Programming, Problem Solving, and Abstraction

Chapter Nine

Problem Solving

Lecture slides prepared by Alistair Moffat

Chapter 9 – Concepts

- Problem solving techniques.
- Generate and test.
- Divide and conquer (the wonders of recursion).
- Randomized approaches.
- Approximation.
- Simulation.

Problem solving

All programming languages offer mechanisms for calculation, selection, iteration (perhaps via recursion), and abstraction.

Most languages also offer facilities for aggregating dissimilar types into structures or records, and for collecting sets of similar type variables into arrays (or sometimes lists).

These techniques are the basic elements of control and data structures – the nuts and bolts of program construction.

Problem solving

But how are programs designed? Where do the algorithms come from?

And how do we know how to tackle a particular problem?

This chapter considers some standard techniques that can be used to solve a range of problems.

9.1 Generate and test

If it is easy to check a candidate answer, then generate and test may be appropriate.

The set of candidates needs to be ordered in some way, so that a program can sequentially test until it finds a solution.

9.1 Generate and test

For example, to find the next prime number bigger than a given value, candidates are tested in order until one is found.

- `isprimefunc.c`

9.2 Divide and conquer

In the divide and conquer strategy, a "smaller" subproblem is solved. That solution is then extended to create a solution to the original problem.

For example, binary search determines if a given item is in a sorted array:

- check the middle item
- if sought item is smaller, check (only) the first half
- if sought item is larger, check (only) the second half

Divide and conquer algorithms often have elegant recursive implementations.

9.2 Divide and conquer

- `binarysearch.c`

Binary search in an array of n items takes time that can be described recursively:

$$\begin{aligned} T(n) &= 1 && \text{if } n \leq 1 \\ T(n) &= 1 + T(n/2) && \text{if } n > 1 \end{aligned}$$

In this case the solution is $T(n) = \lceil \log_2 n \rceil$.

If $n = 10^6$, binary search requires 20 iterations (or recursive calls). If $n = 10^9$, binary search requires 30 iterations.

9.2 Divide and conquer

The Towers of Hanoi problem provides another famous recursion. A set of disks of different diameters are given, and three pegs.

The disks must be moved one at a time; and any given moment, the disks on each peg must be in size order.

If there are n disks on tower A, what sequence of moves is required to transfer them to tower C?

- `hanoi.c`

9.2 Divide and conquer

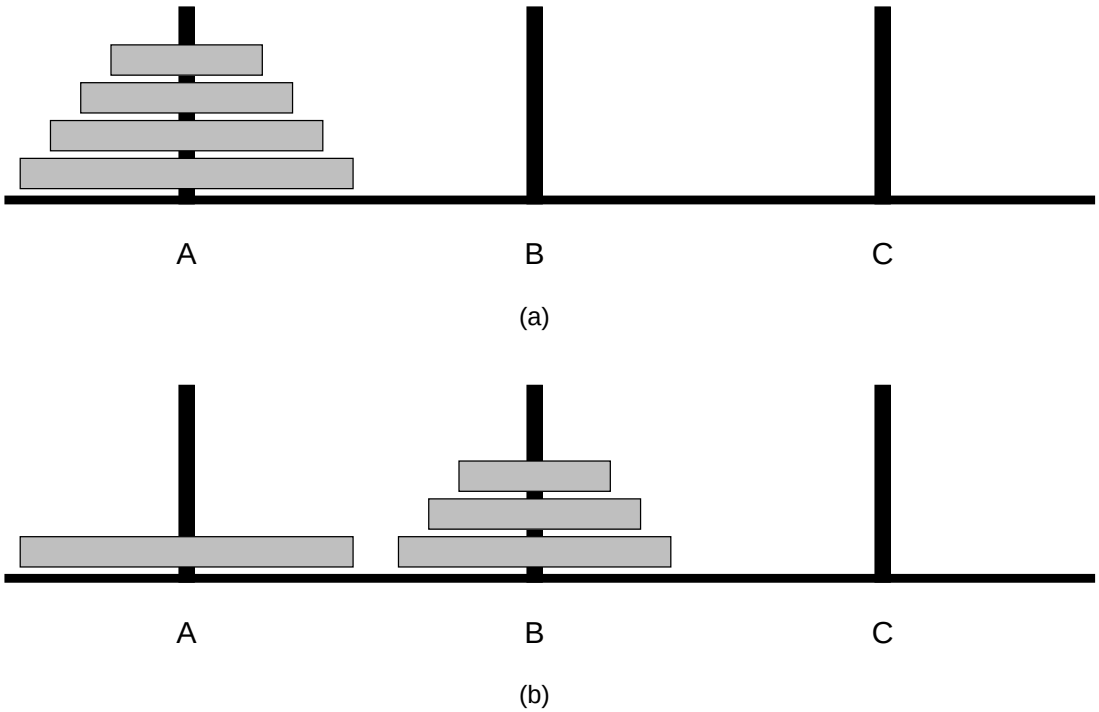
Use of solution of size $n-1$ to construct a solution of size n.

To move n disks from A to C:

- move $n-1$ disks from tower A to tower B, at all times complying with the constraints;
- move the biggest disk from tower A to tower C;
- move $n-1$ disks from tower B to tower C.

During the first move, tower C can be used as a temporary holding area. During the final move, tower A can be used.

9.2 Divide and conquer



Before the initial call to `hanoi('A','B','C',4)`; and then after the first recursive call to `hanoi(from,to,via,3)`.

9.2 Divide and conquer

The largest disk moves just once; the smallest is moved every second move.

To transfer n disks, $2^n - 1$ moves are required. If $n = 20$, and one move is made every second, it takes 11 days. If $n = 35$, it takes 1,000 years.

At a billion operations per second, $n = 65$ takes 1,000 years.

9.2 Divide and conquer

Is there a subset of the following numbers that adds up to exactly 1,000?

```
 34,  38,  39,  43,  55,  66,  67,  84,  85,  91,
101, 117, 128, 138, 165, 168, 169, 182, 184, 186
```

In general, given n integer values, and a target value k, is there a subset of the integers that adds up to exactly k?

What problem solving technique might be used? Generate and test? Or divide and conquer? Or both?

9.2 Divide and conquer

Evaluate all subsets of the `n` items, and if any one of them adds to `k`, return "yes".

Either the last value `A[n-1]` is part of the sum, or it is not; and if it is, a subset sum on the first `n-1` items in the array must add to `k-A[n-1]`.

- `subsetsum.c`

9.2 Divide and conquer

On a 2.7 GHz MacBook Pro (2012), seconds of CPU time as a function of the size of the array:

Size n	Time (seconds)
30	5.4
31	10.8
32	21.7
33	43.2
34	87.0
35	173.0

The running time is proportional to 2^n.

9.2 Divide and conquer

Extrapolating from these figures: two days when $n = 45$; five years when $n = 55$; and five millennia when $n = 65$.

There are problems for which all known algorithms require exponentially growing time.

That is, it may be relatively straightforward to implement an algorithm for some problem, but impossible to execute it!

9.2 Divide and conquer

The greedy heuristic is one particular kind of divide and conquer mechanism.

At each step a greedy process extracts, or deals with somehow, the local option that gives the biggest gain at that step, in the expectation that if a sequence of locally greedy choices are made, then a globally good solution emerges.

9.3 Simulation

The third major problem solving strategy is simulation.

Consider this gambling game: you bet one dollar, and roll two dice. If the total is between eight and eleven, you get $2 back. If the total is twelve, you get $6 back. Otherwise you lose your stake.

Is it worth playing? If you start with a $5 float, and play games until you have either $0 or $20, how many games will you play on average before leaving?

9.3 Simulation

No need for a statistician – we can get a good approximation by simulating the process, and using a random number generator to mimic the rolls of the dice.

- `gamble1.c`

9.3 Simulation

Function `srand` seeds the random number generator.

Each call to `rand` then returns an integer equally likely to be any value between zero and `RAND_MAX`. The header file `stdlib.h` is required.

From a given seed the sequence of random values is always the same, they are pseudo-random.

The seed should be varied each time the program is executed if the simulation is to be used to estimate averages.

9.3 Simulation

Random coins, dice, and floats are easily produced using `srand` and `rand`.

Then, by looping thousands or millions of times, accurate answers can be computed.

- `random.c`
- `gamble2.c`

9.3 Simulation

As another example of the use of randomness, consider the problem of estimating the area inside a complex curve, when all that is available is a function `inside` for deciding if a point is inside or outside the curve.

To estimate the inside area, test a large number of random (x, y) locations in a rectangle of known area.

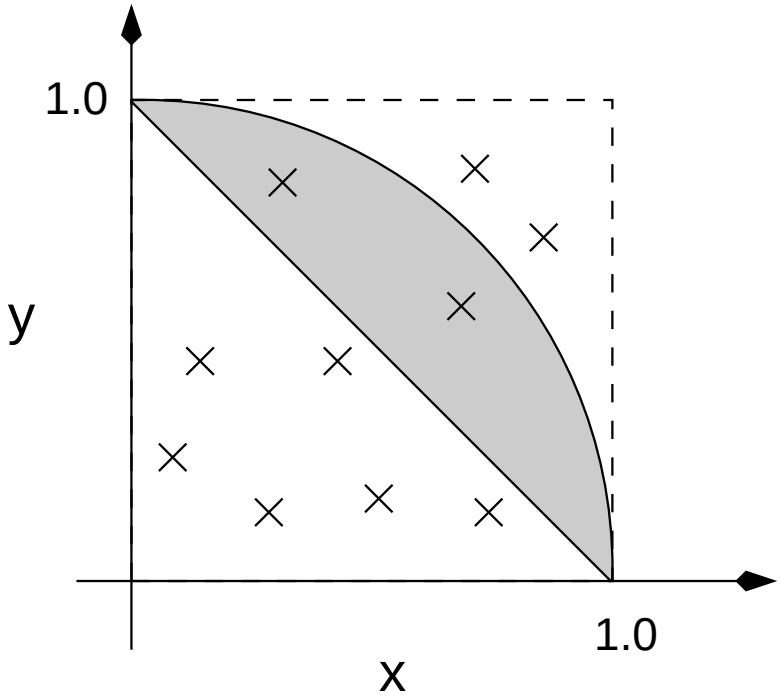
9.3 Simulation

The ratio of "inside" points to total points gives an estimate of the ratio of the inside area to the total area. Repeating millions of times shows the convergence of the estimate.

- `montecarlo.c`

In this example the correct answer is $\pi/4 - 0.5 = 0.2854...$

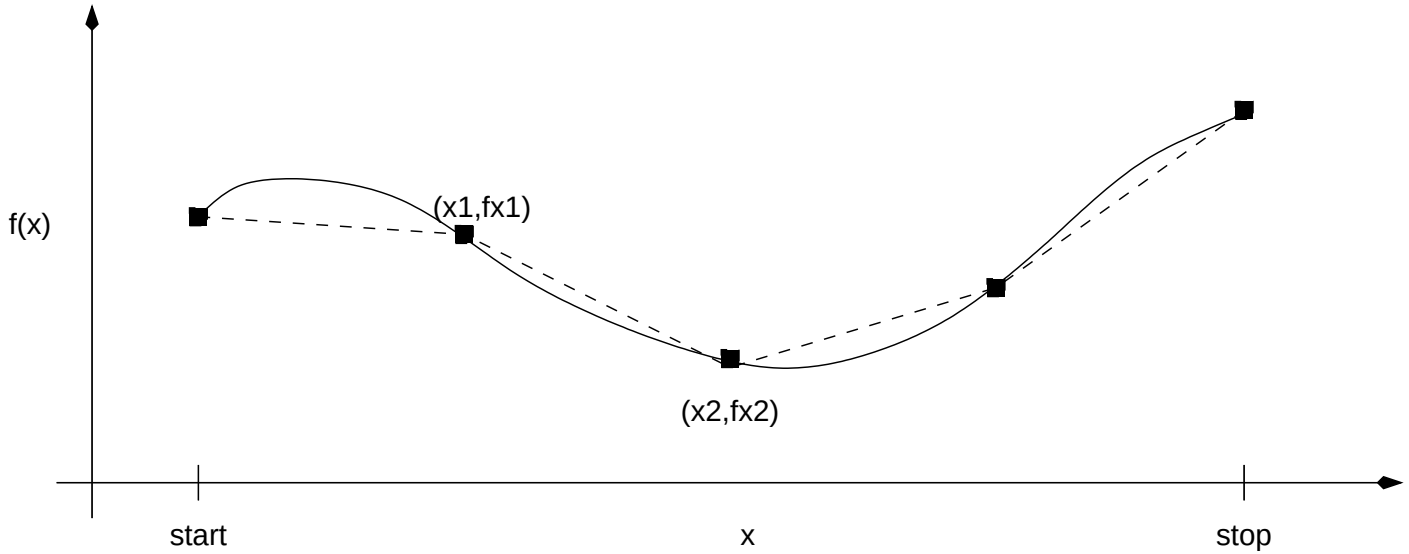
9.3 Simulation



Could also iterate over every location in a mesh of points.

9.4 Approximation techniques

The length of a curve can be estimated by approximating it using a sequence of straight lines.



9.4 Approximation techniques

Using a variable number of steps shows the convergence.

Trying to make the estimate too precise by using a very small step size can lead to accumulated rounding errors, and answers that are less precise.

- `linelength.c`

9.4 Approximation techniques

A similar process can be used to calculate an estimate of the area under a curve, and is called numerical integration.

Again, care needs to taken to ensure that the step size and the rounding errors are in balance.

9.4 Approximation techniques

Another numeric problem is that of root finding – determining the point at which a function crosses the x axis.

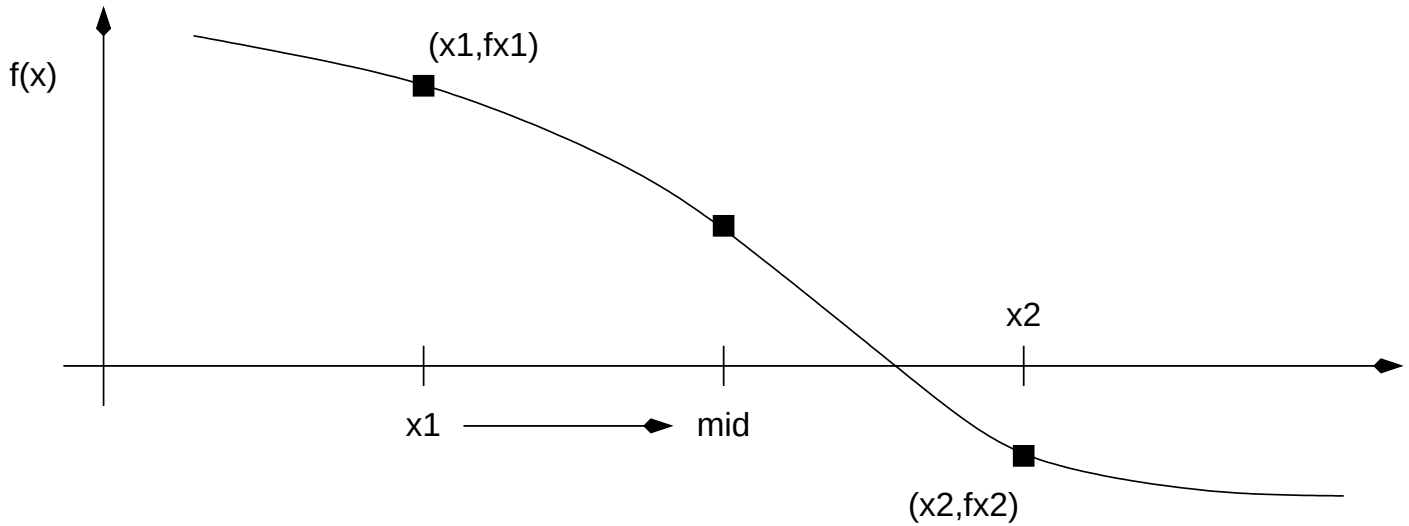
A brute-force approach could be used, stepping along the curve, looking for a change of sign.

A better algorithm uses bisection to rapidly locate an approximation of the root. At each step the mid-point of the current range is tested.

- `bisection.c`

9.4 Approximation techniques

The root is kept sandwiched between two x values at which $f(x)$ has opposite signs.



9.4 Approximation techniques

Termination needs to be judged by two different conditions:

- Whether the root has been found to the required precision (and not by a test of the form `f(x)==0.0`), and
- Whether some preset number of iterations has been exceeded.

The latter is necessary to prevent endless non-converging computation caused by a too-tight tolerance on the approximation.

9.5 Physical simulations

In some approximations the "stepping" is done through time.

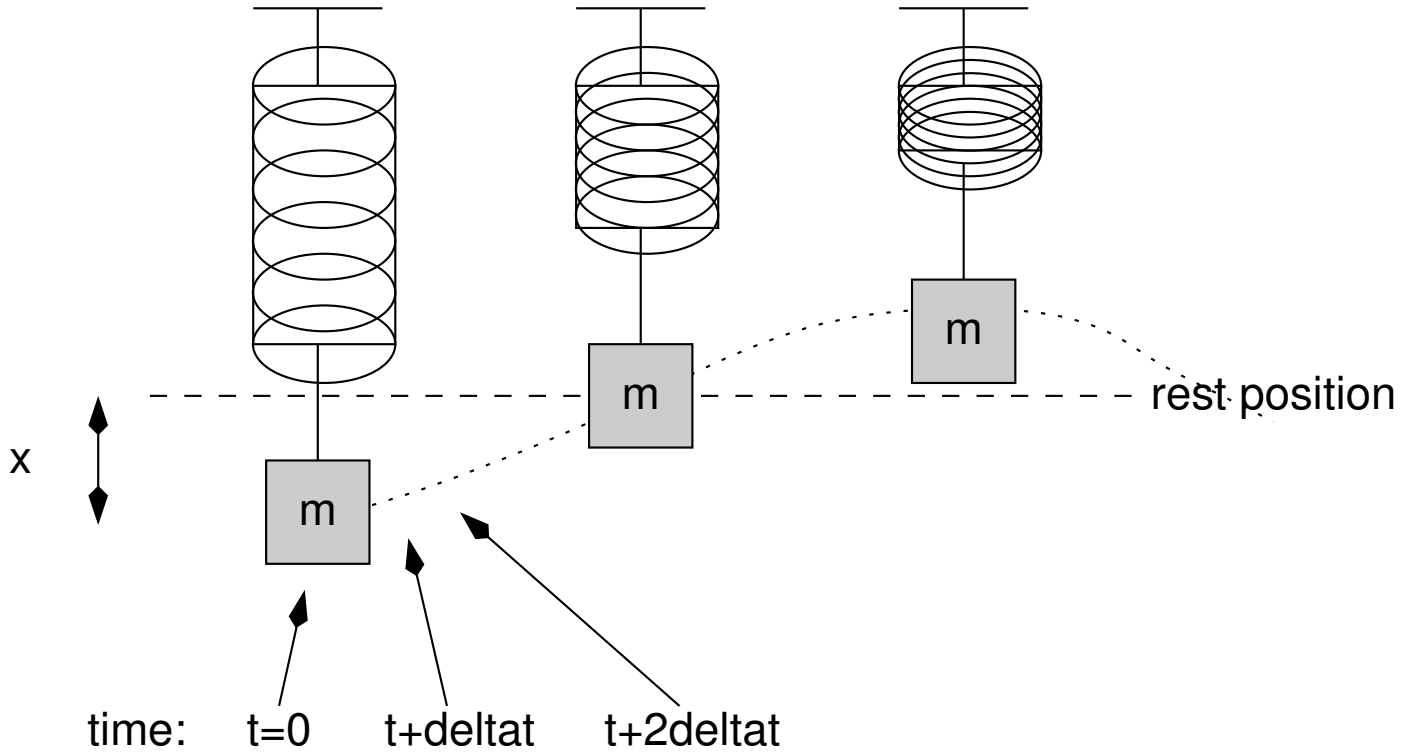
At each instant t, state variables give an estimate of the current location and velocity (and perhaps orientation, axis of rotation, and angular velocity) of all active objects.

Using the current state, an estimate is made for values of all state variables at time $t + \Delta t$.

This process is repeated. Errors will always accumulate, and may make the simulation unstable.

9.5 Physical simulations

Consider an object suspended on a spring.



9.5 Physical simulations

The governing equation is

$$\frac{d^2x}{dt^2} = \frac{1}{m}\left(-kx - c\frac{dx}{dt}\right),$$

where k is the spring constant, c is the damping force (friction, or air resistance), and x is the displacement.

What happens from time $t = 0$ if the object is displaced and then released?

- `spring.c`

9.6 Solution by evolution

Many of the problems you are asked to solve will be similar to ones you have dealt with previously.

Modify one of your previous solutions to fit the new problem, or adapt one from a book.

For example, sorting into decreasing order, or using a key based upon "dd/mm/yyyy" dates, does not require you to re-invent a sorting algorithm.

Chapter 9 – Summary

- Generate and test
- Divide and conquer
- Simulation
- Approximation
- Evolution.