<https://leetcode.com/problems/remove-invalid-parentheses/>

Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return *all the possible results*. You may return the answer in **any order**.

**Example 1:**

Input: s = "()())()"

Output: ["(())()","()()()"]

**Example 2:**

Input: s = "(a)())()"

Output: ["(a())()","(a)()()"]

**Example 3:**

Input: s = ")("

Output: [""]

**Constraints:**

1 <= s.length <= 25

s consists of lowercase English letters and parentheses '(' and ')'.

There will be at most 20 parentheses in s.

**Attempt 1: 2022-12-15**

**Solution 1: Backtracking (60 min)**

**Style 1: 'not use' before 'use' case, no stringbuilder.setLength() required**

class Solution {

    public List<String> removeInvalidParentheses(String s) {

        List<String> result = new ArrayList<String>();

        if(s == null) {

            return result;

        }

        if(s.length() == 0) {

            result.add("");

            return result;

        }

        char[] chars = s.toCharArray();

        // Limit max removal maxRemovableLeft and maxRemovableRight for backtracking boundary.

        // Otherwise it will exhaust all possible valid substrings, not shortest ones

        // The openParenthesisPair argument is to verify that parentheses are matching,

        // for example, "()", visit '(' to increment integer variable openParenthesisPair one,

        // and visit ')' to decrement integer variable openParenthesisPair variable one

        int maxRemovableLeft = 0, maxRemovableRight = 0, openParenthesisPair = 0;

        for(int i = 0; i < chars.length; i++) {

            if(chars[i] == '(') {

                maxRemovableLeft++;

            } else if(chars[i] == ')') {

                if(maxRemovableLeft != 0) {

                    maxRemovableLeft--;

                } else {

                    maxRemovableRight++;

                }

            }

        }

        StringBuilder sb = new StringBuilder();

        // De-duplicate by adding to a HashSet

        Set<String> set = new HashSet<String>();

        dfs(chars, 0, set, sb, maxRemovableLeft, maxRemovableRight, 0);

        return new ArrayList<String>(set);

    }

    private void dfs(char[] chars, int index, Set<String> set, StringBuilder sb, int maxRemovableLeft, int maxRemovableRight, int openParenthesisPair) {

        // Max removal maxRemovableLeft maxRemovableRight and openParenthesisPair are non negative

        if(maxRemovableLeft < 0 || maxRemovableRight < 0 || openParenthesisPair < 0) {

            return;

        }

        // Base case: index does not exceed chars.length

        if(index == chars.length) {

            // Scan from left to right, avoiding invalid strs (on the fly) by checking num of openParenthesisPair == 0

            if(maxRemovableLeft == 0 && maxRemovableRight == 0 && openParenthesisPair == 0) {

                set.add(sb.toString());

            }

            return;

        }

        /\*\*

         \* Refer to

         \* https://discuss.leetcode.com/topic/30743/easiest-9ms-java-solution/3?page=1

         \* To understand it, first make sure you understand the idea of backtracking which is

           extremely powerful. It's like going into a maze, and do a silly exhausted depth

           first search. When you meet the dead end, you need to go back to the last successful

           decision point, and try another direction.

           Back to this question. Based on 3 cases of current char: (, ) or others, we have 3

           directions of at each maze point. Notice that no matter which direction we choose,

           there is exactly one char added. So after going to the dead end and gathering some

           chars, it will go backwards by removing one char at a time. (To make it clearer,

           I've modified the last line)

           So this is what I found extremely useful pattern but some might not know:

           int len = sb.length();

           ... backtracking ...

           sb.setLength(len);

           It has at least two merits:

           (1) StringBuilder is much more efficient than String concat.

           (2) sb.setLength(len) method is naturally adapted to backtracking.

        \*/

        int len = sb.length(); // Prepare checkpoint for backtracking

        char c = chars[index];

        // If it's '(', either use it, or remove it

        if(c == '(') {

            /\*\*

             \* Note: The order to 'not use it' and 'use it' cannot revert

             \* Refer to

             \* https://discuss.leetcode.com/topic/30743/easiest-9ms-java-solution/29?page=2

             \* The order of these 2 calls does matter.

               StringBuilder is mutable - when we use sb.append(c), we've altered sb's value

               (its internal char array) by appending one more character to the end.

               For example, at the beginning, sb.toString() was "a(b)c" , and the current char was '(':

               If we do the 2 calls as in the order as in @yavinci's original code, we will get

               sb.toString().equals("a(b)c") => true

               dfs(s, i + 1, res, sb, rmL - 1, rmR, open);    // not use (

               sb.toString().equals("a(b)c") => true

               dfs(s, i + 1, res, sb.append(c), rmL, rmR, open + 1); // use (

               On the other hand, if we switch the order of those 2 calls,

               sb.toString().equals("a(b)c") => true

               dfs(s, i + 1, res, sb.append(c), rmL, rmR, open + 1); // use (

               sb.toString().equals("a(b)c") => false,

               because sb has been appended, and now it has value "a(b)c(". we'll pass a wrong value

               to the next call.

               dfs(s, i + 1, res, sb, rmL - 1, rmR, open);    // not use (

               Note: To test it in easy way just use single "(" as input

            \*/

            // Not use it

            dfs(chars, index + 1, set, sb, maxRemovableLeft - 1, maxRemovableRight, openParenthesisPair);

            // Use it

            dfs(chars, index + 1, set, sb.append(c), maxRemovableLeft, maxRemovableRight, openParenthesisPair + 1);

        // If it's ')', either use it, or remove it

        } else if(c == ')') {

            // Not use it

            dfs(chars, index + 1, set, sb, maxRemovableLeft, maxRemovableRight - 1, openParenthesisPair);

            // Use it (Optimize by only when additional open parenthesis > 0 then able to use ')')

            if(openParenthesisPair > 0) {

                dfs(chars, index + 1, set, sb.append(c), maxRemovableLeft, maxRemovableRight, openParenthesisPair - 1);

            }

        } else {

            // The input may contain other chars and must remain into final result, just append it

            dfs(chars, index + 1, set, sb.append(c), maxRemovableLeft, maxRemovableRight, openParenthesisPair);

        }

        sb.setLength(len); // Backtracking, lastly set StringBuilder to the last decision point.

    }

}

Time Complexity : O(2^N)

Since in the worst case we will have only left parentheses in the expression and for every bracket

we will have two options i.e. whether to remove it or consider it. Considering that the expression

has N parentheses, the time complexity will be O(2^N)

Space Complexity: O(N)

Because we are resorting to a recursive solution and for a recursive solution there is always stack

space used as internal function states are saved onto a stack during recursion. The maximum depth of

recursion decides the stack space used. Since we process one character at a time and the base case

for the recursion is when we have processed all of the characters of the expression string, the size

of the stack would be O(N). Note that we are not considering the space required to store the valid

expressions. We only count the intermediate space here.

**Style 2: 'use' before 'not use' case, stringbuilder.setLength() required**

class Solution {

    public List<String> removeInvalidParentheses(String s) {

        int maxRemovableLeft = 0, maxRemovableRight = 0, openParenthesisPair = 0;

        char[] chars = s.toCharArray();

        for(int i = 0; i < chars.length; i++) {

            if(chars[i] == '(') {

                maxRemovableLeft++;

            } else if(chars[i] == ')') {

                if(maxRemovableLeft == 0) {

                    maxRemovableRight++;

                } else {

                    maxRemovableLeft--;

                }

            }

        }

        Set<String> set = new HashSet<String>();

        helper(chars, 0, set, maxRemovableLeft, maxRemovableRight, 0, new StringBuilder());

        return new ArrayList<String>(set);

    }

    private void helper(char[] chars, int index, Set<String> set, int maxRemovableLeft, int maxRemovableRight, int openParenthesisPair, StringBuilder sb) {

        if(maxRemovableLeft < 0 || maxRemovableRight < 0 || openParenthesisPair < 0) {

            return;

        }

        if(index == chars.length) {

            if(maxRemovableLeft == 0 && maxRemovableRight == 0 && openParenthesisPair == 0) {

                set.add(sb.toString());

            }

            return;

        }

        int len = sb.length();

        char c = chars[index];

        if(c == '(') {

            // Use

            helper(chars, index + 1, set, maxRemovableLeft, maxRemovableRight, openParenthesisPair + 1, sb.append(c));

            // Added to reset change after 'use' case

            sb.setLength(len);

            // Not use

            helper(chars, index + 1, set, maxRemovableLeft - 1, maxRemovableRight, openParenthesisPair, sb);

        } else if(c == ')') {

            // Use it (Optimize by only when additional open parenthesis > 0 then able to use ')')

            if(openParenthesisPair > 0) {

                helper(chars, index + 1, set, maxRemovableLeft, maxRemovableRight, openParenthesisPair - 1, sb.append(c));

            }

            // Added to reset change after 'use' cas()e

            sb.setLength(len);

            // Not use

            helper(chars, index + 1, set, maxRemovableLeft, maxRemovableRight - 1, openParenthesisPair, sb);

        } else {

            helper(chars, index + 1, set, maxRemovableLeft, maxRemovableRight, openParenthesisPair, sb.append(c));

        }

        sb.setLength(len);

    }

}

Time Complexity : O(2^N)

Since in the worst case we will have only left parentheses in the expression and for every bracket

we will have two options i.e. whether to remove it or consider it. Considering that the expression

has N parentheses, the time complexity will be O(2^N)

Space Complexity: O(N)

Because we are resorting to a recursive solution and for a recursive solution there is always stack

space used as internal function states are saved onto a stack during recursion. The maximum depth

of recursion decides the stack space used. Since we process one character at a time and the base

case for the recursion is when we have processed all of the characters of the expression string,

the size of the stack would be O(N). Note that we are not considering the space required to store

the valid expressions. We only count the intermediate space here.

**Refer to**

<https://leetcode.com/problems/remove-invalid-parentheses/solutions/75034/easiest-9ms-java-solution/>

Here I share my DFS or backtracking solution. It's 10X faster than optimized BFS.

Limit max removal rmL and rmR for backtracking boundary. Otherwise it will exhaust all possible valid substrings, not shortest ones.

Scan from left to right, avoiding invalid strs (on the fly) by checking num of open parens.

If it's '(', either use it, or remove it.

If it's ')', either use it, or remove it.

Otherwise just append it.

Lastly set StringBuilder to the last decision point.

In each step, make sure:

i does not exceed s.length().

Max removal rmL rmR and num of open parens are non negative.

De-duplicate by adding to a HashSet.

Compared to 106 ms BFS (Queue & Set), it's faster and easier. Hope it helps! Thanks.

public List<String> removeInvalidParentheses(String s) {

    int rmL = 0, rmR = 0;

    for (int i = 0; i < s.length(); i++) {

        if (s.charAt(i) == '(') {

            rmL++;

        } else if (s.charAt(i) == ')') {

            if (rmL != 0) {

                rmL--;

            } else {

                rmR++;

            }

        }

    }

    Set<String> res = new HashSet<>();

    dfs(s, 0, res, new StringBuilder(), rmL, rmR, 0);

    return new ArrayList<String>(res);

}

public void dfs(String s, int i, Set<String> res, StringBuilder sb, int rmL, int rmR, int open) {

    if (rmL < 0 || rmR < 0 || open < 0) {

        return;

    }

    if (i == s.length()) {

        if (rmL == 0 && rmR == 0 && open == 0) {

            res.add(sb.toString());

        }

        return;

    }

    char c = s.charAt(i);

    int len = sb.length();

    if (c == '(') {

        dfs(s, i + 1, res, sb, rmL - 1, rmR, open);    // not use (

    dfs(s, i + 1, res, sb.append(c), rmL, rmR, open + 1);      // use (

    } else if (c == ')') {

        dfs(s, i + 1, res, sb, rmL, rmR - 1, open);            // not use  )

    dfs(s, i + 1, res, sb.append(c), rmL, rmR, open - 1);      // use )

    } else {

        dfs(s, i + 1, res, sb.append(c), rmL, rmR, open);

    }

    sb.setLength(len);

}

**Style 3: Add 'lastNotUsed' (equal to 'lastSkipped') variable to naturally avoid duplicates without using set**

class Solution {

    public List<String> removeInvalidParentheses(String s) {

        int maxRemovableLeft = 0, maxRemovableRight = 0, openParenthesisPair = 0;

        char[] chars = s.toCharArray();

        for(int i = 0; i < chars.length; i++) {

            if(chars[i] == '(') {

                maxRemovableLeft++;

            } else if(chars[i] == ')') {

                if(maxRemovableLeft == 0) {

                    maxRemovableRight++;

                } else {

                    maxRemovableLeft--;

                }

            }

        }

        List<String> result = new ArrayList<String>();

        helper(chars, 0, result, maxRemovableLeft, maxRemovableRight, 0, new StringBuilder(), '#');

        return result;

    }

    /\*\*

        Let's consider the case of if (lastSkipped != '(') dfs(s, i + 1, res, temp + c, rmL, rmR, open + 1, '&'); // use (

        Now, this if statement, can only be true when the last character from the previous recursion call was '(',

        in another word, [last skipped != '(']  ==> [last not used != '('] ==> [last used == '('], we only want

        to run current dfs if last used character is '('

        So essentially, the substring you're processing is '((' where the first '(' is at i and the second one

        is at i + 1. The cases where we use the first '(' and skip the next, and the case where the first one

        is used, and the next '(' is skipped, are equivalent.

        Hence, these form a duplicate case. Hence we have this if condition.

    \*/

    private void helper(char[] chars, int index, List<String> result, int maxRemovableLeft, int maxRemovableRight, int openParenthesisPair, StringBuilder sb, char lastNotUsed) {

        if(maxRemovableLeft < 0 || maxRemovableRight < 0 || openParenthesisPair < 0) {

            return;

        }

        if(index == chars.length) {

            if(maxRemovableLeft == 0 && maxRemovableRight == 0 && openParenthesisPair == 0) {

                result.add(sb.toString());

            }

            return;

        }

        int len = sb.length();

        char c = chars[index];

        if(c == '(') {

            // Not use

            helper(chars, index + 1, result, maxRemovableLeft - 1, maxRemovableRight, openParenthesisPair, sb, c);

            // Use

            if(lastNotUsed != c) {

                helper(chars, index + 1, result, maxRemovableLeft, maxRemovableRight, openParenthesisPair + 1, sb.append(c), '#');

            }

        } else if(c == ')') {

            // Not use

            helper(chars, index + 1, result, maxRemovableLeft, maxRemovableRight - 1, openParenthesisPair, sb, c);

            // Use

            if(lastNotUsed != c) {

                helper(chars, index + 1, result, maxRemovableLeft, maxRemovableRight, openParenthesisPair - 1, sb.append(c), '#');

            }

        } else {

            helper(chars, index + 1, result, maxRemovableLeft, maxRemovableRight, openParenthesisPair, sb.append(c), '#');

        }

        sb.setLength(len);

    }

}

Time Complexity : O(2^N)

Since in the worst case we will have only left parentheses in the expression and for every bracket we will

have two options i.e. whether to remove it or consider it. Considering that the expression has N parentheses,

the time complexity will be O(2^N)

Space Complexity: O(N)

Because we are resorting to a recursive solution and for a recursive solution there is always stack space

used as internal function states are saved onto a stack during recursion. The maximum depth of recursion

decides the stack space used. Since we process one character at a time and the base case for the recursion

is when we have processed all of the characters of the expression string, the size of the stack would be O(N).

Note that we are not considering the space required to store the valid expressions. We only count the

intermediate space here.

**Refer to**

<https://leetcode.com/problems/remove-invalid-parentheses/solutions/75034/easiest-9ms-java-solution/comments/78195>

Based on the original solution, handled duplicates and removed the HashSet.

    public List<String> removeInvalidParentheses(String s) {

        int rmL = 0, rmR = 0;

        for (int i = 0; i < s.length(); i++) {

            if (s.charAt(i) == '(') {

                rmL++;

            } else if (s.charAt(i) == ')') {

                if (rmL != 0) {

                    rmL--;

                } else {

                    rmR++;

                }

            }

        }

        List<String> res = new ArrayList<>();

        dfs(s, 0, res, "", rmL, rmR, 0, '&');

        return res;

    }

    public void dfs(String s, int i, List<String> res, String temp, int rmL, int rmR, int open, char lastSkipped) {

        if (rmL < 0 || rmR < 0 || open < 0) {

            return;

        }

        if (i == s.length()) {

            if (rmL == 0 && rmR == 0 && open == 0) {

                res.add(temp);

            }

            return;

        }

        char c = s.charAt(i);

        if (c == '(') {

            if (rmL > 0)

                dfs(s, i + 1, res, temp, rmL - 1, rmR, open, '(');    // not use (

            if (lastSkipped != '(')

                dfs(s, i + 1, res, temp + c, rmL, rmR, open + 1, '&');      // use (

        } else if (c == ')') {

            if (rmR > 0)

                dfs(s, i + 1, res, temp, rmL, rmR - 1, open, ')');            // not use  )

            if (open > 0 && (lastSkipped != ')'))

                dfs(s, i + 1, res, temp + c, rmL, rmR, open - 1, '&');      // use )

        } else {

            dfs(s, i + 1, res, temp + c, rmL, rmR, open, '&');

        }

    }

<https://leetcode.com/problems/remove-invalid-parentheses/solutions/75034/easiest-9ms-java-solution/comments/451896>

Let's consider the case of

if (lastSkipped != '(') dfs(s, i + 1, res, temp + c, rmL, rmR, open + 1, '&'); // use (

**Now, this if statement, can only be true when the last character from the previous recursion call was '(',  in another word, [last skipped != '(']  ==> [last not used != '('] ==> [last used == '('], we only want to run current dfs if last used character is '('So essentially, the substring you're processing is '((' where the first '(' is at**

**i and the second one is at**

**i + 1.**

**The cases where we use the first '(' and skip the next, and the case where the first one is used, and the next '(' is skipped, are equivalent. Hence, these form a duplicate case. Hence we have this if condition.**

**Solution 2:  BFS (60 min)**

class Solution {

    public List<String> removeInvalidParentheses(String s) {

        List<String> result = new ArrayList<String>();

        Set<String> visited = new HashSet<String>();

        Queue<String> q = new LinkedList<String>();

        q.offer(s);

        visited.add(s);

        boolean found = false;

        while(!q.isEmpty()) {

            String cur = q.poll();

            if(isValid(cur)) {

                result.add(cur);

                found = true;

            }

            // Use boolean flag 'found' to block further depth (smaller

            // length string) check, because we only want minimum remove

            // which equal to find maximum length result, but keep check

            // current depth by continue

            if(found) {

                continue;

            }

            for(int i = 0; i < cur.length(); i++) {

                if(cur.charAt(i) == '(' || cur.charAt(i) == ')') {

                    String next = cur.substring(0, i) + cur.substring(i + 1);

                    if(!visited.contains(next)) {

                        q.offer(next);

                        visited.add(next);

                    }

                }

            }

        }

        return result;

    }

    private boolean isValid(String s) {

        int count = 0;

        for(int i = 0; i < s.length(); i++) {

            // When encounter open parenthesis increase count

            if(s.charAt(i) == '(') {

                count++;

            // In the case of valid parentheses, the count cannot be 0

            // if a closing parenthesis is encountered, because when

            // scanning from left to right, since 0 means no '(' available

            // when to build a closure with current ')', acknowledge as

            // no valid open parenthesis available

            } else if(s.charAt(i) == ')') {

                if(count == 0) {

                    return false;

                }

                count--;

            }

        }

        return count == 0;

    }

}

Time Complexity : O(N \* 2^N-1)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/remove-invalid-parentheses/solutions/75032/share-my-java-bfs-solution/>

The idea is straightforward, with the input string s, we generate all possible states by removing one ( or ), check if they are valid, if found valid ones on the current level, put them to the final result list and we are done, otherwise, add them to a queue and carry on to the next level.

The good thing of using BFS is that we can guarantee the number of parentheses that need to be removed is minimal, also no recursion call is needed in BFS.

Thanks to [@peisi](https://leetcode.com/discuss/user/peisi), we don't need stack to check valid parentheses.

Time complexity:

In BFS we handle the states level by level, in the worst case, we need to handle all the levels, we can analyze the time complexity level by level and add them up to get the final complexity.

On the first level, there's only one string which is the input string s, let's say the length of it is n, to check whether it's valid, we need O(n) time. On the second level, we remove one ( or ) from the first level, so there are C(n, n-1) new strings, each of them has n-1 characters, and for each string, we need to check whether it's valid or not, thus the total time complexity on this level is (n-1) x C(n, n-1). Come to the third level, total time complexity is (n-2) x C(n, n-2), so on and so forth...

Finally we have this formula:

T(n) = n x C(n, n) + (n-1) x C(n, n-1) + ... + 1 x C(n, 1) = n x 2^(n-1).

Following is the Java solution:

public class Solution {

    public List<String> removeInvalidParentheses(String s) {

      List<String> res = new ArrayList<>();

      // sanity check

      if (s == null) return res;

      Set<String> visited = new HashSet<>();

      Queue<String> queue = new LinkedList<>();

      // initialize

      queue.add(s);

      visited.add(s);

      boolean found = false;

      while (!queue.isEmpty()) {

        s = queue.poll();

        if (isValid(s)) {

          // found an answer, add to the result

          res.add(s);

          found = true;

        }

        if (found) continue;

        // generate all possible states

        for (int i = 0; i < s.length(); i++) {

          // we only try to remove left or right paren

          if (s.charAt(i) != '(' && s.charAt(i) != ')') continue;

          String t = s.substring(0, i) + s.substring(i + 1);

          if (!visited.contains(t)) {

            // for each state, if it's not visited, add it to the queue

            queue.add(t);

            visited.add(t);

          }

        }

      }

      return res;

    }

    // helper function checks if string s contains valid parantheses

    boolean isValid(String s) {

      int count = 0;

      for (int i = 0; i < s.length(); i++) {

        char c = s.charAt(i);

        if (c == '(') count++;

        if (c == ')' && count-- == 0) return false;

      }

      return count == 0;

    }

}

**Refer to**

[L22.Generate Parentheses (Ref.L20,L301)](note://661954690B6C4F31A22EEC40B79B40BC)