Module 5: Dynamic Programming

Raveen de Silva (K17 202)
Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Tower of Hanoi

Problem

Instance: There are three pegs and n disks of radius $1, 2, \ldots, n$.

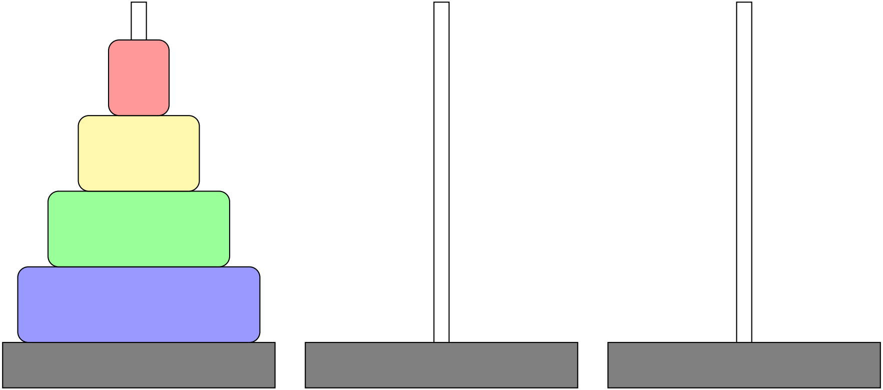
Initially, all disks are on the first peg, in order of radius.

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 (again in order of radius).

Task: Design an algorithm which finds the minimum number of moves to win the game.

Tower of Hanoi



Tower of Hanoi

- To find the minimum number of moves, we'll need a systematic winning strategy.

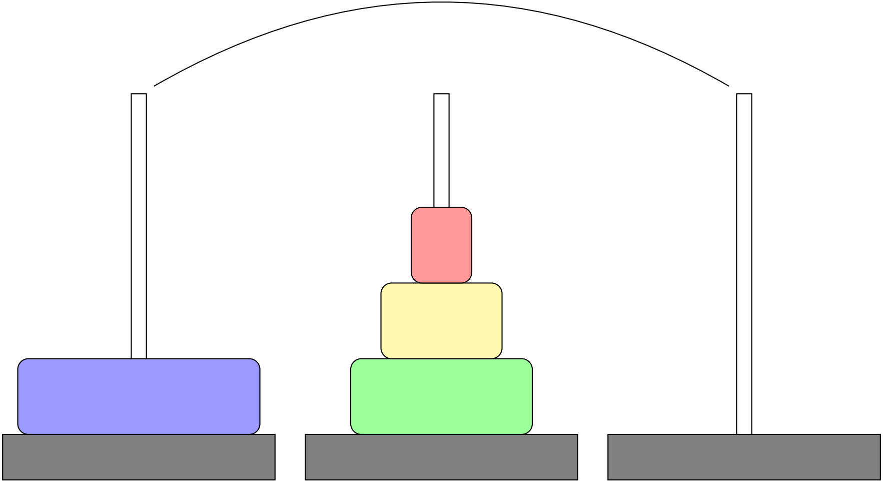
Observation

The largest disk has to move from the first peg to the third peg.

Observation

The largest disk can only move from one peg to another if all the other disks are on the remaining peg.

Tower of Hanoi



Tower of Hanoi

- To minimise the number of moves overall, we should:

  1. move disks 1 to $n-1$ from the first peg to the second peg, using as few moves as possible,

  2. move disk n to the third peg, and finally

  3. move disks 1 to $n-1$ from the second peg to the third peg, using as few moves as possible.

- The first and third parts of this could be solved recursively, noting that disk n does not impact the movement of any smaller disks.

Tower of Hanoi

- The solution we have described is a divide and conquer approach.
  - **Divide:** nothing to do before recursion.
  - **Conquer:** twice, recursively move $n - 1$ disks.
  - **Combine:** add the number of moves in all three steps.
- The time complexity satisfies the recurrence

$$T(n) = 2T(n-1) + \Theta(1),$$

with asymptotic solution $T(n) = \Theta(2^n)$.

- Is there any redundancy?

Tower of Hanoi

- The two recursive steps are equivalent to each other. It should take the same number of moves to complete both of these steps:

  1. move disks 1 to $n-1$ from the first peg to the second peg
  3. move disks 1 to $n-1$ from the second peg to the third peg.

- Here, we have made use of *overlapping subproblems*.
  - In Divide and Conquer, we solved each of the child instances in isolation.
  - If the child instances are equivalent to each other, we can just solve one and then re-use the result.
  - More generally, if they have some overlapping structure, we can store the answers to smaller subproblems and re-use them later.

Tower of Hanoi

- With this observation, we now only need to recurse *once*.

- The time complexity could be modelled using the recurrence

$$T(n) = T(n-1) + \Theta(1),$$

  with asymptotic solution $T(n) = \Theta(n)$.

- More precisely, the minimum number of moves satisfies the recurrence

$$\text{moves}(n) = 2\,\text{moves}(n-1) + 1.$$

Tower of Hanoi

- In general,

  $$\mathrm{moves}(i) = 2\,\mathrm{moves}(i-1) + 1$$

  for all $i \geq 2$.

  - The recursion takes effect from $i = 2$ because it doesn't make sense for $i = 1$; there is no smaller number of disks to recurse to.

- With the base case $\mathrm{moves}(1) = 1$ (why?), we could now solve for $\mathrm{moves}(2), \mathrm{moves}(3), \ldots, \mathrm{moves}(n)$ in succession using the recurrence.

- Each of these takes constant time, so the answer $\mathrm{moves}(n)$ is found in $\Theta(n)$ time.

Tower of Hanoi

Remark

In this example, the recurrence can in fact be solved explicitly to find $\text{moves}(n) = 2^n - 1$.

However, we won't dwell on this because the method we used to find $\text{moves}(n)$, i.e. recursion with subproblem reuse, is more generally applicable.

Exercise

What is the time complexity of calculating $\text{moves}(n)$ using the closed form?

Hint

Recall the portfolio task *Fast Exponentiation*.

What is dynamic programming?

The main idea is to solve a large problem recursively by building from (carefully chosen) subproblems of smaller size.

Substructure property

We must choose subproblems so that

> *answers to smaller subproblems (child instances) can be combined to answer a larger subproblem (parent instance).*

Why is dynamic programming useful?

- Recently we discussed greedy algorithms, where the problem is viewed as a sequence of stages and we consider *only* the locally optimal choice at each stage.

- We saw that some greedy algorithms are incorrect, i.e. they fail to construct a globally optimal solution.

- Also, greedy algorithms are unhelpful for certain types of problems, such as enumeration (“count the number of ways to . . .”).

- Dynamic programming can be used to consider *all* the options at each stage *without* repeating work; “efficient brute force”.

Why is dynamic programming efficient?

- We have already seen one problem-solving paradigm that used recursion: divide-and-conquer.

- D&C aims to break a large problem into *disjoint* subproblems, solve those subproblems recursively and recombine.

- However, DP is characterised by *overlapping subproblems*.

Why is dynamic programming efficient?

Overlapping subproblems property

We must choose subproblems so that

> *the same subproblem occurs several times in the recursion tree.*

When we solve a subproblem, we *store the result* so that subsequent instances of the same subproblem can be answered by just looking up a value in a table.

The parts of a dynamic programming algorithm

- A dynamic programming algorithm consists of three parts:
  - a 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 the trivial subproblems - those for which the recurrence is not required.

Executing a dynamic programming algorithm

- We will always talk about dynamic programming algorithms in an *iterative* (bottom-up) sense:
    - start with the base cases - the “smallest” subproblems, then
    - gradually solve increasingly “large” subproblems using the recurrence.
- Answers to subproblems will be stored in a lookup table (usually an array indexed by the parameters of the subproblem).
- When you implement dynamic programming algorithms using a programming language, you can consider *recursive* (top-down) implementation as an alternative.

Putting it all together

- We need to solve the subproblems in an *order* that respects the dependencies; more on this later.

- The original problem may be one of our subproblems, or it may be solved by combining results from several subproblems, in which case we must also describe this process.

- Finally, we should be aware of the time complexity of our algorithm, which is usually given by multiplying the *number of subproblems* by the *'average' time taken to solve a subproblem using the recurrence*.

Correctness

- To justify the correctness of a dynamic programming algorithm, we need to provide brief justification that
  - the recurrence correctly relates subproblems to each other,
  - the base cases have been correctly solved (usually straightforward)
  - the overall answer has been correctly solved (often straightforward).

How to develop dynamic programming algorithms

- Choose an order to build up the solution.
- Pick a tentative subproblem specification, starting with just enough parameters to answer the overall problem.
    - e.g. if you need to output the best value using up to k items, then number of items taken so far should be a parameter.
- Try to make a recurrence between these subproblems.
    - If you can't determine which transitions are allowed using only these parameters, try adding more, and repeat until it stabilises.
    - Solve the necessary base cases (usually easy).
- Analyse time complexity.
    - Removing unnecessary parameters may reduce the number of subproblems.
    - Use of data structures may speed up the recurrence.

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Hopscotch

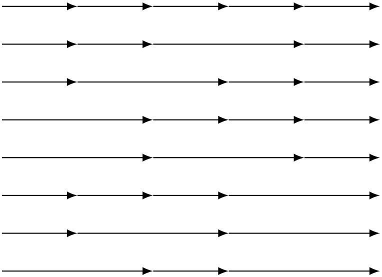
Problem

Instance: 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.

Task: Design a linear time algorithm to count the number of ways that Rui can reach square *n*.

Hopscotch



0	1	2	3	4	5

Hopscotch

- Choose a tentative subproblem specification.
  - For the full problem, we need the number of ways of reaching the last square, so let's associate to each square the number of ways of reaching it, say $\text{num}(i)$.
- Try to form a recurrence between these subproblems.
  - Square i can be reached by:
    - getting to square $i-1$ in any sequence of moves, then taking a step, or
    - getting to square $i-2$ in any sequence of moves, then taking a jump.
  - Therefore $\text{num}(i) = \text{num}(i-1) + \text{num}(i-2)$ for $i \geq 2$.
  - Base cases are $\text{num}(0)$ and $\text{num}(1)$, which we can quickly count in full.
- $O(n)$ subproblems each taking constant time, total time is linear.

Solution

Subproblems: for each $0 \leq i \leq n$, let $P(i)$ be the problem of determining $\text{num}(i)$, the number of ways to reach square i.

Recurrence: for $i \geq 2$,

$$\text{num}(i) = \text{num}(i-1) + \text{num}(i-2),$$

because square i can be reached by:

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

Base case: $\text{num}(0) = 1$ because the only way to reach square 0 is to do nothing, and $\text{num}(1) = 1$ because the only way to reach square 1 is to take a single step.

Hopscotch

Solution (continued)

Order of computation: solve subproblems $P(i)$ in increasing order of i.

Overall answer: $\mathrm{num}(n)$.

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

Hopscotch

i	0	1	2	3	4	5
$\text{num}(i)$	1	1	2	3	5	8

$$\text{num}(i) = \text{num}(i-1) + \text{num}(i-2)$$

Hopscotch

Remark

You might recognise that this is the famous *Fibonacci sequence*.

The closed form is even less useful in this example because it uses irrational numbers.

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Longest Increasing Subsequence

Problem

Instance: a sequence of n real numbers $A[1..n]$.

Task: determine a subsequence (not necessarily contiguous) of maximum length, in which the values in the subsequence are strictly increasing.

Longest Increasing Subsequence

- A natural choice for the subproblems is as follows: for each $1 \leq i \leq n$, let $P(i)$ be the problem of determining the length of the longest increasing subsequence of $A[1..i]$.

- However, it is not immediately obvious how to relate these subproblems to each other.

- A more convenient specification involves $Q(i)$, the problem of determining $\text{opt}(i)$, the length of the longest increasing subsequence of $A[1..i]$ *ending at* the last element $A[i]$.

- Note that the overall solution is recovered by taking the best of the answers to all the subproblems, i.e. the longest increasing subsequence ending at *any* index.

Longest Increasing Subsequence

- We will try to solve $Q(i)$ by extending the sequence which solves $Q(j)$ for some $j < i$.

- Supposing we have already solved all of these earlier subproblems, we now look for all indices $j < i$ such that $A[j] < A[i]$.

- Among those we pick m so that $\text{opt}(m)$ is maximal, and extend that sequence with $A[i]$.

- This forms the basis of our recurrence!

- The recurrence is not necessary if $i = 1$, as there are no previous indices to consider, so this is our base case.

Longest Increasing Subsequence

Solution

Subproblems: for each $1 \leq i \leq n$, let $Q(i)$ be the problem of determining $\text{opt}(i)$, the maximum length of an increasing subsequence of $A[1..i]$ which ends with $A[i]$.

Recurrence: for $i > 1$,

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

Base case: $\text{opt}(1) = 1$.

Longest Increasing Subsequence

i	1	2	3	4	5	6	7	8
$A[i]$	1	5	3	6	2	7	4	8
$\mathrm{opt}(i)$	1	2	2	3	2	4	3	5

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

Longest Increasing Subsequence

Solution (continued)

Order of computation: solve subproblems $Q(i)$ in increasing order of i.

Overall answer: the overall LIS is the best of those ending at some element, i.e. $\max_{1 \leq i \leq n} \operatorname{opt}(i)$.

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

Longest Increasing Subsequence

- Why does this produce optimal solutions to subproblems? We can use an inductive argument.
    - Base cases are trivial.
- We claim that if a sequence of indices $[\ldots, m, i]$ is the LIS ending at index i, then omitting the last entry must leave a LIS ending at index m.
    - In other words, the truncation of an optimal solution for $Q(i)$ must be an optimal solution of some earlier subproblem $Q(m)$.
- This claim is easily proven by contradiction.
    - 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.

Longest Increasing Subsequence

- What if the problem asked for not only the length, but the entire longest increasing subsequence?
- This is a common extension to such problems, and is easily handled.
- In the i^{th} slot of the table, alongside $\text{opt}(i)$ 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 and memory used by the algorithm.

Longest Increasing Subsequence

i	1	2	3	4	5	6	7	8
$A[i]$	1	5	3	6	2	7	4	8
opt(i)	1	2	2	3	2	4	3	5
pred(i)		1	1	2	1	4	3	6

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

Longest Increasing Subsequence

Exercise

Design an algorithm which solves this problem and runs in time $O(n \log n)$.

Activity Selection

Problem

Instance: A list of n activities with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find the *maximal total duration* of a subset of compatible activities.

Activity Selection

- Remember, we used the greedy method to solve a somewhat similar problem of finding a subset with the *largest possible number* of compatible activities, but the greedy method *does not* work for the present problem.

- As before, we start by sorting the activities by their finishing time into a non-decreasing sequence, and henceforth we will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Activity Selection

- We can then specify the subproblems: for each $1 \leq i \leq n$, let $P(i)$ be the problem of finding the duration $t(i)$ of a subsequence $\sigma(i)$ of the first i activities which

  1. consists of non-overlapping activities,
  2. ends with activity i, and
  3. is of maximal total duration among all such sequences.

- As in the previous problem, the second condition will simplify the recurrence.

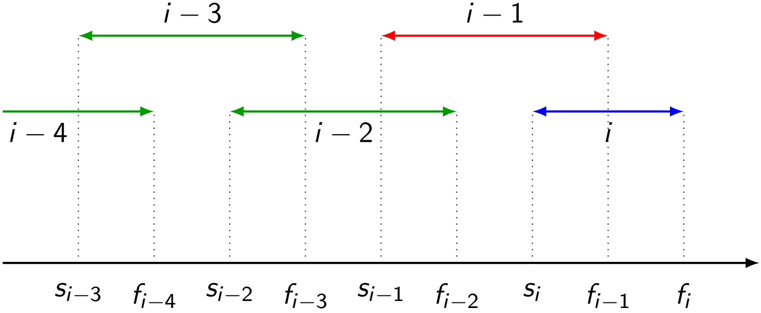
Activity Selection

- We would like to solve $P(i)$ by appending activity i to $\sigma(j)$ for some $j < i$.

- We require that activity i not overlap with activity j, i.e. the latter finishes before the former begins.

- Among all such j, our recurrence will choose that which maximises the duration $t(j)$.

- There is no need to solve $P(1)$ in this way, as there are no preceding activities.


$i-3$
$i-1$
$i-4$
$i-2$
i
s_{i-3}
f_{i-4}
s_{i-2}
f_{i-3}
s_{i-1}
f_{i-2}
s_i
f_{i-1}
f_i

Activity Selection

Solution

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 activities which ends with activity i.

Recurrence: for $i > 1$,

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

Base Case: $t(1) = f_1 - s_1$.

Solution (continued)

Order of computation: solve subproblems $P(i)$ in increasing order of i.

Overall answer: again, considering all possible choices for the last activity selected, we have

$$\max_{1 \leq i \leq n} t(i).$$

Time complexity:

- Sorting takes $O(n \log n)$.
- There are $O(n)$ subproblems each taking $O(n)$.
- The overall answer is found in $O(n)$.

Therefore the time complexity is $O(n^2)$.

- Why does this recurrence produce optimal solutions to subproblems $P(i)$? We apply the same inductive argument!
- Let the optimal solution of subproblem $P(i)$ be given by the sequence of activities $\sigma = [\ldots, m, i]$.
- We claim: the truncated sequence $\sigma' = [\ldots, m]$ gives an optimal solution to subproblem $P(m)$.
- Why is this true? Consider the opposite!

Activity Selection

- Suppose instead that $P(m)$ is solved by a sequence τ' of even larger total duration.

- Then make an activity selection τ by appending activity i to τ'.

- It is clear this is a valid selection of activities with larger total duration than σ. This contradicts the earlier definition of σ as the sequence solving $P(i)$.

- Thus, the optimal sequence for problem $P(i)$ is obtained by extending the optimal sequence for some earlier subproblem with activity i.

Activity Selection

- Suppose we also want to construct the optimal sequence of activities.

- In the i^{th} slot of our table, we should store not only $t(i)$ but the index m from which the optimal solution of $P(i)$ was constructed.

Making Change

Problem

Instance: You are given n types of coin denominations of integer values $v_1 < v_2 < \ldots < v_n$.

Assume $v_1 = 1$ (so that you can always make change for any integer amount) and that you have an unlimited supply of coins of each denomination.

Task: make change for a given integer amount C, using as few coins as possible.

Making Change

Attempt 1

Greedily take as many coins of value v_m as possible, then v_{m-1}, and so on.

- This approach is very tempting, and works for almost all real-world currencies.
- However, it doesn't work for all sequences v_i. In general, we will need to use DP.

Exercise

Design a counterexample to the above algorithm.

Making Change

- 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.

Making Change

- Among all of 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 $\mathrm{opt}(i)$ for amount i by adding one coin of denomination v_m to $\mathrm{opt}(i - v_m)$.

- If $C = 0$ the solution is trivial: use no coins.

Making Change

Solution

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

Recurrence: for $i > 0$,

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

Base case: $\text{opt}(0) = 0$.

Making Change

Solution

Order of computation: solve subproblems $P(i)$ in increasing order of i.

Overall answer: $\text{opt}(C)$.

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

Making Change

Note

In the module on Intractable Problems, we'll see that this is considered to be a *pseudopolynomial* time algorithm, *not* a polynomial time algorithm.

We always determine whether an algorithm runs in polynomial time with reference to the *length* of the input.

- In this case, the input is n, the coin values and C, so the number of bits required to communicate it is $O(n \log C)$.
- On the other hand, the algorithm runs in $O(nC)$, so the running time is *exponential* in the length of the input.

There is no known polynomial time algorithm for this problem!

Making Change

- Why does this produce an optimal solution for each amount $i \leq C$?

- Consider an optimal solution for some amount i, and say this solution includes at least one coin of denomination v_m for some $1 \leq m \leq n$.

- Removing this coin must leave an optimal solution for the amount $i - v_m$, again proven by contradiction.

- By considering all coins of value at most i, we can pick m for which the optimal solution for amount $i - v_m$ uses the fewest coins.

Making Change

- Suppose we were required to also determine the exact number of each coin required to make change for amount C.
- In the i^{th} slot of the table, we would store both $\text{opt}(i)$ and the coin type $k = \text{pred}(i)$ which minimises $\text{opt}(i - v_k)$.
- Then $\text{pred}(C)$ is a coin used in the optimal solution for total C, leaving $C' = C - \text{pred}(C)$ remaining. We then repeat, identifying another coin $\text{pred}(C')$ used in the optimal solution for total C', and so on.

Notation

We denote the k that minimises $\text{opt}(i - v_k)$ by

$$\operatorname*{argmin}_{1 \le k \le n} \text{opt}(i - v_k).$$

Integer Knapsack

Problem

Instance: 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.

Task: Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

Integer Knapsack

- As in previous problem, we solve for each total weight up to C.
- Assume we have solved the problem for all total weights $j < i$.
- We now consider each item, the kth of which has weight w_k. If this item is included, we would have $i - w_k$ weight to fill.
- For whichever m maximises the total value of the optimal solution for $i - w_m$, we choose
  - the optimal packing of weight $i - w_m$ (an earlier subproblem), and
  - one more of item m,

  to obtain a packing of total weight i of the highest possible value.

Integer Knapsack

Solution

Subproblems: for each $0 \leq i \leq C$, let $P(i)$ be the problem of determining

- $\text{opt}(i)$, 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 $i > 0$,

$$\begin{aligned} \text{opt}(i) &= \max_{1 \leq k \leq n, w_k \leq i} \text{opt}\left(i - w_k\right) + v_k \\ m(i) &= \operatorname*{argmax}_{1 \leq k \leq n, w_k \leq i} \left(\text{opt}(i - w_k) + v_k\right). \end{aligned}$$

Solution (continued)

Base case: if $i < \min_{1 \le k \le n} w_k$, then $\text{opt}(i) = 0$ and $m(i) = 0$.

- Setting $m(i) = 0$ employs a dummy value to record that no item was taken.

Order of computation: solve subproblems $P(i)$ in increasing order of i.

Overall answer: $\text{opt}(C)$, as the knapsack can hold *up to C* units of weight.

Time complexity: $O(C)$ subproblems taking $O(n)$, for a time complexity of $O(nC)$; again pseudopolynomial.

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

0-1 Knapsack

Problem

Instance: 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.

Task: Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

0-1 Knapsack

- Let's use the same subproblems as before, and try to develop a recurrence.

Question

If we know the optimal solution for each total weight $j < i$, can we deduce the optimal solution for weight i?

Answer

No! 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.

0-1 Knapsack

- At this point you may object that we can in fact reconstruct the optimal solution for total weight $i - w_k$ by storing the values $m(i)$ as we did earlier, and then backtrack.
- This unfortunately has two flaws:
  1. the optimal solution for $i - w_k$ is not necessarily unique, so we may have recorded a selection including item k when a selection of equal value without item k also exists, and
  2. if all optimal solutions for $i - w_k$ use item k, it is still possible that the best solution for i combines item k with some suboptimal solution for $i - w_k$.
- The underlying issue is that with this choice of subproblems, this problem does not have the *optimal substructure property*.

0-1 Knapsack

- When we are unable to form a correct recurrence between our chosen subproblems, this usually indicates that the subproblem specification is inadequate.

- What extra parameters are required in our subproblem specification?

- We would like to know the optimal solution for each weight without using item k.

- Directly adding this information to the subproblem specification still doesn't lead to a useful recurrence. How could we capture it less directly?

0-1 Knapsack

- For each total weight i, we will find the optimal solution using only the first k items.

- We can take cases on whether item k is used in the solution:

  - if so, we have $i - w_k$ remaining weight to fill using the first $k - 1$ items, and

  - otherwise, we must fill all i units of weight with the first $k - 1$ items.

0-1 Knapsack

Solution

Subproblems: for $0 \leq i \leq C$ and $0 \leq k \leq n$, let $P(i, k)$ be the problem of determining

- $\text{opt}(i, k)$, 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 (largest) index of an item in such a collection.

0-1 Knapsack

Solution (continued)

Recurrence: for $i > 0$ and $1 \leq k \leq n$,

$$\operatorname{opt}(i, k) = \max\left(\operatorname{opt}(i, k-1), \operatorname{opt}(i - w_k, k-1) + v_k\right),$$

with $m(i, k) = m(i, k-1)$ in the first case and k in the second.

Base cases: if $i = 0$ or $k = 0$, then $\operatorname{opt}(i, k) = 0$ and $m(i, k) = 0$.

- Setting $m(i, k) = 0$ employs a dummy value to record that no item was taken.

0-1 Knapsack

Solution (continued)

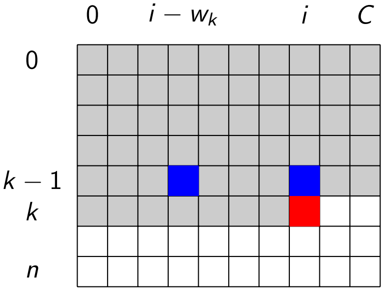
Order of computation:

- 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 (and any order of i).

Overall answer: $\text{opt}(C, n)$.

Time complexity: $O(nC)$ subproblems taking $O(1)$ time, for a total of $O(nC)$.

0-1 Knapsack



0-1 Knapsack

Exercise

Modify the algorithm for 0-1 Knapsack to instead solve Integer Knapsack.

Balanced Partition

Problem

Instance: a set of n positive integers $x_1, \ldots, x_n$.

Task: partition these integers into two subsets S_1 and S_2 with sums Σ_1 and Σ_2 respectively, so as to minimise $|\Sigma_1 - \Sigma_2|$.

Balanced Partition

- Suppose without loss of generality that $\Sigma_1 \geq \Sigma_2$.

- Let $\Sigma = x_1 + \ldots + x_n$, the sum of all integers in the set.

- Observe that $\Sigma_1 + \Sigma_2 = \Sigma$, which is a constant, and upon rearranging it follows that

$$\Sigma_1 - \Sigma_2 = 2\left(\frac{\Sigma}{2} - \Sigma_2\right).$$

- So, all we have to do is find a subset S_2 of these numbers with total sum as close to $\Sigma/2$ as possible, but not exceeding it.

Balanced Partition

- For each integer x_i in the set, construct an item with both weight and value equal to x_i.
- Consider the knapsack problem (with duplicate items not allowed), with items as specified above and knapsack capacity $\Sigma/2$.

Solution

The best packing of this knapsack produces an optimally balanced partition, with set S_1 given by the items outside the knapsack and set S_2 given by the items in the knapsack.

Longest Common Subsequence

Problem

Instance: two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Task: find the length of a longest common subsequence of S, S^*.

Longest Common Subsequence

- A sequence s is a *subsequence* of another sequence S if s can be obtained by deleting some of the symbols of S (while preserving the order of the remaining symbols).

- Given two sequences S and S^* a sequence s is a *Longest Common Subsequence* of S, S^* if s is a subsequence of both S and S^* and is of maximal possible length.

- This can be useful as a measurement of the similarity between S and S^*.

- Example: how similar are the genetic codes of two viruses? Is one of them just a genetic mutation of the other?

Longest Common Subsequence

- A natural choice of subproblems considers prefixes of both sequences, say

$$S_i = \langle a_1, a_2, \ldots, a_i \rangle \text{ and } S_j^* = \langle b_1, b_2, \ldots, b_j \rangle.$$

- If a_i and b_j are the same symbol (say c), the longest common subsequence of S_i and S_j^* is formed by appending c to the solution for S_{i-1} and S_{j-1}^*.

- Otherwise, a common subsequence of S_i and S_j^* cannot contain both a_i and b_j, so we consider discarding either of these symbols.

- No recursion is necessary when either S_i or S_j^* are empty.

Longest Common Subsequence

Solution

Subproblems: for all $0 \leq i \leq n$ and all $0 \leq j \leq m$ let $P(i, j)$ be the problem of determining $\text{opt}(i, j)$, the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \ldots a_i \rangle$ and $S_j^* = \langle b_1, b_2, \ldots, b_j \rangle$.

Recurrence: for all $i, j > 0$,

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

Base cases:

- for all $0 \leq i \leq n$, $\text{opt}(i, 0) = 0$, and
- for all $0 \leq j \leq m$, $\text{opt}(0, j) = 0$.

Longest Common Subsequence

Solution (continued)

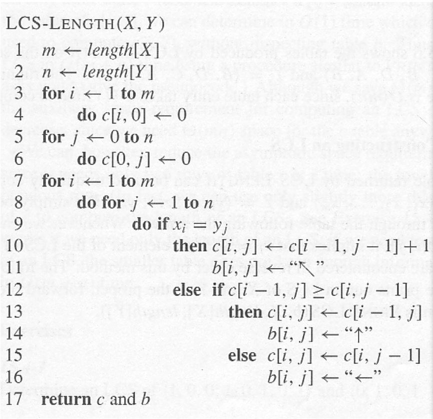
Order of computation: solve the subproblems $P(i, j)$ in lexicographic order (increasing i, then increasing j) to guarantee that $P(i-1, j)$, $P(i, j-1)$ and $P(i-1, j-1)$ are solved before $P(i, j)$, so all dependencies are satisfied.

Overall answer: $\text{opt}(n, m)$.

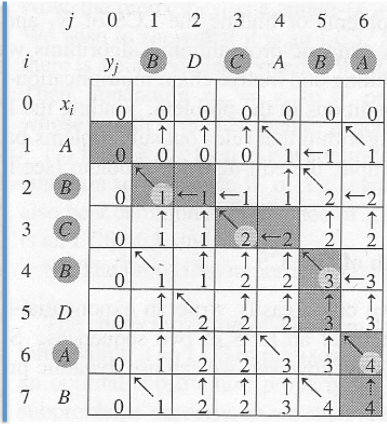
Time complexity: $O(nm)$ subproblems taking $O(1)$ time, for a total of $O(nm)$.

Longest Common Subsequence

To reconstruct the longest common subsequence itself, we can record the direction from which the value opt(i, j) was obtained in the table, and backtrack.



```
LCS-LENGTH(X, Y)
  m ← length[X]
  n ← length[Y]
  for i ← 1 to m
      do c[i, 0] ← 0
  for j ← 0 to n
      do c[0, j] ← 0
  for i ← 1 to m
      do for j ← 1 to n
             do if x_i = y_j
                   then c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← "↖"
                   else if c[i − 1, j] ≥ c[i, j − 1]
                           then c[i, j] ← c[i − 1, j]
                                b[i, j] ← "↑"
                           else c[i, j] ← c[i, j − 1]
                                b[i, j] ← "←"
  return c and b
```



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S, S^*, S^{**}?

Question

Can we do $\text{LCS}\left(\text{LCS}\left(S, S^*\right), S^{**}\right)$?

Answer

Not necessarily!

Longest Common Subsequence

- Let $S = ABCDEGG$, $S^* = ACBEEFG$ and $S^{**} = ACCEDGF$. Then
$$\operatorname{LCS}(S, S^*, S^{**}) = ACEG.$$

- However,

$$\begin{aligned}
&\operatorname{LCS}(\operatorname{LCS}(S, S^*), S^{**}) \\
&= \operatorname{LCS}(\operatorname{LCS}(ABCDEGG, ACBEEFG), S^{**}) \\
&= \operatorname{LCS}(ABEG, ACCEDGF) \\
&= AEG.
\end{aligned}$$

Exercise

Confirm that $\operatorname{LCS}(\operatorname{LCS}(S^*, S^{**}), S)$ and $\operatorname{LCS}(\operatorname{LCS}(S, S^{**}), S^*)$ also give wrong answers.

Longest Common Subsequence

Problem

Instance: three sequences $S = \langle a_1, a_2, \ldots a_n \rangle$, $S^* = \langle b_1, b_2, \ldots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \ldots, c_l \rangle$.

Task: find the length of a longest common subsequence of S, S^* and S^{**}.

Longest Common Subsequence

Solution

Subproblems: for all $0 \leq i \leq n$, all $0 \leq j \leq m$ and all $0 \leq k \leq l$, let $P(i, j, k)$ be the problem of determining $\operatorname{opt}(i, j, k)$, the length of the longest common subsequence of the truncated sequences $S_i = \langle a_i, a_2, \dots, a_i \rangle$, $S^*_j = \langle b_1, b_2, \dots, b_j \rangle$ and $S^{**}_k = \langle c_1, c_2, \dots, c_k \rangle$.

Recurrence: for all $i, j, k > 0$,

$$\operatorname{opt}(i, j, k) = \begin{cases} \operatorname{opt}(i-1, j-1, k-1) + 1 & \text{if } a_i = b_j = c_k \\ \max \begin{pmatrix} \operatorname{opt}(i-1, j, k), \\ \operatorname{opt}(i, j-1, k), \\ \operatorname{opt}(i, j, k-1) \end{pmatrix} & \text{otherwise.} \end{cases}$$

Base cases: if $i = 0$, $j = 0$ or $k = 0$, $\operatorname{opt}(i, j, k) = 0$.

Solution (continued)

Order of computation: solve the subproblems $P(i, j, k)$ in lexicographic order (increasing i, then increasing j, then increasing k) to guarantee that $P(i-1, j, k)$, $P(i, j-1, k)$, $P(i, j, k-1)$ and $P(i-1, j-1, k-1)$ are solved before $P(i, j, k)$, so all dependencies are satisfied.

Overall answer: $\text{opt}(n, m, l)$.

Time complexity: $O(nml)$ subproblems taking $O(1)$, for a total of $O(nml)$.

To reconstruct the longest common subsequence itself, we can record the direction from which the value $\text{opt}(i, j, k)$ was obtained in the table, and backtrack.

Shortest Common Supersequence

Problem

Instance: two sequences $s = \langle a_1, a_2, \ldots a_n \rangle$ and $s^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Task: find a shortest common supersequence S of s, s^*, i.e., a shortest possible sequence S such that both s and s^* are subsequences of S.

Shortest Common Supersequence

Solution

Find a longest common subsequence $\mathrm{LCS}(s, s^*)$ of s and s^*, then add back the differing elements of the two sequences in the right places, in any compatible order.

Example

If

$$s = abacada \text{ and } s^* = xbycazd,$$

then

$$\mathrm{LCS}(s, s^*) = bcad$$

and therefore

$$\mathrm{SCS}(s, s^*) = axbyacazda.$$

Edit Distance

Problem

Instance: Given two text strings A of length n and B of length m, you want to transform A into B.

You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_I, a deletion costs c_D and a replacement costs c_R.

Task: find the lowest total cost transformation of A into B.

Edit Distance

- Edit distance is another measure of the similarity of pairs of strings.

- Note: if all operations have a unit cost, then you are looking for the minimal number of such operations required to transform A into B; this number is called the *Levenshtein distance* between A and B.

- If the sequences are sequences of DNA bases and the costs reflect the probabilities of the corresponding mutations, then the minimal cost represents how closely related the two sequences are.

Edit Distance

- Again we consider prefixes of both strings, say $A[1..i]$ and $B[1..j]$.

- We have the following options to transform $A[1..i]$ into $B[1..j]$:

  1. delete $A[i]$ and then transform $A[1..i-1]$ into $B[1..j]$;

  2. transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$;

  3. transform $A[1..i-1]$ to $B[1..j-1]$ and if necessary replace $A[i]$ by $B[j]$.

- If $i = 0$ or $j = 0$, we only insert or delete respectively.

Solution

Subproblems: for all $0 \leq i \leq n$ and $0 \leq j \leq m$, let $P(i, j)$ be the problem of determining $\text{opt}(i, j)$, the minimum cost of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$.

Recurrence: for $i, j \geq 1$,

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i-1, j) + c_D \\ \text{opt}(i, j-1) + c_I \\ \begin{cases} \text{opt}(i-1, j-1) & \text{if } A[i] = B[j] \\ \text{opt}(i-1, j-1) + c_R & \text{if } A[i] \neq B[j]. \end{cases} \end{cases}$$

Base cases: $\text{opt}(i, 0) = i\, c_D$ and $\text{opt}(0, j) = j\, c_I$.

Edit Distance

Solution (continued)

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.

Overall answer: $\text{opt}(n,m)$.

Time complexity: $O(nm)$ subproblems taking $O(1)$ time, for a total of $O(nm)$.

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Matrix chain multiplication

- Let A and B be matrices. The matrix product AB exists if A has as many columns as B has rows: if A is $m \times n$ and B is $n \times p$, then AB is $m \times p$.

- Each element of AB is the dot product of a row of A with a column of B, both of which have length n. Therefore $m \times n \times p$ multiplications are required to compute AB.

- Matrix multiplication is *associative*, that is, for any three matrices of compatible sizes we have $A(BC) = (AB)C$.

- However, the number of real number multiplications needed to obtain the product can be very different.

Matrix chain multiplication

Suppose A is 10×100, B is 100×5 and C is 5×50.

$$\begin{tikzcd}
& \underbrace{(AB)}_{10\times 5}(C) \arrow[dr, "10 \times 5 \times 50"] & \\
(A)(B)(C) \arrow[ur, "10 \times 100 \times 5"] \arrow[dr, "100 \times 5 \times 50"'] & & (ABC) \\
& (A)\ \underbrace{(BC)}_{100\times 50} \arrow[ur, "10 \times 100 \times 50"'] &
\end{tikzcd}$$

Evaluating $(AB)C$ involves only 7500 multiplications, but evaluating $A(BC)$ requires 75000 multiplications!

Matrix chain multiplication

Problem

Instance: a compatible sequence of matrices $A_1 A_2 ... A_n$, where A_i is of dimension $s_{i-1} \times s_i$.

Task: group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

Matrix chain multiplication

- How many groupings are there?

- The total number of different groupings satisfies the following recurrence (why?):

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i),$$

  with base case $T(1) = 1$.

- One can show that the solution satisfies $T(n) = \Omega(2^n)$.

- Thus, we cannot efficiently do an exhaustive search for the optimal grouping.

Matrix chain multiplication

Note

The number of groupings $T(n)$ is a very famous sequence: the *Catalan numbers*. This sequence answers many seemingly unrelated combinatorial problems, including:

- the number of balanced bracket sequences of length $2n$;
- the number of full binary trees with $n + 1$ leaves;
- the number of lattice paths from $(0, 0)$ to (n, n) which never go above the diagonal;
- the number of noncrossing partitions of an $n + 2$-sided convex polygon;
- the number of permutations of $\{1, \dots, n\}$ with no three-term increasing subsequence.

Matrix chain multiplication

- Instead, we try dynamic programming. A first attempt might be to specify subproblems corresponding to prefixes of the matrix chain, that is, find the optimal grouping for $A_1 A_2 \dots A_i$.

- This is not enough to construct a recurrence; consider for example splitting the chain as

$$(A_1 A_2 \dots A_j)(A_{j+1} A_{j+2} \dots A_i).$$

Matrix chain multiplication

- Instead we should specify a subproblem corresponding to each contiguous subsequence $A_{i+1}A_{i+2}\ldots A_j$ of the chain.
- The recurrence will consider all possible ways to place the outermost multiplication, splitting the chain into the product

$$\underbrace{(A_{i+1}\ldots A_k)}_{s_i \times s_k}\underbrace{(A_{k+1}\ldots A_j)}_{s_k \times s_j}.$$

  - If we wanted to recover the actual bracketing required, we could store alongside each value $\mathrm{opt}(i,j)$ the splitting point k used to obtain it.
- No recursion is necessary for subsequences of length one.

Matrix chain multiplication

Solution

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

Recurrence: for all $j - i > 1$,

$$\text{opt}(i,j) = \min_{i<k<j} \left[\text{opt}(i,k) + s_i s_k s_j + \text{opt}(k,j)\right].$$

Base cases: for all $0 \leq i \leq n-1$, $\text{opt}(i,i+1) = 0$.

Matrix chain multiplication

Solution (continued)

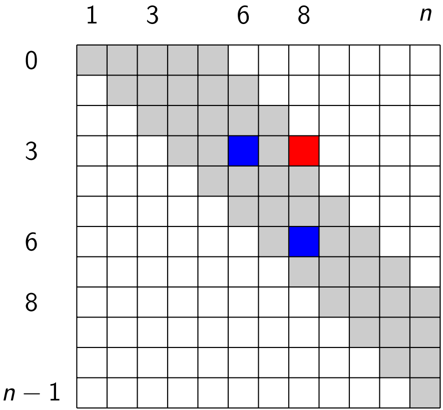
Order of computation:

- 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 $j - i$, i.e. subsequence length.

Overall answer: $\text{opt}(0,n)$.

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

Matrix chain multiplication



$$\mathrm{opt}(3,8) = \min_{3<k<8} [\mathrm{opt}(3,k) + s_3 s_k s_8 + \mathrm{opt}(k,8)]$$

Matrix chain multiplication

Exercise

Not all subproblems take the same amount of time to solve. For a subsequence of length l, only $l - 1$ potential splitting points are considered.

This raises the question of whether we can prove a tighter bound for the time complexity using amortisation.

Does this algorithm actually run in $o(n^3)$, i.e. asymptotically *strictly less than* cubic time? Justify your answer.

Maximising an expression

Problem

Instance: a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9.$$

Task: place brackets in a way that the resulting expression has the largest possible value.

Maximising an expression

- What will be the subproblems?

- Similar to the matrix chain multiplication problem earlier, it's not enough to just solve for prefixes $A[1..i]$.

- Maybe for a subsequence of numbers $A[i+1..j]$ place the brackets so that the resulting expression is maximised?

Maximising an expression

- How about the recurrence?

- It is natural to break down $A[i+1..j]$ into $A[i+1..k] \odot A[k+1..j]$, with cases $\odot = +, -, \times$.

- In the case $\odot = +$, we want to maximise the values over both $A[i+1..k]$ and $A[k+1..j]$.

- This doesn't work for the other two operations!

- We should look for placements of brackets not only for the maximal value but also for the minimal value!

Maximising an expression

Exercise

Write a complete solution for this problem.

Your solution should include the subproblem specification, recurrence and base cases.

You should also specify the order of computation, describe how the overall solution is to be obtained, and analyse the time complexity of the algorithm.

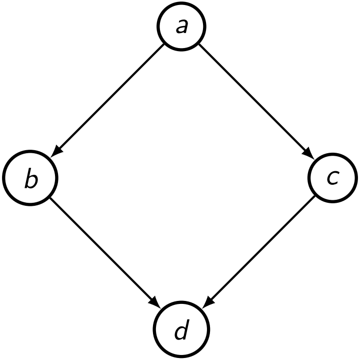
Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Directed acyclic graphs

Definition

Recall that a *directed acyclic graph* (DAG) is exactly that: a directed graph without (directed) cycles.



Topological ordering

Definition

Recall that in a directed graph, a *topological ordering* of the vertices is one in which all edges point “left to right”.

- A directed graph admits a topological ordering if and only if it is acyclic.
- There may be more than one valid topological ordering for a particular DAG.
- A topological ordering can be found in linear time, i.e. $O(|V| + |E|)$.

Shortest path in a directed acyclic graph

Problem

Instance: a directed acyclic graph $G = (V, E)$ in which each edge $e \in E$ has a corresponding weight $w(e)$ (which may be negative), and a designated vertex $s \in V$.

Task: find the shortest path from s to each vertex $t \in V$.

Notation

Let $n = |V|$ and $m = |E|$.

Shortest path in a directed acyclic graph

- If all edge weights are positive, the single source shortest path problem is solved by Dijkstra's algorithm in $O(m \log n)$.

- Later in this lecture, we'll see how to solve the general single source shortest path problem in $O(nm)$ using the Bellman-Ford algorithm.

- However, in the special case of directed acyclic graphs, a simple DP solves this problem in $O(n+m)$, i.e. linear time.

Shortest path in a directed acyclic graph

- 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 such v, plus the weight of the last edge, and selects the minimum of these options.
- The base case is s itself, where the shortest path is obviously zero.

Shortest path in a directed acyclic graph

Solution

Subproblems: for all $t \in V$, let $P(t)$ be the problem of determining $\text{opt}(t)$, the length of a shortest path from s to t.

Recurrence: for all $t \neq s$,

$$\text{opt}(t) = \min\{\text{opt}(v) + w(v, t) \mid (v, t) \in E\}.$$

Base case: $\text{opt}(s) = 0$.

Shortest path in a directed acyclic graph

Question

In what order should we solve the subproblems?

- In any DP algorithm, the recurrence introduces certain dependencies, and it is crucial that these dependencies are respected.
- Here, opt(t) depends on all the opt(v) values for 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.
- It is for this reason that this algorithm does not apply to general directed graphs!

Shortest path in a directed acyclic graph

Solution (continued)

Order of computation: topological order.

Overall answer: the list of values 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 (at its endpoint), so we can use the tighter bound $O(n + m)$.

DP on a directed acyclic graph

- Many problems on directed acyclic graphs can be solved in the same way: first use topological sort, then DP over the vertices in that order.

- If we replace the min in the earlier recurrence by max, we have an algorithm to find the longest path from s to each t. This problem is much harder on general graphs; indeed, there is no known algorithm to solve it in polynomial time.

- Often a graph will be specified in a way that makes it obviously acyclic, with a natural topological order.

Assembly line scheduling

Problem

Instance: You are given two assembly lines, each consisting of n workstations. The k^{th} workstation on each assembly line performs the k^{th} of n jobs.

- To bring a new product to the start of assembly line i takes s_i units of time.
- To retrieve a finished product from the end of assembly line i takes f_i units of time.

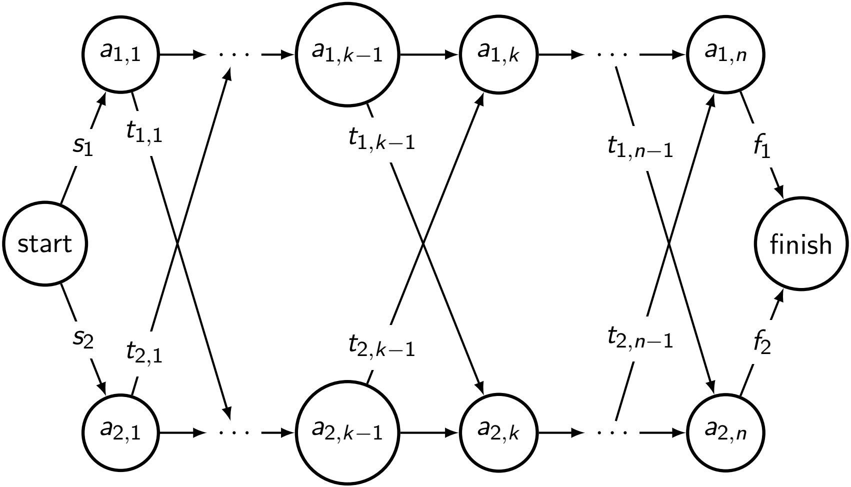
Assembly line scheduling

Problem (continued)

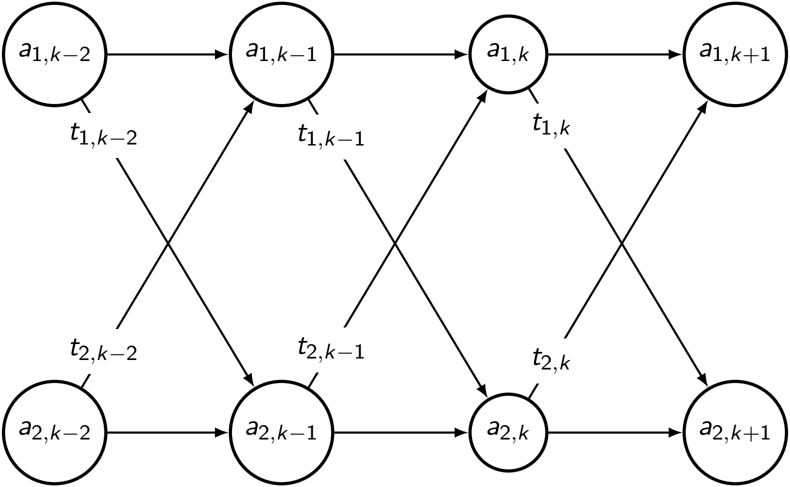
- On assembly line i, the k^{th} job takes $a_{i,k}$ units of time to complete.
- To move the product from station k on assembly line i to station $k+1$ on the other line takes $t_{i,k}$ units of time.
- There is no time required to continue from station k to station $k+1$ on the same line.

Task: Find a *fastest way* to assemble a product using both lines as necessary.

Assembly line scheduling



Assembly line scheduling



Assembly line scheduling

- We will denote internal vertices using the form (i, k) to represent workstation k on assembly line i.
- The problem requires us to find the shortest path from the start node to the finish node, where unlabelled edges have zero weight.
- This is clearly a directed acyclic graph, and moreover the topological ordering is obvious:

  $$\text{start}, (1,1), (2,1), (1,2), (2,2), \ldots, (1,n), (2,n), \text{finish}.$$

  So we can use DP!
- There are $2n + 2$ vertices and $4n$ edges, so the DP should take $O(n)$ time, whereas Dijkstra's algorithm would take $O(n \log n)$.

Assembly line scheduling

- We'll solve for the shortest path from s to each vertex (i, k).
- To form a recurrence, we should consider the ways of getting to workstation k on assembly line i.
- We could have come from workstation $k - 1$ on either line, after completing the previous job.
- The exception is the first workstation, which leads to the base case.

Assembly line scheduling

Solution

Subproblems: for $i \in \{1, 2\}$ and $1 \leq k \leq n$, let $P(i, k)$ be the problem of determining $\text{opt}(i, k)$, the minimal time taken to complete the first k jobs, with the k^{th} job performed on assembly line i.

Recurrence: for $k > 1$,

$$\begin{aligned}
\text{opt}(1, k) &= \min\left(\text{opt}(1, k-1), \text{opt}(2, k-1) + t_{2,k-1}\right) + a_{1,k} \\
\text{opt}(2, k) &= \min\left(\text{opt}(2, k-1), \text{opt}(1, k-1) + t_{1,k-1}\right) + a_{2,k}.
\end{aligned}$$

Base cases: $\text{opt}(1, 1) = s_1 + a_{1,1}$ and $\text{opt}(2, 1) = s_2 + a_{2,1}$.

Solution (continued)

Order of computation: as the recurrence uses values from both assembly lines, we have to solve the subproblems in order of increasing k, solving both $P(1, k)$ and $P(2, k)$ at each stage.

Overall answer: after obtaining $\text{opt}(1, n)$ and $\text{opt}(2, n)$, the overall solution is given by

$$\min\left(\text{opt}(1, n) + f_1, \text{opt}(2, n) + f_2\right).$$

Time complexity:

- Each of $2n$ subproblems is solved in constant time.
- The overall answer is computed in constant time also.

Therefore the overall time complexity is $O(n)$.

Assembly line scheduling

Remark

This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc.

This will be covered in COMP4121 Advanced Algorithms.

Single Source Shortest Paths

Problem

Instance: 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 $s \in V$.

Task: find the weight of the shortest path from vertex s to every other vertex t.

Notation

Let $n = |V|$ and $m = |E|$.

Single Source Shortest Paths

- How does this problem differ to the one solved by Dijkstra's algorithm?

- In this problem, we allow negative edge weights, so the greedy strategy no longer works.

- Note that we disallow cycles of negative total weight. This is only because with such a cycle, there is no shortest path - you can take as many laps around a negative cycle as you like.

- This problem was first solved by Shimbel in 1955, and was one of the earliest uses of Dynamic Programming.

Bellman-Ford algorithm

Observation

For any vertex t, there is a shortest $s - t$ path without cycles.

Proof Outline

Suppose the opposite. Let p be a shortest $s - t$ path, so it must contain a cycle. Since there are no negative weight cycles, removing this cycle produces an $s - t$ path of no greater length.

Observation

It follows that every shortest $s - t$ path contains any vertex v at most once, and therefore has at most $n - 1$ edges.

Bellman-Ford algorithm

- For every vertex t, let's 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

$$p = \underbrace{s \to \ldots \to v}_{p'} \to t,$$

with the final edge going from v to t.

- Then p' must be itself the shortest path from s to v of at most $i - 1$ edges, which is another subproblem!

- No such recursion is necessary if $t = s$, or if $i = 0$.

Bellman-Ford algorithm

Solution

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

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

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

Base cases: $\mathrm{opt}(i, s) = 0$, and for $t \neq s$, $\mathrm{opt}(0, t) = \infty$.

Bellman-Ford algorithm

Solution (continued)

Order of computation: solve subproblems $P(i, t)$ in increasing order of i (and any order of t) since $P(i, t)$ depends only on $P(i-1, v)$ for various v.

Overall answer: the list of values $\text{opt}(n-1, t)$ over all vertices t.

Time complexity:

- There are n rounds: $i = 0, 1, \ldots, n-1$.
- In each round, each edge (v, t) of the graph is considered only once.

Therefore the time complexity is $O(nm)$.

Bellman-Ford algorithm

- How do we reconstruct an actual shortest $s - t$ path?

- As usual, we'll store one step at a time and backtrack. Let $\text{pred}(i, t)$ be the immediate predecessor of vertex t on a shortest $s - t$ path of at most i edges.

- The additional recurrence required is

$$\text{pred}(i, t) = \operatorname*{argmin}_{v:(v,t)\in E} \left[\text{opt}(i-1, v) + w(v, t)\right].$$

Bellman-Ford algorithm

- There are several small improvements that can be made to this algorithm.

- As stated, we build a table of size $O(n^2)$, with a new row for each 'round'.

- It is possible to reduce this to $O(n)$. Including $\text{opt}(i-1, t)$ as a candidate for $\text{opt}(i, t)$ doesn't change the correctness of the recurrence, so we can instead maintain a table with only one row, and overwrite it at each round.

Bellman-Ford algorithm

- The SPFA (Shortest Paths Faster Algorithm) speeds up the later rounds by ignoring some edges. This optimisation and others (e.g. early exit) do not change the worst case time complexity from $O(nm)$, but they can be a significant speed-up in practical applications.

- The Bellman-Ford algorithm can also be augmented to detect cycles of negative weight.

All Pairs Shortest Paths

Problem

Instance: a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight.

Task: find the weight of the shortest path from every vertex s to every other vertex t.

Notation

Let $n = |V|$ and $m = |E|$.

Floyd-Warshall algorithm

- We can use a similar idea, this time in terms of the intermediate vertices allowed on an $s - t$ path.

- Label the vertices of V as $v_1, v_2, \ldots, v_n$.

- Let S be the set of vertices allowed as intermediate vertices. Initially S is empty, and we add vertices $v_1, v_2, \ldots, v_n$ one at a time.

Floyd-Warshall algorithm

Question

When is the shortest path from s to t using the first k vertices as intermediates actually an improvement on the value from the previous round?

Answer

When there is a shorter path of the form

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

Floyd-Warshall algorithm

Solution

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 $\text{opt}(i, j, k)$, the weight of a shortest path from v_i to v_j using only $v_1, \ldots, v_k$ as intermediate vertices.

Recurrence: for all $1 \leq i, j, k \leq n$,

$$\text{opt}(i, j, k) = \min(\text{opt}(i, j, k-1), \text{opt}(i, k, k-1) + \text{opt}(k, j, k-1)).$$

Base cases:

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

Floyd-Warshall algorithm

Solution (continued)

Order of computation: solve subproblems in increasing order of k (and any order of i and j) since $P(i,j,k)$ depends only on $P(i,j,k-1)$, $P(i,k,k-1)$ and $P(k,j,k-1)$.

Overall answer: the table of values $\text{opt}(i,j,n)$ (where all vertices are allowed as intermediates), over pairs of vertices i and j.

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

The space complexity can again be improved by overwriting the table every round.

Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

String matching problem

Suppose you have an alphabet consisting of d symbols. Strings are just sequences of these symbols.

Problem

Determine whether a string $B = b_1 b_2 \dots b_m$ (the *pattern*) appears as a (contiguous) substring of a much longer string $A = a_1 a_2 \dots a_n$ (the *text*).

String matching problem

Applications:

- Check whether a word appears in a document.
- Check if a particular sequence of amino acids exists in a protein.
- Check if a DNA sequence contains a particular gene.

String matching problem

Attempt 1

The “naïve” algorithm for string matching is as follows: for every position i in A, check $a_i \ldots a_{i+m-1}$ against B character-by-character.

Even with early exit, this runs in $O(mn)$ time in the worst case.

Question

Can we do better?

String matching with hashing

- Why was the naïve algorithm slow?
  - Effectively $O(n)$ string comparisons, each taking $O(m)$ time in the worst case.
- How could we speed up the string comparisons? Use hashing!
  - We can compare hash values in constant time.
  - How to *compute* hash values of strings quickly?
  - What to do about collisions?

String matching with hashing

- We now show how hashing can be combined with recursion to produce an efficient string matching algorithm.

- We compute a hash value for the string $B = b_1 b_2 \dots b_m$ in the following way.

- First, map each symbol to a corresponding integer $i \in \{0, 1, 2, \dots, d-1\}$ 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.

Polynomial rolling hash

- We can therefore identify B with a sequence of IDs $\langle b_1, b_2, \ldots, b_{m-1}, b_m \rangle$, each between 0 and $d-1$ inclusive. Viewing these IDs as digits in base d, we can construct a corresponding integer

$$\begin{aligned} h(B) &= h(b_1 b_2 \ldots b_{m-1} b_m) \\ &= d^{m-1} \times b_1 + d^{m-2} \times b_2 + \ldots + d \times b_{m-1} + b_m. \end{aligned}$$

- This can be evaluated efficiently using Horner's rule:

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

  requiring only $m-1$ additions and $m-1$ multiplications.
- Next we choose a large prime number p and define the hash value of B as $H(B) = h(B) \bmod p$.

Polynomial rolling hash

- Recall that $A = a_1 a_2 a_3 \ldots \ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots \ldots a_n$ where $n \gg m$.

- We want to find efficiently all starting points s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_1 b_2 \ldots b_m$ are equal.

- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

  $H(A_s) = d^{m-1} a_s + d^{m-2} a_{s+1} + \ldots + d^1 a_{s+m-2} + a_{s+m-1} \bmod p.$

Polynomial rolling hash

- We can now compare the hash values $H(B)$ and $H(A_s)$.
    - If they disagree, we know that there is no match.
    - If they agree, there might be a match. To confirm this, we'll check it 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 the naïve algorithm.
- This is where recursion comes into play: we do not have compute the hash value $H(A_{s+1})$ "from scratch". Instead, we can compute it efficiently from the previous hash value $H(A_s)$.

Polynomial rolling hash

- Recall that $A_s = a_s a_{s+1} \dots a_{s+m-1}$, so

$$H(A_s) = d^{m-1} a_s + d^{m-2} a_{s+1} + \ldots + a_{s+m-1} \bmod p.$$

- Multiplying both sides by d gives

$$\begin{aligned} d \cdot H(A_s) &= d^m a_s + d^{m-1} a_{s+1} + \ldots + d^1 a_{s+m-1} \\ &= d^m a_s + \left(d^{m-1} a_{s+1} + \ldots + d^1 a_{s+m-1} + a_{s+m}\right) - a_{s+m} \\ &= d^m a_s + H(A_{s+1}) - a_{s+m} \bmod p, \end{aligned}$$

since $A_{s+1} = a_{s+1} \dots a_{s+m-1} a_{s+m}$.

Polynomial rolling hash

- Rearranging the last equation, we have the *recurrence*

$$H(A_{s+1}) = d \cdot H(A_s) - d^m a_s + a_{s+m} \bmod p.$$

- Thus we can compute $H(A_{s+1})$ from $H(A_s)$ in constant time.

Note

When implementing this, we would like to choose large primes to reduce the likelihood of hash collisions. However, this is constrained by the intermediate values in the calculation above, which are up to $d \times p$, and must not overflow a register.

Algorithm

1. First compute $H(B)$ and the base case $H(A_1)$ in $O(m)$ time using Horner's rule.
2. Then compute the $O(n)$ subsequent values of $H(A_s)$ each in constant time using the recurrence above.
3. 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.

Rabin-Karp Algorithm

- Since p was chosen large, the false positives (where $H(A_s) = H(B)$ but $A_s \neq B$) are very unlikely, which makes the algorithm run fast in the average case.

- However, as always when we use hashing, we cannot achieve useful bounds for the worst case performance.

- So we now look for algorithms whose worst case performance is guaranteed to be linear.

Improving the naïve algorithm

- Recall the earlier naïve 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, i.e. matching b_1 with a_{s+1}.

- Maybe part of the matched section might be salvageable?

Improving the naïve algorithm

- Let $B_k = b_1 \dots b_k$ denote a prefix of length k of the pattern B.

- Suppose we are partway through the text, and the k most recent characters matched with B_k.

- If the next character of the text is b_{k+1}, we are happy to extend our match from length k to length $k + 1$.

- What should we do if the next character isn't b_{k+1}?
  - We can't continue this partial matching, but we could try extending some *shorter* partial matching!

Improving the naïve algorithm

- For example, suppose the pattern B is `xyxyxzx`.

- If we have read the text up to `...xyxyx`, we would be maintaining a partial match of length 5.

- We are hoping to see a `z` next, so if the next character is `y`, we can't extend our partial match to length 6.

- But we can fall back on a shorter partial match of length 3 ($\texttt{...xy}\underline{\texttt{xyx}}$), and successfully extend that ($\texttt{...xy}\underline{\texttt{xyxy}}$) instead!

Improving the naïve algorithm

- xyx was suitable to extend because it was both a *prefix* ($\underline{\texttt{xyx}}\texttt{yx}$) and a *suffix* ($\texttt{xy}\underline{\texttt{xyx}}$) of B_5.
    - If it wasn't a prefix of B_5 (and therefore of B), it couldn't be the start of a new partial match.
    - If it wasn't a suffix of B_5, it couldn't be the most recent sequence of characters in the text ($\ldots\texttt{xy}\underline{\texttt{xyx}}$).

Improving the naïve algorithm

- There might be more than one such candidate. Why did we try `xyx` instead of `x` ($\underline{\texttt{x}}\texttt{yxy}\underline{\texttt{x}}$)?
    - Nesting property!
    - A short prefix-suffix must appear:
        - at the start of a long prefix-suffix, because it's a prefix, and
        - at the end of a long prefix-suffix, because it's a suffix.
    - If we failed to extend `xyx`, we could then try extending `x`, *its* longest prefix-suffix.

The failure function

Definition

Let $\pi(k)$ be the length of the longest string which is both a (proper) prefix and suffix of B_k.

$B_{\pi(k)}$ is therefore the longest substring which appears at both the start and the end of B_k (allowing partial but not total overlap).

We will call π the *failure function*, but some authors refer to it as the *prefix function*.

The failure function

- Any time we fail to extend our current partial match (of length k), we could instead try to extend the longest prefix-suffix (of length $\pi(k)$).
    - Compared to the naïve algorithm, this allows us to skip some starting points (such as $\texttt{...x}\underline{\texttt{y}}\texttt{xyx}$).
        - We know $\pi(5)$ is 3, not 4, so the last four characters of the text don't match the first four characters of B.
        - This means there is no chance for a full match of B starting four characters ago.
    - It also allows us to skip matching characters of the text that we have already matched.
        - The most recent three characters of the text ($\texttt{...xy}\underline{\texttt{xyx}}$) don't need to be checked individually.
        - We know from the structure of the *pattern* that they will match.
- If this fails, we could then try to extend *its* longest prefix-suffix (of length $\pi(\pi(k))$), and so on.

The Knuth-Morris-Pratt Algorithm

Algorithm

Maintain pointers l and r into the text, which record the left and right boundaries (inclusive) of our current partial match.

- Initially, $l = 1$ and $r = 0$.
- We'll use $w = r - l + 1$ as shorthand for the length of the current partial match.

The Knuth-Morris-Pratt Algorithm

Algorithm (continued)

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 characters 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 both l and r by one and move on.

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

The Knuth-Morris-Pratt Algorithm

- What is the time complexity of this algorithm?
- For each of n characters of the text, we might need to compare it to several characters of the text; is this also $O(nm)$?
- No! It is actually linear, i.e. $O(n)$.
- After each comparison between a pair of characters, at least one of the pointers l and r moves forwards.
    - If the characters match, we increase r by 1.
    - If the characters don't match, we increase l to shorten the partial match
        - If the match length is zero, we increase r by 1 also.
- Each pointer can take up to n values, so the total number of steps is $O(n)$. This is an example of *amortisation*.

Computing the failure function

- All this assumes that we have computed the failure function π. How would we actually do that?

- Dynamic programming! Let's form a recurrence, using the same ideas as before.

- What's the longest prefix-suffix of B_{k+1}?

    - Ideally, it's the extension of the longest prefix-suffix of B_k, namely $B_{\pi(k)}$.

    - If $b_{k+1} = b_{\pi(k)+1}$, then $\pi(k+1) = \pi(k) + 1$.

    - Otherwise, what's the next best candidate to extend? $B_{\pi(\pi(k))}$ …

Computing the failure function

Solution (continued)

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

Recurrence: for $k \geq 1$,

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

Base case: $\pi(1) = 0$.

Computing the failure function

Solution (continued)

Order of computation: solve subproblems $P(k)$ in increasing order of k.

Overall answer: the entire list of values $\pi(k)$.

Time complexity: $O(m)$, by a similar two-pointers argument.

String matching with 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 $0, 1, \ldots, m$, which correspond to the number of characters matched thus far, and
  - a transition function $\delta(k, \texttt{t})$, where $0 \leq k \leq m$ and $\texttt{t}$ is a symbol in the alphabet.
- Suppose that the last k characters of the text A match B_k, and that $\texttt{t}$ is the next character in the text. Then $\delta(k, \texttt{t})$ is the new state after character $\texttt{t}$ is read, i.e. the largest j so that the last j characters of A (ending at the new character $\texttt{t}$) match the start of the pattern B.
- Clearly the transition function δ is closely related to the failure function π.

String matching with finite automata

As an example, consider the pattern xyxyxzx. The table defining $\delta(k, \mathtt{t})$ 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

String matching with finite automata

This table can be visualised as a state transition diagram. From each state k, we draw an arrow to $\delta(k, \mathtt{t})$ for each character $\mathtt{t}$ that could be encountered next. Arrows pointing to state 0 have been omitted for clarity.

$$B = \mathtt{xyxyxzx}$$

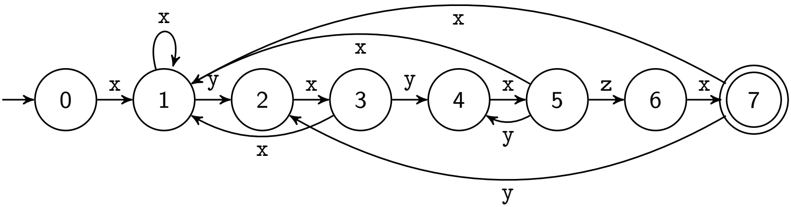


Table of Contents

1. Introduction
2. One Parameter, Constant Recurrence
3. One Parameter, Linear Recurrence
4. Two Parameters, Constant Recurrence
5. Two Parameters, Linear Recurrence
6. Applications to Graphs
7. Applications to String Matching
8. Puzzle

Puzzle

On a rectangular table there are 25 round coins of equal size, and no two of these coins overlap. You observe that in current arrangement, it is not possible to add another coin without overlapping any of the existing coins and without the coin falling off the table (for a coin to stay on the table its centre must be within the table).

Show that it is possible to completely cover the table with 100 coins (allowing overlaps).

That's All, Folks!!