

G52ACE 2017-18

Brute force, heuristics and Dynamic Programming

General Methods

There are various general methods (“paradigms”) for finding solutions to problems.

Common ones include

- Brute force – “generate and test”
- Divide-and-conquer
- Heuristics
- Dynamic Programming

“Brute Force”

- This is roughly “generate and test”
 - Generate all potential solutions
 - Test for which ones are actual solutions
- Example: we could do sorting by
 - Generate all possible permutations
 - Test to see which one is correctly ordered
 - Extremely inefficient, as is $O(n!)$
- Can be useful in some (small) cases
 - E.g. Due to the simplicity

Divide and Conquer

- Recursively, break the problem into smaller pieces, solve them, and put them back together
 - Merge-sort and Quicksort were classic examples of this

Heuristics

- “Heuristic” = “rule of thumb”
 - Generally, meant to mean something that gives better decisions, than the naïve methods, but still not necessarily optimal
- Two common types (the term is over-loaded)
 - Decisions within a procedure that gives optimal answers, but are designed to make it go faster
 - Decisions within a procedure that might not give optimal answers, but are designed to give good answers that are impractical to obtain otherwise

“Heuristics in exact methods”

- These are general methods that works in an algorithm that does give exact or optimal answers
 - But need the heuristics to decrease the (average/typical) runtime
 - Examples:
 - “Admissible heuristic” in A* search (from first year AI) – decreases the search time compared to plain search
 - “pick a random pivot” in quicksort

“Heuristics in inexact methods”

- These are general methods that (generally) are not be guaranteed to give the best possible answers, but that can give good answers quickly
 - Used on problems when the exact methods are too slow, e.g.
 - Timetabling and scheduling
 - Many design problems
 - It is a vast research area, e.g.
 - genetic algorithms
 - metaheuristics (simulated annealing, tabu search, etc, etc)
 - approximate greedy methods.

Greedy algorithms

- A common “heuristic” is to be “greedy”
- Take the decision that looks best in the short term
- Prim’s algorithm for constructing a Minimal Spanning Tree is a ***greedy algorithm***: it just adds the shortest edge without worrying about the overall structure, without looking ahead. It makes a locally optimal choice at each step.

Greedy algorithm: optimal

- Sometimes greedy algorithms can still give optimal answers.
- E.g. Prim's algorithm for constructing a Minimal Spanning Tree is a ***greedy algorithm***:
 - It just adds the shortest edge without worrying about the overall structure, without looking ahead.
 - It makes a locally optimal choice at each step.
 - But it turns out that this is sufficient for the final answer to be optimal

Greedy algorithm: non-optimal

- Sometimes greedy algorithms cannot guarantee to give optimal answers, but often still give optimal answers.
- Example: “Change-giving”:
 - Problem: given a collection of coins (a multi-set, that allows repeated elements), and a desired target for the change. Supply the change in as few coins as possible.

“Min-Coins-Change-Giving”

- INSTANCE:
 - Given a set S of coins and their values $x[]$
 $S = \{ x[1], x[2], \dots, x[n] \}$
(coins can be repeated, so S is actually a multi-set)
 - A target K (the total value to be returned)
- TASK:
 - Find the set with the minimum number of coins but whose total value is **exactly** K .
 - That is, supply the change in as few coins as possible.
 - (Or show it is not possible with the given coins.)

“Min-Coins-Change-Giving”

Examples

- $S = \{ 50, 20, 10 \}$ $K = 15$
 - There is no solution – have to go and get more change 😊
- $S = \{ 50, 20, 10 \}$ $K = 10$
 - There is solution with 1 coin $\{10\}$
- $S = \{ 50, 20, 10 \}$ $K = 0$
 - There is solution with 0 coins $\{\}$
 - Might seem bizarre to consider this – but it is an important special case!
- $S = \{ 50, 20, 10, 10, 5, 2 \}$ $K = 15$
 - There is a solution with 2 coins $\{10, 5\}$ but no solution with 1 coin

“Min-Coins-Change-Giving”

- **INSTANCE:** Given a set S of coins and their values $x[]$
 $S = \{ x[1], x[2], \dots, x[n] \}$, and a target K
- **TASK:** Find the set with the minimum number of coins and whose total value is **exactly** K .
- “Obviously” this can be solved by enumerating all possible subsets of S , selecting those that sum to K , and picking a subset with the fewest elements
 - But with n coins there are 2^n subsets and so this “generate-and-test” naïve algorithm is exponential in the worst case.
 - Can we do better? Firstly, consider greedy strategies:

Change-Giving Greedy algorithm?

- Greedy strategy: start with the largest coin which is available; for the remaining change, again pick the largest coin; and so on.
- Often this will work fine, e.g.
- Coins = {50,50,20,20,10,5,2,2,1,1}. “Change”: 73
 - Greedily pick 50, leaving change = 23
 - Greedily pick 20, leaving change = 3
 - Greedily pick 2, leaving change = 1
 - Greedily pick 1, leaving change = 0
- Answer: $50+20+2+1 = 73$

Change-Giving Greedy algorithm?

- Sometimes it fails, e.g.
- Coins = {5,2,2,2,2}. Change= 8
 - the greedy choice of the largest coin '5' is a 'fatal mistake' – as it not part of any solution
- Coins = {50, 50, 20, 20, 20, 2, 2, 2, 2, 2} Change= 60
 - Greedy method picks { 50, 2, 2, 2, 2, 2 } - 6 coins
 - But can do it with { 20, 20, 20 } – 3 coins

[Aside] Change-giving problem

- “Exercise”: find a fast algorithm for the general version of this problem, or show one does not exist.
- Note:
 - See G52LAC/G53COM lectures on “**P**” and “**NP**”.
 - Not meant as a real exercise for G52ACE !!! 😊
 - see “millennium prize problems” there is a \$1m prize for solving this “exercise” 😊
 - As no one has solved (a general version of it) in many decades

Dynamic Programming (DP)

- DP is a general method that can be suitable when the optimal solutions satisfy a “decomposition property”
- The general idea is roughly:
 - Splitting an optimal solution into sub-solutions corresponds to splitting the problem into sub-problems and the sub-solutions are optimal for the sub-problems
 - So optimal solutions can be built out of optimal solutions of (smaller) sub-problems
 - So solve small sub-problems first and build up towards the full solution.

“DP” for Subset Sum

- Firstly, consider just giving exact change and not worrying about the number of coins. The problem is better known as
- “Subset-Sum”:
 - Given (multi-)set S of positive integers $x[i]$ and a target K
Is there a subset of S that sums to exactly K ?
- (Note: looks innocuous, but can be hard.)

DP for Subset Sum

- Algorithm: Consider the numbers one at a time keeping track of which subset sums are possible.
- Main data structure:
 - Boolean Array, Y , for $[0, \dots, K]$
 - $Y[m] = \text{true}$ iff some subset has been found that sums to m

DP for Subset Sum

- The simple underlying idea is to suppose we have found all the subset-sums for $x[0], \dots, x[i-1]$ and then want to also add the effect of $x[i]$
 - **if some subset summed to m , then with the inclusion $x[i]$ we can also find a subset that sums to $m+x[i]$**
 - Note this works from small sets, with small sums, up to larger sets with larger sums

DP for Subset Sum

Input: $x[0], \dots, x[n-1]$ and K

Initialise all $Y[m] = \text{false}$

$Y[0] = \text{true};$ // As can always provide no change

```
for (i=0 ; i<n ; i++) { // consider effect of x[i]
    for (m=K-x[i] ; m>=0 ; m--) {
        if (Y[m]==true) {
            // m was achievable with x[0]...x[i-1]
            // hence now also m+x[i] is achievable
            if (m+x[i] == K ) return success
            if (m+x[i] < K ) Y[ m+x[i] ] = true
        }
    }
}
```

Complexity

- Outer loop has to consider all the coins
 - hence $O(n)$
- Inner loop scans the entire array Y ,
 - hence $O(K)$
- Overall is $O(n K)$
- Much better than $O(2^n)$

Complexity [aside]

(Following is not assessed)

- Overall is $O(nK)$
 - However, “K” has the “hidden exponential” if it is represented in binary:
 - The relevant input size is the number of bits B that are needed to represent, $B=O(\log(K))$
- The complexity in terms of the size of the binary input is $O(n 2^B)$
 - Is called “pseudo-polynomial”.

Min-Coins version

- Previous just asked if it is possible to do the change
- But want to minimise the coins
- Hence, need to not just track
 - “is a given target possible?”
- But also the minimum number of coins that are needed

DP for Min-Change-Giving

- Algorithm: As before, inspect the coins one at a time keeping track of the best answers obtained with the coins inspected so far
- Main data structure:
 - Integer Array, Y , for $[0, \dots, K]$
 - $Y[m] = -1$ if have not found any sum for m as yet
 - $Y[m] = c \geq 0$ means that have found that can achieve the sum m with c coins.
- Aim: when the algorithm finishes then $Y[K]$ will be the minimum number of coins
 - “Side-effect”: All the values of $Y[m]$ $m < K$, will also be the minimum number for a value of m .

DP for Min-Change-Giving

- The simple underlying idea is to suppose we have found all the best answers for the coins $x[0], \dots, x[i-1]$ and then want to also add the effect of one more coin $x[i]$
 - **if some set summed to m , then with the inclusion $x[i]$ we can also find a subset that sums to $m+x[i]$**
 - **and with one more coin than was recorded as possible for m**
 - If a set of coins had already been found then take the one that gives the minimum.

DP for Subset Sum (schematic)

Input: $x[0], \dots, x[n-1]$ and K

Initialise: $Y[0] = 0$, // as can give a change of 0, with 0 coins
and $Y[m] = -1$ for $m > 0$

```
for (i=0 ; i<n ; i++) { // consider effect of x[i]
    for (m=K-x[i] ; m>=0 ; m--) { // scan array
        if (Y[m] >= 0 ) {
            // value m was achievable with x[0]...x[i-1] using Y[m] coins,
            // so, m+x[i] is now achievable with Y[m]+1 coins
            // but might already have found a better answer
            // stored as Y[m + x[i] ] so then take the best
            if (Y[m + x[i] ] == -1 )
                Y[ m + x[i] ] = Y[m]+1
            else
                Y[ m + x[i] ] = min( Y[m + x[i] ] , Y[m]+1 )
        }
    }
}
```

Why does this work?

- Point of DP is that this really does split
 - Doing with optimal of $N+M$ means the $\text{val}(N)$ and $\text{val}(M)$ are separately done optimally
- Such properties of the solutions are usually expressed by “Bellman Equations”
 - https://en.wikipedia.org/wiki/Bellman_equation
 - DP is a way to exploit these equations and structures.

Comments

- The structure of the algorithm for the change giving has many applications
 - We will see a similar structure in the later lectures for finding shortest paths in graphs
 - Many “advanced but highly effective” algorithms can use dynamic programming methods

Expectations

- Have a broad understanding of the different classes of algorithms and be able to give examples of their applications
- Understand the change-giving problem and the associated DP-style algorithm

Worked example

Input: $x[] = \{5, 2, 2, 2, 1\}$ and $K=6$

Initialise: $Y[0] = 0$ and $Y[m] = -1$ for $m > 0$

for ($i=0$; $i < n$; $i++$) { // consider effect of $x[i]$

 // state of Y here for each k is given below

 for ($m=K-1$; $m \geq 0$; $m--$) { // scan array

 if ($Y[m] \geq 0$) {

$Y[m + x[i]] = \min(Y[m + x[i]] , Y[m] + 1)$

 }

 }

}

$k=0$ $Y[] = [0, -1, -1, -1, -1, -1, -1]$ $Y[0]=0$ for change {}

$k=1$ $Y[] = [0, -1, -1, -1, -1, 1, -1]$ $Y[5]=1$ for change {5}

$k=2$ $Y[] = [0, -1, 1, -1, -1, 1, -1]$ $Y[2]=1$ for change {2}

$k=3$ $Y[] = [0, -1, 1, -1, 2, 1, -1]$ $Y[4]=2$ for change {2,2}

$k=4$ $Y[] = [0, -1, 1, -1, 2, 1, 3]$ $Y[6]=3$ for change {2,2,2}

$k=5$ $Y[] = [0, -1, 1, -1, 2, 1, 3]$ $Y[6]=\min(3, 1+1)=2$ for change {5,1}

Finished: so optimal answer is 2 coins.