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;
}
```

Dry Run Table (Step-by-Step)

We'll list:

- pos: current position in the string
- count: how many characters we've skipped (abbreviated)
- asf: abbreviation-so-far

pos	char	count	asf	Recursive Call
0	p	0	""	choose 'p' → asf = "p"
1	e	0	"p"	choose 'e' → asf = "pe"
2	p	0	"pe"	choose 'p' → asf = "pep"
3	—	0	"pep"	**output: pep**
2	p	1	"pe"	skip 'p' (count = 1)
3	—	1	"pe"	**output: pe1**
1	e	1	"p"	skip 'e' (count = 1)
2	p	0	"p1p"	count > 0 → add 1 then 'p'
3	—	0	"p1p"	**output: p1p**
2	p	2	"p"	skip 'p' (count = 2)
3	—	2	"p"	**output: p2**
0	p	1	""	skip 'p' (count = 1)
1	e	0	"1e"	count > 0 → add 1, then 'e'
2	p	0	"1ep"	choose 'p'
3	—	0	"1ep"	**output: 1ep**
2	p	1	"1e"	skip 'p' (count = 1)
3	—	1	"1e"	**output: 1e1**
1	e	1	""	skip 'e'
2	p	0	"2p"	count = 2 → asf = "2p"
3	—	0	"2p"	**output: 2p**
2	p	2	""	skip 'p'
3	—	3	""	**output: 3**

✅ **Final Output:**

```
pep
pe1
p1p
```

	p2 1ep 1e1 2p 3
Output:- pep pe1 p1p p2 1ep 1e1 2p 3	

All palindromic partition in C++

```
#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;
}
```

Dry Run for Input "abba"

We will track the recursive calls with:

- str: Remaining string to process
- prefix: Currently selected prefix
- ros: Remaining string after prefix
- asf: Accumulated string so far
- Action: What's happening

Step	str	prefix	ros	asf	Action / Reason
1	abba	a	bba	(a)	'a' is palindrome → recurse
2	bba	b	ba	(a)(b)	'b' is palindrome → recurse
3	ba	b	a	(a)(b)(b)	'b' is palindrome → recurse
4	a	a	""	(a)(b)(b)(a)	✅ 'a' is palindrome → print
5	ba	ba	—	—	not palindrome ❌
6	bba	bb	a	(a)(bb)	✅ 'bb' is palindrome → recurse
7	a	a	""	(a)(bb)(a)	✅ 'a' is palindrome → print
8	bba	bba	—	—	not palindrome ❌
9	abba	ab	—	—	not palindrome ❌
10	abba	abb	—	—	not palindrome ❌
11	abba	abba	""	(abba)	✅ 'abba' is palindrome → print

✅ Final Output

(a)(b)(b)(a)
(a)(bb)(a)

	(abba)
Output:- (a)(b)(b)(a) (a)(bb)(a) (abba)	

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 with Table for Input:

- `nboxes = 3`
- `ritems = 2`

We're tracing the recursive calls:

- `cb`: current box index
- `ssf`: selected so far
- `asf`: answer so far

Step	cb	ssf	asf	Action
1	1	0	""	→ i at box 1 → recurse
2	2	1	"i"	→ i at box 2 → recurse
3	3	2	"ii"	→ i at box 3 → recurse
4	4	3	"iii"	✗ too many items (ssf > ritems)
5	3	2	"ii"	→ - at box 3 → ✓ print: `ii-`
6	2	1	"i"	→ - at box 2 → recurse
7	3	1	"i-"	→ i at box 3 → recurse
8	4	2	"i-i"	✓ valid → print: `i-i`
9	3	1	"i-"	→ - at box 3 → recurse
10	4	1	"i--"	✗ too few items
11	1	0	""	→ - at box 1 → recurse
12	2	0	"-"	→ i at box 2 → recurse
13	3	1	"-i"	→ i at box 3 → recurse
14	4	2	"-ii"	✓ valid → print: `-ii`
15	3	1	"-i"	→ - at box 3 → recurse
16	4	1	"-i-"	✗ too few items
17	2	0	"-"	→ - at box 2 → recurse
18	3	0	"--"	→ i at box 3 → recurse
19	4	1	"--i"	✗ too few items
20	3	0	"--"	→ - at box 3 → recurse
21	4	0	"---"	✗ too few items

✓ Final Output

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

Output:-

ii-
i-i
-ii

Friend's pairing in C++

```
#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;
}
```

Function Logic Recap

Dry Run for `n = 3`

Step	i	used	Action	Output (if any)
1	1	`[F, F, F, F]`	`1` unused → go alone: `(1)`	
2	2	`[F, T, F, F]`	`2` unused → go alone: `(2)`	
3	3	`[F, T, T, F]`	`3` unused → go alone: `(3)`	`1.(1) (2) (3)`
4			backtrack to pair `2` and `3`	`2.(1) (2,3)`
5			backtrack to try `1` with `2`	
6	2	`[F, T, T, F]`	`3` unused → alone: `(3)`	`3.(1,2) (3)`
7			backtrack	
8			try `1` with `3`	
9	2	`[F, T, F, T]`	`2` unused → alone: `(2)`	`4.(1,3) (2)`

✓ Final Output

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

Output:-

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

Goldmine2 in C++

```
#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;
}
```

Sample Grid (Visual):

```
[
{ 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 }
]
```

We'll start traversal from **(1,2)** where value = 6

Dry Run Table (DFS Traversal Steps):

Step	Cell Visited	Gold at Cell	Cumulative Sum	Stack (DFS Recursion Path)
1	(1,2)	6	6	(1,2)
2	(0,2)	4	10	(1,2) → (0,2)
3	(0,3)	2	12	(1,2) → (0,2) → (0,3)
4	(0,4)	8	20	...
5	(0,5)	2	22	...
6	(1,5)	4	26	...
7	(2,5)	6	32	...
8	(2,4)	4	36	...
9	(3,4)	2	38	...
10	(3,5)	2	40	...
11	(4,5)	4	44	...
12	(4,4)	2	46	...
13	(4,3)	9	55	...
14	(5,3)	8	63	...
15	(5,4)	5	68	...
16	(5,5)	1	69	...
17	(3,3)	3	72	...
18	(2,3)	1	73	...
19	(2,2)	4	77	...
20	(1,3)	5	82	...
21	(1,1)	3	85	...
22	(2,1)	2	87	...
23	(2,0)	1	88	...
24	(3,0)	2	90	...
25	(4,0)	3	93	...
26	(4,1)	1	94	...
27	(5,1)	7	101	...
28	(5,0)	2	103	...

29	(1,0)	4	107	...
30	(0,1)	1	108	...
31	(3,2)	7	115	...
32	(4,2)	5	120	...

✅ Result:

At the end of this traversal:

- All connected gold cells are visited
- Sum = **120**
- This is the **maximum** among all components

✨ Final Output:

```
Output: 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;
}
```

Dry Run Table for solution(4, 2)

We'll compute this step-by-step recursively:

Function Call	Value Returned	Explanation
solution(1, 2)	0	Base case: Only one person, return 0
solution(2, 2)	(0 + 2) % 2 = 0	Last survivor in 2 people = 0
solution(3, 2)	(0 + 2) % 3 = 2	Last survivor in 3 people = 2
solution(4, 2)	(2 + 2) % 4 = 0	Last survivor in 4 people = 0

✅ **Final Output:**

0

Output:-
0

Largest after k swaps in C++

```
#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;
}
```

Explanation of the Algorithm:

- For every pair (i, j) where i < j, if str[j] > str[i], swap i and j.
- After each swap, check if the new number is greater than the current max_str.
- Recurse with k - 1.
- Backtrack (swap back) to explore other options.

Call#	k	Swap Made (i↔j)	str After Swap	max_str Before	max_str After	Remarks
1	4	0↔6	7234561	1234567	✅ 7234561	New max
2	3	1↔5	7634521	7234561	✅ 7634521	New max
3	2	2↔4	7654321	7634521	✅ 7654321	Final max
4	1	No beneficial swap	-	7654321	7654321	Stop recursion
5	3	1↔4	7534261	7654321	❌	Not greater
6	3	1↔3	7435261	7654321	❌	Not greater
7	2	2↔3 (from 7435261)	7453261	7654321	❌	Still not better
...	-	...	...	...	...	Many paths explored

We only continue recursion when beneficial. As you can see, once 7654321 is reached, **no further recursion produces a better result**, so that becomes the final output.

🏁 **Final Output:**

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;
}
```

Lexicographical Order Output:

This simulates how numbers are sorted like dictionary words:

```
1, 10, 100, ..., 11, 12, ..., 2, 20,
21, ..., 3, 30, ..., 4, 40, ...,
5, ..., 9
```

But only numbers ≤ 40 are printed.

Dry Run Table (Partial for Clarity)

Here's a step-by-step snapshot of what's happening:

Function Call	i	Output	Explanation
`dfs(1, 40)`	1	✓ 1	Valid, print
`dfs(10, 40)`	10	✓ 10	Valid, print
`dfs(100, 40)`	>40	✗	Stop recursion
`dfs(11, 40)`	11	✓ 11	Continue same way
`dfs(12, 40)`	12	✓ 12	…
`...`	…	…	
`dfs(19, 40)`	19	✓ 19	
`dfs(2, 40)`	2	✓ 2	Start from next i
`dfs(20, 40)`	20	✓ 20	
`dfs(21, 40)`	21	✓ 21	
`...`	…	…	
`dfs(40, 40)`	40	✓ 40	Final valid number
`dfs(41, 40)`	>40	✗	Stop here

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 = 3;
    int k = 2;
    vector<vector<int>> ans(k);

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

    return 0;
}
```

Dry Run Table:

Step	i	nos	ans (state)	Action Taken
1	1	0	[[], []]	Put 1 in first empty subset
2	2	1	[[1], []]	Put 2 in subset 0
3	3	1	[[1, 2], []]	Put 3 in subset 0
4	4	1	—	nos != k, discard
5	3	2	[[1, 2], [3]]	✅ Output: [1 2] [3]
6	2	2	[[1], [2]]	✅ Output path starts
7	3	2	[[1, 3], [2]]	✅ Output: [1 3] [2]
8	3	2	[[1], [2, 3]]	✅ Output: [1] [2 3]

📄 **Final Output:**

1. [1 2] [3]
2. [1 3] [2]
3. [1] [2 3]

Output:-
1. [1 2] [3]
2. [1 3] [2]
3. [1] [2 3]

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;
}
```

Key Variables:

Var	Meaning
cb	current box index
ssf	selected so far – number of items placed
items[]	array of 0/1, indicating whether each item (1 to ritems) is used
asf	answer so far – the configuration of items across boxes

Dry Run Table:

cb	items	ssf	asf	Description
1	[0,0]	0	""	Start, box 1
2	[1,0]	1	"1"	Place item 1 in box 1
3	[1,1]	2	"12"	Place item 2 in box 2
4	[1,1]	2	"120"	✓ Output: item1 in box1, item2 in box2
3	[1,1]	2	"102"	item2 in box3
3	[1,0]	1	"10"	skip box 2
4	[1,1]	2	"102"	✓ Output
2	[0,1]	1	"2"	item2 in box 1
3	[1,1]	2	"21"	item1 in box2
4	[1,1]	2	"210"	✓ Output
3	[0,1]	1	"20"	box2 empty
4	[1,1]	2	"201"	✓ Output
2	[0,0]	0	"0"	box1 empty
3	[1,0]	1	"01"	item1 in box2
4	[1,1]	2	"012"	✓ Output
3	[0,1]	1	"02"	item2 in box2
4	[1,1]	2	"021"	✓ Output
3	[0,0]	0	"00"	box2 empty
4	-	0	"000"	✗ Not valid – ssf < ritems

Output:-

120

102
210
201
012
021

Permutation of string in C++

```
#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;
}
```

Goal:

Generate **all permutations of "abc"** using recursion and a frequency map.

Setup:

- fmap: { a:1, b:1, c:1 }
- ts = total size = 3
- cs = current size (starts from 1)
- asf = answer so far

Dry Run Table

Call Stack	fmap (a,b,c)	asf	cs	Output?
generate(1, 3, {1,1,1}, "")				
└ a → generate(2, 3, {0,1,1}, "a")		"a"	2	
└ b → generate(3, 3, {0,0,1}, "ab")		"ab"	3	
└ c → generate(4, 3, {0,0,0}, "abc")		"abc"	4	✓ Print
└ c → backtrack to "ab"				
└ c → generate(3, 3, {0,1,0}, "ac")		"ac"	3	
└ b → generate(4, 3, {0,0,0}, "acb")		"acb"	4	✓ Print
└ b → backtrack to "a"				
└ b → generate(2, 3, {1,0,1}, "b")		"b"	2	
└ a → generate(3, 3, {0,0,1}, "ba")		"ba"	3	
└ c → generate(4, 3, {0,0,0}, "bac")		"bac"	4	✓ Print
└ c → generate(3, 3, {1,0,0}, "bc")		"bc"	3	
└ a → generate(4, 3, {0,0,0}, "bca")		"bca"	4	✓ Print
└ c → generate(2, 3, {1,1,0}, "c")		"c"	2	
└ a → generate(3, 3, {0,1,0}, "ca")		"ca"	3	
└ b → generate(4, 3, {0,0,0}, "cab")		"cab"	4	✓ Print
└ b → generate(3, 3, {1,0,0}, "cb")		"cb"	3	
└ a → generate(4, 3, {0,0,0}, "cba")		"cba"	4	✓ Print

Output:-
cba
cab
bca
bac
acb
abc

Remove Invalid Parenthesis in C++

```
#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;
}
```

Goal:

Remove the **minimum number** of parentheses to make the string valid.

🔧 Step 1: getMin("((((()))")

Step	Char	Stack	Action
1	(	(	push
2	(	((	push
3	(	(((	push
4	(	((((	push
5	(	(((((	push
6	)	((((	pop (match)
7	)	(((	pop (match)
8	)	((	pop (match)

□ Final stack size = ((→ **2 unmatched**
✅ So mra = 2 (Minimum Removals Allowed)

🔁 Step 2: Recursive Dry Run Table

We'll track:

Call #	Current String (str)	Removals Left (mra)	Action Taken	Is Valid (getMin=0)?	Output
1	((((()))	2	Start	❌ (getMin=2)	
2	(((())	1	Removed char at index 0	❌ (getMin=1)	
3	((()))	0	Removed char at index 0	✅ (getMin=0)	✅ ((()))
4	(same string)	0	Duplicate path	✅	(skipped by set)
...	many more paths tried	≤ 0	But not valid	❌	

✅ Only ((())) satisfies getMin == 0 **with exactly 2 removals**

	✅ Your unordered_set prevents printing duplicates 📜 **Final Output:** ((()))
Output:- ((()))	

Subsequence in C++

```
#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;
}
```

Execution Tree:

We'll denote:

- q = remaining string
- a = answer so far

Call #	q	a	Output if base case
1	abc	""	
2	bc	""	
3	c	""	
4	""	""	-
5	""	"c"	c-
6	bc	"b"	
7	c	"b"	
8	""	"b"	b-
9	""	"bc"	bc-
10	abc	"a"	
11	bc	"a"	
12	c	"a"	
13	""	"a"	a-
14	""	"ac"	ac-
15	bc	"ab"	
16	c	"ab"	
17	""	"ab"	ab-
18	""	"abc"	abc-

✅ Final Output (Subsequences with -):

-
c-
b-
bc-
a-
ac-
ab-
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;
}
```

Dry Run Table:

Call	str	ans	Loop i	left = str.substr(0, i+1)	left in dict?	Action Taken
1	microsofthiring	""	0	m	✗	skip
1	microsofthiring	""	1	mi	✗	skip
1	microsofthiring	""	2	mic	✗	skip
1	microsofthiring	""	3	micr	✗	skip
1	microsofthiring	""	4	micro	✗	skip
1	microsofthiring	""	5	micros	✗	skip
1	microsofthiring	""	6	microso	✗	skip
1	microsofthiring	""	7	microsof	✗	skip
1	microsofthiring	""	8	microsoft	✓	Recurse with str=hiring, ans=microsoft
2	hiring	microsoft	0	h	✗	skip
2	hiring	microsoft	1	hi	✗	skip
2	hiring	microsoft	2	hir	✗	skip
2	hiring	microsoft	3	hiri	✗	skip
2	hiring	microsoft	4	hirin	✗	skip
2	hiring	microsoft	5	hiring	✓	Recurse with str="", ans=microsoft hiring
3	""	microsoft hiring	-	—	—	Print: microsoft hiring

✓ Final Output:

microsoft hiring

Output:- microsoft hiring	

Check number exists in array in C++

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

int array11(int nums[], int index, int length) {
    if (index >= length) {
        return 0;
    }
    int small = array11(nums, index + 1, length);
    if (nums[index] == 11) {
        return 1 + small;
    } else {
        return small;
    }
}

int main() {
    int arr[] = {1, 11, 3, 11, 11, 11};
    int length = sizeof(arr) / sizeof(arr[0]);
    cout << array11(arr, 0, length) << endl;
    return 0;
}
```

Input

```
arr = {1, 11, 3, 11, 11, 11}
```

Function Call Tree

```
array11(arr, 0, 6)
→ nums[0] == 1 → skip
→ array11(arr, 1, 6)
  → nums[1] == 11 → count +1
  → array11(arr, 2, 6)
    → nums[2] == 3 → skip
    → array11(arr, 3, 6)
      → nums[3] == 11 → count +1
      → array11(arr, 4, 6)
        → nums[4] == 11 → count +1
        → array11(arr, 5, 6)
          → nums[5] == 11 → count +1
          → array11(arr, 6, 6)
            → index >= length → return 0
```

Dry Run Table

Call	`index`	`nums[index]`	Matches 11?	Return Value
`array11(arr, 0, 6)`	0	1	✗	0 + 4 = 4
`array11(arr, 1, 6)`	1	11	✓	1 + 3 = 4
`array11(arr, 2, 6)`	2	3	✗	0 + 3 = 3
`array11(arr, 3, 6)`	3	11	✓	1 + 2 = 3
`array11(arr, 4, 6)`	4	11	✓	1 + 1 = 2
`array11(arr, 5, 6)`	5	11	✓	1 + 0 = 1
`array11(arr, 6, 6)`	6	N/A	N/A	0

Output

```
4
```

Output:-

```
4
```

Check Palindrome in C++

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

bool isStringPalindrome(const string& input, int s, int e) {
    // Base case: if start index equals end index, the string is a palindrome
    if (s == e) {
        return true;
    }
    // If the characters at the start and end do not match, it's not a palindrome
    if (input[s] != input[e]) {
        return false;
    }
    // If there are more characters to compare, call the function recursively
    if (s < e + 1) {
        return isStringPalindrome(input, s + 1, e - 1);
    }
    return true;
}

bool isStringPalindrome(const string& input) {
    int s = 0;
    int e = input.length() - 1;
    return isStringPalindrome(input, s, e);
}

int main() {
    cout << (isStringPalindrome("abba") ? "true" : "false") << endl;
    return 0;
}
```

Input

```
string = "abba"
```

🔍 Function Call Tree

```
isStringPalindrome("abba", 0, 3)
→ 'a' == 'a' ✓
→ isStringPalindrome("abba", 1, 2)
   → 'b' == 'b' ✓
   → isStringPalindrome("abba", 2, 1)
      → s > e → return true
```

📋 Dry Run Table

Call	s	e	input[s]	input[e]	Match?	Return
isStringPalindrome ("abba", 0, 3)	0	3	'a'	'a'	✓	✓
isStringPalindrome ("abba", 1, 2)	1	2	'b'	'b'	✓	✓
isStringPalindrome ("abba", 2, 1)	2	1	N/A	N/A	Base	✓

🟢 Output

```
true
```

Your program will print:

```
true
```

Output:-
true

Check sorted in C++

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

bool sorted(int arr[], int n) {
    if (n == 1 || n == 0) {
        return true;
    } else if (arr[n - 1] < arr[n - 2]) {
        return false;
    } else {
        return sorted(arr, n - 1);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << boolalpha << sorted(arr, n) << endl;
    return 0;
}
```

Input

```
arr[] = {1, 2, 3, 4, 5}
n = 5
```

🔄 Recursive Calls

We check if the last two elements are in correct order (`arr[n-2] <= arr[n-1]`), and recursively reduce the array size.

📋 Dry Run Table

Call	n	arr[n-2]	arr[n-1]	Comparison	Result
`sorted(arr, 5)`	5	4	5	4 ≤ 5	✅
`sorted(arr, 4)`	4	3	4	3 ≤ 4	✅
`sorted(arr, 3)`	3	2	3	2 ≤ 3	✅
`sorted(arr, 2)`	2	1	2	1 ≤ 2	✅
`sorted(arr, 1)`	1	—	—	Base case	✅

✅ Output

```
true
```

Your program will print:

```
true
```

Output:-
true

Count zeroes in C++

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

int cnt = 0;

int countZerosRec(int input) {
    // Base case for initial input of 0
    if (input == 0 && cnt == 0) {
        return 1;
    }

    // Base case for recursion
    if (input == 0) {
        return cnt;
    }

    // Check if the current last digit is zero
    if (input % 10 == 0) {
        cnt++;
    }

    // Recursive call to process the next digit
    return countZerosRec(input / 10);
}

int main() {
    cout << countZerosRec(10034) << endl;
    return 0;
}
```

Dry Run for countZerosRec(10034)

Call	input	input % 10	is zero?	sum
countZerosRec(10034)	10034	4	✗	0 + next
countZerosRec(1003)	1003	3	✗	0 + next
countZerosRec(100)	100	0	✓	1 + next
countZerosRec(10)	10	0	✓	1 + next
countZerosRec(1)	1	-	✗	0

➞ Total = 1 + 1 = **2**

Output:-
2

Factorial in C++

```
#include <iostream>

using namespace std;

int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        int prev = fact(n - 1);
        return n * prev;
    }
}

int main() {
    cout << fact(6) << endl;
    return 0;
}
```

Dry Run Table for fact(6):

Call Level	n	Recursive Call	Returned Value	Computation
1	6	6 * fact(5)	720	6 * 120
2	5	5 * fact(4)	120	5 * 24
3	4	4 * fact(3)	24	4 * 6
4	3	3 * fact(2)	6	3 * 2
5	2	2 * fact(1)	2	2 * 1
6	1	1 * fact(0)	1	1 * 1
7 (Base)	0	return 1	1	Base case hit

🔼 Final Output:

720

Output:-
720

Min-Max in C++

```
#include <iostream>
#include <climits> // for INT_MAX and INT_MIN
using namespace std;

int getMin(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return min(arr[i], getMin(arr, i + 1, n - 1));
    }
}

int getMax(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return max(arr[i], getMax(arr, i + 1, n - 1));
    }
}

int main() {
    int arr[] = {12, 8, 45, 67, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum element of array: " <<
getMin(arr, 0, n) << endl;
    cout << "Maximum element of array: " <<
getMax(arr, 0, n) << endl;
    return 0;
}
```

Dry Run Table for getMin(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	min(12, getMin(1, 4))	8	min(12, 8)
2	1	8	min(8, getMin(2, 3))	8	min(8, 9)
3	2	45	min(45, getMin(3, 2))	9	min(45, 9)
4	3	67	min(67, getMin(4, 1))	9	min(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

Dry Run Table for getMax(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	max(12, getMax(1, 4))	67	max(12, 67)
2	1	8	max(8, getMax(2, 3))	67	max(8, 67)
3	2	45	max(45, getMax(3, 2))	67	max(45, 67)
4	3	67	max(67, getMax(4, 1))	67	max(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

✔ Final Output:

Minimum element of array: 8
Maximum element of array: 67

Output:-

Minimum element of array: 8
Maximum element of array: 67

Stair Case in C++

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

// Function to calculate number of ways to reach nth step
int staircase(int n) {
    // Base cases
    if (n == 0 || n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }
    // Recursive case
    return staircase(n-1) + staircase(n-2) + staircase(n-3);
}

int main() {
    // Test case
    int n = 7;
    cout << staircase(n) << endl;
    return 0;
}
```

Dry Run Table for `staircase(7)`

Track the **calls** and their **return values** from the bottom up (memoized-style for understanding):

`n`	`staircase(n)` Calculation	Result
0	1 (base case)	1
1	1 (base case)	1
2	2 (base case)	2
3	staircase(2) + staircase(1) + staircase(0)	2 + 1 + 1 = 4
4	staircase(3) + staircase(2) + staircase(1)	4 + 2 + 1 = 7
5	staircase(4) + staircase(3) + staircase(2)	7 + 4 + 2 = 13
6	staircase(5) + staircase(4) + staircase(3)	13 + 7 + 4 = 24
7	staircase(6) + staircase(5) + staircase(4)	24 + 13 + 7 = 44

✅ Final Output:

```
44
```

Output:-
44

Subset Sum in C++

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

// Function to calculate subset sums recursively
void subsetSums(int arr[], int l, int r, int sum) {
    // Base case: if l exceeds r, print the current sum
    if (l > r) {
        cout << sum << " ";
        return;
    }

    // Recursive case: include current element arr[l] in the subset sum
    subsetSums(arr, l + 1, r, sum + arr[l]);
}

int main() {
    // Initialize the array and its length
    int arr[] = {5, 4, 3, 5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Call the function to calculate subset sums, starting with l=0, r=n-1, and initial sum=0
    subsetSums(arr, 0, n - 1, 0);

    return 0;
}
```

Input:

```
int arr[] = {5, 4, 3, 5, 4};
```

This adds:

```
5 + 4 + 3 + 5 + 4 = 21
```

And when `l > r`, it prints `sum`, which is `21`.

Dry Run Table (for your input):

Step	l	r	sum	Action
1	0	4	0	`sum = 0 + arr[0] = 5`
2	1	4	5	`sum = 5 + arr[1] = 9`
3	2	4	9	`sum = 9 + arr[2] = 12`
4	3	4	12	`sum = 12 + arr[3] = 17`
5	4	4	17	`sum = 17 + arr[4] = 21`
6	5	4	21	`l > r`, print `21` and return

Final Output:

```
21
```

Output:-
21

Tiling in C++

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

int tilingways(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return tilingways(n - 1) + tilingways(n - 2);
}

int main() {
    cout << tilingways(4) << endl;
    return 0;
}
```

Function Call	Returns	Reason
tilingways(4)	?	tilingways(3) + tilingways(2)
tilingways(3)	?	tilingways(2) + tilingways(1)
tilingways(2)	?	tilingways(1) + tilingways(0)
tilingways(1)	1	Base case
tilingways(0)	0 ✗	Wrong base case — it **should be 1**
tilingways(2)	1 + 0 = 1	
tilingways(1)	1	Base case
tilingways(3)	1 + 1 = 2	
tilingways(2)	1	Already computed
tilingways(4)	2 + 1 = 3 ✓	

Output:-
3