Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2022 - Week 13

Dr Nick Hayward

Algorithms and Data Structures

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

Algorithms and Data Structures

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
  - ...

Algorithms and Data Structures

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...

Algorithms and Data Structures

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

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

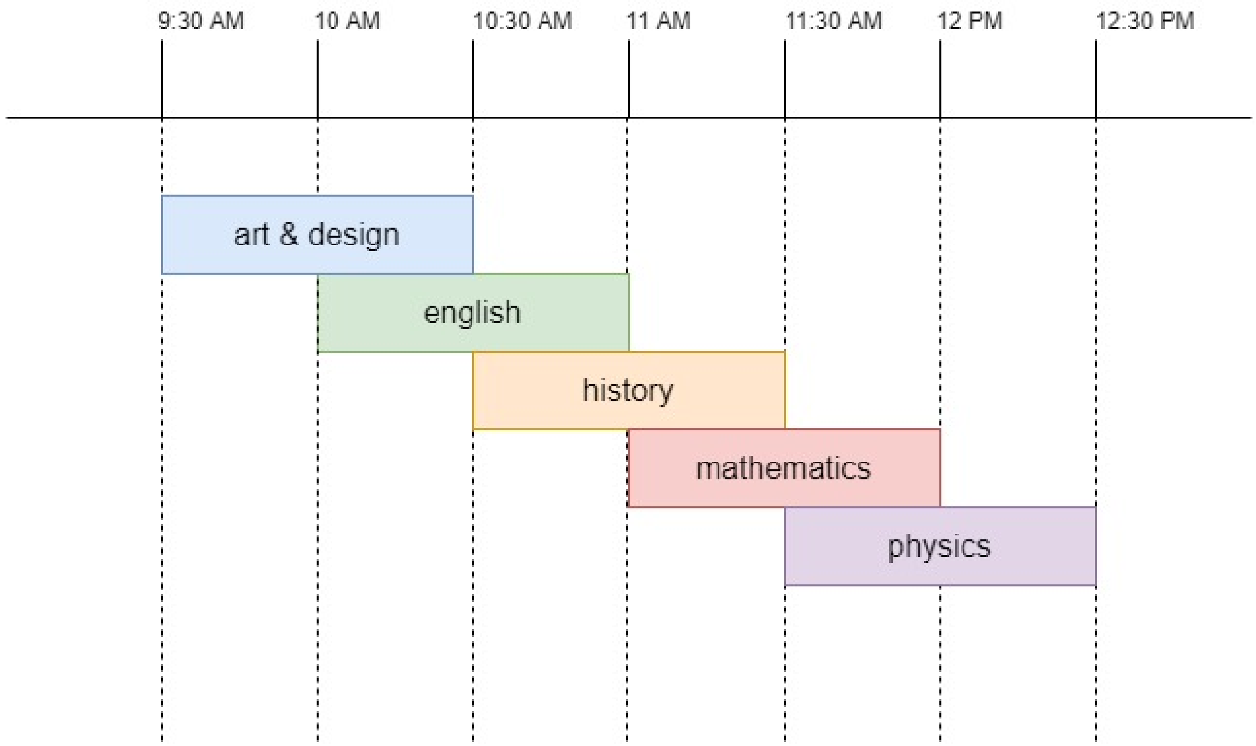

Figure 1: Classroom schedule

- 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...

Algorithms and Data Structures

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

Algorithms and Data Structures

classroom scheduling problem - algorithm requirements - part 3

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

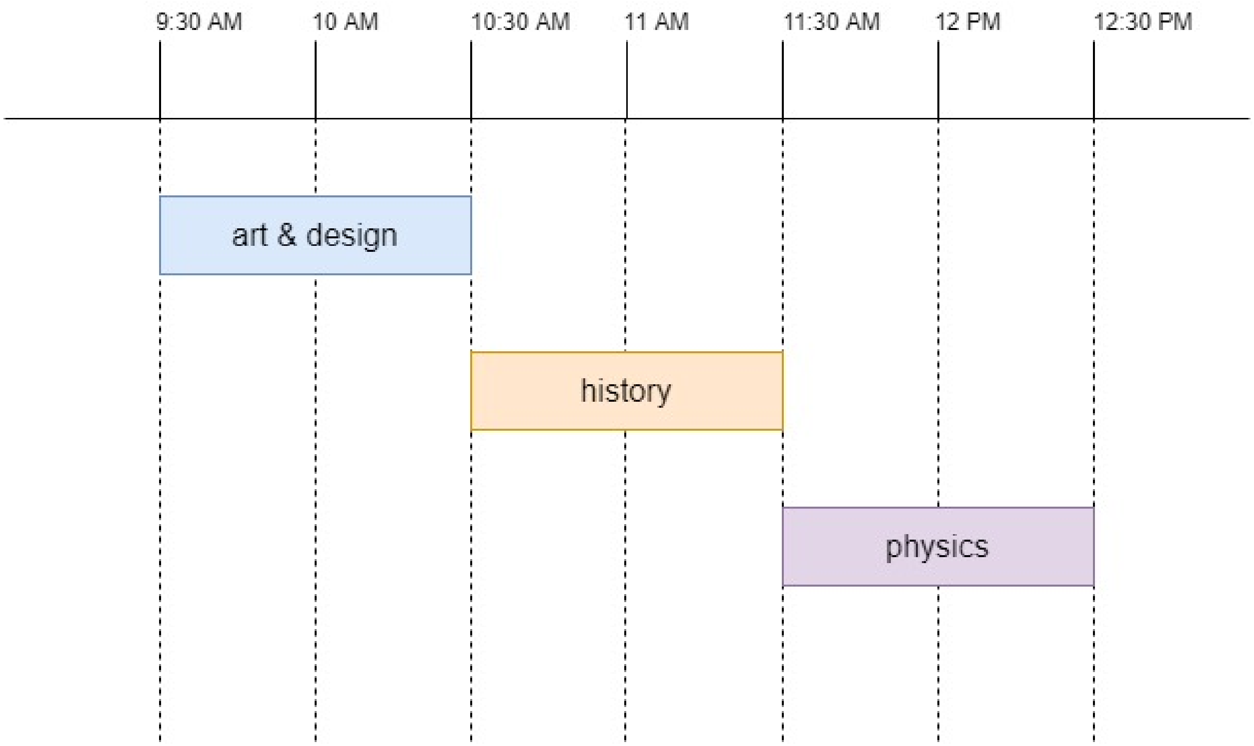

Figure 2: Classroom schedule

- 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…

Algorithms and Data Structures

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

Algorithms and Data Structures

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…

Algorithms and Data Structures

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*

Video - Algorithms and Data Structures

Knapsack problem - part 1 Algorithms - Knapsack Problem - UP TO 1:16

Source - Knapsack problem and genetic algorithms - YouTube

Algorithms and Data Structures

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

---

Algorithms and Data Structures

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.

Algorithms and Data Structures

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` …

Algorithms and Data Structures

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…

Video - Algorithms and Data Structures

Knapsack problem - part 2 Algorithms - Knapsack Problem - UP TO 5:20

An example solution…

Source - Knapsack problem and genetic algorithms - YouTube

Algorithms and Data Structures

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…

Algorithms and Data Structures

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…

Algorithms and Data Structures

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
...	„,

Algorithms and Data Structures

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

- 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...

Algorithms and Data Structures

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

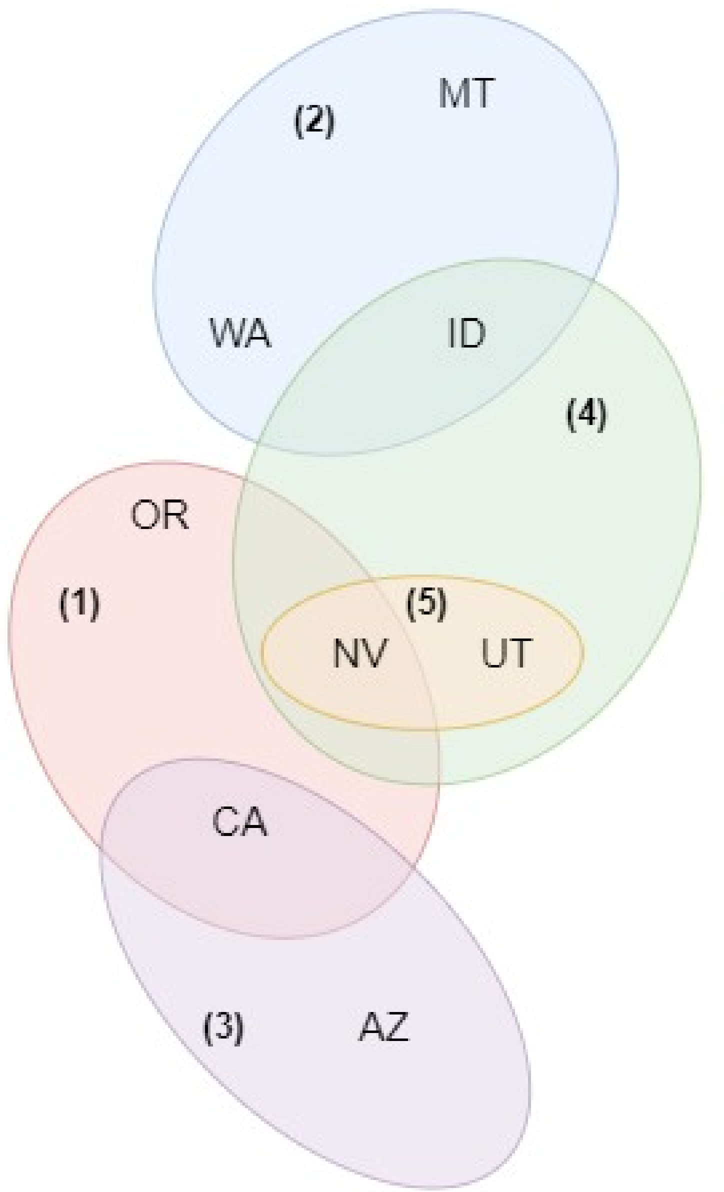

Figure 3: Set Covering - Overlapping Base Stations

Algorithms and Data Structures

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….

Algorithms and Data Structures

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 x 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…

Video - Algorithms and Data Structures

Knapsack problem - part 3 Algorithms - Knapsack Problem - UP TO END

Using genetic algorithms…

Source - Knapsack problem and genetic algorithms - YouTube

Algorithms and Data Structures

approximation algorithms - part 1

- when we deal with such *NP-complete* problems
  - commonly begin by considering *greedy* algorithms
  - act as a *good enough* solution
- *greedy algorithms* give us an approximated solution
  - often a good, usable solution…
- e.g. consider the *set-covering* problem

    - define a working *greedy* algorithm
- algorithm as follows
    - select a station that covers most states in country
        - set needs to cover states that have not already been covered
        - acceptable for set to cover some states with existing coverage
    - then, repeat this selection process until all states are covered…
- this is an example of an *approximation algorithm*

Algorithms and Data Structures

approximation algorithms - part 2

- know that a complete calculation to find exact solution takes too long
- approximation algorithm gives us a working solution
    - and in a useful amount of time
- may still compare and judge such *approximation algorithms*
- e.g. commonly check the following
    - their speed
        - i.e. how fast they are in calculating a workable solution…
    - the quality of the approximation
        - i.e. how close is the result to the expected optimal solution
- *greedy* algorithms are a useful and beneficial choice for such problems
    - simple to design and quick to execute
- e.g. for the *set-covering* problem
    - may see a performance time of `O(n^2)`
    - where `n` defines number of *base stations*

Video - Algorithms and Data Structures

approximation algorithms - heuristics & airports - part 1 Algorithms - Approximation & Heuristics - intro - UP TO 45:16

Source - [Algorithms - YouTube]

Resources

various

- Approximation algorithms - Wikipedia
- How the Mathematical Conundrum Called the 'Knapsack Problem' Is All Around Us - Smithsonian Magazine
- Knapsack problem - Wikipedia
- Networking - Set-covering problem - MIT
- NP-complete - Wikipedia
- NP-complete - NIST
- Set-covering problem - Wikipedia