Week 8

Dynamic Programming

1. Introduction
2. Making change
3. A general framework
4. Matrix chain multiplication
5. Longest common subsequence
6. Floyd-Warshall algorithm

Dynamic programming

Read:

- CLRS Chapter 15.
- We cover Section 15.2, 15.3, 15.4.
- Section 15.1 is a nice introduction, so it would be good if you could also read this.
- We also treat Section 25.2.

You might consider appendix

D.1 "Matrices and matrix operations"

The first example (change-making) is considered in detail, while the other three examples are handled more quickly —

the point here is to understand the principle of "dynamic programming".

When greedy algorithms fail: making change

Suppose we want to solve the Making Change problem of paying 9 pence with 1, 4 and 6 pence coins.

- The greedy algorithm gives 6+1+1+1 rather than the optimal 4+4+1.
- So a new idea is required.
- We need a more systematic way of searching for a solution.
- While we want to avoid (if possible) searching through all possible combinations.

Perhaps we can solve very simple problems, and then proceed recursively:

1. If 0 pence is to be returned, we just use zero coins (whatever the coins are).
2. If we just have one coin, then the solution is also clear.

Dealing with two coins

Now assume that we have two coins with values $d_1 \neq d_2$, and we have to pay the sum of N pence. Consider an optimal solution $a_1 \cdot d_1 + a_2 \cdot d_2$ (assuming a solution exists at all!), using $a_1 + a_2$ coins.

1. Either we use coin d_2 or not.
2. That is, either $a_2 > 0$ or $a_2 = 0$.
3. In the second case, only one coin is left and we are done.
4. So *assume* $a_2 > 0$.
5. We know d_2 is used at least once.
6. Now for the amount $N - d_2$ we know that $a_1 \cdot d_1 + (a_2 - 1) \cdot d_2$ is an optimal solution!
7. Since if there would be a solution using fewer coins, then by using one more coin we would obtain a better solution for the original problem.

Making change: the general idea

We arrive at the idea of a general strategy, to solve the problem with coins $d_1, \ldots, d_n$ and amount N to be payed:

- Look how we do if we use only coins $d_1, \ldots, d_{n-1}$.
- Look how we do if we use d_n once, decreasing N to $n - d_n$.
- Compare the two possibilities, and choose the better.

This scheme is to be applied recursively.

Making change: the general structure

Introduction
Making change
A general framework
Matrix chain multiplication
Longest common subsequence
Floyd-Warshall algorithm

So we can solve the general Making Change problem as follows:

- In order to pay the sum of N pence using n distinct coins $(d_1, d_2, \ldots, d_n)$,

  we set up an $n \times (N+1)$ table $c[1 \ldots n, 0 \ldots N]$.
- In this table, $c[i, j]$ will hold the minimum number of coins required to pay the amount j using only coins $d_1, \ldots, d_i$.

  (If no arrangement of such coins makes up j pence, then we shall have $c[i, j] = \infty$.)
- The solution will then be contained in $c[n, N]$.

The table is filled out recursively according to the case-distinction "use last coin or not".

Bookkeeping for making change

To summarise we fill out the table as follows:

- Clearly $c[i, 0] = 0$ for every i.
- Also, for every j, $c[1, j] = \begin{cases} j \text{ div } d_1 & \text{if } j \bmod d_1 = 0, \\ \infty & \text{otherwise.} \end{cases}$

  (Whenever we cannot make change for amount j using coins $d_1, \ldots, d_i$, we let $c[i, j] = \infty$.)
- For $c[i, j]$ ($i > 1$, $j > 0$), we either:
  - pay j pence using only coins $d_1, \ldots, d_{i-1}$: $c[i, j] \leq c[i-1, j]$
  - or use (at least) one coin d_i, and reduce the problem to that of paying $j - d_i$: $c[i, j] \leq 1 + c[i, j - d_i]$.
- As we want to minimise the number of coins, we choose the better of these two options:

$$c[i, j] = \min\left(c[i-1, j],\ 1 + c[i, j - d_i]\right).$$

The making-change algorithm

```
MAKING-CHANGE(N, d[1..n])                    //Running time O(nN)
 1  for i = 1 to n
 2      c[i,0] = 0
 3  for j = 1 to N
 4      if (j mod d_1)=0
 5          c[1,j] = j div d_1
 6      else c[1,j] = ∞
 7  for i = 2 to n
 8      for j = 1 to N
 9          if j < d_i
10              c[i,j] = c[i−1,j]
11          else c[i,j] = min(c[i−1,j], 1 + c[i,j−d_i])
12  return c
```

The making-change algorithm (continued)

Introduction

Making change

A general framework

Matrix chain multiplication

Longest common subsequence

Floyd-Warshall algorithm

Example: Paying 9 pence using 6, 1 and 4 pence coins (order irrelevant).

Amount	0	1	2	3	4	5	6	7	8	9
$d_1 = 6$	0	∞	∞	∞	∞	∞	1	∞	∞	∞
$d_2 = 1$	$\boxed{0}$	$\boxed{1}$	2	3	4	5	1	2	3	4
$d_3 = 4$	0	$\boxed{1}$	2	3	1	$\boxed{2}$	1	2	2	$\boxed{3}$

Determining an optimal solution

The algorithm Making-Change tells us that $c[3,9] = 3$ coins are sufficient to make up 9 pence – but *which* three coins to use?

The following algorithm will answer that, retracing the solution through the c-table (precomputed by Making-Change). The output is an array $a_1, \ldots, a_n$ of natural numbers ≥ 0 such that

$$a_1 \cdot d_1 + \cdots + a_n \cdot d_n = N$$

and such that $a_1 + \cdots + a_n$ is minimal.

Pay-Out$(c[1..n, 0..N], d[1..n])$

1. **for** $(i = 1; i \leq n; ++i)$ $a[i] = 0$;
2. $i = n$; $j = N$;
3. **while** $j > 0$
4.   **if** $(i = 1)$ $a[1] = c[1][j]$; $j = 0$
5.   **else if** $(c[i,j] = c[i-1,j])$ $i = i - 1$
6.   **else** $a[i] = a[i] + 1$; $j = j - d_i$
7. **return** a

Complexity of finding an optimal solution

- This algorithm involves stepping back (at most) n rows, and making $c[n, N]$ jumps to the left.
- Hence it runs in time $\mathrm{O}(n + N)$.
- Thus it is a negligible addition to the $\mathrm{O}(nN)$ algorithm MAKING-CHANGE.

Of course, we could have computed the array $a_1, \ldots, a_n$ right away directly in MAKING-CHANGE — we don't need array c to be completely computed. As an exercise, modify the code of MAKING-CHANGE to do so.

Why the greedy algorithm fails

As with problems which can be solved by greedy algorithms, Making-Change has the

Optimal Substructure Property: An optimal solution to the problem contains optimal solutions to subproblems.

However, the *greedy-choice property* fails. We now have to consider many potential solutions, which requires added bookkeeping: we need to remember past decisions, and also build solutions from the bottom up.

We do have something though, namely the

Overlapping Subproblems Property: The space of subproblems is small, so an otherwise-obvious

top-down, divide-and-conquer recursive algorithm

would solve the same subproblems *over and over*.

Dynamic programming

The presence of

- **Optimal Substructure Property** and
- **Overlapping Subproblems Property**

characterises **Dynamic Programming**.

With dynamic programming, we take a natural recursive definition, and instead of computing it in a *top-down* fashion, we compute it *bottom-up*, exploiting the **overlapping subproblems property** by only solving each subproblem once.

For example, a top-down algorithm for computing Fibonacci numbers from their definition:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

would run in exponential time, while a bottom-up algorithm, computing $F_0, F_1, F_2, F_3, F_4, \ldots, F_{n-1}, F_n$ would run in linear time.

Matrix chain multiplication

Assumption: Computing $A \cdot B$, where A is a $p \times q$ matrix and B is a $q \times r$ matrix, requires $p \cdot q \cdot r$ scalar multiplications.

Question: What is the most efficient order for multiplying $A_1 \cdot A_2 \cdot \cdots \cdot A_n$ where A_i is a $p_{i-1} \times p_i$ matrix?

Example: A_1, A_2, A_3, A_4 are matrices of dimensions 6×2, 2×5, 5×4, 4×3. Then $A_1 A_2 A_3 A_4$ can be computed in five ways:

$$
\begin{array}{lll}
((A_1 A_2) A_3) A_4 & \rightsquigarrow & 252 \text{ scalar multiplications} \\
(A_1 A_2)(A_3 A_4) & \rightsquigarrow & 210 \text{ scalar multiplications} \\
(A_1 (A_2 A_3)) A_4 & \rightsquigarrow & 160 \text{ scalar multiplications} \\
A_1 ((A_2 A_3) A_4) & \rightsquigarrow & 100 \text{ scalar multiplications} \\
A_1 (A_2 (A_3 A_4)) & \rightsquigarrow & 126 \text{ scalar multiplications.}
\end{array}
$$

So the right choice can make a big difference. But we would like to avoid making this choice by exhaustive search as in the above example, since there are exponentially many possibilities (in the number of matrices).

Example explained

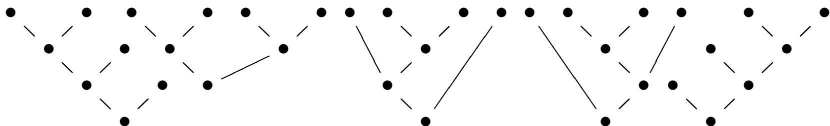
Why does the computation of $((A_1 A_2) A_3) A_4$ need 252 scalar multiplications?

1. $B := A_1 A_2$: 6×2, 2×5 matrices, needs $6 \cdot 2 \cdot 5 = 60$ multiplications, yields 6×5 matrix.
2. $C := B A_3$: 6×5, 5×4 matrices, needs $6 \cdot 5 \cdot 4 = 120$ multiplications, yields 6×4 matrix.
3. $C A_4$: 6×4, 4×3 matrices, needs $6 \cdot 4 \cdot 3 = 72$ multiplications;

So altogether $60 + 120 + 72 = 252$ multiplications are needed.

How many possibilities are there?

The possibilities to multiply n matrices correspond exactly to the (unlabelled!) ordered rooted trees with n leaves, where every non-leaf node has exactly two children:



The number of such rooted trees is

$$\frac{\binom{2n-2}{n-1}}{n}$$

so for $n = 4$ we get $\binom{6}{3}/4 = 5$. In general this number grows exponentially.

The key observation

- If $n = 1$ then the problem is trivial (nothing to multiply). Also for $n = 2$ it's trivial, but we don't need that.
- Now regarding the product $A_1 \cdot \dots \cdot A_n$, $n \geq 2$: In the tree of operations we consider the root. It has two children, representing a partitioning of $\{1, \dots, n\}$ into two intervals.
- So the point is to choose $1 \leq k < n$, and to compute $B := A_1 \cdot \dots \cdot A_k$ and $C := A_{k+1} \cdot \dots \cdot A_n$ optimally.
- When choosing k, for its evaluation we have to add up the costs for the two subcalculations of B and C plus the cost for the single computation of $B \cdot C$.
- Optimality is obvious.

So here we just have a one-dimensional recursive descent (while for change-making it was two-dimensional).

Note: A prerequisite for dynamic programming is the ability to precisely calculate the cost of what is to be optimised.

A dynamic programming solution

Let $m[i,j]$ be the cost (number of scalar multiplications) of computing $A_i \times A_{i+1} \times \cdots \times A_j$ using an optimal order of multiplying the matrices. Then

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) & \text{if } i < j. \end{cases}$$

- If we implement this as a top-down recursive algorithm, it would run in exponential time (since again we have two recursive calls, where the size of the subproblems is only slightly decreased).
- However, there are only a quadratic number of subproblems!
- The recursive algorithm solves these over and over.
- So we use a bottom-up dynamic-programming algorithm.

Again: how to get the (full) solution?

- $m[1, n]$ only tells us how many scalar multiplications are necessary in the optimal solution, not what the optimal solution is (i.e., the order in which to multiply the matrices).
- For this, we can maintain a second table s, and record in $s[i, j]$ the value of k from which the minimal value of $m[i, j]$ was computed.

The algorithm

Matrix-Chain-Order(p) //Running time $O(n^3)$

```
1   n = length(p)−1
2   for i = 1 to n
3       m[i,i] = 0
4   for l = 2 to n
5       for i = 1 to n−l+1
6           j = i+l−1; m[i,j] = ∞
7           for k = i to j−1
8               q = m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
9               if q < m[i,j]
10                  m[i,j] = q; s[i,j] = k
11  return (m,s)
```

Variable l is the "span", that is, how many matrices are involved in the current consideration:

1. The initialisation corresponds to $l = 1$.
2. The still trivial case of $l = 2$ is then considered.
3. Then $l = 3$, and so on.

Running the example

For our example, $p = (6, 2, 5, 4, 3)$, and the tables computed are:

m	1	2	3	4
1	0	60	88	100
2	–	0	40	64
3	–	–	0	60
4	–	–	–	0

s	1	2	3	4
1	0	1	1	1
2	–	0	2	3
3	–	–	0	3
4	–	–	–	0

So we obtain $A_1((A_2 A_3)A_4)$ (with 100 multiplications) as optimal solution.

Note that peculiar way how we run through the m-array:

1. First $l = 1$, that is, setting the diagonal to zero.
2. Then $l = 2$, considering all pairs $(1, 2), (2, 3), \dots,$ and setting them to their directly given value.
3. Then $l = 3$, considering all pairs $(1, 3), (2, 4), \dots,$ where now for the cuts (given by k) all computations are already present. And so on.

Printing the matrix-chain

Using the s-table, we can then reconstruct and print out an optimal Matrix Chain as follows (using s as computed by MATRIX-CHAIN-ORDER):

MATRIX-CHAIN-PRINT(i, j)

```
1  if i = j
2      print "A"_i
3  else print "("
4      MATRIX-CHAIN-PRINT(i, s[i, j])
5      MATRIX-CHAIN-PRINT(s[i, j] + 1, j)
6      print ")"
```

We would then invoke this algorithm initially by

MATRIX-CHAIN-PRINT$(1, n)$.

For our example, we get the output: $(A_1((A_2A_3)A_4))$.

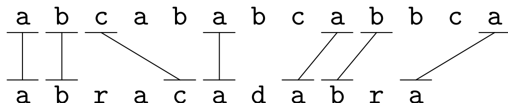
Remarks

- Alternatively, as before, instead of the post-processing we could also just modify MATRIX-CHAIN-ORDER to record the optimal steps found during the process of their computation.
- If we use another algorithm for multiplying two matrices, like Strassen's algorithm (as seen in week 3), then we just need to replace the term "$p_{i-1} \cdot p_k \cdot p_j$" in the minimisation-equation with another (appropriate) term.
- Though this would need precise calculations (not just some big-Oh estimation).

Longest common subsequence (LCS)

Problem: Given two sequences $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$, find the longest common subsequence.

Example: The longest common subsequence of the sequences



is abcaaba and is of length 7.

There are 2^m subsequences of X, so we would like avoiding to check them all with Y.

Fortunately, we can express this as a dynamic programming problem, and derive an efficient algorithm to solve it.

Dynamic programming LCS

Let $c[i,j]$ denote the length of the LCS of $x_1 \dots x_i$ and $y_1 \dots y_j$.

Then $c[m,n]$ is the length of the LCS we are trying to compute.

A recursive definition for c, exploiting an obvious optimal subproblem property, is given as follows.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Computing the length of the LCS

The following is a dynamic-programming algorithm which computes $c[i,j]$, as well as a further table $b[i,j]$ to use for constructing the LCS.

```
LCS-LENGTH(X, Y)                              //Running time O(mn)
 1  m = length(X) ;   n = length(Y)
 2  for i = 0 to m
 3        c[i,0] = 0
 4  for j = 0 to n
 5        c[0,j] = 0
 6  for i = 1 to m
 7        for j = 1 to n
 8              if x_i = y_j
 9                    c[i,j] = c[i−1,j−1] + 1 ;   b[i,j] = "↖"
10              else if c[i−1,j] ≥ c[i,j−1]
11                          c[i,j] = c[i−1,j] ;   b[i,j] = "↑"
12                    else c[i,j] = c[i,j−1] ;   b[i,j] = "←"
13  return (c, b)
```

Printing the LCS

```
PRINT-LCS(b, X, i, j)                    //Running time O(m + n)
1  if i=0 or j=0
2        return
3  else if b[i, j] = "↑"
4              PRINT-LCS(b, X, i−1, j)
5        else if b[i, j] = "←"
6                    PRINT-LCS(b, X, i, j−1)
7              else PRINT-LCS(b, X, i−1, j−1); print x_i
```

Note the motivation for the arrow-labels:

1. "↑" means to go one row up (same column).
2. "←" means go one column left (same row).
3. "↖" means to one row up and one column left.

Example

LCS = BCBA

	j	0	1	2	3	4	5	6	7	8
i		y_j	**B**	D	**C**	B	B	A	**B**	**A**
0	x_i	0	0	0	0	0	0	0	0	0
1	A	**0**	↑0	↑0	↑0	↑0	↑0	↖1	←1	↖1
2	**B**	0	↖**1**	←**1**	←1	↖1	↖1	↑1	↖2	←2
3	**C**	0	↑1	↑1	↖**2**	←**2**	←**2**	←**2**	↑2	↑2
4	**B**	0	↖1	↑1	↑2	↖3	↖3	←3	↖**3**	←3
5	D	0	↑1	↖2	↑2	↑3	↑3	↑3	↑**3**	↑3
6	**A**	0	↑1	↑2	↑2	↑3	↑3	↖4	←4	↖**4**

Note how to reconstruct the solution:

1. Start in the bottom-right corner, and follow the directed graph given by the arrows.
2. For fields marked by "↖" read off the corresponding letter.

All-pairs shortest paths

Problem: Calculating the shortest route between any two cities from a given set of cities $1, 2, \ldots, n$.

Input: A matrix $d_{i,j}$ $(1 \leq i, j \leq n)$ of nonnegative values indicating the length of the direct route from i to j.
Note: $d_{i,i} = 0$ for all i; and if there is no direct route from i to j, then $d_{i,j} = \infty$.

Output: A shortest distance matrix $s_{i,j}$ indicating the length of the shortest route from i to j.

We shall give a recursive definition for s, which can be computed by a dynamic-programming algorithm.

The Floyd-Warshall algorithm

Let $s_{i,j}^{(k)}$ denote the shortest distance from i to j which only passes through cities $1, 2, \ldots, k$ (besides i, j). A recursive definition for $s_{i,j}^{(k)}$ is given as follows.

$$s_{i,j}^{(k)} = \begin{cases} d_{i,j} & \text{if } k = 0, \\ \min\left(s_{i,j}^{(k-1)}, s_{i,k}^{(k-1)} + s_{k,j}^{(k-1)}\right) & \text{if } k > 0. \end{cases}$$

Floyd-Warshall-1(d, n)

```
1  s^(0) = d
2  for k = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              s_{i,j}^(k) = min( s_{i,j}^(k-1), s_{i,k}^(k-1) + s_{k,j}^(k-1) )
```

This algorithm runs in $O(n^3)$ time and space. However, we can safely remove the superscripts from s (can you see why?), and achieve $O(n^2)$ space.

Constructing shortest paths

To construct the shortest paths, we maintain a *predecessor matrix* $\pi_{i,j}$ in which $\pi_{i,j}$ denotes the predecessor of j on some shortest path from i to j. (If $i = j$ or there is no such path, then $\pi_{i,j} = \text{NIL}$.) The final algorithm for computing s and π:

FLOYD-WARSHALL(d, n)

```
1   s = d
2   for i = 1 to n
3       for j = 1 to n
4           if i=j or d_{i,j}=∞
5               π_{i,j} = NIL
6           else π_{i,j} = i
7   for k = 1 to n
8       for i = 1 to n
9           for j = 1 to n
10              x = s_{i,k} + s_{k,j}
11              if x < s_{i,j}
12                  s_{i,j} = x; π_{i,j} = π_{k,j}
```

Example

$s^{(0)}/\pi^{(0)} = s^{(1)}/\pi^{(1)}$

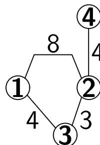
	1	2	3	4
1	$0/\text{NIL}$	$8/1$	$4/1$	∞/NIL
2	$8/2$	$0/\text{NIL}$	$3/2$	$4/2$
3	$4/3$	$3/3$	$0/\text{NIL}$	∞/NIL
4	∞/NIL	$4/4$	∞/NIL	$0/\text{NIL}$

$s^{(2)}/\pi^{(2)}$

	1	2	3	4
1	$0/\text{NIL}$	$8/1$	$4/1$	$12/2$
2	$8/2$	$0/\text{NIL}$	$3/2$	$4/2$
3	$4/3$	$3/3$	$0/\text{NIL}$	$7/2$
4	$12/2$	$4/4$	$7/2$	$0/\text{NIL}$

$s^{(3)}/\pi^{(3)} = s^{(4)}/\pi^{(4)}$

	1	2	3	4
1	$0/\text{NIL}$	$7/3$	$4/1$	$11/2$
2	$7/3$	$0/\text{NIL}$	$3/2$	$4/2$
3	$4/3$	$3/3$	$0/\text{NIL}$	$7/2$
4	$11/3$	$4/4$	$7/2$	$0/\text{NIL}$



d	1	2	3	4
1	0	8	4	∞
2	8	0	3	4
3	4	3	0	∞
4	∞	4	∞	0