IE5600 Applied Programming | Week 4

Dynamic Programming

Andrew Lim

Agenda Today

- Updated Course Schedule – www.limandrew.org/ie5600
- Group Formation
- Group Project 1 - Due in 2 Weeks
- Dynamic Programming
- Time Dependent Shortest Path (Pan Binbin)

Group Assignment for Group Projects

	TAN RAYEN	e0175998@u.nus.edu						
	TAN SHUWEN	e0384826@u.nus.edu		CHANG LIANG	e0014813@u.nus.edu		LEOW WEE SHENT JORD	e0176071@u.nus.edu
	EOW CHENG ZHENG	e0344189@u.nus.edu		NI XUE	e0006939@u.nus.edu		GU RUIXUE	e0251049@u.nus.edu
A	YASMIN BINTE AHMAD	e0384222@u.nus.edu	D	THIRUGNANA SAMBANI	e0010871@u.nus.edu	G	CHONG WOONKIAT	e0452659@u.nus.edu
	LEE MING HOR	e0452894@u.nus.edu		NGOH SISUI	e0452722@u.nus.edu		KHOO EE QING	e0384226@u.nus.edu
	XIE XINGCHEN	e0384332@u.nus.edu		PENG JINGMING	e0383707@u.nus.edu		IRNA BINTI JUMAHAT	e0343996@u.nus.edu
	LIM WEI YANG DYLAN	e0176039@u.nus.edu		LIM ZHEN WEI ZAN	e0176128@u.nus.edu		WU TONG	e0546064@u.nus.edu
	SEAH CHOON KONG	e0030914@u.nus.edu		HU XINPING	e0344056@u.nus.edu		WANG HAI	e0008442@u.nus.edu
B	WU ZHUOCHUN	e0546183@u.nus.edu	E	HUANG XUAN	e0384269@u.nus.edu	H	LEI HAO	e0014898@u.nus.edu
	CHIN KAR KAY	e0385170@u.nus.edu		LV JISHAODONG	e0341657@u.nus.edu		TANG CHOR THENG	e0384417@u.nus.edu
	LOO WENG HENG	e0450255@u.nus.edu		TAMILARASAN SO TEYG	e0319471@u.nus.edu		SOH LI JING	e0452786@u.nus.edu
	LAI JUN GUO SCOTT	e0175273@u.nus.edu		HOONG AN SHENG SAM	e0175262@u.nus.edu		WANG XIN	e0408701@u.nus.edu
	JIAO YANG	e0450317@u.nus.edu		HUO TIANMING	e0007875@u.nus.edu		ZHOU LINLI	e0452923@u.nus.edu
C	WANG YONGJIE	e0516177@u.nus.edu	F	TOH HAN WEI	e0344017@u.nus.edu	I	LOW SIN FOU	e0344083@u.nus.edu
	KWOK JEFFERY	e0384210@u.nus.edu		DONG ZHENG	e0450318@u.nus.edu		SUTAVERAYA SURATAN	e0176486@u.nus.edu
	SHI HANG	e0450191@u.nus.edu		DOMINIQUE YEO ZONG	e0452886@u.nus.edu		KEN LIM JUNE KUANG	e0344074@u.nus.edu

Recap - Greedy Method

- Knapsack
  - Decide the value of X_1, X_2, X_3 …

$$\max \sum_{i=1}^{n} p_i x_i$$

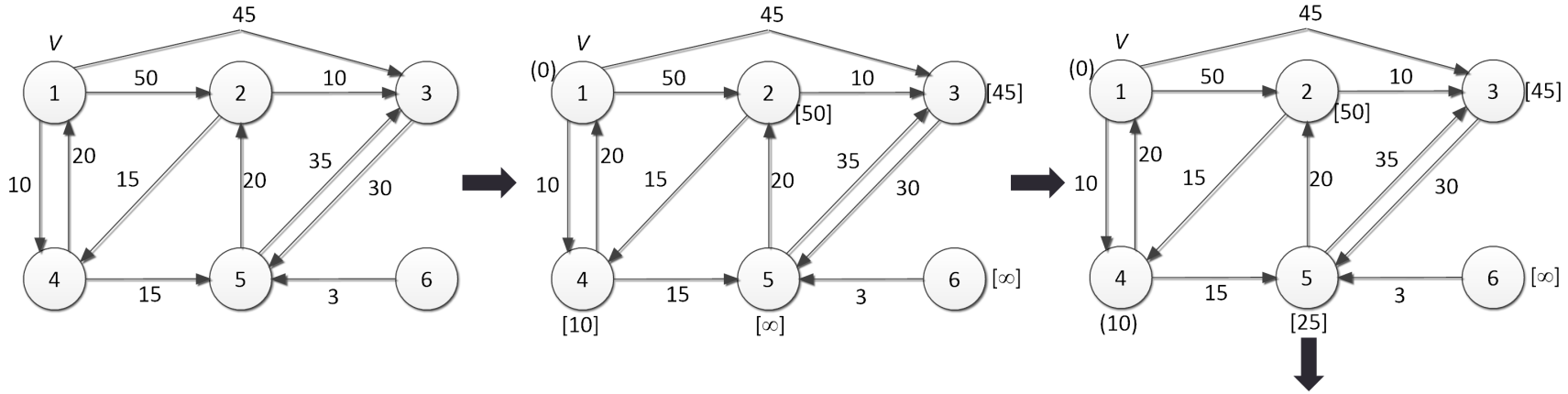
such that

$$\sum_{i=1}^{n} w_i x_i \leq m,\ 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

$n = 3, m = 20$
$(p_1, p_2, p_3) = (25, 24, 15)$
$(w_1, w_2, w_3) = (18, 15, 10)$

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	16.5	24.25
1	2/15	0	20	28.2
0	2/3	1	20	31
0	1	1/2	20	31.5

- Shortest Path from vertex *i* to vertex *j*
  - Decide which vertex is the next vertex closest to *i* until vertex *j* is reached.



- Optimal Solution can be found by making decision one at a time and never making an erroneous decision – Greedy Method.
- For many problems, this is not possible to make stepwise decision.

Shortest Path

- Suppose we want to find a shortest path from vertex *i* to vertex *j*
- Let A_i be the vertices adjacent from vertex *i*
- Which is the vertices in A_i should be the second vertex on the path to *j*?
  - There is no way to decide at this time and guarantee that future decisions leading to an optimal sequence can be made
- On the other hand, if we want to find a shortest path from vertex *i* to all other vertices in G, then at each step a correct decision can be made

Principle of Optimality

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decision must constitute an optimal decision sequence with regards to the state resulting from the first decision

Difference between Greedy and Dynamic programming

- Greedy Method
  - only one decision sequence may be generated
- Dynamic Programming
  - Many decision sequences maybe generated
  - Sequences containing suboptimal subsequences cannot be optimal, based on the principle of optimality holds, and so will not (as far as possible) be generated

Principle of Optimality – Example Shortest Path

- Consider the shortest path problem from i to j.
- Assume that $i, i_1, i_2, \ldots, i_k, j$ is a shortest path from i to j.
- Starting from an initial vertex i, a decision has been made to go to vertex i_1.
- Following this decision, the problem state is defined by i_1 and we need to find a path from i_1 to j.
- It is clear that the sequence $i_1, i_2, \ldots, i_k, j$ must be a shortest path from i_1 to j. If not let $i_1, r_1, r_2, \ldots, r_q, j$ be the shortest path from i_1 to j. Then the path $i, i_1, r_1, r_2, \ldots, r_q, j$ would be shorter than path $i, i_1, i_2, \ldots, i_k, j$

The KNAP(*l*,*j*,*Y*) Problem

$$\max \sum_{i=l}^{i \leq j} p_i x_i$$

$$\text{subject to } \sum_{i=l}^{i \leq j} w_i x_i \leq Y$$

$$x_i = 0 \text{ or } 1, l \leq i \leq j$$

KNAP(*1,n,m*): Principle of Optimality

$$\text{KNAP}(l,j,Y):$$

$$\max \sum_{i=l}^{i\leq j} p_i x_i$$

$$\text{subject to } \sum_{i=l}^{i\leq j} w_i x_i \leq Y$$

$$x_i = 0 \text{ or } 1, l \leq i \leq j$$

Let $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$ respectively.
If $y_1 = 0$, then $y_2, y_3, \ldots, y_n$ must constitute an optimal solution for the problem $\text{KNAP}(2, n, m)$
If $y_1 = 1$, then $y_2, y_3, \ldots, y_n$ must be an optimal sequence for the problem $\text{KNAP}(2, n, m - w_1)$.

0/1 Knapsack Formulation

Let $g_j(y)$ be the value of an optimal solution solution to KNAP$(j+1, n, y)$

Clearly, $g_0(m)$ is the value of an optimal solution to KNAP$(1, n, m)$

The possible decisions for x_1 are 0 and 1

From the Principle of Optimality:
$g_0(m) = \max\{g_1(m), g_1(m - w_1) + p_1\}$

An Example

Consider the case in which $n = 3$ and $m = 6$

$w_1 = 2, w_2 = 3, w_3 = 4;\ p_1 = 1, p_2 = 2, p_3 = 5$

The value of :

$$g_0(6) = \max\{g_1(6), g_1(4) + 1\}$$
$$g_1(6) = \max\{g_2(6), g_2(3) + 2\}$$
$$g_2(6) = \max\{g_3(6), g_3(2) + 5\} = \max\{0, 5\} = 5$$
$$g_2(3) = \max\{g_3(3), g_3(3 - 4) + 5\} = \max\{0, -\infty\}=0$$
$$g_1(6) = \max\{5, 2\} = 5$$
$$g_1(4) = \max\{g_2(4), g_2(4 - 3) + 2\}$$
$$g_2(4) = \max\{g_3(4), g_3(4 - 4) + 5\} = \max\{0, 5\} = 5$$
$$g_2(1) = \max\{g_3(1), g_3(1 - 4) + 5\} = \max\{0, -\infty\} = 0$$
$$g_1(4) = \max\{5, 0\} = 5$$

There $g_0(6) = \{5, 5 + 1\} = 6$

0/1 Knapsack – Looking Backwards

Looking backward on the sequence of decisions, $x_1, x_2, \ldots, x_n$, we see that:

$f_j(y) = \max\{f_{j-1}(y), f_{j-1}(y - w_j) + p_j\}$ where $f_j(y)$ is the value of an optimal solution to $\text{KNAP}(1, j, y)$ with $f_0(y) = 0$ for all $y, y \geq 0$, and $f_0(y) = -\infty$ for all $y, y < 0$.

From this, $f_1, f_2, \ldots, f_n$ can be successively obtained.

0/1 Knapsack

$f_j(y) = \max\{f_{j-1}(y), f_{j-1}(y - w_j) + p_j\}$ where $f_j(y)$ is the value of an optimal solution to $\text{KNAP}(1, j, y)$ with $f_0(y) = 0\ \forall y, y \geq 0$, and $f_0(y) = -\infty\ \forall y, y < 0$.

Let $S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}\}$

S^i is obtained by merging S_1^i and S^{i-1}.

If m is the max weight, we are not interested in any pairs (P, W) with $W > m$. Given any (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$ then the pair (P_j, W_j) can be discarded.

0/1 Knapsack

Consider the knapsack instance $n = 3$, $m = 6$ and $(p_1, p_2, p_3) = (1, 2, 5)$ and $(w_1, w_2, w_3) = (2, 3, 4)$ we have:

$S^0 = \{(0, 0)\}, S_1^1 = \{(1, 2)\}$
$S^1 = \{(0, 0), (1, 2)\}, S_1^2 = \{(2, 3), (3, 5)\}$
$S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\},$
$S_1^3 = \{(5, 4), (6, 6), \cancel{(7, 7)}, \cancel{(8, 9)}\}$
$S^3 = \{(0, 0), (1, 2), (2, 3), \cancel{(3, 5)}, (5, 4), (6, 6)\}$

Let $S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}\}$
S^i is obtained by merging S_1^i and S^{i-1}.

All Pairs Shortest Paths

Let $G = (V, E)$ be a directed graph with n vertices.
Let *cost* be a cost adjacency matrix for G such that
$cost(i, i) = 0, 1 \leq i \leq n$, $cost(i, j)$ is the edge cost
of $e(i, j)$ if $e(i, j) \in E$ else $cost(i, j)$ is ∞.

The all pairs shortest path problem is to determine a
matrix A such that $A(i, j)$ is the length of a
shortest path from (i, j).

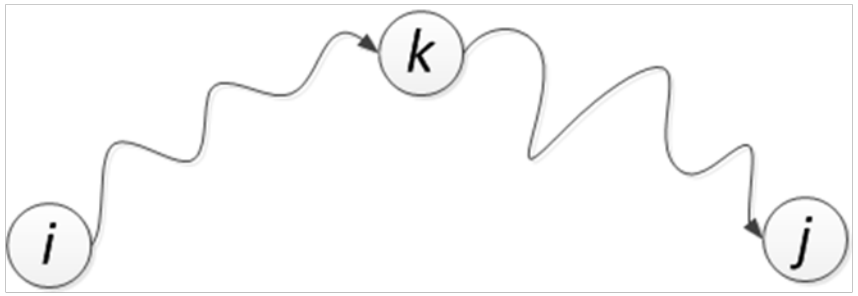
A can be obtained by repeatedly applying the single source
shortest path algorithm with different starting point.

Each application of the single source shortest path algorithm
takes $O(n^2)$ time, A can be obtained in $O(n^3)$ time.

An Alternate Method

An alternative $O(n^3)$ method that requires a weaker restriction on edge weight instead of requiring the edge weight to be non negative, we only require that there is no cycle of negative length.

If there is a cycle of negative length, then shortest path between any two vertices on this cycle has length $-\infty$.



A shortest path i to j where $i \neq j$ goes thru some intermediate vertex k.
No negative cycle.
Paths i to k and k to j must also be the shortest else that path i to j can't be shortest

Principle of Optimality

If there is a cycle of negative length, then the shortest path between any two vertices on this cycle is $-\infty$.

Let's look at the shortest path between i and j.
If k is an intermediate vertex on this shortest path that has the highest index, then we have the following:

$$A(i,j) = \min\{\min_{1 \le k \le n}\{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\}$$

Principle of Optimality

Clearly, $A^0(i,j) = cost(i,j), 1 \leq i \leq n, 1 \leq j \leq n.$

A shortest path from i to j through no vertex higher than k either goes through vertex k or it does not.
If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$.
If it does not, then no intermediate vertex has vertex greater than $k-1$. Hence $A^k(i,j) = A^{k-1}(i,j)$.

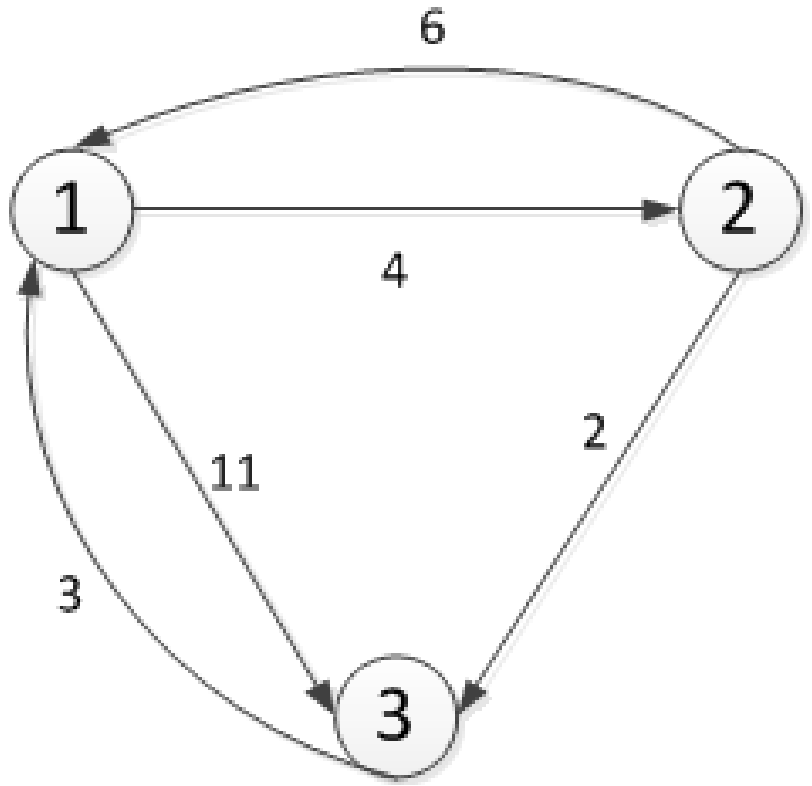
Combining, we get
$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$

The Algorithm

As $A^k(i,k) = A^{k-1}(i,k)$ and $A^k(k,j) = A^{k-1}(k,j)$, the kth column and row do not change when A^k is formed.

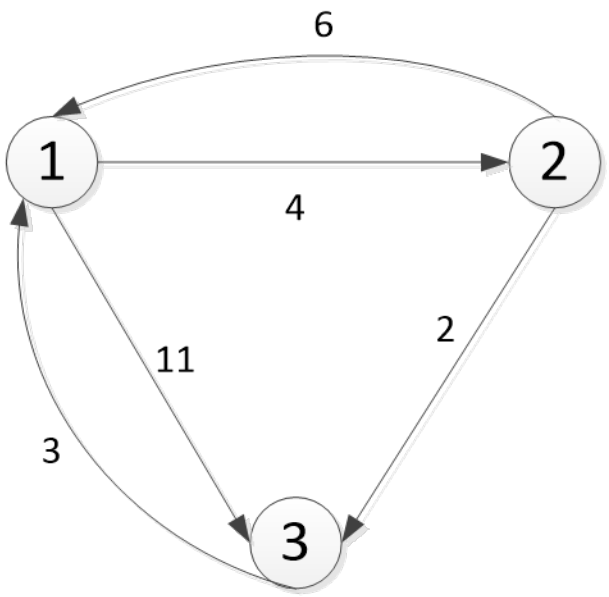
```
//C++ code to compute all pairs shortest path
for (int i=1; i<=n; i++)
 for (int j=1; j<=n; j++)
   A[i][j] = cost[i][j];

for (int k=1; k<=n; k++)
 for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++)
    A[i][j]= min(A[i][j], A[i][k]+A[k][j])
```

An Exercise



A0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

An Exercise – What is A1, A2, A3?



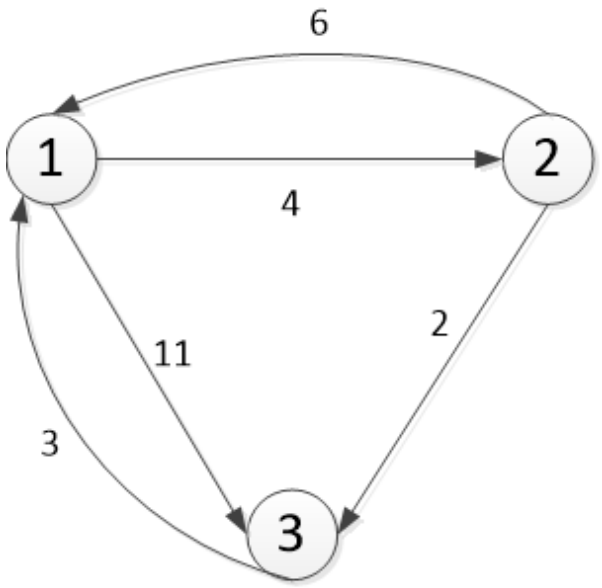
A0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A1	1	2	3
1	0		
2		0	
3			0

A2	1	2	3
1	0		
2		0	
3			0

A3	1	2	3
1	0		
2		0	
3			0

Answer



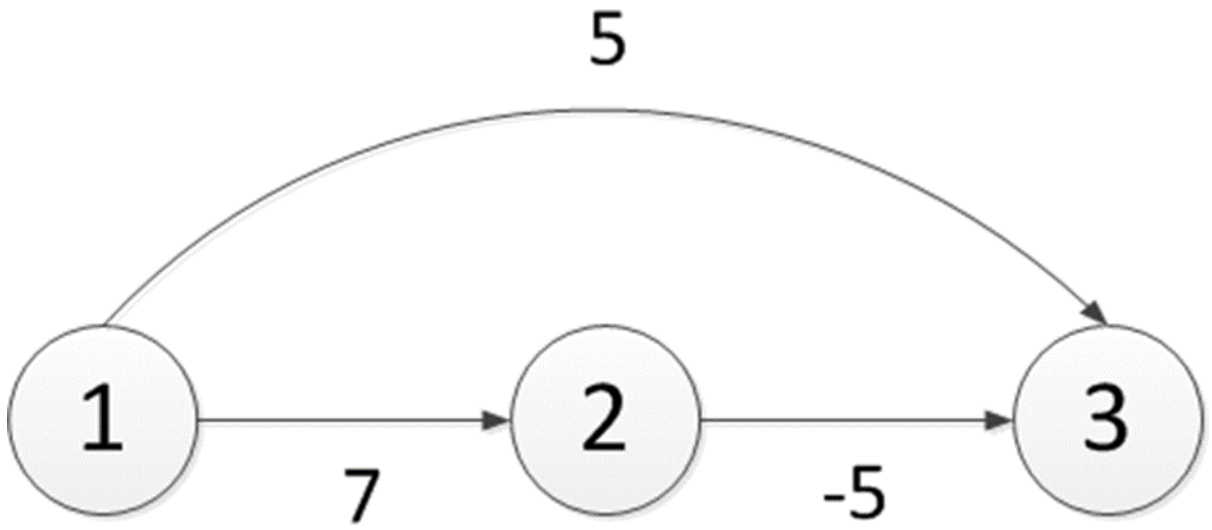
A0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A3	1	2	3
1	0	4	11
2	5	0	2
3	3	7	0

Find the shortest path from vertex 1 to vertex 3



String Editing

Given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$ where $x_i, 1 \leq i \leq n$ are $y_j, 1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*.

We would like to transform X to Y using a sequence of *edit operations* on X.

The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with each operation.

String Editing

The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence.

The problem of string editing is to identify a min-cost sequence of edit operations that will transform X to Y

$D(x_i)$ - cost of deleting the symbol x_i from X
$I(y_j)$ - cost of inserting the symbol y_j into X
$C(x_i, y_j)$ - cost of changing the symbol x_i of X into y_i

An Example

$X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$
$Y = y_1, y_2, y_3, y_4 = b, a, b, b$
$D(x_i) = 1$, $I(y_j) = 1$ and
$C(x_i, y_j) = 2$ if $x_i \neq y_j$ and 0 otherwise

Method 1:
$D(x_i), \forall 1 \leq i \leq 5$ and $I(y_j), \forall 1 \leq j \leq 4$. Cost $= 9$.

Method 2:
$D(x_1)$, $D(x_2)$, $I(y_4)$. Cost $=3$.

Principle of Optimality

$cost(i, j)$ is the min cost of any edit sequence for transforming $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ for $0 \leq i \leq n$ and $0 \leq j \leq m$.

Compute $cost(i, j)$ for each i and j.
$cost(n, m)$ is the cost of an optimal edit sequence.

If $i = j = 0$, $cost(i, j) = 0$ since the two sequences are identical, i.e. empty.
If $j = 0, i > 0$ then $cost(i, 0) = cost(i - 1, 0) + D(x_i)$.
If $i = 0, j > 0$ then $cost(0, j) = cost(0, j - 1) + I(x_j)$.

Principle of Optimality

If $i > 0, j > 0$ then
$cost(i,j) = \min\{cost(i-1,j) + D(x_i),$
$cost(i-1,j-1) + C(x_i, y_j), cost(i,j-1) + I(y_j)\}$

Let $X = a, a, b, a, b$
and $Y = b, a, b, b$

j

i	0	1	2	3	4
0	0	1	2	3	4
1	1				
2	2				
3	3				
4	4				
5	5				

String Editing

If $i > 0, j > 0$ then
$cost(i,j) = \min\{cost(i-1,j) + D(x_i),$
$cost(i-1,j-1) + C(x_i,y_j), cost(i,j-1) + I(y_j)\}$
Let $X = a,a,b,a,b$ and $Y = b,a,b,b$

$cost(1,1) = \min\{cost(0,1) + D(x_1), cost(0,0) +$
$C(x_1,y_1), cost(1,0) + I(y_1)\} = \min\{2,2,2\} = 2$

$cost(1,2) = \min\{cost(0,2) + D(x_1), cost(0,1) +$
$C(x_1,y_2), cost(1,1) + I(y_2)\} = \min\{3,1,2+1\} = 1$

String Editing – Going thru the loops

j

i	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	Cost(5,4)

$$cost(5,4) = \min\{cost(4,4) + D(x_5), cost(4,3) + C(x_5, y_4), cost(5,3) + I(y_4)\} = ?$$

Dynamic Programming Exercises

Contiguous Subsequence Maximum Sum

A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

> *Input:* A list of numbers, $a_1, a_2, \ldots, a_n$.
> *Output:* The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Optimal Sequence of Hotels

You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

YuckDonalds

Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

Reconstituting Words

You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function `dict`$(\cdot)$: for any string w,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise .} \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

Pebbling a checkerboard

Pebbling a checkerboard. We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

(a) Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a *type*, which is the pattern occurring in the last column.

(b) Using the notions of compatibility and type, give an $O(n)$-time dynamic programming algorithm for computing an optimal placement.

Multiplication Operation

Let us define a multiplication operation on three symbols a, b, c according to the following table; thus $ab = b$, $ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Find an efficient algorithm that examines a string of these symbols, say *bbbbac*, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input *bbbbac* your algorithm should return *yes* because $((b(bb))(ba))c = a$.

Palindrome

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Longest Common Substring

Given two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$, we wish to find the length of their *longest common substring*, that is, the largest k for which there are indices i and j with $x_ix_{i+1} \cdots x_{i+k-1} = y_jy_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

String Processing

A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n, regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of $20 + 17 = 37$, while doing position 10 first has a better cost of $20 + 10 = 30$.

Give a dynamic programming algorithm that, given the locations of m cuts in a string of length n, finds the minimum cost of breaking the string into $m + 1$ pieces.

Counting Heads

Counting heads. Given integers n and k, along with $p_1, \ldots, p_n \in [0, 1]$, you want to determine the probability of obtaining exactly k heads when n biased coins are tossed independently at random, where p_i is the probability that the ith coin comes up heads. Give an $O(n^2)$ algorithm for this task.[2] Assume you can multiply and add two numbers in $[0, 1]$ in $O(1)$ time.

Longest Common Subsequence

Given two strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$, we wish to find the length of their *longest common subsequence*, that is, the largest k for which there are indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ with $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{j_1} y_{j_2} \cdots y_{j_k}$. Show how to do this in time $O(mn)$.

Triangulation of minimum cost

You are given a convex polygon P on n vertices in the plane (specified by their x and y coordinates). A *triangulation* of P is a collection of $n-3$ diagonals of P such that no two diagonals intersect (except possibly at their endpoints). Notice that a triangulation splits the polygon's interior into $n-2$ disjoint triangles. The cost of a triangulation is the sum of the lengths of the diagonals in it. Give an efficient algorithm for finding a triangulation of minimum cost. (*Hint:* Label the vertices of P by $1, \ldots, n$, starting from an arbitrary vertex and walking clockwise. For $1 \leq i < j \leq n$, let the subproblem $A(i, j)$ denote the minimum cost triangulation of the polygon spanned by vertices $i, i+1, \ldots, j$.)

Card Collection

Consider the following game. A "dealer" produces a sequence $s_1 \cdots s_n$ of "cards," face up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* strategy is suboptimal.

(b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

Cutting Cloth

Cutting cloth. You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

Chance of Winning

Suppose two teams, A and B, are playing a match to see who is the first to win n games (for some particular n). We can suppose that A and B are equally competent, so each has a 50% chance of winning any particular game. Suppose they have already played $i + j$ games, of which A has won i and B has won j. Give an efficient algorithm to compute the probability that A will go on to win the match. For example, if $i = n - 1$ and $j = n - 3$ then the probability that A will win the match is $7/8$, since it must win any of the next three games.

Garage Sale

The *garage sale problem* (courtesy of Professor Lofti Zadeh). On a given Sunday morning, there are n garage sales going on, $g_1, g_2, \ldots, g_n$. For each garage sale g_j, you have an estimate of its value to you, v_j. For any two garage sales you have an estimate of the transportation cost d_{ij} of getting from g_i to g_j. You are also given the costs d_{0j} and d_{j0} of going between your home and each garage sale. You want to find a tour of a *subset* of the given garage sales, starting and ending at home, that maximizes your total benefit minus your total transportation costs.

Give an algorithm that solves this problem in time $O(n^2 2^n)$. (*Hint:* This is closely related to the traveling salesman problem.)

Coin Changing 6-17,6-18,6-19

6.17. Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem.

Input: $x_1, \ldots, x_n; v$.

Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?

6.18. Consider the following variation on the change-making problem (Exercise 6.17): you are given denominations $x_1, x_2, \ldots, x_n$, and you want to make change for a value v, but you are allowed to use each denomination *at most once*. For instance, if the denominations are $1, 5, 10, 20$, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).

Input: Positive integers $x_1, x_2, \ldots, x_n$; another integer v.

Output: Can you make change for v, using each denomination x_i at most once?

Show how to solve this problem in time $O(nv)$.

Coin Changing 6-17,6-18,6-19

6.19. Here is yet another variation on the change-making problem (Exercise 6.17).

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v using at most k coins; that is, we wish to find a set of $\leq k$ coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 and $k = 6$, then we can make change for 55 but not for 65. Give an efficient dynamic-programming algorithm for the following problem.

> *Input:* $x_1, \ldots, x_n$; k; v.
>
> *Question:* Is it possible to make change for v using at most k coins, of denominations $x_1, \ldots, x_n$?

6.17. This problem reduces to Knapsack with repetitions. The total capacity is v and there is an item i of value x_i and weight x_i for each coin denomination. It is possible to make change for value v if and only if the maximum value we can fit in v is v. The running time is $O(nv)$.

6.18. Use the same reduction as in 6.17, but reduce to Knapsack without repetition. The running time is $O(nv)$.

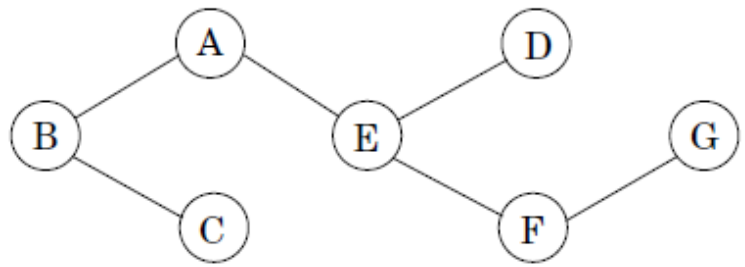
6.19. This is similar to 6.17 and 6.18. The problem reduces to Knapsack without repetition with a capacity of v, but this time we have k items of value x_i and weight x_i for each coin denomination x_i, i.e. a total of kn items. The running time is $O(nkv)$.

Vertex Cover

A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.

> *Input:* An undirected tree $T = (V, E)$.
> *Output:* The size of the smallest vertex cover of T.

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



Subset Sum

Give an $O(nt)$ algorithm for the following task.

Input: A list of n positive integers $a_1, a_2, \ldots, a_n$; a positive integer t.
Question: Does some subset of the a_i's add up to t? (You can use each a_i at most once.)

3-Partition

Consider the following 3-PARTITION problem. Given integers $a_1, \ldots, a_n$, we want to determine whether it is possible to partition of $\{1, \ldots, n\}$ into three disjoint subsets I, J, K such that

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i$$

For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is *yes*, because there is the partition $(1, 8)$, $(4, 5)$, $(2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$ the answer is *no*.

Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and in $\sum_i a_i$.

Sequence alignment - Exercise 6.26

Sequence alignment. When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are *aligned*. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of x and y is a way of matching up these two strings by writing them in columns, for instance:

$$\begin{array}{cccccccc} - & A & T & - & G & C & C \\ T & A & - & C & G & C & A \end{array}$$

Here the "$-$" indicates a "gap." The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix δ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

Give a dynamic programming algorithm that takes as input two strings $x[1 \ldots n]$ and $y[1 \ldots m]$ and a scoring matrix δ, and returns the highest-scoring alignment. The running time should be $O(mn)$.

Alignment with gap penalties

Alignment with gap penalties. The alignment algorithm of Exercise 6.26 helps to identify DNA sequences that are close to one another. The discrepancies between these closely matched sequences are often caused by errors in DNA replication. However, a closer look at the biological replication process reveals that the scoring function we considered earlier has a qualitative problem: nature often inserts or removes entire substrings of nucleotides (creating long gaps), rather than editing just one position at a time. Therefore, the penalty for a gap of length 10 should not be 10 times the penalty for a gap of length 1, but something significantly smaller.

Repeat Exercise 6.26, but this time use a modified scoring function in which the penalty for a gap of length k is $c_0 + c_1 k$, where c_0 and c_1 are given constants (and c_0 is larger than c_1).

Local Sequence Alignment

Local sequence alignment. Often two DNA sequences are significantly different, but contain regions that are very similar and are *highly conserved*. Design an algorithm that takes an input two strings $x[1 \ldots n]$ and $y[1 \ldots m]$ and a scoring matrix δ (as defined in Exercise 6.26), and outputs substrings x' and y' of x and y, respectively, that have the highest-scoring alignment over all pairs of such substrings. Your algorithm should take time $O(mn)$.

Exon chaining

Exon chaining. Each gene corresponds to a subregion of the overall genome (the DNA sequence); however, part of this region might be "junk DNA." Frequently, a gene consists of several pieces called exons, which are separated by junk fragments called introns. This complicates the process of identifying genes in a newly sequenced genome.

Suppose we have a new DNA sequence and we want to check whether a certain gene (a string) is present in it. Because we cannot hope that the gene will be a contiguous subsequence, we look for partial matches—fragments of the DNA that are also present in the gene (actually, even these partial matches will be approximate, not perfect). We then attempt to assemble these fragments.

Let $x[1 \ldots n]$ denote the DNA sequence. Each partial match can be represented by a triple (l_i, r_i, w_i), where $x[l_i \ldots r_i]$ is the fragment and w_i is a weight representing the strength of the match (it might be a local alignment score or some other statistical quantity). Many of these potential matches could be false, so the goal is to find a subset of the triples that are consistent (nonoverlapping) and have a maximum total weight.

Show how to do this efficiently.

Reconstructing evolutionary trees by maximum parsimony

Reconstructing evolutionary trees by maximum parsimony. Suppose we manage to sequence a particular gene across a whole bunch of different species. For concreteness, say there are n species, and the sequences are strings of length k over alphabet $\Sigma = \{A, C, G, T\}$. How can we use this information to reconstruct the evolutionary history of these species?

Evolutionary history is commonly represented by a tree whose leaves are the different species, whose root is their common ancestor, and whose internal branches represent speciation events (that is, moments when a new species broke off from an existing one). Thus we need to find the following:

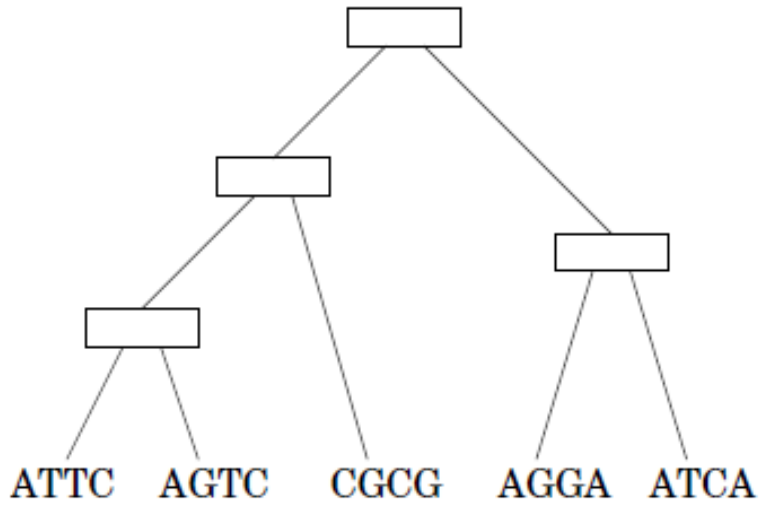
- An evolutionary tree with the given species at the leaves.
- For each internal node, a string of length k: the gene sequence for that particular ancestor.

For each possible tree T, annotated with sequences $s(u) \in \Sigma^k$ at each of its nodes u, we can assign a score based on the principle of *parsimony*: fewer mutations are more likely.

$$\text{score}(T) = \sum_{(u,v) \in E(T)} (\text{number of positions on which } s(u) \text{ and } s(v) \text{ disagree}).$$

Finding the highest-score tree is a difficult problem. Here we will consider just a small part of it: suppose we know the structure of the tree, and we want to fill in the sequences $s(u)$ of the internal nodes u. Here's an example with $k = 4$ and $n = 5$:

Reconstructing evolutionary trees by maximum parsimony



(a) In this particular example, there are several maximum parsimony reconstructions of the internal node sequences. Find one of them.

(b) Give an efficient (in terms of n and k) algorithm for this task. (*Hint:* Even though the sequences might be long, you can do just one position at a time.)