Abbreviation in C++	

``
#include <iostream>
#include <string>
using namespace std;
 	
class Abbreviation {
public:
    static void solution(string str, string asf, int count, int pos) {
        if (pos == str.length()) {
            if (count == 0) {
                cout << asf << endl;
            } else {
                cout << asf << count << endl;
            }
            return;
        }
 	
        if (count > 0) {
            solution(str, asf + to_string(count) + str[pos], 0, pos + 1);
        } else {
            solution(str, asf + str[pos], 0, pos + 1);
        }
 	
        solution(str, asf, count + 1, pos + 1);
    }
};
 	
int main() {
    string str = "pep";
    Abbreviation::solution(str, "", 0, 0);
    return 0;
}
``
	see below

```cpp
#include <iostream>
#include <string>
using namespace std;

class Abbreviation {
public:
    static void solution(string str, string asf, int count,
int pos) {
        if (pos == str.length()) {
            if (count == 0) {
                cout << asf << endl;
            } else {
                cout << asf << count << endl;
            }
            return;
        }

        if (count > 0) {
            solution(str, asf + to_string(count) + str[pos],
0, pos + 1);
        } else {
            solution(str, asf + str[pos], 0, pos + 1);
        }

        solution(str, asf, count + 1, pos + 1);
    }
};

int main() {
    string str = "pep";
    Abbreviation::solution(str, "", 0, 0);
    return 0;
}
```

Step-by-Step Execution:

The function solution() uses recursion to generate all the abbreviations. It has two main actions:

1. **Include the current character with an abbreviation count if any.**
2. **Include the current character as is.**

- str: The string for which we need to find abbreviations.
- asf: The abbreviation formed so far.
- count: The number of characters skipped (abbreviated).
- pos: The current position in the string.

1. **Initial Call**: solution("pep", "", 0, 0)
   - **pos = 0**, **count = 0**, **asf = ""**
   - Two options:
     1. Skip character at pos = 0 (first 'p')
     2. Include character at pos = 0 (first 'p')
   - Recur on both options.

2. **First Option**: Skip character at pos = 0
   - Call solution("pep", "", 1, 1) (increment count by 1)

3. **Call**: solution("pep", "", 1, 1)
   - **pos = 1**, **count = 1**, **asf = ""**
   - Two options:
     1. Skip character at pos = 1 (character 'e')
     2. Include character at pos = 1 (character 'e')
   - Recur on both options.

4. **First Option**: Skip character at pos = 1
   - Call solution("pep", "1", 2, 2) (skip 'e' and increment count)

5. **Call**: solution("pep", "1", 2, 2)
   - **pos = 2**, **count = 2**, **asf = "1"**
   - Two options:
     1. Skip character at pos = 2 (second 'p')
     2. Include character at pos = 2

	(second 'p') ○ Recur on both options. 6. **First Option**: Skip character at pos = 2 ○ Call solution("pep", "12", 3, 3) (skip 'p') 7. **Call**: solution("pep", "12", 3, 3) ○ **pos = 3**, **count = 3**, **asf = "12"** ○ Base case reached (pos == str.length()) ○ Output: "12" 8. **Second Option**: Include character at pos = 2 ○ Call solution("pep", "1p", 0, 3) (include 'p' and reset count) 9. **Call**: solution("pep", "1p", 0, 3) ○ **pos = 3**, **count = 0**, **asf = "1p"** ○ Base case reached (pos == str.length()) ○ Output: "1p" 10. **Backtrack to Call 4**: solution("pep", "", 1, 1) ○ **pos = 1**, **count = 1**, **asf = ""** ○ Second option: Include character at pos = 1 ('e') ○ Call solution("pep", "e", 0, 2) (reset count and include 'e') 11. **Call**: solution("pep", "e", 0, 2) ○ **pos = 2**, **count = 0**, **asf = "e"** ○ Two options: 1. Skip character at pos = 2 (second 'p') 2. Include character at pos = 2 (second 'p') ○ Recur on both options. 12. **First Option**: Skip character at pos = 2 ○ Call solution("pep", "e1", 1, 3) (skip

	'p' and increment count) 13. **Call**: solution("pep", "e1", 1, 3) ○ **pos = 3**, **count = 1**, **asf = "e1"** ○ Base case reached (pos == str.length()) ○ Output: "e1" 14. **Second Option**: Include character at pos = 2 ○ Call solution("pep", "ep", 0, 3) (include 'p' and reset count) 15. **Call**: solution("pep", "ep", 0, 3) ○ **pos = 3**, **count = 0**, **asf = "ep"** ○ Base case reached (pos == str.length()) ○ Output: "ep" 16. **Backtrack to Initial Call**: solution("pep", "", 0, 0) ○ **pos = 0**, **count = 0**, **asf = ""** ○ Second option: Include character at pos = 0 ('p') ○ Call solution("pep", "p", 0, 1) (include 'p' and reset count) 17. **Call**: solution("pep", "p", 0, 1) ○ **pos = 1**, **count = 0**, **asf = "p"** ○ Two options: 1. Skip character at pos = 1 (character 'e') 2. Include character at pos = 1 (character 'e') ○ Recur on both options. 18. **First Option**: Skip character at pos = 1 ○ Call solution("pep", "p1", 1, 2) (skip 'e' and increment count) 19. **Call**: solution("pep", "p1", 1, 2) ○ **pos = 2**, **count = 1**, **asf = "p1"** ○ Two options: 1. Skip character at pos = 2

	(second 'p') 2. Include character at pos = 2 (second 'p') o Recur on both options. 20. **First Option**: Skip character at pos = 2 o Call solution("pep", "p12", 2, 3) (skip 'p' and increment count) 21. **Call**: solution("pep", "p12", 2, 3) o **pos = 3**, **count = 2**, **asf = "p12"** o Base case reached (pos == str.length()) o Output: "p12" 22. **Second Option**: Include character at pos = 2 o Call solution("pep", "p1p", 0, 3) (include 'p' and reset count) 23. **Call**: solution("pep", "p1p", 0, 3) o **pos = 3**, **count = 0**, **asf = "p1p"** o Base case reached (pos == str.length()) o Output: "p1p" 24. **Backtrack to Call 17**: solution("pep", "p", 0, 1) o **pos = 1**, **count = 0**, **asf = "p"** o Second option: Include character at pos = 1 ('e') o Call solution("pep", "pe", 0, 2) (include 'e' and reset count) 25. **Call**: solution("pep", "pe", 0, 2) o **pos = 2**, **count = 0**, **asf = "pe"** o Two options: 1. Skip character at pos = 2 (second 'p') 2. Include character at pos = 2 (second 'p') o Recur on both options. 26. **First Option**: Skip character at pos = 2

	○ Call solution("pep", "pe1", 1, 3) (skip 'p' and increment count) 27. **Call**: solution("pep", "pe1", 1, 3) ○ **pos = 3**, **count = 1**, **asf = "pe1"** ○ Base case reached (pos == str.length()) ○ Output: "pe1" 28. **Second Option**: Include character at pos = 2 ○ Call solution("pep", "pep", 0, 3) (include 'p' and reset count) 29. **Call**: solution("pep", "pep", 0, 3) ○ **pos = 3**, **count = 0**, **asf = "pep"** ○ Base case reached (pos == str.length()) ○ Output: "pep"
Output:- pep pe1 p1p p2 1ep 1e1 2p 3	

All palindromic partition in C++	

<pre>#include <iostream>
#include <string>
using namespace std;
 	
class AllPalindromicPartition {
public:
    static void main() {
        string str = "abba";
        sol(str, "");
    }
 	
    static void sol(string str, string asf) {
        if (str.length() == 0) {
            cout << asf << endl;
            return;
        }
 	
        for (int i = 0; i < str.length(); i++) {
            string prefix = str.substr(0, i + 1);
            string ros = str.substr(i + 1);
            if (isPalin(prefix)) {
                sol(ros, asf + "(" + prefix + ")");
            }
        }
    }
 	
    static bool isPalin(string s) {
        int li = 0;
        int ri = s.length() - 1;
        while (li < ri) {
            if (s[li] != s[ri]) {
                return false;
            }
            li++;
            ri--;
        }
        return true;
    }
};
 	
int main() {
    AllPalindromicPartition::main();
    return 0;
}</pre>
	Dry Run for Input: "abba"
	
	1st Level of Recursion:
	• prefix = "a", ros = "bba", and "a" is a palindrome.
	• Call sol("bba", "(a)").
	
	2nd Level of Recursion:
	• prefix = "b", ros = "ba", and "b" is a palindrome.
	• Call sol("ba", "(a)(b)").
	
	3rd Level of Recursion:
	• prefix = "b", ros = "a", and "b" is a palindrome.
	• Call sol("a", "(a)(b)(b)").
	
	4th Level of Recursion:
	• prefix = "a", ros = "", and "a" is a palindrome.
	• Output (a)(b)(b)(a).
	
	Backtracking to Explore Other Partitions:
	
	3rd Level (Exploring Longer Prefixes):
	• prefix = "bb", ros = "a", and "bb" is a palindrome.
	• Call sol("a", "(a)(bb)").
	
	4th Level:
	• prefix = "a", ros = "", and "a" is a palindrome.
	• Output (a)(bb)(a).
	
	Backtracking to 2nd Level:
	• prefix = "bba" is not a palindrome. Skip.
	
	Backtracking to 1st Level:
	• prefix = "ab" and prefix = "abb" are not palindromes.
	• prefix = "abba", ros = "", and "abba" is a palindrome.

<table>
<tr><td></td><td>Output (abba).</td></tr>
<tr><td colspan="2">Output:-

(a)(b)(b)(a)
(a)(bb)(a)
(abba)</td></tr>
</table>

Combinations in C++

```
#include <iostream>
using namespace std;

void combinations(int cb, int nboxes, int ssf, int
ritems, string asf) {
    if (cb > nboxes) {
        if (ssf == ritems) {
            cout << asf << endl;
        }
        return;
    }
    combinations(cb + 1, nboxes, ssf + 1, ritems, asf +
"i");
    combinations(cb + 1, nboxes, ssf, ritems, asf + "-");
}

int main() {
    int nboxes = 3;
    int ritems = 2;
    combinations(1, nboxes, 0, ritems, "");
    return 0;
}
```

Dry Run for nboxes = 3 and ritems = 2

Initial call:
combinations(1, 3, 0, 2, "")

Recursive Tree:

1. Box 1: Place "i" → combinations(2, 3, 1, 2, "i")
    - Box 2: Place "i" → combinations(3, 3, 2, 2, "ii")
        - Box 3: Leave "-" → **Print: ii-**
    - Box 2: Leave "-" → combinations(3, 3, 1, 2, "i-")
        - Box 3: Place "i" → **Print: i-i**
2. Box 1: Leave "-" → combinations(2, 3, 0, 2, "-")
    - Box 2: Place "i" → combinations(3, 3, 1, 2, "-i")
        - Box 3: Place "i" → **Print: -ii**
    - Box 2: Leave "-" → combinations(3, 3, 0, 2, "--")
        - Box 3: Place "i" → **Not valid (ssf = 1)**

Output:-

```
ii-
i-i
-ii
```

Friend’s pairing in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int counter = 1;

void solution(int i, int n, vector<bool>& used, string asf) {
    if (i > n) {
        cout << counter << "." << asf << endl;
        counter++;
        return;
    }

    if (used[i]) {
        solution(i + 1, n, used, asf);
    } else {
        used[i] = true;
        solution(i + 1, n, used, asf + "(" + to_string(i) + ") ");
        for (int j = i + 1; j <= n; j++) {
            if (!used[j]) {
                used[j] = true;
                solution(i + 1, n, used, asf + "(" + to_string(i) + "," + to_string(j) + ") ");
                used[j] = false;
            }
        }
        used[i] = false;
    }
}

int main() {
    int n = 3;
    vector<bool> used(n + 1, false);
    solution(1, n, used, "");
    return 0;
}
```

Dry Run

Input: n = 3
Function Call: solution(1, 3, used = {false, false, false, false}, "")

1. i=1 (not used) → Mark 1 used → Call solution(2, 3, used={false, true, false, false}, "(1) ")
    - i=2 (not used) → Mark 2 used → Call solution(3, 3, used={false, true, true, false}, "(1) (2) ")
        - i=3 (not used) → Mark 3 used → Call solution(4, 3, used={false, true, true, true}, "(1) (2) (3) ")
            - Base case: Print 1.(1) (2) (3)
        - Backtrack: Unmark 3.
    - Backtrack: Unmark 2.
    - i=2 → Pair 2,3 → Mark 2,3 used → Call solution(4, 3, used={false, true, true, true}, "(1) (2,3) ")
        - Base case: Print 2.(1) (2,3)
    - Backtrack: Unmark 2,3.

- Backtrack: Unmark 1.

2. i=1 → Pair 1,2 → Mark 1,2 used → Call solution(3, 3, used={false, true, true, false}, "(1,2) ")
    - i=3 (not used) → Mark 3 used → Call solution(4, 3, used={false, true, true, true}, "(1,2) (3) ")
        - Base case: Print 3.(1,2) (3)
    - Backtrack: Unmark 3.

- Backtrack: Unmark 1,2.

3. i=1 → Pair 1,3 → Mark 1,3 used → Call solution(3, 3, used={false, true, false, true}, "(1,3) ")
    - i=2 (not used) → Mark 2 used → Call solution(4, 3, used={false, true, true, true}, "(1,3) (2) ")
        - Base case: Print 4.(1,3) (2)
    - Backtrack: Unmark 2.

- Backtrack: Unmark 1,3.

Output:-

```
1.(1) (2) (3)
2.(1) (2,3)
3.(1,2) (3)
4.(1,3) (2)
```

Goldmine2 in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int maxGold = 0;

void travel(vector<vector<int>>& arr, int i, int j,
vector<vector<bool>>& visited, vector<int>& bag) {
    if (i < 0 || j < 0 || i >= arr.size() || j >=
arr[0].size() || arr[i][j] == 0 || visited[i][j]) {
        return;
    }
    visited[i][j] = true;
    bag.push_back(arr[i][j]);
    travel(arr, i - 1, j, visited, bag);
    travel(arr, i, j + 1, visited, bag);
    travel(arr, i, j - 1, visited, bag);
    travel(arr, i + 1, j, visited, bag);
}

void getMaxGold(vector<vector<int>>& arr) {
    int rows = arr.size();
    int cols = arr[0].size();
    vector<vector<bool>> visited(rows,
vector<bool>(cols, false));

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (arr[i][j] != 0 && !visited[i][j]) {
                vector<int> bag;
                travel(arr, i, j, visited, bag);

                int sum = 0;
                for (int val : bag) {
                    sum += val;
                }
                if (sum > maxGold) {
                    maxGold = sum;
                }
            }
        }
    }
}

int main() {
    vector<vector<int>> arr = {
        {0, 1, 4, 2, 8, 2},
        {4, 3, 6, 5, 0, 4},
        {1, 2, 4, 1, 4, 6},
        {2, 0, 7, 3, 2, 2},
        {3, 1, 5, 9, 2, 4},
        {2, 7, 0, 8, 5, 1}
    };

    getMaxGold(arr);
    cout << maxGold << endl;

    return 0;
}
```

Step-by-Step Execution

Initial Setup:

- maxGold = 0
- visited initialized to false for all cells.
- Rows = 6, Cols = 6.

Outer Loop Iteration (i = 0, j = 0):

- Cell (0,0) is 0, skip it.

(i = 0, j = 1):

- Cell (0,1) is 1, not visited.
- Start travel function:
  - bag = [1]
  - Mark (0,1) as visited.
  - Explore neighboring cells:
    - (0,2): Add 4 to bag → bag = [1,4].
    - (1,2): Add 6 to bag → bag = [1,4,6].
    - Continue visiting valid cells → bag = [1,4,6,5,3,4,2].
  - Total sum of bag = 25.
- Update maxGold = 25.

(i = 0, j = 2, j = 3, ..., j = 5):

- All these cells are either visited or part of the same cluster.

Outer Loop Iteration (i = 1):

(i = 1, j = 0):

- Cell (1,0) is 4, not visited.
- Start travel function:
  - bag = [4]
  - Visit neighboring cells:
    - (2,0): Add 1 → bag = [4,1].
    - Continue visiting → bag = [4,1,2,3,7,9,5].
  - Total sum of bag = 31.
- Update maxGold = 31.

(i = 1, j = 1, ..., j = 5):

	- All cells are either visited or part of the same cluster. **Outer Loop Iteration (i = 2 to i = 5):** Continue similar logic for unvisited cells and new clusters. **Cluster at (4,3) and `(5,3):** - Explore the large gold cluster: - bag = [9,5,8,7,2]. - Total sum = **120**. - Update maxGold = 120. **Final Output:** **Maximum Gold Collected: 120**
Output:- 120	

Josephus in C++	

```#include <iostream>```
```using namespace std;```
 	
```int solution(int n, int k) {```
```   if (n == 1) {```
```   return 0;```
```   }```
```   int x = solution(n - 1, k);```
```   int y = (x + k) % n;```
```   return y;```
```}```
 	
```int main() {```
```   int n = 4;```
```   int k = 2;```
```   cout << solution(n, k) << endl;```
```   return 0;```
```}```
	Step-by-Step Execution:
	
	The function uses recursion to solve the Josephus problem. The base case is when n = 1, where the last remaining person is at position 0. The recursive case computes the position of the last person standing for n-1 people and then adjusts it by the step k using modulo operation.
	
	1. **Initial Call**: solution(4, 2)
	o **n = 4**, **k = 2**
	o Call solution(3, 2) (since n - 1 = 3)
	
	2. **Second Call**: solution(3, 2)
	o **n = 3**, **k = 2**
	o Call solution(2, 2) (since n - 1 = 2)
	
	3. **Third Call**: solution(2, 2)
	o **n = 2**, **k = 2**
	o Call solution(1, 2) (since n - 1 = 1)
	
	4. **Base Case**: solution(1, 2)
	o **n = 1**, **k = 2** (base case)
	o Return 0 (last remaining person at position 0)
	
	5. **Returning from Third Call**: solution(2, 2)
	o Result from solution(1, 2) is 0
	o Calculate y = (0 + 2) % 2 = 0
	o Return y = 0
	
	6. **Returning from Second Call**: solution(3, 2)
	o Result from solution(2, 2) is 0
	o Calculate y = (0 + 2) % 3 = 2
	o Return y = 2
	
	7. **Returning from First Call**: solution(4, 2)
	o Result from solution(3, 2) is 2
	o Calculate y = (2 + 2) % 4 = 0
	o Return y = 0

	Final Output: The position of the last remaining person (zero-indexed) is 0, so the output is 0.

Output:-
0

Largest after k swaps in C++

```cpp
#include <iostream>
using namespace std;

string max_str;

void findMaximum(string str, int k) {
    // Base case: When k swaps are used up
    if (k == 0) {
        return;
    }

    int n = str.length();

    // Find the maximum digit available for current
position
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // If digit at position j is greater than digit at
position i, swap them
            if (str[j] > str[i]) {
                swap(str[i], str[j]);

                // Check if current string is larger than
previously found max
                if (str > max_str) {
                    max_str = str;
                }

                // Recur for k-1 swaps on the modified string
                findMaximum(str, k - 1);

                // Backtrack: Swap again to revert to
original string
                swap(str[i], str[j]);
            }
        }
    }
}

int main() {
    string str = "1234567";
    int k = 4;

    // Initialize max_str with the original string
    max_str = str;

    // Find the maximum number possible after k
swaps
    findMaximum(str, k);

    // Print the maximum number found
    cout << max_str << endl;

    return 0;
}
```

Dry Run of the Code

Input:

- str = "1234567"
- k = 4

Step-by-Step Execution

Initialization

- max_str = "1234567"

First Level (k = 4)

- Outer loop: i = 0
  - Inner loop: j = 1
    - Swap: "2134567"
      - max_str = "2134567"
      - Recur with k = 3.

Second Level (k = 3)

- Outer loop: i = 0
  - Inner loop: j = 1
    - No swap (digits are the same).
  - Inner loop: j = 2
    - Swap: "3124567"
      - max_str = "3124567"
      - Recur with k = 2.

Third Level (k = 2)

- Outer loop: i = 0
  - Inner loop: j = 1, j = 2: No change (smaller results).
  - Inner loop: j = 3
    - Swap: "4123567"
      - max_str = "4123567"
      - Recur with k = 1.

Fourth Level (k = 1)

	- Outer loop: i = 0 - Inner loop: j = 4 - Swap: "5123467" - max_str = "5123467" - Recur with k = 0. **Base Case (k = 0)** - Stop recursion and backtrack. **Backtracking** - Undo each swap and check other combinations. - Repeat similar logic for other positions (i = 1, 2, ...). **Final max_str** The largest number found after 4 swaps is: **7654321**
Output:- 7654321	

Lexicographic order in C++

```
#include <iostream>
using namespace std;

void dfs(int i, int n) {
    if (i > n) {
        return;
    }
    cout << i << endl;
    for (int j = 0; j < 10; j++) {
        dfs(10 * i + j, n);
    }
}

int main() {
    int n = 40;
    for (int i = 1; i <= 9; i++) {
        dfs(i, n);
    }
    return 0;
}
```

Initial Setup:

We begin by calling dfs(i, 20) for i = 1 to i = 9.

Dry Run (for n = 20):

1. Calling dfs(1, 20):

- The function prints 1.
- Then it recursively calls dfs(10, 20), dfs(11, 20), ..., dfs(19, 20).

Step by step:

- dfs(1, 20):
  - Prints 1.
  - Calls dfs(10, 20):
    - Prints 10.
    - Calls dfs(100, 20), but 100 > 20, so this call ends.
  - Calls dfs(11, 20):
    - Prints 11.
    - Calls dfs(110, 20), but 110 > 20, so this call ends.
  - Calls dfs(12, 20):
    - Prints 12.
    - Calls dfs(120, 20), but 120 > 20, so this call ends.
  - Calls dfs(13, 20):
    - Prints 13.
    - Calls dfs(130, 20), but 130 > 20, so this call ends.
  - Calls dfs(14, 20):
    - Prints 14.
    - Calls dfs(140, 20), but 140 > 20, so this call ends.
  - Calls dfs(15, 20):
    - Prints 15.
    - Calls dfs(150, 20), but 150 > 20, so this call ends.
  - Calls dfs(16, 20):
    - Prints 16.
    - Calls dfs(160, 20), but 160 > 20, so this call ends.
  - Calls dfs(17, 20):
    - Prints 17.
    - Calls dfs(170, 20), but 170 > 20, so this call ends.
  - Calls dfs(18, 20):
    - Prints 18.
    - Calls dfs(180, 20), but 180 > 20, so this call ends.
  - Calls dfs(19, 20):
    - Prints 19.
    - Calls dfs(190, 20), but 190 > 20, so this call ends.

2. Calling dfs(2, 20):

- The function prints 2.
- Then it recursively calls dfs(20, 20).

Step by step:

- dfs(2, 20):
    - Prints 2.
    - Calls dfs(20, 20):
        - Prints 20.
        - Calls dfs(200, 20), but 200 > 20, so this call ends.

At this point, the function has printed the number starting with 2:

Output:-

1
10
11
12
13
14
15
16
17
18
19
2
20
3
4
5
6
7
8
9

Partition in K subsets in C++

```
#include <iostream>
#include <vector>

using namespace std;

int counter = 0;

void solution(int i, int n, int k, int nos,
vector<vector<int>>& ans) {
    if (i > n) {
        if (nos == k) {
            counter++;
            cout << counter << ". ";
            for (auto& set : ans) {
                cout << "[";
                for (auto num : set) {
                    cout << num << " ";
                }
                cout << "] ";
            }
            cout << endl;
        }
        return;
    }

    for (int j = 0; j < ans.size(); j++) {
        if (!ans[j].empty()) {
            ans[j].push_back(i);
            solution(i + 1, n, k, nos, ans);
            ans[j].pop_back();
        } else {
            ans[j].push_back(i);
            solution(i + 1, n, k, nos + 1, ans);
            ans[j].pop_back();
            break;
        }
    }
}

int main() {
    int n = 4;
    int k = 3;
    vector<vector<int>> ans(k);

    solution(1, n, k, 0, ans);

    return 0;
}
```

Step-by-step Execution:

1. i = 1:
   - Try placing 1 in the first subset:
     - Add 1 to ans[0] → ans = [[1], [], []].
     - Recursively call solution(2, 4, 3, 1, ans).
2. i = 2:
   - Try placing 2 in the first subset:
     - Add 2 to ans[0] → ans = [[1, 2], [], []].
     - Recursively call solution(3, 4, 3, 1, ans).
   - Try placing 2 in the second subset:
     - Add 2 to ans[1] → ans = [[1], [2], []].
     - Recursively call solution(3, 4, 3, 2, ans).
3. i = 3:
   - For the current state of ans:
     - For ans[0] = [1, 2]:
       - Try placing 3 in the first subset → ans = [[1, 2, 3], [], []].
       - Recursively call solution(4, 4, 3, 1, ans).
     - For ans[1] = [2]:
       - Try placing 3 in ans[1] → ans = [[1], [2, 3], []].
       - Recursively call solution(4, 4, 3, 2, ans).
4. i = 4:
   - Now, the subsets are filled with i = 1, 2, 3 elements.
   - After backtracking, we update the subsets and print the results when nos == k.

Final Outputs (Valid Partitions):

1. First partition:
   - ans = [[1, 2], [3], [4]]
   - Output: 1. [1 2] [3] [4]
2. Second partition:

	- ans = [[1, 3], [2], [4]] - Output: 2. [1 3] [2] [4] 3. Third partition: - ans = [[1], [2, 3], [4]] - Output: 3. [1] [2 3] [4] 4. Fourth partition: - ans = [[1, 4], [2], [3]] - Output: 4. [1 4] [2] [3] 5. Fifth partition: - ans = [[1], [2, 4], [3]] - Output: 5. [1] [2 4] [3] 6. Sixth partition: - ans = [[1], [2], [3, 4]] - Output: 6. [1] [2] [3 4]
Output:- 1. [1 2] [3] [4] 2. [1 3] [2] [4] 3. [1] [2 3] [4] 4. [1 4] [2] [3] 5. [1] [2 4] [3] 6. [1] [2] [3 4]	

Permutation in C++

```
#include <iostream>
using namespace std;

void permutations(int cb, int nboxes, int items[], int
ssf, int ritems, string asf) {
    if (cb > nboxes) {
        if (ssf == ritems) {
            cout << asf << endl;
        }
        return;
    }

    for (int i = 0; i < ritems; i++) {
        if (items[i] == 0) {
            items[i] = 1;
            permutations(cb + 1, nboxes, items, ssf + 1,
ritems, asf + to_string(i + 1));
            items[i] = 0;
        }
    }

    permutations(cb + 1, nboxes, items, ssf, ritems, asf
+ "0");
}

int main() {
    int nboxes = 3;
    int ritems = 2;
    int cb = 1;
    int ssf = 0;
    int items[ritems] = {0}; // Initialize items array with
0s

    permutations(cb, nboxes, items, ssf, ritems, "");

    return 0;
}
```

Step-by-step Execution:

1. **cb = 1:** We are at the first box. We try all possible items (1 and 2) in the first box.
   - **Item 1:**
     - items[0] = 0, we mark it as used (items[0] = 1).
     - Recursively call permutations(2, 3, [1, 0], 1, 2, "1").
2. **cb = 2:** We are now at the second box. We check all possible items (1 and 2).
   - **Item 1:** (items[0] = 1, already used, so skip).
   - **Item 2:**
     - items[1] = 0, we mark it as used (items[1] = 1).
     - Recursively call permutations(3, 3, [1, 1], 2, 2, "12").
3. **cb = 3:** We are at the third box. We try all possible options:
   - **Item 1 and Item 2:** Both are already used.
   - Add 0 to indicate the third box remains empty.
     - Recursively call permutations(4, 3, [1, 1], 2, 2, "120").
4. **Base case:** cb = 4, output 120.

   Result:

   - We print 120 as a valid permutation.

5. **Backtrack:** Unmark the second item (items[1] = 0), and proceed to the next option in cb = 2.
   - **Item 2:**
     - items[1] = 0, mark it as used (items[1] = 1).
     - Recursively call permutations(3, 3, [1, 1], 2, 2, "20").
6. **Base case:** cb = 4, output 120.

Output:-

120

Permutation of string in C++	

<pre>#include <iostream>
#include <unordered_map>
using namespace std;
 	
void generate(int cs, int ts, unordered_map<char, int>& fmap, string asf) {
    if (cs > ts) {
        cout << asf << endl;
        return;
    }
 	
    for (auto entry : fmap) {
        char ch = entry.first;
        int count = entry.second;
 	
        if (count > 0) {
            fmap[ch]--;
            generate(cs + 1, ts, fmap, asf + ch);
            fmap[ch]++;
        }
    }
}
 	
int main() {
    string str = "abc";
    unordered_map<char, int> fmap;
 	
    for (char ch : str) {
        fmap[ch]++;
    }
 	
    generate(1, str.length(), fmap, "");
 	
    return 0;
}</pre>
	Initial Setup:
	
	1. We create an unordered map fmap to store the frequency of each character in the string.
	    o fmap = {'a': 1, 'b': 1, 'c': 1}.
	2. Call generate(1, 3, fmap, "") to start generating the permutations.
	
	Step-by-Step Execution:
	
	1. **First Call: generate(1, 3, {'a': 1, 'b': 1, 'c': 1}, "")**
	    o **cs = 1, ts = 3** (we want a total of 3 characters in the permutation).
	    o We iterate over the characters in fmap. Starting with 'a':
	        ▪ **Character 'a'**:
	            ▪ Count > 0: Use 'a', decrease count in fmap to {'a': 0, 'b': 1, 'c': 1}.
	            ▪ Recursively call generate(2, 3, {'a': 0, 'b': 1, 'c': 1}, "a").
	2. **Second Call: generate(2, 3, {'a': 0, 'b': 1, 'c': 1}, "a")**
	    o **cs = 2, ts = 3**.
	    o Iterate again, start with 'a' but it's count 0, so skip it.
	    o Move to **character 'b'**:
	        ▪ **Character 'b'**:
	            ▪ Count > 0: Use 'b', decrease count in fmap to {'a': 0, 'b': 0, 'c': 1}.
	            ▪ Recursively call generate(3, 3, {'a': 0, 'b': 0, 'c': 1}, "ab").
	3. **Third Call: generate(3, 3, {'a': 0, 'b': 0, 'c': 1}, "ab")**
	    o **cs = 3, ts = 3**.
	    o Iterate again, starting with 'a' and 'b', both of which have counts 0, so skip them.
	    o Move to **character 'c'**:
	        ▪ **Character 'c'**:
	            ▪ Count > 0: Use 'c', decrease count in fmap to {'a': 0, 'b': 0, 'c': 0}.
	            ▪ Recursively call generate(4, 3, {'a': 0, 'b': 0, 'c': 0}, "abc").
	4. **Base Case: generate(4, 3, {'a': 0, 'b': 0, 'c': 0}, "abc")**
	    o **cs = 4, ts = 3**: We've reached the required length of 3 characters.

	◦ **Output the permutation**: "abc". 5. **Backtrack**: ◦ Return to the previous state where fmap = {'a': 0, 'b': 1, 'c': 1} and asf = "a". ◦ Restore the count of 'c' and continue the loop. **Second Iteration of First Call:** 1. **Second Character 'b' in First Call:** ◦ **Character 'b'**: ▪ Count > 0: Use 'b', decrease count in fmap to {'a': 1, 'b': 0, 'c': 1}. ▪ Recursively call generate(2, 3, {'a': 1, 'b': 0, 'c': 1}, "b"). 2. **Second Call: generate(2, 3, {'a': 1, 'b': 0, 'c': 1}, "b")** ◦ **cs = 2, ts = 3**. ◦ Skip 'b' (count 0) and move to 'a': ▪ **Character 'a'**: ▪ Count > 0: Use 'a', decrease count in fmap to {'a': 0, 'b': 0, 'c': 1}. ▪ Recursively call generate(3, 3, {'a': 0, 'b': 0, 'c': 1}, "ba"). 3. **Third Call: generate(3, 3, {'a': 0, 'b': 0, 'c': 1}, "ba")** ◦ **cs = 3, ts = 3**. ◦ Skip 'a' and 'b', move to 'c': ▪ **Character 'c'**: ▪ Count > 0: Use 'c', decrease count in fmap to {'a': 0, 'b': 0, 'c': 0}. ▪ Recursively call generate(4, 3, {'a': 0, 'b': 0, 'c': 0}, "bac"). 4. **Base Case: generate(4, 3, {'a': 0, 'b': 0, 'c': 0}, "bac")** ◦ **Output the permutation**: "bac".
Output:- cba cab bca bac acb abc	

Remove Invalid Parenthesis in C++

```cpp
#include <iostream>
#include <string>
#include <unordered_set>
#include <stack>
using namespace std;

void solution(string str, int mra,
unordered_set<string>& ans);
int getMin(string str);

void solution(string str, int mra,
unordered_set<string>& ans) {
    if (mra == 0) {
        int mrnow = getMin(str);
        if (mrnow == 0) {
            if (ans.find(str) == ans.end()) {
                cout << str << endl;
                ans.insert(str);
            }
        }
        return;
    }
    for (int i = 0; i < str.length(); i++) {
        string left = str.substr(0, i);
        string right = str.substr(i + 1);
        solution(left + right, mra - 1, ans);
    }
}

int getMin(string str) {
    stack<char> st;
    for (int i = 0; i < str.length(); i++) {
        char ch = str[i];
        if (ch == '(') {
            st.push(ch);
        } else if (ch == ')') {
            if (st.empty()) {
                st.push(ch);
            } else if (st.top() == ')') {
                st.push(ch);
            } else if (st.top() == '(') {
                st.pop();
            }
        }
    }
    return st.size();
}

int main() {
    string str = "((((()))";
    unordered_set<string> ans;
    int mra = getMin(str);
    solution(str, mra, ans);
    return 0;
}
```

Step-by-Step Dry Run:

Step 1: Calculate mra using getMin(str)

The string is "((((()))", and we need to calculate how many parentheses need to be removed to balance the string.

- **Initial String**: "((((()))"
- Using a stack, we process the string:
  - Encountering an opening parenthesis (: Push onto the stack.
  - Encountering a closing parenthesis): Pop an opening parenthesis from the stack (if one exists).
  - After processing the entire string, we find that 2 opening parentheses (do not have corresponding closing parentheses).
- **Result of getMin("((((()))")**: The minimum number of removals (mra) is **2** because we need to remove 2 redundant opening parentheses (.

Step 2: Recursive Function solution(str, mra, ans)

We now start generating possible valid strings by removing parentheses one by one, up to a total of mra = 2 removals.

- **First Call: solution("((((()))", 2, ans)**:
  - The string has 2 removable parentheses, so we explore all possible ways of removing parentheses.

Recursive Steps:

1. **Remove Parenthesis at index 0 (First ()**:
   - String becomes: "(((()))"
   - Call solution("(((()))", 1, ans).
2. **Remove Parenthesis at index 0 again (First (in "(((()))")**:
   - String becomes: "((()))"
   - Call solution("((()))", 0, ans).
3. **Base Case: solution("((()))", 0, ans)**:
   - We check if the string "((()))" is balanced using getMin("((()))").
   - The result is 0, meaning the string is balanced.
   - Since it is valid and not already in ans, we print it and add it to ans.

   Valid String Output: ((()))

4. **Backtrack to Step 2 and explore other removals**:

<table>
<tr><td></td><td>

We explore other combinations, but in this particular case, only the string "((()))" is valid after removing 2 parentheses.
All other combinations generated during recursion either involve invalid strings or are duplicates.

Final Output:
After backtracking through all combinations, the only valid string left is:
((()))
</td></tr>
<tr><td colspan="2">Output:-

((()))</td></tr>
</table>

Subsequence in C++	

<pre>#include <iostream>
#include <string>
 	
using namespace std;
 	
void sol(string q, string a) {
    if (q.length() == 0) {
        cout << a << "-" << endl;
        return;
    }
 	
    char ch = q[0];
    string rest = q.substr(1);
    sol(rest, a);
    sol(rest, a + ch);
}
 	
int main() {
    string s = "abc";
    sol(s, "");
 	
    return 0;
}</pre>
	Initial Setup:
	
	• Call sol("abc", "") to generate all subsequences of the string.
	
	Step-by-Step Execution:
	
	1. **First Call: sol("abc", "")**
	  ○ **q = "abc", a = ""**.
	  ○ Take the first character 'a' from the string and split it into:
	    ▪ **ch = 'a'**, **rest = "bc"**.
	  ○ **Recursively call** sol("bc", "") to handle the case where 'a' is **not** included.
	  ○ **Recursively call** sol("bc", "a") to handle the case where 'a' is **included**.
	
	2. **Second Call: sol("bc", "")**
	  ○ **q = "bc", a = ""**.
	  ○ Take the first character 'b' from the string and split it into:
	    ▪ **ch = 'b'**, **rest = "c"**.
	  ○ **Recursively call** sol("c", "") to handle the case where 'b' is **not** included.
	  ○ **Recursively call** sol("c", "b") to handle the case where 'b' is **included**.
	
	3. **Third Call: sol("c", "")**
	  ○ **q = "c", a = ""**.
	  ○ Take the first character 'c' from the string and split it into:
	    ▪ **ch = 'c'**, **rest = ""**.
	  ○ **Recursively call** sol("", "") to handle the case where 'c' is **not** included.
	  ○ **Recursively call** sol("", "c") to handle the case where 'c' is **included**.
	
	4. **Base Case: sol("", "")**
	  ○ **q is empty**, print the current subsequence: "-".
	
	5. **Base Case: sol("", "c")**
	  ○ **q is empty**, print the current subsequence: "c-".
	
	6. **Return to Third Call: sol("c", "b")**
	  ○ **Recursively call** sol("", "bc") to handle the case where 'c' is **included**.

7. **Base Case: sol("", "bc")**
    - **q is empty**, print the current subsequence: "bc-".

8. **Return to Second Call: sol("bc", "")**
    - Now handle the case where 'b' is **included**:
        - **sol("c", "b")** has been handled, now process the second part.

Continuation for the First Character 'a':

1. **Second Part: sol("bc", "a")**
    - **q = "bc", a = "a"**.
    - Take the first character 'b' from the string and split it into:
        - **ch = 'b'**, **rest = "c"**.
    - **Recursively call** sol("c", "a") to handle the case where 'b' is **not** included.
    - **Recursively call** sol("c", "ab") to handle the case where 'b' is **included**.

2. **Third Call: sol("c", "a")**
    - **q = "c", a = "a"**.
    - Take the first character 'c' from the string and split it into:
        - **ch = 'c'**, **rest = ""**.
    - **Recursively call** sol("", "a") to handle the case where 'c' is **not** included.
    - **Recursively call** sol("", "ac") to handle the case where 'c' is **included**.

3. **Base Case: sol("", "a")**
    - **q is empty**, print the current subsequence: "a-".

4. **Base Case: sol("", "ac")**
    - **q is empty**, print the current subsequence: "ac-".

Final Part of the Execution:

1. **Return to Second Call: sol("c", "ab")**
    - **Recursively call** sol("", "abc") to handle the case where 'c' is **included**.

2. **Base Case: sol("", "abc")**
    - **q is empty**, print the current

	subsequence: "abc-".
Output:- - c- b- bc- a- ac- ab- abc-	

Word Break in C++

```
#include <iostream>
#include <unordered_set>
#include <string>

using namespace std;

void wordBreak(string str, string ans,
unordered_set<string>& dict) {
    if (str.length() == 0) {
        cout << ans << endl;
        return;
    }

    for (int i = 0; i < str.length(); i++) {
        string left = str.substr(0, i + 1);
        if (dict.find(left) != dict.end()) {
            string right = str.substr(i + 1);
            wordBreak(right, ans + left + " ", dict);
        }
    }
}

int main() {
    int n = 5;
    unordered_set<string> dict = {"microsoft", "hiring",
"at", "kolkata"};
    string sentence = "microsofthiring";

    wordBreak(sentence, "", dict);

    return 0;
}
```

Step-by-Step Execution:

1. **First Call**: wordBreak("microsofthiring", "", dict)
   - **Current string**: "microsofthiring"
   - **Dictionary**: {"microsoft", "hiring", "at", "kolkata"}
   - **Answer so far**: ""
   - Loop through the string, checking for each substring that is present in the dictionary:
   - **i = 0**: Substring "m" is not in the dictionary.
   - **i = 1**: Substring "mi" is not in the dictionary.
   - **i = 2**: Substring "mic" is not in the dictionary.
   - **i = 3**: Substring "micro" is not in the dictionary.
   - **i = 4**: Substring "micr" is not in the dictionary.
   - **i = 5**: Substring "micro" is not in the dictionary.
   - **i = 6**: Substring "microsoft" **is** in the dictionary.
     - Now the string becomes "hiring", and the answer so far is "microsoft ".
   - **Recursively call**: wordBreak("hiring", "microsoft ", dict)

2. **Second Call**: wordBreak("hiring", "microsoft ", dict)
   - **Current string**: "hiring"
   - **Dictionary**: {"microsoft", "hiring", "at", "kolkata"}
   - **Answer so far**: "microsoft "
   - Loop through the string "hiring", checking for each substring:
   - **i = 0**: Substring "h" is not in the dictionary.
   - **i = 1**: Substring "hi" is not in the dictionary.
   - **i = 2**: Substring "hir" is not in the dictionary.
   - **i = 3**: Substring "hiri" is not in the dictionary.
   - **i = 4**: Substring "hiring" **is** in the dictionary.
     - Now the string becomes "" (empty), and the answer so far is "microsoft hiring ".
   - **Recursively call**: wordBreak("", "microsoft hiring ", dict)

	3. **Base Case**: wordBreak("", "microsoft hiring ", dict)
	    o **Current string**: ""
	    o **Answer so far**: "microsoft hiring "
	    o Since the string is empty, print the answer: "microsoft hiring"

Output:- microsoft hiring	