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.

<table>
<thead>
<tr>
<th>$i$</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[i]$</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Design a program that writes all cycles of a permutation.

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

```cpp
void print_cycles(const vector<int>& P) {
    int n = P.size();
    vector<bool> visited(n, false);
    int i = 0;
    while (i < n) {
        bool cycle = false;
        while (!visited[i]) {
            if (!cycle) cout << '('; // Not the first element
            else cout << '>'; // Not the first element
            cycle = true;
            visited[i] = true;
            i = P[i];
        }
        if (cycle) cout << ')' << endl;
        // We have returned to the beginning of the cycle
        i = i + 1;
    }
}
```

- 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.
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

Permutations

```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;
            }
        }
    }
}
```

// 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
};
```
# Permutations

```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);
}
```

```cpp
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.

```
7  12  3  6  1  4  6  5  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.

```
Value:  3  6  1  4  6  5  2
      false  true  true  false  false  true  false
Chosen: false  true  true  false  false  true  false
```

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

```
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);
}
```

- 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.
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×m grid.
How many different routes are there from the bottom left corner to the upper right corner only using right and up moves?

Some properties:
- paths(n, 0) = paths(0, m) = 1
- paths(n, m) = paths(m, n)
- If n > 0 and m > 0: paths(n, m) = paths(n-1, m) + paths(n, m-1)
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) {
    if (n == 0 || m == 0) return 1;
    return paths(n - 1, m) + paths(n, m - 1);
}

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

// 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];
}

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

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

\[
M[i][j] = \binom{i+j}{i} = \binom{i+j}{j}
\]

• 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;
}
```

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.