Whiteboard

目录

1. Introduction to the course and to the C Language
    1. Software engineering and programming
    2. What is Computer Science?
        1. Euclid and Algorithm Speed
            1. Comparing Euclid and exhaustive search
        2. Recursive vs Iterative Euclid
    3. Timing programs in UNIX

Introduction to the course and to the C Language

Aim of the course is to:

- teach important, fundamental algorithms
- have you understand their properties (speed, size and use)
- have you think like a **computer scientist** and act like a **software engineer**
    - design systems by selecting the right algorithm and data structures for a particular job
    - test and debug systematically

The course was called *Algorithms and Data Structures*

- Algorithms
    - instructions to solve a problem, or how to do something
    - must be clear/unambigious
    - example: bake a cake, go to uni, build a house, find the greatest common divisor ...
- Data structures
    - specifies how data is to be stored
    - inseparable from algorithms
    - most of the exciting modern developments in programming have been in data structures more than algorithms

DS&As are implemented in the C programming language.

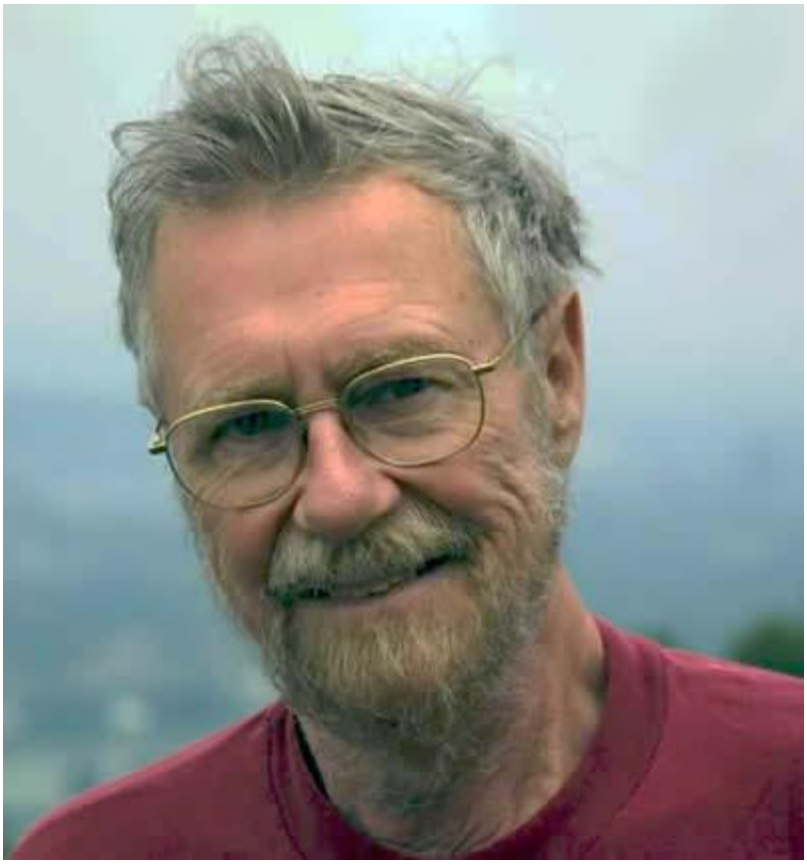
Here's a quote:

> A programming language is a tool that has a profound influence on our **thinking habits**

... from Edsgar Dijkstra

- it's hard to separate the language from the way we **think** about DS&A ...
    - and obviously the choice of language impacts the performance, maintainability, portability and extensibility
- C/C++ require multi-layered **thinking**
    - high-level: you must structure (layer) algorithms and data to manage complexity
    - low-level: you can address memory directly

E. W. Dijkstra [1930-2002]



- in the 1950s and 1960s, 'programming' was considered an art (like solving a puzzle)
  - but as complexity of algorithms increased
  - and demands for quality (reliability) increased (e.g. NASA)
  - 'programming' was in crisis
- Dijkstra (studied as a theoretical physicist)
  - developed a new programming style based on 'science'
  - devised a way of programming that made programs provably correct
  - he coined the phrase *structured programming* (received the Turing award in 1972 for this)
    - this led to the birth of the discipline *software engineering*

The great mathematician Donald Knuth said in 1974:

> A revolution is taking place in the way we write programs and teach programming, because we are beginning to understand the associated mental processes more deeply. It is impossible to read the recent book *Structured Programming* [by Dijkstra, Dahl and Hoare], without having it change your life.

Dijkstra (controversial) quotes:

> Computer science is no more about computers than astronomy is about telescopes

> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense

> Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better

> The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer

Another Dijkstra quote that every academic touts:

> Program testing can be used to show the presence of bugs, but never to show their absence (1972)

Yet another quote:

> Object-oriented programming is an exceptionally bad idea which could only have originated in California

- *Why did he say this?*

- Dijkstra's research: proving formal correctness of programs
  - uses a calculus developed by Hoare, design/write a program that provably satisfies a formal specification
  - certain applications such as financial transactions, health care, (space) travel, hardware design...
    - critical: 100% correct, 100% of the time
    - verification techniques must be used, including formal correctness proofs
- you cannot apply Hoare's calculus to an OO program/design in general
  - mathematical models of OO programs don't exist in general
  - e.g. problems: inheritance and types/sub-types, polymorphism (type determined at runtime)
    - **Is this a problem for OO?**
      - OO is suited to business-like applications where...
        - fast development is important
        - demonstrated correctness in common use-cases is enough
          - testing is adequate (and easier)
    - answer is hence **no** (I think)

Another view of imperative vs OO, and verifiability:

- the OO paradigm is a layer on top of an imperative program
  - verification techniques can prove the imperative program layer is correct, to a point
- the OO layer enables complex, dynamic designs that (only) humans can understand
  - this layer is impossible to mathematically model

And another, more cynical, view

- structured programs can be mathematically modelled and verified
  - the structure restricts what the program may do, but that makes them analysable
- take it up 'up one level', unstructured programs (i.e. use **goto** statements) can be complex
  - 'goto' statements introduce static complexity that is impossible to analyse
  - **spaghetti code**
  - another Dijkstra quote, taken from an article called *Go To Statement Considered Harmful,* published in 1968

> The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program

- OO programs take this to another level again
  - the added complexity is dynamic (a static analysis is meaningless)
  - **spaghetti behaviour**

- see http://crypto.stanford.edu/~blynn/c/object.html for a devastating commentary of OO

Software engineering and programming

A programmer:

- writes a program
- is primarily a personal activity

A software developer:

- writes a software component
- makes a deliberate choice of data structure and algorithm
- understands the complexity
- is part of a team in practice

classical ENGINEERING

- products are hard, tangible, immutable (e.g. a TV, a bridge, ...)
- requirements are 'fixed' initially and do not change
- their design must allow for (mass) manufacturing/production
  - hence 2 phases: 'design' and 'manufacture'
- products usually suffer fatigue, i.e. wear out
- MTBF: *Mean Time Between Failures* relative to lifetime is long

software ENGINEERING

- s/w is 'soft', i.e. intangible
- requirements changed continually, even during development, and even after it is finished
- effort is in making the one and only copy
- the 2nd phase in classical engineering ('manufacturing') is trivial
  - make as many copies as you like
  - so just one phase: 'designing' and 'building' happen concurrently (more or less)
- is sometimes 'mission critical' (failure results in economic disaster or even death)
- never wears out
- MTBF very short (e.g. operating systems)
  - a reset' button on computers is essential
  - expectation of failure

Testing: continuous versus discrete

- test a bridge
  - drive a heavy train over it: if it holds ...
    - ... have proved everything lighter than a train can safely travel over the bridge
    - this is a law of gravity
  - testing in classical engineering is often based on physics
    - it is continuous world
- test software
  - there is no equivalent to the 'law of gravity'
  - software exists in the discrete world of logic
  - any value of any input variable may make the software fail
  - an infinite number of tests are needed to 'prove' safety

Definition of Software Engineering

```
The application of a systematic, disciplined, quantifiable approach to
the development, operation and maintenance of software; that is, the
application of engineering to software.
```

-- Institute of Electrical and Electronics Engineers (IEEE), 1993

Software falls at the cusp of:

- literature: a program is a readable 'specification' (with style, elegance, maintainable, extendable, ...)
- philosophy: a program is a logical 'flow of thought' that must be sound
- engineering: a program fulfills a function: it often *must* be correct, and *must* perform

A software engineer is a scientist as well as an engineer.

What is Computer Science?

- Wikipedia: definition of computer science
- What is *science*?
  - The key aspect is **classification**: things need to be classified
    - E.g. classification of species, minerals, clouds, stars ...
- What things do we classify?
  - Well, what do we do in computer science?

```
PROBLEM ---> ALGORITHM : PROGRAM : COMPUTER PROCESSES
```

- Classifying processes and programs doesn't buy us much - what does it mean?
- Classifying algorithms is useful.
- Classifying problems is also possible (which is surprising as "problem" seems so vague)

Classifying Algorithms

- What aspects of algorithms are we interested in?
  - speed (time it takes)
  - memory used
    - amount of memory
    - locality of reference
  - correctness (we are ONLY interested in correct algorithms)

Euclid and Algorithm Speed

Measuring the time it takes for a program to run for some input is NOT enough to tell you its speed.

- we classify algorithms by how the speed changes between different sized inputs: i.e. their relative speed in the operating range

For example, consider two algorithms to compute the Greatest Common Divisor (GCD) of x and y, assuming y is the smallest:

1. A **smart** algorithm to compute the GCD is:

```
if (y == 0) then GCD(x,0) = x
else GCD(x,y) = GCD(y, x mod y)
```

Example: x=56 and y=21

```
GCD(56, 21) ->
GCD(21, 56 mod 21) = GCD(21, 14) ->
GCD(14, 21 mod 14) = GCD(14, 7) ->
GCD(7,  14 mod 7)  = GCD(7, 0) ->
7
```

- This is called Eukleídēs' algorithm: it is the oldest known algorithm to appear in print (*300BC*).
  - The original proposition from the Greek mathematician, Eukleídēs, active in Alexandria (the *founder of geometry*):

When two unequal numbers are set out, and the less is continually subtracted in turn from the greater, if the number which is left never measures the one before it until a unit is left, then the original numbers are relatively prime.

- *What is this in C?*

2. A **brute force** algorithm to compute the the GCD is:

```
consider each number from 1 to x
   if it is a factor of x and a factor of y then record it
   the solution is the greatest such number
```

- This is a general style of algorithm called *exhaustive search*
- *What is this in C?*

Comparing Euclid and exhaustive search

To test the two ways of finding the GCD, we will:

- execute Euclid's' program on a **super slow** 1Hz computer (i.e. 1 instruction per second!)
- execute the *exhaustive-search* program on a **fast** 2.1GHz Mac (i.e. 2.1 billion instructions per second)
  - this machine is 2.1 billion times faster than a 1Hz machine!

Here were the execution times:

	x	y	GCD	euclid(x,y) on a super	gcdExhaustive(x,y)
				slow computer	on a fast computer
1	121	55	11	1 sec	0.001 secs
2	12345	23456	1	90 secs	0.001 secs
3	1234567	2345678	1	~60 secs	0.005 secs
4	12345678	23456789	1	~120 secs	1.5 sec
5	123456789	234567890	1	~120 secs	17 secs
6	1234567890	2345678901	9	120 secs	180 secs
7	12345678900	23456789000	100	210 secs	2040 secs

What's the moral of the story?

What is the performance difference between the two algorithms on CSE's 'wagner'?

We compute the GCD of *x*, an 18 digit number, and a series of numbers *y* ranging from 10 digits (10^9) to 18 digits (10^{17}).

x	y	GCD	euclid(x,y)	gcdExhaustive(x,y) (compiled with *dcc -O3*)
876,543,210,987,654,321	10^9	1	0.02 secs	10.01 secs
876,543,210,987,654,321	10^{10}	1	0.02 secs	$9.25 * 10^1$ secs (1min 32.5secs)
876,543,210,987,654,321	10^{11}	1	0.02 secs	$9.45 * 10^2$ secs (15min 45secs)
...				
876,543,210,987,654,321	10^{17}	1	0.02 secs	$\sim 9 * 10^8$ secs (31.7yrs)†

† Estimated by extrapolation of course

In the last test case, how many modulus operations does the exhaustive algorithm do?

- The gcd is 1 (so there are no common factors), the smallest number is 10^{17}, and there are 2 operations per iteration
  - number of operations is $2 * 10^{17}$

In the last test case, how many modulus operations does the Euclid algorithm do?

- Each recursive call does a single modulus operation.
- 25 recursive calls are required to compute *euclid(876543210987654321, 10^{17})*.

Rec. call	*x*	*y*
	876543210987654321	100000000000000000
1	100000000000000000	76543210987654321
2	76543210987654321	23456789012345679
...	...	...
22	68587	27435

23	27435	13717
24	13717	1
25	1	0

- number of operations is 25
  - could make it faster by using a loop, but clearly that is not a factor here
    - in fact, most of the 0.02 secs will be recursion 'housekeeping'

What about in general, for any pair of numbers *(x,y)* for $x \geq y$

How many modulus operations does the exhaustive-search algorithm take in general?

- Look at the loop in *gcdExhaustive(x,y)* again:

```
切换行号显示

   1      for (long i=1; i<=smallest; i++) { // go only as far as
the smallest
   2          if (x%i == 0 && y%i == 0) {
   3             divisor = i;                  // found another one
   4          }
   5      }
```

  - this loop does *2*y* modulus operations, where *y* is the smallest of the pair
- The loop in the early-exit version goes in the opposite direction, and exits early

```
切换行号显示

   1      for (long i=smallest; i>=1 && !divisor; i--) { //
decrement, halt when divisor is nonzero
   2          if (x%i == 0 && y%i == 0) {
   3             divisor = i;                  // found the greatest
   4          }
   5      }
```

  - it also does *2*y* operations (in the worst case when gcd =1)
    - (but if *x* is a multiple of *y* in needs just 2 operations (*Can you see why?*)

How many modulus operations does Euclid's algorithm take in general?

What is the best-case behaviour of Euclid's algorithm?

- in both of the following cases Euclid's algorithm calls the modulus operation just once
  - *euclid(x,1) = euclid(1,0) = 1*
  - *euclid(n*x,x) = euclid(x,0) = x where n=1,...*

What is the worst-case behaviour of Euclid's algorithm?

- Euler's algorithm in essence states (forgetting the boundary condition for the moment) that

  GCD(x,y) = GCD(y, x mod y)
  - we saw above that the **number of operations exhaustive search performs is a function of *y* only**
  - it is 'easy' to see that this will also be the case for Euclid's algorithm. Can you see why?

Recursive vs Iterative Euclid

Euclid's implementation in C was recursive.

- we called it a 'smart' algorithm
- ... but recursion is never smart from an efficiency point of view
    - at run-time, a function clones itself in memory for each recursive call
        - very expensive in time and space
- as it stands, *euclid.c* may be conceptually smart, but it is practically dumb

Can we make Euclid conceptually and practically smart?

Easy: replace recursion by iteration.

```
切换行号显示

   1 int euclidRecurse(int x, int y) {            int euclidIterate(int
x, int y) {
   2     if (y == 0) {                                while (y != 0) {
   3         return x;                                    int tmp = x;
   4     }                                                x = y;
   5     else {                                           y = tmp%y;
   6         return euclidRecurse(y, x%y);            }
   7     }                                            return x;
   8 }                                            }
```

The difference in:

- space
    - recurse: a clone of the function in memory for each step (quickly exhausts memory)
    - iterate: 1 extra variable *tmp* (an *int* here)
- time
    - recurse: time to clone the function for each step (parameter passing results in $x \leftarrow y$ and $y \leftarrow x\%y$)
    - iterate: one extra assignment ($tmp = x$) for each step (as well as assignments $x \leftarrow y$ and $y \leftarrow tmp\%y$)

The number of steps in the algorithm is crucial:

- in Euclid's algorithm, it is tiny (remember: less than Lamé's number)
    - so the difference between Euclid recurse and Euclid iterate is too small to measure

In 'serious' algorithms, the number of steps is often as big as the input, or much much larger

- ... making recursion not only impractical, but impossible

Timing programs in UNIX

The *time* command returns:

- **real** = time you wait (not useful, systems multitask), also called clock time
- **user** = CPU time when your program is actually running
- **sys** = CPU time taken in system calls (your program is not running but is still using time)

Hence the **user** + **sys** time is the actual time taken by the program.

Examples:

- *date* is a UNIX command that returns the date.

```
prompt$ date
Fri 10 May 2019 17:56:53 AEST
```

- You could time the date command.

```
prompt$ time date
Fri 10 May 2019 17:58:53 AEST

real    0m0.047s
user    0m0.002s
sys     0m0.005s
```

- *sleep* is a UNIX command
  - *sleep 3* will make UNIX 'sleep' for 3 seconds (the prompt will not return for 3 seconds)

```
prompt$ time sleep 3

real    0m3.006s
user    0m0.000s
sys     0m0.004s
```

- You can time the compiler.

```
prompt$ time dcc euclid.c

real    0m1.581s
user    0m0.912s
sys     0m0.308s
```

- You can time the executable.

```
prompt$ time ./a.out

real    0m0.022s
user    0m0.004s
sys     0m0.012s
```

- You can even *time* time.

```
prompt$ time time

real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

  - ... and of course *time* time time

```
prompt$ time time time

real  0m0.000s
user  0m0.000s
sys   0m0.000s
```

Lec01Intro (2019-06-06 10:30:18由AlbertNymeyer编辑)