ISSUED ON
12 MARCH 2014

Engineering Tripos Part IA FIRST YEAR

Paper 4: Mathematical Methods

COMPUTING

Examples Paper 2

Straightforward questions are marked †.
Tripos standard (but not necessarily Tripos length) questions are marked ∗.
Hints and answers can be found at the back of the paper.

Questions 1 to 7 are about numerical programming. The Digital Circuits and Information Processing course (Paper 3) also includes material on binary representation, which you might find useful when attempting these questions. You are encouraged to search on-line for the IEEE standard relating to floating point representation (IEEE-754).

1. † (a) Convert the following unsigned binary numbers to decimal.
   (i) 1 (ii) 11111111 (iii) 11011101
   (iv) 1.1 (v) 0.01 (vi) 10.00011

   (b) Convert the following decimal numbers to a standard unsigned binary representation (not an exponential representation).
   (i) 4 (ii) 128 (iii) 93
   (iv) 2.5 (v) 1.75 (vi) 8.2

2. (a) Convert the following IEEE standard single precision floating point numbers into decimal.

(i)	0	10000001	00000000000000000000000
(ii)	1	01111111	10000000000000000000000
(iii)	0	11000000	01100000000000000000001
(iv)	0	00000000	00000000000000000000000

   (b) Convert the following decimal numbers into IEEE standard single precision floating point format.
   (i) 1.0 (ii) -6.0 (iii) 0.21875 (iv) 8×10^6

   (c) *Optional for students with considerable programming experience and access to a computer.* Write a program which directly accesses the computer's memory to display the stored bit patterns for arbitrary `float` variables. Use your program to check your answers to parts (a) and (b).

3. Work out the largest and smallest numbers that can be stored in the IEEE standard single precision floating point format. Assume that all mantissas are normalised, that the exponent 11111111 is reserved for the special number ∞ and that the exponent 00000000 is reserved for the special number 0.

4. Consider the following segment of C++ code (note that the numbers in the variable declarations also appear in Question 2).

```
float f1 = 1.0;        double d1 = 1.0;
float f2 = -6.0;       double d2 = -6.0;
float f3 = 0.21875;    double d3 = 0.21875;
float f4 = 8.0E+6;     double d4 = 8.0E+6;
float f5, f6;          double d5, d6;

f5 = f1 + f3;          d5 = d1 + d3;
f6 = f3 + f4;          d6 = d3 + d4;
```

   What would you expect to be the final contents of the variables `f5`, `f6`, `d5` and `d6`? Justify your answers.

5. The following C++ code segment is compiled and run on a standard PC.

```
const float f1 = 0.9;
const float f2 = 0.5;
const float f3 = 0.9 - 1E-7;
cout.precision(20); // Make sure cout displays lots of decimal places
cout << "f1 is " << f1 << endl;
cout << "f2 is " << f2 << endl;
cout << "f3 is " << f3 << endl;
cout << "f1 - f3 is " << f1 - f3 << endl;
```

   To the user's surprise, the console displays:

```
f1 is 0.89999997615814208984
f2 is 0.5
f3 is 0.89999991655349731445
f1 - f3 is 5.9604644775390625e-08
```

   Explain why this isn't really so surprising.

6. Consider the following C++ code segment.

```
float x = 1.5707963;
double y = 1.5707963;
cout << "tan(x) is " << tan(x) << endl;
cout << "tan(y) is " << tan(y) << endl;
```

When the code is executed, the console displays:

```
tan(x) is 1.32454e+07
tan(y) is 3.73205e+07
```

Explain why there is such a huge difference between the two results.

7. * The following C++ code prints out the sequence of numbers generated by the difference equation $x_n = 3x_{n-1} - 2x_{n-2}$ with initial values x_1 and x_2.

```
float x1, x2, xn, xnm1, xnm2;
x1 = 1.0;
x2 = 1.00000005;
xnm2 = x1; xnm1 = x2;
for (n = 3; n <= limit; n++) {
   xn = 3.0 * xnm1 - 2.0 * xnm2;
   cout << n << ' ' << xn << endl;
   xnm2 = xnm1; xnm1 = xn;
}
```

When executed, the code prints a sequence with $x_n = 1.0$ for all n. When line 3 of the code is changed to `x2 = 1.00000006`, the sequence shows x_n growing exponentially, becoming so large that it overflows the allowable range for a `float` by the time $n = 153$. Explain this behaviour.

Questions 8 to 13 are about searching, sorting and algorithmic complexity.

8. † What is meant by *algorithmic complexity*? Write down the complexity of:

   (a) Printing the first number in a list of n numbers.

   (b) Finding the mean of a list of n numbers.

   (c) Adding two $n \times n$ matrices.

   (d) Multiplying two $n \times n$ matrices.

   (e) Listing all the positive n-digit decimal integers.

9. The following program implements a variation of *Eratosthenes's sieve* for finding prime numbers.

```
int main() {
  // Variation of Sieve of Eratosthenes for finding prime numbers

  int i, j;
```

```
  const int n = 100000;
  bool prime[n];

  // Initialise array
  for (i = 2; i < n; i++) prime[i] = true;

  for (i = 2; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (j % i == 0) prime[j] = false;

  // Display results
  for (i = 2; i < n; i++) if (prime[i]) cout << i << endl;

  return 0;
}
```

(a) Satisfy yourself that the program does indeed find and print prime numbers.

(b) What is the largest number checked to see if it is prime?

(c) What is the complexity of this implementation of Eratosthenes's sieve? Show how to modify one line of the program to improve the complexity to $\mathcal{O}(n^{3/2})$. Estimate the speedup for the case $n = 100000$.

10. The 11-digit mobile telephone numbers of 100 Cambridge students are to be stored in a hash table of length 1000. Two hash functions are proposed:

    (a) `i = x % 1000;`

    (b) `i = x / 100000000;`

    where `int x` is the phone number and `i` is the index. Explain why one of these will almost certainly perform much better than the other.

11. † Which of exchange sort and QuickSort would you expect to be faster for sorting a list of (a) 10 and (b) 1000000 items? Justify your answers.

12. * The following function searches for the value of `x` for which the function `f(x)` is zero. It requires an initial range `lo` to `hi` within which the root is thought to lie.

```
float f(float x); // arbitrary function of x

float find_root(float lo, float hi)
{
  const float limit = 0.0001; // required precision
  float mid, f_mid, f_lo, f_hi;
```

```
  mid = 0.5 * (hi + lo);
  if (hi - lo < limit) return mid;

  f_mid = f(mid); f_lo = f(lo); f_hi = f(hi);
  if (f_lo * f_mid < 0.0) return find_root(lo, mid);
  else return find_root(mid, hi);
}
```

(a) State which search principle is being used. How sensitive is the execution time to the required precision `limit`?

(b) The function, while containing no obvious bugs, is not very carefully written. Identify the ways in which it can fail and suggest some possible improvements.

13. * Consider the following C++ implementation of the *bucket sort* algorithm.

```
void BucketSort(int list[], int n)
{
  const int max_digits = 10; // maximum number of digits in an int
  const int max_size = 10000; // maximum number of elements in list
  int i, j, k, l, digit, counts[10], buckets[10][max_size];
  long long int a = 10; // can store a bigger number than a normal int

  for (i = 1; i <= max_digits; i++, a = a*10) { // loop over each digit

    for (k = 0; k < 10; k++) counts[k] = 0; // reset bucket counts

    for (j = 0; j < n; j++) { // distribution pass
      digit = (list[j] % a) / (a/10);
      buckets[digit][counts[digit]] = list[j];
      counts[digit]++;
    }

    j = 0; // gathering pass
    for (k = 0; k < 10; k++)
      for (l = 0; l < counts[k]; l++, j++) list[j] = buckets[k][l];
  }
}
```

The sorting algorithms presented in the lecture course all work *in situ*: they require very little memory on top of the list's own storage requirements. In contrast, bucket sort trades memory for speed. The algorithm sorts lists of positive integers. It loops over each of the integers' digits, starting with the least significant and working

towards the most significant. For each digit, there is a *distribution pass* followed by a *gathering pass.* In the distribution pass, the numbers are put into buckets depending on the value of the digit under consideration. In the gathering pass, the numbers are copied back, one bucket after another, into the original list.

(a) Satisfy yourself that bucket sort works. To do this, work through the code by hand to sort the list $\{9, 692, 45, 142\}$. Keep track of the contents of `list`, `buckets` and `counts`.

(b) What is the algorithmic complexity of bucket sort? What are its memory requirements as a function of `n`?

Suitable past Tripos questions: 2002 Qus. 10b, 10c & 11, 2003 Qu. 10, 2004 Qus. 10 & 11, 2005 Qu. 11, 2006 Qu. 11, 2007 Qus. 5c & 12, 2008 Qus. 11 & 12

Hints

1. (a) The binary digits to the left of the decimal point represent $2^0, 2^1 \ldots 2^n$, while the digits to the right of the decimal point represent $2^{-1}, 2^{-2} \ldots 2^{-m}$. (b) First write the decimal number as a sum of powers of two. For a non-integer, you'll need negative powers of two as well as positive powers.

2. (a) The single precision floating point format has a sign bit s, followed by an 8-bit exponent e and a 23-bit mantissa m. The exponential base is 2, there is a bias of -127 in the exponent and, since the numbers are for the most part normalised, an implicit leading '1.' in the mantissa. The decimal equivalent of the stored number is therefore

$$(-1)^s \times 1.m \times 2^{e-127}$$

where the mantissa $1.m$ and the exponent e are in binary, and therefore need converting into decimal first. Let's do (i) as an example. Exponent $e = 10000001_2 = 129_{10}$. Mantissa $1.m = 1.0_2 = 1.0_{10}$. Number $= (-1)^0 \times 1.0 \times 2^{129-127} = 1.0 \times 2^2 = 4$.
(b) To convert from decimal to floating point, we divide or multiply the number by two until we arrive at a number which is 1 point something: this is the mantissa, while the exponent is deduced from the number of times we had to divide or multiply by two. Let's do (ii) as an example. $-6.0 = -1.5 \times 2^2$. Mantissa is 1.5. Exponent is 129 (2 after bias). Mantissa in binary is 1.1. Exponent in binary is 10000001.

0	10000001	10000000000000000000000

4. First of all, note that `8.0E+6` is C++ jargon for 8.0×10^6. Before numbers are added, their exponents must first be matched by shifting the mantissa of the smaller number to the right. A double precision floating point number (ie. a `double`) has a 52 bit mantissa, compared with 23 bits for a single precision `float`.

5. Note that 0.5 can be represented precisely in floating point form, since it is a simple power of 2 ($0.5 \equiv 2^{-1}$). But how large a mantissa do we need to represent 0.9 precisely in floating point form? Perhaps more than 23 bits? The difference between `f1` and `f3` should be 10^{-7}, but the computer's answer is very different to this. However, given the precision with which the computer can represent `f1` and `f3` in the first place, could we really expect a more accurate result? This is *subtractive cancellation.*

6. Because of the floating point precision, `y` will be closer to 1.5707963 than `x`. Remember that `tan` expects its argument in radians, and also note that 1.5707963 is very close to $\pi/2$. Sketch $\tan\theta$ around $\theta = \pi/2$. You should be able to see what's going on now.

7. First solve the difference equation by hand for the initial conditions $x_1 = x_2 = 1.0$. What would happen if x_2 was just a tiny bit bigger than 1.0? Considering the precision of the floating point representation, what might `x2` be after initialisation with the statement `x2 = 1.00000005`? Might `x2 = 1.00000006` lead to a different outcome?

9. (a) Remember that if `j % i == 0`, then `j` is wholly divisible by `i`. (c) Work through the program by hand for the case `n = 10`. You should find that you don't eliminate any further prime numbers beyond `i = 3`. This is no accident, and indicates how to change the limit on the `i` loop to improve the complexity of the algorithm.

10. A good hash function should distribute the telephone numbers evenly through the hash table. This way, there won't be too many collisions causing numbers to be stored out of position. Collisions are a bad thing, since they slow down both storage and retrieval. Bear in mind that all mobile telephone numbers from a particular service provider start with the same few digits.

11. The execution time of an $\mathcal{O}(n^i)$ algorithm is kn^i. For small n, the constant of proportionality k may be more important than the power i.

12. (a) Ask yourself how many extra iterations would be required if `limit` were halved. (b) The function can fail in four distinct ways. Here are typical ways to trigger each of the four failure modes: (i) `lo > hi`; (ii) `lo = 0.0`, `hi = 2.0` and `f(x) = 1.0 - x`; (iii) `f_lo` and `f_hi` have the same sign; and (iv) `limit` is smaller than the floating point machine accuracy. Think of ways to fix (i), (ii) and (iv). For case (iii), the best we can do is to return an error code.

13. The trickiest line in the program is the one that isolates successive digits of a number:
```
digit = (list[j] % a) / (a/10);
```
We start with `a = 10` and then multiply `a` by 10 each time round the outer `i` loop. Let's illustrate the operation of this line with the number 45. Remember that this is integer arithmetic, so after the division by `a/10` any fractional part is discarded.

i	a	operation
1	10	`45%10 = 5; 5 / (10/10) = 5; digit = 5;`
2	100	`45%100 = 45; 45 / (100/10) = 4; digit = 4;`
3	1000	`45%1000 = 45; 45 / (1000/10) = 0; digit = 0;`

Answers

1. (a) (i) 1 (ii) 255 (iii) 221 (iv) 1.5 (v) 0.25 (vi) 2.09375 (b) (i) 100
   (ii) 10000000 (iii) 1011101 (iv) 10.1 (v) 1.11 (vi) 1000.00110011 (approximately)

2. (a) (i) 4 (ii) -1.5 (iii) 5.07×10^{19} (iv) 0

   (b) (i)

0	01111111	00000000000000000000000

   (ii)

1	10000001	00000000000000000000000

   (iii)

0	01111100	11000000000000000000000

   (iv)

0	10010101	11101000010010000000000

3. Largest is $2^{128} = 3.40 \times 10^{38}$. Smallest is $2^{-126} = 1.18 \times 10^{-38}$.

4. `f5 = 1.21875; f6 = 8000000; d5 = 1.21875; d6 = 8000000.21875`

8. (a) $\mathcal{O}(1)$ (b) $\mathcal{O}(n)$ (c) $\mathcal{O}(n^2)$ (d) $\mathcal{O}(n^3)$ (e) $\mathcal{O}(10^n)$

9. (b) 99999 (c) Change `i` loop to: `for (i = 2; i < sqrt(n); i++)`. Speedup is 316.

11. (a) Exchange sort. (b) QuickSort.

12. (a) Binary search, execution time depends only logarithmically on the precision.

13. (b) Complexity $\mathcal{O}(n)$, memory $\approx 10n$ extra `int`'s.

Gábor Csányi
March 2013

Lent 2014