Lecture 6
Algorithmic Techniques Part 1

COMP26120

Giles Reger

March 2019

Algorithmic Problem-Solving

Question: Given a computational task, how do we devise algorithms to solve it?

Algorithmic Problem-Solving

Question: Given a computational task, how do we devise algorithms to solve it?

Answer: Turn it into something you know how to solve.

Algorithmic Problem-Solving

Question: Given a computational task, how do we devise algorithms to solve it?

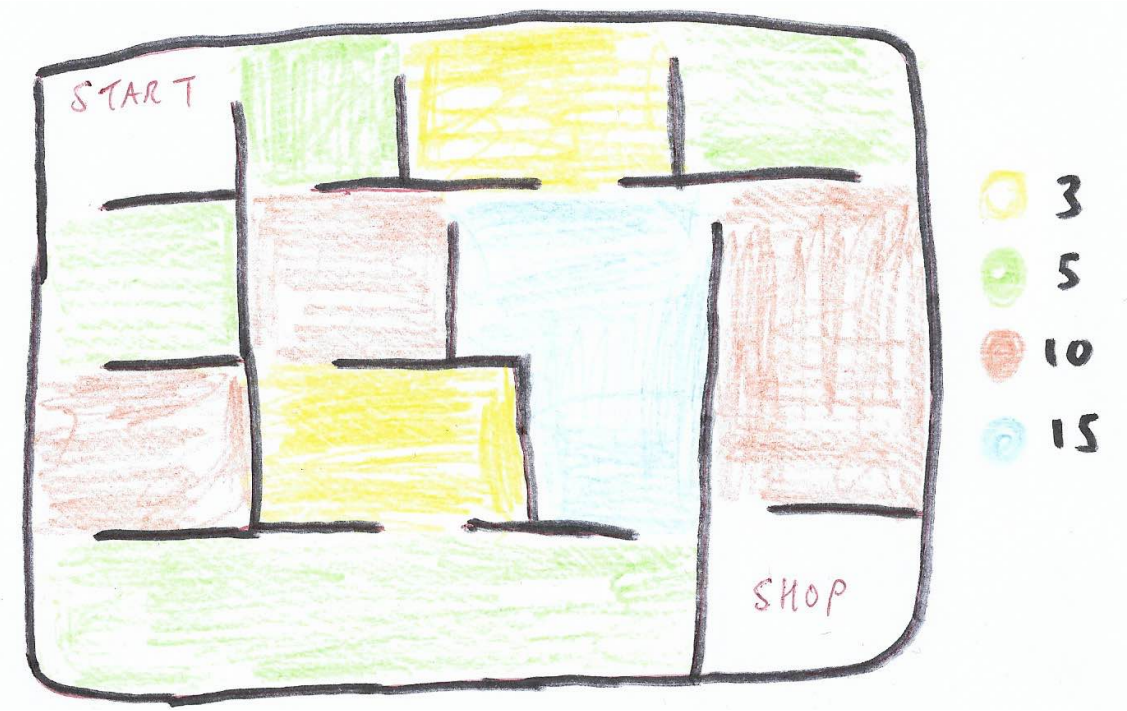
Answer: Turn it into something you know how to solve.

In reality:

- Step 1 Understand what the problem is
- Step 2 Get the problem into a form where you know how to solve it either by reducing to a known algorithm or applying a known technique
- Step 3 Solve it

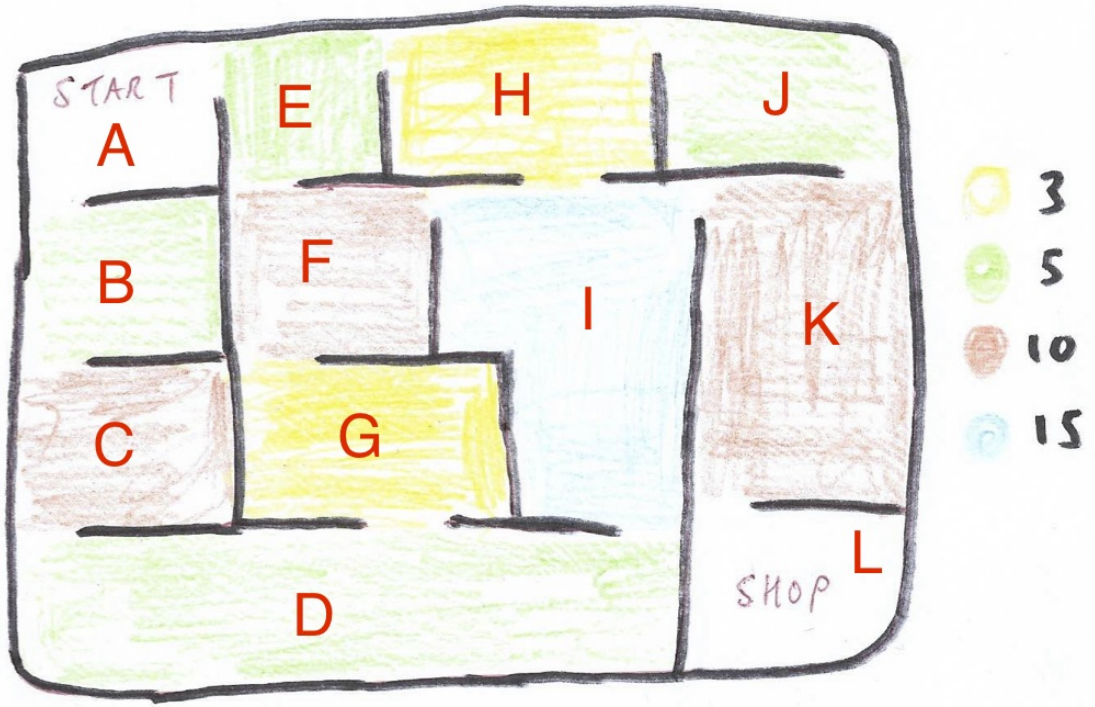
Art Gallery Visit

Our family is going to an Art Gallery but my Dad will want to look at every painting. I want to get to the shop as quickly as possible. I've colour- coded the map of the Gallery in terms of how many paintings there are in each section. How can I spend the least time in the Gallery?



Art Gallery Visit

Our family is going to an Art Gallery but my Dad will want to look at every painting. I want to get to the shop as quickly as possible. I've colour- coded the map of the Gallery in terms of how many paintings there are in each section. How can I spend the least time in the Gallery?



Organising Marking

I have a list of submissions that need marking. Each submission is of the form $(timestamp, studentid, work)$ where some students may have made multiple submissions. Student Ids are 4 digit numbers. The list is currently ordered by timestamp. I want to mark students in order of attendance in lectures, which I have also recorded. I need to mark the most recent submission. How do I get the list of submissions to mark in the right order?

Submissions	*Attendance*
$(496910, 2081, blob1)$	$(2081, 5)$
$(497420, 6440, blob2)$	$(7712, 0)$
$(498230, 2081, blob3)$	$(0423, 4)$
$(499100, 7712, blob4)$	$(6440, 4)$
...	...

Passing on a Message

I need to get a message from my office to all other offices in the building. The message can be passed on by somebody in one office going to another office and telling them the message. Each person only knows where some offices are and nobody wants to visit more than one office; they will go to another office and then back to their own. These trips will will waste some time (e.g. they won't be working) but some people gossip more than others so will waste different amounts of time.

I know

- Who is in which office $(pA, o1), (pB, o2), \ldots$
- Which offices a person knows about $(pA, o2), (pA, o3), (pB, o1) \ldots$
- How much time each person will waste $(pA, 4), (pB, 2), \ldots$

What's the least amount of time I need to waste?

The Birthday Party

My son is having a superhero-themed birthday party and wants to invite all of his friends. He wants everybody to have a superhero costume but some pairs of friends do not get along and cannot wear the same costume. He writes down a list of such pairs, e.g. $(a, b), (c, d), \ldots$. I get bulk discounts on costumes for the same superhero so want to get as few different superhero costumes as possible.

Known Algorithms

You already know how to do some things:

- Sorting
- Searching in lists, trees, graphs
- (Simple) Amortization arguments
- Graph algorithms such as Shortest Path
- Heuristic search
- Graph colouring
- Reduction to boolean constraint solving

Different Kinds of Problems

- Find a card with a number ≥ 5

Different Kinds of Problems

- Find a card with a number ≥ 5

- There may be many correct solutions

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?

- There may be many correct solutions

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?

- There may be many correct solutions
- There may be one unique solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16

- There may be many correct solutions
- There may be one unique solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16

- There may be many correct solutions
- There may be one unique solution
- There may be a best solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16
- Make 37

- There may be many correct solutions
- There may be one unique solution
- There may be a best solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16
- Make 37

- There may be many correct solutions
- There may be one unique solution
- There may be a best solution
- There may be more than one best solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16
- Make 37
- Make 4

- There may be many correct solutions
- There may be one unique solution
- There may be a best solution
- There may be more than one best solution

Different Kinds of Problems

- Find a card with a number ≥ 5
- Which is the card with the smallest number?
- Make 16
- Make 37
- Make 4

- There may be many correct solutions
- There may be one unique solution
- There may be a best solution
- There may be more than one best solution
- There may be no solutions

Algorithmic Toolbox

Find a/the solution

0. Brute Force (Enumeration/Backtracking)
1. Divide and Conquer

Also optimisation problems

2. Branch and Bound
3. Greedy Algorithms
4. Local Heuristic Search
5. Dynamic Programming
6. Linear Programming

Brute Force (Enumeration/Backtracking)

Often the simplest 'solution' is to enumerate all possible answers and search this enumeration

This is brute force as you solve it by applying effort rather than brain

Brute Force (Enumeration/Backtracking)

Often the simplest 'solution' is to enumerate all possible answers and search this enumeration

This is brute force as you solve it by applying effort rather than brain

Often the enumeration contains choices e.g. for graph colouring we could brute force by guessing a colour for a node and then exploring neighbours. But a previous choice might be bad, in which case we need to backtrack.

Backtracking is a way of organising the search of a structured solution space that can more quickly eliminate large parts of the space

However, backtracking is still often exponential

Divide-and-Conquer

Recall last semester (lecture 6) used in merge sort and binary search.

General idea:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough solve directly.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

This is a top-down approach (relevant next week).

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \qquad (\textit{assuming } n \textit{ is a power of } 2)$$

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \qquad (\textit{assuming } n \textit{ is a power of } 2)$$

We can rewrite $I \cdot J$

$$(I_h 2^{n/2} + I_l) \cdot (J_h 2^{n/2} + J_l)$$

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \qquad (\textit{assuming } n \textit{ is a power of } 2)$$

We can rewrite $I \cdot J$

$$I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \qquad (\textit{assuming } n \textit{ is a power of } 2)$$

We can rewrite $I \cdot J$

$$I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

Recursively split numbers in half to get 4 smaller subproblems

Solve $T(n) = 4T(n/2) + cn$ and get $\Theta(n^2)$ (bitwise shift is constant)

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \qquad (\textit{assuming } n \textit{ is a power of } 2)$$

We can rewrite $I \cdot J$ (using $(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h$)

$$I_h J_h 2^n + ((I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l) 2^{n/2} + I_l J_l$$

Divide-and-Conquer Example: Multiplying Big Integers

Consider big integers represented by more bits than processors can handle

Addition of n-bit numbers is $O(n)$ but naive multiplication is $O(n^2)$

To compute $I \cdot J$ we can represent I and J with their higher and lower bits

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned} \quad (\textit{assuming } n \textit{ is a power of } 2)$$

We can rewrite $I \cdot J$ (using $(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h$)

$$I_h J_h 2^n + ((I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l) 2^{n/2} + I_l J_l$$

Recursively split numbers in half to get 3 smaller subproblems

Solve $T(n) = 3T(n/2) + cn$ and get $\Theta(n^{log_2 3})$ which is $O(n^{1.585})$

Divide-and-Conquer - Applications

Divide-and-conquer is of very wide application. A few examples:

- Efficient sorting algorithms - both Mergesort and Quicksort have average time complexity of $O(N \times \log(N))$, whereas most simple general sorting algorithms are slower, $O(N^2)$.
- Fast integer multiplication: Integer multiplication by long multiplication is $O(N^2)$, but there are fast $O(N \times \log(N))$ divide-and-conquer algorithms.
- Fast matrix multiplication: Standard matrix multiplication is $O(N^3)$, divide-and-conquer algorithms produce algorithms $O(N^{2.808\ldots})$ (and even down to $O(N^{2.376\ldots})$).
- Nearest neighbour problems: Given a set of N points in 2D or 3D (or n dimensions), find two nearest points.
  Divide-and-conquer algorithm is $O(N \times \log(N))$.

Branch-and-Bound

Backtracking is for finding a/the solution not optimisation.

To extend this to optimisation instead of stopping when we find a solution we carry on until the best solution is found.

We also organise search using a scoring mechanism that returns a lower/upper bound on a branch and we use this to select the next branch to explore. This is also known as best-first search.

You will explore this idea properly in the Knapsack Lab.

Greedy Methods - An Example

Problem: Suppose we have a set of coins of various denominations (values) and we wish to pay for an item of cost V with a minimum number of coins.

How do we select the coins to do this?

Suppose (as in the UK) we have coins of values 1, 5, 10 and 20 pence, and we wish to pay for an item costing 37 pence.

We need to choose a minimum number of coins to do this. How?

Greedy Methods - An Example

Problem: Suppose we have a set of coins of various denominations (values) and we wish to pay for an item of cost V with a minimum number of coins.

How do we select the coins to do this?

Suppose (as in the UK) we have coins of values 1, 5, 10 and 20 pence, and we wish to pay for an item costing 37 pence.

We need to choose a minimum number of coins to do this. How?

Answer: Choose the biggest value coins that we can at each stage: choose a 20, then a 10, then a 5, then a 1, then a 1, and then we are done! 5 coins are needed and this is a minimum.

This is called a greedy strategy.

Greedy Methods - An Example (continued)

Question: Does a greedy strategy always work for this problem?

For these coin values, it always works. Why? Answer is not obvious!

Consider coin values 1, 10 and 6 and we have item costing 12.

Then the greedy strategy fails. We choose a 10 and then two 1s, to give 3 coins. But we could have chosen 2 coins of value 6.

The General Idea

Any optimisation problem can be though of as having multiple configurations where we have a function $f(x)$ that scores configurations and we are looking for the configuration with a maximum/minimum score.

A greedy algorithm progresses by always picking the best next configuration.

If the problem satisfies the greedy-choice property then this approach will always find the optimal solution.

The greedy-choice property is that a global optimal configuration can be reached by a series of locally optimal choices.

Examples: Prim's and Dijkstra's

You have already been greedy

Prim's algorithm for minimum spanning tree (iteratively add the next shortest path to the tree that connects a new node) is greedy

Dijkstra's algorithm for single source shortest paths (iteratively select the unvisited node with the shortest distance) is greedy

Examples: Task Scheduling

We have a set of tasks $(s_1, f_1), (s_2, f_2), \ldots$ with start times s_i and finish times f_i. Each task needs to be performed on a machine (which can run 1 task). How many machines do we need to schedule all tasks?

Examples: Task Scheduling

We have a set of tasks $(s_1, f_1), (s_2, f_2), \ldots$ with start times s_i and finish times f_i. Each task needs to be performed on a machine (which can run 1 task). How many machines do we need to schedule all tasks?

Sort the tasks by start time
Create a list *active* of length 0
For each (s_i, f_i) search *active* for a value $< s_i$ and replace it by f_i.
If there is none then expand *active*
The minimum number of machines is the size of *active*

Examples: Task Scheduling

We have a set of tasks $(s_1, f_1), (s_2, f_2), \ldots$ with start times s_i and finish times f_i. Each task needs to be performed on a machine (which can run 1 task). How many machines do we need to schedule all tasks?

Sort the tasks by start time
Create a list *active* of length 0
For each (s_i, f_i) search *active* for a value $< s_i$ and replace it by f_i.
If there is none then expand *active*
The minimum number of machines is the size of *active*

Can argue for optimality by contradiction argument

- If not optimal then we wrongly extended *active* for task i
- That means an active task j was finished, but then $f_j < s_i$
- But then we would not have extended *active*, contradiction

Greedy Methods for Optimisation Problems

A greedy strategy attempts to find a maximum (or minimum) solution, by maximising (or minimising) the value of the choice at intermediate stages.

- For some optimisation problems a greedy strategy is always successful.
- For some optimisation problems, a greedy strategy solves some instances but not others (as in the coins example above).
- For some optimisation problems, a greedy strategy may not give optimal solutions but may lead to good approximations
- For some optimisation problems, a greedy strategy is not applicable – no useful solutions are produced.