E-GENTING PROGRAMMING COMPETITION 2003

PUBLIC LECTURE

10 JANUARY 2004

LECTURE NOTES

Fourth draft
Jonathan Searcy
9 January 2004

Table of Contents

1 INTRODUCTION

1.1 Format of the Lecture

Ladies and Gentlemen, welcome ...

This afternoon we will have some fun discovering how to solve last year's programming competition questions.

The programming competition questions posed four problems. The contestants could attempt one or more problems. The questions had varying levels of difficulty.

The questions were:

Question	Description	Marks
1.	Microbank A commercial reporting program with performance requirements.	200
2.	M44 A problem in data structures and algorithms and spatial geometry.	400
3.	Windscreen Wipers Development of a state machine to drive a stepping motor mechanism	400
4.	25 Factorial A problem in extended precision arithmetic	100

The most interesting problem is M44, so I will start with that.

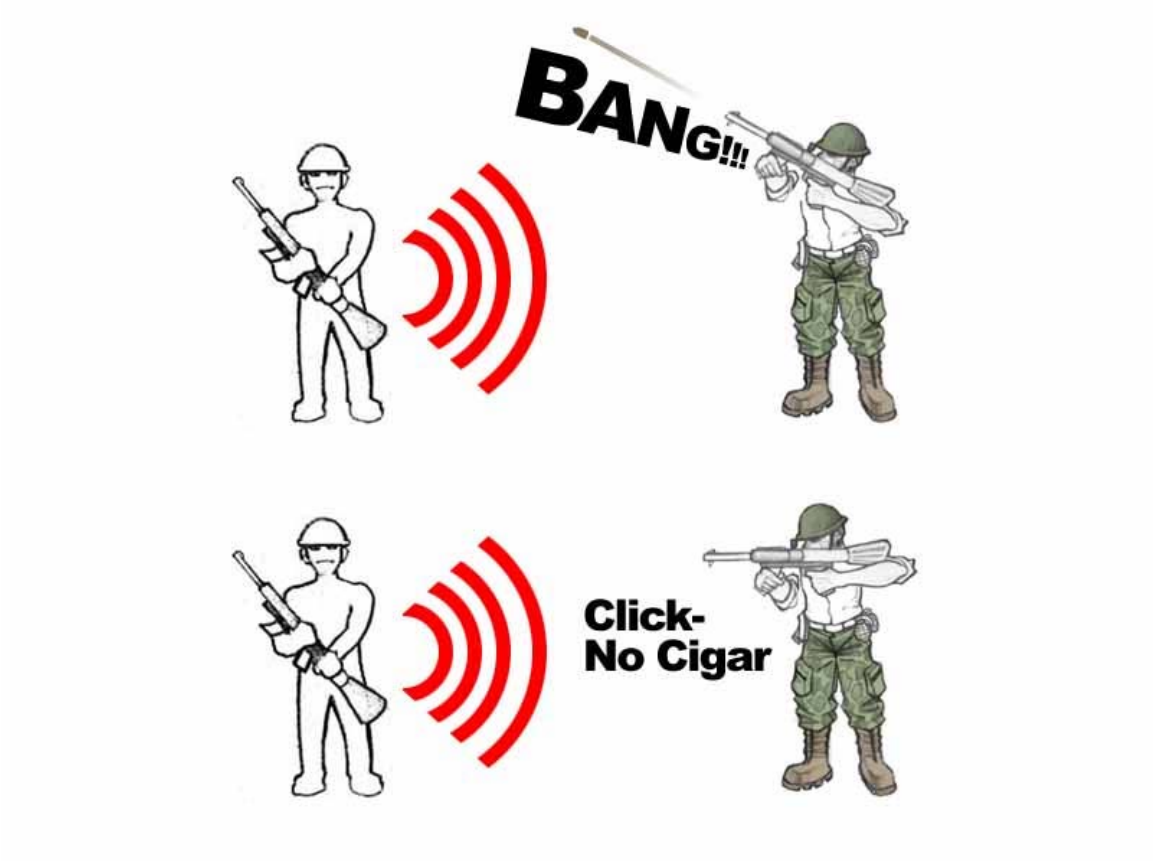
After figuring out how to solve M44, I will take a short break and give Mr Goh Boon Yeow the stand to deal with the Microbank problem.

After Boon Yeow has sorted out Microbank, I will come back and explain how to solve 25 Factorial and Windscreen Wipers.

2 M44

2.1 Preamble

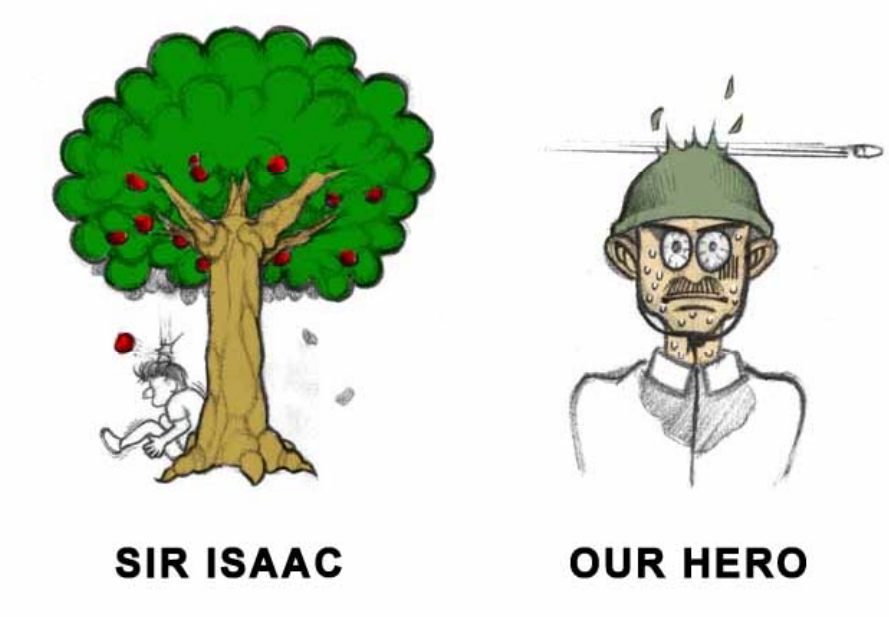
The idea of the M44 was to create a failsafe rifle that would refuse to fire when it was pointing at the holder of another M44.



Each M44 emits a signal that is sensed by every other M44. When the M44 receives a signal from another M44, it stores the relative location of the other M44 in an internal database. If the M44 is not pointing at another M44, it will fire, but if it is pointing at another M44, click – no cigar.

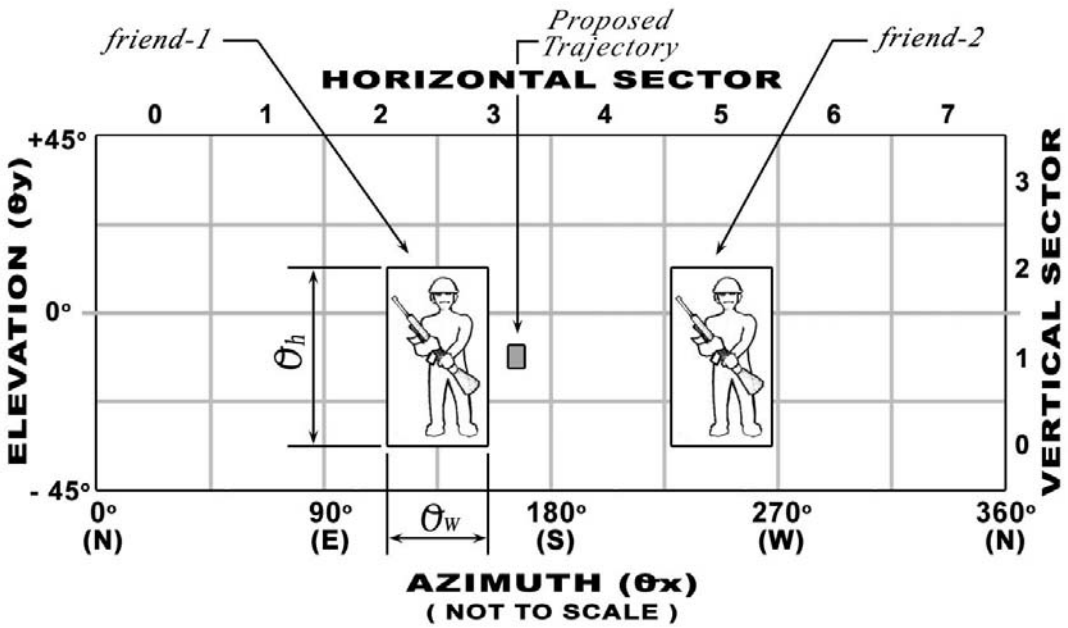
The original M44 simply stored the locations of all the other M44s in an array and executed a simple sequential search to identify conflicts, but this created a problem. When there were a large number of other M44s in the vicinity, it could take several seconds to search the array – more or less the life expectancy of a soldier using the weapon

2.2 A Near-Newtonian Experience



In what was referred to as a 'near Newtonian experience with a high velocity sniper round', our hero came up with the idea of dividing the search field into sectors and only searching the sectors that the trajectory of the bullet was likely to enter. Let's look at this in more detail.

2.3 Azimuth-Elevation Co-ordinate System

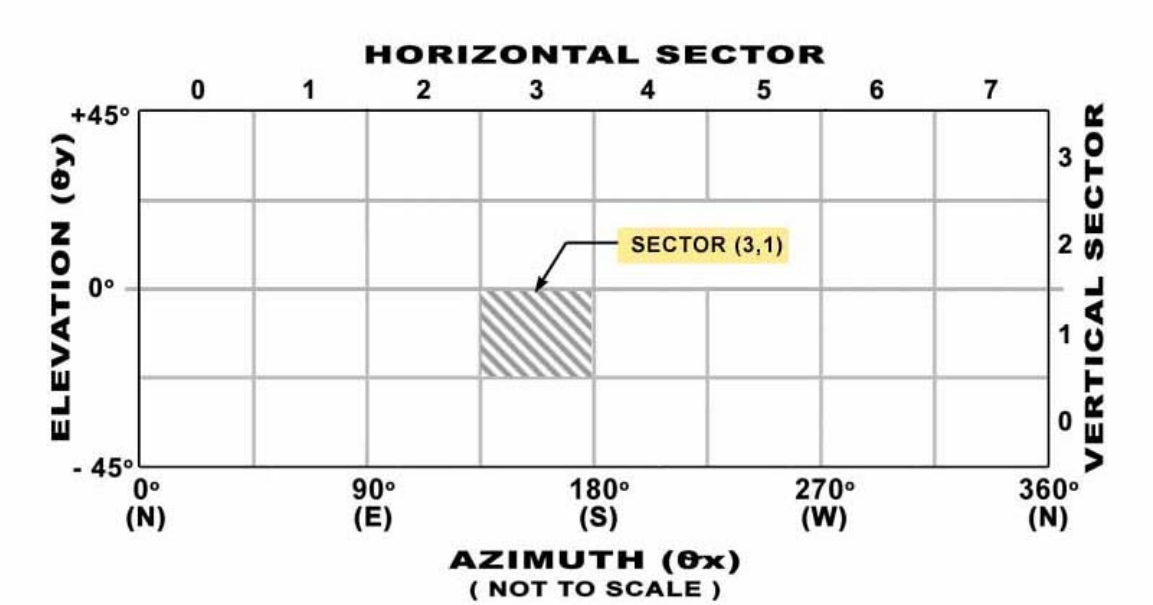


To solve the problem we have to deal with several changes in the co-ordinate system, but these changes are incidental to the problem. The pseudo-code in the question paper explains how to do the co-ordinate system transformations, though you might need a rudimentary understanding of trigonometry to make sense of it.

In the end, the field-of-fire is represented in an azimuth-elevation co-ordinate system. Azimuth is the direction in which the gun is pointing, in degrees east of North. Elevation is the angle between the gun barrel and the horizon.

The profiles of other soldiers holding M44s, which we call 'friends', are represented as rectangles in the azimuth-elevation co-ordinate system. A proposed firing trajectory is also represented by a rectangle.

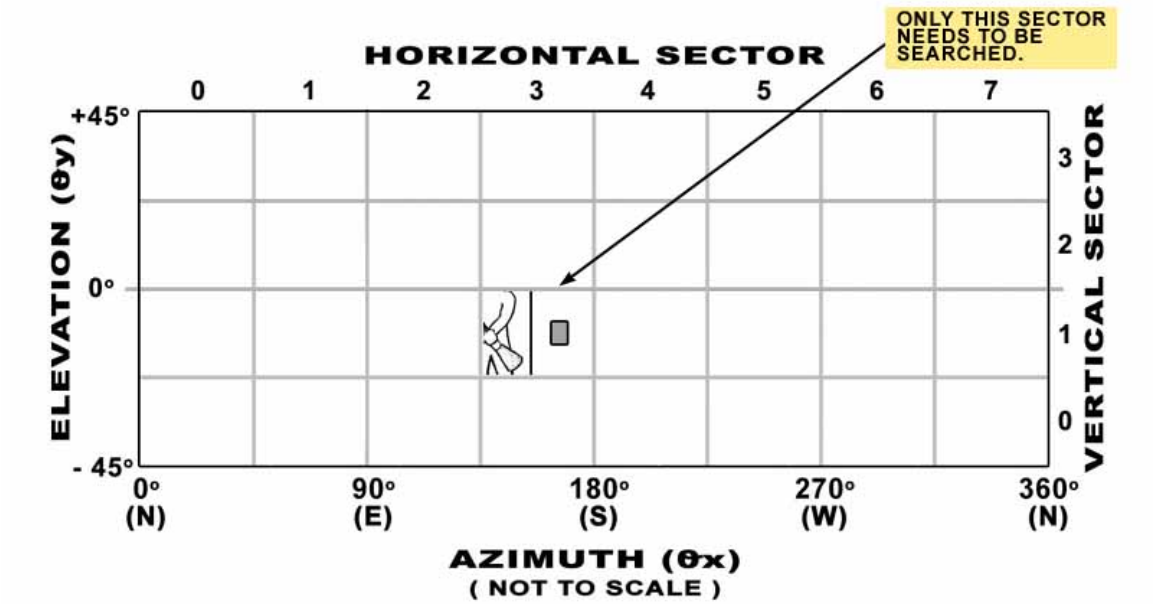
2.4 Sectors



The Nanopian Engineer suggests that the field-of-fire be divided into sectors in both the horizontal and vertical directions. In the azimuth-elevation co-ordinate system, a sector is represented by a square between the grid lines.

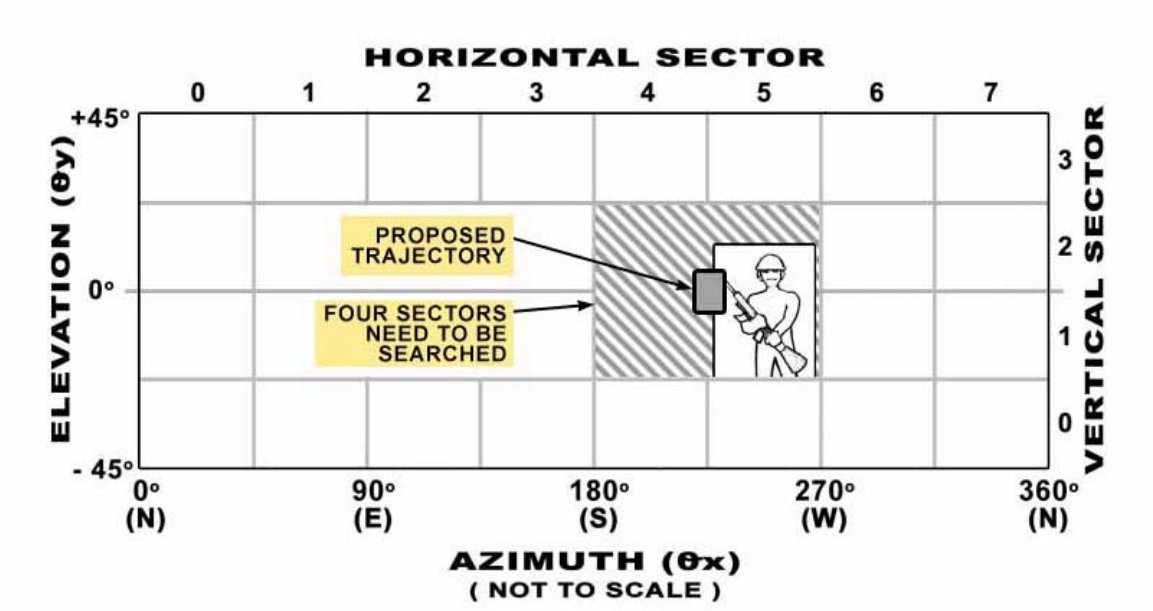
Each sector has a horizontal sector number and a vertical sector number. For example, here is sector (3,1).

2.5 Restricting the Extent of the Search



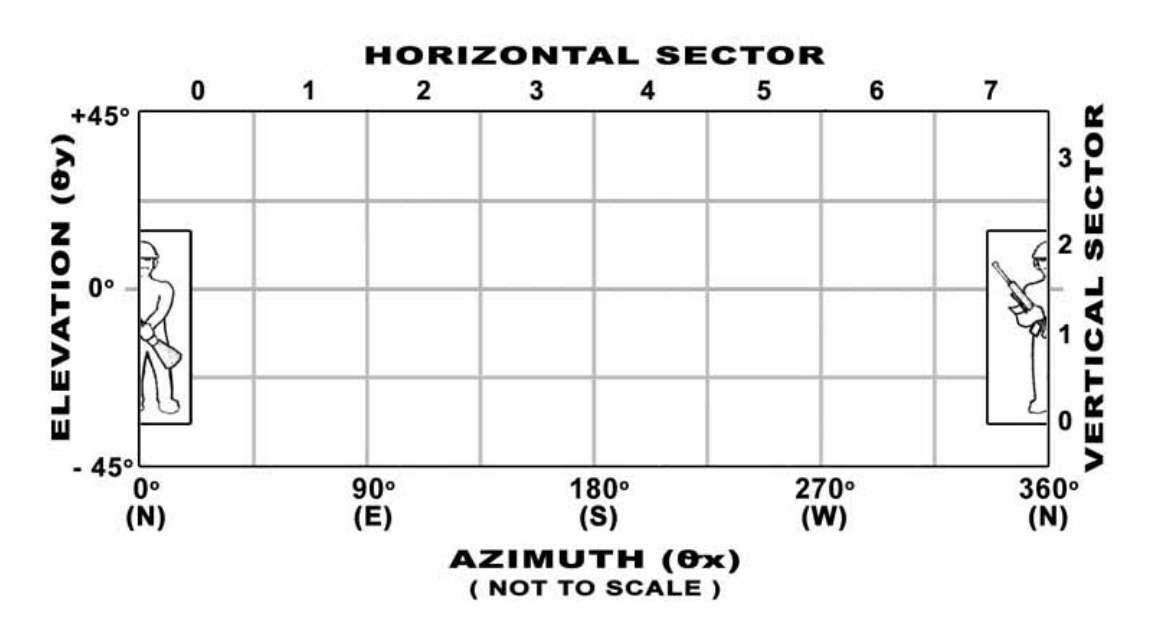
The program need only search the sectors that the proposed trajectory enters to ensure that no friends will be hit. In the example in the question paper, the program need only search sector (3,1).

2.6 Multiple Sectors may need to be Searched



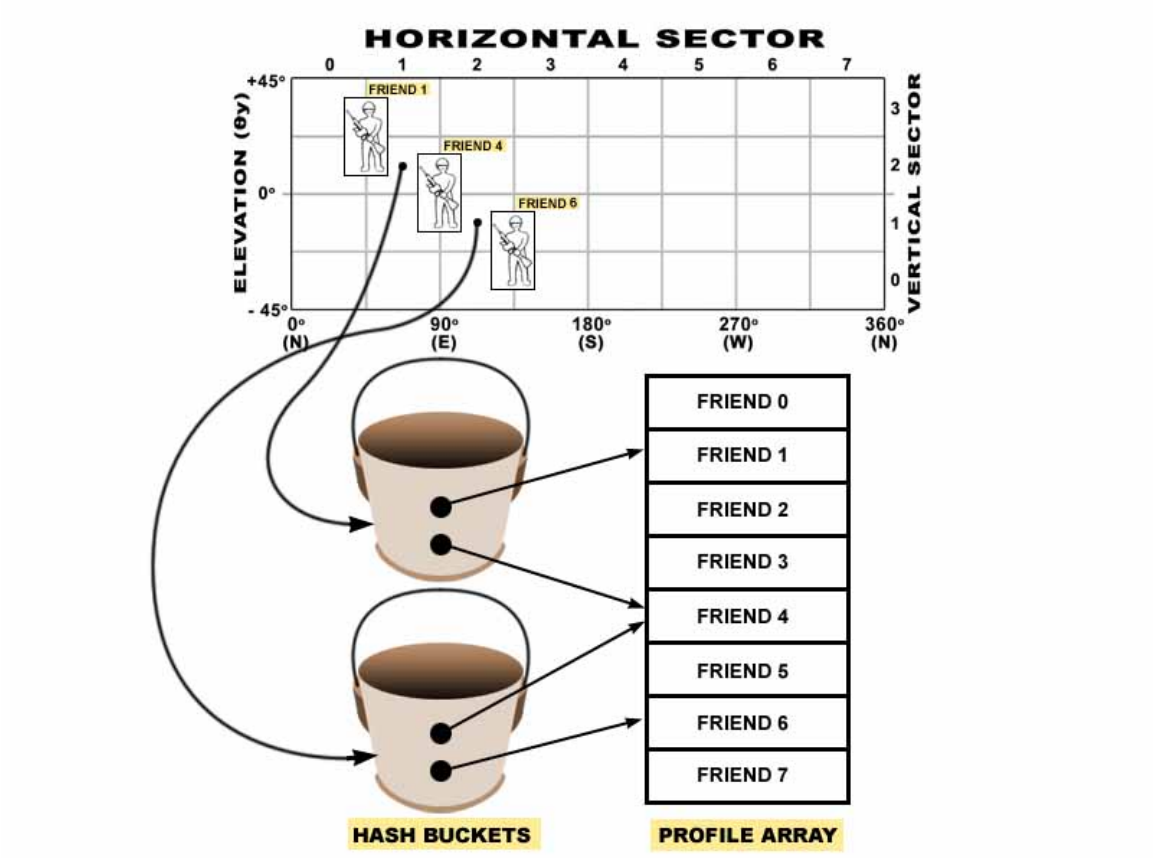
Although the example in the question paper only involved a search of one sector, depending on the direction of the proposed trajectory, several sectors might need to be searched.

2.7 Azimuth is a Cyclic Domain



A further complication is created by the circular nature of the horizontal direction. A soldier north of the origin appears in both the low-numbered and high-numbered sectors.

2.8 Bucket Data Structure



When I was studying Mechanical Engineering in 1976, we were taught that to solve an engineering problem, almost invariably, the first thing to do is to draw a picture. This rule extends to Software Engineering as well. It's just that instead of drawing pictures of beams and gears, we draw pictures of data flows and data structures.

This picture of the bucket data structure is quite easy do derive from the description of the structure in the question paper.

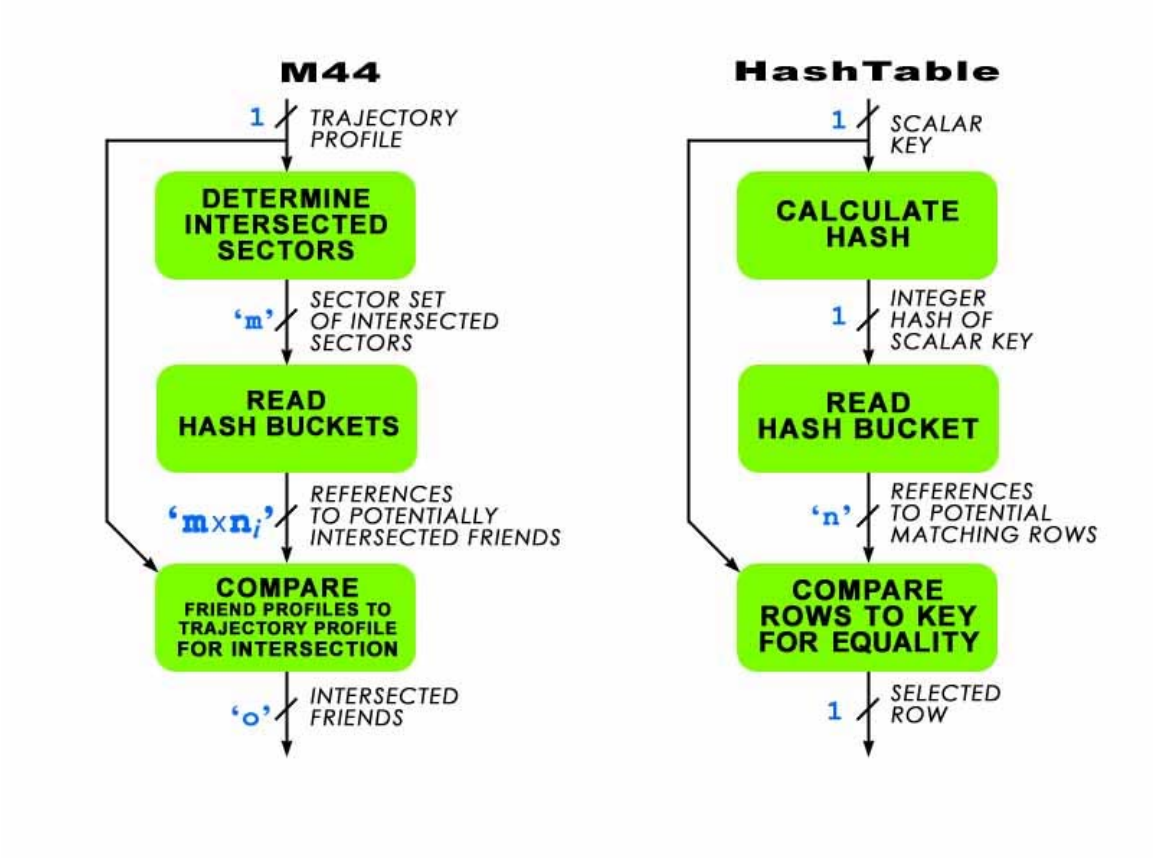
The paper says that 'a hash-bucket [should] be created for each combination of horizontal and vertical sectors'. So, we draw a picture of a hash-bucket for a sample of the horizontal and vertical sector combinations.

The paper says that 'a reference to each friend [should] be put in the hash bucket of each sector that the profile of the friend intersects'. So we draw a reference from each bucket to the row that contains the profile of the friend.

This diagram is merely a stylised entity relationship diagram.

Now, when we want to determine the friends that might be hit by a shot fired into sector (1,2), we can look in bucket (1,2) to find the references to the friends in sector (1,2) and instead of checking eight friends, we only need to check two.

2.9 A Common Mistake



All the competitors who attempted to solve the M44 problem in the competition fell for a classic trap. The question paper referred to the bucket data structure as a 'hash-type algorithm'. The competitors immediately assumed that they could use the Java *HashTable* class to solve it.

Let's compare the data flows of the M44 bucket data structure with those of the Java *HashTable* data structure. This slide is a side-by-side comparison of the row selection operations. The high-level similarities are obvious, but can we use the *HashTable* class to solve the M44 problem? To answer this question we must consider the detail of what the two operations do and how they work.

The first phase of analysing a trajectory profile is to convert a two-dimensional rectangle into a 'sector set'. In this case, the sector set is the set of sectors that the trajectory profile intersects. As we saw before, a trajectory profile can intersect multiple sectors, hence the term 'sector set'.

In comparison, the first phase of a conventional hash-table lookup is to convert a scalar key into integer hash of the scalar key.

In the M44, the trajectory profile is a two-dimensional key and the sector set is a set of two-attribute tuples. In the *HashTable* class, the input is a scalar key and the hash value is an integer.

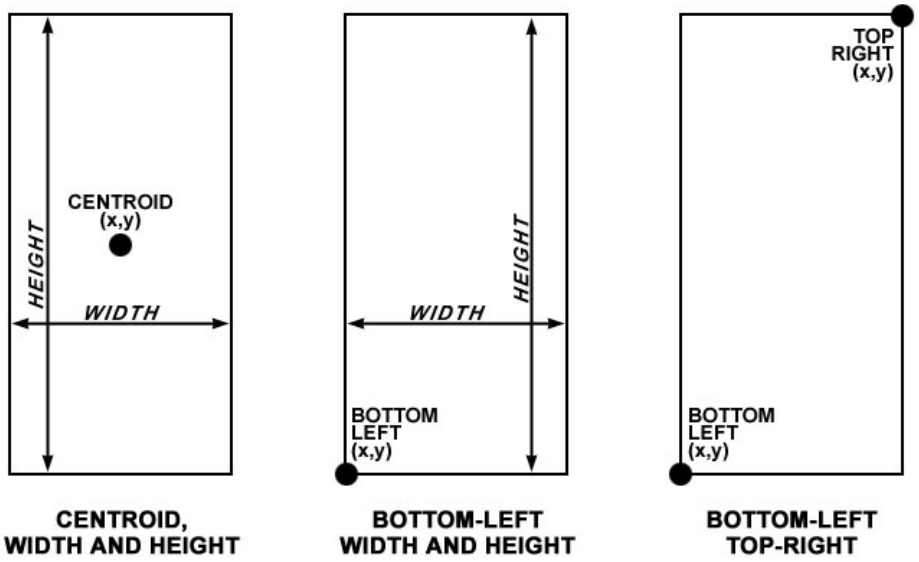
In the last phase, the M44 program must compare the friend profiles in the hash buckets to the trajectory profile and test for intersection. Multiple friends may be intersected by the trajectory.

In comparison, a conventional hash table compares the scalar key with the keys of the rows in the bucket and returns a single selected row.

These differences in the first and last phases make the HashTable class quite unsuitable for implementing a solution to the M44 problem.

In short, as is often the case, if we try the shortcut, we get the run-around. M44 is a no-*HashTable* zone. M44 involves the development of a special-purpose data structure and algorithm.

2.10 Profile Data Type



It is clear that we are going to need to store and process a compound data type that will contain the dimensions of a profile. This slide displays some alternatives.

The first form stores the x-y co-ordinates of the centroid and the width and height of the profile. It is the structure of the input data. The question paper explains how to calculate these dimensions.

The second form is the form of the Java *Rectangle* class.

The third form is the most convenient form for evaluating intersections and generating sector sets.

We must now choose between the three alternatives.

2.11 How will the Profile Data Type be used?

Usage	Centroid, height and width	Bottom-left, height and width	Bottom-left and top-right
Load from input data	Trivial	Easy	Easy
Generate sector set	Very messy	Messy	Clean
Test for intersection	Very messy	Messy	Clean

It is trivial to load the centroid form from the input data because it is the same form as the input data, but it is very messy to use this form for generating sector sets because half the height must be continually subtracted and added to the centre point to find the extremities of the profile.

It is not hard to load the bottom-left, width and height form from the input data, but it is messy to use because the width and height must be added to the bottom-left point to get the top and right extremities.

Again, it is not hard to load the bottom-left and top-right form from the input data. It is a one-off conversion of the centroid form into the extremity form, but this form is clean to use because physical extremities are available for direct comparison. It is by far the best choice.

But, you might say, what about the Java *Rectangle* class. The *Rectangle* class is inappropriate because the elements of the *Rectangle* class are integers whereas the fields in our *Profile* data structure must be floating point numbers. Not only is M44 a no *HashTable* zone, it's a no *Rectangle* zone too. M44 is an exercise in programming, not an exercise in playing around with Lego.

2.12 Domains of the Profile Components

$left \le x(pointInProfile) < right$
$bottom \le y(pointInProfile) < top$
$0° \le left < 360°$
$left \le right < left+5.73°$
$-45° \le bottom < +45°$
$bottom \le top < +45°$

Let us consider the fine details of the domains of the Profile components.

To avoid abutting profiles actually intersecting, we should make the domain of a point inside a profile include the left and bottom boundaries, but exclude the top and right boundaries. In a practical sense it's not mandatory, but in it adds a certain elegance to the design.

In order to guarantee that the intersection algorithms will work, we need to restrict the domain of the left boundary to zero to 360 degrees, with zero included and 360 excluded.

A little quick arithmetic will determine that the maximum width of a profile is less than 5.73°. This is the width of a friend the minimum distance from the origin.

If we allow the right boundary to exceed 360° so that it always greater than the left boundary we will greatly simplify the detection of intersections.

Because the test data are restricted to centroid elevations between –30 and +30 degrees, we do not need to concern ourselves with wrapping in the vertical direction.

2.13 The Test for Intersection

In an non-cyclic domain, if a profile is not:

above the other profile; or

below the other profile; or

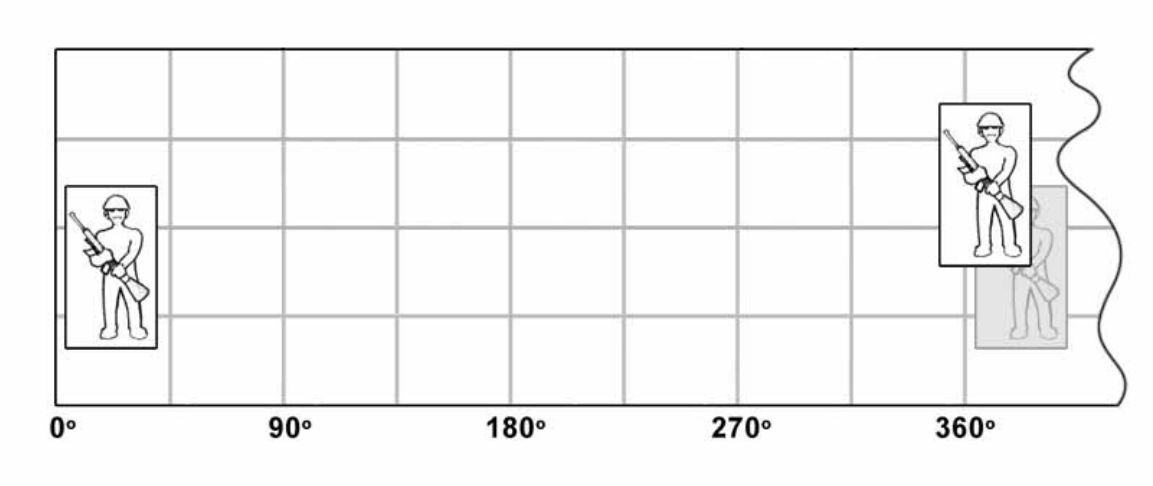
left of the other profile; or

right of the other profile

then it intersects the other profile.

And if you cannot convert that into code, then what are you doing here?

2.14 But Azimuth is a Cyclic Domain



In the above display, if we apply the simplistic non-cyclic formula for the intersection of two rectangles, we will conclude that the two profiles do not intersect.

But if we draw in the first repetition of the first friend a full turn to the right of his normalised position, we can see that there is an intersection.

Because the left boundary is restricted to range 0° to 360° we need only consider the first repetition of each profile in order to determine whether or not the profiles intersect in the cyclic domain.

2.15 The Expression for Intersection in the Cyclic Domain

If:

! NonCyclicIntersect (profile, otherProfile) and

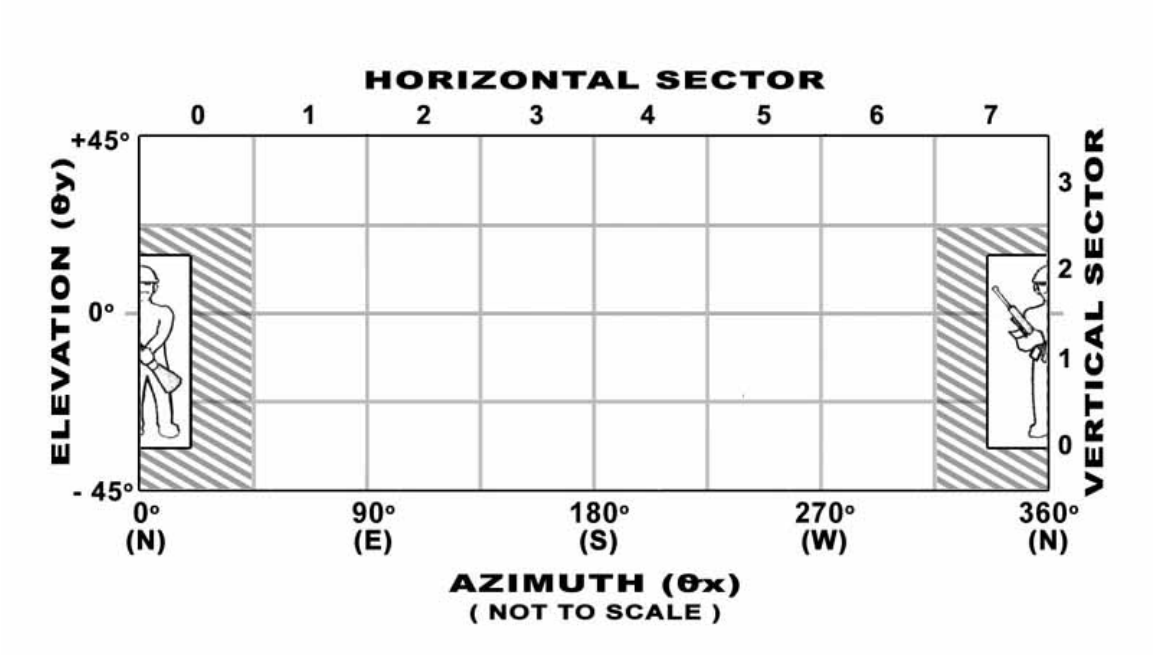
! NonCyclicIntersect (profile, otherProfile + 360°) and

! NonCyclicIntersect (profile + 360°, otherProfile)

then profile does not intersect otherProfile in the cyclic domain.

Putting the preceding concept into a form suitable for programming we can write this.

2.16 Sector Set Data Type



We must now consider the structure of a sector set. The sector set structure will be used to select the buckets into which references to a friend's profile must be placed. It will also be used to determine the buckets that must be scanned when a trajectory profile is being tested.

The most difficult situation occurs when a profile is to the north of the origin. In this case the sector-set must wrap from the right to the left.

2.17 How will the Sector Set be used?

Usage	Bottom-left, height and width	Bottom-left and top-right
Load from profile	Slightly harder	As easy as it gets
Expansion into sectors	Not too hard	Somewhat tricky

This is an analysis table similar to the one we used to select the best form for the profile data type.

Both these forms could, potentially be used. I prefer the height and width form because it facilitates the use of a for-loop. If you use the top-right form the top-right position must be the sector number of the right most column inside the sector set and the exit test needs to be in the middle of the loop. I think this expansion is messier so I chose the first form in my published example.

2.18 Converting a Profile into a Sector Set

```
sectorSet.left = (int)(profile.left / SECTOR_WIDTH);
right = (int)(profile.right / SECTOR_WIDTH);
if (right * SECTOR_WIDTH != profile.right) right ++ ;
sectorSet.width = right – sectorSet.left;
```

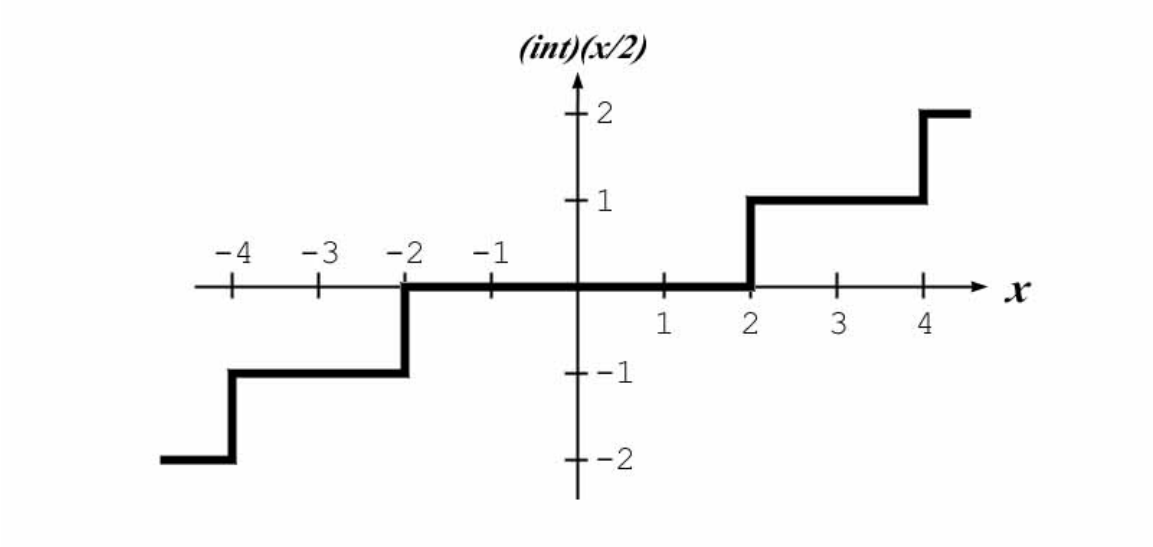
This code segment converts a profile into a sector set in the horizontal direction.

The first statement is simply a truncating integer divide of the left boundary of the profile.

The right boundary is a rounded-up integer divide. In the real domain, I think it's easiest to do this with a truncating divide and a conditional increment of the result, though alternatives may well exist.

The width of the sector set is simply the right boundary minus the left boundary.

2.19 Integer Divide truncates towards Zero



The preceding formula will work perfectly satisfactorily for the horizontal direction, but the vertical direction has components that are both positive and negative. Its range is -45° to +45°. This slide is a graph of an integer truncation of the result of a floating-point division. Both simple integer division and integer truncations of the result of a floating-point division truncate towards zero. If we were to use the simple formula we would wind up with a double-width segment on the horizontal plane.

2.20 Calculating the Vertical Sector Range

```
sectorSet.bottom = (int)((profile.bottom + 45°) / SECTOR_HEIGHT);
top = (int)((profile.top + 45°) / SECTOR_HEIGHT);
if (top * SECTOR_HEIGHT != profile.top) top ++ ;
sectorSet.height = top - sectorSet.bottom;
```

The problem is easily overcome by adding a constant 45° offset so that the calculations are always done in the positive domain.

The offset also ensures that the sector numbers are always zero or positive.

2.21 Expanding a Sector Set

```
for (i = 0; i < sectorSet.width; i++) {
    x = (sectorSet.left + i) % NO_OF_HORIZONTAL_SECTORS;
    for (j = 0; j < sectorSet.height; j++) {
        y = sectorSet.bottom + j;
        // Insert friend into bucket identified by
        // (x,y) or search bucket (x,y) for possible
        // trajectory intersections.
    }
}
```

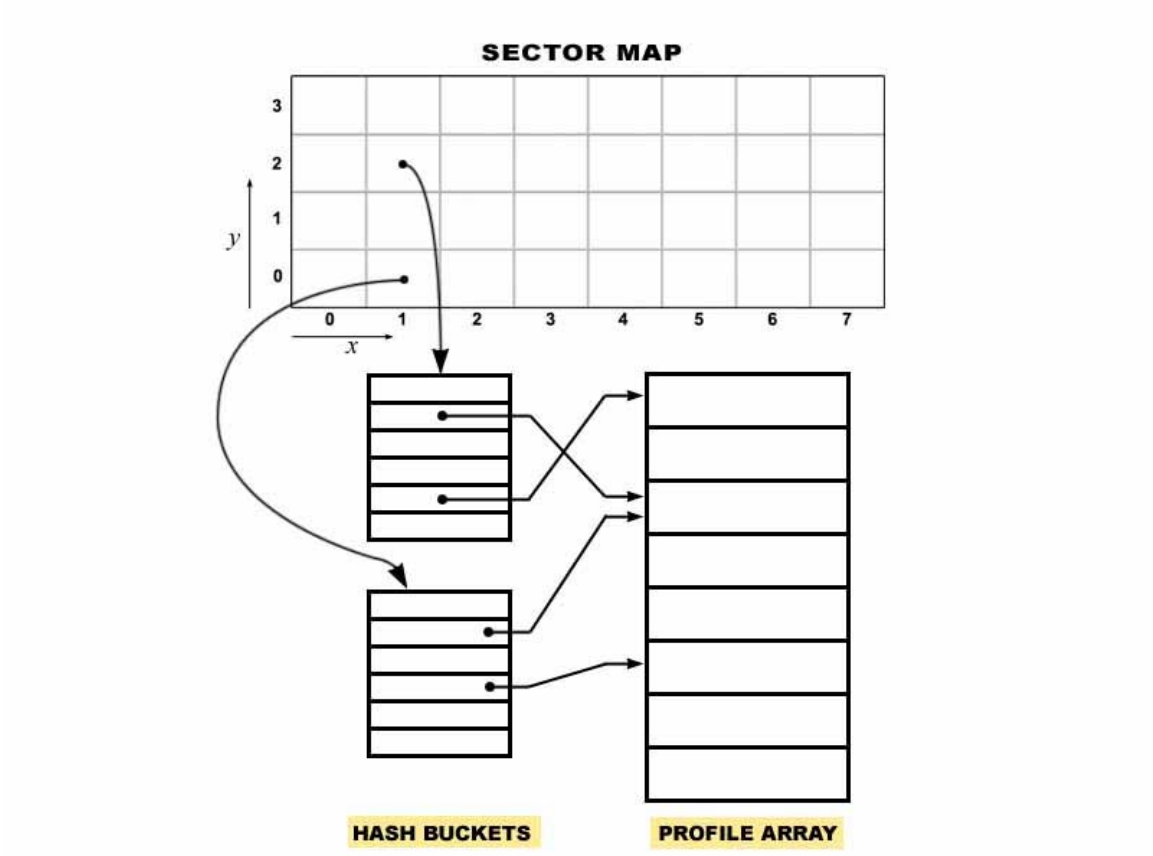
Now let's look at sector set expansion.

I used a slightly more complicated algorithm in the example solution, but this program works just as well and is easier to understand.

The index 'i' simply counts through the columns in the sector set. 'x', the horizontal sector number, is the left boundary plus 'i' modulo the number of horizontal sectors. The modulo operation deals with the wrap-around effect we observed earlier.

Counting through the vertical sectors is much the same, but there is no need to deal with a wrap-around, though it is harmless to do so.

2.22 Designing the Bucket Data Structure



Having dealt with the design of profile and sector set data structures and figured out an algorithm for the expansion of sector sets, we are now in a position to complete the high-level data structure design.

This diagram is simply a schematic of the earlier one.

It is clear from the diagram that we need to create three data structures.

First, there is a sector map. The purpose of the sector map is to translate the x-y co-ordinates that come out of the sector set expansion algorithm into a reference to a hash bucket.

Next, there are the hash buckets themselves.

Last there is profile array that contains the profiles of the friends.

To implement the sector map using anything other than a two-dimensional array is certifiable insanity.

There are some choices for the hash buckets themselves. The two that make any kind of sense are a fixed-length array with a member count and a linked list. Linked lists would be by far the superior choice because fixed length arrays would waste a lot of memory.

We know from the question paper that we are required to load 100,000 friends, so a simple 100,000-element array is the obvious choice for profile array.

From here it's simply a matter of plugging the data structures and algorithms I've mentioned into the pseudo-code provided in the question paper and Bingo, the problem is solved.

2.23 Results of a Trial Run

Number of sectors		Search time per trajectory on a 500MHz Pentium (microseconds)
Horizontal	Vertical	
1	1	18,000
90	22	25
180	45	10
360	90	6
720	180	5

Now let's take a look at the performance of the algorithm. One sector in either direction is equivalent to the original sequential search. As a reference point it takes 18,000us on our 500MHz Pentium. It would take much longer on a battery-powered microprocessor inside the butt of a rifle.

In comparison a grid of 720 by 180 sectors reduces the search time to 5us per trajectory, an improvement of a factor of over three and a half thousand.

2.24 Summary

- Use diagrams to represent spatial problems;
- Use a stylised entity relationship diagram to visualise a data structure;
- Use data flow diagrams to identify processes and data movements;
- Engage brain before plugging in packages;
- Tabulate design alternatives to choose the best;
- Choose the data structure that will be easiest to use;
- If you can't conceptualise a logical function, consider it's complement;
- Use a diagram to visualise intersection in a cyclic domain;
- Integer truncation rounds towards zero;
- Use simple data structures – arrays and linked lists.

3 25 Factorial

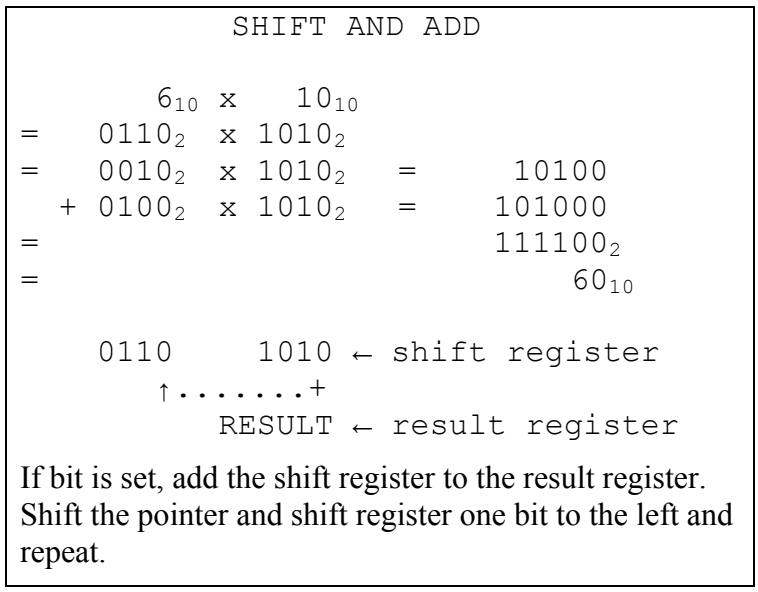
3.1 The Question

Write a program to calculate and display the exact decimal value of 25-factorial (i.e. 25 * 24 * 23 * ... * 3 * 2) without the aid of any language or operating system supplied extended-precision arithmetic functions.

Let's consider the 25-Factorial problem.

The 25 Factorial problem is to calculate value of 25 x 24 x 23 all the way down to 2. The problem is that 25 factorial is approximately 2^{84} or 10^{26} and the system we're working on doesn't support extended precision arithmetic.

3.2 Shift and Add



The basic algorithm for multiplying on a computer is known as the 'shift and add' algorithm. This slide shows basically how it works.

To multiply 6 by 10 in binary, we first of all split six into two and four. The result is then two by ten plus four by ten. Breaking one of the operands down into a sequence of powers of two greatly simplifies the multiplication operation because multiplying by a power of two is a simple shift. For example four times 'n' is 'n' shifted left two places.

In practice, a multiplication operation is done with a pointer and a shift register. The pointer points to a bit in the first operand, and the second operand is placed in a shift register. If the bit over the pointer is set, the contents of the shift register are added to the result. The pointer and shift register are then shifted one bit to the left and the process is repeated.

3.3 The Complication

```
x = 1;
for (i = 2; i <= 25; i++) x *= i;

do {
    push (x % 10);
    x /= 10;
} while (x != 0);
while (pop (&c)) putchar(c + '0');
```

We could simply use the shift and add algorithm to progressively calculate the required product. The registers could be implemented as arrays and the shift and add operations could be programmed to operate on the arrays, but there's a complication. We're required to display the result in decimal.

This slide shows what we would need to do if the computer had sufficient precision to store 25-factorial. First, we would need to calculate the value of 25-factorial in binary. But then, to convert the binary number into a decimal string we would need to progressively divide the bindary number by 10 and take the remainder. To convert a binary number into a decimal string we need divide and remainder operations as well.

Binary division can be done with a shift and subtract algorithm that is more-or-less the reverse of the shift and add algorithm. But now we have to write two extended-precision functions, one to multiply the factors and another to display the result. That's starting to look like a lot of hard work. Maybe there's an easier way.

3.4 Decimal Register

Decimal Register			
0...9	0...9	0...9	0...9

With a decimal register each cell contains a value between zero and nine.

Converting the contents of a decimal register into a decimal string is easy. You simply skip the leading zeros and then display the string of digits in the remaining cells.

But how difficult is it to multiply decimal registers?

3.5 Multiplying by a Digit

```
    1281  x    7
=      1  x    7  =       7
  +   8   x    7  =      56
  +  2    x    7  =     14
  + 1     x    7  =     7
=                     8967

carry = 0;
r[3]=1, r[2]=2, r[1]=8, r[0]=1;
for (i = 0; i < 4; i++) {
    x = r[i] * 7 + carry;
    r[i] = x % 10;
    carry = x / 10;
}
```

We can multiply by a digit-sized factor using an algorithm closely related to the shift and add algorithm. We can break the large number into digits and powers of ten. Multiply each digit-sized component of the large number by the digit-sized factor and then add the shifted products together to get the result.

This program fragment solves the problem.

3.6 Multiplying by a Two-Digit Number

```
bigNumber * (d[1],d[0]) =      bigNumber * d[0]
                         + bigNumber * d[1] * 10
```

We could multiply our big number by a two-digit register by multiplying the big number by the least significant digit and then adding the big number multiplied by the most significant digit shifted one decimal place to the left. But this is complicated in comparison to the easy answer.

3.7 The Simple Answer

```
char   r[PRECISION];
short  i, j, x, carry;

memset (r, 0, PRECISION);
r[0] = 1;
for (i = 2; i <= 25; i++) {
    carry = 0;
    for (j = 0; j < PRECISION; j++) {
        x = r[j] * i + carry;
        r[j] = x % 10;
        carry = x / 10;
    }
}
```

This is more-or-less the answer that we published on the Internet.

If we examine the original algorithm for multiplying by a digit, you will see that we can multiply by a number significantly greater than a digit without disturbing the integrity of the algorithm. Certainly a two-digit number represents no problem.

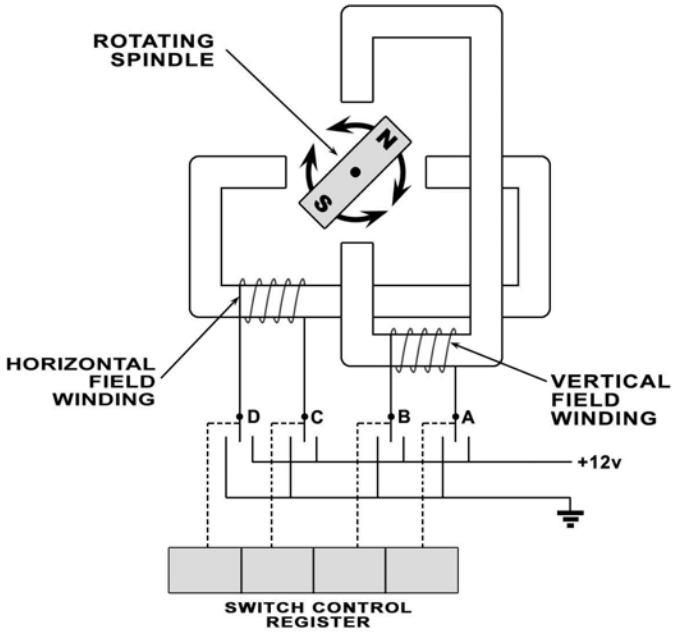
A mere 23 source instructions, far less than a page, could have earned the 100 points. The question was a give-away.

3.8 Summary

- Theory of shift and add;
- Store data in the form that is easiest to use;
- Decimal string multiplication;
- Progressive simplification.

4 WINDSCREEN WIPERS

4.1 The Stepping Motor



The Windscreen Wiper problem involved driving a windscreen wiper blade with a stepping motor.

Conceptually, the stepping motor consisted of a permanent magnet mounted on a spindle with two field windings that could be energised in sequence to drive the spindle in one direction or the other.

When switch A is switched to +12V and switch B is switched to ground (i.e. 0V), the conventional current flows from the common of switch A to the common of switch B. This causes the core of the A-B winding to be magnetised so that its top is a north pole and its bottom is a south pole. This, in turn causes the spindle to rotate clockwise until the north pole of the spindle is at the bottom. By operating switches A to D in the correct order, the spindle can be made to rotate in either direction.

4.2 Step Table

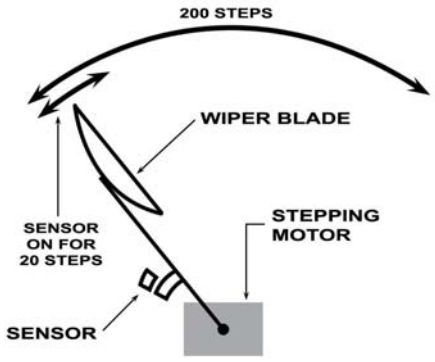
For clockwise rotation	**SCR VALUE**				**For anti-clockwise rotation**
	D	**C**	**B**	**A**	
↓	0	0	0	1	↑
	0	1	0	0	
	0	0	1	0	
	1	0	0	0	

The question paper included this table of the SCR values for rotating the spindle in either direction.

To rotate the spindle clockwise, the values in the first row should be loaded into the SCR, and then the second row, and then the third and fourth rows and then the first row again.

To rotate the spindle anticlockwise, simply reverse the order of the rows.

4.3 The Required Motion



The wiper blade was required to move backwards and forwards in an arc that represented 200 quarter-turn steps of the stepping motor.

A sensor was provided to determine the initial location of the wiper blade. The blade had to stop and reverse 20 steps to the left of the point where the sensor switched on and 180 steps to the right of the point where the sensor switched off.

4.4 Acceleration and Deceleration

Inter-Step Time (milliseconds)															
42	17	13	11	10	9	8	8	7	7	6	6	6	6	6	5

As an additional complication, the wiper blades were required to decelerate before reversing and then gradually accelerate up to the full speed of 200 steps per second. The acceleration and deceleration were required to prevent overloading the blade mechanism and avoid skipped steps.

4.5 Operating System Interface

```
class WiperControl {
     //...
     public void setSCR (int scrVal);
     public boolean getSensorState();
     //...
}
interface WiperInterface {
     void initialise (WiperControl control);
     void startWiping ();
     void stopWiping ();
     void tick ();
}
```

The operating system interface consisted of a `WiperControl` class that represented the operating system and a `WiperInterface` interface that the control program had to implement.

The control program, which the contestants had to write, could set the state of the SCR by calling `setSCR` in the `WiperControl` class.

The control program could test the state of the sensor by calling `getSensorState`.

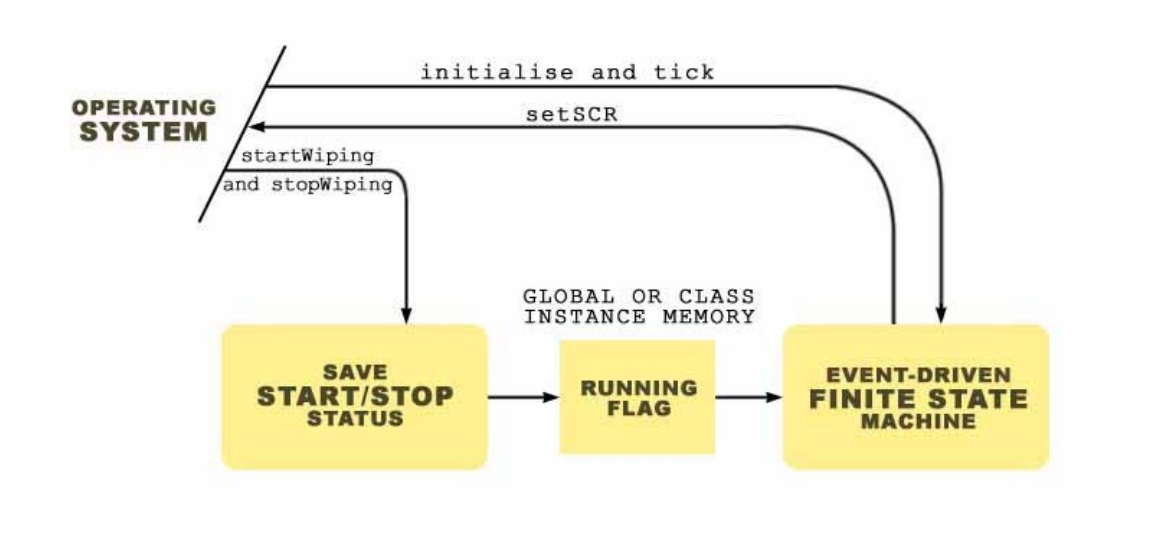
The operating system would call the `initialise` method when power was applied to the wiper system. The `initialise` method had to sample the state of the sensor and, if necessary, return the wiper blades to the left most starting position.

The operating system would call the `startWiping` method to start the wiper blades moving back and forth, and the `stopWiping` method to stop the wiper blades moving back and forth. When stopped, the wiper blades had to move back to the left most starting position before stopping. The `startWiping` and `stopWiping` methods could be called at any time. If, for example, `stopWiping` was called, but before the wiper blades got to their left most starting position, `startWiping` was called, the wiper blades had to continue wiping as if `stopWiping` had never been called in the first place.

The operating system would call the `tick` method once every millisecond to provide a time base for the control program.

The control program was not allowed to hog control in any of the methods. If it did so, control would not be transferred to any of the other methods and the system would hang.

4.6 Synchronising the Start/Stop Instructions



The first problem is to devise a means to convey the information passed via the asynchronous `startWiping` and `stopWiping` calls to the finite state machine that will control the stepping motor.

The easy way to do this is to create a running flag in global or class instance memory. `startWiping` can set the running flag and `stopWiping` can reset it. The event-driven finite state machine can sample the state of the running flag each time the wiper blade reaches the left-most position. It can then start a new cycle or stop, depending on the state of the flag.

4.7 Top-to-Bottom Pseudo-Code

```
-- UNINITIATED state
SCR = 0
phase = 0
if sensor is off:
     SCR = STEP_TABLE[ phase ];
     stepsToEnd = 20
     while stepsToEnd != 0
          wait for 42 ticks -- RESETTING state
          if sensor is off
               stepsToEnd = 20
          else
               stepsToEnd --
          phase = (phase + 4 - 1) % 4
          SCR = STEP_TABLE[ phase ]
     end while
end if
enter normal operating mode
```

Now to the design of the finite state machine.

Virtually all problems of this type are best tackled by pseudo-coding the top-to-bottom sequence that has to be executed. This excerpt is adapted from the pseudo-code in the introductory comments in the published solution to the Windscreen Wiper problem. It describes the start-up sequence for returning the windscreen wipers to their initial location.

Once you've devised the top-to-bottom pseudo-code, an almost mechanical process can be used to translate it into a finite state machine.

To convert the top-to-bottom pseudo code into a finite state machine, allocate one state for the uninitiated machine, and then one more for each 'wait' operation in the pseudo code.

4.8 State Transition Table

State	Event	Processing

UNINITIATED	`initiate`	SCR = 0
		phase = 0
		if sensor is off:
		  SCR = STEP_TABLE[phase];
		  stepsToEnd = 20
		  waitRemaining = 42
		  state = RESETTING
		else
		  enter normal operating mode
		end if.

RESETTING	`tick`	decrement waitRemaining
		if waitRemaining == 0
		  if sensor is off
		  stepsToEnd = 20
		  else
		  decrement stepsToEnd
		  phase = (phase + 4 – 1) % 4
		  SCR = STEP_TABLE[phase]
		  if stepsToEnd != 0
		  waitRemaining = 42;
		  else
		  enter normal operating mode
		  end if
		end if.

The start-up pseudo-code can now be translated into a state transition table.

A similar process can be used to create the procedures for the normal operating mode. First pseudo-code what the program has to do and then translate the pseudo code into a state transition table.

Although state transition diagrams are widely taught in universities, in practice, state transition tables are much more useful. The problem with state transition diagrams is that they run out of steam very quickly. If you have a machine with 10 or 20 states and roughly the same number of stimuli, there will be so many crossing lines on your diagram that it will be illegible. Moreover there is nowhere to describe the executable procedure in a state transition diagram. All up, the table representation is much more useful.

4.9 Re-synchronising the Theoretical Wiper Position

The question paper required the program to:

- decelerate the blades so that they stop **180 steps after the sensor switches off**;
- reverse the direction of the motor, accelerate the blades, **adjust the theoretical location of the blades** when the sensor switches on...

In effect, the control program had to re-synchronise the theoretical wiper position each time the wiper blades passed the sensor threshold.

Moreover, because no attempt was made during initialisation to confirm the initial starting position if the sensor was on, the initial starting position could be anywhere to the left of the sensor. Merely running 200 steps and reversing the direction of the motor might result in the wiper blade overrunning the right-most position, running off the windscreen and scratching the bonnet of the car.

We were also required to adjust the theoretical location of the blades while they were travelling to the left. This was required because there might have been some obstruction that stopped the windscreen wiper halfway through its travel. If this were to occur, and the wiper were programmed to run 200 steps to the left, the wiper blades would run off the windscreen and scratch the bonnet of the car on the other side. Yet another irritating warranty claim.

What was needed was a no-man's land in the middle of the arc that could be adjusted to compensate for the variable arc length. In the example solution, this was accomplished with two displacement variables: `stepsFromStart` and `stepsToEnd`. `stepsFromStart` was used to manage the acceleration. `stepsToEnd` was set to 180 or 20 at the sensor threshold to adjust the total arc length.

Designs that used only one position variable tended to have problems if an obstruction restricted the arc length to less than twice the number of steps in the acceleration table.

4.10 Making Use of Tables

```
static final int    STEP_TABLE[] = {1, 4, 2, 8};
                                   // Step sequence table
static final int    NEXT_PHASE[][] =
     {{1, 2, 3, 0}, {3, 0, 1, 2}};
                                   // Next phase map

// To move one step to the right:
phase = NEXT_PHASE[0][phase];
wiperControl.setSCR(STEP_TABLE[phase]);

// To move one step to the left:
phase = NEXT_PHASE[1][phase];
wiperControl.setSCR(STEP_TABLE[phase]);
```

The Windscreen Wiper problem is made much easier by the use of tables. For example, consider this excerpt from the example solution. The SCR values from the question paper can be loaded into a step table that maps an index in the range 0 to 3 to the value that needs to be loaded into the switch control register.

In the worked example, the function for incrementing and decrementing the phase counter was also implemented with a table, though there are certainly other acceptable ways of doing it.

Table driven designs typically execute quickly and are easy to program. It's a good idea to make full use of the technique.

4.11 Summary

- Transfer asynchronously changing data from one process to another in global or class instance memory;
- Convert the top-to-bottom sequence into an event-driven finite state machine;
- Correctly program the variable arc length.
- Make use of tables.