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 realtionship 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

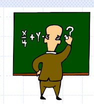
- Dynamic Programming is a general algorithm design paradigm:
  - Iteratively solves small subproblems which are combined to solve overall problem.
- Fibonacci numbers defined
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$, for n > 1
- Recursive solution:
  - int fib(int x)
    if (x=0) return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

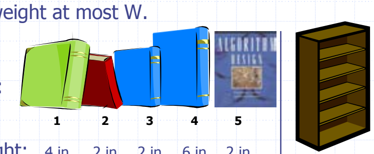
The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - 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: $\sum_{i \in T} w_i \le W$

Example

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

Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

"knapsack"

9 in

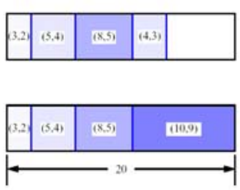
Solution:
- 5 (2 in)
- 3 (2 in)
- 1 (4 in)

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={(3,2),(5,4),(8,5),(4,3),(10,9)} benefit-weight pairs

Best for S_4:

Best for S_5:



A 0/1 Knapsack Algorithm, Second Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k,w] = best selection from S_k with weight exactly equal to w
- Good news: this does have subproblem optimality:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., best subset of S_k with weight limit exactly w is either the best subset of S_{k-1} w/ weight w or the best subset of S_{k-1} w/ weight w-w_k plus benefit of item k.

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]:

$$B[k,j] = \begin{cases} 0 & \text{if } k=0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j],\, B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

Towards the 0/1 Knapsack Algorithm

$$B[k,j] = \begin{cases} 0 & \text{if } k=0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j],\, B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- 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.

```
Algorithm rec01Knap(S, W):
  Input: set S of k items w/ benefit b_1, b_2, …
    b_k; weights w_1, w_2, … w_kj and max.
    weight W
  Output: benefit of best subset with
    weight at most W
  if k=0 then   {S = emptyset}
    return 0
  remove item k (benefit-weight (b_k,w_k))
    from S
  if w_k > W then  {item k does not fit}
    return rec01Knap(S,W)
  return max(rec01Knap(S,W),
    rec01Knap(S,W-w_k) + b_k)
```

Towards the 0/1 Knapsack Algorithm

$$B[k,j] = \begin{cases} 0 & \text{if } k = 0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j], B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- Modified recursive version that stores subproblem solutions
  - First allocate global array B of size n+1 by W
  - Then initialize all entries of B[i,j] to −1
  - B stores results of recursive calls
  - Entries in B are computed when necessary
- This is considered a dynamic programming version.

Algorithm *rec01Knap(S, W)*:
  Input: set S of k items w/ benefit $b_1, b_2, \ldots, b_k$; weights $w_1, w_2, \ldots w_k$; and max. weight W
  Output: benefit of best subset with weight at most W
  if k=0 **then return 0**
  remove item **k (benefit-weight** (b_k,w_k)) from **S**
  if B[k-1, W]= −1 **then** B[k-1,W]=*rec01Knap(S,W)*
  if $w_k > W$ **then**
    return B[k-1, W]
  if B[k-1, W- w_k]= −1 **then**
    B[k-1,W - w_k]=*rec01Knap(S,W -w$_k$)*
  return max(B[k-1, W], B[k-1,W - w_k]+b_k)

The 0/1 Knapsack Algorithm- Iterative

$$B[k,j] = \begin{cases} 0 & \text{if } k = 0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j], B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- Recursive computation not necessary
- Compute iteratively, bottom-up
- All B[k-1,*] must be computed before B[k,*] because of subproblem dependencies
- This is also dynamic programming.

Algorithm *01Knapsack(S, W)*:
  Input: set S of n items w/ benefit b_i and weight w_i; max. weight W
  Output: benefit of best subset with weight at most W
  for $w \leftarrow 0$ **to** W **do** {base case}
    B[0,w] ← 0
  for $k \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** W **do**
      if $w_k > j$ **then**
        B[k,j] ← B[k-1,j]
      else
        B[k,j] ← max(B[k-1,j], B[k-1,j-w_k]+b_k)

The 0/1 Knapsack Algorithm- Iterative

$$B[k,j] = \begin{cases} 0 & \text{if } k = 0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j], B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- Not necessary to use all the space
- Keep track of one row at a time
- Overwrite results from previous row as new values computed
- Must compute right to left (W downto 1) so that the next row (B[k,*]) uses results from the previous row (B[k-1,*]).
- Simplify this to get version in book.

Algorithm *01Knapsack(S, W)*:
  Input: set S of n items w/ benefit b_i and weight w_i; max. weight W
  Output: benefit of best subset with weight at most W
  for $w \leftarrow 0$ **to** W **do** {base case}
    B[~~0,~~ w] ← 0
  for $k \leftarrow 1$ **to** n **do**
    for $j \leftarrow W$ **downto** 1 **do**
      if $w_k > j$ **then**
        B[~~k,~~ j] ← B[~~k-1,~~ j]
      else
        B[~~k,~~ j] ← max(B[~~k-1,~~ j], B[~~k-1,~~ j-w_k]+b_k)

The 0/1 Knapsack Algorithm- Iterative

$$B[k,j] = \begin{cases} 0 & \text{if } k = 0 \\ B[k-1,j] & \text{if } w_k > j \\ \max\{B[k-1,j], B[k-1,j-w_k]+b_k\} & \text{otherwise} \end{cases}$$

- Not necessary to use all the space
- Keep track of one row at a time
- Overwrite results from previous row as new values computed
- Must compute right to left (W downto 1) so that the next row (B[k,*]) uses results from the previous row (B[k-1,*]).
- Simplify this to get version in book.

Algorithm *01Knapsack(S, W)*:
  Input: set S of n items w/ benefit b_i and weight w_i; max. weight W
  Output: benefit of best subset with weight at most W
  for $w \leftarrow 0$ **to** W **do** {base case}
    B[w] ← 0
  for $k \leftarrow 1$ **to** n **do**
    for $j \leftarrow W$ **downto** 1 **do**
      if $w_k > j$ **then**
        B[j] ← B[j]
      else
        B[j] ← max(B[j], B[j-w_k]+b_k)

The 0/1 Knapsack Algorithm

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- 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

Algorithm *01Knapsack(S, W)*:
  Input: set S of n items w/ benefit b_i and weight w_i; max. weight W
  Output: benefit of best subset with weight at most W
  for $w \leftarrow 0$ **to** W **do**
    B[w] ← 0
  for $k \leftarrow 1$ **to** n **do**
    for $w \leftarrow W$ **downto** w_k **do**
      if B[w-w_k]+b_k > B[w] **then**
        B[w] ← B[w-w_k]+b_k

line-breaking problem

- Given sequence of words from one paragraph
- Return where line-breaks should occur
- Minimize empty space on each line (except for last line of paragraph)

line-breaking problem

- A simple version:
  - letters and spaces have equal width
  - input is set of n word lengths, w_1, w_2, … 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

    $$\sum_{k=i}^{j} w_i \le L$$

  - Penalty for extra spaces $X = L - \sum_{k=i}^{j} w_i$ is X^3
  - 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 ← n-1 downto 0 do
   if (w[i] + w[i+1] + … + w[n-1] < L)
      lineB[i] ← 0;
   else
      mincost ← Infinity;
      k ← 1;
      while (k words starting from w[i] fit on a line)
              // meaning (w[i] + w[i+1] + … + w[i+k-1] <= L)
        linecost ← penalty from placing words w[i] to w[i+k-1]
              on one line.
        totalcost ← linecost + lineB[i+k];
        mincost ← min(totalcost, mincost)  // track min. so far
        k++;
      lineB[i]=mincost;
```


linebreak DP cost

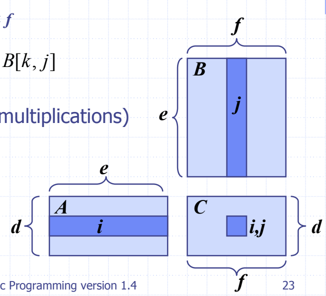
- O(nL); L is maximum width
- Linear if L is considered constant
- Space O(n).

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

    $$C[i,j] = \sum_{k=0}^{e-1} A[i,k]*B[k,j]$$

  - $O(def)$ time (def multiplications)



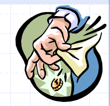
Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A = A_0*A_1*…*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize? [for minimizing ops]
- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - 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 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

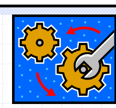
Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

A "Recursive" Approach

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If subproblems were not optimal, neither is global solution.

A Characterizing Equation

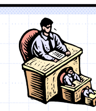


- Define global optimal in terms of optimal subproblems, by checking all possible locations for final multiply.
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note that subproblems are not independent--the **subproblems overlap** (are shared)

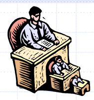
A Dynamic Programming Algorithm



- Construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- Array $N_{i,j}$ stores solutions
- Running time: $O(n^3)$

Algorithm *matrixChain*(*S*):
Input: sequence *S* of *n* matrices to be multiplied
Output: number of operations in an optimal paranthesization of *S*
for $i \leftarrow 1$ **to** $n-1$ **do**
  $N_{i,i} \leftarrow 0$
for $b \leftarrow 1$ **to** $n-1$ **do**
  for $i \leftarrow 0$ **to** $n-b-1$ **do**
    $j \leftarrow i+b$
    $N_{i,j} \leftarrow$ +**infinity**
    for $k \leftarrow i$ **to** $j-1$ **do**
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

A Dynamic Programming Algorithm Visualization



$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious 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