Chapter 11

Dynamic Programming

It is time to tie up some loose ends when it comes to search and optimization problems. So far, we have seen problems where we could formulate recursions (Chapter 7), where we needed to exhaustively try all recursive choices to solve them (Chapter 9) and just now problems where we just guessed (and hopefully proved) what choice was correct (Chapter 10).

Sometimes, we mentioned in passing that recursions could be solved in linear time. In other problems we had to resort to the exponential complexity of backtracking. This chapter explains when the specific approach we have used to avoid backtracking (without resorting to greedy algorithms) work. It's one of the most important paradigms in algorithmic problem solving – *dynamic programming*.

We begin with a familiar example, the change-making problem with a different set of denominations, and follow up with a *lot* of standard problems and techniques.

11.1 Making Change Revisited

In the previous chapter, we solved the change-making problem with denominations 1, 2, and 5 greedily. The best choice of coin for a given T was always the greatest one that didn't exceed T. After getting your hopes up, Exercise 10.2 asked you to prove that the case with coins worth 1, 6 and 7 could not be solved in the same greedy fashion.

So, how is this variant solved? Well, you already know how to solve the 1, 2, 5 case non-greedily in linear time. Formulate the recursion

$$\text{Change}(T) = 1 + \min \begin{cases} \text{Change}(T-1) & \text{if } T \geq 1 \\ \text{Change}(T-6) & \text{if } T \geq 6 \\ \text{Change}(T-7) & \text{if } T \geq 7 \end{cases} \tag{11.1}$$

with base case $\text{Change}(0) = 0$ and solve it iteratively in the way that we showed in Chapter 7. The code was pretty simple, storing the answers in an array as we go along:

```
1: procedure ChangeMaking(integer T)
2:     answers ← new array of size T + 1
3:     ▷ base case
4:     answers[0] ← 0
```

```
5:    for i = 1 → T do
6:        answers[i] ← 1 + answers[i − 1]
7:        if i ≥ 6 then
8:            answers[i] ← min(answers[i], 1 + answers[i − 6])
9:        if i ≥ 7 then
10:           answers[i] ← min(answers[i], 1 + answers[i − 7])
11:   return answers[T]
```

Problem 11.1.

The Gourmet gourmeten *(all subtasks)*

What makes the iterative computation able to speed up this particular recursion from exponential to linear, but we can't do the same for e.g. the max clique problem? The theoretical answer is that recursions allow a space-time trade-off where we can reduce the time complexity to *the number of subproblems* we recursively solve, multiplied by the time it takes to solve a single subproblem. For change-making, we need to solve up to $T + 1$ subproblems in total, and each subproblem is a constant amount of work. In the backtracking problems, there were simply an exponential number of subproblems. For max clique, we had to go through all vertex subsets to see if they were cliques. To count the number of subproblems, look for how many different values the parameters to the recursive function can take. The space trade-off is that we must save the answers for each subproblem.

This property distinguishing the recursions that can be sped up using the iterative method from those that can't is in the theory called ***overlapping subproblems***. It is a fancy term that means "during backtracking, you call your recursive procedure with the same parameters many times". Generating subsets does not revisit the same subproblem, so the technique does not help. Change-making revisits the same subproblem an exponential number of times, so the technique gives us an exponential speedup.

Exercise 11.2. For the following recursive functions, determine whether their subproblems overlap (i.e. will be called multiple times with the same arguments).

```
1: procedure Fibonacci(integer i)
2:     if i < 2 then
3:         return i
4:     return Fibonacci(i − 1) + Fibonacci(i − 2)
```

```
2: procedure DFS(vertex v, array seen)
2:     seen[v] ← true
3:     for all neighbors u to v do
4:         if not seen[u] then
5:             DFS(u)
```

```
3: procedure Power(real b, integer e)
```

```
2:    if e = 1 then
3:        return b
4:    returnPower(b, ⌊e/2⌋) · Power(b, ⌈e/2⌉)
```

This is essentially all that ***dynamic programming***, or *DP* is. You take a problem that admits a self-recursive solution (i.e. it can be solved by reducing it to smaller instances of itself) and compute each subproblem exactly once. If a subproblem is indirectly used many times during a recursion, you win time.

As an intuitive mental model, this explanation is not very good. It doesn't tell you when you should go for a recursive solution because you can apply dynamic programming, or what recursion you should choose to make it work. The remainder of the chapter describes a range of ways in which you can think about recursion to find DP solutions.

11.2 Paths in a DAG

The standard problem for dynamic programming problems is about paths in a directed, acyclic graphlongest directed acyclic graph, a DAG. In Section 8.6 we learned how to topologically order the vertices of a DAG, so that every edge points from a later vertex to an earlier vertex. While this problem is important in its own right, it's also valuable to study in order to understand DP. As we soon see, the topological ordering carries importance for DP.

Longest Path in a DAG – longestpathinadag

Given a directed, acyclic graph, find a longest path in it.

Solution. This problem is best approached with the framework for recursive choices that we developed in Chapter 7. Consider the longest path $p_1 \to p_2 \to \cdots \to p_l$ in the graph. The path must start at a vertex, in this case p_1. Now channel your inner Tasha the kitty (page 99) and ask: what is the first edge on this longest path? This can be any one of those that point out from p_1. We should pick the edge $p_1 \to v$ such that the longest path starting at v is as long as possible. Any path that starts at v can be extended with one more edge $p_1 \to v$, so the longest path starting at v plus this edge is the longest path starting at p_1. This gives us the recursion we are after:

```
1: procedure LongestPath(vertex v)
2:     length ← 0
3:     for each out-edge v → u do
4:         length = max(length, LongestPath(u) + 1)
5:     return length
```

The length of the overall longest path is then $\max_v \text{LongestPath}(v)$.

To avoid computing LongestPath(v) for the same v an exponential number of times, we use the iteration trick of computing it for each v in…increasing order? Hmm, this recursion is clearly more complex than the previous ones. In the recursions seen so far we could find a simple ordering of the parameters to ensure that the recursive subproblems had already been computed when we reached a subproblem.

We somehow need to find an ordering of the vertices such that whenever there is an edge $v \to u$, the answer for u must have been before that of v. A-ha! That's exactly what a topological ordering guarantees us. We thus need to first find this ordering before we can iteratively compute the recursion:

```
1: procedure IterativeLongestPath(vertices V)
2:     answers ← new array of size V.size
3:     order ← TopologicalOrder(V)
4:     for v in order do
5:         for each out-edge v → u do
6:             answers[v] = max(answers[v], answers[u] + 1)
7:     return the maximum of length
```

The above pseudo code is $\Theta(V + E)$ for V vertices and E edges – the innermost loop does constant-time work and loops once over every single edge.

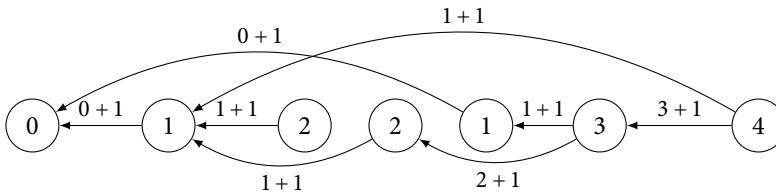

Figure 11.1: The length of the longest path starting at each vertex and how it is computed.

We have cheated a bit so far. The problem asked for *the longest path*, but we have so far only computed its length. The reconstruction works just like the BFS reconstruction for shortest path does. For each v, make sure to also store the edge $v \to u$ where u had the longest path in another vector *next*. Given this, the reconstruction is easy:

```
1: procedure LongestPathReconstruction(answers, next)
2:     i ← the index where answers is greatest
3:     while answers[i] ≠ 0 do
4:         add i → next[i] to the path
5:         i = next[i]
```

□

Exercise 11.3. The path can be reconstructed using only *answers* and the input graph. How?

Memoization and Memory

Do we really need to find a topological ordering to recurse on the paths of a DAG? Depending on how experienced of a programmer you are, the answer *no* might be surprising. The recursive function can be adapted slightly to avoid the exponential blow-up using ***memoization*** or ***top-down*** dynamic programming (in contrast to ***bottom-up***, which the iterative computation is called). The idea is that since the recursive function returns the same value every time it's called, we can store the return value after the first call and return it immediately for subsequent call. In software engineering, storing the results of computations for later reuse is called *caching*. When applied directly to the return value of a function in this manner it's called memoization. Memoizing the function for the longest path in a DAG looks like this:

```
1: memo ← new array of size V filled with −1
2: procedure LongestPath(v)
3:     if memo[v] ≠ −1 then
4:         return memo[v]
5:     length ← 0
6:     for each out-edge v → u do
7:         length = max(length, LongestPath(u) + 1)
8:     return memo[v] = length
```

The return values for the function is stored in *memo*, which is initially filled with a sentinel value −1 that allows us to distinguish between values for which the function has been computed and those not yet called. When the answer is always non-negative, −1 is the standard choice.

The benefit of memoization is twofold. First, we don't have to find the topological ordering of the function calls in the recursion. As we just saw, this ordering can need an explicit topological sorting to construct. Secondly, with memoization the recursion only calls the subproblems that can affect the answer we are interested in rather than all possible ones. In some problems, we can win constant factors due to this. In extreme cases an entire parameter can turn out to be unnecessary, such as in the Ferry Loading problem in Section 11.3. When that happens, you normally need to realize it before coding though, in order to reduce the number of states to something that fits in memory.

Problems on DAG paths are not uncommon in contests, and thanks to memoization become easy to code. Typically, the solution – in fact, the solutions for most memoized DP problems – looks very similar to the one we just showed you.

Problem 11.4.

BAAS	baas
Safe Passage	safepassage

Memoization is not without downsides. There are a lot of function calls in recursive

solutions, and they carry some overhead. This problem is not as bad in C++ as in many other languages, but it is still noticeable. When the number of states in your DP solution is running a bit high, you might want to consider coding it iteratively. The performance gain from avoiding recursion is normally greater than the benefit of not computing a few unnecessary subproblems.

In top-down DP, the memory usage is for the most part clear and unavoidable. If a DP has N states, the top-down solution uses $\Omega(N)$ memory to store all the states. For a bottom-up solution, the situation is quite different. If we choose the order in which the subproblems are computed well, we seldom need to store the answers to all of them all the time. Consider e.g. the change-making problem again, which had the following recursion:

$$\text{Change}(T) = 1 + \min \begin{cases} \text{Change}(T-1) & \text{if } T \geq 1 \\ \text{Change}(T-6) & \text{if } T \geq 6 \\ \text{Change}(T-7) & \text{if } T \geq 7 \end{cases} \tag{11.2}$$

If we compute $\text{Change}(T)$ for values of T in the order $0, 1, 2, 3, \ldots$, once the answer for $\text{Change}(k)$ has been computed, the answers for $k-7, k-8, \ldots$ are never used again. During the entire process, only the answers to the 7 last subproblems are needed. This $\Theta(1)$ memory usage is pretty neat compared to the $\Theta(K)$ usage needed to compute $\text{Change}(K)$ otherwise. In the worst case, such as when recursion is over a general directed graph, this doesn't help.

Competitive Tip

Generally, memory limits are very generous nowadays, somewhat diminishing the art of optimizing memory in DP solutions. It can still be a good exercise to think about improving the memory complexity of the solutions we look at, for the few cases where these limits are still relevant.

11.3 Standard Techniques

By now, you might have realized that a lot of what we discussed in these 3 last chapters have very much in common with the mental model of recursion from Chapter 7. The reason for this is very simple – that chapter was written to be the common ground for the recursive search techniques to make them easier to talk about. Almost all the recursive problems described in that chapter are susceptible to dynamic programming to get a polynomial-time solution. In this section we look at some more complex recursions, but the basic principles remain the same.

More Choices

We start with a variation of the classical *stock trading* DP problem.

Short Sell – shortsell

LiU Coding Challenge 2018

Simone is trading the cryptocurrency CryptoKattis, used to improve your Kattis ranklist score without solving problems. She is very perceptive of online judge trends and noticed that many coders are switching to the new online judge Doggo. She thinks that Kattis is falling out of fashion which will cause CryptoKattis to decline in value. Careful market research allows her to estimate the prices of a CryptoKattis in dollars during the next $N \leq 100\,000$ days denoted $P_1, \ldots, P_N$. She intends to use this data to perform a *short sell*. One day she will borrow 100 CryptoKattis from a bank and sell it for dollars. At a later day, she will purchase 100 CryptoKattis and repay her loan to the bank. Every day between and including these two days, she must pay K dollars in interest. What's the maximum profit she can make by choosing these two days optimally?

Solution. On a given day, Simone has three choices: lend the CryptoKattis and sell them, do nothing, or buy CryptoKattis to repay her loan. The correct question for a recursive solution is: if she only performs the first two actions, what is the most cash $C(i)$ she can have on hand at the end of the i'th day? The profit she can make by selling on day i would then be $C(i) - 100P_i$, and to solve the problem we'd take the maximum over all i. There are three cases when computing $C(i)$. If she has not bought CryptoKattis yet, the answer is 0. If she has already bought she must pay interest today, so she loses K money from the previous day's maximum cash, i.e. $C(i-1) - K$. This can also be the day she does the short sell, giving her $100P_i - K$ money. Together, these cases gives us the recursion

$$\mathrm{C}(i) = \max \begin{cases} 0 \\ 100P_i - K \\ C(i-1) - K \end{cases} \tag{11.3}$$

DP solutions for recursions in only a single parameter are predominantly written iteratively. This DP only depends on the answer for $i-1$, so we don't need an array to store past results.

```
1: procedure ShortSell(N, P, K)
2:     C ← 0
3:     profit ← 0
4:     for each price p in P do
5:         C ← max(0, 100p − K, C − K)
6:         profit ← max(profit, C − 100p)
7:     return profit
```

□

Exercise 11.5. Adapt the solution to Short Sell to the case where Simone is allowed to perform several short sells, as long as they don't overlap.

Problem 11.6.

The Stock Market borsen *(all subtasks)*

Radio Commercials	commercials	
Going to School	skolvagen	*(all subtasks)*

The following problem was one of the examples we gave on incorrect greedy algorithms. We return to it now, as perhaps our first "real" dynamic programming problem that recurses on more than one parameter.

Bookshelves – bokhyllor

By Arash Rouhani. Swedish Olympiad in Informatics 2012, School Qualifiers.

You are buying bookshelves to fit all your books. Books come in three sizes; a small book has width 1, a medium book width 2 and a large book width 3. Each bookshelf has the same width $L \leq 20$. Given the amount of books you have (at most 20 of each size), compute the number of bookshelves needed if the books are placed optimally on the shelves.

Solution. The first step in solving dynamic programming problems is to reformulate it in the recursive "sequence of choices" form we are used to. In this problem, no choices are given to us – we must define them ourselves.

How should you think when formulating recursive choices that are supposed to work well with dynamic programming? The core idea is that after you recursively try some number of choices, you arrive at a situation where the important thing is **not what choices you made**, only their consequences. For example, in the change-making problem, when solving a recursive subproblem T' it's completely irrelevant which coins we picked to get there. The optimal solution for the remaining T' money is the same no matter if we arrived there after recursively using 1000 coins of value 5 or 5000 coins of value 1. Not all problems can be formulated in this way. For example, say that you try to find the longest path starting at some vertex in a general graph using recursion. Is it enough to formulate a recursive function LongestPath(v) that recursively tries what vertex next to visit? No! The choices that the recursive function can make depends on previous choices, so this does not work.

In this problem, we propose the following construction method. Fill each bookshelf that we buy one at a time. At any given time, we have four choices. We might either place a book on the current shelf (one choice per book type) or we can buy a new bookshelf. During this construction, we need to keep track of only four things: how many books we have yet to place of each of the three types and how much space we have left on the current shelf. This results in the following recursion:

```
1: procedure Bookshelves(s, m, l, widthLeft)
2:     if s = m = l = 0 then
3:         return 0                          ▷ base case – we have no more books to place
4:     answer ← ∞
5:     if s > 0 and widthLeft ≥ 1 then
6:         answer ← min(answer, Bookshelves(s − 1, m, l, widthLeft − 1))
```

```
7:  if m > 0 and widthLeft ≥ 2 then
8:      answer ← min(answer, Bookshelves(s, m − 1, l, widthLeft − 2))
9:  if l > 0 and widthLeft ≥ 3 then
10:     answer ← min(answer, Bookshelves(s, m, l − 1, widthLeft − 3))
11: if widthLeft < L then
12:     answer ← min(answer, Bookshelves(s, m, l, L) + 1)
13: return answer
```

The function is invoked with Bookshelves$(s, m, l, 0)$ since we in the beginning need to buy a first shelf. We have not added the memoization here, and will typically not do so in this chapter to avoid cluttering – coding that is up to you.

Exercise 11.7. Why is the check for *widthLeft* $< L$ needed on line 11?

To find the time complexity of the solution, we compute the number of valid parameter combinations, called the *states* of the recursion. The values s, m, l only decrease and are at least 0, while *widthLeft* is between 0 and L, so there are $O(s \cdot m \cdot l \cdot L)$ states. The function itself takes constant time, so the number of states is also the total time complexity. □

Problem 11.8.

Dance Dance Revolution	dansmatta
Nikola	nikola

Note that the recursive construction method we used can construct *every* possible placement of books on the shelves! When we impose a construction method we must ensure that it covers the optimal possibility too! In difficult problems it's sometimes the choice of recursive construction that makes the solution fast enough, for example by not testing certain states that can never be optimal. Different ways of looking at a problem can give you a variety of recursions and parameters representing your subproblems. This next example demonstrates the importance of choosing what parameters to include with care.

Ferry Loading – lastafarjan

By Oskar Werkelin Ahlin. Swedish Olympiad in Informatics 2013, Online Qualifiers.

A ferry is to be loaded with $N \leq 200$ cars of different lengths waiting to board in a long line. The ferry consists of four lanes, each of the same length $L \leq 60$. When the next car in the line enters the ferry, it picks one of the lanes and parks behind the last car in that line. There must be a safety margin of 1 meter between any two parked cars.

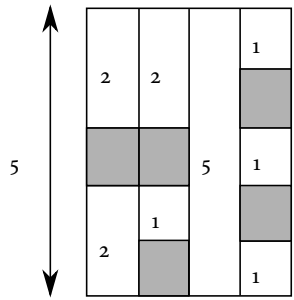

Figure 11.2: An optimal placement on a ferry of length 5 meters of cars with lengths 2, 1, 2, 5, 1, 1, 2, 1, 1, 2 meters. Only the first 8 cars could fit on the ferry.

How many cars can park on the ferry if they choose the lanes optimally?

Solution. As a simplification, increase the initial length of the ferry by 1 to accommodate an imaginary safety margin for the last car in a lane in case it is completely filled, and increment the lengths of each car by 1 so that we don't have to care about the safety margin at all.

The problem is already given in the sequence of choices form that we know how to translate to a recursion. We have an ordered list of cars and each one has 4 choices – one for each lane it could go into. If a car of length m chooses a lane, the remaining length of the chosen lane is reduced by this amount. After the first c cars have parked on the ferry, the only thing that has changed are the remaining lengths of the ferry lanes.

This suggests a DP solution with nL^4 states, each state representing the number of cars so far placed and the lengths of the four lanes:

```
1:  procedure Ferry(car, lanes)
2:      if car = N then
3:          return 0                          ▷ base case – we have no more cars to place
4:      answer ← 0
5:      for i from 0 to 3 do
6:          if lanes[i] ≥ carLengths[car] then
7:              newLanes ← lanes
8:              subtracting carLengths[car] from newLanes[i]
9:              answer ← max(answer, 1 + Ferry(car + 1, newLanes))
10:     return answer
```

Unfortunately, memoizing this procedure would not be sufficient. The number of states is $200 \cdot 60^4 \approx 2.6 \cdot 10^9$, which requires gigabytes of memory.

To improve the solution, we think about what DP actually is. Dynamic programming is all about taking a list of choices we made and compressing it to only the information that can affect future choices – like ignoring what lane each car went into, instead only keeping

track of what cars remain and how much space is left. This removes information that is redundant for the recursion. Our suggested solution still has some lingering redundancy though.

In Figure 11.2 from the problem statement, we have an example assignment of the cars $2, 1, 2, 5, 1, 1, 2, 1$. These must use a total of $3 + 2 + 3 + 6 + 2 + 2 + 3 + 2 = 23$ meters of space on the ferry. Let $U(c)$ be the total length of the first c cars. This function is invertible – two different c always give different $U(c)$. Let u_1, u_2, u_3, u_4 be the total length of how all the cars that have parked in each lane so far, so that $U(c) = u_1 + u_2 + u_3 + u_4$. The four terms on the right are parameters in our memoization together with c. The left one isn't, but it is uniquely determined from c, which is a parameter.

This means that for some fixed values of *lanes*, there's only a single possible value that *car* can have. This means that we don't need to include it as a parameter *in the memoization* – it's still fine to have in the recursion, since we use it. This simplification leaves us with $60^4 \approx 13\,000\,000$ states, well within reason. □

Problem 11.9.
Buying Coke coke

Interval DP

A common DP category is recursing over intervals of a sequence. Within an interval, we often want to perform some action on a element within the interval and recursively solve a problem on the two subintervals we get after splitting the interval at that element. First, we show a problem that contains most of the elements an interval DP problem can have.

Zuma – zuma

By Goran Žužić. Croatian Olympiad in Informatics 2009/2010, round 5

One day Mirko stumbled upon a sequence of $N \leq 100$ colored marbles. He noticed that if he touches K ($K \leq 5$) or more consecutive marbles of the same color he could wish them to magically vanish, although he doesn't have to do that immediately. When marbles vanish, the marbles to the left and right of them will move towards each other and close up the hole formed. Note that Mirko needs to touch marbles to make them vanish – there are no chain reactions.

Fortunately, Mirko brought an inexhaustible supply of marbles from home, so he can insert a marble of any color anywhere in the sequence (even at the beginning or end). Find the smallest number of marbles he must insert into the sequence to make all of the marbles vanish.

Solution. This task is mainly an exercise in finding the correct subproblem and method of constructing a solution. The key idea is to study marble 1 in the sequence and ask what happens to it. We could insert some marbles of the same color right at the start and remove it. If that's not the optimal solution, there must be another marble of the same color later on in the sequence that, after removing all the intermediate marbles, is removed in the same group as the marble 1.

If that marble is the i'th one, we must clear the entire interval of marbles $[2, i-1]$ before 1 and i can touch (as we assumed should happen). Solving the interval $[2, i-1]$ is *independent* of what we do in the rest of the sequence. We don't know what i is the correct one to test, so we loop over all choices and take the best one. This is how most interval DP solutions work: within an interval, split it into two by looping over all possible "breaking points".

After deciding to merge with the i'th marble, we still have to solve the interval $[i, N]$, now with that extra marble 1. Furthermore, there might be some marble $j > i$ that is *also* supposed to be part of the group used to vanish marble 1. Then we must clear the interval $[i+1, j-1]$, and solve the remainder of the interval $[j, N]$, where we now have **two** extra marbles of the same color as j.

Note that there are only three things that varies in our process: the two endpoints of the interval and the number of marbles we accumulate immediately to the left of the interval. It seems like the correct subproblem is: "how many marbles $M(l, r, s)$ must be added to clear the interval $[l, r]$ if we have s marbles of the same color as the l'th immediately to the left of it?" The first step of the recursion is then the expression $\min_{l<i<r} M(l+1, i-1, 0) + M(i, r, s+1)$ where the i'th and l'th marbles must have matching color.

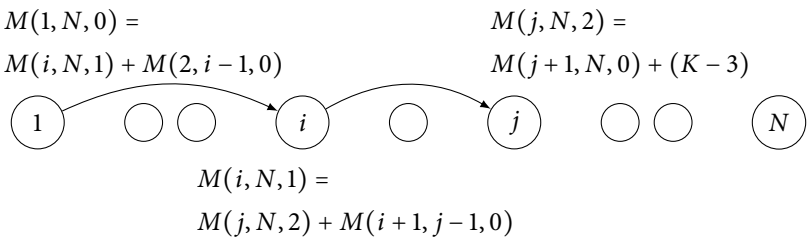

Figure 11.3: A visualization of the Zuma recursion. Marbles 1, i and j are to be merged. The rest of the interval is split up into three independent intervals that are solved recursively.

The other part of this process is vanishing marbles. At some point when solving a subproblem $M(l, r, s)$, the right answer might be that the l'th marble should not be merged with anything in $[l+1, r]$. We must then insert $K-1-s$ extra marbles, to get a group of K. This is the second part: $M(l+1, r, 0) + K - 1 - s$. Note that if $s > K-1$ this would be negative. This means that we never need to increase S beyond $K-1$, so we cap it at that in our recursion.

The DP has N^2K states taking $\Theta(N)$ time, for a complexity of $\Theta(N^3K)$. □

Sometimes we do DP on circular intervals. This is done slightly differently.

Running Routes – runningroutes

By Nathan Mytelka. 2019 ICPC North American Qualifier Contest. CC BY-SA 3. Shortened.

Polygonal School wants to increase enrollment, but they are unsure if their gym can support having more students. The gym floor is a regular n-sided polygon, affectionately called P. The coach has drawn several running paths on the floor. Each path is a straight line segment connecting two distinct vertices of P. During gym class, the coach assigns each student a different running path. The coach does not want students to collide, so each student's path must not intersect any other student's path, even at their endpoints.

Given all the paths, find the size of the largest non-intersecting subset.

Solution. If we solved the problem on a line, there would be little difference between this problem and Zuma. The relevant subproblem would be counting the maximum number of paths $R(l, r)$ within the interval $[l, r]$. For such a subproblem, there are two possibilities. Either l is the first endpoint of a path and some $l < i \leq r$ is the second, whereupon all remaining paths must lie either in $[l+1, i-1]$ or $[i+1, r]$ to avoid crossing the path $l \leftrightarrow i$ – note the two disjoint subproblems that arose – or it is not. In that case, all intervals lie in $[l+1, r]$. The recursion is thus $R(l, r) = \max(R(l+1, r), \max_{l<i\leq r}\{R(l+1, i-1)+R(i+1, r)\}$ where $l \leftrightarrow i$ must be a valid running path.

Solving the problem on a circle is not very different. The vertices of P can still be numbered 1 to n. The primary conceptual difference is that an interval may continue past n back to 1, giving us a right endpoint that is *smaller* than the left endpoint. For example, the interval $[n, 1]$ would represent only the two vertices n and 1, while the interval $[1, n]$ would represent the entire circle. With that in mind, the only change that needs to be made to the recursion is that the interval $l < i \leq r$ should be every vertex from l to r going clockwise. After this change, the answer is computed as the subproblem $R(1, n)$. □

This particular circular interval DP is among the simpler in that it's unusually easy to reduce to the line case. Sometimes that's harder, but the idea of thinking in intervals $[l, r]$ going clockwise from l to r is typically the right way.

Problem 11.10.
Arranging Hat arranginghat

Subset DP

Earlier in the chapter, we mentioned the longest path problem in general graphs. It was clear that we couldn't write a dynamic programming solution to it using only the location of the last vertex added to a path like we could for DAGs. The naive backtracking solution would take something like $\Theta(n!)$ time. There would be n choices for where to start, $n-1$ neighbors for each of them to visit, then $n-2$ choices for the third vertex, etc.

DP can't help us to do better than exponentially, but it can drastically reduce the time compared to that backtracking using the ***subset DP*** technique. It refers to doing dynamic programming where one of the parameters is a subset of some largest set.

Subset DP has two main use cases. For backtracking over permutations constructed one element at a time, we may be able to modify the recursion into only caring about *which* elements have been added to the permutation so far, rather than in what order. The result would be a reduction of the number of recursive calls from $n!$ to 2^n. That's what happens in the next problem, where we solve a variant of the NP-complete *traveling salesman problem*, a close cousin to the longest path problem, after which we examine the second use case.

Amusement Park – tivoli

By Arash Rouhani. Swedish Olympiad in Informatics 2012, Online Qualifiers.

Lisa has just arrived at an amusement park, and wants to visit each of the $N \leq 15$ attractions exactly once. For each attraction, there are two identical facilities at different locations in the park. Given the coordinates in the XY-plane of all the facilities, determine which facility Lisa should choose for each attraction to minimize the total distance she must walk. Lisa starts at the entrance at $(0, 0)$ and must return there after visiting every attraction.

Solution. Consider a partial walk constructed by naive backtracking, where we have visited attractions $s_1, \ldots, s_k$ and currently stand at the i'th facility of type j at (x, y). When deciding where to go next, it's completely irrelevant in what order the s_i were visited. We only care about what the set $S = \{s_1, \ldots, s_k\}$ is, since we do not need to visit any of the attractions in S again. A good DP state thus seems to be (S, i, j). Note that i, j only have at most 30 possibilities – two for each attraction. We also need to add an extra possible for the starting position. Since we have at most 15 kinds of attractions, the set S of visited attractions has 2^{15} possibilities. This gives us $31 \cdot 2^{15} \approx 10^6$ states. Each state can be computed in $\Theta(N)$ time, by looping over what attraction to visit next. All in all, we get a complexity of $\Theta(N^2 2^N)$.

```
1: procedure AmusementPark(i, j, S)
2:     if S contains every attraction then
3:         return dist((x_{i,j}, y_{i,j}), (0, 0))          ▷ tour is done – go back to the entrance
4:     answer ← ∞
5:     for every attraction s not yet in S do
6:         answer ← AmusementPark(s, 0, S ∪ {s}) + dist((x_{i,j}, y_{i,j}), (x_{s,0}, y_{s,0}))
7:         answer ← AmusementPark(s, 1, S ∪ {s}) + dist((x_{i,j}, y_{i,j}), (x_{s,1}, y_{s,1}))
8:     return answer
```

To code DP over subsets we generally use bitsets to represent the subset, since these map very cleanly to integers (and therefore indices into a memoization array). Revisit Section 6.5 if you don't remember how this works.. □

Subsets also appear naturally as a parameter in the backtracking recursion we base the DP on.

MeTube – dutub

By Arash Rouhani. Swedish Olympiad in Informatics 2018, School Qualifiers.

You should really be asleep by now, but you've kept watching *one more video* on MeTube all night... There are $C \leq 10$ different video categories (such as algorithm tutorials, funny cat videos and dishwasher repairs) that you are interested in. Each of the $N \leq 30$ videos on MeTube can belong to one or more categories (like a cat repairing a dishwasher as an example of a brute force algorithm). Before going to bed, you want to watch have watched videos in each category.

Given the videos, their categories and the lengths of each video, compute the minimum time you need to watch at least one video from each category.

Solution. Like in most cases where solution candidates are subsets of something, we apply a recursive solution where each video in turn either is included or excluded from the solution. After making this choice for the first k videos, the only state we need to keep track of is what categories the videos picked so far belonged to. This is a subset of size 2^C, which together with k gives $N2^C$ states, each of which takes constant time to process. □

Problem 11.11.

Programming Team Selection	programmingteamselection	
Paths	paths	*(all subtasks)*

Digit DP

Digit DP is a class of problems where we count the number with some certain properties, up to some given limit. These properties are characterized by having the classical properties of DP problems, i.e. being easily computable if we would construct the numbers digit-by-digit by remembering very little information about what those numbers actually were.

Palindrome-Free Numbers – palindromefree

By Antti Laaksonen. Baltic Olympiad in Informatics 2013

A string is a palindrome if it remains the same when it is read backwards. A number is palindrome-free if it does not contain a palindrome with a length greater than 1 as a substring. For example, 16276 is palindrome-free whereas 17276 contains the palindrome 727. The number 10102 is not valid either, since it has 010 as a substring (even though 010 is not a number itself).

Calculate the number of palindrome-free numbers between two given integers $0 \leq a \leq b \leq 10^{18}$.

Solution. A common simplification when solving counting problems on an interval $[a, b]$ is to solve the problem for $[0, a-1]$ and $[0, b]$ instead. The answer is then difference between the answer for the second interval minus the answer for the first one. These problems are much easier when counting the numbers in an interval starting at 0 instead.

Now comes an essential observation to turn the problem into a standard application of digit DP. Palindromes as general objects are very unwieldy in our situation. An iterative construction of numbers has to check digits far back in the number since any of them could be the edge of a palindrome. Fortunately, it turns out that any palindrome must

contain a rather short palindromic subsequence, namely one of length 2 (for even-length palindromes), or length 3 (for odd-length palindromes). Consequently, we only need to care about the last two digits when constructing the answer recursively. When adding a digit to a partially constructed number, it may not be equal to either of the last two digits.

Before arriving at the general solution, we solve the problem when the upper limit is $999\ldots999$ – exactly n digits of 9. Counting how many numbers we can construct of this type is then a straightforward recursion, where we construct the number one digit at a time (starting with the largest one), keeping track of the last two digits. We show C++ code for this since problem the implementation can be a bit tricky.

Snippet 11.1: Palindrome-free numbers, only 9's

```
long long palFree(int at, int len, int b1, int b2) {
    // We are done constructing the number
    if (at == len)
        return 1;
    long long answer = 0;
    for (int digit = 0; digit < 10; digit++) {
        // This digit would create a palindrome, so skip it
        if (d == b2 || d == b1)
            continue;
        // If the number has a leading zero and we add a new leading
        // zero, we make sure that the new "previous digit" is a
        // leading zero instead to avoid the palindrome check.
        if (b2 == -1 && d == 0) {
            answer += sol(at + 1, len, -1, -1);
        } else {
            answer += sol(at + 1, len, b2, d);
        }
    }
    return answer;
}

// We start the construction with an empty number with leading zeroes.
palindromeFree(0, N, -1, -1);
```

We fix the length of all numbers to have length n, by giving shorter numbers leading zeroes. Since leading zeroes in a number are not subject to the palindrome restriction, they must be treated differently. We represent them by the special digit -1 instead, resulting in 11 possible "digits". Once this function is memoized, it will have $n \cdot 2 \cdot 11 \cdot 11$ different states. Each state takes constant time to compute, so time-wise this is not an issue at all since n is bounded by 19 in the problem.

Once a solution has been formulated for this simple upper limit, extending it to a general upper limit is much easier. We first save the upper limit as a sequence of digits L. Then we need to differentiate between two cases in our recursive function. Either the *at* digits we have added so far are equal to the *at* first digits of the upper limit or they already form a smaller prefix than the upper limit. In the first case, we can't add a digit larger than

the next digit of the upper limit, or we the number would exceed the upper limit. In the other case the number we are making is lower than that of the upper limit no matter what digits we add.

These changes result in our final solution. Pay careful attention to it. While slightly tricky to get right the first time, most digit DP solutions follow this template.

Snippet 11.2: Palindrome-free numbers, general

```
vector<int> L;

long long palFree(int at, int len, bool equalToLimit, int b1, int b2) {
    // We are done constructing the number
    if (at == len)
        return 1;
    long long answer = 0;
    int maxDigit = 10;
    if (equalToLimit) {
        maxDigit = L[at];
    }
    for (int digit = 0; digit < equalToLimit; digit++) {
        // This digit would create a palindrome, so skip it
        if (d == b2 || d == b1)
            continue;

        bool newEqualToLimit = equalToLimit && digit == L[at];

        // If the number has a leading zero and we add a new leading
        // zero, we make sure that the new "previous digit" is a
        // leading zero instead to avoid the palindrome check.
        if (b2 == -1 && d == 0) {
            answer += sol(at + 1, len, equalToLimit, -1, -1);
        } else {
            answer += sol(at + 1, len, equalToLimit, b2, d);
        }
    }
    return answer;
}

// We start the construction with an empty number with leading zeroes.
palindromeFree(0, N, true, -1, -1);
```

□

Problem 11.12.

V	v
Hill Numbers	hillnumbers
Digit Sum	digitsum

Tree DP

Moving on to the last DP technique in this chapter, solutions become more abstract. When doing ***tree DP*** – dynamic programming on trees – subproblems and their parameters start

to lose their status as "just an optimized backtracking algorithm". This is in part because it's harder to visualize backtracking algorithms on tree problems as compared to problems on sequences, so the solutions seem less natural.

To do DP on trees, we start by rooting it arbitrarily and then solve subproblems for each subtree instead. The hope is that it's easy to compute the answer for the subtree of a vertex using the answers of it's children's' subtrees. That's often where the DP ends, memoizing a recursion performed on subtrees. Sometimes, we also use a *second* DP solution in order to combine the subtree answers if the problem is complicated enough. Just as for digit DP, most tree DP problems are very similar and tend to follow the structure, so make sure to internalize the following solution!

Fire Exits – fireexits

A museum has $N \leq 2000$ rooms connected by $N - 1$ corridors such that one can get from any room to every other room. The museum has no fire exits, but local regulations require that exits should be placed in a subset of rooms such that it's possible to get to an exit passing through no more than D corridors. Building exits in different rooms have different costs. Given those costs, compute the minimum cost to construct the required exits.

Solution. The rooms and corridors of the museum form a tree, which we start by arbitrarily rooting. As always in DP problems, we try to find choices to recurse on. Here, we have two choices for the root r of the tree – we either build or don't build a fire exit at it. In the first case we get a subproblem for each child v_i: computing the minimum building cost for that subtree if we already have a fire exit at distance 1 from v_i somewhere above v_i in the tree. For the purposes of the DP, we generalize the subproblem to letting the fire exit be at some arbitrary distance d from v_i instead. It will become clear why later on.

The other case is the tricky aspects of tree DP. Without an exit in r, an exit built somewhere in the subtree of e.g. v_1 could be the closest exit to vertices in the subtree v_2 in an optimal solution. Only two things matter for this spillover between subtrees though; what subtree has the fire exit closest to r, and the exit's distance to r. This yields a second subproblem for each v_i: computing the minimum building cost for a subtree if we promise to build an exit in the subtree not more than some known distance d away from v_i.

The answer to the full problem can be expressed as a subproblem of either type. For the first type, having an imaginary fire exit at distance $D + 1$ from the root for the first kind, and for the second promising to build a fire exit at most distance D from the root for the second kind. We also only have $O(N^2)$ states (d is always capped by D which is capped by N), so if we can solve the subproblems in constant time we're fine. However, we must first check that, for a given v and d, the subproblems can be solved recursively using only the answers to both questions for each child of v. Note that the two recursions are allowed to call each other! This multi-DP trick is a neat one that seldom is strictly necessary but can help to simplify some DP solutions.

We start with a slower solution that we then optimize. Call the first subproblem

$\text{HasUp}(v, d)$ and the second one $\text{NeedUp}(v, d)$. Let's compute $\text{HasUp}(v, d)$ first, i.e. there is already a fire exit d steps away from a vertex v further up in the tree. If we build a exit at v for cost c_v, we compute the total cost of the subtree as

$$c_v + \sum_{\text{child } u} \text{HasUp}(u, 1)$$

since each child u now has a fire exit at distance 1. The hard case is if v doesn't get an exit, and the exits may spill over between the child subtrees. There are two subcases here, depending on if the closest exit to v is the one d steps upwards in the tree, or in a subtree. In the first case, if $d \le D$, v needs no new exit and the closest exit to the children from further up is at distance $d + 1$, i.e. the cost is $\sum_{\text{child } u} \text{HasUp}(u, d + 1)$.

In the second case, loop over the smallest distance $d' < D$ from one of the children to an exit, as well as what child c has the exit in its subtree. That exit has distance $d' + 2$ to all the other children, so the total cost can be computed as

$$\text{NeedUp}(c, d') + \sum_{\text{child c'} \neq c} \text{HasUp}(c', d' + 2).$$

The recursion for $\text{NeedUp}(v, d)$ is similar. We either build an exit at v for the same cost as in the HasUp case, or one of the children must have built a exit that is at most distance $d - 1$ away from v. This last case can be computed in the same way too: loop over what the actual minimum distance $d' < d$ of one of the children is, and compute the cost using the same formula as in the first case.

```
1:  procedure HasUp(v, d)
2:      answer ← c_v + Σ_{child u} HasUp(u, 1)
3:      if d ≤ D then
4:          answer ← min(answer, Σ_{child u} HasUp(u, d + 1))
5:      for 0 ≤ d' < D do
6:          for child c do
7:              answer ← min(answer, NeedUp(c, d') + Σ_{c'≠c} HasUp(c', d' + 2))
8:      return answer
9:  procedure NeedUp(v, d)
10:     answer ← c_v + Σ_{child u} HasUp(u, 1)
11:     for 0 ≤ d' < d do
12:         for child c do
13:             answer ← min(answer, NeedUp(c, d') + Σ_{c'≠c} HasUp(c', d' + 2))
14:     return answer
```

These functions have three nested loops: one for d', one for c, and then there's a sum over all children to v. In the worst case, that could be $O(N^3)$ time. With N^2 subproblems we'd be looking at an upper bound of $O(N^5)$ time, way too much for $N = 2000$. We'll now start our journey to $O(1)$ per state!

The first factor N we win for free. When looping over all children in tree problems, remember that *on average* a vertex only has $O(1)$ children (there are N vertices and $N-1$ children in total).

In HasUp we win a second linear factor since the expression computed by the nested loop is actually independent of d. Thus we can compute it only once for each v, amortizing it out to $O(N)$ per state.

The same thing does not work as-is in NeedUp, since the nested loop actually depends on d. We throw in yet another useful DP trick. The only difference between NeedUp$(v, d-1)$ and NeedUp(v, d) is that the loop in the latter case includes NeedUp$(c, d-1)$ + $\sum_{c' \neq c}$ HasUp$(c', d+1)$, so the function can be simplified to

```
1: procedure NeedUp(v, d)
2:     answer ← c_v + Σ_{child u} HasUp(u, 1)
3:     answer ← min(answer, NeedUp(v, d − 1))
4:     for child c do
5:         answer ← min(answer, NeedUp(c, d − 1) + Σ_{c'≠c} HasUp(c', d + 1))
6:     return answer
```

Now, only a single factor N remains, which is the one hidden in the linear-time computation of the expression

$$\text{NeedUp}(c, d-1) + \sum_{c' \neq c} \text{HasUp}(c', d+2)).$$

We need one final trick which might be the most common one when working on this sort of thee problems. When the loop changes from a child c_a to a child c_b, the sum only changes with two terms: HasUp$(c_a, d+1)$ − HasUp$(c_b, d+1)$. The sum can thus be updated in constant time for each loop iteration.

After these optimizations, only computations that take linear time in the amount of children remain, which as stated is amortized $O(1)$ over all the vertices, so the total complexity is $\Theta(N^2)$. □

Exercise 11.13. In HasUp we have no explicit base cases. Is this a problem?

Exercise 11.14. In HasUp it may be the case that the function is invoked with $d = D+1$ in the recursive calls on lines 4 and 7. What does this subproblem represent? Can the function be called with even higher d?

Problem 11.15.
Chicken Joggers joggers

This solution was pretty hard and introduced a lot of tricks. In fact, that was most of our tree DP toolbox. Read it through a few times until you feel that you fully understand each step, and you will see that most tree DP problems become easy. The highlights were: amortized time complexity over all children, winning linear factors by updating sums

within loops that change with $O(1)$ factors, and noticing large amounts of overlap between subproblems (that NeedUp(v, d) and NeedUp$(v, d-1)$ computed almost the same thing) in order to avoid loops in the recursion. These techniques are general to all tree problems of this nature, not only the ones where we need DP.

11.4 Standard Problems

In your DP toolbox, you also need to master the following standard problems. The border between standard problems and techniques is blurry. The first problem, *Knapsack*, has so many common ubiquitous variations that it might be viewed as a technique instead. Similarly, traveling salesman – considered here as just an application of the bitset DP – could be considered a standard problem. The practical difference in the book is we look at the "normal" formulation of the standard problems rather than applied examples.

Knapsack

We start with the "traditional" knapsack problem and then briefly mention a few variations.

Knapsack – knapsack

You have a knapsack with an integer capacity C, and n different objects, the i'th with an integer weight w_i and value v_i. Select a subset of the items with maximal value, such that the sum of their weights does not exceed the capacity of the knapsack.

Solution. The common DP solution to this is $\Theta(nC)$, but depending on the constraints other solutions are also possible[1]. For very large C you can do a 2^n brute-force search, trying all subsets of the items, or with a little more effort do meet in the middle to reduce it to $2^{\frac{n}{2}}$.

Our approach is the normal one for DP on choosing subsets: we view the subset as a sequence of choices, where we either include an item or not. In this particular problem, the resulting DP state becomes very small. After including a few items, we are left only with the remaining items and a smaller knapsack to solve the problem for.

Letting $K(c, i)$ be the maximum value using at most weight c and the i first items, we get the recursion

$$K(c, i) = \max \begin{cases} K(c, i-1) \\ K(c - w_i, i-1) + v_i \quad \text{if } w_i \le c. \end{cases}$$

Translating this recursion into a bottom-up solution gives a compact algorithm:

```
1: procedure KNAPSACK(C, n, V, W)
2:     best ← new (n + 1) × (C + 1) array filled with −∞
3:     best[0][0] = 0
```

[1] In fact, $O(C \max w_i)$ can be done, but that's complex.

```
4:    for i from 1 to n − 1 do
5:        best[i] ← best[i − 1]
6:        for j from 0 to C do
7:            if W[i − 1] ≤ j then
8:                best[i][j] ← max(best[i][j], best[i − 1][j − W[i − 1]] + V[i − 1])
9:    return best
```

Exercise 11.16. How can you do the above DP in only $O(C)$ memory?

This computes the maximum values, but doesn't explicitly construct the subset. The reconstruction is very similar to the longest path in a DAG reconstruction but now has two parameters, which makes it look more complicated.

```
1: procedure KnapsackConstruct(C, n, V, W)
2:     best ← Knapsack(C, n, V, W)
3:     bestCap ← C
4:     for i from C to 0 do
5:         if best[n][i] > best[n][bestCap] then
6:             bestCap ← i
7:     for i from n to 1 do
8:         if W[i − 1] ≤ bestCap then
9:             newVal ← best[i − 1][bestCap − W[i − 1]] + V[i − 1]
10:            if newVal = best[i][bestCap] then
11:                add item i to the answer
12:                bestCap ← bestCap − W[i − 1]
```

□

Problem 11.17.

Exact Change	exactchange2
Canonical Coin Systems	canonical

The most common variation is when we are allowed to use each item an unlimited number of times. We formulate our DP solution in a similar way. The subproblem is the same, i.e. the maximum value obtainable by filling a knapsack of weight c using items $i, i-1, \ldots, 0$. In such a situation, we have two choices. We can either use i, leaving us with the same set of items but using at most $c - w_i$, or decide that we have finished using item i and look at the set of items $i-1, \ldots, 0$. The recursion is only changed by two characters:

$$K(c, i) = K(c, i-1) + K(c - w_i, i).$$

In Section 9.4 we briefly looked at a meet in the middle solution to the subset sum problem. This can also be viewed as a knapsack variant, where we ignore all the values and instead only focus on whether we *can* fill the knapsack with a certain weight or not. The knapsack DP can be adapted for subset sum with only minor changes, and conversely, the subset sum MITM can easily solve knapsack.

Problem 11.18.

Walrus Weights	walrusweights
Restaurant Orders	orders
Muzicari	muzicari

Longest Common Subsequences and Substrings

Another well-known class of DP problems is finding ***longest common subsequences and substrings*** of two sequences (see page 401 for a reminder of the terms).

Longest Common Subsequence

Given two sequences $A = \langle a_1, a_2, ..., a_n \rangle$ and $B = \langle b_1, b_2, ..., b_m \rangle$, find a longest sequence $c_1, ..., c_k$ that is a subsequence of both A and B.

Solution. When dealing with DP problems on pairs of sequences, a natural subproblem is *prefixes* of A and B. Here, some case analysis on the last letters of the strings A and B is enough for a solution. If the last letter of A is not part of a longest common subsequence, we can simply ignore it, and solve the resulting subproblem where the last letter of A is removed. The same applies to B. The remaining case is that both the last letter of A and the last letter of B are part of a longest common subsequence. In this case they can correspond to the same letter in a common subsequence, so that the remainder of the subsequence is the longest one we get after removing these two from the end of A and B. This yields a recursive formulation, which takes $\Theta(|A||B|)$ to evaluate since each state needs $\Theta(1)$ time:

$$\text{lcs}(n, m) = \max \begin{cases} 0 & \text{if } n = 0 \text{ or } m = 0 \\ \text{lcs}(n-1, m) & \text{if } n > 0 \\ \text{lcs}(n, m-1) & \text{if } m > 0 \\ \text{lcs}(n-1, m-1) + 1 & \text{if } a_n = b_m \end{cases}$$

□

Problem 11.19.

Prince and Princess	princeandprincess
Knight Search	knightsearch

Finding the longest common **substring** is not very different. The subproblem is instead finding the longest common substring that ends exactly at positions n and m in the strings, rather than the longest common substring of the prefixes. Then the answer for the cases where the last letters doesn't match is 0, so the recursion becomes

$$\text{lcs}(n, m) = \begin{cases} \text{lcs}(n-1, m-1) + 1 & \text{if } a_n = b_m \\ 0 & \text{otherwise} \end{cases}$$

and we find the answer by taking the max of all $\text{lcs}(n, m)$ results.

Longest Increasing Subsequence

Finding the ***longest increasing subsequence*** (LIS) is just as easy as finding a longest common subsequence. The LIS of a sequence A is actually the longest common sequence of A and A but sorted. This reduction gives you a simple way to compute the LIS in quadratic time.

Fortunately[2] the LIS can be found even faster than $\Theta(N^2)$ that the common subsequence reduction results in.

Longest Increasing Subsequence – longincsubseq

Given a sequence of n integers $s_1, \ldots, s_n$, find a longest increasing subsequence $a_1, \ldots, a_k$, i.e. where $a_i < a_{i+1}$.

Proof. The correct subproblem is finding the LIS of some prefix $[0, i]$ that ends with the element i. Let's denote the length of it $\mathrm{LIS}(i)$. A recursion that takes linear time per i to evaluate is simple. Among the smaller elements to the left we find the one with the longest increasing subsequence and extend it:

$$\mathrm{LIS}(i) = 1 + \max_{0 \le j < i \text{ and } s_i > s_j} \mathrm{LIS}(j).$$

To speed it up we do something very clever. We compute the values of $\mathrm{LIS}(i)$ in the order $i = 0, 1, \ldots$. At the same time, we keep an array where $B[x]$ contains the *smallest* value s_i of the sequence such that $\mathrm{LIS}(i) = x$, so far. With this array, we can rewrite the recursion in the following way:

$$\mathrm{LIS}(i) = 1 + \max_{s_i > B[x]} x.$$

Keeping the array updated is easy, after computing $\mathrm{LIS}(i)$ we set $B[\mathrm{LIS}(i)] = s_i$. This approach is still quadratic in the worst case.

To proceed, we must realize that B is a sorted array. If there is a LIS of length n that ends with an element x, we can just throw away the last element to get a LIS of length $n-1$ that ends with an element strictly smaller than x. This gives us the inequality $B[n-1] < B[n]$, so B is indeed sorted. Evaluating $\max_{s_i > B[x]} x$ is now a matter of finding the greatest x in B such that $B[x] < s_i$ in a sorted array B. This is easily done with `lower_bound` in C++ which you might remember from page 46. This function takes only logarithmic time, so that the full solution is $\Theta(N \log N)$.

To reconstruct the sequence, we need some extra bookkeeping. That code is very subtle, so we include it in C++. The implementation is slightly different from the description. Instead of $\mathrm{LIS}(i) = 1 + \max_{s_i > B[x]} x$ the equivalent computation $\mathrm{LIS}(i) = \min_{s_i \le B[x]} x$ (where this is taken to be $|B|$ if there is no such B) is used. Since B is sorted and contains no duplicates, these expressions are equivalent, but the latter is easier to compute with `lower_bound`.

[2] Or unfortunately, since this means more for you to remember!

Snippet 11.3: Longest increasing subsequence with reconstruction

```
vector<int> lis(const vector<int>& S) {
  if (S.empty()) return vector<int>();
  vi prev(S.size());
  vector<pair<int, int>> B;
  for (int i = 0; i < (int)S.size(); i++) {
    // The smallest x where B[x] >= S[i]
    int x = lower_bound(B.begin(), B.end(), make_pair(S[i], 0)) - B.begin();
    if (x == B.size()) {
      B.emplace_back();
    }
    B[x] = {S[i], i};
    prev[i] = x == 0 ? 0 : B[x - 1].second;
  }
  int len = B.size();
  int at = B.back().second;
  vi ans(len);
  while (len--) {
    ans[len] = at;
    at = prev[at];
  }
  return ans;
}
```

The meaning of `prev[i]` is to store the index of the element that precedes s_i in the LIS that ends at s_i, which is what enables the backtracking. To find it, we store not only sequence values in B, but also the index of the value as a pair. □

Exercise 11.20. How would you modify the solution to find the longest *non-decreasing* subsequence (meaning that it's enough that $a_i \leq a_{i+1}$)?

Problem 11.21.

Alphabet	alphabet
Panda Chess	pandachess
Train Sorting	trainsorting
Manhattan Mornings	manhattanmornings

Set Cover

We end the chapter with an application of subset DP on another classical NP-complete problem.

Set Cover

You are given a collection of subsets $S_1, S_2, ..., S_k$ of some larger set S of size n. Find a minimum number of subsets $S_{a_1}, S_{a_2}, ..., S_{a_l}$ such that

$$\bigcup_{i=1}^{l} S_{a_i} = S$$

i.e., *cover* the set S by taking the union of as few of the subsets S_i as possible.

Solution. For small k and large n, we can solve the problem in $\Theta(n2^k)$ by testing each of the 2^k covers. In the case where we have a small n but k can be large, this becomes intractable. Let us instead apply the principle of dynamic programming. In a brute force approach, we would perform k choices. For each subset, we would try including it or excluding it. After deciding which of the first m subsets to include, what information is relevant? If we consider what the goal of the problem is – covering S – it would make sense to record what elements have been included so far. This little trick leaves us with a DP of $\Theta(k2^n)$ states, one for each subset of S we might have reached, plus counting how many of the subsets we have tried to use so far. Computing a state takes $\Theta(n)$ time, by constructing the union of the current cover and the set we might potentially add. The recursion thus looks like:

$$\text{cover}(C,k) = \begin{cases} 0 & \text{if } C = S \\ \min(\text{cover}(C,k+1), \text{cover}(C \cup S_k, k+1)) & \text{else} \end{cases}$$

This is a fairly standard DP solution. The interesting case occurs when n is small, but k is *really* large, say, $k = \Theta(2^n)$. In this case, our previous complexity $\Theta(nk2^n)$ turns into $\Theta(n4^n)$. That's too slow for anything but very small n. To avoid this, we must rethink our DP a bit.

The second term of the recursive case of $\text{cover}(C,k)$, i.e. $\text{cover}(C \cup S_k, k+1)$, actually degenerates to $\text{cover}(C, k+1)$ if $S_k \subseteq C$. When k is large, this means many states are essentially useless. In fact, at most n of our k choices will actually result in us adding something, since we can only add a new element at most n times.

We have been in a similar situation before when solving the backtracking problem *Basin City Surveillance* in Section 9.2. We were plagued with having many choices at each state, where a large number of them would fail. Our solution was to limit our choices to a set where we *knew* an optimal solution would be found.

Applying the same change to our set cover solution, we should instead do DP over our current cover, and only try including sets which are not subsets of the current cover. So, does this help? How many subsets are there, for a given cover C, which are not its subsets? If the size of C is m, there are 2^m subsets of C, meaning $2^n - 2^m$ subsets can add a new element to our cover.

To find out how much time this needs, we use two facts. First of all, there are $\binom{n}{m}$ subsets of size m of a size n set. Secondly, the sum $\sum_{m=0}^{n} \binom{n}{m} 2^m = 3^m$. If you are not familiar with this notation or this fact, you probably want to take a look at Section 18.3.1 on binomial coefficients.

So, summing over all possible extending subsets for each possible partial C, we get:

$$\sum_{m=0}^{n} \binom{n}{m} (2^n - 2^m) = 2^n \cdot 2^n - 3^n = 4^n - 3^n$$

Closer, but no cigar. Intuitively, we still have a large number of redundant choices. If our cover contains, say, $n-1$ elements, there are 2^{n-1} sets which can extend it, but they all extend it in the same way – adding the last element. This sounds wasteful, and avoiding it probably the key to getting an asymptotic speedup.

It seems that we are missing some key function which, given a set A, can answer the question: "is there a subset S_i, that could extend our cover with some subset $A \subseteq S$?". If we had such a function, computing all possible extensions of a cover of size m would instead take time 2^{n-m} – the number of possible extensions to the cover. Last time we managed to extend a cover in time $2^n - 2^m$, but this is exponentially better!

The sum results in something different this time:

$$\begin{aligned}\sum_{m=0}^{n} \binom{n}{m} 2^{n-m} &= \sum_{m=0}^{n} \binom{n}{n-m} 2^{n-m} \\ &= \sum_{m=0}^{n} \binom{n}{m} 2^{m} \\ &= 3^n\end{aligned}$$

It turns out our exponential speedup in extending a cover translated into an exponential speedup of the entire DP.

We are not done yet – this entire algorithm depended on the existence of the magical "can we extend a cover with a subset A?" function. Sometimes, this function may be fast to compute. For example, if $S = \{1, 2, ..., n\}$ and the family S_i consists of all sets whose sum is less than n, an extension is possible if and only if *its* sum is also less than n. In the general case, our S_i are not this nice. Naively, one might think that in the general case, an answer to this query would take $\Theta(nk)$ time to compute, by checking if A is a subset of each of our k sets. Yet again, the same clever trick comes to the rescue.

If we have a set S_i of size m available for use in our cover. just how many possible extensions could this subset provide? Well, S_i itself only have 2^m subsets. Thus, if we for each S_i mark for each of its subsets that this is a possible extension to a cover, precomputation only takes 3^n time (by the same sum as above).

Since both steps are $O(3^n)$, this is also our final complexity. □

Exercise 11.22. This last step can be done in time $\Theta(k + n2^n)$. How?

Problem 11.23.

Square Fields (Hard)	squarefieldshard
Map Colouring	mapcolouring

Additional Exercises

Problem 11.24.

Selling Spatulas	sellingspatulas

Spiderman's Workout	spiderman	
Narrow Art Gallery	narrowartgallery	
Cheating a Boolean Tree	cheatingbooleantree	
Welcome to Code Jam	welcomehard	
Bus Planning	busplanning	
Maximizing Winnings	maximizingwinnings	
Nine Packs	ninepacks	
Hiding Chickens	hidingchickens	
The Uxuhul Voting System	uxuhulvoting	
Nested Dolls	nesteddolls	
Aspen Avenue	aspenavenue	
Presidential Elections	presidentialelections	
Constrained Freedom of Choice	constrainedfreedomofchoice	
Tight words	tight	
Springoalla	springoalla	*(all subtasks)*
Balanced Diet	balanceddiet	

Notes

Dynamic programming is one of the most useful algorithmic techniques to know. It appears in almost every contest at least once in, as you noticed, a large number of shapes and forms. There are many more DP techniques which we did not go through here, mostly related to different ways that a DP solution can be optimized.

The term itself is often attributed to Richard Bellman, one of the authors who the *Bellman–Ford* algorithm for shortest paths in weighted graphs is named after, which is very much a dynamic programming algorithm.

While many NP-complete problems such as TSP, knapsack, subset sum, set cover and so on have simple DP solutions that are fast enough for contest problems, they are seldom the ones of best proven time complexity or fastest in practice (which are two very different things).