Edinburgh Napier
UNIVERSITY

ALGORITHMS & DATA STRUCTURES SET08122

TOPIC 04: *ALGORITHMS & SEARCHING*

Dr Simon Wells
s.wells@napier.ac.uk
http://www.simonwells.org

TL/DR

- Smaller problems are easier to solve than bigger problems (but knowing how to break a problem up makes all the difference
- Once have data in a structure we probably want to retrieve & use it…

Edinburgh Napier
UNIVERSITY

At the end of this lecture you will be able to:

- Break down problems into smaller problems (that are [hopefully] easier to solve)
- Recognise different approaches to problem solving
- Search for data within a data structure
- Sort data (to make it easier/quicker to find data)

Edinburgh Napier
UNIVERSITY

OVERVIEW

- Algorithms & their design
- Some search paradigms:
  - Linear Search
  - Binary Search

Edinburgh Napier
UNIVERSITY

PART 0 :: ALGORITHMS & PROBLEM SOLVING

DATA STRUCTURES & ALGORITHMS

- Data structures are, at least in one sense, static
    - We put data in, and the data rests there
    - Waiting for someone to interact with it
- Without algorithms to *operate* on them, data structures don't really do much
- Whilst algorithms are finite lists of instructions that solve problems, produce an output, & terminate
- They are also the primary method through which we interact with our data structures
- We've already seen how algorithms define the construction procedures & APIs for our data structures - think of inserting data into any of our basic data structures

Edinburgh Napier UNIVERSITY

CLASSIFYING ALGORITHMS

- Algorithms aren't just an undifferentiated collection of lists of instructions however
- They can be classified on the basis of various characteristics
- These can help folk to see how various algorithms group together or are differentiated
- Classification by:
  - **purpose** | **implementation** | **design paradigm** | **probabilistic/heuristic paradigm**

PURPOSE

- Each algorithm has a goal, e.g.
  - Sort data in ascending or descending order
  - For example: The quick sort (coming up, amongst others)
  - Group algorithms together on the basis of what they do
  - We do this fairly naturally, e.g. we talk about search algorithms or sorting algorithms
- NB. Number of possible goals is very large so might want to construct larger sub-groups of purposes

What else?

IMPLEMENTATION

- Algorithms can be classified on the basis of how they are internally organised,
  - on the main implementational process that is used to get from the start state to the termination state

Can anyone thing of a way that we could sort our algorithms from labs so far on the basis of implementation?

- Perhaps consider the difference between our algorithms for constructing a linked list and those for constructing a tree…

Edinburgh Napier
UNIVERSITY

IMPLEMENTATION

Four common implementational approaches:

- **Recursive or Iterative** - Algorithm repeatedly calls itself until a certain condition is matched. Common in functional programming (e.g. Haskell, Erlang)
- **Logical or procedural** - A logical algorithm has problem expressed in terms of axioms, to which rules are applied to deduce a solution (e.g. Prolog)
- **Serial or parallel** - Serial algorithms execute one step at a time whereas a parallel algorithm allows multiple steps to occur together (which can take advantage of multiple CPUs, Cores, Threads, etc.)
- **Deterministic or non-deterministic** - Deterministic algorithms solve problem using a predefined process at each step whereas non-deterministic perform a best guess (guided by heuristics)

APPROACH TO DESIGN

- How an algorithms interacts with the problem domain gives us another basis for classification

How do you solve problems?

Think of the coursework specification. That starts off looking like a big task. How do we usually handle big tasks?

- Turns out there are a whole bunch of ways that a problem domain can be tackled

DESIGN APPROACHES

- Divide & Conquer - Repeatedly reduce problem into smaller independent instances of same problem (usually recursive) until small enough to solve easily, e.g. **binary search algorithm**
- Dynamic Programming - If optimal solution can be constructed from overlapping optimal solutions to sub-problems then can avoid recomputing solutions (related to D&C above). Memoization is a useful related technique (storing results of expensive calculations), e.g.
- Greedy Methods - Similar to dynamic programming but solutions to sub-problems don't need to be known at each stage. Instead make a *greedy* choice of what looks the best solution at the moment, e.g. Kruskal's (graph) algorithm
- Linear Programming - Express problem as a set of linear inequalities then attempt to maximise or minimise the inputs
- Reduction (transform & conquer) - Solve problem by transforming into another problem, e.g. find median in an unsorted list; transform into sorted list then take middle value; Goal: to find simplest transformation possible
- Graphs - Model problem as a graph then apply a graph exploration algorithm
- Machine Learning - New chart entry this year ;) - Increasingly used to tackle problems. Usually data intensive + lots of different sub-approaches

PROBABILISTIC & HEURISTIC APPROACHES

- Probabilistic Search - Build probabilistic model of candidate solutions for a given search space
- Genetic - Find solutions by taking inspiration from biological or evolutionary processes, e.g. define generation, randomly mutate, test against benchmark, allow percentage of best performers to survive to next generation, repeat
- Heuristic - Find an acceptable approximate solution rather than an optimal solution, e.g. if time or resource mean optimal is not practical

PROBLEM SOLVING

- We've seen that we can classify our algorithms on the basis of several factors:
  - **purpose** | **implementation** | **design paradigm** | **probabilistic/heuristic paradigm**
- But how does this help us write new algorithms to solve new problems?
- This is useful when faced with a new problem (especially if it is a difficult problem)
  - Can use our understanding of the variety of algorithmic approaches to inform a strategy for tackling new problems
    - For example: it's almost never wrong to initially try to divide & conquer a problem
  - Can look at what has already been tried then apply a different tactic to see if the results improve

Edinburgh Napier
UNIVERSITY

PART 1 :: SEARCHING

SEARCHING

- Any algorithm to retrieve information stored in a data structure
  - e.g. Finding data in collections like arrays, lists, &c.
- Searching for & retrieving data becomes an acute problem at scale
  - e.g. Search engines return results in a split second (user expectation) but have to get those results from perhaps billions of pages (not quite this straightforward but a useful scenario)

In what other circumstances might searching a lot of data be needed?

SEARCH KEYS

- An *algorithm* is used to find items that have specific properties within a collection of items
- Items might be
  - basic data-types or objects
  - database records or structs
  - elements of a search space
- The datum that defines the search target is known as the **search key** or, less frequently, **search term**

Edinburgh Napier UNIVERSITY

SEARCH STRATEGIES

- Search algorithms implement **search strategies**
- There are many strategies , some are simple, others more complex, all with different characteristics.
- Basic differentiation based upon the data being searched:
  - Is the data organised in any way - if it is then this influences how we can approach finding a particular element within the collection
  - What else do we know about the data? - sometimes the nature of the data and it's sorting can give us a little boost in our search strategies
    - For example, is the data effectively infinite - think of when you can calculate a given value in a sequence but cannot calculate the entire series

SORTED VS UNSORTED

- Strategies that work on unsorted data are simple and robust but can have very poor performance characteristics
- Most efficient search strategies rely on **sorted** data
  - This is why we often talk about searching & sorting s though they are inextricably linked

Edinburgh Napier UNIVERSITY

LINEAR SEARCH

- The **simplest** search algorithm
- List need not be ordered
- Essentially a brute force or exhaustive search
    - Check each element of list in sequence until either you find the desired element or the list is exhausted
- Worst case? Have to check every element in list
- But, VERY simple to implement, practical when list is quite short or when performing a single search on an unordered list
- Trade-off between set-up/sort time & search time for other algorithms versus cost of searching with linear search

BINARY SEARCH

- Find the position of a target value within a sorted array
- Compare target value to value of middle element of array
- if target = element then search is finished
- if target < element then repeat in lower half of array
- if target > element then repeat in upper half of array
- Target will either be found or not found
- Each iteration will half the search space

SEARCH ALGORITHM SELECTION

- Linear search or binary search are usually the two basic choices for searching
- If small amount of unsorted data or time taken to sort data will exceed time taken to search through it then a **Linear search** may be appropriate
  - Otherwise **Binary Search**
- However various data structures can be constructed in sorted order, make it simple to partition the search space, or to jump to the approximate location of your search key so that the number of comparisons can be minimised
- There are also some modifications of binary search that have useful properties for the right niche search niche.

Edinburgh Napier UNIVERSITY

JUMP SEARCH

- Algorithm for sorted arrays
- Check fewer elements by skipping a fixed number of elements instead of searching all elements
- Because array is sorted our comparison tells us if we've jumped too far
- Can then perform linear search of the interval
- Time complexity is O(√n) which is between linear search, O(n), and Binary Search, O(Log n)
- Why? Advantage of jump search is that it only needs to jump back once so if the jumping back operation is costly then jump search can become more efficient than binary search (limited circumstances)

INTERPOLATION SEARCH

- Sorted Arrays (notice a pattern yet)
- If values are uniformly distributed then this *can* be an improvement over binary search (again narrow circumstances)
- Binary search always goes to middle element - what if we relax this assumption?
- Uses a simple formula to calculate a *probe position* on each iteration instead of just using the middle element. Formula relies on the distribution assumption above.

INTERPOLATION SEARCH

Calculating Probe Position:

pos = lo + [(x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo])]

arr[]	==> Array where elements need to be searched
x	==> Element to be searched
lo	==> Starting index in arr[]
hi	==> Ending index in arr[]

Search Algorithm:

Step1: In a loop, calculate the value of "pos" using the probe position formula.
Step2: If it is a match, return the index of the item, and exit.
Step3: If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.
Step4: Repeat until a match is found or the sub-array reduces to zero.

EXPONENTIAL SEARCH

- Misleading name (actually runs in O(Log n) time)
- Useful when you need to search unbounded data, e.g. size of array is infinite
- Can work better than Binary search in a bounded array when the target is closer to the first element

TERNARY SEARCH

- **Ternary:** *composed of three parts*
- An uncommon form of divide & conquer
- Only slightly less efficient than binary search
    - $2\text{Log}_3 n$ versus $\text{Log}_2 n$
    - Ternary search does slightly more comparisons than binary search in worst case
- Use case: If trying to find minimum or maximum of a unimodal function
- Split search domain into 3 then check whether the min|max is in the first third, last third, or central third

SUMMING UP SEARCHING

- We only really have two ways of searching:
  - Entire collection if unsorted (linear search)
  - Divide & conquer (binary search) if collection is sorted
    - Can get additional benefits if we have insight into the specific nature of our data & it's structure/ distribution
    - In which case some variations on divide & conquer are available in very limited circumstances
- However, trees & graphs can give us additional ways to access our data - for example
  - Minimal spanning tree (*later*) is essentially a search for a set of nodes such that the sum of the edges connecting them is minimised.
  - Route finding can algorithms search for a set of nodes and edges that link a start position tom an end positing

There are more ways to think of searching than just getting an item from a collection

SUMMARY

- Algorithms - quick overview of how algorithms can be classified
- Searching
  - Linear Search & Binary Search (+ some variations)

Edinburgh Napier
UNIVERSITY

QUESTIONS ???

Edinburgh Napier UNIVERSITY

WHAT DID WE LEARN?

- *We can now…*

- Break down problems into smaller problems (that are [hopefully] easier to solve)
- Recognise different approaches to problem solving
- Search for data within a data structure

COMING UP…

- The biggest efficiency when searching is found when dealing with sorted data
- If we can sort our data then we can retrieve items more swiftly
    - *assuming that the sort procedure takes less time than a linear search*
- So we should probably look at how to sort our collections next…