Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 14 - Part 1

Dr Nick Hayward

Final Assessment

Course total = 30%

- continue to develop your app concept and prototypes
  - *working app*
    - must implement algorithms and data structures
  - *explain design decisions*
    - describe patterns used in design and development of app
    - structures, organisation of code and logic
  - *explain testing and analysis*
  - *show and explain implemented differences from DEV week*
    - where and why did you update the app?
    - perceived benefits of the updates?
  - *how did you respond to peer review?*
- anything else useful for final assessment...
- consider outline of content from final report outline
- ...

All project code must be pushed to a repository on GitHub.

n.b. present your own work contributed to the project, and its development...

Final Report

Report due on Thursday 30th April 2020 @ 11.15am

- final report outline - coursework section of website
  - *PDF*
  - *group report*
  - ***extra individual report*** *- optional*
- include repository details for project code on GitHub

greedy algorithms - intro

- a key consideration for working with algorithms
  - *identification of problems that have no fast algorithmic solution*
- awareness of such *NP-complete* problems
  - *a particularly useful skill to develop*
  - *certainly beneficial in algorithm design and development*
- to help with such problems
  - *often consider approximation algorithms*
- i.e. options we may use to quickly define an approximate solution
  - *e.g. to an NP-complete problem*
- may also consider *greedy* strategies
  - *provide simple options and patterns for resolution of such problems*

Video - Algorithms and Data Structures

NP-complete problems - intro

Algorithms - NP-Complete Problems - intro - UP TO 36:02

Source - Algorithms - YouTube

greedy algorithms - sample problems

- to help us consider such problems
  - *review some common examples to help conceptualise such resolution patterns.*
- e.g. review the following well-known problems
  - *classroom scheduling problem*
  - *knapsack problem*
  - *set-covering problem*
  - *...*

classroom scheduling problem

- a classroom is available for lectures
- want to ensure we can schedule as many classes as possible
    - *schedule during a defined time period*
- i.e. interested in optimal use of resources
    - *within a finite, constrained period of time...*

Algorithms and Data Structures

classroom scheduling problem - worked example - part 1

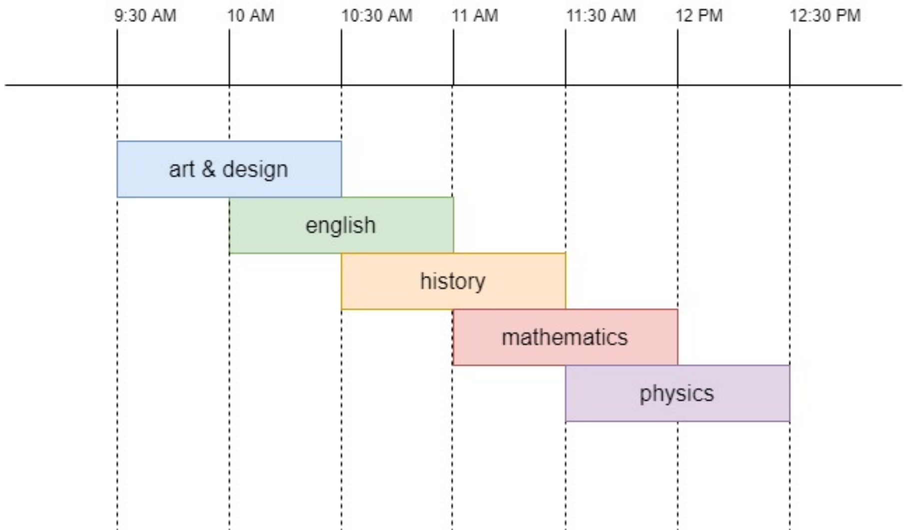
- begin by defining each class and its current scheduled hours
- e.g.

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
  - *cannot currently schedule each of these classes in the classroom*
  - *there are time overlaps*
  - *and scheduling issues...*

classroom scheduling problem - worked example - part 2

- want to able to schedule as many classes as possible
  - *i.e. in this classroom*
  - *need to manage following schedule*
  - *ensure we fit most classes in current available time*
- e.g. current schedule is as follows
  - *including overlapping classes*



Classroom schedule

Algorithms and Data Structures

classroom scheduling problem - algorithm requirements - part 1

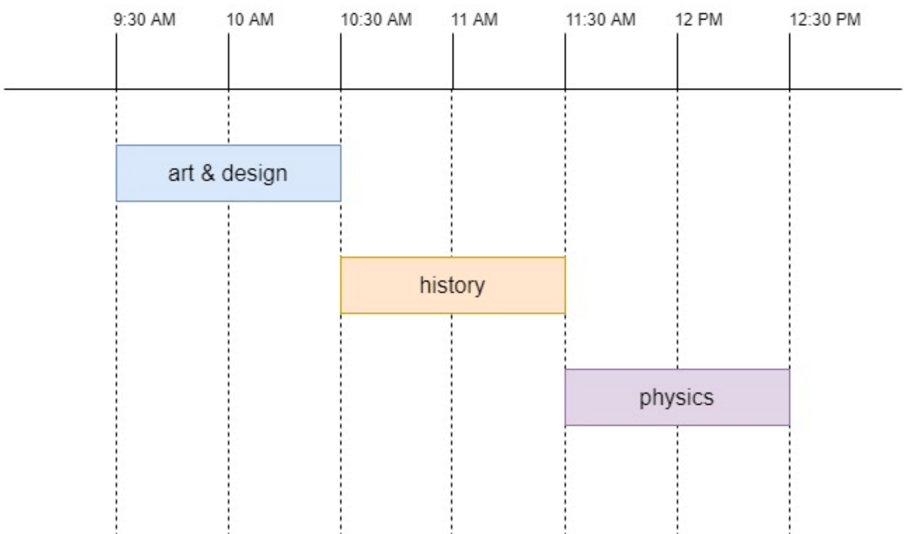
- define an algorithm to solve this problem for scheduling the classes
- whilst it may, initially, seem like a difficult problem to solve
  - *the algorithm is deceptively simple...*
- e.g. we may conceptually define this algorithm as follows
  - *select class that ends soonest...*
    - now the first class scheduled
  - *then, select a class that starts after this first class*
    - again, choose the class that ends soonest...
  - *repeat this pattern until schedule is full*
    - no more class will fit...

classroom scheduling problem - algorithm requirements - part 2

- if we apply this basic algorithmic solution
  - *update our classroom schedule as follows*
- *1. **art & design** - 9:30 AM to 10:30 AM*
  - *from our current classes, Art & Design finishes soonest*
  - *add that to our updated schedule*
- then, we need to identify a class that starts after 10:30 AM
  - *and, again, ends soonest of available classes*
- *2. **history** - 10:30 AM to 11:30 AM*
- repeat these checks
  - *update the schedule with the next class*
- *3. **physics** - 11:30 AM to 12:30 PM*

classroom scheduling problem - algorithm requirements - part 3

- now identified classes we may schedule for this classroom
  - *i.e. during the available timescale*



Classroom schedule

- 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...*

classroom scheduling problem - algorithm requirements - part 4

- if we conceptualise a *greedy* algorithm
  - *at each step we're choosing the optimal selection*
- for this worked example
  - *simply picking a class*
  - *a class that ends soonest from matching options*

classroom scheduling problem - algorithm requirements - part 5

- as a developer, at each step of the algorithm
    - *choosing optimal local solution*
- this will then produce, at the end of the algorithm
    - *a globally optimal solution*
- this simple algorithm is now able to find optimal solution
    - *i.e. to this scheduling problem*
- *greedy* algorithms may not solve all problems
    - *but they are simple to write and test...*

knapsack problem

- another similiar example is the *knapsack problem*
- commonly perceived as an example of
  - *resource allocation*
  - *combinatorial optimisation*
- knapsack problem is conceptually simple to define and understand
  - *given a group of items - each with known value and weight*
  - *need to determine number of items we may fit in a given knapsack*
  - *knapsack of fixed size and capacity*
- i.e. need to calculate combined weight of these items
  - *ensure optimised collection is less than or equal to a set limit*
- likewise, need to ensure combined value is as high as possible...
- there are known constraints and requirements
  - *allow us to calculate optimal distribution of items*
  - *and associated best use of knapsack*

knapsack problem - worked example

- common example for this problem
  - *a burglar who needs to choose best goods*
  - *goods that will fit in their knapsack*
- burglar needs to grab a collection of items
  - *items with highest value*
  - *items they can carry in their bag*
- e.g. knapsack is able to carry a weight up to 20 kilograms
  - *approximately 44 pounds*
  - *trying to maximise total value of items carried in this bag*

knapsack problem - algorithm requirements - part 1

- if we consider an algorithmic solution
  - *might initially consider a greedy approach*
  - *use to try and solve this problem...*
- e.g.
  - *begin by picking item with highest value that will fit in bag*
  - *then, pick next expensive item that will fit in the bag*
  - *then repeat...*

Algorithms and Data Structures

knapsack problem - algorithm requirements - part 2

- **n.b.** this approach will not work for this example problem
  - *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 most expensive item is the *TV*
  - *add that to the knapsack*
- it also weighs 15kg
  - *we may not add any of the other items.*

knapsack problem - algorithm requirements - part 3

- bag currently has a weight of 15kg with a value of $2500
- using this approach the highest value we may add is `$2500`
- clearly see that this is not best combination of items
- if we choose the *Computer* and *Violin*
  - *the value of the knapsack would now equal `$2700`...*

knapsack problem - algorithm requirements - part 4

- *greedy* strategy does not give an optimal solution to this problem
- if we consider the outcome
  - *it comes very close to the optimal solution*
- i.e. a quick use of this strategy will often be good enough to solve such problems
- for many problems
  - *an algorithm may solve the problem quickly and to a good enough standard*
- i.e. in this example
  - *only lost out on a potential* `$200`
  - *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 use of *greedy* algorithms
  - *commonly referred to as the set-covering problem*
- another *NP-complete* problem
  - *particularly useful as we consider approximation algorithms in general*
- outline of the problem is, again, deceptively simple to consider and understand
- e.g. a defined set of elements and a collection of sets
  - *these sets, when unified, same as initial set of elements*
  - *commonly known as the universe*
- problem requires identification of smallest union of sets
  - *union known to be equal to the universe...*

set-covering problem - worked example - part 1

- consider a problem to check for mobile internet coverage in a country
  - *coverage provided by a network of base stations in each state*
- internet coverage used to create a company
  - *company provides mobile data coverage for whole country*
- want to offer this service at lowest possible cost
  - *requires low setup and coverage costs*
- customer should be able to use service anywhere in country
- network service with full coverage across each of country’s states
- trying to minimise number of *base stations*
  - *i.e. stations needed to be able to create a working, country-wide network…*

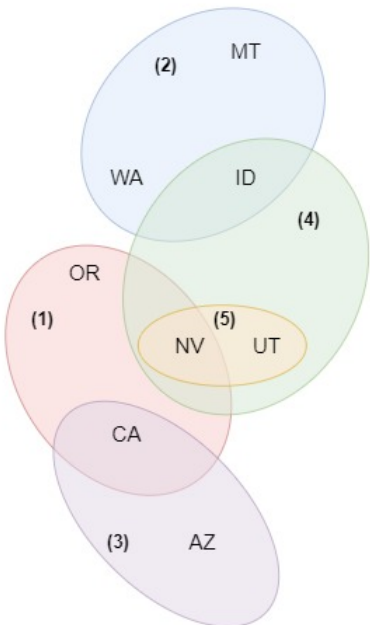
set-covering problem - worked example - part 2

- begin by compiling a sample of *base stations*
  - *those stations available to our company and network*
- e.g.

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
...	,,,

set-covering problem - worked example - part 3

- clearly see that each station covers a given region of states
  - *also some overlap between stations and states*



Set Covering - Overlapping Base Stations

- need to calculate smallest set of *base stations*
  - *i.e. smallest set to cover required country area*
- may seem a simple problem to solve
  - *in practice, a difficult and time consuming problem to resolve...*

set-covering problem - algorithm requirements - part 1

- to solve this problem use following initial outline
- outline used to determine a set of *base stations*
- e.g.
  - *define each and every available subset of base stations for given coverage area*
    - commonly known as *power set*
    - 2^n possible subsets for this problem
  - *choose set with smallest number of base stations*
    - i.e. stations that meet coverage requirements for defined area
    - e.g. base stations for country

set-covering problem - algorithm requirements - part 2

- problem is not the calculation itself
  - *long time to calculate each and every potential matching subset of stations*
- it takes `O(2^n)` time
- dealing with `2^n` base stations
- calculation will be feasible for a smaller set of base stations
- this time quickly becomes impractical
- algorithm no longer a working solution to this problem....

set-covering problem - algorithm requirements - part 3

- e.g.

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

- need to find a way to deal with such problems
- a solution that provides a working approximation
  - *and in a time useful for practical application...*

Resources

various

- How the Mathematical Conundrum Called the ‘Knapsack Problem’ Is All Around Us - Smithsonian Magazine
- Knapsack problem - Wikipedia
- Networking - Set-covering problem - MIT
- Set-covering problem - Wikipedia

videos

- NP-Complete problems - intro - up to 36:02