

---

## Dynamic Programming Notes

This file includes my learning notes of Youtube video (<https://youtu.be/oBt53YbR9Kk>). Hats off to the lecturer, Alvin Zablan, for such a great tutorial. The content list is given as

### Contents

Section I Motivation of having dynamic programming .....	2
Section II Memoisation .....	5
Section III Grid Traveller problem.....	6
Section IV Can-Sum issue.....	8
Section V How-Sum Problem .....	10
Section VI Best-Sum Problem.....	11
Section VII Can-Construct Problem.....	12
Section VII All-Construct Problem .....	14
Section VIII Tabulation.....	15
Section IV Can-Sum Tabulation .....	16
Section VI Construct Tabulation .....	18
Section VII Conclusions.....	20

## Section I Motivation of having dynamic programming

Questions such as calculating the 40th number of the Fibonacci sequence and counting the number of different ways to move through a 6x9 grid can all be categorised into the dynamic programming (DP) cluster. There are two main parts in DP, Memoisation (记忆化) and Tabulation (列表化).

To understand why we need DP, we need to first settle the concept of algorithm complexity. In Java, the recursive algorithm for Fibonacci sequence can be given as Fig. 1.

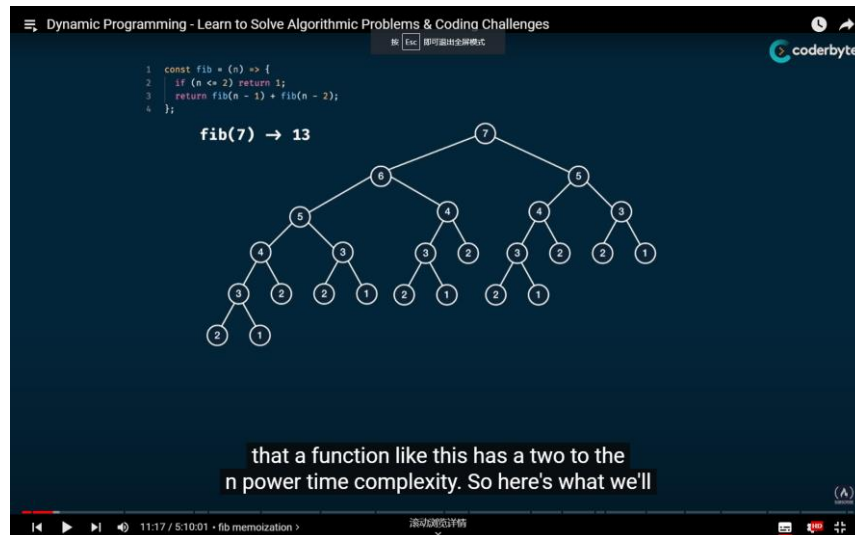


Fig. 1 Illustration recursive algorithm for Fibonacci sequence

However, it seems a bit hard for us to directly consider the complexity of the algorithm. Hence, let's pick another example in Fig. 2. If we want to obtain the fifth element in the sequence, we need to calculate for 5 times. This indicates that the time complexity of the algorithm is  $O(n)$ . Similarly, we have that the space complexity is also  $O(n)$ .



Fig. 2 Illustration of a simple algorithm

If we consider another simple algorithm as shown in Fig. 3, the time complexity is  $O(n/2)$ , but in computer science, we treat it the same as  $O(n)$ . The space complexity is also  $O(n)$ . If you are careful enough, you may have discovered that the algorithm in Fig. 1 is a combination of the one in Fig. 2 and the one in Fig. 3.

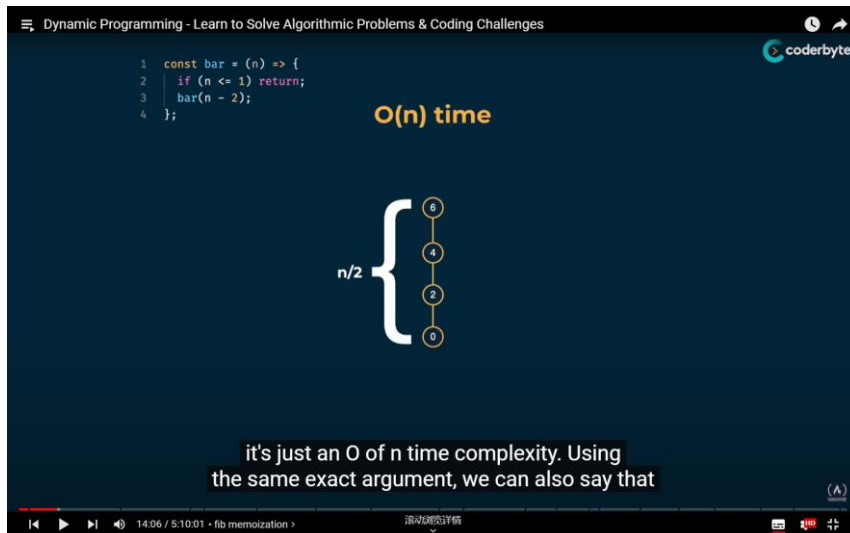


Fig. 3 Another simple algorithm

Consider another “Dib” function as illustrated in Fig. 4. Based on previous discussions, we can see that the time complexity of the Fibonacci algorithm is  $O(2^{n/2})$ , which is treated as  $O(2^n)$ . Hence, if we run the above problem with the recursive design, then the complexity of the algorithm is also  $O(2^n)$ . Meanwhile, the space complexity is  $O(n)$  because if we want to calculate the value for  $n = 5$ , every path we need to calculate contains five element at most (5-4-3-2-1).

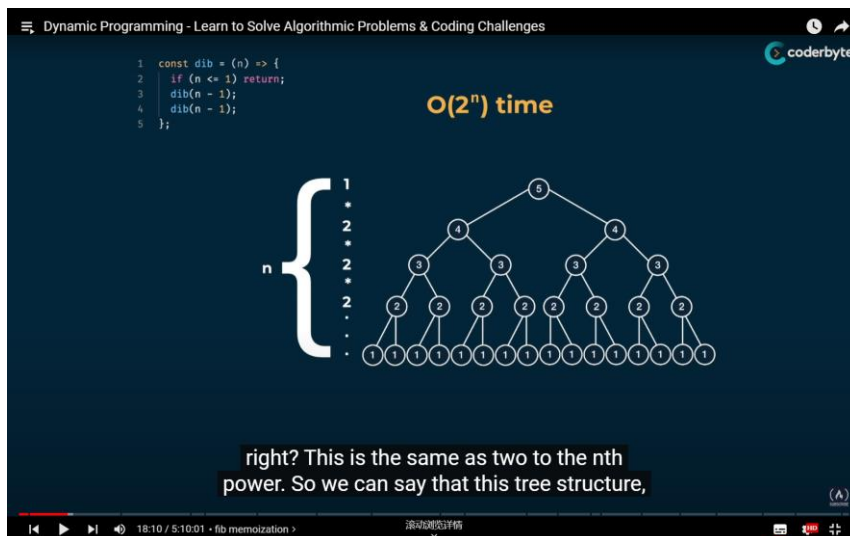


Fig. 4 Complexity of the “Dib” algorithm

Consider another function “Lib” in Fig. 5. Based on our discussion regarding Fig. 4, we know that the time complexity of the algorithm is  $O(2^{n/2})$  because the “height” of the algorithm tree is approximately  $n/2$ . According to the concept, the actual time complexity is  $O(2^n)$ .

As discussed, the Fibonacci algorithm can be seen as the combination of “Foo” and “Bar”, which is equivalent to “Fib = Foo + Bar”. In the same sense, we also have “Foo = Dib/2” and “Bar = Lib/2” and the fact that “Dib  $\geq$  Lib” in the sense of time complexity. Therefore, for the Fibonacci algorithm, we have “Lib  $\leq$  Fib  $\leq$  Dib” in the sense of time complexity. Hence, we have that the time complexity of the recursive Fibonacci algorithm is also  $O(2^n)$ , which explains why it takes “forever” for us to calculate the 50<sup>th</sup> element of the Fibonacci sequence by recursive design.

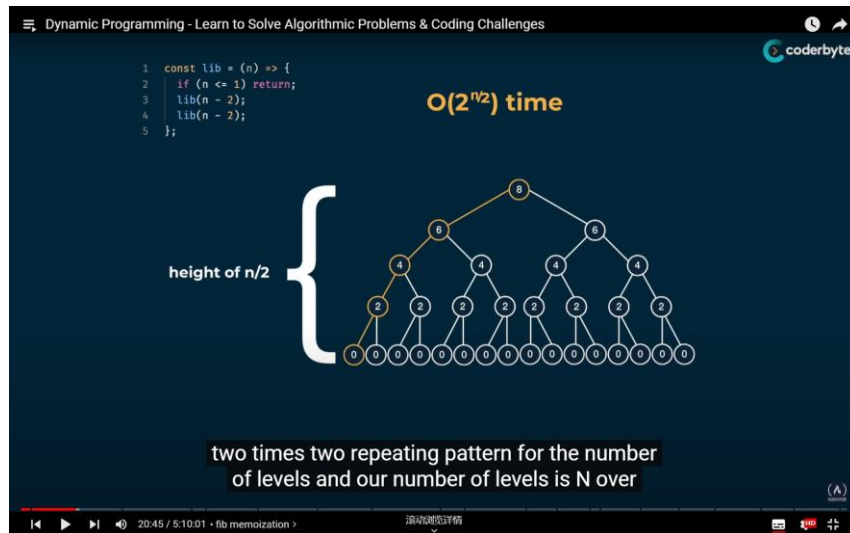


Fig. 5 Complexity of the “Lib” function

Now we have our motivation, to reduce time complexity. To solve this issue, we also need to understand “what’s wrong with the recursive design”. From the algorithm tree in Fig. 6, we can see that a large portion of the calculation is duplicated. Then we have our goal, which is to memorise what we have obtained and trim those unnecessary steps.

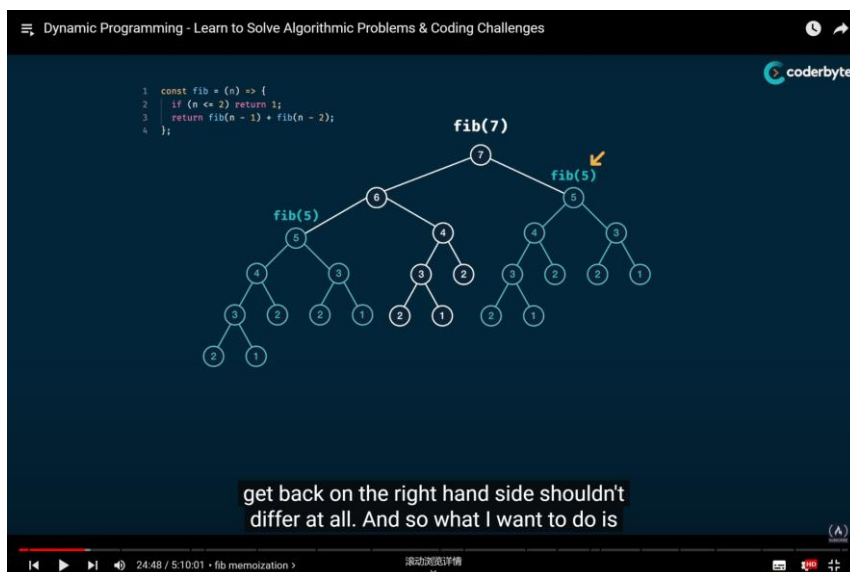
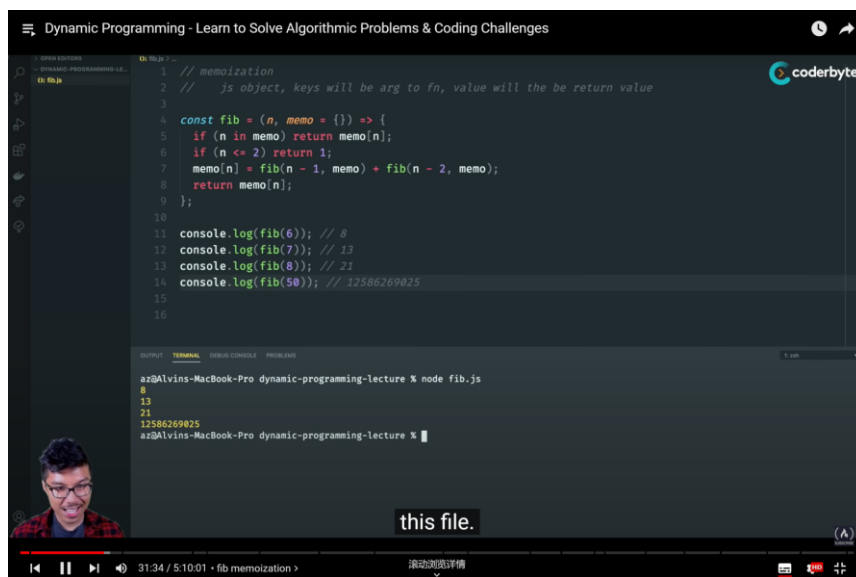


Fig. 6 What’s wrong with the recursive design?

## Section II Memoisation

To reduce complexity, we can introduce one notebook “Memo” into the algorithm so that we can store what we have calculated to avoid duplicate running (see Fig. 7). Now, we can get the values of a Fibonacci sequence in a much faster speed!

Judging by the running speed, we can see that the algorithm design is significantly improved, which means that the algorithm tree of the DP process should be a lot different than the one of the recursive design. In Fig. 8, we present the new algorithm tree of the DP process. You can see that a lot of the unnecessary calculations are omitted, which saves our time. From the figure, we can see that the time complexity and the space complexity of the DP structure are both  $O(n)$ .



The screenshot shows a video player interface with a code editor. The code is a JavaScript function for calculating Fibonacci numbers using memoization. The output in the terminal shows the results of calling the function for values 6, 7, 8, and 50.

```
1 // memoization
2 // js object, keys will be arg to fn, value will be the return value
3
4 const fib = (n, memo = {}) => {
5   if (n in memo) return memo[n];
6   if (n <= 2) return 1;
7   memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
8   return memo[n];
9 };
10
11 console.log(fib(6)); // 8
12 console.log(fib(7)); // 13
13 console.log(fib(8)); // 21
14 console.log(fib(50)); // 12586269025
15
16
```

Terminal output:

```
az@Alvins-MacBook-Pro dynamic-programming-lecture % node fib.js
8
13
21
12586269025
az@Alvins-MacBook-Pro dynamic-programming-lecture %
```

Fig. 8 The updated DP structure

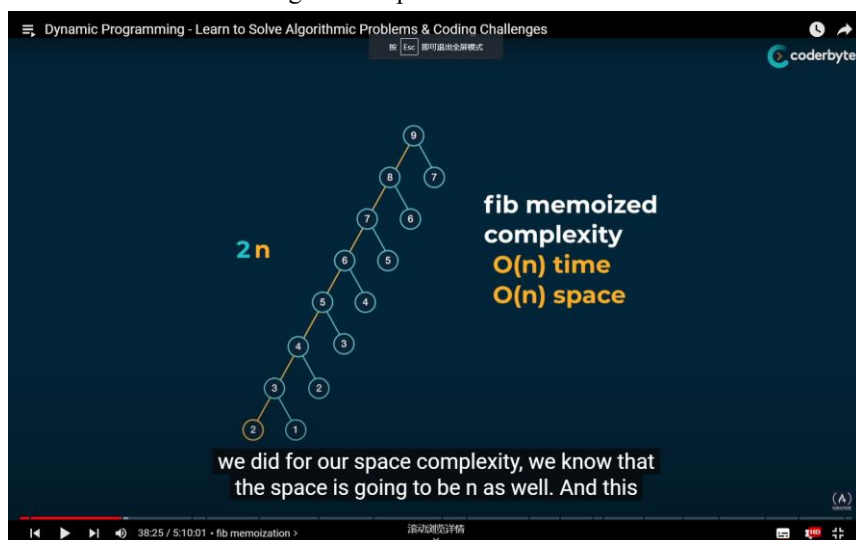


Fig. 9 The new algorithm tree when DP is applied

### Section III Grid Traveller problem

Now we move on to a more complex problem, the grid traveller issue. Suppose you are a traveller on a 2D grid. You begin in the top-left corner and your goal is to travel to the bottom-right corner. You can only move down or right, then in how many ways can you travel to your destination?

Before going deep into this question, let's first settle some basic scenarios:

1. When at least one of the row-column dimensions is 0, the answer is 0 because we can't go anywhere.
2. When at least one of the row-column dimensions is 1, the answer is 1 because there's only one way.

The above two scenarios can be treated as the basic child circumstances that don't need to be calculated. Similar to the Fibonacci issue, we can see that the time complexity of the recursive design is  $O(2^{n+m})$ , where  $n$  and  $m$  are row number and column number, respectively. Meanwhile, judging by the height of the tree, we can see that the space complexity is  $O(n + m)$ .

Then our main concern is actually the same, to memorise the results we have obtained to avoid duplication. However, the issue is a bit more complex because we have multiple variables that determine the result of one function. Hence, the concept of dictionary is introduced. By creating a dictionary, we can have a one-on-one match between a key (which includes the variables we have in a string/integer format) and the function value (the number of possible paths from the beginning to the destination).

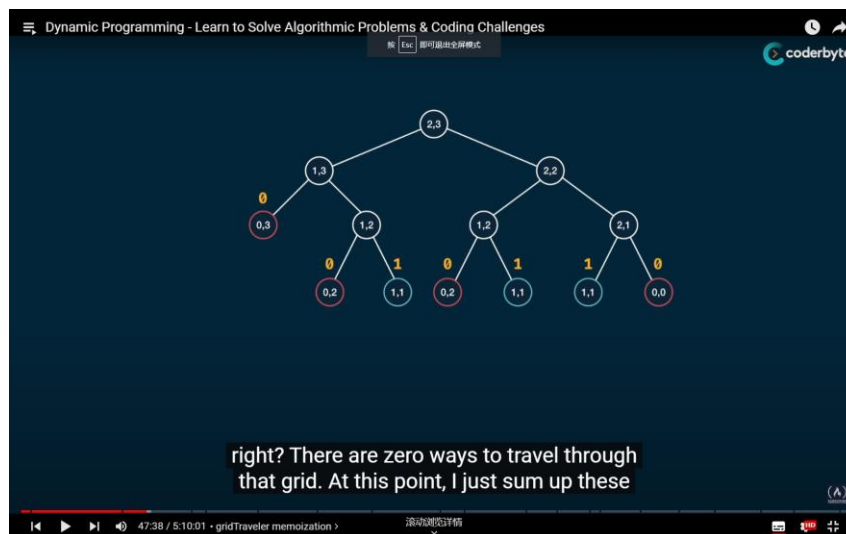


Fig. 10 The algorithm tree when row=2, column=3

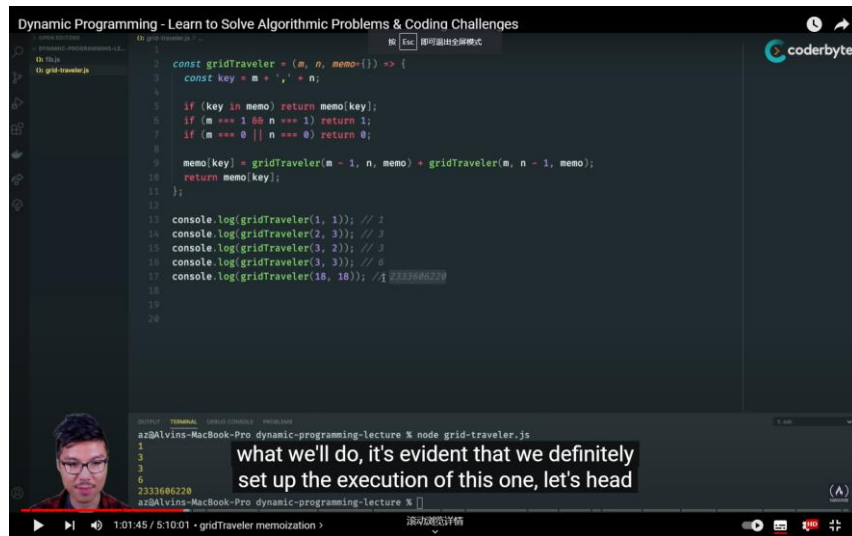
By the time we have a set of input (row and column numbers), we can start the irritation by asking if the keys we are looking for already has a corresponding answer. If yes, then we just use the answer directly. If not, then we calculate the current circumstance by adding up the result of two child circumstances, where we have one less row, and where we have one less column, which further leads to Fig. 11.

In the sense of time complexity, because there will roughly be  $m * n$  kinds of situation in total (actually a bit less because we stop calculating when row or column is 1). Hence, we made an improvement from  $O(2^{n+m})$  to  $O(n + m)$  in time complexity, while the space complexity stays the same.

To sum up, we need to follow a two-step procedure to have a good DP structure.

Step 1: Visualise the problem as a tree and make it work in a recursive design.

Step 2: Introduce a memo object (or a notebook as you may want) to store the answers you have calculated to make your algorithm efficient. Also, always have basic cases that directly returns values.



```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

what we'll do, it's evident that we definitely set up the execution of this one, let's head

Fig. 11 Algorithm design of the grid traveller issue

#### Section IV Can-Sum issue

Now we consider a new problem, the Can-Sum issue. The question is, suppose we have a target number  $m$  and a possible selection array  $S$ , which includes  $n$  nonnegative independent variables. If we assume that we can use every independent element in  $S$  for nonnegative number of times. Then our goal is to develop an algorithm that can help us determine if there is a combination of arbitrary elements from  $S$  that can help us obtain the target number  $m$ .

For example, if we have  $m = 7$ , then we should obtain a Boolean answer “True” when  $S = [3,4]$  because we have  $m = 3 + 4$ . Accordingly, when  $S = [2, 4]$ , we will return “False” because we will run out of choices.

Again, we start with the recursive design, which means that there will be no “memory-based” structure that records the circumstances we have calculated. While we are trying to determine the time complexity, we need to consider the worst-case scenario. As usual, we will use the algorithm tree for illustration. First, we need to know about the height of the tree. Suppose we have element “1” in  $S$ , then the longest path we have contains  $m$  nodes in total.

For each node, we will have  $n$  different choices to check. Hence, each node would split into  $n$  different sub-nodes in the next level. Therefore, the time complexity of the algorithm can be given as  $O(n^m)$ . Because the height of the tree is  $m$  nodes, then the space complexity is  $O(m)$ .

**Remark 1.** Regarding the worst-case scenario that we have element “1” in  $S$ , the longest path should contain  $m + 1$  nodes because the iteration won’t stop until we hit “0” or something negative. However, if there is a “1” in  $S$ , then every nonnegative element should result in “True”. Hence, we have that the height of the algorithm tree is at most  $m$  nodes high.

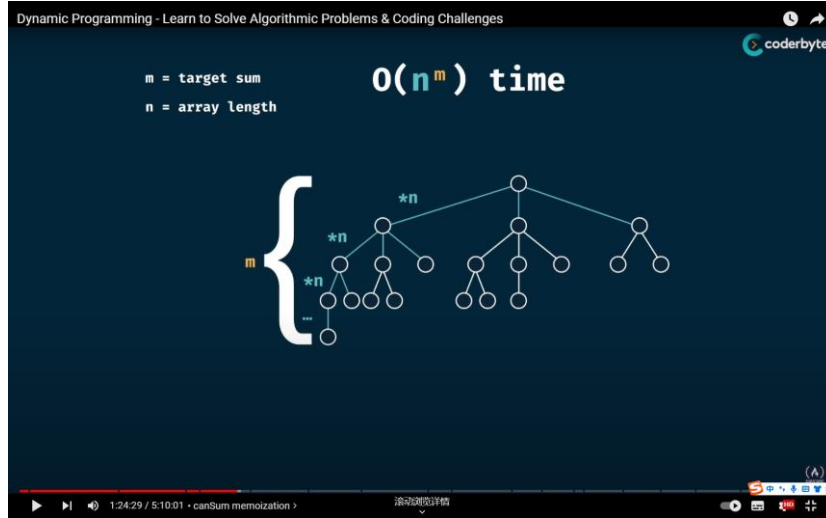


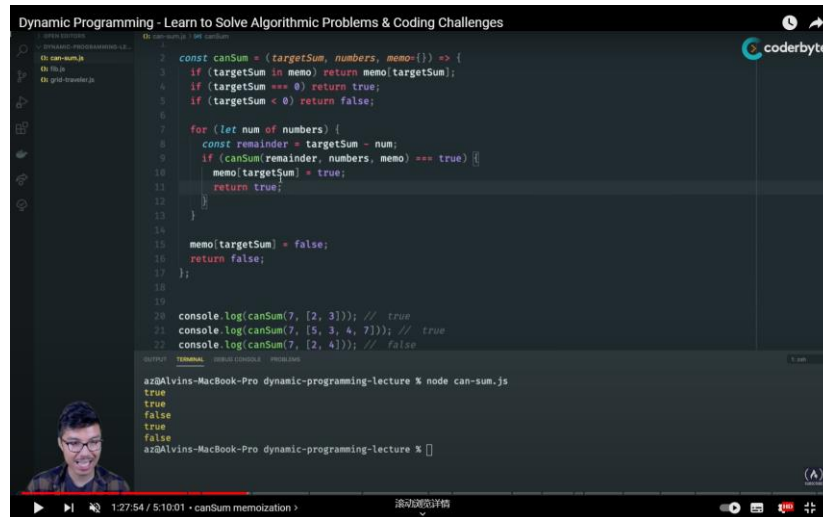
Fig. 12 Recursive design of Can-Sum issue

Just as the video mentioned, the recursive algorithm for case  $m = 300$ ,  $S = [7,14]$  will take a long time to finish. Then it is time for us to include the memory-based structure, our notebook (or I would usually call it the result dictionary).

Accordingly, we have the DP-based design in Fig. 13. Regarding the time complexity of the program,



because the recursive calculation will not stop until we hit 0, then there should be at most  $m$  target numbers we need to check. For each possible target number, the result is also determined by its  $n$  child scenarios (in specific, the results of  $m - S(1)$ ,  $m - S(2), \dots, m - S(n)$ ). Hence, there should be about  $n * m$  possible scenarios we need to check. Then the time complexity should be  $O(n * m)$ . The space complexity remains to be  $O(m)$  because we only need to store the result of  $m$  scenarios at one time to ensure we can have the correct result.



```
Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges
1  const canSum = (targetSum, numbers, memo={}) => {
2    if (targetSum in memo) return memo[targetSum];
3    if (targetSum === 0) return true;
4    if (targetSum < 0) return false;
5
6
7    for (let num of numbers) {
8      const remainder = targetSum - num;
9      if (canSum(remainder, numbers, memo) === true) {
10       memo[targetSum] = true;
11       return true;
12     }
13   }
14
15   memo[targetSum] = false;
16   return false;
17 };
18
19 console.log(canSum(7, [2, 3])); // true
20 console.log(canSum(7, [5, 3, 4, 7])); // true
21 console.log(canSum(7, [2, 4])); // false
22
23 az@Alvins-MacBook-Pro dynamic-programming-lecture % node can-sum.js
true
true
false
true
false
az@Alvins-MacBook-Pro dynamic-programming-lecture %
```

Fig. 13 The DP-based algorithm design for Can-Sum

---

### Section V How-Sum Problem

Then we move on to the next problem, the How-Sum problem. This problem is still based on a target number  $m$  and a possible selection array  $S$ . If we can obtain the target number with the given array, we need to return an arbitrary list that contains a feasible combination. Otherwise, we return None/False.

The overall logic stays the same, but we need to alter what's returned from the basic cases. As mentioned, we have the following three basic cases:

1. When the target number is 0, the current combination works, meaning that we need to record the choices we made. Hence, we return an empty list `[]` to the above layer to start recording.
2. When the target number is negative, the current combination is not valid. Hence, we return `None/False` (remember to distinguish the difference between `None` and `[]`) to the above layer.
3. When the target number is a valid key in the dictionary (memo) we have, then return the recorded corresponding value to the above layer.

Then we have the new circumstances again. Similarly, we will use a For loop to examine the all the possible circumstances. In each subbranch, suppose the element we choose from  $S$  is  $S(1)$ , then we need to use the designed function to obtain the result for  $m - S(1)$ . If the returned value is an array, then we break the loop and add the current choice  $S(1)$  to the array and return to the previous layer. Otherwise, we will wait until we run out of options and return `None/False` to indicate that this node cannot return a valid answer.

According to Fig. 12, there will be at most  $n^m$  nodes in total. However, we may need to copy the array returned by the previous layer to act as the returned value, and the array can be at most  $m$ -unit long (suppose that we have an array with ones). Hence, the time complexity is  $O(n^m m)$ .

According to the algorithm tree, each sub-path contains at most  $(m + 1)$  nodes. If we start returning an empty array from the  $(m + 1)$ th node to start recording, then the array will contain 1 element when we are at the  $m$ th node. If we move up another layer, the array will contain 2 elements at the  $m - 1$ th node. Following this order, the array contains  $m$  elements when we are at the first node. Hence, the space complexity is  $O(m)$ .

By using the dictionary function to record the calculated scenarios, we can reduce the time complexity to  $O(nm^2)$  because there could be at most  $m$  target numbers, each target number has at most  $n$  sub-scenarios and in each scenario, we need to copy an array that is at most  $m$ -unit long.

Meanwhile, because there will be at most  $m$  target numbers and the recorded returned result of each scenario is at most  $m$ -unit long, the space complexity is  $O(m^2)$ .

---

## Section VI Best-Sum Problem

Then we upgrade the problem into Best-Sum, and the final goal is to obtain the shortest combination that can help us obtain the target number. The basic algorithm design stays the same, while we need to add a comparison-based structure to help us obtain the optimal answer. Accordingly, we keep the following basic scenarios from the previous section:

1. When the target number is 0, the current combination works, meaning that we need to record the choices we made. Hence, we return an empty list `[]` to the above layer to start recording.
2. When the target number is negative, the current combination is not valid. Hence, we return `None/False` (remember to distinguish the difference between `None` and `[]`) to the above layer.
3. When the target number is a valid key in the dictionary (memo) we have, then return the recorded corresponding value to the above layer.

While we are looping through all the possible choices in the possible selection array  $S$ , we won't break the loop until all choices have been examined. If the returned value from one of the sub-scenarios is not `None/False`, then we choose:

1. Save the returned value as the current optimal choice when we don't have a feasible answer before.
2. If we had an optimal choice in our hand, then compare the length of the two choices and only save the shorter one.

The time complexity remains to be  $O(n^m m)$  for the recursive design while the space complexity changes to  $O(m^2)$  because we have at most  $m$  scenarios where we need to save the optimal combination and the combination is at most  $m$ -unit long.

Regarding the DP-based design, both the time complexity and the space complexity are the same to the ones in How-Sum problem.

---

## Section VII Can-Construct Problem

Now we move on to the Can-Construct problem. There are two inputs for this question, a target word “target” and a “wordBank” array that contains all possible choices for us to construct “target”. Our desired output is a Boolean variable, whether “True” or “False”. This question is quite similar to the Can-Sum problem because it is a **Decision Problem**.

Different from the Can-Sum problem, we are now facing an issue correlated with strings instead of integers now. Hence, we need to clarify the conditions that determines if one substring from the wordBank is a valid candidate that can form the “target”.

Suppose we have the “target” as “Potato”, and the “wordBank” is [“Poto”, “ta”, “tato”, “P”]. Then we can see that “Poto” is not a valid candidate. Then some might say that the second substring “ta” is a valid candidate because we have “ta” in “Potato”. However, it’s actually not. If we did treat “ta” as a valid candidate, then we certainly need to get rid of it to make a new “target” for the next round of checking. Then the new “target” would be “Poto” and it perfectly matches the first element in “wordBank”, which indicates that we would have “True” as our final result.

However, according to the information we have, the answer should be “False” because we cannot arbitrarily break down the basic elements in “wordBank” to construct the “target”. Hence, while checking if a substring is a valid candidate, we need to see if the substring can perfectly match **a part of the “target” from index “0”**. Then the basic structure of the Can-Construct problem is given as

1. If “target” is an empty string, then return “True”.
2. Else if “wordBank” is an empty list, return “False”.
3. Else if the result of the current “target” already exists in the dictionary, return the stored values.
4. Else, we start to loop from all the possible choices in “wordBank” to see if we can have a perfect zero-indexed match (the initial letter of the current choice should be the same as the one of “target”).  
If so, we record the result, update the target word and see if the new target word can be constructed.  
If no match is found after completing the loop, we record it in the dictionary and return False.

Now let’s consider the complexity of the algorithm. First, we start with the recursive (memory-less) design. Suppose the “target” is a string with  $m$  letters, then the algorithm tree will be at most  $m$ -level deep (in the worst-case scenario where we only take away one letter each level). If the “wordBank” contains  $n$  feasible choices, then there will be at most  $n$  subbranches in each level. Hence, there are  $n^m$  nodes in total. While we are at each node, we also need to loop from the beginning of the current target to see if our choice is a match and then cut our target shorter for the next iteration. This means that we will have at most  $m$  calculations to make on the target string. Therefore, the time complexity of the recursive design should be  $O(n^m m)$ . Accordingly, the space complexity should be  $O(m^2)$  because we only need to store the results of  $m$  different scenarios at one time step (again, consider the longest path we can have) and the longest content to save has the length of  $m$  units (see Fig. 14).

For the DP-based structure, we shrink the time complexity to  $O(nm^2)$  because there will be at most  $m$  different keys in the dictionary. For example, if we have “Potato” as the target, then there will be at most 6 scenarios: “Potato”, “otato”, “tato”, “ato”, “to” and “o”. For each key, we need to check through all the feasible selections in “wordBank” to see if there’s a result. And in each sub-scenario, we will loop from

the beginning of the target to the end. And the space complexity remains to be  $O(m^2)$ .

The screenshot shows a video player interface for a video titled "canConstruct". At the top, it says "Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges" and "codobyte". Below this, the title "canConstruct" is displayed, followed by the code: `m = target.length` and `n = wordBank.length`. The main content compares two approaches: "brute force" and "memoized". For "brute force", the time complexity is  $O(n^m * m)$  and the space complexity is  $O(m^2)$ . For "memoized", the time complexity is  $O(n * m^2)$  and the space complexity is  $O(m^2)$ . An arrow points from the brute force side to the memoized side. Below this, a subtitle reads: "class. So overall, we definitely prefer this second solution, because we remove some exponential,". The video player controls at the bottom show a progress bar at 2:38:23 / 5:10:01 and the title "canConstruct memoization".

<u>brute force</u>		<u>memoized</u>
$O(n^m * m)$ time	→	$O(n * m^2)$ time
$O(m^2)$ space		$O(m^2)$ space

class. So overall, we definitely prefer this  
second solution, because we remove some exponential,

Fig. 14 Can-Construct Problem

If we want to solve “Sum-Construct” problem, which is to calculate the total amount of ways to obtain the target string, we will use the same structure. The complexity analysis stays the same.

---

## Section VII All-Construct Problem

Then we pick a more complex scenario, where we need to collect all the possible choices in an array. In specific, we need to return an array of array(s). For example, if we have “target=Potato” and “wordBank=[“Po”, “to”, “ta”, “tato”]”, then we need to return “[[“Po”, “ta”, “to”]; [“Po”, “tato”]]”.

Besides, there are some special scenarios:

1. If “target=[]”, then we return “[[]]”, which is a 2-dimensional array. (You can also imagine this as a  $1 \times 0$  matrix)
2. If the “wordBank” cannot help us get “target”, then we return “[]”, which is an empty array.

Apart from the above two basic scenarios, we also need to settle the case where we need to specific how can we make a new target word from the “wordBank”. Firstly, we need to loop through all the elements in “wordBank” to see if the current target word starts with anyone of them.

If we have an element “A” that satisfies the above condition, we need to dig deeper to see if the rest of the strings can be constructed by the “wordBank”. If yes, we need to add “A” in the beginning of every returned answer and hardcopy the list as the corresponding value in the dictionary. Besides, we also need to be ready for the case where we need to fill some new answers into one existed key in the dictionary, which means we need to fill in all the sub-lists after the original value (of lists).

## Section VIII Tabulation

Memoisation is not the only way to carry out the DP-based process. And now, let's move on to "Tabulation". Basically, "Tabulation" is a method that projects our issue into the shape of a list (or an array). Take the Fibonacci question mentioned in Section I as an example, if we want to obtain the  $n$ th number in the Fibonacci sequence, then we have  $n + 1$  scenarios to consider  $(0, 1, 2, \dots n)$ .

Instead of saving the value of each element in a dictionary, we can instead use a list/an array, which is illustrated in the figure below:

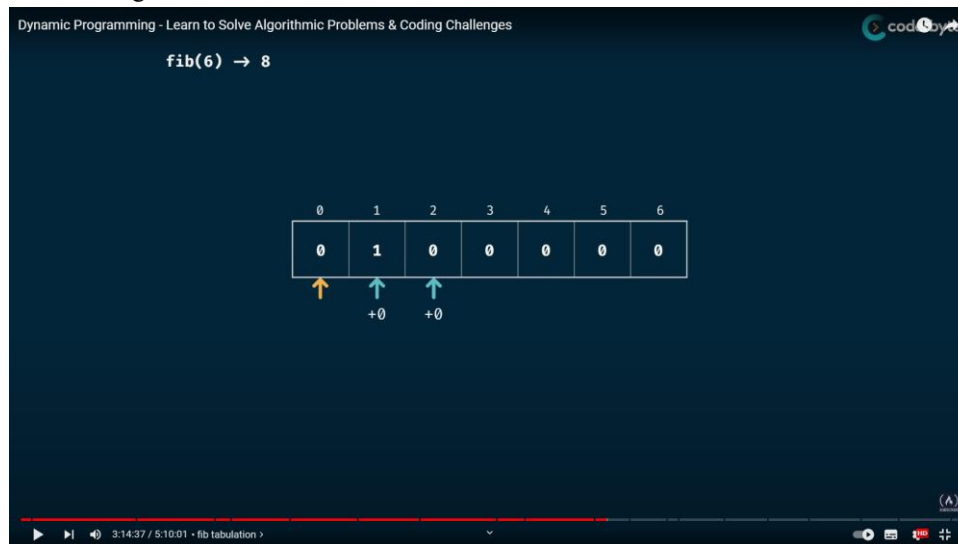


Fig. 15 The tabulation approach for Fibonacci sequence

After settling the basic scenarios, we can then use a For loop to calculate the remaining values. Regarding the time complexity and space complexity, they remain to be  $O(n)$  simultaneously, which is quite straight forward as shown in Fig. 15.

For the Grid Traveller issue, the concept of using tabulation is more straightforward. Different from the Video, I think constructing an empty list with the dimension of  $\text{row}_{\text{num}} \times \text{col}_{\text{num}}$  is more than enough because the cases where we have 0 rows or 0 columns should have a return value of "0".

Then the element  $\text{List}[i][j]$  we construct should represent the result when we  $(i + 1)$  rows and the  $(j + 1)$  columns. When either  $i$  or  $j$  is 0, the returned value is 1. Otherwise, we have

$$\text{List}[i][j] = \text{List}[i - 1][j] + \text{List}[i][j - 1]$$

Accordingly, the basic steps for the Tabulation approach are

1. Visualise the problem as a table.
2. Initialise the list with a proper size according to the input/basic parameters.
3. Settle the basic cases (elements in the list)
4. Find a proper logic to iterate through the table and calculate non-basic elements with currently available elements

#### Section IV Can-Sum Tabulation

In terms of the Can-Sum problem, it is easy to understand that we need to construct an  $(m + 1)$ -dimensional list, where  $m$  is the value of the target number. Because the expected answer is True/False, we can initialise all the elements as “False”.

Personally, I think the biggest difference between the Memoisation-based approach and the Tabulation-based approach is the way of thinking. In Memoisation-based approach, we usually follow the forward-thinking pattern. Take the Can-Sum scenario for an example, if a target number  $m$  is given, then whether  $m$  can be formed by numbers  $n = [n_1, n_2]$  is determined by the OR operation of the following two results:

1. Can  $m - n_1$  be constructed by  $n$ ?
2. Can  $m - n_2$  be constructed by  $n$ ?

We will keep iterating until the new target number is 0 or negative, which are our basic cases. The basic cases will have a solid “True” or “False” to return. Then the algorithm will bring these solid answers up to help determine our final answer.

But the Tabulation-based approach is a lot different and works in a reverse-thinking fashion. In Tabulation, we work from 0 and see what we can obtain with  $n = [n_1, n_2]$ . In other words, we first settle that the answer is “True” when  $m = n_1$  or  $m = n_2$ . Suppose we have  $n_1 < n_2$ , we then jump to  $m = n_1$  and turn the result of  $m = n_1 + n_1$  and  $m = n_1 + n_2$  to be “True”. Afterwards, we move on to  $m = n_2$  and turn the result of  $m = n_2 + n_1$  and  $m = n_2 + n_2$  to be “True”, etc. This process can be illustrated as the following figure:

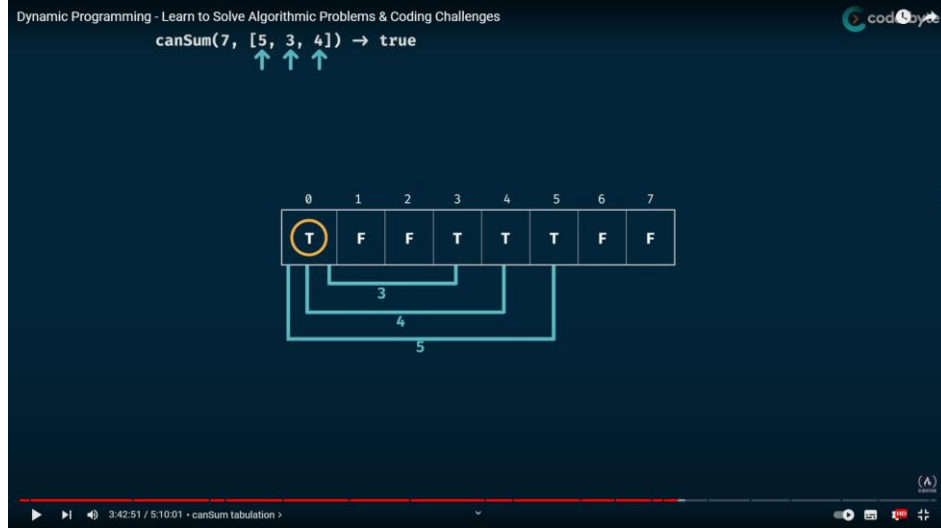


Fig. 16 The Tabulation-based process of Can-Sum problem

Then we need to settle the basic scenario. From the previous discussion, it is clear that 0 is the basic scenario. Hence, we always have  $List[0] = True$ . And the rest is we start with  $List[0]$  and loop through all the elements in the list. Because the length of the list is  $(m + 1)$ , the space complexity of the above Tabulation-based design is  $O(m)$ . For each element, we need to loop through all elements ( $n$  in total) in “numbers”. Hence, the time complexity is  $O(mn)$ . While the space complexity is  $O(m)$ .



## Section V How-Sum Tabulation and Best-Sum Tabulation

Next, we swing back to the How-Sum problem. As required, we need to return any feasible combination that can let us get the target number, meaning that the returned values should be a list.

Then we need to settle the basic scenario, which is when the target is 0, the returned value should be an empty list “[ ]”. As always, we need to loop through all the elements in numbers. Instead of only giving True or False, we now need to include some contents in the array. For example, when we start with  $m = 0$  and  $n_1 = 5$  as the following figure, we need to include  $n_1$  into the array to have [5] for element List[5]. Similarly, when we start with  $m = 3$  and  $n_1 = 5$ , we need to have List[8] = [3, 5].

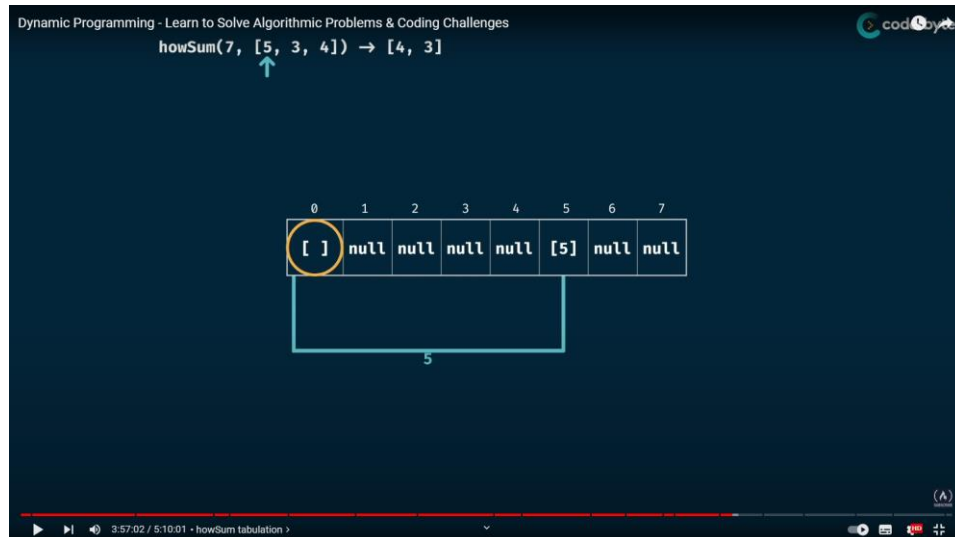


Fig. 17 How-Sum Tabulation

The time complexity is  $O(m^2n)$ . Apart from the  $mn$  times of iteration involved in the algorithm, each iteration may contain the process of copying a previous returned list (because there are multiple lists involved here, we will use “returned list” to represent lists such as List[8] = [3,5]), while the list contains at most  $m$  elements. Because each slot of the list may contain a non-empty returned list and the returned list can be at most  $m$ -element long (when the returned list only contains a bunch of ones), the space complexity is  $O(m^2)$ .

For the Best-Sum one, we only need to add an extra logic:

1. If this place is empty (null/none) when we try to fill in, then directly fill the new list in.
2. If this place is taken, then we compare the length of the existed content and the new content and choose the shorter one.

The complexities remain the same as the How-Sum design.

## Section VI Construct Tabulation

This section includes the notes for the rest of the three problems, Can-Construct, Count-Construct and All-Construct. What's different is that we need to consider strings instead of numbers.

Similarly, the size of the Tabulation list is correlated with our inputs. Take the Can-Construct problem as an example, we can have the following list when target is "abcdef" and wordBank is ["ab", "abc", "d", "de", "f"]:

""	"a"	"ab"	"abc"	"abcd"	"abcde"	"abcdef"
True	False	False	False	False	False	False

And the exact reasons are:

1. Because we need to construct the target word in a certain order (in other words, we can't have "abcd" when we only have "ac" and "db" in our hands), we can break the target string into seven possible scenarios.
2. Empty string is a basic scenario where we always return "True" because we don't need anything to construct an empty string.

Again, we need to loop through every choice in the wordBank for each scenario to see what we can actually obtain with the wordBank. After completing the loop in `List[0] = ""`, we have

""	"a"	"ab"	"abc"	"abcd"	"abcde"	"abcdef"
True	False	True	True	False	False	False

Then we switch to `List[1] = "a"`. From the previous result, we observed that it is not possible to obtain "a" with the given wordBank, so we skip all the looping process because it will not affect the result list. Then for `List[2] = "ab"`, because nothing in wordBank starts with "c", the table will not change. For `List[3] = "abc"`, we will change the result list to the following one because `"abc" + "d" = "abcd"` and `"abc" + "de" = "abcde"`:

""	"a"	"ab"	"abc"	"abcd"	"abcde"	"abcdef"
True	False	True	True	True	True	False

After doing the same process for every element in the list, we have

""	"a"	"ab"	"abc"	"abcd"	"abcde"	"abcdef"
True	False	True	True	True	True	True

Once we understand this designing concept for strings, the algorithm designs for some similar topics can be carried out with the same structure, while the result values vary. For Count-Construct, the expected outcome is the number of possible ways. Hence, we initialise the list as

""	"a"	"ab"	"abc"	"abcd"	"abcde"	"abcdef"
1	0	0	0	0	0	0

While looping, suppose we are looping under the basic condition of `list[1]` and the current choice from wordBank is `j` (`j` is a string) and we have `list[1] + j = list[1 + len(j)]`, then we have `list[1 + len(j)] += list[1]`.

---

For All-Construct, the expected outcome is an array of arrays, then we need to initialise the list as

“”	“a”	“ab”	“abc”	“abcd”	“abcde”	“abcdef”
[[] ]	None	None	None	None	None	None

When we have  $\text{list}[1] + j = \text{list}[1 + \text{len}(j)]$ , we need to first have a temporary variable `temp` that satisfies  $\text{temp}[k] = \text{list}[1][k] + j$ , where  $\text{temp}[k]$  is the  $k$ th element of `temp`. And then add `temp` to  $\text{list}[1 + \text{len}(j)]$ .

Although the DP-based process can help us save more time and space, it might still crash while facing complex issues like All-Sum and All-Construct because the problems demand us to return **all** the answers, which is equivalent to checking all possible scenarios, leading to exponential time complexity.

---

## **Section VII Conclusions**

Some tips for the design of DP algorithms:

1. Draw a basic algorithm tree first (be sure that your example covers multiple scenarios)
2. Think what tasks/processes are overlapped when we implement the recursive design
3. Settle the basic scenarios (those that directly result in an input)
4. Think recursively for Memoisation and iteratively for Tabulation

And the course is finished.