Dynamic Programming

Outline and Reading
- The General Technique (§5.3.2)
- 0-1 Knapsack Problem (§5.3.3)
- Matrix Chain-Product (§5.3.1)

Dynamic Programming revealed
- Break problem into subproblems that are
  - shared
  - have subproblem optimality (optimal subproblem solution helps solve overall problem)
  - subproblem optimality means can write recursive relationship between subproblems!
  - Defining subproblems is hardest part!
- Compute solutions to small subproblems
- Store solutions in array A.
- Combine already computed solutions into solutions for larger subproblems
- Solutions Array A is iteratively filled
- (Optional: reduce space needed by reusing array)

Computing Fibonacci
- Dynamic Programming is a general algorithm design paradigm:
  - Iteratively solves small subproblems which are combined to solve overall problem.
- Fibonacci numbers defined
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$, for $n > 1$
- Recursive solution:
  - int fib(int x)
    - if (x=0) return 0;
    - else if (x=1) return 1;
    - else return fib(x-1) + fib(x-2);
- Dynamic Programming Solution:
  - for i ← 2 to x do
    - f[i] ← f[i-1] + f[i-2];
  - return f[x];

Reducing Space for Computing Fibonacci
- store only previous 2 values to compute next value
  - int fib(x)
    - if (x=0) return 0;
    - else if (x=1) return 1;
    - else
      - int last ← 1; nextlast ← 0;
      - for i ← 2 to x do
        - temp ← last + nextlast;
        - nextlast ← last;
        - last ← temp;
      - return temp;

The General Dynamic Programming Technique
- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - Simple subproblems: the subproblems can be defined in terms of a few variables, such as $j$, $k$, $l$, $m$, and so on.
  - Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
  - Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).
The 0/1 Knapsack Problem

Given: A set S of n items, with each item i having
- bi - a positive benefit
- wi - a positive weight
Goal: Choose items with maximum total benefit but with weight at most W.
If we are not allowed to take fractional amounts, then this is the 0/1 knapsack problem.
In this case, we let T denote the set of items we take
Objective: maximize \( \sum_{i \in T} b_i \)
Constraint: \( \sum_{i \in T} w_i \leq W \)

Example

Given: A set S of n items, with each item i having
- bi - a positive benefit
- wi - a positive weight
Goal: Choose items with maximum total benefit but with weight at most W.

Example:

<table>
<thead>
<tr>
<th>Item</th>
<th>Benefit</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$20</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>$3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>$6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$25</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>$80</td>
<td>9</td>
</tr>
</tbody>
</table>

Items: 1, 2, 3, 4, 5

Weight: 4 in 2 in 2 in 6 in 2 in 9 in
Benefit: $20 $3 $6 $25 $80

Solution:
- 1 (2 in)
- 3 (2 in)
- 5 (2 in)
- 9 (4 in)

A 0/1 Knapsack Algorithm, First Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider S={S1,S2,S3,S4,S5} benefit-weight pairs

Best for S2:


Best for S3:

Towards the 0/1 Knapsack Algorithm

- S_k: Set of items numbered 1 to k = \{(b_1,w_1), (b_2,w_2), ..., (b_k,w_k)\}
- Define B[k,j] = maximum benefit of optimal subset from S_k with total weight at most j
- Recursive definition of B[k,j]:
  - \( B[k,j] = \begin{cases} 
  0 & \text{if } k = 0 \\
  \max\{B[k-1,j], B[k-1,j-w_k] + b_k\} & \text{otherwise} 
  \end{cases} \)

Towards the 0/1 Knapsack Algorithm

- B[k,j] = maximum benefit of optimal subset from S_k with total weight at most j
- Recursive version of algorithm based on recursive subproblem relationship.
- Not a dynamic programming version.
Towards the 0/1 Knapsack Algorithm

The 0/1 Knapsack Algorithm - Iterative

The 0/1 Knapsack Algorithm

The 0/1 Knapsack Algorithm - Iterative

The book version:

- When value does not change from one row to the next, then no need to assign same value.
- Running time: \(O(nW)\)
- Not a polynomial-time algorithm if \(W\) is large
- This is a pseudo-polynomial time algorithm

Dynamic Programming version 1.4 16

Dynamic Programming version 1.4 15

Dynamic Programming version 1.4 13

Dynamic Programming version 1.4 17

Dynamic Programming version 1.4 18

Dynamic Programming version 1.4 19
A simple version:
- letters and spaces have equal width
- input is set of \( n \) word lengths, \( w_1, w_2, \ldots w_n \)
- also given line width limit \( L \)
- each length \( w_i \) includes one space
- Placing words \( i \) up to \( j \) on one line means \( \sum_{k=i}^{j} w_k \leq L \)
- Penalty for extra spaces \( X = L - \sum_{k=i}^{j} w_k \) is \( X^2 \)
- Minimize sum of penalties from each line (no last line penalty)

Example problem
- Paragraph is:
  Those who cannot remember the past are condemned to repeat it.
- Word lengths are 6,4,7,9,4,5,4,10,3,7,4.
- Suppose line width \( L = 17 \).
- Find an optimal way of separating words into lines that minimizes penalty.

**Linebreak DP**
- for \( i \leftarrow n-1 \) downto 0 do
  - if \( (w[i] + w[i+1] + \ldots + w[n-1] < L) \)
    - \( \text{lineB}[i] \leftarrow 0; \)
  - else
    - \( \text{mincost} \leftarrow \infty; \)
    - \( k \leftarrow 1; \)
    - while \( (k \text{ words starting from } w[i] \text{ fit on a line}) \)
      // meaning \( (w[i] + w[i+1] + \ldots + w[i+k-1] \leq L) \)
    - \( \text{linecost} \leftarrow \text{penalty from placing words } w[i] \text{ to } w[i+k-1] \)
      on one line.
    - \( \text{totalcost} \leftarrow \text{linecost} + \text{lineB}[i+k]; \)
    - \( \text{mincost} \leftarrow \min(\text{totalcost}, \text{mincost}) \) // track min. so far
    - \( k++; \)
    - \( \text{lineB}[i] = \text{mincost}; \)

**Linebreak DP cost**
- \( O(nL); L \) is maximum width
- Linear if \( L \) is considered constant
- Space \( O(n) \).

**Matrix Chain-Products**
- Review: Matrix Multiplication.
  - \( C = A \times B \)
  - \( A \) is \( d \times e \) and \( B \) is \( e \times f \)
  - \( C[i,j] = \sum_{k=0}^{e} a[i,k] \times b[k,j] \)
- \( O(\text{def}) \) time \( (\text{def} \text{ multiplications}) \)

- **Example**
  - \( B \) is \( 3 \times 100 \)
  - \( C \) is \( 100 \times 5 \)
  - \( D \) is \( 5 \times 5 \)
  - \((B \times C) \times D\) takes 1500 + 75 = 1575 ops
  - \( B \times (C \times D)\) takes 1500 + 2500 = 4000 ops
### An Enumeration Approach

**Matrix Chain-Product Alg.:**
- Try all possible ways to parenthesize $A = A_0 \times A_1 \times \ldots \times A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

**Running time:**
- The number of paranethesizations is equal to the number of binary trees with $n$ nodes
- This is **exponential**!
- It is called the Catalan number, and it is almost $4^n$.
- This is a terrible algorithm!

### A Greedy Approach

**Idea #1:** repeatedly select the product that uses (up) the most operations.

**Counter-example:**
- A is $10 \times 5$
- B is $5 \times 10$
- C is $10 \times 5$
- D is $5 \times 10$
- Greedy idea #1 gives $(A \times B) \times (C \times D)$, which takes $500 + 1000 + 500 = 2000$ ops
- $A \times ((B \times C) \times D)$ takes $500 + 250 + 250 = 1000$ ops

### A “Recursive” Approach

**Idea #2:** repeatedly select the product that uses the fewest operations.

**Counter-example:**
- A is $101 \times 11$
- B is $11 \times 9$
- C is $9 \times 100$
- D is $100 \times 99$
- Greedy idea #2 gives $A \times ((B \times C) \times D)$, which takes $9999 + 89991 + 89100 = 189090$ ops
- $(A \times B) \times (C \times D)$ takes $9999 + 89991 + 89100 = 189090$ ops

The greedy approach is not giving us the optimal value.

### A Characterizing Equation

Define global optimal in terms of optimal subproblems, by checking all possible locations for final multiply.
- Recall that $A_i$ is a $d_{i-1} \times d_i$ dimensional matrix.
- So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_id_{k+1}d_{j+1}\}$$

Note that subproblems are not independent—the subproblems overlap (are shared)

### A Dynamic Programming Algorithm

**Algorithm matrixChain(m):**
- Input: sequence $S$ of $n$ matrices to be multiplied
- Output: number of operations in an optimal parenthesization of $S$

```
for i = 0 to n-1 do
    N[i][i] = 0
for b = 0 to n-1 do
    for i = 0 to n-b-1 do
        j = i + b
        N[i][j] = infinity
        for k = i to j do
            N[i][j] = min(N[i][j], N[i][k-1] + N[k][j] + d_id_{k+1}d_{j+1})
```

Running time: $O(n^3)$
A Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the $N$ array by diagonals.
- $N_{ij}$ gets values from previous entries in $i$-th row and $j$-th column.
- Filling in each entry in the $N$ table takes $O(n)$ time.
- Total run time: $O(n^3)$.
- Getting actual parenthesization can be done by remembering "k" for each $N$ entry.

$N_{ij} = \min \{ N_{ik} + N_{kj} + d_i d_k d_j \}$