Algorithm Design

Me me big slow

Daiwei Chen

<2019-02-11 Mon>

Contents

1 Divide & Conquer

- Break the problem down and solve smaller problems
- Combine small solutions to form the bigger solution
- Ask yourself: Is solving problems and combining better?
  - Recursion is not free.
- Do keep in mind: Getting a solution is better than no solution. As long as it's "fast enough".

2 Greedy Solutions

- Dr. Eloe: "This is the YOLO approach to algorithms, don't look back."
- Short-term gain over long term benefit.
- Build the solution one step at a time.
- At each step, choose the most beneficial choice.

  Locally optimal For that specific step.

2.1 Example - Real life

You've got to head to GS after Colden Hall, but it's cold as **f u c k**. So let's building hop in order to get some warmth. With each step, let's choose the closest building.

1. Union
2. Admin Building
3. GS

2.2 Example - Algorithm

Coin changing Given a list of denominations, what is the smallest # of coins I can dispense for a given amount of change?

denoms = [50, 25, 10, 5, 1]
A = 70
Start with 50, then 2 dimes.

```
# Denominations is always sorted, and you always have a penny
def giveChange (A:int):
    d = 0 # index of our current coin.
    while A > 0:
        c = A // denoms[d] # integer division in python 3
        print(c, "@", denoms[d])
        A -= c*denoms[d]
        d += 1
```

- O(n) on average, since you may have to check every single denomination

A = 18 gives 10, 5, 3x1s

- However if you have
  denoms = [10, 6, 1] and
  A = 12, you get back 3 coins, the optimal answer is 2

Greedy Heuristic It's an algorithm applied to optimization where you may not get the most optimal answer, but you'll still get an *okay* answer.

- The question is, is it worth it to spend all this time to get the most optimal solution or something just "good enough". And in certain cases, the optimal solution is very impractical.

3 "Hard" Problem

3.1 0-1 Napsack Problem

You wanna be a very efficient thief. You arrive at a room where there are a lot of items. Each item has a Weight and Value. You have a bag that has infinit volume, but has a max weight capacity. You cannot take part of an item.

Brute force To calculate all of the possibilities is about 2^n possibilities.

Let's see this:

n	Approximate time
10	1 MS = .0000001 seconds
20	1 ms = .001 s
30	1 second
40	18.3 min
50	13 days
100	$4 * 10^{13}$ years

3.1.1 George Dantzig

Sort by decreasing $\frac{V_i}{W_i}$ $O(n \log n)$

Grab items that fit into the bag $O(n)$

$O(n \log n) + O(n) = O(n \log n)$

$O(n \log n) * O(n) = O(n^2 \log n)$

3.2 Filling a room

You are filling up a room with activities, each activity has a start and end time. How can you fit as many activities into a room as possible?

1. Sort your activities by finish time.
2. Pick the first and earliest finish time, then pick the next event with the earliest finish time without overlapping.
3. Works well! Greedy algorithm can maximize the event number, but it won't work if you wanted the maximum usage of time then this won't work.

4 Dynamic Programming

- Break the problem into subproblems.
- Solve all possible subproblems.
  - Store solutions somewhere for later.
- Use solutions to solve bigger problems.
- Dynamic Programming is useful for *most* optimization problems that involves on reducing the size of the problem into smaller problems.

4.1 Example: Figs

Here's a recursive solution

```
def fib(n):
    return (n in (1, 0)) ? 1 : fib(n-1) + fib(n-1)
```

Here's the **DYNAMIC** approach

```
def fib(n):
    fs = [1, 1]
    for i in 2..n:
        fs.append(fs[i-1]+fs[i-2])
    return fs[n]
```

Basically, you remember your previous solutions. These sequences are generated linearly. Therefor you have a better runtime and Big O. You can apply this to coin changing, room allocation, and also the 0-1 napsack problem.

4.2 Back to Coin Changing

$d = [1, 6, 10] $A = 12$ Here's the table for dynamic programming:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	2	3	4	5	1	2	3	4	5	6	2
2	0	1	2	3	4	5	1	2	3	4	1	2	2

C[i][j] = min # of coins w/ denom of d[0..i] to give back j cents of change.
C[2][12] contains the answer.
The x symbolizes each smaller problem, each smaller coin change.
Each y (the row) represents how many coins we're allowed to use in terms of our denominations.
On each row, you calculate the optimal solution using solutions you've used before. For example, on (6, 1), you decide it'll be one 6 cent coin and zero 1 cent coins.

Let's look at some **code**.

```
c = [n][A+1]
c[i][0] = 0 # Fills the first collumn with 0s
c[0][j] = j # Fills the first row with its own row index

'''
for all cells within c:

c[i][j] =

c[i-1][j] if j < d[j] # While you cannot fit in another coin, just use the old answer.
min(c[i][j-d[i]] + 1, c[i-1][j]) else # Otherwise check for the min between both answers.
'''
```

Since you're always looking for the minimum amount. On each cell fill, you look at the potential amount for the current row, but also comparing it to the previous row's answer.

4.3 Rod Cutting Problem

L_i	1	2	3	4	5	6	7	8	9	10
P_i	1	5	8	9	10	17	17	20	24	30

You're in the retail business now and you want to cut a rod a certain way. You have a rod of length n and each length of L has a price of P. How would you cut the rod in order to **MAXIMIZE PROFIT** *wow that's a lotta m o n e y* You have 2^{n-1} possibilities to cut the material. This is actually exponential increase. Brute forcing will end your career in CS instantly.

Let's make a new table:
i is the length of the rod.
r_i is the maximum revenue for a rod of length i.
s_i is the size of the first cut for a rod of length i.
On each iteration, you check if it's more profitable to use a previous calculated optimal solution + however much more. Or you can also check if which combinations of the previous optimal solutions vs newer solutions you create is the better choice.

i	r_i	s_i
0	0	0
1	1	1
2	5	2
4	10	2
5	13	2/3
6	17	6
7	18	1/6/2/3
8	22	2/6
9	25	3/6
10	30	10

At the end of the table creation. You cut the first time, then use the table to determine where to cut next by "cutting the first time again". In the example of $n = 4$, you start at $i = 4$, and use S_4 to cut by 2 and make 5 dollars, then you're at $i = 2$ and follow S_2 to cut off another 2 and make 5 dollars more.
So what's the complexity? On each i, we have to consider i cases. Thus it is

$$\sum_{l=1}^{n}\sum_{i=1}^{l} 1 = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} = \Theta(n^2)$$

This big Θ isn't technically polynomial. It is *Pseudo-Polynomial.*
Let's look at some source code:

```
# This will generate the r and s for the solution "lookup" table
def rod_cut(p: list, n: int):
    r = []*(width/n+1)
    s = []*(width/n+1)
    r[0] = 0
    for length in range(1, n+1):
        best = -INF
        for i in range(1, length+1):
            if best < p[i] + r[length-i]:
                best = p[i] + r[length-i]
                s[length] = i
        r[length] = best
    return r, s

# This will actually provide the solution
def printSoln(r: list, s: list, n: int):
    print("Total sale price:", r[n])
    while n > 0:
        print("Cut: ", s[n])
        n -= s[n]
```

To generate the list, on each operation, you calculate if using a previous example is better or making a new way of cutting it is better. And you save that best for the length i.

4.4 Longest Common Subsequence

A subsequence is a sequence of elements that appear in the order of the initual sequence. Example:
NBUD is a subsequence of
S **N** O W **B** O **U** N **D**
The Longest Common Subsequence between two sequences.

A, B : Sequences
-> 1 Indexed
c[i][j]: length of the LCS between A[1:i] to B[1:j]
-> Θ Indexed

Let's look at some code:

```
# All of this is 1 indexed
if i==0 or j==0:
    c[i][j]=0

if a[i] != b[i]:
    c[i][j] = max(c[i-1][j], c[i][j-1])
else:
    c[i][j] = 1+c[i-1][j-1]
```

4.4.1 Example

A = ['A', 'B', 'C', 'D', 'E', 'F']
B = ['D', 'E', 'F', 'A', 'B', 'C']

c	*	A	B	C	D	E	F
*	0	0	0	0	0	0	0
D	0	0	0	0	1	1	1
E	0	0	0	0	1	2	2
F	0	0	0	0	1	2	3
A	0	1	1	1	1	2	3
B	0	1	2	2	2	2	3
C	0	1	2	3	3	3	3

```
A = 'NORTHWEST'.split('')
B = 'BEARCATS'.split('')
```

c	*	N	O	R	T	H	W	E	S	T
*	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	1	1	1
A	0	0	0	0	0	0	0	1	1	1
R	0	0	0	1	1	1	1	1	1	1
C	0	0	0	1	1	1	1	1	1	1
A	0	0	0	1	1	1	1	1	1	1
T	0	0	0	1	2	2	2	2	2	2
S	0	0	0	1	2	2	2	2	3	3