Algorithms and Data Structures

Week 11 – Lecture A

Dynamic Programming Continued…

Dynamic Programming

- Last week we saw several “definitions” of Dynamic Programming:
  - Clever Brute Force;
  - Guessing + Recursion + Memoization;
  - Finding shortest path in problem dependency DAG.
- Remember: dynamic programming is not an algorithm; it is a technique for constructing algorithms.
- This week we will examine some more problems that can be solved using dynamic programming techniques.

DP Steps

- We can break the process into 5 general steps (with analysis):
  1. Define the sub-problems (determine number of sub-problems);
  2. Guess the solution to a sub-problem (determine number of guesses);
  3. Relate sub-problems via recursion (determine time per sub-problem);
  4. Solve the sub-problems systematically, ensure sub-problem dependency is acyclic (#sub-problems $\times$ time per sub-problem);
     1. Recurse and memoize (top down);
     2. Topological sort and loop (bottom up);
  5. Solve original problem by combining sub-problem solutions (O(1)).

Dynamic Programming IV
Text Justification

DP IV: Text Justification

- Break text into lines:
  - Justified (left and right edges are straight);
  - We want “good” lines – not too much white space.
- E.g.
  - Text: “The quick brown fox jumps over the unbelievably lazy dog.”
  - Lines:

  - Formally, given a sequence of *N* words, find the optimal set of line breaks, *i*, which maximize the total “goodness” of the resulting lines.

Greedy Algorithm

- The greedy strategy is obvious and easy to implement:
  - Pack as many words as possible onto each successive line;
  - If the next word doesn't fit start a new line.
- This strategy was used in Microsoft Word until recent versions.
- It often results in really ugly type.
- Newer versions of Word use a better algorithm:
  - "borrowed" from Donald Knuth's T_EX typesetting program;
  - Uses dynamic programming.

Defining the Problem

- Before we can solve this problem, we need to define what we mean by a good line.
- Let us consider a sequence of words, taken from the input text, starting at word i and ending at word $j-1$.
- We define *natural*(i, j) as the width of this sequence without justification.

- We also define *length* as the desired justified line length.

Badness

- We will now define *badness*(i, j), the measure of how well the sequence of words fits into the line, as follows:
  - *badness*(i, j) = (*length* – *natural*(I, j))^3 if *natural*(i, j) $\leq$ *length*;
  - = ∞ if *natural*(i, j) > *length.*
- The exact function used for calculating badness is somewhat arbitrary and will vary between software packages.
- We can now proceed to constructing our DP algorithm to solve the text justification problem.

Brute-Force Solution

- Let us start be looking at how we might solve the problem using brute force.
- We would consider every possible arrangement of our *N* words into lines.
- For each word we need to determine whether it starts a line:
  - This is a binary (yes/no) decision so, for *N* words, we have 2^N possible arrangements;
  - We compute the total badness for each arrangement and pick the minimum.

1. Identify the Sub-Problems

- We need to identify sub-problems in such a way that:
  - The best solution to the sub-problem is part of the overall solution;
  - The remaining part of the problem is of the same type as the original problem.
- The, not entirely, obvious sub-problem to consider is the contents of the *first* line of our justified text.
  - In other words, where does the second line of the justified text begin?
  - The sub-problem is to solve the justification problem for the text starting at some word, i, such that the badness of the current line plus the best solution for the remainder is minimized.
  - The second line can start at any word except the first.
  - Therefore, the number of sub-problems is N–1.

2. Guess

- Now we have defined the sub-problems, it should be obvious what we need to guess:
- Where do we start the next line?
- If our current sub-problem is to justify the list of words starting at word i then our guesses will be all the words after word i.
- We make all possible guesses, $j=i+1..N$, for the next break.
- This is $O(N)$ guesses.

3. Relate Sub-Problems via Recursion

- Our sub-problem involves finding the best choice of *j*, the word which starts the next line, given the list of words starting at word *i*.
- If **`just(i)`** is the desired solution, we can set up the following recursive procedure:
  - ```
    Procedure just(i)
     best = ∞
     if i == n+1 return 0
     for j = i+1, n+1
          if best > badness(i,j)+just(j) then
               best = badness(i,j)+just(j)
          fi
     rof
     return best
    end just
    ```
  - The running time per sub-problem, ignoring the recursion, is O(*N*).
    ```

4. Find the Topological Order

- In this case, to compute `just(i)`, we need to evaluate `just(j)` where $j > i$.
- This means that we need to evaluate the sub-problems in reverse order, starting at `just(n)` and working back to `just(1)`.
- The topological order is simply N, $N-1$, …, 2, 1.
- Our total running time:
  - # sub-problems $\times$ time per sub-problem;
  - $N \times O(N)$;
  - $O(N^2)$.

5. Solve the Original Problem

- It should now be clear, given the solutions to the sub-problems, how to solve the original problem:
  - It is simply to evaluate **`just(1)`**.
- Note: although, in our example, we return the total badness of the remaining justified text, in practice we need to also know *where* the line break occurs.
  - We can fix this, in the usual way, by recording *parent pointers*:
  - Remember which guess gave the best result
- Note: the detail of memoization and bottom up implementation is left as an exercise. ☺

Dynamic Programming V
Matrix Multiplication

Matrix Multiplication

- If we have two matrices (rectangular tables of values), A and B, we can compute the product as a third matrix, C, in the following way.
- Find the *dot product* between each row, i, in A and each column, j, in B. Record this as element (i, j) of C.
- The dot product is the sum of products of corresponding elements.
- For this process to be defined the number of columns in A must be the same as the number of rows in B.

An Example

- Let A and B be the following Matrices:
- $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$, $\mathrm{B} = \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$
- Then C is calculated as follows:
- $c(1,1) = 1 \times -1 + 2 \times 2 = 3$
- $c(1,2) = 1 \times 2 + 2 \times -1 = 0$
- $c(2,1) = 3 \times -1 + 4 \times 2 = 5$
- $c(2,2) = 3 \times 2 + 4 \times -1 = 2$

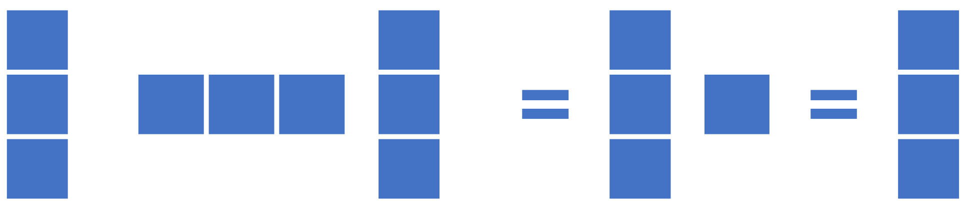
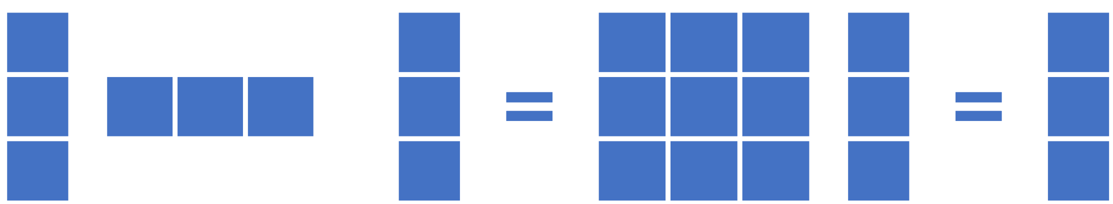
▶ $5 \quad -1 \quad 6 \quad 2$

▶ $5 \quad 2 \quad 6 \quad -1$

▶

▶

An Example

- Let A, B and C be three matrices with sizes 3 by 1, 1 by 3 and 3 by 1 respectively:
- ABC can be computed as A(BC) or (AB)C.
    - Let us look at both of these:
    - A(BC) 
    - (AB)C 

Generalization

- If we need to evaluate the product of n matrices:
- $A_0 \times A_1 \times \ldots \times A_{n\text{-}1}$
- We can perform this in $O(4^n)$ different ways:
  - Each has a potentially different cost.
- What is the minimum cost for the overall computation.
- In other words,
  - What is the optimum sequence of matrix multiplications to perform?
- Once again, we can solve this with dynamic programming.

DP V: Parenthesization

- We can restate the problem as follows:
- Given a sequence of *n* matrices; find the optimal locations for *n*-1 pairs of balanced parentheses, such that each pair contains exactly two matrices or parenthesized sets of matrices.
- E.g. given matrices ABCD, possible parenthesizations are:
- A(B(CD)), A((BC)D), (AB)(CD), (A(BC))D, ((AB)C)D
- Let us approach this problem in the same way as we have used with the other problems.

1, 2: Identify the Sub-Problems and Guess

- This is probably the most difficult part of this (and other) DP problems.
- In this case the sub-problem is best stated as:
  - Which is the *last* product to be computed?
- This is equivalent to splitting the sequence of matrices into two shorter sequences.
- There are O(n^2) sub-problems.
- I.e. find i where the two sequences are A[0, i) and A[i, n)
- The guess in this case should now be obvious:
  - The value of i; the location of the last multiplication pair.

3. Relate Sub-Problems via Recursion

- We can now produce the recursive solution in terms of a pair of functions:
  - **`cost(i,j,k)`**: the cost of performing the last multiplication of the sequence of matrices A[*i*, *j*) at location *k*;
  - **`best(i,j)`**: the minimum cost of multiplying together the sequence of matrices A[*i*, *j*).
  - **`best(i,j)`** is defined as follows:
  - ```
    Procedure best(i,j)
      if i==j return 0
      b=∞
      for k=i+1 to j-1
            b = min(b, cost(i,j,k)+best(i,k)+best(k,j))
      rof
      return b
    end best
    ```
  - The running time per sub-problem is O(*n*).
    ```

4, 5: Order and Final Solution

- The topological order is tricky:
  - Increasing sequence size. (Pairs, triples, etc.)
- The total cost is, once again, the number of sub-problems times the cost per sub-problem:
  - $O(n^2)$x $O(n)$;
  - $O(n^3)$.
- This compares favourably with brute force:
  - Try all possible parenthesizations;
  - $O(4^n)$ possibilities.
- The final problem solution is **`best(0,n)`**.