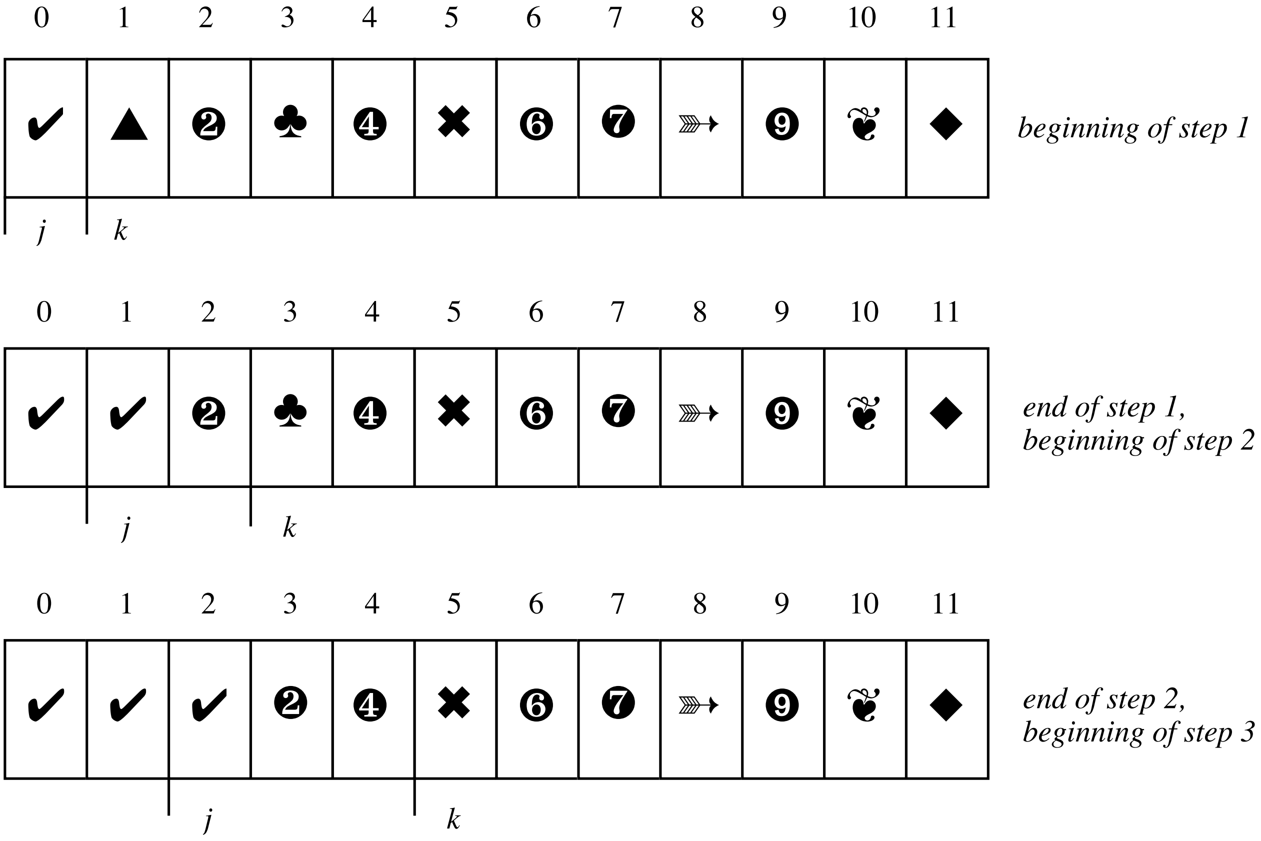
The inner loop is just program (ii'), shifting the segment $A[j..k-1]$ rightwards one position.

I've re-used what I've already constructed – that's a principle of program design. It doesn't always lead to a good solution ...

After each step *j* moves up one, following the segment, and *k* moves up two (one to follow the segment, one to add the next element).

Note that k *moves through the odd numbers. Does this happen with an odd-number-sized array?*

On the twelve-element array we can see how this program has the correct result:



0	1	2	3	4	5	6	7	8	9	10	11
✔	▲	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆
j	*k*										

beginning of step 1

0	1	2	3	4	5	6	7	8	9	10	11
✔	✔	❷	♣	❹	✖	❻	❼	➳	❾	❦	◆
	j		*k*								

end of step 1,
beginning of step 2

0	1	2	3	4	5	6	7	8	9	10	11
✔	✔	✔	❷	❹	✖	❻	❼	➳	❾	❦	◆
		j			*k*						

end of step 2,
beginning of step 3

... and so on. This program is indeed stamping on the elements that it leaves behind.

The elements that are to be altered in the twelve-element example are $A[6\mathrm{K}\ 11]$; the predicate calculus specification is therefore bound to be $\forall i(6 < i < 12 \rightarrow ..\text{something or other}..)$.

In longhand the specification is

$$A'[6] = A[0] \wedge A'[7] = A[2] \wedge A'[8] = A[4] \wedge$$
$$A'[9] = A[6] \wedge A'[10] = A'[8] \wedge A'[11] = A[10]$$

and it doesn't take infinite ingenuity with arithmetic to spot that this is equivalent to

$$\forall i(6 \leq i < 12 \rightarrow A'[i] = A[(i-6) \times 2])$$

Then for an *n*-element array we have

$$\forall i(n \div 2 \leq i < n \rightarrow A'[i] = A[(i - n \div 2) \times 2])$$

Those with a good grasp of arithmetic

remembering that $(n \div 2) \times 2$ *isn't always n*

may like to check this specification carefully, to be sure that it's valid whether *n* is odd or even.

The interesting thing about the predicate calculus specification is that it looks just like an array shift, so there ought to be a program based on it.

And, of course, there is. It’s a rightwards shift, so we start at the right-hand end:

```
V  for (i=n-1; i>=n/2; i--) A[i]=A[(i-n/2)*2];
```

This program isn’t quite as fast as it might be, but what is interesting is that it is much faster than program (iv), and its advantage increases as the problem gets larger (i.e. as *n* increases).

program (ii') had $i > 3$ as its **COND***; program (v) has $i \geq n \div 2$. Why does one use ($>$) and the other ($\geq$)?*

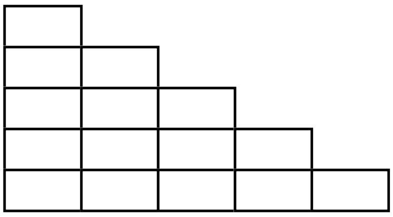
The time taken by program (iv) is ‘quadratic’ in *n*, and the time taken by program (v) is linear in *n*.

We have already seen that the time taken by program (ii') is roughly proportional to the size $k - j$ of the segment it shifts.

Program (iv) executes program (ii') repeatedly, the first time with $k - j = 1$, the second with $k - j = 2$, ... , the last time with $k - j = (n \div 2) - 1$.

You should know that the sum of a series $1 + 2 + ... + x$ is $x(x - 1)/2$, which is $x^2/2 - x/2$, which is roughly proportional to x^2.

Another way of looking at is that the execution times form a triangle:



... and so on

and everybody knows that the area of a triangle is half that of a rectangle ...

So the time taken by program (iv) is roughly proportional to $(n \div 2)^2$, which means it's roughly proportional to n^2.

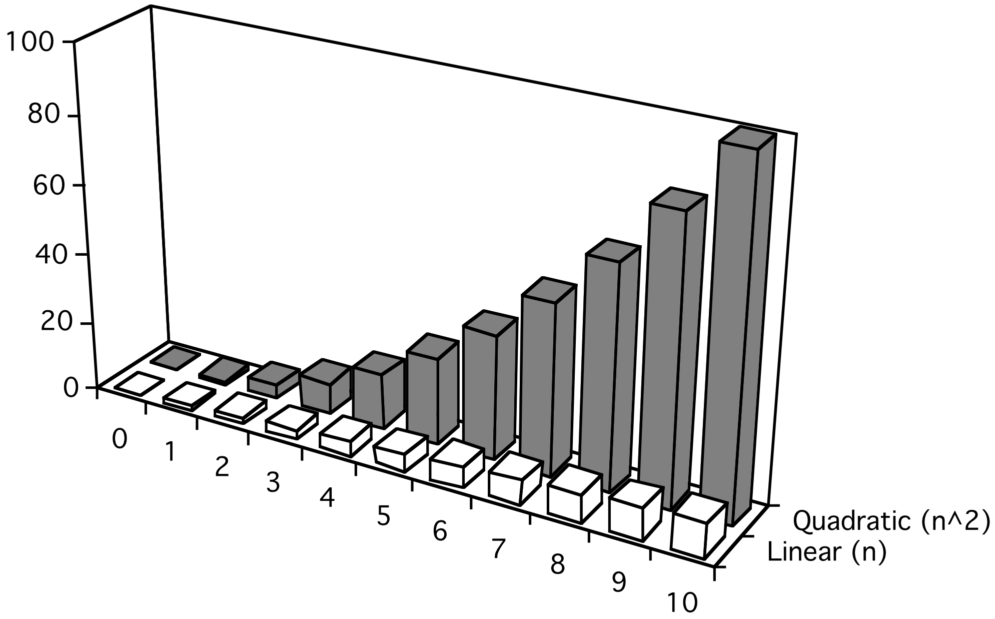
But the time taken by program (v) is roughly proportional to n

by an argument just like that about program ii'

By experiment, in the lab, with large examples you should be able to demonstrate that this argument has hit on a truth: program (v) is *much faster* than program (iv). The fact that it uses only one variable instead of three is just icing on the cake.

Linear vs. Quadratic time.

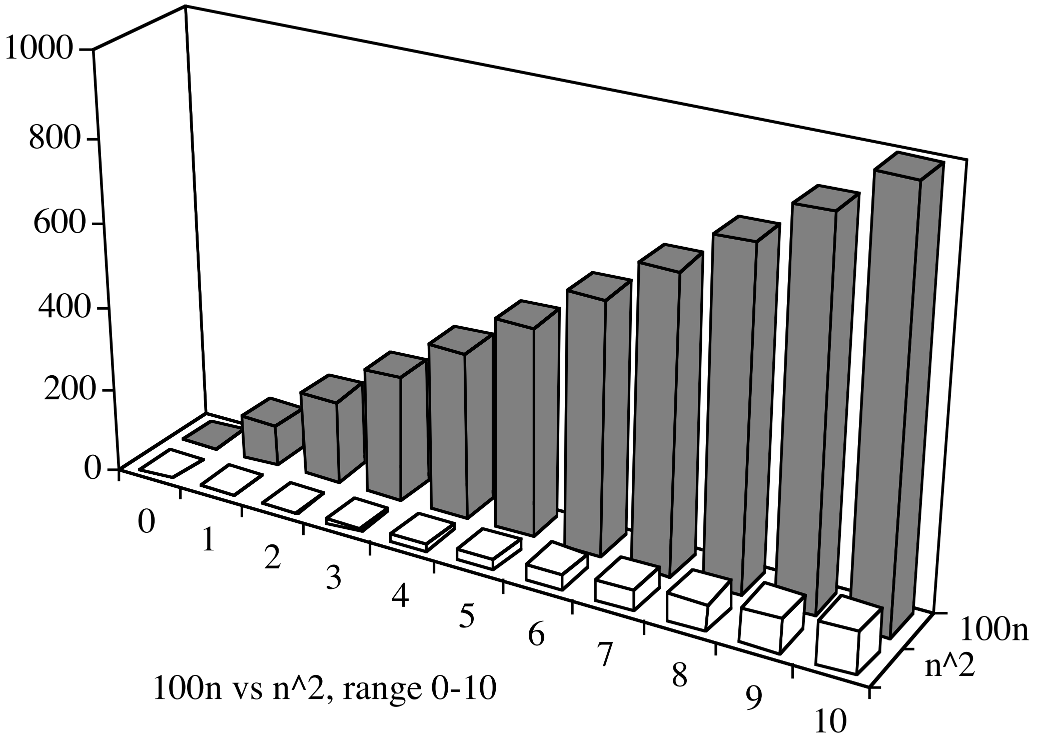
This diagram shows why linear-time programs are, in general, to be preferred to quadratic-time programs.



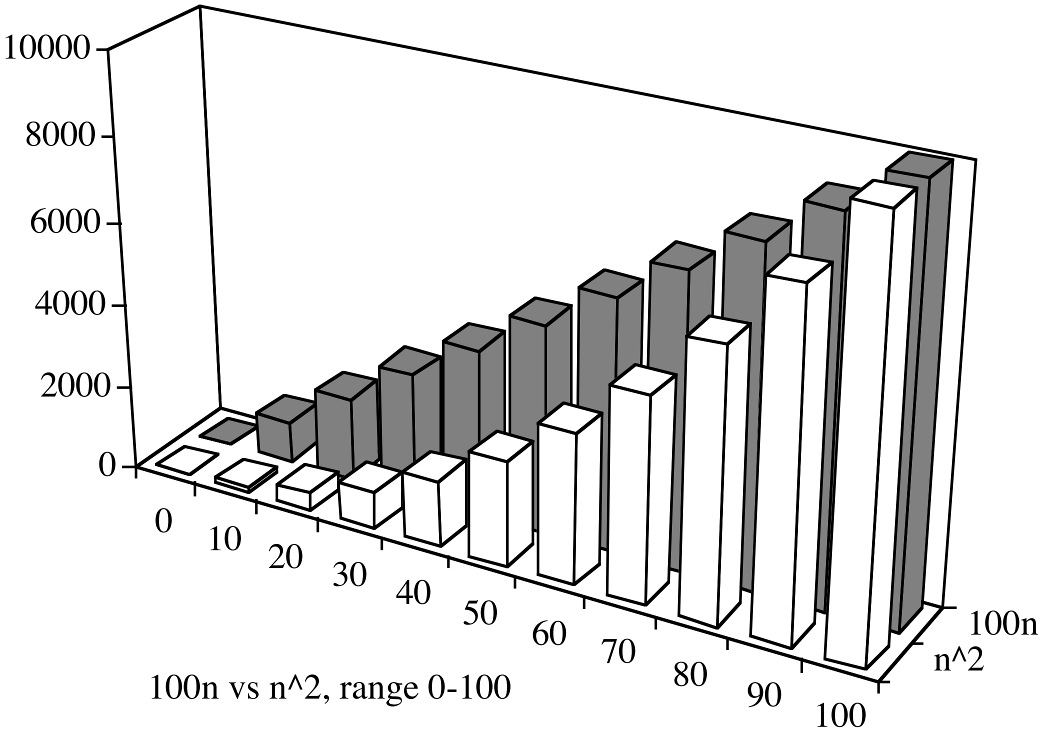
The execution time of a quadratic-time program grows much faster than that of a linear-time program.

__It doesn’t matter how steeply the linear graph is sloped__, because the quadratic graph grows more steeply at every step. Eventually it will grow more steeply than and will overtake the linear graph.

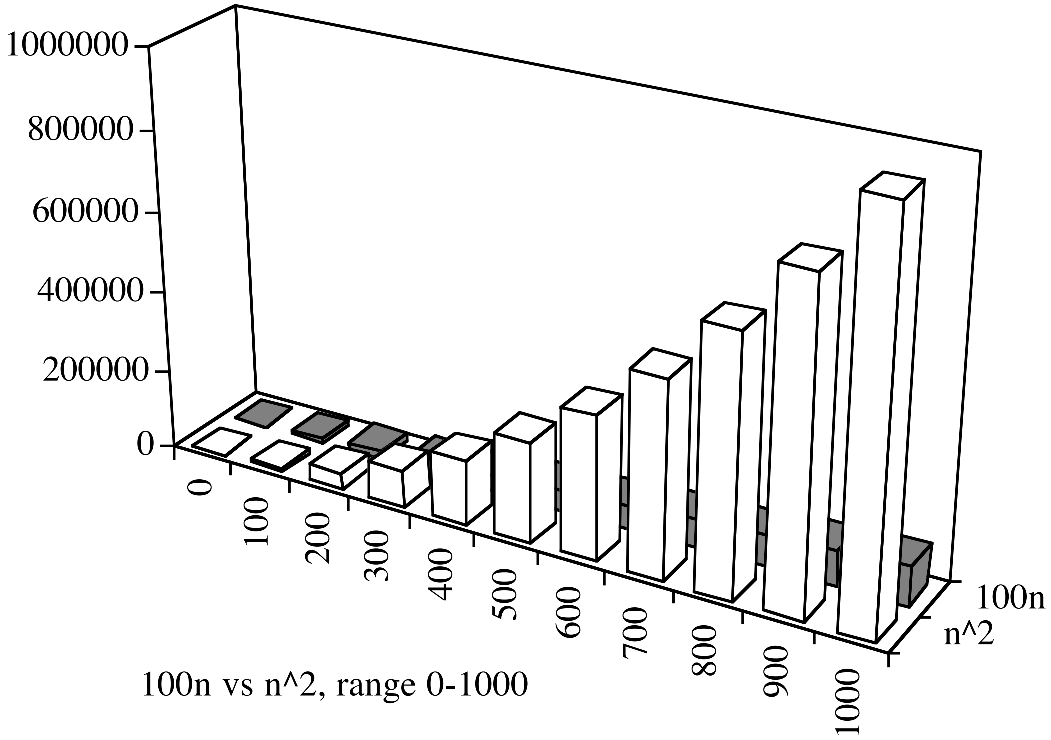
Three diagrams of of 100n (linear) vs n^2 (quadratic):



100n vs n^2, range 0-10



100n vs n^2, range 0-100



100n vs n^2, range 0-1000

If the linear and quadratic graphs cross at some problem size *M*, then at 10*M* the quadratic-time program *must* be 100 times slower than the linear-time program.

When one program is quadratic and the other linear, it is a complete waste of effort to speed up the quadratic program.

> <u>unless</u> *you are sure that the quadratic time program is faster on small problems and that it will always be used on small problems.*

Speeding up the solution

One speed-up is to take repetitive calculations outside a loop:

```
v'  mid = n/2;
    for (i=n-1; i>=mid; i--) A[i]=A[(i-mid)*2];
```

and again:

```
v'' mid = n/2;
    for (i=n-1,j=(n-1-mid)*2; i>=mid; i--,j-=2)
      A[i]=A[j];
```

do these change make a significant difference?

Notice, by the way, that once we had a correct program (program (v)), it was easy to speed it up. The hard part was the arithmetic, and that had all been done by that stage.

Key points

technical language matters.

programming is about minute detail.

correctness; efficiency; correctness vs efficiency.

predicate calculus specifications about arrays.

assignments obliterate as well as copying.

rightward shifts start at the right.

rouigh estimates of cost can tell us a lot.

array shifting is a linear problem.

the area of a triangle is half that of a rectangle.

a quadratic-time program is much slower than a linear-time program, given a sufficiently large problem.