Introduction to Algorithms

Department of Computer Science and Engineering

East China University of Science and Technology

Lecture 05

Dynamic programming

Today's Tasks

- Dynamic Programming
  - Optimal substructure
  - Overlapping subproblems

Dynamic programming

- **Dynamic programming** is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.
- In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

Dynamic Programming

- Dynamic programming (dynamic programming) is a branch of operational research, is to solve the decision-making process (decision Process) mathematical optimization methods.
- At the beginning of the 1950s R. E. Bellman et al in the research on multi stage decision process optimization problem, put forward the principle of optimality of the famous principle (of Optimality), the multi stage process into a series of single stage, one by one solution, created to solve this kind of process.

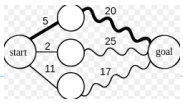
Dynamic Programming

- If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems.
- In the optimization literature this relationship is called the Bellman equation.
- Originally introduced by Richard E. Bellman in (Bellman 1957)

Dynamic Programming

- "Programming" often refer to Computer Programming.
  - But it's not always the case, such as Linear Programming, Dynamic Programming.
- "Programming" here means Design technique, it's a way of solving a class of problems, like divide-and-conquer.

Multi-stage Decision



- **Figure 1.** Finding the shortest path in a graph using optimal substructure; a straight line indicates a single edge;
- a wavy line indicates a shortest path between the two vertices it connects (among other paths, not shown, sharing the same two vertices);
- the bold line is the overall shortest path from start to goal.

dynamic programming--two key attributes

- There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.
- If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called "divide and conquer" instead .
- This is why merge sort and quick sort are not classified as dynamic programming problems.

optimal substructure

- Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.
- For example, given a graph G=(V,E), the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p.
- If p is truly the shortest path, then it can be split into sub-paths p1 from u to w and p2 from w to v such that these, in turn, are indeed the shortest paths between the corresponding vertices (by the simple cut-and-paste argument described in Introduction to Algorithms). Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman–Ford algorithm or the Floyd–Warshall algorithm does.

Overlapping sub-problems

- Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.
- For example, consider the recursive formulation for generating the Fibonacci series: Fi = Fi−1 + Fi−2, with base case F1 = F2 = 1. Then F43 = F42 + F41, and F42 = F41 + F40. Now F41 is being solved in the recursive sub-trees of both F43 as well as F42. Even though the total number of sub-problems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once.

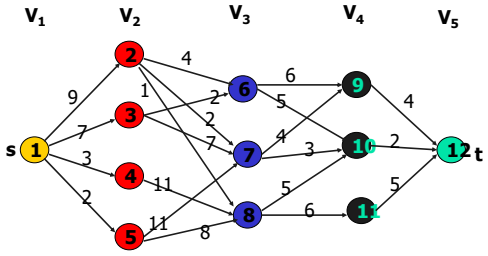
This can be achieved in either of two ways:Top-down approach

- Top-down approach: This is the direct fall-out of the recursive formulation of any problem.
- If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

This can be achieved in either of two ways:Bottom-up approach

- Bottom-up approach: Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F41 and F40, we can directly calculate the value of F42.

The multi stage decision



The multi stage decision

- The multisegment graph problem is the minimum cost path from s to t.

Example

Longest Common subsequence

Example: LCS

- Longest Common Subsequence (LCS)
  - Which is a problem that comes up in a variety of contexts： Pattern Recognition in Graphics, Revolution Tree in Biology, etc.
  - Given two sequences x[1 . . m]and y[1 . . n], find a longest subsequence common to them both.
- Why we address “a” but not “the”?
  - Usually the longest comment subsequence isn't unique.

Longest Common subsequence

- **Two possible DNA string;**
  - **S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA**
  - **S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA**
- **Our goal: Compare the two string to determine how “similar” the two strands are.**
- **Issue: The similarity can be defined in many ways, here we use the “longest common subsequence” to describe the similarity**
- **GTCGTCGGAAGCCGCCGAA**

Concepts

- • Subsequence and substring
- – A substring of a string is a continuous part of the string
- – A subsequence of a string x is a string y with its elements in x in the same order.

Example: LCS

- Longest Common Subsequence (LCS)
  - Which is a problem that comes up in a variety of contexts：Pattern Recognition in Graphics, Revolution Tree in Biology, etc.
  - Given two sequences x[1 . . m]and y[1 . . n], find a longest subsequence common to them both.
- Why we address “a” but not “the”?
  - Usually the longest comment subsequence isn't unique.

Sequence Pattern Matching

- Find the first appearance of string T in string S.
- S: a b a b c a b c a c b a b
- T: a b c a c
- BF Thought:
  - Pointer i and j to indicate current letter in S and T
  - When mismatching, j always backstep to 1
  - How about i?
  - i-j+2

Sequence Matching Function

```
int Index(String S,String T, int pos)
{
  i=pos; j=1;
      while(i<= length(S) && j<= length(T))
      {
        if(S[i]==T[j]){++i;++j;}
        else {i=i+j-2;j=1；}
      }
      if(j>length(T))
    return (i-length(T));
  else return 0;
}
```

Analysis of BF Algorithm

- What’s the worst time complexity of BF algorithm for string S and T of length m and n ?
- Thought of case

  S=00000000000000000000000001

  T=0000001
- How to improve it?
  - KMP Algorithm

KMP algorithm of T: abcac

- S: a b a b c a b c a c b a b
- T: a b c

  ↕i

  ↕j
- S: a b a b c a b c a c b a b
- T: a b c a c
- S: a b a b c a b c a c b a b
- T: (a)b c a c

How to do it?

- When mismatching, i does not backstep, j backstep to some place particular to structure of string T.
- T: a b a a b c a c
- next[j]: 0 1 1 2 2 3 1 2
- T: a a b a a d a a b a a b
- next[j]: 0 _ _ _ _ _ _ _ _ _ _ _
- How to get next[j] given a string T?

How to do it?

- When mismatching, i does not backstep, j backstep to some place particular to structure of string T.

$$next[j]=\begin{cases}0 & \text{while } j=1\\ Max\{k|1<k<j \wedge t_1t_2\ldots t_{k-1}=t_{j-(k-1)}\,t_{j-k}\ldots\,t_{j-1}\} & \text{it's not an empty set}\\ 1 & \text{other case}\end{cases}$$

Get next[j]

j:	1	2	3	4	5	6	7	8
T:	a	b	a	a	b	c	a	c
next[j]:	0	1	1	2	2	3	1	2

- Next[1]=0
- Suppose next[j]=k, $t_1t_2\ldots t_{k-1}= t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$ next[j+1]=?
- if $t_k=t_j$, next[j+1]=next[j]+1
- Else treat it as a sequence matching of T to T itself, we have $t_1t_2\ldots t_{k-1}= t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$ and $t_k\neq t_j$,so we should compare $t_{next[k]}$ and t_j. If they are equal, next[j+1]=next[k]+1, else backstep to next[next[k]], and so on.

Implementation of next[j]

```
void next(string T, int next[])
{
  i=1;j=0; next[1]=0;
  while (i<length(T))
  {
    if(j==0 or T[i]==T[j])
    {
      ++i;++j;next[i]=j;
    }
    else j=next[j];
  }
}
```

- Analysis of KMP

Longest Common Subsequence

- x: A B C B D A B
- y: B D C A B A
- What is a longest common subsequence?
  - BD?
  - Extend the notation of subsequence. Of the same order, but not necessarily successive.
  - BDA?BDB?BCBA?BCAB?
  - Is there any of length 5?
  - We can mark BCBA and BCAB with functional notation LCS(x,y), but it's not a function.

Brute-force LCS algorithm

- How to find a LCS?
  - Check every subsequence of x[1 . . m]to see if it is also a subsequence of y[1 . . n].
- Analysis
  - Given a subsequence of x, such as BCBA , How long does it take to check whether it's a subsequence of y?
  - O(n) time per subsequence.

Analysis of brute LCS

- Analysis
- How many subsequences of x are there?
- 2^m subsequences of x.
- Because each bit-vector of length m determines a distinct subsequence of x.
- So, worst-case running time is ?
- $O(n2^m)$,which is an exponential time.

Towards a better algorithm

- Simplification:
  - Look at the length of a longest-common subsequence.
  - Extend the algorithm to find the LCS itself.
- Now we just focus on the problem of computing the length.
- Notation: Denote the length of a sequence s by |s|.
- We want to compute is |LCS(x,y)| . How can we do it?

Towards a better algorithm

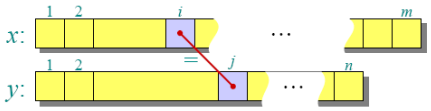
- Strategy: Consider prefixes of x and y.
  - Define c[i, j] = |LCS(x[1 . . i], y[1 . . j])|. And we will calculate c[i,j] for all i and j.
  - If we reach there, how can we solve the problem of |LCS(x, y)|?
  - Simple, |LCS(x, y)|=c[m,n]

Towards a better algorithm

- Theorem.

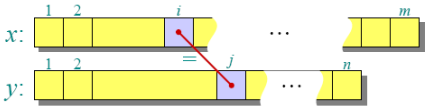
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise.} \end{cases}$$

- That's what we are going to prove.
- Proof.
- Let's start with case x[i] = y[j]. Try it.



Towards a better algorithm

- Suppose c[i, j] = k, and let z[1 . . k] = LCS(x[1 . . i], y[1 . . j]). what z[k] here is?
- Then, z[k] = x[i] (= y[j]), why?
- Or else z could be extended by tacking on x[i] and y[j].
- Thus, z[1 . . k–1] is CS of x[1 . . i–1] and y[1 . . j–1]. It's obvious to us.



A Claim easy to prove

- Claim: z[1 . . k–1] = LCS(x[1 . . i–1], y[1 . . j–1]).
- Suppose w is a longer CS of x[1 . . i–1] and y[1 . . j–1].
- That means |w| > k–1.
- Then, cut and paste: w||z[k] is a common subsequence of x[1 . . i] and y[1 . . j] with | w||z[k] | > k.
- Contradiction, proving the claim.

Towards a better algorithm

- Thus, c[i–1, j–1] = k–1, which implies that c[i, j] =c[i–1, j–1] + 1.
- The other case is similar. Prove by yourself.
- Hints:
- if z[k] = x[i], then z[k] ≠ y[j], c[i,j]=c[i,j-1]
- else if z[k] = y[j], then z[k] ≠ x[i], c[i,j]=c[i-1,j]
- else c[i,j]=c[i,j-1] = c[i-1,j]

Dynamic-programming hallmark

- **Dynamic-programming hallmark #1**

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Optimal substructure

- In problem of LCS, the base idea:
- If z= LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.
- If the substructure were not optimal, then we can find a better solution to the overall problem using cut and paste.

Recursive algorithm for LCS

- LCS(x, y, i, j) //ignoring base case

```
if x[i] = y[ j]
    then c[i, j] ←LCS(x, y, i–1, j–1) + 1
    else c[i, j] ←max{ LCS(x, y, i–1, j) ,
                       LCS(x, y, i, j–1) }
return c[i,j]
```

- What's the worst case for this program?
  - Which of these two clauses is going to cause us more headache?
  - Why?

the worst case of LCS

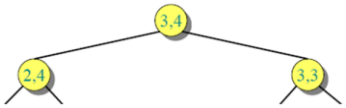
- The worst case is x[i] ≠ y[j] for all i and j
- In which case, the algorithm evaluates two sub problems, each with only one parameter decremented.
- We are going to generate a tree.

Recursion tree

- m=3, n=4

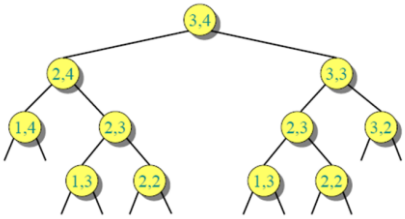
Recursion tree

- m=3, n=4



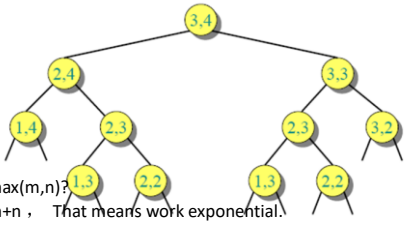
Recursion tree

- m=3, n=4



Recursion tree

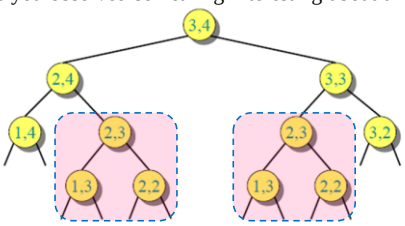
- What is the height of this tree?



- max(m,n)?
- m+n， That means work exponential.

Recursion tree

- Have you observed something interesting about this tree?



- There's a lot of repeated work. The same subtree, the same subproblem that you are solving.

Repeated work

- When you find you are repeating something, figure out a way of not doing it.
- That brings up our second hallmark for dynamic programming.

Dynamic-programming hallmark

- **Dynamic-programming hallmark #2**

Overlapping subproblems
A recursive solution contains a "small" number of distinct subproblems repeated many times.

Overlapping subproblems

- The number of nodes indicates the number of subproblems. What is the size of the former tree?
- 2^{m+n}
- What is the number of distinct LCS subproblems for two strings of lengths m and n?
- m*n.
- How to solve overlapping subproblems?

Memoization

- Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.
- Here is the improved algorithm of LCS. And the basic idea is keeping a table of c[i,j].

Improved algorithm of LCS

- LCS(x, y, i, j) //ignoring base case

```
if c[i, j] = NIL
  then if x[i] = y[ j]
        then c[i, j] ←LCS(x, y, i–1, j–1) + 1
       else c[i, j] ←max{ LCS(x, y, i–1, j) ,
                          LCS(x, y, i, j–1) }
  return c[i, j]
```

Same as before

- How much time does it take to execute?

Analysis

- Time = Θ(mn), why?
- Because every cell only costs us a constant amount of time .
- Constant work per table entry, so the total is Θ(mn)
- How much space does it take?
- Space = Θ(mn).

Dynamic programming

- Memoization is a really good strategy in programming for many things where, when you have the same parameters, you're going to get the same results.
- Another strategy for doing exactly the same calculation is in a bottom-up way.
- IDEA of LCS: make a c[i,j] table and find an orderly way of filling in the table: compute the table bottom-up.

Dynamic-programming algorithm

-

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

Dynamic-programming algorithm

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Time = ?

Θ(mn)

Dynamic-programming algorithm

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

- How to find the LCS ?

Dynamic-programming algorithm

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

- Reconstruct LCS by tracing backwards.

Dynamic-programming algorithm

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

And this is just one path back. We could have a different LCS.

Cost of LCS

- Time = Θ(mn).
- Space = Θ(mn).
- Think about that:
  - Can we use space of Θ(min{m,n})?
- In fact, We don't need the whole table.
- We could do it either running vertically or running horizontally, whichever one gives us the smaller space.

Further thought

- But we can not go backwards then because we've lost the information in front rows.
- HINT: Divide and Conquer.

Have FUN !

example

- *X* = *A*, *B*, *C*, *B*, *D*, *A*, *B*
- *Z* = *B*, *C*, *D*, *B* a subsequence of X
- with corresponding index sequence 2, 3, 5, 7.

example

- *X* = *A*, *B*, *C*, *B*, *D*, *A*, *B*
- *Y* = *B*, *D*, *C*, *A*, *B*, *A*,
- *B, C, A* is a common subsequence of both *X* and *Y* ;
- *B, C, B, A* is also common subsequence of both *X* and *Y* .
- sequence *B, C, B, A* is an LCS of *X* and *Y,* since there is no common subsequence of length 5 or greater

Longest Common subsequence problem

- Given two sequences
  - $X=<x_1, x_2, x_3,\ldots,x_m>$ **x[1..m]**
  - $Y=<y_1, y_2, y_3,\ldots,y_n>$ **y[1..n]**
- Find the longest common subsequence of both of the two given strings.

Step1: Analyzing the problem

- A brute-force method:
- Check every subsequence of x[1..m] to see if it is also a subsequence of y[1..n].
  - How many subsequences for string X?
  - Check each one is also a subsequence of string Y.
  - Complexity?

brute-force method

- **Analysis**
  - Checking = O(n) time per subsequence.
  - 2^m subsequences of.
  - Worst-case running time = $O(n2^m)$
    = exponential time

Towards a better algorithm

- **Simplification:**

1. Look at the **length** of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

- Notation: Denote the length of a sequences *by* $|s|$.
- Strategy: Consider prefixes of *x and y.*

– Define c[i, j] = |LCS(x[1..i]), y[1..j])|.

– Then, c[m, n] = |LCS(x, y)|.

Step 2: A recursive solution

Theorem.

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

Proof:

Optimal substructure of an LCS

- Let X=<x_1,x_2,…,x_m> ,Y=<y_1,y_2,…,y_n> be two sequences and let Z=<z_1,z_2,…,z_k> be a LCS of X and Y.

– If x_m=y_n, then z_k=x_m=y_n and Z_{k-1} is an LCS of X_{m-1} andY_{n-1}

– If x_m !=y_n, then z_k != x_m implies that Z_k is an LCS of X_{m-1} and Y_n

– If x_m != y_n, then z_k != y_n implies that Z_k is an LCS of X_m and Y_{n-1}

Example 3

Matrix-chain multiplication

Matrix Chain-Products

- Review: Matrix Multiplication.
- ***C*** = ***A*********B***
- ***A*** is ***p*** × ***q*** and ***B*** is ***q*** × ***r***

MATRIX-MULTIPLY(*A*, *B*)

- **if** *columns*[*A*] ≠ *rows*[*B*]
- **then error** "incompatible dimensions"
- **else for** *i* ← 1 **to** *p*
- **for** *j* ← 1 **to** *r*
- *C*[*i*, *j*] ← 0 ***O*(*p*·*q*·*r*)** time
- **for** *k* ← 1 **to** *q*
- *C*[*i*, *j*] ← *C*[*i*, *j*] + *A*[*i*, *k*] · *B*[*k*, *j*]
- **return** *C*

Matrix-chain multiplication

Goal : given a sequence of matrices $A_1, A_2, \ldots A_n$, find an optimal order of multiplication.

Generally. A_i has dimension $p_{i-1} * p_i$ and we'd like to minimize the total number of scalar multiplications.

Example

- A_1 is 3 × 100 A_2 is 100 × 5 A_3 is 5 × 5
- $(A_1 * A_2) * A_3$ takes
- 1500 + 75 = 1575 ops
- $A_1 * (A_2 * A_3)$ takes
- 1500 + 2500 = 4000 ops

Problem: How to parenthesize?

Brute force method

Matrix Chain-Product Alg.:

- Try all possible ways to parenthesize

$A = A_1 A_2 A_3 \ldots A_n$

P(n):number of way of parenthesizing.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- Catalan number grows as $\Omega(4^n/n^{3/2})$

Step1: Optimal substructure

- Define **subproblems**:
- Find the best parenthesis of $A_i A_{i+1} A_{i+2} \ldots A_j$.
- Notation: $A_{i,j}$ represents $A_i \ldots A_j$
- Any parenthesis of $A_{i,j}$ where i<j must split into two products of the form $A_{i,k}$ and $A_{k+1,j}$
- Optimal substructure: If the optimal parenthesis splits the product as $A_{i,k}$ and $A_{k+1,j}$ then parenthesis for the sub-problems $A_{i,k}$ and $A_{k+1,j}$ must each be optimal.

Step 2: A recursive solution

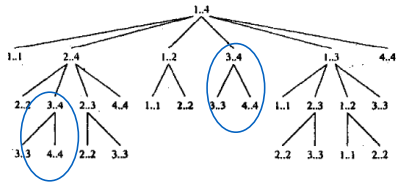
Let m[i,j] denote the number of operations done by this subproblem.

m[1,n]:The optimal solution for the whole problem

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

❖Note that subproblems are not independent–the **subproblems overlap**.

Overlapping subproblems

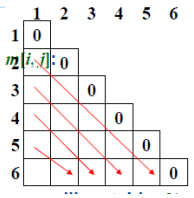


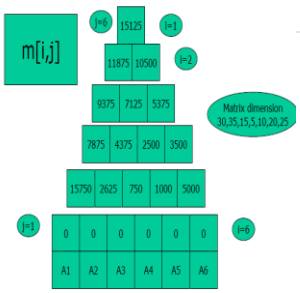
RECURSIVE-MATRIX-CHAIN(p, i, j)

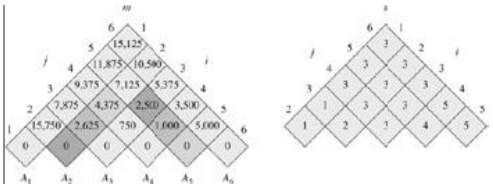
- 1 **if** $i = j$
- 2 **then return** 0
- 3 $m[i, j] \leftarrow \infty$
- 4 **for** $k \leftarrow i$ **to** j - 1
- 5 **do** $q \leftarrow$ RECURSIVE-MATRIX-CHAIN(p, i, k) + RECURSIVE-MATRIX-CHAIN(p,k + 1, j)+ $p_{i-1} p_k p_j$
- 6 **if** $q < m[i, j]$
- 7 **then** $m[i, j] \leftarrow q$
- 8 **return** $m[i, j]$

Step 3: Computing the optimal costs

- **MATRIX-CHAIN-ORDER(p)**
- 1 $n \leftarrow length[p]$ - 1
- 2 **for** $i = 1$ **to** n
- 3 do $m[i, i] = 0$
- 4 **for** $l = 2$ **to** n ▷l is the chain length.
- 5 do **for** $i = 1$ **to** $n - l + 1$
- 6 do $j = i + l - 1$
- 7 $m[i, j] = \infty$
- 8 **for** $k = i$ **to** $j - 1$
- 9 do $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
- 10 **if** $q < m[i, j]$
- 11 **then** $m[i, j] = q$
- 12 $s[i, j] = k$
- 13 **return** m and s







***A1*: 30 × 35**
***A2*: 35 × 15**
***A3*: 15 × 5**
***A4*: 5 × 10**
***A5*: 10 × 20**
***A6*: 20 × 25**

Step 4: Constructing an optimal solution

- • S[i,j] records the values of k such that the optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between A_k and A_{k+1}.

- PRINT-OPTIMAL-PARENS(s, i, j)
- **if** $i = j$
- **then** print "A"$_i$
- **else** print "("
- PRINT-OPTIMAL-PARENS(s, i, $s[i, j]$)
- PRINT-OPTIMAL-PARENS(s, $s[i, j] + 1$, j)
- print ")"

Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- Running time:$O(n^3)$

MEMOIZED-MATRIX-CHAIN(p)

- 1 $n \leftarrow length[p] - 1$
- 2 **for** $i \leftarrow 1$ **to** n
- 3 **do for** $j \leftarrow i$ **to** n
- 4 **do** $m[i, j] \leftarrow \infty$
- 5 **return** LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)

- 1 **if** $m[i, j] < \infty$
- 2 **then return** $m[i, j]$
- 3 **if** $i = j$
- 4 **then** $m[i, j] \leftarrow 0$
- 5 **else for** $k \leftarrow i$ **to** $j - 1$
- 6 **do** $q \leftarrow$ LOOKUP-CHAIN(p, i, k)+ LOOKUP-CHAIN(p,$k + 1$, j) + $pi_{-1}\, p_k\, p_j$
- 7 **if** $q < m[i, j]$
- 8 **then** $m[i, j] \leftarrow q$
- 9 **return** $m[i, j]$

Example 4

Knapsack problem

Knapsack problem

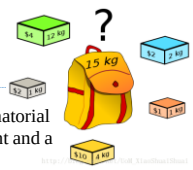
- There are two versions of the problem:
- (1) "0-1 knapsack problem" and
- (2) "Fractional knapsack problem"

- (1) Items are indivisible; you either take an item
- or not. Solved with *dynamic programming*
- (2) Items are divisible: you can take any fraction
- of an item. Solved with a *greedy algorithm*.

0-1 Knapsack problem

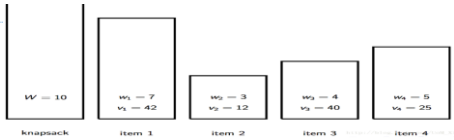
- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some **weight w_i** and benefit **value b_i** (all w_i , b_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

The knapsack problem



The knapsack problem is a problem in combinatorial optimisation: given n items, each with a weight and a value.
Find the most valuable selection of items that will fit in the knapsack.
As shown in the picture, knapsack capacity: 15 kg
weights: 1, 12, 2, 1, 4 kg values: 2, 4, 2, 1, 10 dollarsIf we choose the yellow, grey, blue and orange items, it will create the value of 2+2+1+10=15 dollars and has a weight of 8kg less than knapsack capacity.
This is the most valuable selection.
How can we find the most valuable selection?

方法一 暴力破解(Brute force)



Here is an example. We find all combinations of items and then compute its sum of weights and values.

Set	Weight	Value
∅	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	54
{1, 3}	11	NF
{1, 4}	12	NF

Set	Weight	Value
{2, 3}	7	52
{2, 4}	8	37
{3, 4}	9	65
{1, 2, 3}	14	NF
{1, 2, 4}	15	NF
{1, 3, 4}	16	NF
{2, 3, 4}	12	NF
{1, 2, 3, 4}	19	NF

方法二 动态规划算法(Dynamic Programming Algorithm)

- Dynamic programming is an algorithm design technique that is sometimes suitable when we want to solve a recurrence relation and the recursion involves overlapping instances.
- What is the recurrence relation in the knapsack problem?
- Let K(i, w) be the value of the best choice of items amongst first i items fitting in knapsack capacity w.
- Amongst first i items we either pick item i or we don't.
- For a choice which includes item i.
- We need find the optimal choice of first i-1 items to fit in w-wi. Hence, the value is K(i-1, w-wi) + wi.
- Notes the prerequisite that wi ≤ w.
- For a choice which excludes item i.
- This choice is the same with the choice that chooses optimal solution amongst i-1 items to fit in w.
- The value is K(i-1, w).

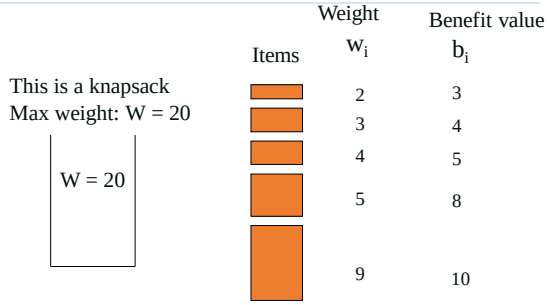
方法二 动态规划算法(Dynamic Programming Algorithm)

- (Pseudocode) function Knapsack()

```
function Knapsack()
    for i ← 0 to n do
        K[i][0] ← 0
    for j ← 1 to W do
        K[0][j] ← 0
    for i ← 1 to n do
        for j ← 1 to W do
            if j < wi then
                K[i][j] ← K[i-1][j]
            else
                K[i][j] ← max(K[i-1][j], K[i-1][j-wi]+vi)
    return K[n][W]
```

- The time complexity is **Θ(nW)**.
  Cause the basic operation is comparison "j < wi". It was executed ***n*W*** times.

a picture



0-1 Knapsack problem

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \le W$$

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

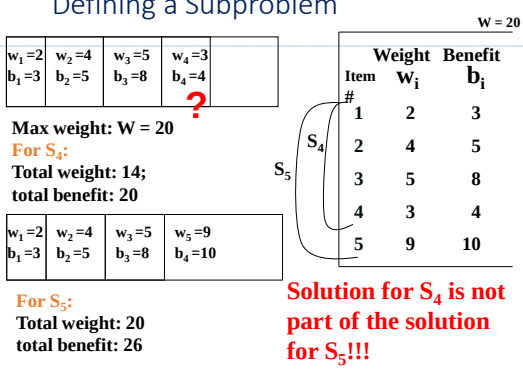
- Since there are *n* items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to *W*
- Running time will be $O(2^n)$

- Can we do better?
- dynamic programming
- We need to carefully identify the subproblems

Step1: Defining a Subproblem

- S_n = *{items labeled 1, 2, ..k ,..n}*
- S_k = *{items labeled 1, 2, .. k}*
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?

Defining a Subproblem



Defining a Subproblem

- Let's add another parameter: ***w***, which will represent the exact weight for each subset of items
- The subproblem then will be to compute ***B[k,w]***

B[k,w] :The maximum value obtained by filling a knapsack of size w with items taken from $\{u_1, u_2, \ldots, u_k\}$ only in an optimal way.

- B[k,w-w_k]+b_k: The maximum value obtained by filling a knapsack of sizew-w_k with items taken from $\{u_1, u_2, \ldots, u_k\}$ in on optimal way plus the value of item u_i. This case applies only if w≥w_k and it amounts to adding item u_k to the knapsack.

Step2: Recursive Formula for subproblems

- Recursive formula for subproblems:

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

- The best subset of S_k that has the total weight *w*, either contains item *k* or not.
- **First case: w_k>*w*.** Item *k* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable
- **Second case: w_k <=*w*.** Then the item *k* can be in the solution, and we choose the case with greater value

Step3: Computing the optimal costs

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
      if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                  B[i,w] = b_i + B[i-1,w- w_i]
            else
                  B[i,w] = B[i-1,w]
      else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W                          O(W)
  B[0,w] = 0
for i = 0 to n                          Repeat n times
  B[i,0] = 0
  for w = 0 to W                        O(W)
      < the rest of the code >
```

What is the running time of this algorithm? O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

```
for w = 0 to W
      B[0,w] = 0
```

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

```
for i = 0 to n
      B[i,0] = 0
```

Example (4)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0				
3	0				
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=1
$w-w_i$ =-1

```
if w_i <= w // item i can be part of the solution
   if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_i + B[i-1,w- w_i]
   else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (5)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0				
4	0				
5	0				

i=1
b_i=3
w_i=2
w=2
w-w_i =0

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0				
5	0				

i=1
b_i=3
w_i=2
w=3
w-w_i=1

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (7)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0				

i=1
b_i=3
w_i=2
w=4
w-w_i=2

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (8)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=1
b_i=3
w_i=2
w=5
w-w_i=2

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (9)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=1
w-w_i=-2

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (10)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	**3**		
3	0	3			
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=2
w-w_i=-1

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (11)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=3
w-w_i=0

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (12)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

i=2
b_i=4
w_i=3
w=4
w-w_i=1

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (13)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

i=2
b_i=4
w_i=3
w=5
w-w_i=2

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0** → **0**		
2	0	3	3 → **3**		
3	0	3	4 → **4**		
4	0	3	4		
5	0	3	7		

i=3
b_i=5
w_i=4
w=1..3

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

i=3
b_i=5
w_i=4
w=4
w- w_i=0

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ **7**	

i=3
b_i=5
w_i=4
w=5
w- w_i=1

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (16)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

i=3
b_i=5
w_i=4
w=1..4

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (17)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	**7**

i=3
b_i=5
w_i=4
w=5

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Step4: Constructing an optimal solution

- How to do?
- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary

An example

- Capacity: 9; 4 items sizes: 2, 3, 4, 5; values: 3, 4, 5, 7.

The maximum value is 12 and there are two optimal solutions: Consider put item i into the knapsack
1,2,3 and 3,4.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7
3	0	0	3	4	5	7	8	9	9	12
4	0	0	3	4	5	7	8	10	11	12

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems.
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **O(m*n)** vs. **O(n * 2^m)**
  - 0-1 Knapsack problem: **O(W*n)** vs. **O(2^n)**

Summary

- Difference with Divide and conquer
  - Repeatedly solving the sub-problems
- Applied fields
  - Optimization problems
- The four steps
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom up fashion
  - Construct an optimal solution from computed information

Dynamic programming

- DP is a method for solving certain kind of problems
- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

Optimal substructure

- Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming might apply.

Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common