Notes - Algorithms & Data Structures - Greedy Algorithms - Part 1

- Dr Nick Hayward

A brief intro to greedy algorithms and associated solutions.

Contents

Intro

A key consideration for working with algorithms is the identification of problems that have no *fast* algorithmic solution.

An awareness of such *NP-complete* problems is a particularly useful skill to develop, and certainly beneficial in algorithm design and development.

To help with such problems, we may often consider *approximation* algorithms. In effect, options we may use to quickly define an approximate solution to an *NP-complete* problem.

We may also consider *greedy* strategies, which provide simple options and patterns for the resolution of such problems.

Sample problems

To help us consider such problems, we may review some common examples to help conceptualise such resolution patterns.

For example, we may review the following well-known problems

- classroom scheduling problem
- knapsack problem
- set-covering problem
- ...

Classroom scheduling problem

A classroom is available for lectures, but we want to ensure we can schedule as many classes as possible during a defined time period.

In effect, we're interested in optimal use of resources within a finite, constrained period of time.

worked example

So, we may begin by defining each class and its current scheduled hours.

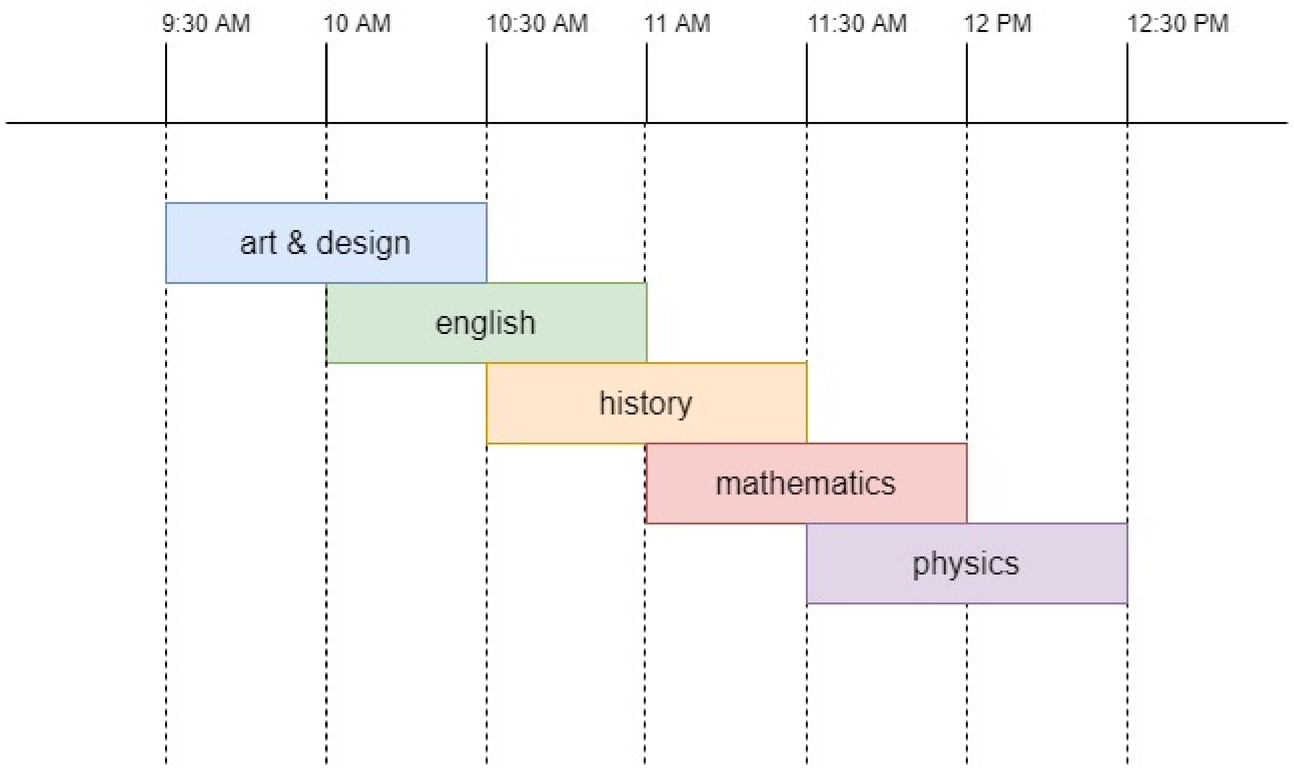
For example,

class	start time	end time
art & design	9:30 AM	10:30 AM
english	10 AM	11 AM
history	10:30 AM	11:30 AM
mathematics	11 AM	12 PM
physics	11:30 AM	12:30 PM

As we can see in this table, we cannot currently schedule each of these classes in the classroom. There are time overlaps, and scheduling issues.

We want to able to schedule as many classes as possible in this classroom. However, we need to manage the following schedule to ensure we fit the most classes in the current available time.

For example, our current schedule is as follows, including overlapping classes.



algorithm requirements

So, we have to define an algorithm to solve this problem for scheduling the classes.

However, whilst it may, initially, seem like a difficult problem to solve, the algorithm is deceptively simple.

For example, we may conceptually define this algorithm as follows

- select the class that ends soonest...
  - this is now the first class scheduled
- then, select a class that starts after this first class
  - again, choose the class that ends soonest...
- repeat this pattern until the schedule is full
  - no more class will fit...

If we apply this basic algorithmic solution, we may update our classroom schedule as follows.

1. art & design - 9:30 AM to 10:30 AM

From our current classes, we can see that *Art & Design* finishes soonest. So, we may add that to our updated schedule.

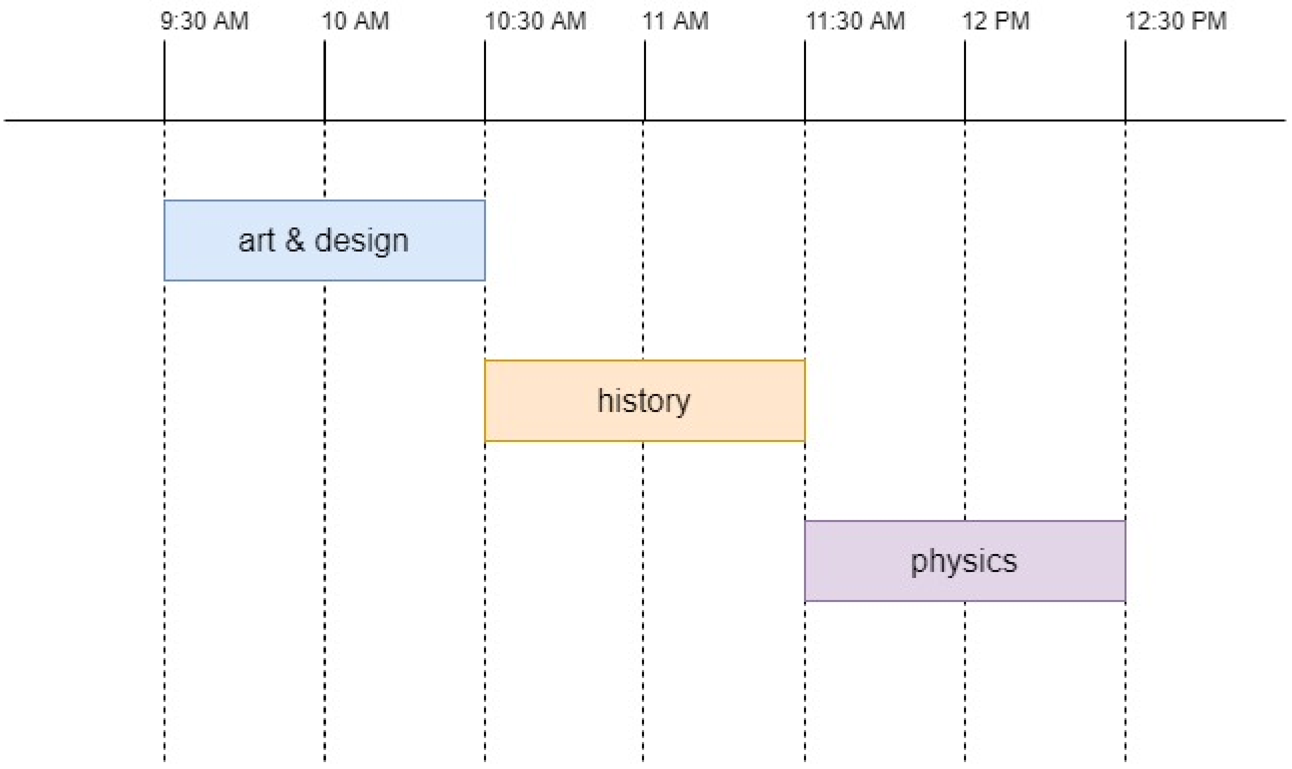
Then, we need to identify a class that starts after 10:30 AM and, again, ends the soonest of the available classes.

2. history - 10:30 AM to 11:30 AM

We repeat these checks and, again, update the schedule with the next class.

3. physics - 11:30 AM to 12:30 PM

So, we have now identified the classes we may schedule for this classroom during the available timescale.



Whilst this algorithm may appear overly simplistic for a difficult problem, we can see a clear benefit of *greedy* algorithms. They are easy to implement for such problems.

If we conceptualise a *greedy* algorithm, we can see that at each step we are, in fact, choosing the optimal selection.

For this particular worked example, we are simply picking a class, one that ends the soonest from the matching options.

As a developer, at each step of the algorithm we are, effectively, choosing the optimal *local* solution. This will then produce, at the end of the algorithm, a *globally* optimal solution.

So, we can see that this simple algorithm is now able to find the optimal solution to this scheduling problem.

Whilst *greedy* algorithms may not solve all problems, they are simple to write and test.

Knapsack problem

Another similar example is the *knapsack problem*.

Commonly perceived as an example of *resource allocation* and *combinatorial optimisation*, the knapsack problem is conceptually simple to define and understand.

Given a group of items, each with known value and weight, we need to determine the number of items we may fit in a given knapsack of fixed size and capacity.

In effect, we need to calculate the combined weight of these items to ensure an optimised collection is less than or equal to a set limit. Likewise, we need to ensure the combined value is as high as possible.

So, there are known constraints and requirements to allow us to calculate the optimal distribution of items, and the associated best use of the *knapsack*.

worked example

A common example for this problem is a burglar who needs to choose the best goods that will fit in their knapsack.

They need to grab a collection of items with the highest value, which they can carry in their bag.

For example, the knapsack is able to carry a weight up to 20 kilograms, approximately 44 pounds. So, we're trying to maximise the total value of the items carried in this bag.

algorithm requirements

If we consider an algorithmic solution, we might initially consider a *greedy* approach to solving this problem.

For example,

- begin by picking the item with the highest value that will fit in the bag
- then, pick the next expensive item that will fit in the bag
- then repeat...

However, this approach will not work for this example problem. If we consider the following items

item	weight	value
TV	15 kg	$2500
Computer	10 kg	$1500
Violin	7 kg	$1200

We know the bag can carry up to 20 kg of items. We can see the most expensive item is the *TV*, so we add that to the knapsack. However, it also weighs 15kg, which means we may not add any of the other items.

The bag currently has a weight of 15kg with a value of $2500. So, using this approach the highest value we may add is `$2500`.

However, we can clearly see that this is not the best combination of items. If we chose the *Computer* and *Violin*, the value of the knapsack would now equal `$2700`.

So, the *greedy* strategy does not give an optimal solution to this problem. However, if we consider the outcome, we can see that it comes very close to the optimal solution. In effect, a quick use of this strategy will often be good enough to solve such problems.

For many problems, all we need for an algorithm is that it will solve the problem quickly and to a good enough standard.

i.e. in this example, we only lost out on a potential $200, but the calculation was fast and easy to execute.

This type of scenario is where *greedy* algorithms prove very useful - easy to write, and quick to execute.

Set-covering problem

A related example for considering the use of *greedy* algorithms is commonly referred to as the *set-covering* problem.

It is another *NP-complete* problem, and particularly useful as we consider approximation algorithms in general.

The outline of the problem is, again, deceptively simple to consider and understand.

There is a defined set of elements and a collection of sets. These sets, when unified, is the same as the initial set of elements, commonly known as the *universe*.

The problem itself requires an identification of the smallest union of sets, which are known to be equal to the universe.

worked example

If we consider a problem to check for mobile internet coverage in a country, which is provided by a network of *base stations* in each state.

We need this internet coverage to be able to create a company that provides mobile data coverage for the whole country. We want to offer this service at the lowest possible cost, and this requires low setup and coverage costs. A customer should be able to use the service anywhere in the country with full coverage across each of the country's states.

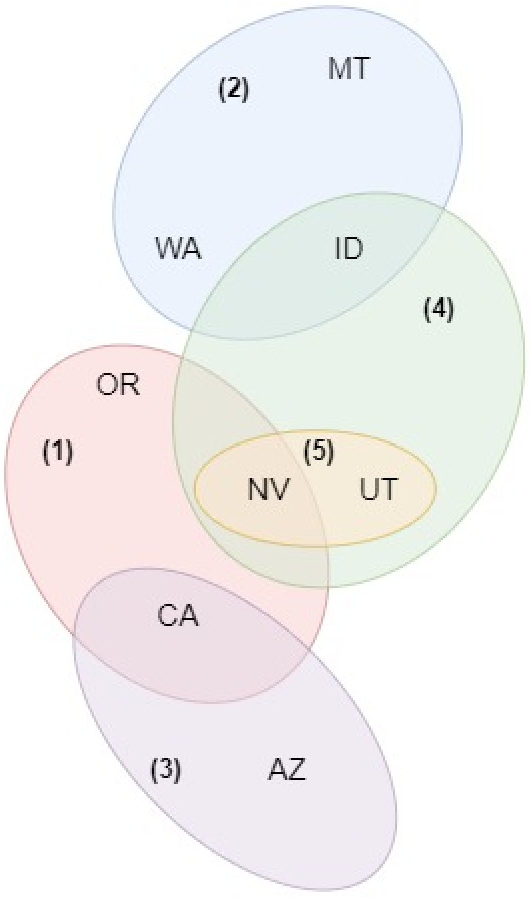
So, we're effectively trying to minimise the number of *base stations* we need to be able to create a working, country-wide network.

We begin by compiling a sample of *base stations* available to our company and network.

For example,

base station	**state coverage**
station one	OR, NV, CA
station two	WA, ID, MT
station three	CA, AZ
station four	ID, NV, UT
station five	NV, UT
...	,,,

We can clearly see that each station covers a given region of states, and there is also some overlap between stations and states.



So, we need to calculate the smallest set of *base stations* to cover the required country area.

Initially, this may seem a simple problem to solve, but it is a difficult and time consuming problem to resolve.

algorithm requirements

To solve this problem, we may use the following initial outline to determine a set of *base stations*.

- define each and every available subset of base stations for the given coverage area
  - commonly known as the *power set*
  - there are `2^n` possible subsets for this problem
- choose the set with the smallest number of *base stations* that meet the coverage requirements for the defined area
  - e.g. base stations for the country

The problem, of course, is not the calculation itself. It takes a long time to calculate each and every potential matching subset of stations.

In fact, it takes `O(2^n)` time, because, as noted, we are dealing with `2^n` base stations. This calculation will be feasible for a smaller set of base stations.

However, we can see how this time quickly becomes impractical and no longer a working solution to this problem.

For example,

number of base stations	required calculation time
5	3.2 seconds
10	102.4 seconds
100	4 x 10^{21} years
...	....

So, we need to find a way to deal with such problems that provide a working approximation in a time useful for practical application.