Dynamic Programming

Solve large problem by systematically solving limited subproblems

Unlike with divide-and-conquer, the subproblems are not independent: two subproblems may well contain the same subsubproblem.

Typically one calculates a table (possibly multi-dimensional) of solutions for all the subproblems.

Simple example: Fibonacci numbers.
Calculating $F_0, F_1, \ldots, F_n$ is much faster than recursion.

Matrix Chain Multiplication

Let $A$ be an $p \times q$ matrix, and $B$ a $q \times r$ matrix. Their product is defined to be the $p \times r$ matrix $C$ where

$$C(i, j) = \sum_{k=1}^{q} A(i, k) \cdot B(k, j)$$

Clearly, the calculation is $\Theta(pqr)$: we need $\Theta(pqr)$ scalar multiplications.

One can multiply two $n \times n$ matrices in $o(n^3)$ steps, but we won’t get involved with this.

Recall: matrix multiplication is associative:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

In other words, when we need to evaluate a long product

$$A_1 \cdot A_2 \cdot A_3 \ldots A_{n-1} \cdot A_n$$

we can place parentheses wherever we like.

We cannot rearrange the terms though (matrix multiplication is not commutative).

How many ways are there to fully parenthesize this product?

Equivalently: How many full binary trees on $n$ leaves are there?
In general, need to solve the following recurrence:

\[
P(n) = \begin{cases} 
1 & \text{if } n = 1, \\
\sum_{i=1}^{n-1} P(i)P(n-i) & \text{otherwise.}
\end{cases}
\]

A convoluted horror.

Take a look at Knuth’s “Concrete Mathematics”.

\[P(n) = C_{n-1} \text{ (Catalan numbers):} \]

\[C_n = \frac{1}{n+1} \binom{2n}{n} = \Omega(n^{-3/2}4^n)\]

This is vaguely reminiscent of Huffman coding, or merging a collection of already sorted lists.

But there are important differences: we must keep the matrices in order. We can choose the shape of the tree, but not the leaf-order.

Also, the cost function is more complicated:

If \( A \) is \( p \times q \) matrix, \( B \) \( q \times r \) then the cost of multiplying them is \( p \cdot q \cdot r \) and the result has dimension \( p \times r \).

As a consequence, the simple Greedy Algorithm in Huffman coding won’t work here.

Greedy Algorithm: to get a globally minimal solution, do the locally cheapest thing at each step.
Problem:
How do we find the best parenthesization?

Brute force is out. Need some smart idea.

Let’s do reverse engineering. Suppose
\[(A_1 \cdot A_2 \cdots A_k) \cdot (A_{k+1} \cdots A_n)\]
is an optimal parenthesization.

Total cost:
\[\text{cost}(1, k) + \text{cost}(k + 1, n) + p_0 \cdot p_k \cdot p_n.\]

Crucial Insight:
Both \((A_1 \cdot A_2 \cdots A_k)\) and \((A_{k+1} \cdots A_n)\) must be optimal.

---

First Attack: Hash Tables

Compute optimal parenthesization recursively, top-down.

Let \(C(i,j)\) be the optimal number of multiplications for
\(A_i \cdot A_{i+1} \cdots A_{j-1} \cdot A_j\), where \(1 \leq i \leq j \leq n.\)

We need \(C(1,n)\). Recursion is easy:

\[
\begin{align*}
C(i,j) &= \begin{cases} 
0 & \text{if } i = j, \\
\min_{\sum_{k=i}^{j-1}} C(i,k) + C(k+1,j) + p_{i-1}p_ip_j & \text{otherwise}.
\end{cases}
\end{align*}
\]

Must avoid multiple calls to \(C(i,j)\):

▶ Store already compute values in some container, called memoizing.

Requires clever data structure: insertion, search, and lookup should all be \(O(1)\).

May sound impossible, but hash tables get close.

---

Second Attack: Dynamic Programming

Bottom-up approach: compute an array \(C[i,j]\).

For \(i = j\) can set \(C(i,i) = 0.\)

Then compute all \(C(i,j)\) for \(d = j - i, d = 1, 2, \ldots, n - 1.\)

for \(d = 1 \ldots n-1\) do
for \(i = 1 \ldots n-d\) do
\[
\begin{align*}
j &= i + d; \\
k &= i \ldots j-1; \\
c &= C[i,k] + C[k+1,j] + p[i-1]p[k]p[j]; \\
& \text{if } c < C[i,j] \\
& C[i,j] = c;
\end{align*}
\]

Does this really work?

LHS: \(C[i,j]\)

RHS: \(C[i,k]\) and \(C[k+1,j]\) where \(i \leq k < j.\)

But then \(j - i = d\) and \(k - i, j - (k + 1) < d.\)

Hence we can prove correctness by induction on \(d.\)

Running time is clearly \(O(n^3)\) and in fact \(\Theta(n^3).\)

Table requires more thought, but is slightly faster than recursion and memoizing.
n = 6 and dimensions 30, 35, 15, 5, 10, 20, 25.

Optimal number of scalar multiplications: 15125.

```
0  15750  7875  9875  11875  15125
∞  0   2625  4375  7125  10500
∞  ∞   0   750   2500  5375
∞  ∞  ∞   0  1000   3500
∞  ∞  ∞  ∞   0   5000
∞  ∞  ∞  ∞  ∞   0
```

Pop Quiz: What is the corresponding parenthesization?

We need the split value k for each i, j pair:

\[ C(i, j) = C(i, k) + C(k + 1, j) + p_{i-1}p_kp_j \]

for some \( i \leq k < j \).

Use another array \( S(i, j) \).

```
chainMultiply( i, j )
{ 
  if( i==j ) return A[i];

  k = S[i,j];
  X = chainMultiply( i, k );
  Y = chainMultiply( k+1, j );
  return X * Y; // matrix multiplication
}
```

Making Change: Greed Fails

A last warm-up exercise.
Suppose you have an unlimited supply of quarters, dimes, nickels, and pennies. How does one make change for some number \( x \) of pennies?

Let \( 1 \leq d_1 < d_2 < \ldots < d_n \) be the available denominations.
Really have to compute non-negative integer coefficients \( x_i \) such that

\[ \sum_{i=1}^{n} x_i d_i = x \]

May not have solution for some \( x \) (though can get all sufficiently large as long as GCD of the \( d_i \) is 1, right?).

Simple Greedy Algorithm

```
i = n;
while( x > 0 ) { 
x[i] = x / d[i];
x = x - x[i] * d[i];
i--; }
```

Optimal Evaluation

The algorithms (top-down and bottom-up) only compute the number of multiplications needed to evaluate the chain product.

They do not actually provide any information on how to realize this optimal number.

Must augment the algorithms to obtain this information.

First question: how does one represent the parenthesization as a data structure?
Unfortunately, does not always work:
\[ d_1 = 3, \ d_2 = 5 \text{ and } x = 9. \]

- Try dynamic programming.

Need subproblems. Tempting to tinker with \( x \) only, but does not work.

Here is a clever trick:

**Constraint \( k \): only use denominations \( d_1, \ldots, d_k \).**

\[ k = 0 \text{ only allows } x = 0. \]

\[ k = n \text{ is no constraint at all.} \]

Somehow get \( k + 1 \) from \( k \).

More precisely, compute a 2-dim matrix \( C(x, k) \) where

\[ C(x, k) = \# \text{ coins needed for } x \text{ with constraint } k. \]

hence

\[ C(x, k) = \sum_{i=1}^{k} x_i \text{ where } \sum_{i=1}^{k} x_i d_i = x \]

Use \( C(x, k) = \infty \) if \( x \) is not constructible.

Recursive formula:

\[ C(x, k) = \min \left( C(x, k-1), C(x-d_k, k+1) \right) \]

- Could use memoizing for a direct recursive implementation.

---

**Dynamic Programming**

Alternatively, we can build the \( n \times x \) table directly, in a row by row, left-to-right fashion.

Example: \( d_1 = 2, \ d_2 = 5, \ d_3 = 10. \)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| - | 1 | - | 2 | - | 3 | - | 4 | - | 5 | - | 6 | - | 7 | - | 8 |   |   |   |   |   |
| - | 1 | - | 2 | 1 | 3 | 2 | 4 | 3 | 2 | 4 | 3 | 5 | 4 | 3 | 5 |   |   |   |   |   |
| - | 1 | - | 2 | 1 | 3 | 2 | 4 | 3 | 1 | 4 | 2 | 5 | 3 | 2 | 4 |   |   |   |   |   |

Does not solve the problem of computing the actual \( x_k \). Easy to modify to get coefficients.

Alternatively, can reconstruct the coefficients from the table.

\( C(x, k) = C(x-d_k, k+1) + 1: \) add 1 to \( x_k \) in \( k \)-solution for \( x-d_k \)

Otherwise, same as for \( C(x, k-1) \).

Clearly, there is a better way to calculate the number of coins needed to construct \( x \).

Another interesting question is to count the number of way one can make change.

But we won’t get involved . . .
The diff Utility

A very real problem: comparing text files.
diff is a Unix utility that compares two files and lists their differences. (Aside: most Unix utilities developed in the 70’s focus on crucial ideas, well worth studying).

E.g. diff very useful to keep track of changes made in a program (revision control systems).
Or to compare output of two programs (your homework).

Basic idea: match as many whole lines in file A with lines appearing in the same order in file B.

Everything not matched is considered a discrepancy.
Can be construed as result of insertion/deletion.

In the end, return a list of all insert/delete operations that are needed to transform A into B.

Longest Common Subsequence

Consider a sequence \( A = (a_1, a_2, \ldots, a_n) \).
\( S = (s_1, s_2, \ldots, s_k) \) is a subsequence of \( A \) iff there is an index sequence \( 1 \leq p_1 < p_2 < \ldots < p_k \leq n \) such that
\[ a_{p_i} = b_i. \]

Note: scattered subsequence, need not be contiguous.

Given two sequences \( A \) and \( B \), \( S \) is a longest common subsequence (LCS) iff \( S \) is a subsequence of \( A \) and of \( B \), and has maximal length.

Example:
\( A = (a, b, c, b, d, a, b) \),
\( B = (b, d, c, a, b, a) \),
\( S = (b, d, a, b) \) or \( S = (b, c, b, a) \).

First Attack

Brute force: Compare all lines in file \( A \) with all lines in file \( B \), and determine which are shared between the two files.

Sounds only mildly promising: takes \( \Theta(nm) \) using two loops, (where \( m = \) length of \( A \), \( n = \) length of \( B \)) even if we assume comparison of two lines is \( O(1) \).

Also, not really enough: \( A = a, b, c, d \) and \( B = b, a, d, c \).

All lines match. We need to keep track of relative positions.

\[
\begin{array}{l}
1 & a & b \\
2 & b & a \\
3 & c & d \\
4 & d & c \\
\end{array}
\]

\( A, 1 = B, 2 \) and \( A, 4 = B, 3 \) OR
\( A, 2 = B, 1 \) and \( A, 3 = B, 4 \)

For dynamic programming we need a way to solve LCS for \( A \) and \( B \) by solving subproblems. Clearly, have to work on substrings of \( A \) and \( B \). But how?

Let \( A_i \) be the prefix of \( A \) of length \( i \).
\( A = (a_1, a_2, \ldots, a_n) \), \( B = (b_1, b_2, \ldots, b_n) \), \( S = (s_1, s_2, \ldots, s_k) \) a LCS of \( A \) and \( B \).

\[ \text{Crucial Insight:} \]
\[ \bullet a_m = b_n \text{ implies } S_k = a_m, S_{k-1}, \text{ LCS of } A_{m-1} \text{ and } B_{n-1}, \]
\[ \bullet a_m \neq b_n \text{ and } s_k \neq a_m \text{ implies } S \text{ is LCS of } A_{m-1} \text{ and } B, \]
\[ \bullet a_m \neq b_n \text{ and } s_k \neq b_n \text{ implies } S \text{ is LCS of } A \text{ and } B_{n-1}. \]
This yields a recursive description of LCSs.
Let \( a \neq b \) and write \( Aa \) for a sequence whose last term is \( a \).

\[
\begin{align*}
LCS(Aa, Ba) &= LCS(A, B) \iff a \\
LCS(Aa, Bb) &= LCS(A, Bb) \text{ or } LCS(Aa, B)
\end{align*}
\]

Read this with a grain of salt (\( LCS \) is not really a function, but a multi-valued function).

In some sense, this is the solution: we have a recursive function, can either use memoizing or dynamic programming to get reasonable efficiency.

But, there are a few details . . .

How about dynamic programming?
Compute array \( C \), initialize \( C(i, 0) = 0 \) and \( C(0, j) = 0 \).

```plaintext
for( i = 1; i <= m; ++i )
for( j = 1; j <= n; ++j )
  if( a[i] == b[j] )
    C[i,j] = C[i-1,j-1] + 1;
  else
    C[i,j] = max( C[i-1,j], C[i,j-1] );
```

Note: all values on right hand side are already computed.
This crucial for dynamic programming!

Running time is trivially \( \Theta(nm) \).

\[25\]

\[26\]

\[27\]

\[28\]
generateLCS( i, j )
{
  if( i == 0 || j == 0 )
    return empty sequence

  switch( T[i,j] )
  {
    case D:
      return Append( genLCS( i-1, j-1 ), a[i] );
    case U:
      return genLCS( i-1, j );
    case L:
      return genLCS( i, j-1 );
  }
}

Initial call is generateLCS( m, n ).

We can ditch the T array completely: can reconstruct entries from the C array. Cuts memory requirement in half.

Still expensive: for m and n around 10000 the C matrix alone still has size about 100,000,000.

If only length is needed, can store only (part of) one row and one column.

In industrial quality code (e.g., the real diff), have to make do with an approximation algorithm: produces a reasonably long common subsequence quickly. Match is not optimal, but works on very long files.

Example

A = 4, 2, 4, 1, 1, 3 and B = 4, 4, 1, 3, 3, 2, 1, 3.
S = 4, 4, 1, 1, 3

C matrix

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\
1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 \\
1 & 2 & 3 & 3 & 3 & 3 & 4 & 4 \\
1 & 2 & 3 & 4 & 4 & 4 & 4 & 5 \\
\end{array}
\]

T matrix

\[
\begin{array}{cccccccc}
D & D & L & L & L & L & L & L \\
U & U & U & U & U & D & L & L \\
D & D & L & L & L & U & U & U \\
U & U & D & L & L & L & D & L \\
U & U & D & U & U & U & D & L \\
U & U & U & D & D & L & U & D \\
\end{array}
\]

Suppose A is a sequence of integers of length n.

Find an \( O(n^2) \) algorithm to determine the longest, monotonically increasing subsequence of A.

Example:

A = 8, 1, 3, 10, 1, 3, 7, 1, 2, 6, 7, 3, 2, 4, 5, 2, 5, 10, 5, 2
produces
S = 1, 2, 3, 4, 5, 10
Recall the problem of matching lines in two text files $A$ and $B$. It is not enough to compute a LCS, but we must know where in the files these lines occur.

Easy: modify $\text{genLCS}(i, j)$:

```
... 
  case D: 
    return Append(genLCS(i-1, j-1), (ij)); 
...
```

Returns a list of pairs $(i, j)$ such $a_i = b_j$.

Example:

$A = 2, 1, 1, 1, 2, 1, B = 1, 4, 2, 4, 1, 2, 4, 2$ and $S = 2, 1, 2$.

Matching lines: $(1,3), (2,5), (5,8)$

---

**Example:** two short files.

```
this is line 1
this is line 2
this is line 3
this is line 4
this is line 5
this is line 6
this is line 7
this is line 8
this is line 9
this is line 10
```

Standard diff output.

```
2d1
< this is line 2
5c4
< this is line 3
--- 
> this is line 6
6a6,7
> this is line 9
> this is line 10
```

---

**Summary**

Find a way to express an optimal solution of problem in terms of solutions to subproblems.

Easily translated into a recursive function.

Multiple calls to same subproblem may occur, so must use memoizing.

If all (or at least many) possible subproblems actually occur in the computation, better to construct a table by hand.

- Table method more efficient than memoizing, code much simpler.
- Must be careful with order of computation.
Algorithm Design Techniques

Some general ideas on how to construct algorithms.

- Brute force
- Divide-and-conquer
- Dynamic programming
- Greedy algorithms
- Scan algorithms
- Graph exploration

General tools: recursion, memoizing (hashing).