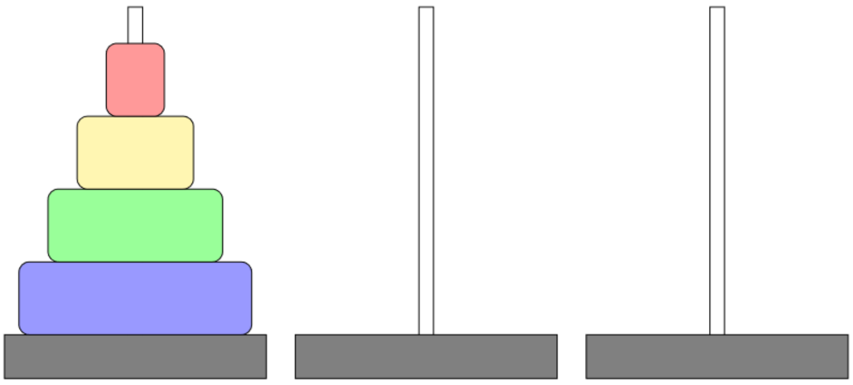
Dynamic Programming - Notes

Alperen Onur (z5161138)

August 2024

1 Tower of Hanoi

There are three pegs and n disks of radius $1, 2, ..., n$. Initially, all disks are on the first peg, in order of radius. In each move, you can move one disk from one peg to another, but you can never place a larger disk on top of a smaller disk. To win the game, stack all disks on the third peg in order of radius. We need to find the minimum number of moves to win the game.



We notice that in this problem we have overlapping subproblems. Since recursive steps involved in solving the problem are equivalent to each other. We'd need to eventually move disk 1 to $n-1$ from the first peg to the second peg and then move disks 1 to $n-1$ (perhaps in reverse order to how the were placed) from the second peg to the third peg. So we only need to recurse once $T(n) = T(n-1) + O(1)$ with asymptotic solution $T(n) = 0(n)$. More specifically, this minimum number of moves satisfy the recurrence $moves(n) = 2moves(n-1) + 1$.

Recurrence.

$$moves(i) = 2moves(i-1) + 1 \text{ for all } i \geq 2.$$

The recursion takes effect from $i = 2$ because it doesn't make sense for $i = 1$ (recall that we are solving two subproblems "at once").

Base case.
$moves(1) = 1$. This essentially acts as a termination of the recurrence and gives us a way to solve subproblems of greater size.

2 Dynamic Programming

Choose subproblems carefully such that answers to smaller subproblems can be combined to answer a larger subproblem. Greedy algorithms generally fail when more needs to be considered than the best solution at each stage. This is particularly the case the for problems which require us to count the number of ways to do something. Dynamic programming considers all options at each stage without repeating work. This is done with the idea of overlapping subproblems that are stored in a lookup table (or cache if you prefer), such that these child subproblems can be evaluated in constant time by parent subproblems.

There are three important parts to a dynamic programing algorithm that must be included in solutions

- a clear and well defined definition of the subproblems;
- a recurrence relation, which determines how the solutions to smaller subproblems are combined to solve a larger subproblem, and
- any base cases, which are trivial subproblems - those for which the recurrence is not required.

The correctness of our algorithm will need to include how the subproblems in our recurrence relate subproblems to each other, that our base case has been solved correctly and that the overall answer has been correctly solved.

3 Spaced Subset

A subset of $\{1, ..., n\}$ is said to be spaced if it does no contain any pair of consecutive integers. Let A be an array of n distinct positive integers. Our task is to find the maximum sum of the values that come from an spaced subset of indices.

Subproblems.
For $1 \leq i \leq n$, define $opt(i)$ as the maximum sum of a spaced subset using indices from $\{1, ..., i\}$.

Recurrence.

$$opt(i) = max[A[i] + opt(i-2), opt(i-1)]$$

Order of computation.
Compute in order of increasing i with a final solution $opt(n)$.

Time Complexity.
We have n subproblems with each subproblem taking $O(1)$ time to solve. Therefore $O(n)$ is our final time complexity.

4 k-Zeroed Bitstring

In this problem, we will design an algorithm that counts the number of ways to form a bit string of length n such that the 0 bits are always in groups of length k. To understand this problem statement, if $n = 3$ and $k = 2$, then possible strings are 111, 100, 001 since each string has the 0 bits occurring in groups of 2.

Subproblems
Let $num(i)$ denote the number of binary strings of length i such that all of the 0 bits are in groups of k.

Base Cases
Now, if $i < k$, we see that there is only one possible string since we can't fit any 0 bits. Therefore, we have k base cases; specifically, if $0 \leq i \leq k$, then $num(i) = 1$.

Recurrence
We may now assume that $i \geq k$ for the recurrence

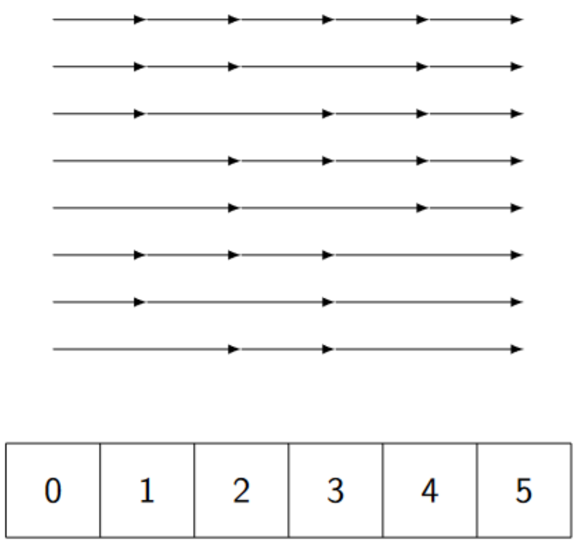
$$num(i) = num(i-1) + num(i-k)$$

Order of computation
The order of evaluation is in increasing order of i with a final solution of $num(n)$.

Time complexity
We have n subproblems and each subproblem takes constant time to compute. Therefore, the running time is $O(n)$.

5 Hopscotch

Rui is playing a version of hopscotch on a linear court. They start in square 0 and want to get to square n. From any square, Rui can step one square forward, or jump two squares forward. Design a linear time algorithm to count the number of ways that Rui can reach square n.



Subproblems
For each $1 \leq i \leq n$, let $P(i)$ be the problem of determining $num(i)$, the number of wats to reach square i.

Recurrence
For $i \geq 2$,

$$num(i) = num(i-1) + num(i-2),$$

because square i can be reached by:

- any sequence of moves to reach square $i-1$, then a step; reaching i via a step, or
- any sequence of moves to reach square $i-2$, then a jump; reaching i via a jump

Base Case
$num(0) = 1$ because there is only 1 way to reach square 0, to do nothing. $num(1) = 1$ is the other base case because the only way to reach square 1 is to take a single step. Note that the definition of this second subproblem is necessary since at $i = 1$, the recurrence can not solve the subproblem since a solution for $i = -1$ is undefined.

Order of computation
Solve subproblems $P(i)$ in increasing order of i and our overall answer will be $num(n)$.

Time complexity
$O(n)$ subproblems each taking $O(1)$, for a total of $O(n)$.

6 k-Hopscotch

Consider the following directed graph $G_{n,k} = (V_n, E_{n,k})$, defined as follows

- $V_n = \{0, ..., n\}$
- $E_n = \{i \rightarrow i+1 | 0 \leq i \leq n\} \bigcup \{i \rightarrow i+k | 0 \leq i \leq n-3\}$

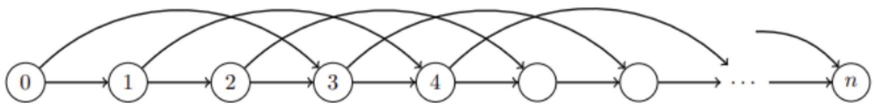


Figure 1: *The graph $G_{n,3}$.*

A path from vertex $i \to j$ is a sequence of vertices $v_1, v_2, ..., v_m$, such that $v_1 = i, v_m = j$, and there exists an edge from v_r to v_{r+1} for all $1 \le r \le m$. Design an $O(n)$ algorithm to compute the number of paths from vertex 0 to vertex n in G_n.

Subproblems
For each $0 \le i \le n$, let $NumPaths(i)$ denote the number of paths from vertex 0 to vertex i.

Recurrence
To get to vertex i, we must have either arrived from vertex $i - 1$ or vertex $i - k$. Therefore, the number of unique paths ending at vertex i is the number of unique paths ending at vertex $i - 1$, or the number of unique paths ending at vertex $i - k$.

$$NumPaths(i) = NumPaths(i - 1) + NumPaths(i - k)$$

Base cases

$$NumPaths(i) = 1 \text{ for all } 0 \le i \le k$$
$$NumPaths(i) = 0 \text{ for all } i < 0$$

Order of computation
Since $NumPaths(i)$ relies on the solution to $NumPaths(i-1)$ and $NumPaths(i-k)$, our order of computation is in increasing order of i, with the final solution being $NumPaths(n)$.

Time Complexity
There are n subproblems, one for each i, and each subproblem takes $O(1)$ time to sove; time complexity is $O(n)$.

7 Longest increasing subsequence

Given a sequence of n real numbers $A[1..n]$ determine a subsequence of maximum length, in which the values in the subsequence are strictly increasing.

To solve such a problem, we notice the fact that a subproblem that defines $P(i)$ as determining the length of the longest increasing subsequence of $A[1..i]$ doesn't immediately relate subproblems to each other. A better defined subproblem involves a subproblem $Q(i)$ which considers extending the sequence which solves $Q(j)$ for some $j < i$. If we have already solved prior subproblems, then we now look for indices $j < i$ that $A[j] < A[i]$. Among those, we pick m so that $opt(m)$ is maximal, and extend the sequence with $A[i]$. This is because our subsequence does not necessarily have to be contiguous.

Subproblems
For each $1 \le i \le n$, let $Q(i)$ be the problem of determining $opt(i)$, the maximum length of an increasing subsequence of $A[1..i]$ which ends with $A[i]$.

Recurrence
For $i > 1$,

$$opt(i) = 1 + max_{j<i \wedge A[j]<A[i]}(opt(j))$$

Base case
$opt(1) = 1$

Order of computation
Solve subproblems $Q(i)$ in increasing order of i. The overall answer will be the best of those ending at some element.

Time Complexity
$O(n)$ subproblems each taking $O(n)$, and overall answer calculated in $O(n)$, for a total of $O(n^2)$.

Justification
For a sequence of indices $[..., m, i]$ is the longest increasing subsequence ending at index i, then omitting the last entry must leave a longest increasing subsequence at index m; the truncation for solution $Q(i)$ must be an optimal solution for some earlier subproblem $Q(m)$. But if there was a longer subsequence ending at index m,

we could append index i to that, making an even longer subsequence ending at index i. Therefore, we have a contradiction and furthermore, our algorithm produces a optimal solution to subproblems.

Modification to solve similar problems
If we wanted to for example, find the values of the longest increasing subsequence rather than just the length we can modify our solution to do so. In the ith slot of the table, alongside $opt(i)$ we also store the index m such that the optimal solution for $Q(i)$ extends the optimal solution for $Q(m)$. After all subproblems have been solved, the longest increasing subsequence can be recovered by backtracking through the table. This contributes only a constant factor to the time complexity.

8 Activity Selection

Given a list of n activities with starting times s_i and finishing times f_i. No two activities can take place simultaneously. Find the maximal total duration of a subset of compatible activities.

Begin by sorting the activities by their finishing time into a non-decreasing sequence, and henceforth we will assume $f_1 \leq f_2 \leq ... \leq f_n$. We can then specify the subproblems, for each $1 \leq i \leq n$, then $P(i)$ be the problem of finding the duration $t(i)$ of a subsequence $\sigma(i)$ of the first i activities which

- Consists of non-overlapping activities,
- ends with activity i, and
- is a maximal total duration among all such sequences

We would like to solve $P(i)$ by appending activity i to $\sigma(j)$ for some $j < i$ if activity i does not overlap with activity j. Among all such j, our recurrence will choose that which maximises the duration $t(j)$.

Subproblems
For each $1 \leq i \leq n$, let $P(i)$ be the problem of determining $t(i)$, the maximal duration of a non-overlapping subsequence of the first i activites which ends with activity i.

Recurrence
For $i > 1$,

$$t(i) = (f_i - s_i) + max_{j<i \wedge f_j < s_i}(t_j).$$

Base Case
$t(1) = f_1 - s_1$. **Order of Computation**
Solve subproblems $P(i)$ in increasing order of i. The overall answer $max_{1 \leq i \leq n}(t(i))$ considers all possible choices for the last activity selected.

Time Complexity
Sorting in $O(n \log n)$. There are $O(n)$ subproblems each taking $O(n)$ time to solve $O(n^2)$.

Justification We will form an **inductive argument**. We claim the optimal solution of subproblem $P(i)$ be given by the sequence of activities $\sigma = [..., m, i]$. We claim that $\sigma^{'} = [..., m]$ gives an optimal solution to subproblem $P(m)$. To prove this, consider the opposite. Suppose instead that $P(m)$ is solved by a sequence $\varsigma^{'}$ of even larger total duration. Then make an activity selection ς by appending activity i to $\varsigma^{'}$. It is clear that this is a valid selection of activities with larger total duration than σ. This contradicts the earlier definition of σ as the sequence solve $P(i)$. Thus, the optimal sequence for problem $P(i)$ is obtain by extending the optimal sequence for some earlier subproblem with activity i.

9 Making Change

You are given n types of coin denominations of integer values $v_1 < v_2 < .. < v_n$. Assume $v_1 = 1$ and that you have an unlimited supply of coins of each denomination. Make change for a given integer amount C using as few coins as possible.

We will try to find the optimal solution for not only C, but every amount up to C. Assume we have found optimal solutions for every amount $j < i$ and now want to find an optimal solution for amount i. We consider each coin v_k as part of the solution for amount i, and make up the remaining amount $i - v_k$ with the previously computed optimal solution. Among all these optimal solutions,, which we find in the table we are constructing recursively, we pick one which uses the fewest number of coins. Supposing we choose coin m, we obtain an optimal solution for amount i by adding one coin of denomination v_m to $opt(i - v_m)$.

Subproblems
For each $0 \leq i \leq C$, let $P(i)$ be the problem of determining $opt(i)$, the fewest coins needed to make chance for an amount i.

Recurrence
For $i > 0$,

$$opt(i) = 1 + min_{1 \leq \ k \leq n \ \wedge \ v_k \leq i}(opt(i - v_k))$$

Base case
$opt(0) = 0$

Order of computation
Solve $P(i)$ in increasing order of i and our overall answer is $opt(C)$

Time Complexity
$O(C)$ subproblems each taking $O(n)$ for a total of $O(nC)$

10 Integer Knapsack

You have n items, the ith of which has weight w_i and value v_i. All weights are integers. You also have a knapsack of capacity C. **You can take each item any integer number of times**. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

We solve for each total weight up to C; $j < i$. Consider each item, the kth of which has weight w_k to include. If it is included, it would have $i - w_k$ wight to fill. The best packing of item k is one of item k plus the best packing of weight $i - w_k$. We should consider all items k in this way, and pick the one that gives the largest total value.

Subproblems
For each $0 \leq i \leq C$, let $P(i)$ be the problem of determining $opt(i)$, the maximum value that can be achieved using up to i units of weight, and $m(i)$, the index of an item in such a collection.

Recurrence For $i > 0$,

$$\begin{aligned} opt(i) &= max_{1 \leq k \leq n \wedge w_k \leq i}[v_k + opt(i - w_k)] \\ m(i) &= argmax_{a \leq k \leq n \wedge w_k \leq i}[v_k + opt(i - w_k)] \end{aligned}$$

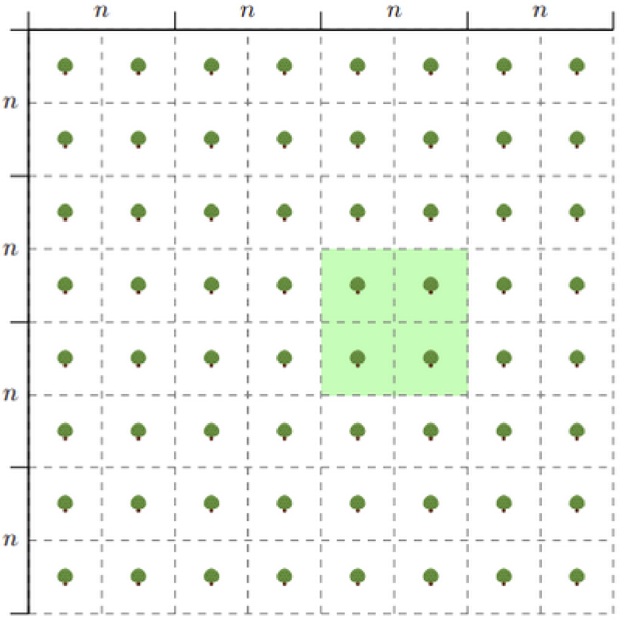
Base Case
If $i < min_{1 \leq k \leq n}[w_k]$, then $opt(i) = 0$ and $m(i) = 0$.

Order of Computation
Solve subproblems $P(i)$ in increasing order of i. The overall answer is $opt(C)$ as the knapsack can hold up to C units of weight.

Time Complexity
$O(C)$ subproblems taking $O(n)$ time for a time complexity of $O(nC)$ time complexity.

11 Prefix Sums

You are in a square orchard, with orange trees on a $4n \times 4n$ grid. For example, if $n = 2$, the orchard has 64 trees in this layout



You would like to purchase oranges from n^2 trees forming an $n \times n$ square section of the orchard. Fortunately, you know exactly how many oranges are in each tree, given an array $A[1..4n][1..4n]$ where $A[i][j]$ is the number of oranges in a tree at (i, j). You would like to find the $n \times n$ square containing the largest number of oranges.

Subproblems
Let $oranges(i, j)$, denote the sum of oranges contained in tree (k, l) fir all $k \leq i$, $l \leq j$. That is, the total number of oranges contained in the rectangle starting at $(1, 1)$ and ending at (i, j).

Recurrence

$$oranges(i, j) = oranges(i + n, j + n) - oranges(i + n, j) - oranges(i, j + 1) + oranges(i, j)$$

Base Cases
$oranges(i, 0) = oranges(0, j)$ for all i, j.

Order of computation
Subproblems can be solved in increasing order of i, followed by increasing order of j.

Time complexity
There are n^2 subproblems taking $O(1)$ time to solve; $O(n^2)$.

12 0-1 Knapsack

You have n items, the ith of which has weight w_i and value v_i. All weights are integers. You also have a knapsack of capacity C. **You can take each item only at most once**. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

This modification on the knapsack causes some problems. Just because we know the optimal solution for each total weight $j < i$ we can not deduce the optimal solution for weight i. If we begin our solution for weight i with item k, we have $i - w_k$ remaining weight to fill. However, we did not record whether item k was itself used in the optimal solution for that weight. We need an extra parameter in our recurrence that is also well defined to solve this. For each total weight i, we find a solution using only the first k items. If k is used in the solution then we have $i - w_k$ remaining weight to fill using the first $k - 1$ items and otherwise we must fill all i units of wight with the first $k - 1$ items.

Subproblems
For $0 \le i \le C$ and $0 \le k \le n$, let $P(i,k)$ be the problem of determining

- $opt(i,k)$, the maximum value that can be achieved using up to i units of weight and using only the first k items, and
- $m(i,k)$, the index of an item in such collection

Recurrence
For $i > 0$ and $i \le k \le n$,

$$opt(i,k) = max[v_k + opt(i - w_k, k-1), opt(i, k-1)]$$

with $m(i,k) = k$ or $m(i,k) = m(i, k-1)$ respectively.

Order of Computation

- When we get to $P(i,k)$, the recurrence requires us to have already solved $P(i,k-1)$ and $P(i - w_k, k-1)$
- This is guaranteed if we solve subproblems $P(i,k)$ in increasing order of k.
- Our overall answer will be $opt(C,n)$

Time Complexity
$O(nC)$ subproblems taking $O(1)$ time for a total of $O(nC)$.

13 Longest Common Subsequence & Shortest Common Subsequence

Suppose you have two sequences $A = [a_1, a_2, ..., a_n]$ and $B = [b_1, b_2, ..., b_m]$. Find the length of a longest common subsequence of A and B. A sequence is a subsequence of another sequence S if s can be obtained by deleting some of the symbols of S.

Consider prefixes of both subsequences, say $A_i = [a_1, a_2, ..., a_i]$ and $B_j = [b_1, b_2, ..., b_j]$. If a_i and b_j are the same symbol, (SAY z) the longest common subsequence of A_i and B_j is formed by appending z to the solution for A_{i-1} and B_{j-1}. Otherwise, a common subsequence of A_i and B_j cannot contain matches for both a_i and b_j so we consider discarding each of these symbols in turn.

Subproblems
For all $0 \le i \le n$ and all $0 \le j \le m$ let $P(i,j)$ be the problem of determining $opt(i,j)$, the length of the longest common subseqeunce of the truncated sequences $A_i = [a_1, a_2, ..., a_i]$ and $B_j = [b_1, b_2, ..., b_j]$.

Recurrence

$$opt(i,j) = \begin{cases} 1 + opt(i-1, j-1) \text{ if } a_i = b_j \\ max[opt(i-1, j), opt(i, j-1)] \text{ otherwise} \end{cases}$$

Base cases
For all $0 \le i \le n$, $opt(i,0) = 0$ and for all $0 \le j \le m$, $opt(0,j) = 0$.

Order of Computation
Solve the subproblems $P(i,j)$ in lexicographic order (increasing i, then increasing j) to guarantee that $P(i-1,j), P(i,j-1)$ and $P(i-1,j-1)$ are solved before $P(i,j)$, so all dependencies are satisfied. So our overall answer is $opt(n,m)$.

Time Complexity
$O(nm)$ subproblems taking $O(1)$ time for a total of $O(nm)$.

Shortest common subsequence
This is a simple modification of our solution. Find the longest common subsequence $LCS(A,B)$, then add back the differing elements of the two sequences in the right places, in any compatible order.

14 Matrix chain multiplication

You have a sequence of compatible matrices $A_1A_2...A_n$ where A_i is of dimension $s_{i-1} \times s_i$. Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

Specify a subproblem corresponding to each contiguous subsequence $A_{i+1}A_{i+2}...A_j$ of the chain. The recurrence will consider all possible ways to place the outermost multiplication, splitting the chain into the product $(A_{i+1}...A_k)(A_{k+1}..A_j)$.

Subproblems
For all $0 \leq i \leq j \leq n$, let $P(i,j)$ be the problem of determining $opt(i,j)$, the fewest multiplications needed to compute the product $A_{i+1}A_{i+2}..A_j$.

Recurrence
For all $j - i > 1$,

$$opt(i,j) = min_{i<k<j}[s_i s_k s_j + opt(i,k) + opt(k,j)]$$

Base cases
For all $0 \leq i \leq n-1$, no multiplications are required for the single-matrix chain A_{i+1}, so $opt(i,i+1) = 0$.

Order of computation

- To solve a subproblem $P(i,j)$, we must have already solved $P(i,k)$ and $P(k,j)$ for each $i < k < j$.
- The simplest way to ensure this is to solve the subproblems in increasing order of $j - 1$
- Our overall answer is $opt(0,n)$

Time Complexity
$O(n^2)$ subproblems taking $O(n)$ time for a total of $O(n^3)$.

15 Grid Paths

Suppose we have a grid G with n rows and m columns; but this time there is a set of cells S, known in advance, that you want to avoid. You are guaranteed that the starting cell is not in S, Your task is to count the number of paths from the top-left cell to the bottom-right cell, that do not include any cell in S. The only valid moves along these paths are to step one cell rightwards or one cell downwards.

Subproblems
Let $NumPaths(i,j)$ be the number of paths from $(1,1)$ to (i,j) without stepping on any cell in S.

Recurrence
$$NumPaths(i,j) = \begin{cases} 0 \text{ if } (i,j) \in S \\ NumPaths(i-1,j) + NumPaths(i,j-1) \text{ otherwise} \end{cases}$$
Base Case
$NumPaths(i,0) = 0$ and $NumPaths(0,j) = 0$

Order of Computation
In increasing order of i, then j. Or, in increasing of j, then i. Or increasing order of $i + j$. The final solution is $NumPaths(m,n)$.

Time Complexity
We have mn many subproblems, all solvable in constant time; final time complexity is $O(mn)$.

Justification
Suppose for a cell (i,j) that we have solved its two predecessors $(i-1,j),(i,j-1)$ correctly. Then, note that we cannot traverse any cell in S; by definition of subproblem, by being on a path including such a cell, we have taken at least on cell in S. Therefore, those subproblems are solved correctly, with solution 0. Then, notice that this cell 'ends' paths, since its solution contributes no paths to the cells below and to the right, so no paths through this cell will be counted in future solutions.

Then, since there exists no other paths to (i, j), the computed solution must be the sum of the number of paths to the cells directly above and to the left. Hence we solve (i, j) correctly, given our base cases and predecessors.

16 Maximum Dot Product

Suppose you have two arrays $A[1..m], B[1..n]$. The task is to return the maximum dot product between non-empty subsequences of A, B. The dot product of two subsequences of length k is denoted to $a \cdot b = \sum_{i=1}^{k} a_i b_i$.

Subproblems
Let $MaxDotProduct(i, j)$ be the maximum dot product produced by taking non-empty subsequences of $A[1..i], B[1..j]$.

Recurrence

$$MaxDotProduct(i, j) = max \begin{cases} MaxDotProduct(i, j-1) \\ MaxDotProduct(i-1, j) \\ MaxDotProduct(i-1, j-1) + A[i] \cdot B[j] \end{cases}$$

Base Case
$MaxDotProduct(i, 0) = 0$ and $MaxDotProduct(0, j) = 0$

Final Solution
$MaxDotProduct(m, n)$

Time Complexity
mn many subproblems, all solvable in constant time; $O(mn)$

17 Palindromes

Let $S[1..n]$ be a string of n characters. A palindromic subsequence is a subsequence of S that also forms a palindrome. Recall that a palindrome is a string that is read the same forward and backwards, such as kayak and racecar. Design an $O(n^2)$ algorithm that returns the length of the longest palindrome subsequence.

Subproblem For $1 \leq i \leq j \leq n$, let $lps(i, j)$ denote the length of the longest palindromic subsequence of $S[i..j]$.

Recurrence
We consider two different cases. If $S[i] = S[j]$, then we should take both endpoints. If $S[i] \neq S[j]$, then we must discard one of the endpoints.

$$lps(i, j) = \begin{cases} 2 + lps(i+1, j-1) \text{ if } S[i] = S[j] \\ max[lps(i+1, j), lps(i, j-1)] \text{if } S[i] \neq S[j] \end{cases}$$

Base case
If $i = j$, then $lps(i, i) = 1$ and if $i > j$ then $lps(i, j) = 0$.

Order of Computation
Solve the subproblems in order of increasing $j - 1$ with a final solution of $lps(1, n)$.

Time Complexity
There are n^2 subproblems with each taking constant time to solve; $O(n^2)$.

18 Shortest path in directed acyclic graphs

If all edge weights are positive, the single source shortest path is solved by Dijkstra's algorithm in $(m \log n)$. We can do better with the Bellman-Ford algorithm to solve it in $O(mn)$ time. However, in the special case of DAG's, a simple DP solves the problem in $O(n + m)$.

The natural subproblems are the shortest path to each vertex. Each vertex v with an edge to t is a candidate for the penultimate vertex in an $s - t$ path. The recurrence considers the path to each v, plus the weight of the last edge, and select the minimum of these options.

Subproblems
For all $t \in V$, let $P(t)$ be the problem of determining $opt(t)$, the length of the shortest path from $s \to t$.

Recurrence
For all $t \neq s$,

$$opt(t) = min_{v:(v,t)\in E}[opt(v) + w(v,t)]$$

Base case
$opt(s) = 0$ **Order of computation**
Here, $opt(t)$ depends on all the $opt(v)$ values for the vertices v with outgoing edges to t, so we need to solve $P(v)$ for each such v before solving $P(t)$. We can achieve this by solving the vertices in topological order, from left to right. All edges point from left to right, so any vertex with an outgoing edge to t is solved before t is. Our overall answer is $opt(t)$ over all vertices t.

Time Complexity
At first it appears that each of n subproblems is solved in $O(n)$ time, giving a time complexity of $O(n^2)$. However, each edge is only considered once, so we can use the tighter bound $O(n + m)$.

Modify for longest path
If you replace min in the recurrence by max, you have an algorithm to find the longest path from s to each t. This can only be done if the graph is specifically acyclic and in topological order.

19 Bellman-Ford Algorithm

Consider a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight, and a designated vertex $s \in V$. Find the weight of the shortest path from vertex s to every other vertex t. Note that there are no cycles of total negative weight. We can not apply a greedy algorithm (Dijkstras) because of the addition of negative edge weights. We consider a DP approach; Bellman-Ford.

Bellman-Ford algorithm
For every vertex t, lets find the weight of a shortest $s - t$ path consisting of at most i edges, for each i up to $n - 1$. Suppose the path in question is $p = s \to .. \to v \to t$ and another path $p^{'} = s \to .. \to v$. Then $p^{'}$ must itself the shortest path from s to v at most $i - 1$ edges, which is another subproblem.

Subproblems
For all $0 \leq i \leq n - 1$ and all $t \in V$, let $P(i, t)$ be the problem of determining $opt(i, t)$, the length of a shortest path from s to t which contains at most i edges.

Recurrence
For all $i > 0$ and $t \neq s$,

$$opt(i, t) = min_{v:(v,t)\in E}[opt(i - 1, v) + w(v, t)]$$

To reconstruct the actual shortest path, you'll need predecessors to back track and as such, another recurrence.

$$pred(i, t) = argmin_{v:(v,t)\in E}[opt(i - 1, v) + w(v, t)]$$

Base cases
$opt(i, s) = 0$, and for $t \neq s$, $opt(0, t) = \infty$

Order of computation
Solve subproblems $P(i, t)$ in increasing order of i since $P(i, t)$ depends only on $P(i - 1, v)$ for various v. The overall answer is the list of values $opt(n - 1, t)$ over all vertices t.

Time Complexity
There are n rounds $i = 0, 1, ..., n - 1$. In each round, each edge (v, t) of the graph is considered only once. Therefore the time complexity is $O(nm)$. It is possible however to reduce to to $O(n)$ observing that $opt(i-1, t)$ as a candidate for $opt(i, t)$ and as such, maintaining a table with only one row and overwrite it each round.

20 Floyd-Warshall algorithm

Consider a directed weight graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight. Find the weight of the shortest path from every vertex s to every other vertex t. We note that there can only be a shorter path from s to t that is better than the previous round if there is a shorter path of form.

$$s \to \underbrace{\dots}_{v_1,\dots,v_{k-1}} \to v_k \to \underbrace{\dots}_{v_1,\dots,v_{k-1}} \to t.$$

Subproblems
For all $1 \leq i, j \leq n$ and $0 \leq k \leq n$, let $P(i, j, k)$ be the problem of determining $opt(i, j, k)$, the weight of a shortest path from v_i to v_j using only $v_1, .., v_k$ as intermediate vertices.

Recurrence
For all $1 \leq i, j, k \leq n$,

$$opt(i, j, k) = min[opt(i, j, k-1), opt(i, k, k-1) + opt(k, j, k-1)]$$

Base cases

$$opt(i, j, 0) = \begin{cases} 0 \text{ if } i = j \\ w(i, j) \text{ if } (v_i, v_j) \in E \\ \infty \text{ otherwise} \end{cases}$$

Order of Computation
Solve subproblems in increasing order of k since $P(i, j, k)$ depends only on $P(i, j, k-1)$, $P(i, k, k-1)$ and $P(k, j, k-1)$. Te overall answer is the table of values $opt(i, j, n)$, over pairs of vertices i and j.

Time Complexity
$O(n^3)$ subproblems taking $O(1)$ time, for a total of $O(n^3)$.

21 Rabin-Karp Algorithm

Determine whether a string $B = b_1b_2...b_m$ appears as a contiguous substring of a much longer string $A = a_1a_2...a_n$. We can use hashing to form a fast algorithm to compare the two strings. Compute the hash of B as follows

- Map each symbol to a corresponding integer $i \in \{0, 1, 2, ..., d-1\}$ so as to identify each string with a sequence of these integers. Hereafter, when we refer to an integer a_i or b_i, we really mean the ID of the symbol a_i or b_i. We can therefore identify B with a sequence of IDs each between 0 and $d-1$ inclusive. Viewing the digits in base d, construct the following integer and then evalute using Horner's rule

$$h(B) = h(b_1b_2..b_{m-1}b_m)$$
$$h(B) = b_m + d \times (b_{m-1} + d \times (b_{m-2} + ... + d \times (b_2 + d \times b_1)...)),$$

  requiring only $m-1$ additions and $m-1$ multiplications

Next we choose a large prime number p and define the has value of B as $H(B) = h(b) mod p$

Now we want to efficiently find all starting points s such that the string of length m is of form $a_sa_{s+1}..a_{s+m-1}$ and string $b_1b_2...b_m$ are equal. For each contiguous substring $A^s = a_sa_{s+1}...a_{s+m-1}$ of string A we also compute its hash value as

$$H(A^s) = d^{m-1}a_s + d^{m-2}a_s + 1 + .. + d^1a_{s+m-2} + a_{s+m-1} mod p$$

We now compare the hash values of $H(B)$ and $H(A^s)$. If they disagree, there is no match. If there agree, there might be a match. To confirm this, we'll check character by character. There are $O(n)$ substrings A^s to check. If we compute each hash value $H(A^s)$ in $O(m)$ time, this is no better than a greedy approach. However if we use recursion, we don't need to compute the hash value $H(A^{s+1})$ from scratch. Instead, compute it efficiently from the previous hash value $H(A^s)$. We get a recurrence of

$$H(A^{s+1}) = d \cdot H(A^s) - d^m a_s + a_{s+m} mod p$$

Algorithm.

- First compute $H(B)$ and the base case $H(A^1)$ in $O(m)$ time using Horner's rule.
- Then compute the $o(n)$ subsequent values of $H(A^s)$ each in constant time using the recurrence above.
- Compare each $H(A^s)$ with $H(B)$, and if they are equal then confirm the potential match by checking the strings A^s and B character-by-character.

Since p was chosen large, the false positives are very unlikely, which makes it fast in the average case, we consider an algorithm with a guaranteed worst case performance.

22 Rabin Karp matching

Consider the string $S = abccbbcabc$ over the alphabet $\sum = \{a, b, c\}$. Use the Rabin-Karp algorithm to determine whether S contains the substring $S^{'} = bca$.

Algorithm.
There are 3 characters in out alphabet, we perform the substitution

$$\{a, b, c\} \to \{0, 1, 2\}$$

Therefore, S becomes 0122112012 and $S^{'}$ becomes 120.

Now we break our string S into groups of three to get

$$\begin{array}{ll}
abc & 012 \pmod 5 \equiv 5 \pmod 5 = 0 \\
bcc & 122 \pmod 5 \equiv 17 \pmod 5 = 2 \\
ccb & 221 \pmod 5 \equiv 25 \pmod 5 = 0 \\
cbb & 211 \pmod 5 \equiv 22 \pmod 5 = 2 \\
bbc & 112 \pmod 5 \equiv 14 \pmod 5 = 4 \\
bca & 120 \pmod 5 \equiv 15 \pmod 5 = 0 \\
cab & 201 \pmod 5 \equiv 19 \pmod 5 = 4 \\
abc & 012 \pmod 5 \equiv 5 \pmod 5 = 0
\end{array}$$

Check through all hash collisions and see if strings match. Once we perform the hashing modulo 5, we can check if there are any other substring that map to 0. The false positives are *abc* and *ccb*.

23 Knuth-Morris-Pratt Algorithm

In the Rabin-Karp algorithm, for each starting position s in the text A, match each character of the pattern B in order until a conflict is found. When a conflict is found, we return to the beginning. However, a matched section might be salvageable.

Failure Function.
Let $\pi(k)$ be the length of the longest string which is both a prefix and suffix of some pattern B_k up until the kth symbol. $B_{\pi(k)}$ is therefore the longest substring which appears at both the start and end of b_K (Allowing partial but not total overlap).

Any time we fail to extend our current partial match, we could instead try to extend the longest prefix-suffix of length $\pi(k)$. If this fails, we could try then ti extend its longest prefix-suffix (of length $\pi(\pi(k))$) and so on.

Algorithm.
Maintain pointers l and r into the text, which record the left and right boundaries of our current partial match. Initially $l = 1$ and $r = 0$. We'll use $w = r - l + 1$ as a shorthand for the length of the current partial match. Now, scan forwards through the text

- Compare the next character of the text a_{r+1} to b_{w+1}. If they agree, then we can extend the partial match.

- Otherwise, shorten the partial match: reduce w to $\pi(w)$ by increasing l by the appropriate amount, and repeat the previous step.
- If the character ever agree, increase r by one and move on to the next character of the text.
- If a match of length 0 can't be extended, simply increase by l and r and move on.

If the match length w reaches m, report a match for the entire pattern. Then reduce w to $\pi(m)$ and continue, to detect any overlapping full matches.

Time complexity
After each comparison between a pair of characters, at least one of the pointers l and r moves forwards. Each pointer can take up to n values, so the total number of steps is $O(n)$. This is all done assuming we have computed the failure function. We can do it using dynamic programming.

Subproblems. Let $P(k)$ be the problem of determining $\pi(k)$, the length of the longest prefix-suffix of B_k.

Recurrence
For $k \geq 1$,

$$\pi(k+1) = \begin{cases} \pi(k)+1 \text{ if} b_{k+1} = b_{\pi(k)+1} \\ \pi(\pi(k))+1 \text{ else if} b_{k+1} = b_{\pi(\pi(k)+1} \\ . \\ . \\ . \\ \ldots\ldots \end{cases}$$

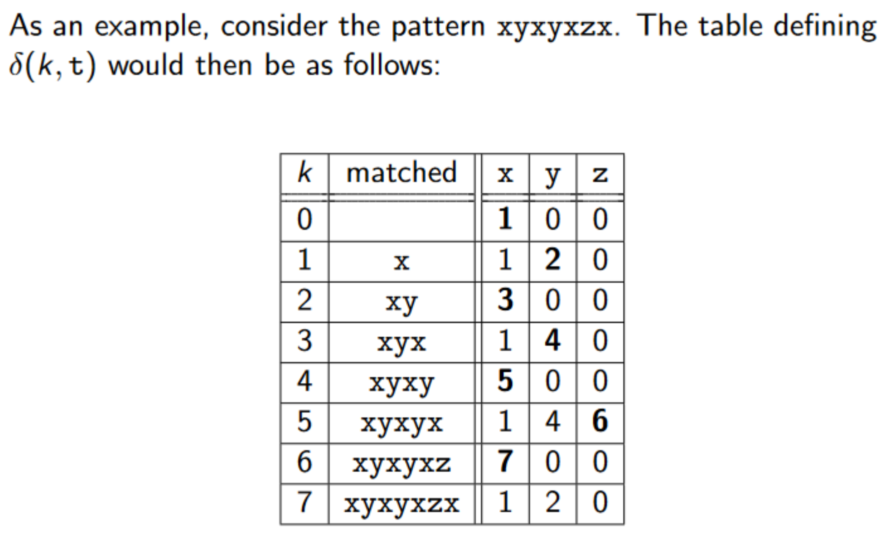
Base case
$\pi(1) = 0$
Order of computation
Solve subproblems $P(k)$ in increasing order of k. Th overall answer is $O(m)$.

Time complexity
$O(m)$

Finite automata
The Knuth-Morris-Pratt algorithm can be conceptualised using a finite automaton item A string matching finite automaton for a pattern B of length m has

- $m+1$ many states $0, 1, ..., m$ which correspond to the number of characters matches thus far, and
- a transition function $\delta(k, t)$, where $0 \leq\leq m$ and t is a symbol in the alphabet.

Suppose that the last k characters of the text A match B_k, and that t is the next character in the text. Then $\delta(k, t)$ is the new state after t is read, i.e. the largest j so that the last j characters of A match the start of pattern B. Clearly the transition function δ is closely related to the failure function π.



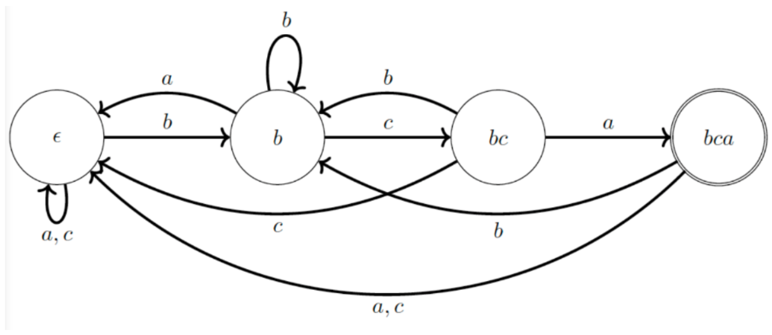
As an example, consider the pattern xyxyxzx. The table defining $\delta(k, t)$ would then be as follows:

k	matched	x	y	z
0		**1**	0	0
1	x	1	**2**	0
2	xy	**3**	0	0
3	xyx	1	**4**	0
4	xyxy	**5**	0	0
5	xyxyx	1	4	**6**
6	xyxyxz	**7**	0	0
7	xyxyxzx	1	2	0

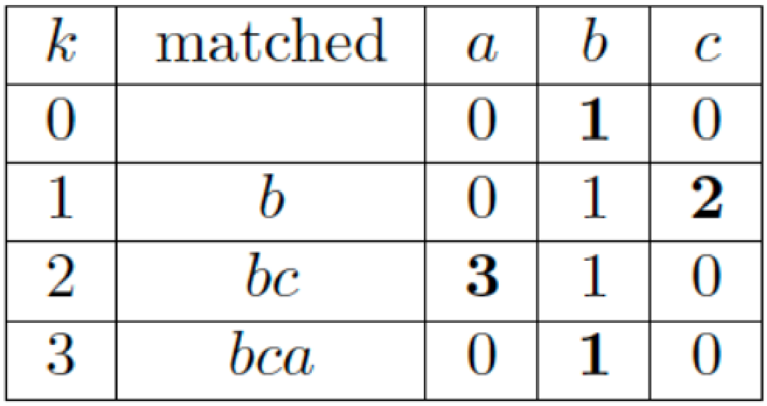
24 Knuth-Morris-Pratt matching

Using a finite automaton for $S^{'} = bca$ and transition table, check whether the string $S = abccbbcabc$ over alphabet $\sum = \{a, b, c\}$ has substring $S^{'}$.

Finite automaton



Each particular transition tells us where to proceed when we've seen a particular substring of $S^{'}$. We have four states represented by the prefixes ϵ, b, bc, bca. The accepting state is the state representing bca. Using the finite automaton, we can recreate the transition table. **Transition Table**



k	matched	a	b	c
0		0	**1**	0
1	b	0	1	**2**
2	bc	**3**	1	0
3	bca	0	**1**	0

Algorithm.
We stimulate the word S on our finite automaton. Since the automaton is deterministic, there is exactly one state at each iteration of the string. We also see that, if the string hits the accepting state (bca) at any iteration of the algorithm, then $S^{'}$ must appear as a substring of S. Therefore, if we hit the accept state any iteration, then $S^{'}$ is a substring of S; otherwise $S^{'}$ is not a substrong of S.