Algorithms & Complexity Revision

jxz12@ic.ac.uk

Complexity

- Lots of algebra
- **Induction** and **Recurrence** relations to prove cost & correctness
- These are hard! You often have to be able to guess to answer, which means you need practice to see the common patterns.

2012

c) For each of the algorithms below derive the corresponding complexity.

i) The problem is solved by dividing the initial problem into two subproblems of half the size, recursively solving each subproblem, and then combining the solutions in cubic time, i.e. $O(n^3)$. [3]

ii) The problem is solved by dividing the initial problem into two subproblems each of quater of the size, recursively solving each subproblem, and then combining the solutions in time $O(\sqrt{n})$. [3]

iii) The problem is solved by recursively solving one subproblem of size $n-1$, where n is the size of the initial problem, and then performing additional operations requiring linear time, i.e. $O(n)$. [3]

The answer is n(n+1)/2, but why?

1. by diagram
2. by clever algebra
3. by induction (like climbing a ladder)

Divide & conquer and the **Master theorem**

- Master theorem is all about divide and conquer
- ***RECURSION*** - you split up the problem, but at a cost (recombining).
- If you're keen, try coming up with the recursive algorithm to solve the towers of Hanoi, which has a beautiful solution

As a note, the reason recursion is hard to wrap your head around is because it's top down, and not bottom up. This means you *don't explicitly know* how many iterations are going to run beforehand, and it's often difficult to work out (or even not a determinate number).

2015

1. Give a tight bound for each of the following recurrence relations.

Carefully justify your answers.

a) $T(n) = 2T(n/3) + 1,$

[4]

b) $T(n) = 5T(n/4) + n,$

[4]

c) $T(n) = 7T(n/7) + n,$

[4]

d) $T(n) = 49T(n/25) + n^{3/2}\log(n).$

Hint: This recurrence does not directly fit in the statement of the Master theorem. To derive the asymptotic behaviour of $T(n)$ use the tree decomposition used in the proof of the Master theorem.

[8]

$$T(n) = aT(n/b) + O(n^d),$$

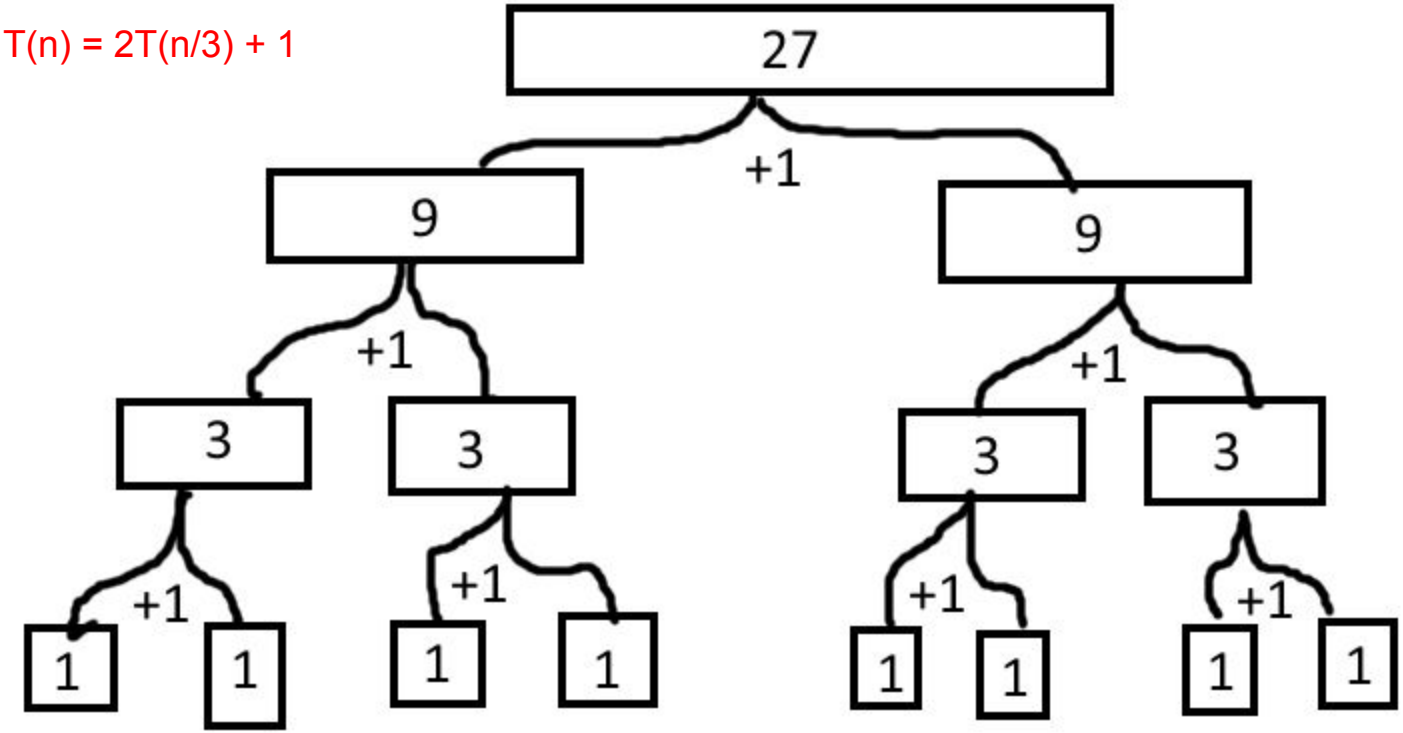
where $a > 0, b > 1$ and $d \geq 0$. Then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b(a) \\ O(n^d \log(n)), & \text{if } d = \log_b(a) \\ O(n^{\log_b(a)}); & \text{if } d < \log_b(a). \end{cases}$$

T(n) = aT(n/b) + c

a determines splitting per level (width)
b determines how many levels (height)
c determines the cost of splitting

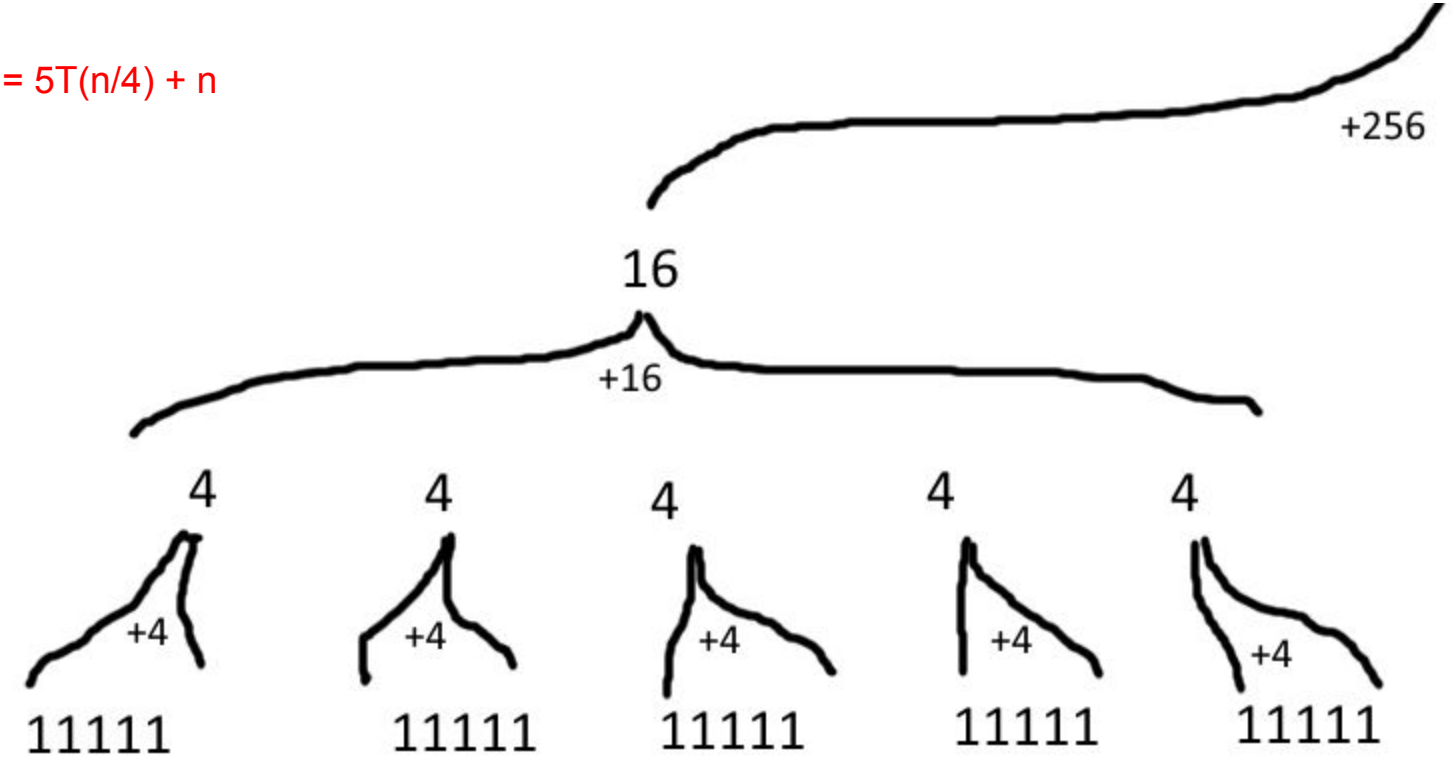
a) T(n) = 2T(n/3) + 1



Total work is 7 + 8 = 15. The amount of 1s at the bottom is actually much smaller now than the amount at the top!

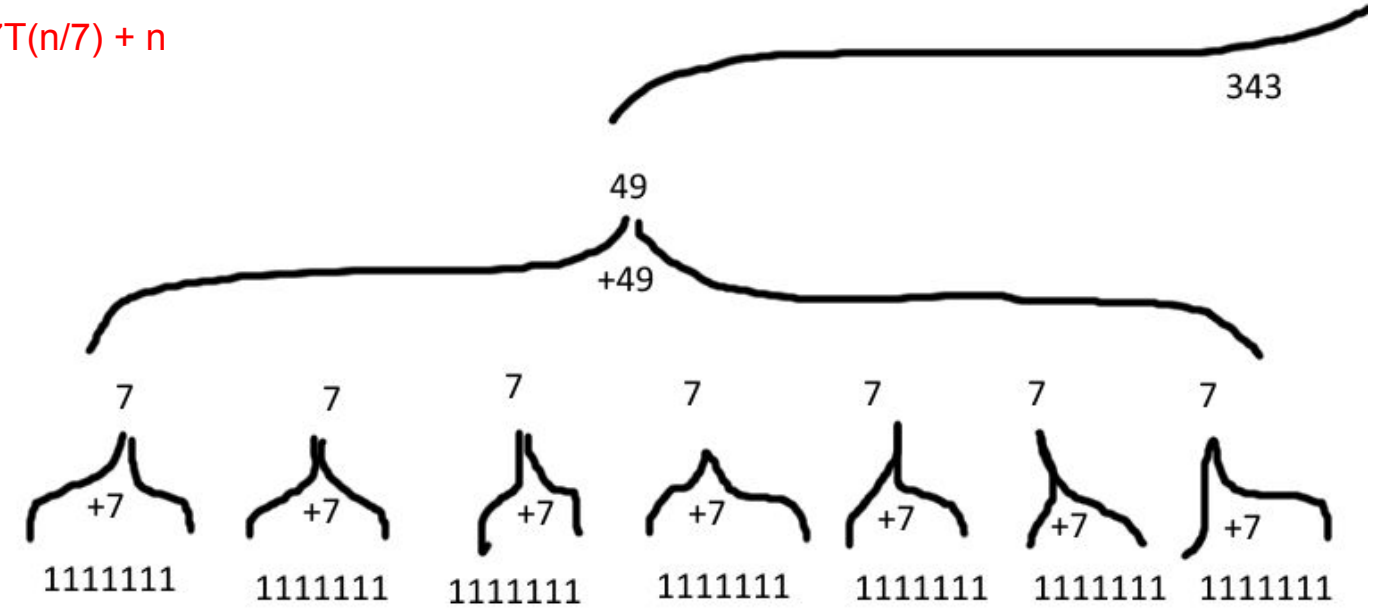
(imagine c = O(n))

b) T(n) = 5T(n/4) + n



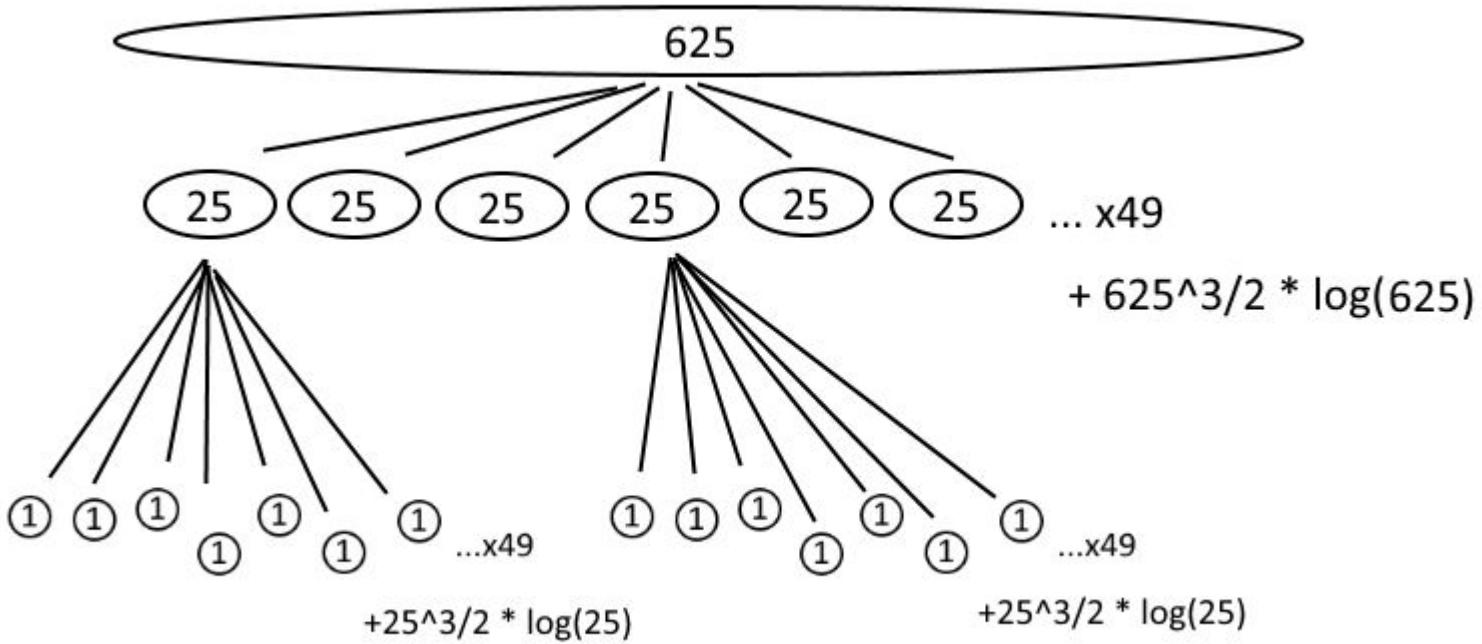
In this case the amount of 1s at the bottom is actually bigger than the number at the top. Since recombining is O(n), that means the splitting dominates

c) T(n) = 7T(n/7) + n



The number of 1s at the bottom is the same as at the top. The amount of work splitting is also the same at each level. Intuitively they grow equally, which is why case 2 applies.

d)



We can still use the logic and intuition of the master theorem to predict that the combination terms will dominate. However, the question asks us to derive the whole thing ourselves, and that's why we need to use a summation and solve it.

Using the tree to derive the summation...

k=0	625^(3/2) * log(625)	625 = n
k=1	25^(3/2) * log(25)	25 = n/25
k=2	1^(3/2) * log(1)	1 = n/25^2
		(so use n/25^k)
(where k is the level of the tree)		=> (n/25^k)^(3/2)*log(n/25^k)

As seen in the official answers. However they leave out the splitting into 49! So the actual equation is
=>
(49^k)*(n/25^k)^(3/2)*log(n/25^k)

when you do the algebra, the 49^k term does vanish, which I'll show now…

Dynamic Programming

- All about making the **right choices**
- It’s difficult, but I’ll try to show some things to help you visualise the **memoisation** better
- A trick to doing this is to draw out the memoised values in a table
- You can also directly draw the recursion tree in determinate examples, like rod cutting

Rod Cutting

- First step,focus on solving a different problem - what's the maximum value we can get from the rod
- This allows us to make this equation:

$$m[w] = \max_{w_i \leq w}(v_i + m[w - w_i])$$

(intuition: ASSUMING that you make this cut, the optimal remaining cuts are ALSO the optimal for the remaining length. Simply take the maximum of all these assumptions)

(You're not expected to be able to think up of something this clever yourself, don't worry, but you may be walked through a derivation like in 2015's paper, which I'll explain here)

Rod cutting… but in an RPG?!

```python
attrs = [
    {'cost': 2, 'effect': lambda x: x+2, 'name': 'bigger sword'},
    {'cost': 3, 'effect': lambda x: x+3, 'name': 'flaming sword'},
    {'cost': 3, 'effect': lambda x: x*2, 'name': 'pet lion'}
]

memo = {}
choices = {}

def rodcut(w):
    if w < 0:
        return -999

    if w not in memo:
        # initialise to 0
        memo[w] = 0
        for attr in attrs:
            # if you don't know lambdas, they are simply variables you can treat like functions
            attack = attr['effect'](rodcut(w - attr['cost']))
            # keep track of the max
            if attack > memo[w]:
                memo[w] = attack
                choices[w] = attr['name']

    return memo[w]


rodcut(5)
print(memo)
print(choices)
```

$$m[w] = \max_{w_i \leq w}(v_i + m[w - w_i])$$

Example in lecture notes, handout 2 page 16

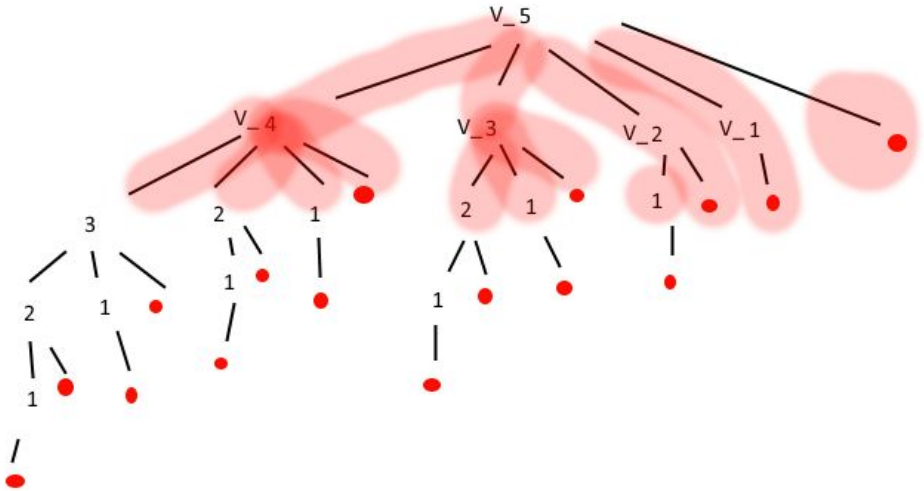
Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

```
skills = [
    {'cost': 1, 'effect': lambda x: x+1, 'name': '1'},
    {'cost': 2, 'effect': lambda x: x+5, 'name': '2'},
    {'cost': 3, 'effect': lambda x: x+8, 'name': '3'},
    {'cost': 4, 'effect': lambda x: x+9, 'name': '4'},
    {'cost': 5, 'effect': lambda x: x+10, 'name': '5'},
    {'cost': 6, 'effect': lambda x: x+17, 'name': '6'},
    {'cost': 7, 'effect': lambda x: x+17, 'name': '7'},
    {'cost': 8, 'effect': lambda x: x+20, 'name': '8'},
]
```

Memo table:

	w: 0	1	2	3	4	5	6	7	8
-∞...	-∞	-∞	0	1	5	8	10	13	17	18	22

Rod cutting recursion tree



2^n -> n^2

$$V_n = \begin{cases} \max_i(p_i + V_{n-i}) & n>0 \quad \text{RECURSE} \\ 0 & n=0 \quad \text{BASE CASE} \end{cases}$$

2015

relationships between c_1, c_2, c_3 that lead to a suboptimal packing if we use the above greedy algorithm. [5]

b) We now describe a dynamic programme to solve the knapsack problem optimally. To this end we introduce the following subproblems. Consider the items in some arbitrary order and let $C(v,i)$ be the optimal value one gets from solving the knapsack problem, with the first i items in the chosen order, and where the capacity of the rucksack is given by v. To solve the general problem, we have to find $C(W,n)$.

i) Derive a relationship between $C(v,i)$, $C(v',i-1)$, for some $v' \leq v$. [4]

ii) Propose an algorithm for finding $C(W,n)$ and the corresponding optimal packing. [6]

iii) Derive its complexity in terms of n and W. [4]

iv) Apply the above dynamic programme to the following example: $n = 5$, $W = 11$, $w_1 = 1, w_2 = 2, w_3 = 5, w_4 = 6, w_5 = 7$ and $c_1 = 1, c_2 = 6, c_3 = 18, c_4 = 22, c_5 = 28$, i.e. compute $C(11,5)$ and the items to be packed to achieve optimal packing. [6]

previous lecturer had knapsack in notes that's why it's not in answers! so I'll solve them here

Knapsack

- The exact same as rod cutting, except we can only use each length once
- This introduces a new constraint! Not only is the weight constrained, but each length can only be used once (or each skill can only be applied once in the RPG example)

$$m[i,\, w] = \max(m[i-1,\, w],\, m[i-1, w-w_i] + v_i)$$

(intuition: in this case, we only take the maximum of 2 options - one with the object taken, one without. IF TAKEN, the optimal for the remaining space in our bag is the optimal for a smaller bag of the same size, AND all the other choices. IF NOT TAKEN, the optimal is for the same for the same size bag, AND all the other choices.

Knapsack in the same RPG

```
attrs = [
    {'cost': 2, 'effect': lambda x: x+2, 'name': 'bigger sword'},
    {'cost': 3, 'effect': lambda x: x+3, 'name': 'flaming sword'},
    {'cost': 3, 'effect': lambda x: x*3, 'name': 'pet lion'},
]

memo = {}

def knapsack(w,i):
    if w < 0:
        return -999
    if i < 0:
        return 0

    if (w,i) not in memo:
        temp1 = knapsack(w,i-1)
        temp2 = attrs[i]['effect'](knapsack(w-attrs[i]['cost'], i-1))

        memo[(w,i)] = max(temp1, temp2)

    return memo[(w,i)]


knapsack(8,2)
print(memo)
```

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i)$$

Back to 2015, here's the equivalent

```
attrs = [
    {'cost': 1, 'effect': lambda x: x+1},
    {'cost': 2, 'effect': lambda x: x+6},
    {'cost': 5, 'effect': lambda x: x+18},
    {'cost': 6, 'effect': lambda x: x+22},
    {'cost': 7, 'effect': lambda x: x+28},
]

```

but the only reasonable way to fill this table is to go **bottom up!**

Example pseudocode (actual python available too)

```
memo[w,i] = -infinity if w<0 and 0 if i<0

function knapsack(_w,_i):
    for w from 0 to _w:
        for i from 0 to _i:
            temp1 = memo[w,i-1]
            temp2 = value[i] + memo[w-cost[i], i-1]
            memo[w,i] = max(temp1, temp2)
```

remember: $m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i)$

w \ i	-∞…	-∞…	-∞…	-∞…	-∞…
	0	0	0	0	0
	1	1	1	1	1
0…	1	6	6	6	6
0…	1	7	7	7	7
0…	1	7	7	7	7
0…	1	7	18	18	18
0…	1	7	19	22	22
0…	1	7	24	24	28
0…	1	7	25	28	29
0…	1	7	25	29	34
0…	1	7	25	29	35
	1	7	25	40	40

2016 question on polynomials: https://imperial.cloud.panopto.eu/Panopto/Pages/Sessions/List.aspx#folderID=%223c4974f8-3a3d-416f-bbcb-92167a26d6a0%22

Code from this lecture: https://github.com/jxz12/AlgoPlexRPG

- Some cool algorithm visualisations: https://bost.ocks.org/mike/algorithms/