Combinatorial problems

Jordi Cortadella
Department of Computer Science
Cycles in permutations

- Let \( P \) be a vector of \( n \) elements containing a permutation of the numbers 0...\( n-1 \).
- The permutation contains cycles and all elements are in some cycle.

\[
\begin{array}{cccccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  P[i] & 6 & 4 & 2 & 8 & 0 & 7 & 9 & 3 & 5 & 1 \\
\end{array}
\]

Cycles:
- \((0 6 9 1 4)\)
- \((2)\)
- \((3 8 5 7)\)

Design a program that writes all cycles of a permutation.
Cycles in permutations

- Use an auxiliary vector (visited) to indicate the elements already written.
- After writing one permutation, the index returns to the first element.
- After writing one permutation, find the next non-visited element.
Cycles in permutations

// Pre: P is a vector with a permutation of 0..n-1
// Prints the cycles of the permutation

void print_cycles(const vector<int>& P) {
    int n = P.size();
    vector<bool> visited(n, false);

    int i = 0;
    while (i < n) {

        // All the cycles containing 0..i-1 have been written
        bool cycle = false;
        while (not visited[i]) {
            if (not cycle) cout << '(';
            else cout << ' ';// Not the first element
            cout << i;
            cycle = true;
            visited[i] = true;
            i = P[i];
        }
        if (cycle) cout << ')' << endl;

        // We have returned to the beginning of the cycle
        i = i + 1;
    }
}
Permutations

- Given a number N, generate all permutations of the numbers 1...N in lexicographical order.

For N=4:

1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
// Structure to represent the prefix of a permutation.  
// When all the elements are used, the permutation is  
// complete.  
// **Note:** used[i] represents the element i+1

```cpp
struct Permut {
    vector<int> v; // stores a partial permutation (prefix)
    vector<bool> used; // elements used in v
};
```

<table>
<thead>
<tr>
<th>v:</th>
<th>3</th>
<th>1</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>used:</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
void BuildPermutation(Permut& P, int i);

// Pre: P.v[0..i-1] contains a prefix of the permutation.
// P.used indicates the elements present in P.v[0..i-1]
// Prints all the permutations with prefix P.v[0..i-1] in
// lexicographical order.
```cpp
void BuildPermutation(Permut& P, int i) {
    if (i == P.v.size()) {
        PrintPermutation(P); // permutation completed
    } else {
        // Define one more location for the prefix
        // preserving the lexicographical order of
        // the unused elements
        for (int k = 0; k < P.used.size(); ++k) {
            if (not P.used[k]) {
                P.v[i] = k + 1;
                P.used[k] = true;
                BuildPermutation(P, i + 1);
                P.used[k] = false;
            }
        }
    }
}
```
```cpp
int main() {
    int n;
    cin >> n; // will generate permutations of {1..n}
    Permut P; // creates a permutation with empty prefix
    P.v = vector<int>(n);
    P.used = vector<bool>(n, false);
    BuildPermutation(P, 0);
}

void PrintPermutation(const Permut& P) {
    int last = P.v.size() - 1;
    for (int i = 0; i < last; ++i) cout << P.v[i] << " ";
    cout << P.v[last] << endl;
}
```
Sub-sequences summing $n$

• Given a sequence of positive numbers, write all the sub-sequences that sum another given number $n$.

• The input will first indicate the number of elements in the sequence and the target sum. Next, all the elements in the sequence will follow, e.g.

6

\[
\begin{array}{cccccc}
7 & 12 \\
\text{number of elements} & \text{target sum} \\
\end{array}
\]

sequence

\[
\begin{array}{ccccccc}
3 & 6 & 1 & 4 & 6 & 5 & 2 \\
\end{array}
\]
Sub-sequences summing $n$

> 7 12 3 6 1 4 6 5 2
3 6 1 2
3 1 6 2
3 4 5
6 1 5
6 4 2
6 6
1 4 5 2
1 6 5
4 6 2
Sub-sequences summing \( n \)

- How do we represent a subset of the elements of a vector?
  - A Boolean vector can be associated to indicate which elements belong to the subset.

<table>
<thead>
<tr>
<th>Value:</th>
<th>3</th>
<th>6</th>
<th>1</th>
<th>4</th>
<th>6</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chosen:</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

represents the subset \( \{6, 1, 5\} \)
Sub-sequences summing $n$

- How do we generate all subsets of the elements of a vector? Recursively.
  - Decide whether the first element must be present or not.
  - Generate all subsets with the rest of the elements
Sub-sequences summing $n$

• How do we generate all the subsets that sum $n$?

  – Pick the first element (3) and generate all the subsets that sum $n-3$ starting from the second element.

  – Do not pick the first element, and generate all the subsets that sum $n$ starting from the second element.
Sub-sequences summing $n$

```c++
struct Subset {
    vector<int> values;
    vector<bool> chosen;
};

void main() {
    // Read number of elements and sum
    int n, sum;
    cin >> n >> sum;

    // Read sequence
    Subset s;
    s.values = vector<int>(n);
    s.chosen = vector<bool>(n, false);
    for (int i = 0; i < n; ++i) cin >> s.values[i];

    // Generates all subsets from element 0
    generate_subsets(s, 0, sum);
}
```
void generate_subsets(Subset& s, int i, int sum);

// Pre: s.values is a vector of n positive values and
//      s.chosen[0..i-1] defines a partial subset.
//      s.chosen[i..n-1] is false.

// Prints the subsets that agree with
// s.chosen[0..i-1] such that the sum of the
// chosen values in s.values[i..n-1] is sum.

// Terminal cases:
//   · sum < 0 → nothing to print
//   · sum = 0 → print the subset
//   · i >= n → nothing to print
```c
void generate_subsets(Subset& s, int i, int sum) {
    if (sum >= 0) {
        if (sum == 0) print_subset(s);
        else if (i < s.values.size()) {
            // Recursive case: pick i and subtract from sum
            s.chosen[i] = true;
            generate_subsets(s, i + 1, sum - s.values[i]);
            // Do not pick i and maintain the sum
            s.chosen[i] = false;
            generate_subsets(s, i + 1, sum);
        }
    }
}
```
void print_subset(const Subset& s) {

    // Pre: s.values contains a set of values and
    // s.chosen indicates the values to be printed
    // Prints the chosen values

    for (int i = 0; i < s.values.size(); ++i) {
        if (s.chosen[i]) cout << s.values[i] << " ";
    }
    cout << endl;
}

Lattice paths

We have an $n \times m$ grid.

How many different routes are there from the bottom left corner to the upper right corner only using right and up moves?
Lattice paths

Some properties:

- \( \text{paths}(n, 0) = \text{paths}(0, m) = 1 \)
- \( \text{paths}(n, m) = \text{paths}(m, n) \)
- If \( n > 0 \) and \( m > 0 \):
  \[
  \text{paths}(n, m) = \text{paths}(n-1, m) + \text{paths}(n, m-1)
  \]
Lattice paths

// Pre: n and m are the dimensions of a grid
// (n \geq 0 and m \geq 0)
// Returns the number of lattice paths in the grid

int paths(int n, int m) {
    if (n == 0 || m == 0) return 1;
    return paths(n - 1, m) + paths(n, m - 1);
}
Lattice paths

• How large is the tree (cost of the computation)?
• Observation: many computations are repeated
### Lattice paths

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>20</td>
<td>35</td>
<td>56</td>
<td>84</td>
<td>120</td>
<td>165</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>35</td>
<td>70</td>
<td>126</td>
<td>210</td>
<td>330</td>
<td>495</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>6</td>
<td>21</td>
<td>56</td>
<td>126</td>
<td>252</td>
<td>462</td>
<td>792</td>
<td>1287</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>7</td>
<td>28</td>
<td>84</td>
<td>210</td>
<td>462</td>
<td>924</td>
<td>1716</td>
<td>3003</td>
</tr>
</tbody>
</table>

\[ M[i][0] = M[0][i] = 1 \]

\[ M[i][j] = M[i-1][j] + M[i][j-1], \quad \text{for } i > 0, j > 0 \]
Lattice paths

// Pre: n and m are the dimensions of a grid
//       (n ≥ 0 and m ≥ 0)
// Returns the number of lattice paths in the grid

int paths(int n, int m) {
    vector<vector<int>> M(n + 1, vector<int>(m + 1));
    // Initialize row 0
    for (int j = 0; j <= m; ++j) M[0][j] = 1;

    // Fill the matrix from row 1
    for (int i = 1; i <= n; ++i) {
        M[i][0] = 1;
        for (int j = 1; j <= m; ++j) {
            M[i][j] = M[i - 1][j] + M[i][j - 1];
        }
    }
    return M[n][m];
}
# Lattice paths

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(1,0)</td>
<td>(2,0)</td>
<td>(3,0)</td>
<td>(4,0)</td>
<td>(5,0)</td>
<td>(6,0)</td>
<td>(7,0)</td>
<td>(8,0)</td>
</tr>
<tr>
<td>1</td>
<td>(1,1)</td>
<td>(2,1)</td>
<td>(3,1)</td>
<td>(4,1)</td>
<td>(5,1)</td>
<td>(6,1)</td>
<td>(7,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
</tr>
<tr>
<td>2</td>
<td>(2,2)</td>
<td>(3,2)</td>
<td>(4,2)</td>
<td>(5,2)</td>
<td>(6,2)</td>
<td>(7,2)</td>
<td>(8,2)</td>
<td>(9,2)</td>
<td>(10,2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,3)</td>
<td>(4,3)</td>
<td>(5,3)</td>
<td>(6,3)</td>
<td>(7,3)</td>
<td>(8,3)</td>
<td>(9,3)</td>
<td>(10,3)</td>
<td>(11,3)</td>
</tr>
<tr>
<td>4</td>
<td>(4,4)</td>
<td>(5,4)</td>
<td>(6,4)</td>
<td>(7,4)</td>
<td>(8,4)</td>
<td>(9,4)</td>
<td>(10,4)</td>
<td>(11,4)</td>
<td>(12,4)</td>
</tr>
<tr>
<td>5</td>
<td>(5,5)</td>
<td>(6,5)</td>
<td>(7,5)</td>
<td>(8,5)</td>
<td>(9,5)</td>
<td>(10,5)</td>
<td>(11,5)</td>
<td>(12,5)</td>
<td>(13,5)</td>
</tr>
<tr>
<td>6</td>
<td>(6,6)</td>
<td>(7,6)</td>
<td>(8,6)</td>
<td>(9,6)</td>
<td>(10,6)</td>
<td>(11,6)</td>
<td>(12,6)</td>
<td>(13,6)</td>
<td>(14,6)</td>
</tr>
</tbody>
</table>

\[
M[i][j] = \binom{i + j}{i} = \binom{i + j}{j}
\]
• In a path with $n+m$ segments, select $n$ segments to move right (or $m$ segments to move up)

• Subsets of $n$ elements out of $n+m$
// Pre: n and m are the dimensions of a grid
//      (n ≥ 0 and m ≥ 0)
// Returns the number of lattice paths in the grid

int paths(int n, int m) {
    return combinations(n + m, n);
}

// Pre: n ≥ k ≥ 0
// Returns \binom{n}{k}

int combinations(int n, int k) {
    if (k == 0) return 1;
    return n*combinations(n - 1, k - 1)/k;
}
Lattice paths

Computational cost:

- Recursive version: \( O\left(\binom{n+m}{m}\right) \)

- Matrix version: \( O(n \cdot m) \)

- Combinations: \( O(m) \)
Lattice paths

• How about counting paths in a 3D grid?
• And in a k-D grid?
Summary

• Combinatorial problems involve a finite set of objects and a finite set of solutions.

• Different goals:
  – Enumeration of all solutions
  – Finding one solution (satisfiability)
  – Finding the best solution (optimization)

• This lecture has only covered enumeration problems.

• Recommendations:
  – Use inductive reasoning (recursion) whenever possible.
  – Reuse calculations.