```
FUNCTION is_palindrome:
  INPUT: string str
  OUTPUT: bool [TRUE or FALSE]
  USAGE: flag = is_palindrome(str)
BEGIN

  n = length(str)
  IF n <= 1, RETURN TRUE
  ELSE, RETURN str[0] == str[n-1] AND
                    is_palindrome(str[1,n-2])

END // is_palindrome
```

ALGORITHMS AND DATA STRUCTURES

RECURSION (II)

NON-MATH EXAMPLE: PALINDROMES

- Recursion is not only applicable to math problems
  - Any time you can devise a reduced self-similar decomposition strategy to a problem, it admits a recursive solution
- **Example**: Check whether `string` is a palindrome
  - *palindrome* is a string that reads identically backward and forward, e.g. “kayak”, “noon”, “race car”

NON-MATH EXAMPLE: PALINDROMES

- Final task:
  - Use recursion to test if `string` is a palindrome
- Recursive strategy:
  - Check to see that the first and last characters are the same
  - Check whether the substring generated by removing the first and last characters is itself a palindrome

NON-MATH EXAMPLE: PALINDROMES

- Inefficient implementation

```
bool isPalindrome(string str)
{
 int len = str.length();
 if(len <= 1) {
  return true;
 } else {
  return str[0] == str[len - 1]
        && isPalindrome(str.substr(1, len - 2));
 }
}
```

- How to speed it up?
  - Remove redundancy in implementation

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

NON-MATH EXAMPLE: PALINDROMES

- To improve the implementation
  - Do not calculate `string` length every time
    - call `length()` once then pass that information down
  - Do not call the method `substr(...)`
    - avoid copying of substrings in recursive calls
  - Both require changing the prototype of the original recursive function

THINKING RECURSIVELY

- Two approaches to programming:
  - Reductionism: the belief that the whole of an object can be understood merely by understanding its parts
  - Holism: the belief that the whole is more that the sum of the parts that make it up
- When coding, it is useful to go back and forth between the two strategies
  - however…

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

THINKING RECURSIVELY

- When designing recursive strategies
  - Reductionism is the enemy
  - Stick to a holistic approach
- How?
  - Adopt recursive leap of faith
  - Choose right decomposition level
  - Identify base case(s)

THINKING RECURSIVELY

- Remember:
  - if there's a problem, it lies in your recursive implementation
  - not in the recursive mechanism itself
    - solve problem at a single level of recursion
    - looking further down won't help
  - check always your formulation of the recursive decomposition

AVOIDING ERRORS WITH RECURSION

1. Do you begin by checking for the simple case(s)?
   - Make sure you solved them correctly
2. Does your recursion make the problem simpler?
   - Check for nonterminating recursion
3. Does your recursion eventually reach the simple case(s)?
4. Are the subproblems really identical to the whole problem?
   - Check that solutions to subproblems provide the complete solution when combined (relates to leap of faith)

RECURSIVE STRATEGIES

- When problem come as mathematical definition
  - easy to apply recursion
- For most complex problems this is not the case
  - some problems have long iterative solution
  - but have a short recursive solution
- Remember: length of the code does ***not*** relate to complexity of the solution

RECURSIVE STRATEGIES

- There are several instances of complex problems not easily written mathematically
  - The towers of Hanoi
  - Graphical recursion
  - The subset sum problem
  - String permutations

RECURSIVE STRATEGIES

- There are several instances of complex problems not easily written mathematically
  - The towers of Hanoi
  - Graphical recursion
- Let us look at two such complex examples:
  - The subset sum problem
  - String permutations

EXAMPLE: THE SUBSET SUM PROBLEM

- Given a sequence of integers `a := {a`$_1$`, a`$_2$`, a`$_3$`, …,a`$_N$`}` find out if it is possible to find a subsequence such that its sum gives a target sum `target`
- Example
  - If `a := {-2, 1, 3, 8}` and `target = 7`, the answer is yes, b/c there is a subsequence `b := {-2, 1, 8}` that amounts to 7
  - If `target = 5` the answer is no, there is no such sub-sequence that adds up to 5

SEARCH FOR A RECURSIVE SOLUTION

- Pseudocode for predicate solution

```
PREDICATE subsetSumExists:
 INPUT: integer_set set, integer target
 OUTPUT: boolean
 USAGE: subsetSumExists(seq, num)
BEGIN
 IF set is empty // base case
  RETURN target is 0;
 ELSE
  Recursive call to simplify the problem
END // subsetSumExists
```

- The subset could be find by either removing or keeping an element (*inclusion/exclusion pattern*)

SOLUTION: INCLUSION/EXCLUSION PATTERN

- Inclusion/exclusion pattern (C++ pseudocode)

```
bool subsetSumExists(Set<int> set, int target)
{
  if(set.isEmpty())
    return target == 0;
  else
    int element = set.first();
    Set<int> rest = set - element;
    return subsetSumExists(rest, target)
           || subsetSumExists(rest, target - element);
}
```

- Strategy has two branches, one including and another not including a particular element, this strategy is called the *inclusion/exclusion pattern*

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXERCISE: PERMUTATIONS OF STRINGS

- How to generate all permutations of a `string` of fixed length?
  - For instance: given a `string intro = “ABC”;` we should generate the `set {“ABC”, “ACB”, “BAC”, “BCA”, “CAB”, “CBA”}`
  - using a function `generatePermutations(intro);`
- Think recursively
  - Base case: empty or literal `string`
  - Divide problem into smaller ones (*divide-and-conquer*)

EXERCISE: PERMUTATIONS OF STRINGS

- A possible recursive strategy

```
set<string> generatePermutations(const string & str)
{
  set<string> final;

  if (str.size() == 0) {
    final.insert("");
    return final;
  } else {
    char first = str[str.size() - 1];
    string tmp = str.substr(0, str.size() - 1);
    set<string> s_tmp = generatePermutations(tmp);
    final = putCharInGrooves(s_tmp, first);
    return final;
  }
}
```

SUMMARY

- Recursion is similar to stepwise refinement in that both methods simplify a big problem to subproblems
  - The difference: recursion divides the problem into subproblems similar to the original one
- To use recursion, you must be able to find the base case and a recursive decomposition
- Understanding a recursive program would be easier if you keep a holistic approach as opposed to a reductionist one