COMP90038

Algorit plexity

Lecture 18: Dynami ng

(with thanks to Harald Sønde hael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

Recap

- **Hashing** is a standard way of implementing the abstract data type “dictionary”, a collection of <attribute name, value> pairs.

- A **key** *k* identifies each rec iently to a positive integer. The set *K* of keys can be u

- The **hash address** is calculated through a n $h(k)$, which points to a location in a **hash table**.
  - Two different keys could have the same address (a collision).

- The challenges in implementing a **hash table** are:
  - Design a robust hash function
  - Handling of same addresses (collisions) for different key values

Hash Functions

- The hash function:
  - Must be easy (cheap) to compute.
  - Ideally distribute keys evenly across the hash table.

- Three examples:
  - Integer: $h(n) = n \bmod m$.
  - Strings: sum of integers or concatenation of binaries

Concatenation of binaries

- Assume a binary representation of the 26 characters
  - We need **5 bits** per character (0 to 31)
- Instead of adding, we **concatenate** the binary strin
- Our hash table is of size 101 (***m* is prime**)
- Our key will be ‘MYKEY’

char	a	bin(a)	char	a	bin(a)	char	a	bin(a)
				9	01001	S	18	10010
				10	01010	T	19	10011
C	2			11	01011	U	20	10100
D	3			12	01100	V	21	10101
E	4			13	01101	W	22	10110
F	5	00101	O	14	01110	X	23	10111
G	6	00110	P	15	01111	Y	24	11000
H	7	00111	Q	16	10000	Z	25	11001
I	8	01000	R	17	10001			

Concatenating binaries

	STRING					KEY	KEY mod 101
	M	Y	K	E	Y		
int	12	24	10	4	24		
bin(int)	01100				11000		
Index	4				0		
32^(index)	1048576	32768	1024		1		
a*(32^index)	12582912	786432	10240	128	24	13379736	64

- By concatenating the strings, we are basically multiplying by 32
- We use Horner's rule to calculate the Hash:

$$p(x) = (((((a_3 \boxtimes x) \boxplus a_2) \boxtimes x) \boxplus a_1) \boxtimes x) \boxplus a_0$$

Handling Collisions

- Two main types:
  - Separate Chaining
    - Compared with **sequential search**, it reduces the number of comparisons by a factor of *m*
    - Good for dynamic envi
    - Deletion is easy
    - Uses more storage
  - Linear probing
    - Space efficient
    - Worst case performance is poor
    - It may lead to **clusters of contiguous cells** in the table being occupied
    - Deletion is almost impossible

Double Hashing

- **Double hashing** uses a second hash function s to determine an **offset** to be used in probing for a free cell.
    - It is used to alleviate the clustering problem in linear probing.

- For example, we may ch

- By this we mean, if $h(k)$ is occupied, he $s(k)$, then $h(k) + 2s(k)$, and so on.

- This is another reason why **it is good to have *m* being a prime number**. That way, using $h(k)$ as the offset, we will eventually find a free cell if there is one.

Rehashing

- The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to **rehash** when it reaches, say, 0.9.

- Rehashing means alloc le (typically about twice the current size), revisiting each ite ing its hash address in the new table, and inserting it.

- This **“stop-the-world”** operation will introduce long delays at unpredictable times, but it will happen relatively infrequently.

An exam question type

- With the hash function $h(k) = k \bmod 7$. Draw the hash table that results after inserting in the given order, the following values

- When collisions are ha

  - separate chaining

  - linear probing
  - double hashing using $h'(k) = 5 - (k \bmod 5)$

- Which are the hash addresses?

Solution

Index	0	1	2	3	4	5	6
Separate Chaining						19	13
						26	48
Linear Probing	13	48		17		19	26
Double Hashing		48	26	17		19	13

Rabin-Karp String Search

- The Rabin-Karp string search algorithm is based on string hashing.

- To search for a string p (of length m) in a larger string s, we can calculate *hash*(p) and then check every subs [illegible] has the same hash value. Of course, if it has, the string [illegible] we need to compare them in the usual way.

- If $p = s_i \ldots s_{i+m-1}$ then the hash values are the same; otherwise the values are **almost certainly** going to be different.

- Since false positives will be so rare, the $O(m)$ time it takes to actually compare the strings can be ignored.

Rabin-Karp String Search

- Repeatedly hashing strings of length *m* seems like a bad idea. However, the hash values can be calculated **incrementally**. The hash value of the length-*m* substring of *s* that starts at position *j* is:

- where *a* is the alphabet size. From that w next hash value, for the substring that starts at position *j*+1, **quite cheaply**:

$$\text{hash}(s, j+1) = (\text{hash}(s, j) - a^{m-1}\text{chr}(s_j)) \times a + \text{chr}(s_{j+m})$$

- modulo *m*. Effectively we just subtract the contribution of s_j and add the contribution of s_{j+m}, for the cost of two multiplications, one addition and one subtraction.

An example

- The data ‘31415926535’
- The hash function $h(k) = k \bmod 11$
- The pattern ’26’

STRING	3	1	4	1				6	5	3	5
31 MOD 11		9									
14 MOD 11			3								
41 MOD 11				8							
15 MOD 11					4						
59 MOD 11						4					
92 MOD 11							4				
26 MOD 11								4			

Why Not Always Use Hashing?

- Some drawbacks:

  - If an application calls f sorted order, a hash table is no good.

  - Also, unless we use separate chaining, irtually impossible.

  - It may be hard to predict the volume of data, and rehashing is an expensive “stop-the-world” operation.

When to Use Hashing?

- All sorts of information retrieval applications involving thousands to millions of keys.

- Typical example: Symbol t The compiler hashes all (variable, function, etc.) n ion related to each – no deletion in this case.

- When hashing is applicable, it is usually superior; a well-tuned hash table will outperform its competitors.

- **Unless** you let the load factor get too high, or you botch up the hash function. It is a good idea to print statistics to check that the function really does spread keys uniformly across the hash table.

Dynamic programming

- **Dynamic programming** is a bottom-up problem solving technique. The idea is to divide the problem into smaller, overlapping ones. The results are tabulated and used to find the complete solution.

- An example is the approach o find the Fibonacci numbers:


```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    result ← F[n]
    if result = 0 then
        result ← FIB(n − 1) + FIB(n − 2)
        F[n] ← result
    return result
```

- an array that stores partial results, initialized to zero
- If F[n]=0, then this partial result has not been calculated, hence the recursion is calculated
- If F[n]≠0, then this value is used.

Dynamic programming and Optimization

- Dynamic programming is often used on **Optimization** problems.
  - The objective is to find the solution with the lowest cost or highest profit.

- For dynamic program **ptimality principle** must be true:

  - An optimal solution to a problem is co ptimal solutions to its subproblems.

- While not always true, this principle holds more often than not.

Dynamic programming vs. Divide-and-Conquer

- While the two techniques divide the problem into smaller ones, there is a basic difference:

  - In D&C, the sub-problech other, while in DP the problems are **depende**

  - Because of the dependencies, in DP w the solutions to the sub-problems in order to be re-used. That does not happen in D&C.

- Think about MergeSort for a moment. Do you keep the solution from one branch to be re-used in another?

The coin row problem

- You are shown a group of coins of different denominations ordered in a row.

- **You can keep some of** **o not pick two adjacent ones.**

  - Your objective is to **maximize your pro** want to take the largest amount of money.

- This type of problems are called **combinatorial**, as we are trying to find the **best** possible **combination** subject to some **constraints**

The coin row problem

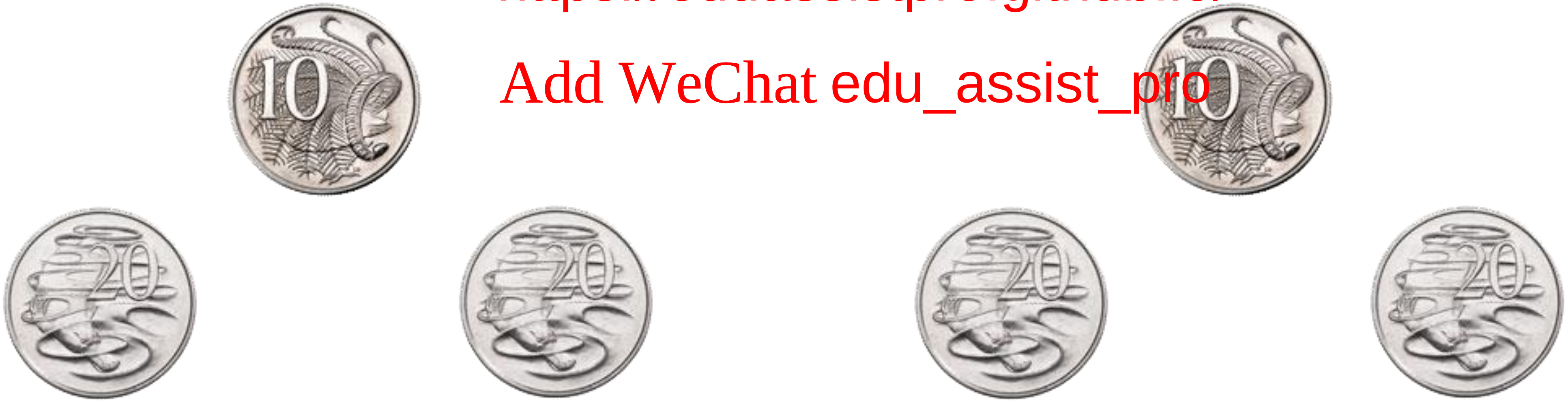
- Let's visualize the problem. Our coins are [20 10 20 50 20 10 20]

The coin row problem

- We cannot take these two.
  - It does not fulfil our constraint (We cannot pick adjacent coins)

The coin row problem

- We could take all the 20s (Total of 80).
  - Is that the maximum profit? Is this a greedy solution?



The coin row problem

- Can we think of a **recursion** that help us solve this problem? What is the smallest problem possible?

- If instead of a row of s d one coin
  - We have only one choice.

- What about if we had a row of two?
  - We either pick the first or second coin.

The coin row problem

- If we have a row of three, we can pick the middle coin or the two in the sides. Which one is the optimal?

The coin row problem

- If we had a row of four, there are sixteen combinations

- For simplicity, I will repres as binary strings:
  - ‘0’ = leave the coin
  - ‘1’ = pick the coin

- Eight of them are not valid (in optimization lingo **unfeasible**), one has the worst profit (0)

- Picking one coin will always lead to lower profit (in optimization lingo **suboptimal**)

0	0000	PICK NOTHING (NO PROFIT)
1	0001	SUBOPTIMAL
2	0010	SUBOPTIMAL
3	0011	UNFEASIBLE
4	0100	SUBOPTIMAL
5	0101	
6	0110	UNFEASIBLE
7	0111	UNFEASIBLE
8	1000	SUBOPTIMAL
9	1001	
10	1010	
11	1011	UNFEASIBLE
12	1100	UNFEASIBLE
13	1101	UNFEASIBLE
14	1110	UNFEASIBLE
15	1111	UNFEASIBLE

The coin row problem

- Let's give the coins their values [c_1 c_2 c_3 c_4], and focus on the **feasible** combinations:
  - Our choice is to pick two coins [c_1 0 c_3 0] [0 c_2 0 c_4] [c_1 0 0 c_4]

- If the coins arrived in sequ reach c_4, the best that we can do is either:

  - Take a solution at step 3 [c_1 0 c_3 0]
  - Add to one of the solutions at step 2 the new [c_1 0 0 c_4]

- Generally, we can express this as the recurrence:

$$S(n) = \max\left(c_n + S\left(n-2\right), S\left(n-1\right)\right) \;\text{ for } n > 1$$

$$S(1) = c_1$$

$$S(0) = 0$$

The coin row problem

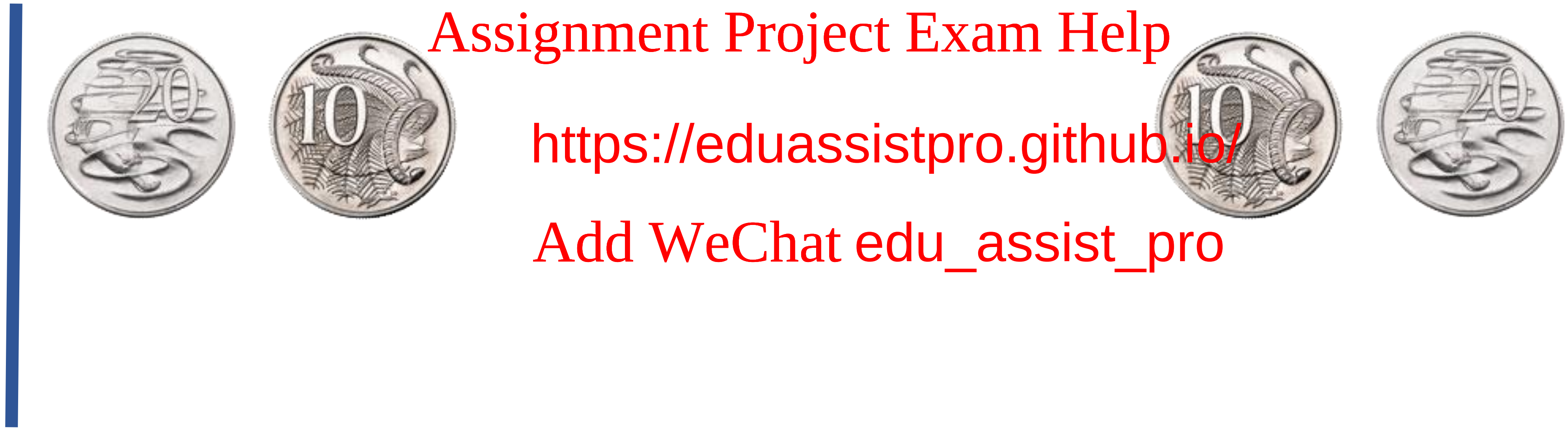
- Given that we have to backtrack to S(0) and S(1), we store these results in an array.

- Then the algorithm is:

$$
\begin{array}{l}
\textbf{function } \textsc{CoinRow}(C[\cdot], n) \\
\quad S[0] \leftarrow 0 \\
\quad S[1] \leftarrow C[1] \\
\quad \textbf{for } i \leftarrow 2 \text{ to } n \textbf{ do} \\
\quad\quad S[i] \leftarrow max(S[i-1], S[i-2] + C[i]) \\
\quad \textbf{return } S[n]
\end{array}
$$

The coin row problem

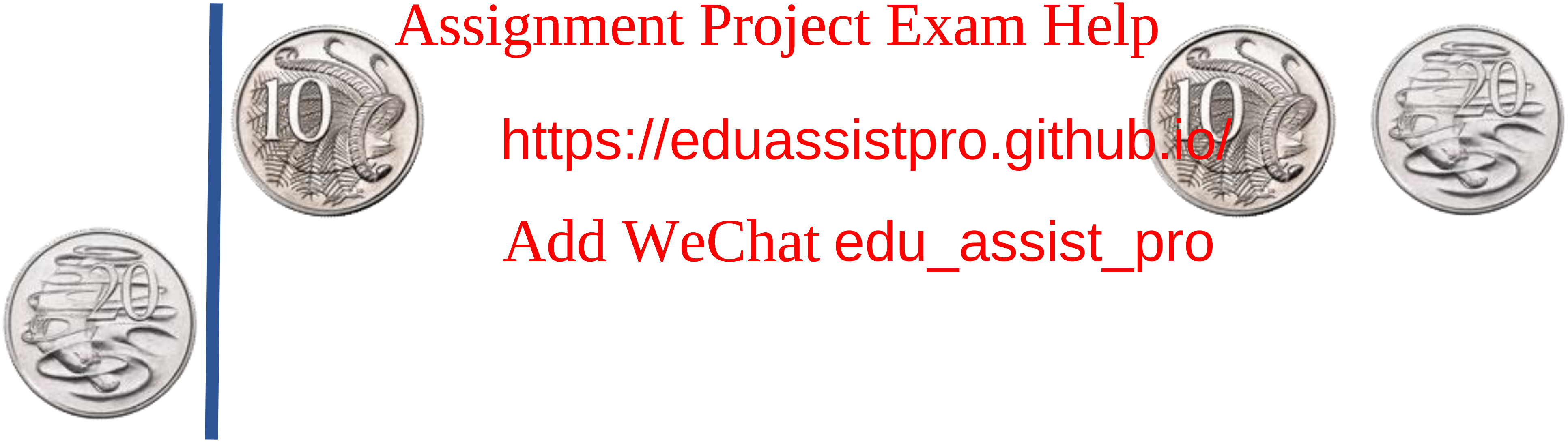
- Lets run our algorithm in the example. Step 0.



- S[0] = 0.

The coin row problem

- Step 1



- S[1] = 20

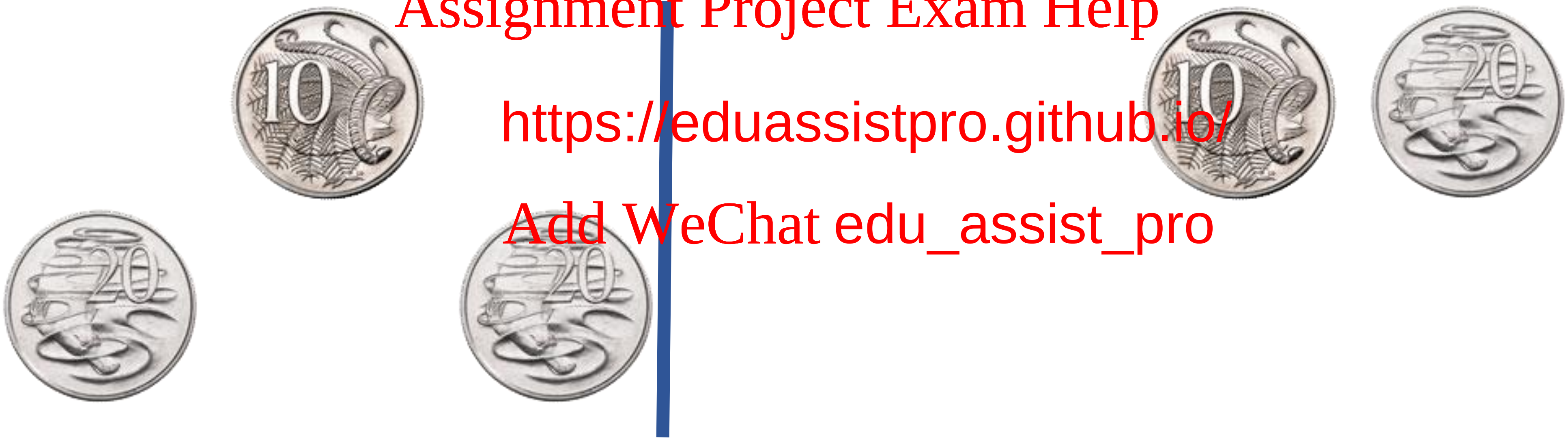
The coin row problem

- Step 2

- S[2] = max(**S[1] = 20**, S[0] + 10 = 0 + 10) = 20

The coin row problem

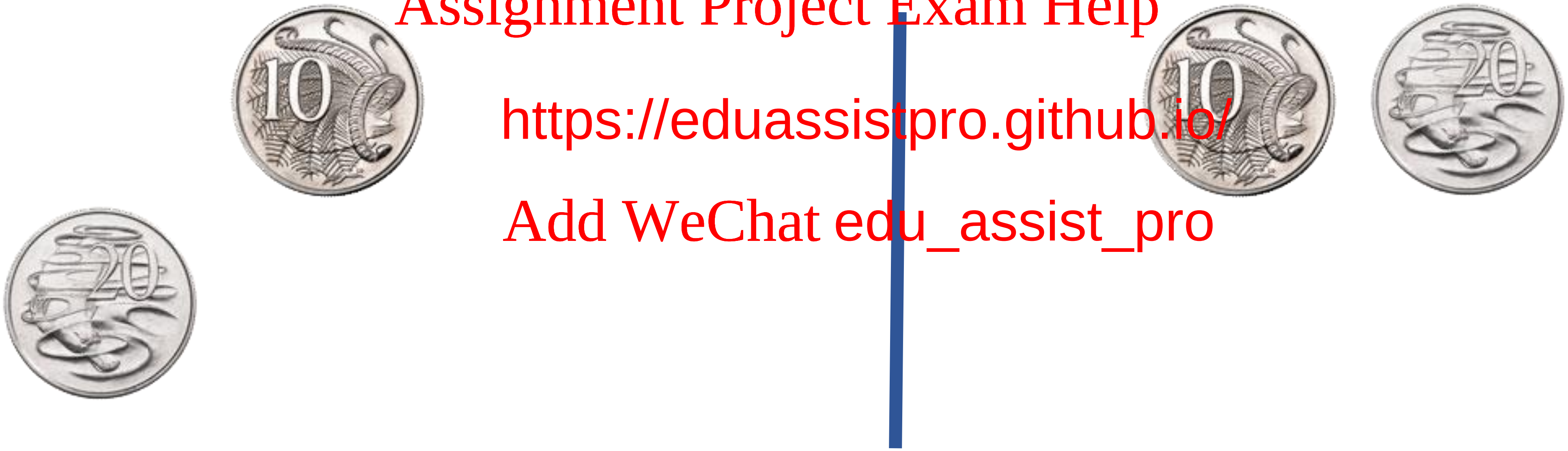
- Step 3



- S[3] = max(S[2] = 20, **S[1] + 20 = 20 + 20 = 40**) = 40

The coin row problem

- Step 4



- S[4] = max(S[3] = 40, **S[2] + 50 = 20 + 50 = 70**) = 70

The coin row problem

- At step 5, we can pick between:
  - **S[4] = 70**
  - S[3] + 20 = 60

- At step 6, we can pick b
  - S[5] = 70
  - **S[4] + 10 = 80**

- At step 7, we can pick between:
  - S[6] = 80
  - **S[5] + 20 = 90**

		1	2	3	4	5	6	7
	0	20	10	20	50	20	10	20
STEP 0	0							
	0	20						
	0	20	20					
		20	20	40				
ST		20	20	40	70			
ST		20	20	40	70	70		
STEP 6	0	20	20	40	70	70	80	
STEP 7	0	20	20	40	70	70	80	90

SOLUTION		1	1	1	1	1	1	1
				3	4	4	4	4
							6	7

Two insights

- In a sense, dynamic programming allows us to take a step back, such that we pick the best s **wly arrived information.**

- If we used a brute-force approach s a**ustive search** for this problem:
  - We had to test 33 feasible combinations.
  - Instead we tested 5 combinations.

The knapsack problem

- You previously encountered the **knapsack problem**:

- Given a list of n items with:
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$

- and a knapsack (container) of capacity W

- Find the **combination** of items with the **highest value** that would **fit into the knapsack**

- All values are positive integers

The knapsack problem

- This is another combinatorial optimization problem:
  - In both the coin row a are **maximizing profit**
  - Unlike the coin row problem which ha **le** <coin value>, we now have **two variables** <item weight, item

The knapsack problem

- The critical step is to find a good answer to the question: **what is the smallest version of the problem that I could solve first?**
  - Imagine that I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**
  - What if the capacity was 2 and the weight 1. Does it fit? **Do I have capacity left?**

- Given that we have **two v** lation is formulated over **two parameters**:

  - the **sequence of items considered so far** {1, 2,
  - the **remaining capacity** $w \leq W$.

- Let $K(i,w)$ be the value of the best choice of items amongst the first i using knapsack capacity w.
  - Then we are after $K(n,W)$.

The knapsack problem

- By focusing on $K(i,w)$ we can express a recursive solution.

- Once a new item i arri it or not.

  - **Excluding *i*** means that at is, which items were selected before i arrived with the sam apacity.

  - **Including *i*** means that the solution also includes the subset of previous items **that will fit into a bag of capacity $\boldsymbol{w-w_i}$** ≥ 0, i.e., $K(i-1,w-w_i) + v_i$.

The knapsack problem

- Let us express this as a recursive function.

- First the base **state**:

- Otherwise:

$$K(i,w) = \begin{cases} \max(K(i-1,w), K(i-1,w-w_i)+v_i) & \text{if } w \geq w_i \\ K(i-1,w) & \text{if } w < w_i \end{cases}$$

The knapsack problem

- That gives a correct, although inefficient, algorithm for the problem.

- For a bottom-up solutio to write the code that systematically fills a **two-****dimensional table** of n+1 rows and W+1 columns.

- The algorithm has both time and space complexity of $O(nW)$

The knapsack problem

- Lets look at the algorithm, step-by-step.

- The data is:
  - The knapsack capacity
  - The values are {42, 12, 40, 25}
  - The weights are {7, 3, 4, 5}

The knapsack problem

- On the first **for loop**:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0								
42	7	1		0								
				0								
				0								
				0								

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

The knapsack problem

- On the second **for loop**:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0								
				0								
				0								
				0								

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

The knapsack problem

- Now we advance row by row:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0								
				0								
				0								
				0								

for $i \leftarrow 0$ to n **do**
    $K[i,0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0,j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i,j] \leftarrow K[i-1,j]$
        else
            $K[i,j] \leftarrow max(K[i-1,j], K[i-1,j-w_i]+v_i)$
return $K[n,W]$

The knapsack problem

- Is the current capacity (j=1) sufficient?

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	?							
				0								
				0								
				0								

for $i \leftarrow 0$ to n **do**
    $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow K[i-1, j]$
        else
            $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

The knapsack problem

- We won’t have enough capacity until j=7

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
				0								
				0								
				0								

for $i \leftarrow 0$ to n **do**
    $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow K[i-1, j]$
        else
            $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

- i = 1
- j = 7
- K[1-1,7] = K[0,7] = 0
- K[1-1,7-7] + 42 = K[0,0] + 42 = 0 + 42 = 42

The knapsack problem

- Next row. We won't have enough capacity until j=3

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
				0	0	0	12					
				0								
				0								


```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i = 2
- j = 3
- K[2-1,3] = K[1,3] = 0
- K[2-1,3-3] + 12 = K[1,0] + 12 = 0 + 12 = 42

The knapsack problem

- But at *j*=7, it is better to pick 42

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
				0	0	0	12	12	12	12	42	
				0								
				0								


```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- $i = 2$
- $j = 7$
- $K[2\text{-}1,7] = K[1,7] = 42$
- $K[2\text{-}1,7\text{-}3] + 12 = K[1,4] + 12 = 0 + 12 = 12$

The knapsack problem

- Next row: at j=4, it is better to pick 40

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
				0	0	0	12	12	12	12	42	42
				0	0	0	12	40				
				0								

- i = 3
- j = 4
- K[3-1,4] = K[2,4] = 12
- K[3-1,4-4] + 40 = K[2,0] + 40 = 0 + 40 = 40

The knapsack problem

- What would happen at j=7?
- Can you complete the table?

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
				0	0	0	12	12	12	12	42	42
				0	0	0	12	40	40	40		
				0								

for $i \leftarrow 0$ to n **do**
    $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow K[i-1, j]$
        else
            $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill:
  - It finds the solution to [illegible] **tance**, most of which are unnecessary

- In this situation, a top-down approach, with **memoing**, is preferable.
  - There are many implementations of the memo table.
  - We will examine a simple array type implementation.

The knapsack problem

- Lets look at this algorithm, step-by-step

- The data is:
  - The knapsack capacity W = 8
  - The values are {42, 12, 40, 25}
  - The weights are {7, 3, 4, 5}


```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if F(i, j) < 0 then
        if j < w(i) then
            value = MFKnap(i − 1, j)
        else
            value = max(MFKnap(i − 1, j), v(i) + MFKnap(i − 1, j − w(i)))
        F(i, j) = value
    return F(i, j)
```

- F is initialized to all -1, with the exceptions of i=0 and j=0, which are initialized to 0.

The knapsack problem

- We start with *i*=4 and *j*=8

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	**-1**

function $\text{MFKNAP}(i, j)$
    if $i < 1$ or $j < 1$ **then**
        return 0
    if $F(i, j) < 0$ **then**
        if $j < w(i)$ **then**
            $value = \text{MFKNAP}(i-1, j)$
        else
            $value = \max(\boxed{\text{MFKNAP}(i-1, j)}, v(i) \boxed{+ \text{MFKNAP}(i-1, j-w(i))})$
        $F(i, j) = value$
    return $F(i, j)$

- *i* = 4
- *j* = 8
- *K*[4-1,8] = K[3,8]
- *K*[4-1,8-5] + 25 = *K*[3,3] + 25

The knapsack problem

- Next is *i*=3 and *j*=8

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	**-1**
				0	-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFKNAP}(i, j)$
    if $i < 1$ or $j < 1$ **then**
        return 0
    if $F(i, j) < 0$ **then**
        if $j < w(i)$ **then**
            $value = \text{MFKNAP}(i-1, j)$
        else
            $value = \max(\boxed{\text{MFKNAP}(i-1, j)}, \boxed{v(i) + \text{MFKNAP}(i-1, j-w(i))})$
        $F(i, j) = value$
    return $F(i, j)$

- *i* = 3
- *j* = 8
- *K*[3-1,8] = K[2,8]
- *K*[3-1,8-4] + 40 = *K*[2,4] + 40

The knapsack problem

- Next is *i*=2 and *j*=8

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	**-1**
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1

function $\mathrm{MFK{\scriptsize NAP}}(i, j)$
    if $i < 1$ or $j < 1$ **then**
        return 0
    if $F(i, j) < 0$ **then**
        if $j < w(i)$ **then**
            $value = \mathrm{MFK{\scriptsize NAP}}(i-1, j)$
        else
            $value = \max(\boxed{\mathrm{MFK{\scriptsize NAP}}(i-1, j)}, v(i) \boxed{+ \mathrm{MFK{\scriptsize NAP}}(i-1, j-w(i))})$
        $F(i, j) = value$
    return $F(i, j)$

- i = 2
- j = 8
- K[2-1,8] = K[1,8]
- K[2-1,8-3] + 12 = K[1,5] + 12

The knapsack problem

- Next is *i*=1 and *j*=8
- Here we reach the bottom of this recursion

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	**0**
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	**42**
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1

function $\mathrm{MFK}\textsc{nap}(i, j)$
    if $i < 1$ or $j < 1$ **then**
        return 0
    if $F(i, j) < 0$ **then**
        if $j < w(i)$ **then**
            $value = \mathrm{MFK}\textsc{nap}(i-1, j)$
        else
            $value = \max(\boxed{\mathrm{MFK}\textsc{nap}(i-1, j)}, \boxed{v(i) + \mathrm{MFK}\textsc{nap}(i-1, j-w(i))})$
        $F(i, j) = value$
    return $F(i, j)$

- *i* = 1
- *j* = 8
- *K*[1-1,8] = K[0,8] = 0
- *K*[1-1,8-7] + 42 = *K*[0,1] + 42 = 0 + 42 = 42

The knapsack problem

- Next is *i*=1 and *j*=5.
- As before, we also reach the bottom of this branch.

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	0	-1	-1	42
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1
				0	-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFKNAP}(i, j)$
  if $i < 1$ or $j < 1$ **then**
    return 0
  if $F(i, j) < 0$ **then**
    if $j < w(i)$ **then**
      $value = \text{MFKNAP}(i - 1, j)$
    else
      $value = \max(\boxed{\text{MFKNAP}(i - 1, j)}, v(i) \boxed{+ \text{MFKNAP}(i - 1, j - w(i)))}$
    $F(i, j) = value$
  return $F(i, j)$

- $i = 1$
- $j = 5$
- $K[1\text{-}1,5] = K[0,5] = 0$
- j - $w[1] = 5\text{-}8 < 1 \rightarrow$ return 0

The knapsack problem

- We can trace the complete algorithm, until we find our solution.

- The states visited (18) .

  - Unlike the bottom-up approach, in wh d all the states (40).

- Given that there are a lot of places in the table never used, the algorithm is less space-efficient.
  - You may use a hash table to improve space efficiency.

i	j	value
0	8	0
0	1	0
1	8	42
0	5	0
1	5	0
2	8	42
0	4	0
1	4	0
0	1	0
1	1	0
2	4	12
3	8	52
0	3	0
1	3	0
1	0	0
2	3	12
3	3	12
4	8	52

A practice challenge

- Can you solve the problem in the figure?

  - W = 15
  - w = [1 1 2 4 12]
  - v = [1 2 2 10 4]

- FYI the answer is $15/15Kg

Next lecture

- We apply dynamic pro h problems (transitive closure and all-pairs sh lting algorithms are known as **Warshall's** and **Floyd's**.