Review: Dynamic Programming

- **Dynamic Programming** is a technique for algorithm design. It is a **tabular method** in which we break down the problem into subproblems, and place the solution to the subproblems in a matrix. The matrix elements can be computed:

  - iteratively, in a bottom-up fashion;

  - recursively, using *memoization*.

- **Dynamic Programming** is often used to solve optimization problems. In these cases, the solution corresponds to an objective function whose value needs to be optimal (e.g. maximal or minimal). Usually it is sufficient to produce one optimal solution, even though there may be many optimal solutions for a given problem.

Developing Dynamic Programming Algorithms

Four steps:

(i) Characterize the structure of an optimal solution.

(ii) Recursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution in a bottom up fashion.

(iv) Construct an optimal solution from the computed information.

Matrix Chain Multiplication

Basics

Let **A** be a $p \times q$ matrix and let **B** be a $q \times r$ matrix. Then we can multiply $A_{p \times q} * B_{q \times r} = C_{p \times r}$, where the elements of $C_{p \times r}$ are defined as:

$$c_{ij} \; = \; \sum_{k=1}^{q} a_{ik} b_{kj}$$

The straightforward algorithm to compute $A_{p \times q} * B_{q \times r}$ takes $p * q * r$ multiplications.

Chains of Matrices

Consider $A_1 \cdot A_2 \cdot ... \cdot A_n$. We can compute this product if the number of columns in A_i is equal to the number of rows in A_{i+1} ($\mathsf{Cols}[A_i] = \mathsf{Rows}[A_{i+1}]$) for every $1 \leq i \leq n - 1$.

How to order the multiplications?

Notice that matrix multiplication is **associative**, i.e.
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$. This results in many possible
paranthesizations (i.e. orderings of matrix multiplications).

A product of matrices is **fully parenthesized** if it is either a
single matrix or the product of two fully parenthesized matrix
products.
For example, $A_1 \cdot A_2 \cdot A_3 \cdot A_4$, can be fully parenthesized as:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$
$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$
$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$
$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$

The parenthesization is important

Different parenthesizations of $A_1 \cdot ... \cdot A_n$ may lead to different total numbers of scalar multiplications.

For example, let A_1 be a 10×100 matrix, A_2 be a 100×5 matrix, and A_3 be a 5×50 matrix. Then

$$
\begin{aligned}
cost[(A_1 \cdot A_2) \cdot A_3] &= (10 * 100 * 5) + (10 * 5 * 50) \\
&= 7,500 \text{ multiplications}
\end{aligned}
$$

And

$$
\begin{aligned}
cost[A_1 \cdot (A_2 \cdot A_3)] &= (100 * 5 * 50) + (10 * 100 * 50) \\
&= 75,000 \text{ multiplications}
\end{aligned}
$$

There are $\Omega(2^n)$ possible parenthesizations

We can think of a parenthesization as a binary parse tree.

Examples:

- left-branching: $((...((A_1 \cdot A_2) \cdot A_3)...) \cdot A_n)$

- right-branching: $(A_1 \cdot (...(A_{n-2} \cdot (A_{n-1} \cdot A_n))...))$

Total number of paranthesizations is:

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

$P(n)$ is the sequence of **Catalan numbers**, which grows as $\Omega(2^n)$ (*Exercise 15.2-3*) $\Rightarrow$ Brute force is ruled out!

The Matrix Chain Multiplication Problem

Input:

A chain of n matrices $\langle A_1, A_2, ..., A_n \rangle$ such that
$\text{Cols}[A_i] = \text{Rows}[A_{i+1}]$, for $i = 1, 2, ..., n$

Output:

An optimal, fully parenthesized product $A_1 \cdot A_2 \cdot ... \cdot A_n$ (i.e., in a way that minimizes the total number of scalar multiplications).

A Dynamic Programming Solution: Step (i)

Step (i) *characterize the structure of an optimal solution.*

Assume that the optimal way to multiply $A_1 A_2 ...... A_n$ is $(A_1 A_2 ... A_k)(A_{k+1} ... A_n)$, for some k.

Its associated cost is:

$$cost(A_1 ... A_n) = cost(A_1 ... A_k) + cost(A_{k+1} ... A_n) + \\ rows[A_1] \cdot col[A_k] \cdot col[A_n] \qquad (1)$$

If $cost(A_1 ... A_n)$ is minimal, then both both $cost(A_1 ... A_k)$ and $cost(A_{k+1} ... A_n)$ are minimal. Why?

A Dynamic Programming Solution: Step (ii)

Step (ii) *Recursively define the value of an optimal solution.*

Consider the general case of multiplying $A_i, ... A_j$.

Use an array p[] to record the dimensions of matrices. Thus, each matrix A_i is $p_{i-1} \times p_i$, where $p_{i-1} = rows[A_i]$, $p_i = col[A_i]$.

Define $m[i, j]$ to be the minimum cost of multiplying $A_i, ... A_j$.

Then:

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

Compute m[i,j]: A Simple Recursive Approach

Directly compute m[i, j] based on the recursive solution:

$\textsc{Recursive-Matrix-Chain}\ (p, i, j)$

  if $i = j$ return 0;

  $\text{m}[i, j] := \infty;$

   for $k := i$ to $j - 1$

     $q := \text{RECURSIVE-MATRIX-CHAIN}\ (p, i, k)$

      $+\ \text{RECURSIVE-MATRIX-CHAIN}\ (p, k + 1, j)$

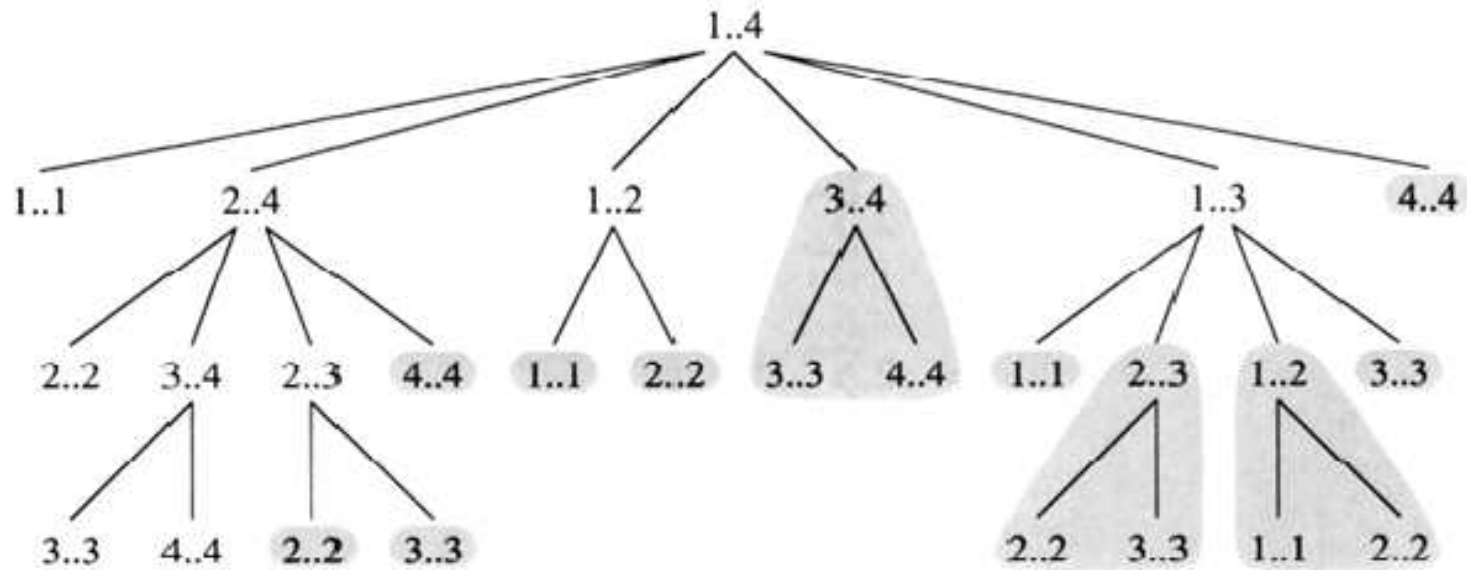 $+\ p_{i-1} \cdot p_k \cdot p_j$

    if $q < m[i, j]$

      $m[i, j] := q;$

  return $m[i, j]$
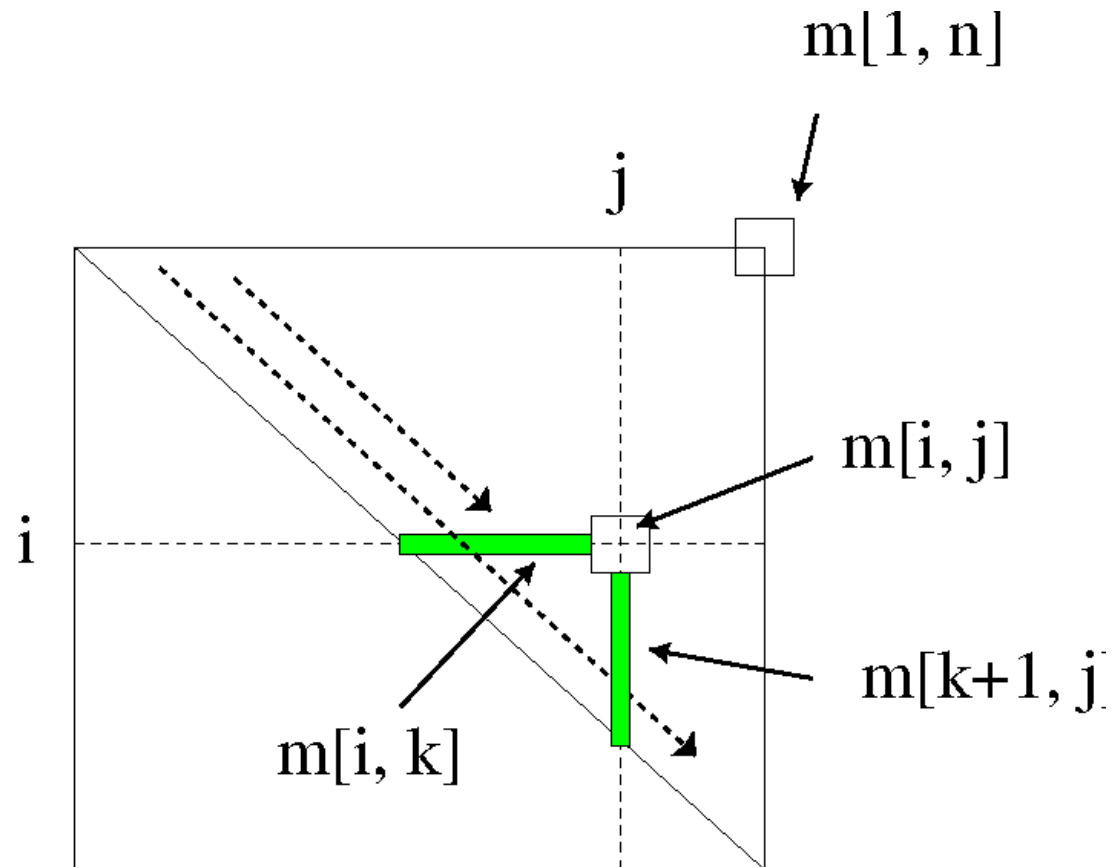
Recursive-Matrix-Chain(p, 1, 4)

Time Analysis

The following recurrence relation describes the running time of RECURSIVE-MATRIX-CHAIN():

$$
\begin{aligned}
T(1) &\geq 1 \\
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\
&= 2 \sum_{i=1}^{n-1} T(i) + n
\end{aligned}
$$

Exercise

Prove that $T(n) \geq 2^{n-1}$

Compute m[i,j]: A Bottom-Up Iterative Approach

$m[1, n]$

j

$m[i, j]$

i

$m[k+1, j]$

$m[i, k]$

We can compute $m[1..n]$ in $O(n^3)$ steps.

Bottom-Up Iterative Example

Example:

$$
\begin{array}{ll}
A_1 & \text{30x35} \\
A_2 & \text{35x15} \\
A_3 & \text{15x5} \\
A_4 & \text{5x10} \\
A_5 & \text{10x20} \\
A_6 & \text{20x25}
\end{array}
$$

Let's fill in part of the m array for this example.

Bottom-Up Iterative Example

j

i	1	2	3	4	5	6
1	0	15750	7815	9375	11875	15125
2		0	2625	4375	3125	10510
3			0	750	2500	5375
4				9	1000	3500
5					0	5000
6						0

Working it out

$$m[2,6] = \min_{2 \le k \le 6} \{m[2,k] + m[k+1,6] + p_1 \cdot p_k \cdot p_6\}$$

$$= \min \begin{cases} m[2,2] + m[3,6] + 35 \times 15 \times 25 & = 18,500 \\ m[2,3] + m[4,6] + 35 \times 5 \times 25 & = 10,510 \\ m[2,4] + m[5,6] + 35 \times 10 \times 25 & = 18,125 \\ m[2,5] + m[6,6] + 35 \times 20 \times 25 & = 24,625 \end{cases}$$

Thus, $S[2,6] = k = 3$.

Iterative Algorithm

Step (iii): *Compute values in a bottom-up fashion.*

MATRIX-CHAIN-ORDER(p)

    for $i := 1$ to n

        $m[i, i] := 0;$

    for $l := 2$ to n   // l is the moving line

        for $i := 1$ to $n - l + 1$

            $j := i + l - 1;$

            $m[i, j] := \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j]$

$$+ p_{i-1} * p_k * p_j \}$$

Complexity is $\Theta(n^3)$

MATRIX-CHAIN-ORDER(p)

    for $i := 1$ to n $\Theta(n)$

        $m[i,i] := 0;$

    for $l := 2$ to n $O(n)$

        for $i := 1$ to $n - l + 1$ $O(n)$

            $j := i + l - 1;$

            $m[i,j] := \min_{i \le k < j} \{m[i,k] + m[k+1,j]$ $O(n)$

$$+ p_{i-1} * p_k * p_j\}$$

Overal: $O(n^3)$.   It's also $\Omega(n^3)$ (homework).

Step (iv): Constructing an optimal solution

Use another table $s[1..n, 1..n]$. Each entry $s[i,j]$ records the value of k such that the optimal parenthesization of $A_i A_{i+1}...A_j$ splits the product between A_k and A_{k+1}.

MATRIX-CHAIN-ORDER(p)

$$
\begin{aligned}
&\text{for } l := 2 \text{ to n} \\
&\qquad \text{for } i := 1 \text{ to } n - l + 1 \\
&\qquad\qquad j := i + l - 1; \\
&\qquad\qquad m[i,j] := \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j \} \\
&\qquad\qquad s[i,j] := \operatorname*{argmin}_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j \}
\end{aligned}
$$

Step (iv): Constructing an optimal solution

How do we find the actual optimal parenthesization?
Notice that $s[1, n]$ is the position of the outmost multiplication.

$$(A_1...A_{s[1,n]})(A_{s[1,n]+1})....A_n)$$

Similarly, the outermost multiplication for the left-hand side is at the position $s[1, s[1, n]]$, and the outermost multiplication for the right-hand side is at the position $s[s[1, n] + 1, n]$.

We can generalize this to give a simple recursive algorithm to print the minimal parenthesization.

Printing optimal parenthesization

PRINT-OPTIMAL-PARENS(s, i, j)

    if i = j

            print $''A''_i$;

    else

            print "(";

                PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$

                PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$

            print ")";

S[1..n, 1..n]

j

	1	2	3	4	5	6
i 1		1	1	3	3	3
2			2	3	3	3
				3	3	3
4					4	5
5						5
6						

PRINT-OPTIMAL-PARENS (s, 1, 6):

$((A_1 (A_2 A_3))) ((A_4 A_5) A_6)$

The Recursive Approach Revisited

As we've seen already, we can solve the matrix chain multiplication problem recursively using the following algorithm:

$\text{RECURSIVE-MATRIX-CHAIN}\ (p, i, j)$
    if $i = j$ return 0;
    $m[i, j] := \infty$;

    for $k := i$ to $j - 1$
           $q := \text{RECURSIVE-MATRIX-CHAIN}\ (p, i, k)$
               $+ \text{RECURSIVE-MATRIX-CHAIN}\ (p, k + 1, j)$
               $+ p_{i-1} \cdot p_k \cdot p_j$
        if $q < m[i, j]$
            $m[i, j] := q$;

    return $m[i, j]$

Time Analysis Revisited

The following recurrence relation describes the running time of RECURSIVE-MATRIX-CHAIN():

$$
\begin{aligned}
T(1) &\geq 1 \\
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\
&= 2 \sum_{i=1}^{n-1} T(i) + n
\end{aligned}
$$

Exercise

Prove that $T(n) \geq 2^{n-1}$

We can speed up the recursive version by using the **Memoization** trick.

Memoization (pages 387–389)

We can make the recursive version more efficient than the straightforward approach that we gave earlier.

Basic Idea: Since the recursive version recomputes many of it's values, we can simply "remember" the values that we've already computed, and not recompute them.

Notice that this requires that we save some extra information to tell us that we have already computed some value.

We can create a memoized version for the matrix chain problem.

Memoized Matrix Chain

Memoized-Matrix-Chain (p) {
    Initialization();
    Lookup-Chain$(p, 1, n)$;
}

First, we set all of the values $m[i, j] = \infty$ to indicate that they have not been computed, yet.

Initialization() {
   for $i = 1$ to n do
     for $j = i$ to n do
       $m[i, j] = \infty$;
}

Memoized Matrix Chain: Recursion

Lookup-Chain (p, i, j) {
   if $m[i, j] < \infty$
       return $m[i, j]$;
   if $(i = j)$
       $m[i, j] := 0$;
   else
       for $k := i$ to $j - 1$
           $q :=$ Lookup-Chain (p, i, k) + Lookup-Chain $(p, k + 1, j)$
               $+ \; p_{i-1} \cdot p_k \cdot p_j$;
         if $q < m[i, j]$
           $m[i, j] := q$;
   return $m[i, j]$
}

Advantages?

Memoization gives an algorithm that is roughly as fast as the iterative version – in practice it is slower by a constant factor, due to the recursion overhead and table maintenance.

Think of how to analyze this algorithm – notice that the standard recurrence relation analysis cannot be used.

Advantages of memoization:

- It is easier to code than the iterative version, thus it is less error prone.

- It solves only those subproblems that are definitely required (useful when not all subproblems in the subproblem space need to be solved).

Dynamic Programming: Key Ingredients

Two ingredients of optimization problems that lead to a dynamic programming solution:

- **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to sub-problems.

- **Overlapping sub-problems:** same subproblem will be visited again and again (i.e. subproblems share subsubproblems).

An example where optimal sub-structure doesn't hold

Longest Simple Path problem:
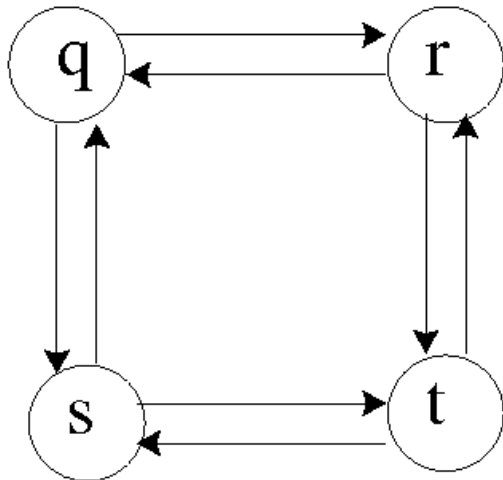
A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure.

The path $q \to r \to t$ is a longest simple path from q to r, but the subpath $r \to t$ is not a longest simple path from r to t.

Design and Analysis of Algorithms: Lecture 17

Overlapping sub-problems

Recursive-Matrix-Chain(p, 1, 4)
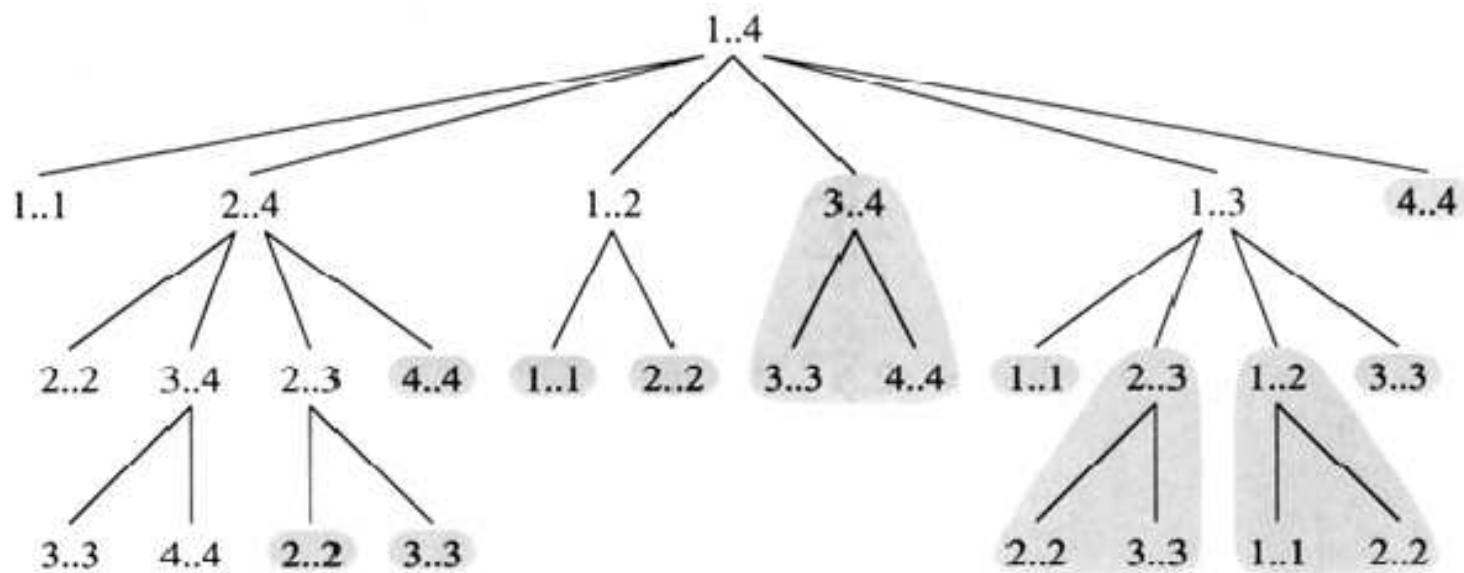
Overlapping sub-problems

Fibonacci numbers:

$$F(100) = F(99) + F(98)$$
$$= (F(98) + F(97)) + (F(97) + F(96))$$
$$= \dots$$

Divide & Conquer vs. Dynamic Programming

- Divide & Conquer is indicated when the sub-problems are independent.

- Dynamic Programming is indicated when the sub-problems share common sub-sub-problems.

Greedy Method vs. Dynamic Programming

• Both require the solution of the optimization problem to have *optimal substructure*.

• In Dynamic Programming, the optimal solution of a problem depends on the optimal solution of its sub-problems, so the computation is carried out in a *bottom-up* manner.

• In the Greedy method, a decision is made at each step before solving the subproblem, so a greedy algorithm usually runs in a *top-down* fashion.

• Greedy method uses only local information to make decision. We need to prove that locally optimal decisions lead to a globally optimal solution, and this is where cleverness may be required.