Data Structures and Algorithms
Algorithm Design
Greedy Methods and Divide & Conquer

See references in Goodrich & Tamassia to Greedy Methods & Divide & Conquer

- Optimisation Problems
- Greedy Methods
- Proving Greedy Methods Optimal
- Divide & Conquer

> Programs = Algorithms + Data Structures
> Niklaus Wirth

Algorithms are often designed using common techniques, including:

- Greedy Methods
- Divide & Conquer
- Dynamic Programming
- Backtracking
- Brute Force
- Branch & Bound
- Linear Programming
- Integer Programming
- Neural Networks
- Genetic Algorithms
- Simulated Annealing
- ...

Optimisation Problems

Comprise

- A set of **constraints**
- An **optimisation function**

A **solution** that satisfies the constraints is **feasible**. A feasible solution with the best possible value for the optimisation function is **optimal**.

Change Making

Intuition: attempt to construct a sum using the minimum number of coins.

A set of coin *denominations*, e.g. $\{1, 2, 5, 10, 20, 50\}$

A *target*, e.g. $67p$

A **solution** is a sequence of coins $c_i \in denominations$, e.g. $[20, 20, 20, 2, 2, 2, 1]$ or $[20, 20, 20, 5, 1, 1]$

The **constraint** is $\Sigma_i^n c_i = target$

The **optimisation function** is the length of the sequence, e.g. 7 or 6

The **optimal solution** has the minimum sequence length, i.e. 6

Many real world problems can be formulated as optimisation problems. **Operations Research** studies such problems, amongst others, for large organisations like Airlines or Strategic commands etc.

Example problems:

- Find the shortest journey, visiting Edinburgh, Newcastle, Carlisle, Inverness, Glasgow.
- Find the maximum volume of arbitrary-sized crates that can be packed into a container.

Greedy Algorithm Design Technique

Key idea: Attempt to construct the optimal solution by repeatedly taking the 'best' feasible solution.

Greedy Alg. 1: Change Making

Chose the largest possible denomination coin at each stage:

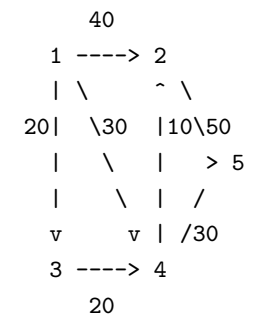
```
Target          Coin Selected
  67
```

```
Optimal solution =
```

Exercise: Trace the algorithm for target 48p, giving the optimal solution.

Greedy Alg. 2: Simple Shortest Path

Chose the *cheapest* edge to traverse from the end of the current path.



simpleShortestPath(1,5) =
length =

ShortestPath(1,5) =
length =

Dijkstra's Shortest Path: Choose the cheapest edge to traverse from *any* reached vertex.

Is the Greedy Algorithm Optimal?

It is usually easy to produce a greedy algorithm for a problem.

However the algorithm may not always find an optimal solution.

Hence it is essential to prove that a proposed algorithm does find an optimal soluion.

Change Making Correctness Proof (by contradiction)

Fifties, twenties, tens, fives, twos and pennies are currency denominations. Twenties are smaller denominations than fifties; and tens are smaller denominations twenties; etc.

Let F10, T10, T, F, TW and P respectively, be the number of fifties, twenties, tens, fives, twos and pennies in the change generated by the greedy algorithm. Let f10, t10, t, f, tw and p, respectively, be the number of fifties, twenties, tens, fives, twos and pennies in the change generated by an optimal algorithm.

We make the following observations:

1. From the way the greedy algorithm works, it follows that the total amount of change given in lower denominations is less than the value of the next higher denomination. That is, the change given in pennies is less than 2p; the change given in pennies and twos is less than 5p; etc. Therefore, T10 $<$ 3, T $<$ 2, F $<$ 2, TW $<$ 3, P $<$ 2.

2. For the optimal change, we can establish t10 $<$ 3, t $<$ 2, f $<$ 2, tw $<$ 3, p $<$ 2. To see this, note that if t10 >= 3, we can replace three twenties by a fifty and a ten and provide the change using one less coin. This is not possible as t10+t+f+tw+p is the fewest number of coins with which the change can be provided. Similarly for the other denominations. Hence, *the total amount of change given in lower denominations is less than the value of the next higher denomination.*

Now if F10 != f10, then either the greedy or the optimal solution must provide 50 pence or more in lower denominations. This violates the above observations. So, F10 = f10. Similarly, if T10 != t10, then either the greedy or the optimal solution must provide 20p or more in lower denominations which violates the above observations, so, T10 = t10. We can show T = t, F = f, TW = tw and P = p in a similar way. Therefore, the greedy and optimal solutions are the same.

Algorithm Design Technique 2: Divide & Conquer

Key idea:

1. **Divide** the problem into smaller independent subproblems
2. **Solve** the subproblems
3. **Combine** the solutions

Good for parallel programming

D&C1: Detecting a Counterfeit

Problem: you are given a balance and a bag of 16 coins, one of which may be counterfeit. Counterfeight coins are lighter than genuine coins.

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

Algorithms: 1. Naive: compare coin 1 with coins 2,3, ... 16
Max. No. comparisons =

2. Better: Weigh pairs
Max. No. comparisons =

3. Best:

1. **Divide** the coins into two sets of eight: A and B
2. **Solve:** weigh A and B
3. **Combine:** Counterfeit present iff (if and only if) A != B

Max. No. comparisons =

Exercise: Does this algorithm work if there are

- at most two counterfeit coins, i.e. 0, 1 or 2 counterfeits?
- 0 or 3 counterfeit coins?
- some natural number 0, 1, 2, .. 16 of counterfeit coins?

D&C2: Identifying the Counterfeit

Problem: locate the counterfeit coin in the scenario above.

1,2,3,4,5,6,7,8,9,10,11,12,13,*14*,15,16

Algorithms 1, and 2 from previous slide work, requiring at most 15 and 8 comparisons respectively.

Exercise: How many comparisons to identify 14 as the counterfeit using Algorithm 1? And using Algorithm 2?

1. **Divide** the coins into two equal-size sets : A and B
2. **Solve:** weigh A and B, lighter set contains counterfeit, so apply D&C to it
3. **Combine:** return counterfeit coin

Max. No. comparisons = 4

D&C3: Quicksort

Quicksort was devised by C.A.R. Hoare and sorts a sequence (list/array/file/etc) into ascending order as follows.

1. **Divide:** Pick an element of the sequence (typically the first element) as the **pivot**. Split the sequence into two: `his`: all elements greater than the pivot, and `los` all elements smaller than the pivot.
2. `Solve:` Apply D&C to `his` giving `hiSort` and apply D&C to `los` giving `loSort`
3. `Combine:` Append `loSort`, pivot and `hiSort`

Backtracking

Key idea: First organise the candidate solutions to make searching easy, e.g. a graph or tree. Then search the solution space depth-first, and if the node at the end of the current search proves infeasible, mark it as dead and backtrack to the previous node.

Live nodes are part of the current solution under investigation (often on a path).
Dead nodes are part of solutions that have proved to be infeasible.

More example backtracking algorithms in Weiss Section 10.5

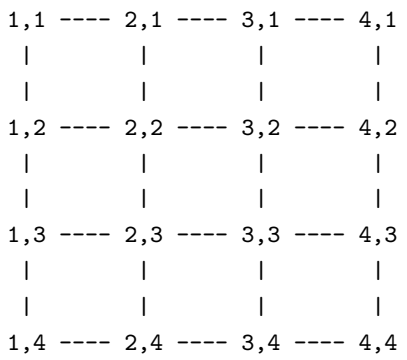
Maze Search Problem

Find a path through a maze from top left (1,1) to bottom right (4,4) avoiding obstacles

Maze is represented as a matrix:

	1	2	3	4
1	0	0	0	0
2	0	1	1	1
3	0	0	1	0
4	0	0	0	0

Solution space is an undirected graph:



Every path from (1,1) to (4,4) is a solution, but some cross obstacles, and are not feasible.

Search Objective: to find a feasible path from top-left to bottom-right.

A Backtracking Alg: Maze seach

DFS tries to move Right, Down, Left, Up

Depth First Search

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

Backtrack!

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

Depth First Search

0	0	0	0
0	1	1	1
0	0	1	0
0	0	0	0

Path:

Note: Backtracking is easily implemented in Logic languages like Prolog, and in *lazy* functional languages like Haskell.

Exercise: Trace the algorithm for the following maze.

0	0	0	0
0	1	0	1
0	1	1	1
0	0	0	0

Branch&Bound

Key idea: Use a breadth first search of the solution space, but **prune** suboptimal solutions

Good for parallelism: there are many solutions to consider simultaneously

But ... multiple solutions consume resources, including memory

Branch&Bound Alg: Maze Search

```
Breadth First Search

  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    2 paths


  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    2 paths
```



```
  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    3 paths

Top path is infeasible
Prune: chose shorter, or arbitrarily
       between equal-length
  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    2 paths
```



```
  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    1 paths


  0   0   0   0
  0   1   1   1
  0   0   0   0
  0   0   1   0    1 path
```

Path:

Dynamic Programming

Key idea: Store optimal solutions to subproblems, and use them to solve bigger problems

Often used to produce an efficient version of a recursively specified problem

More examples in Weiss Section 10.3.2 onwards

Fibionacci Numbers

The fibionacci numbers are a sequence of numbers defined as follows (in SML)

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

i.e. 0,1,1,2,3,5,8,13,21,34

Fibionacci numbers have many properties, e.g. the sum of the squares of two consecutive fibionacci numbers is also a fibionacci number.

Naive Fib Function

A recursive definition is natural:

```
public static int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-1);
}
```


Efficiency of `fib`

`fib(5)`

Dynamic Programming Fib

Idea: store previous fibionacci numbers

```
public static int fib(int n)
{
  int [] fibs;
  fibs[0] = 0;               // Initialise array
  fibs[1] = 1;
  for (int i = 2; i <= n; i++)
    fibs[i] = fibs[i-1] + fibs[i-2];
  return fibs[n];
}
```


Concluding Thoughts

There are many clever algorithms and data structures out there.

Pick suitable algorithms and data structures for your problem, and try to reuse code.

If you implement your own algorithms and data structures: aim for generic code.

Happy Hacking!