Recursion and Problem Solving

1 Problem Solving and Backtracking

Backtracking is an algorithm paradigm that can be used for finding one or more solutions to certain types of computational problems. It works by incrementally building partial solutions, checking if a partial solution can be extended to a complete solution, and discarding a partial solution if it determines that it cannot be so extended. It is best understood through examples. We begin with a practical, non-computational, example.

Imagine that you are in a forest without a map. The forest consists of thick brush and trees with many paths through them. You are at a point that we will call point **A**. You are trying to get to point **B**. There are many forks along the path you are on, sometimes it forks into over a dozen paths. Without a map that lets you know which paths to take, you need a methodical way to get to point **B**. It may even be possible that there is no path from **A** to **B**. The only thing you are guaranteed is that none of these paths are cyclical, so you will never return to a place you already visited by going straight on any path from that point.

Backtracking is a method that can organize your search for **B**. Suppose you use the following rules:

1. Walk straight until you come to a fork in the road (the end of the current partial solution) or you reach **B**. If you reach **B**, you are done (a complete solution.)
2. At every fork in the road, always choose the rightmost path that you have not yet tried (a systematic way to extend a partial solution.) If, when you come to a fork, you have tried all of the paths possible at that fork already, then turn around and walk back to the closest fork you just left (this is the backtrack part), mark the branch on which you traveled back as "tried already" (a discarded partial solution), and then repeat this step.
3. If walking back to the nearest fork (backtracking) brings you back to point **A** because all branches of the first fork failed to reach **B**, and there are no other branches to try at **A**, then there is no path to **B**.

This algorithm will allow you to reach **B** if there is a path to it. The aspects of it that make it an example of backtracking are that it methodically tries all solutions by tracing backwards when a search leads to failure and choosing the next possible search. This is an example of a problem that is considered to be solved if you find a single solution. Usually if you are lost in a forest you just want to find the way out, rather than finding all possible ways out. If we wanted to find all possible paths from **A** to **B**, then in step 1, we would not terminate if we reached **B**. In addition, in step 2, we would not consider paths as being discarded unless they failed to reach **B**.

For backtracking to be a useful technique, the problem to be solved must admit to two conditions:

1. There has to be a concept of a partial candidate solution.
2. There must be an efficient (fast) test to determine whether a partial solution can be completed to a valid solution.

1.1 Backtracking and Recursion

Recursion is a useful tool for writing backtracking algorithms because it implicitly traces back to the last guess for you. When a recursive function returns to the point immediately after it was called, it has traced backwards. By putting the recursive calls into either a loop or a set of cascading if-statements, you can organize the function to systematically check all possible guesses.

Example 1: The Sum of Four Squares

LaGrange's 4-squares theorem states that any natural number can be written as the sum of four squares (including 0 if need be.) Suppose we wanted an algorithm that, given any natural number, could find four squares whose sum equals the number. Backtracking can do this. The idea is to pick a square less than the number, subtract it, and then find three numbers that add up to the difference. If we succeed, then we found the four numbers. If we do not, then we back up to the first number we picked and try a different first number. We can proceed by picking the smallest first number first, or by picking the largest first number first. Either way we can proceed methodically. The algorithm below picks the smallest number first.

```
int tryout_solution( int number, int num_squares )
{
    int i;

    if ( number == 0 ) {
        // If number == 0 it can be written as the sum of however many 0's we
        // need so we have succeeded.
        return true;
    }

    // assert: number > 0
    if (num_squares == 0 )
        // If we reach here, since number > 0 and we have used up our quota of
        // four squares, we could not find 4 squares whose sum is number
        return false;

    // try to find a number i such that i*i is less than number
    // if we do, then subtract i*i from the number and recursively
    // do the same thing for number-i*i with one less square than before.
    // if one particular i fails, we try the next i. This is the backtracking
    // part.
    for ( i = 1; i*i <= number; i++ ) {
        int square = i*i;
        if ( tryout_solution(number - square, num_squares -1 ) ) {
            printf ( " %d" , square );
            if ( num_squares < 4 )
                printf( " + " );
            return true;
        }
    }
    return false;
}
```

The above function would be called with an initial value of 4 for the second parameter, `num_squares`, as in:

```
if ( tryout_solution( value, 4 ) )
    printf ( " = %d\n", value );
```

The function attempts to find `num_squares` squares that sum to `number`. If `number` is zero, it is trivial to satisfy so it returns `true`. Since each recursive call diminishes `num_squares`, it is possible that it is zero. If `number` is not zero but `num_squares` is zero, it means that it has run out of chances – it has used four squares but their sum is not the original number, so it returns `false`. Otherwise there is still hope – for each square less than `number`, it calls `tryout_solution(number-square, num_squares-1)`, hoping that one of those squares will result in `tryout_solution()` returning `true`. If it does, it means that it found the remaining squares and that `square` is one of the squares that add up to `number`. It prints the value of `square` (with a '+' after if need be), and returns to the calling function. If the main program is named `find_lagrange_squares`, the output could look like

```
$ find_lagrange_squares  2000
1764 +  196 +  36 +  4 = 2000
```

Example 2: The Eight Queens Problem

The eight queens problem is based on chess. A chess board is an eight by eight grid of squares. A queen is a piece that can attack another piece if and only if that piece lies in the same row or column as the queen, or along either of the two diagonals through the queen's square. The Eight Queens Problem asks for a set of eight squares on which to place eight queens in such a way that none can attack any other.

There is a natural backtracking solution to this problem. Since there must be exactly one queen in each column of the board, and exactly one queen in each row, every queen must be in its own unique row and column. We arbitrarily use the columns of the board to organize the search. Assume the columns are numbered 1 to 8. Try to think recursively now. Imagine that you have placed queens on the board already and that the queens that have been placed so far cannot attack each other. In other words, so far the queens on the board are a potential solution. Initially this is true because no queens are on the board. You have been placing the queens on the board by putting them in successive columns. Now suppose further that you have a means of checking, for any given potential position in which you want to place the current queen, whether doing so would put it under attack by one of the queens already on the board. The task at the moment is to place the queen in the current column, so you try each row position in that column to see if it leads to a solution. If there is a row that is safe in this column, then you place the queen and recursively advance to the next column, trying to place the next queen there, unless of course you just placed the 8th queen, in which case you found a solution. However, if there is no row in the current column that is safe for the queen, then you have to backtrack – you have to go back to the queen you placed in the preceding column and try a different row for it, repeatedly if necessary, until it is safe, applying this same recursive strategy there. If you backtrack to the very first queen and cannot find a row for it, then there is no solution to this problem.

The following pseudocode function implements this strategy and is a good example of a backtracking algorithm.

```
bool placeQueens( int current column, int current row)
{
    // Base case.  Try to place Queen in a column after end of board
    // if we reach here it means we placed all queens on board
    if ( Current Column >= 8)  {
        successfully placed all queens so exit with success
     }
     bool isQueenPlaced = false;
     while (Queen is not placed in current Column  &&
            Current Row < 8)
     {
         // If the queen can be attacked in this position, then
         // try moving it to the next row in the current column.
         if (isUnderAttack(current row, current column))
             current row = current row + 1;
         else {
            // Queen is not under attack so this position is good.
            // Advance to next column and try starting in row = 0
            isQueenPlaced = placeQueens(current column+1, 0);

            // If it wasn't possible to put the new Queen in the next
            // column, backtrack by deleting the new Queen and
            // removing the last Queen placed and moving it down one row.
```

```
                if (!isQueenPlaced)
                    current row ++;
             } // end if
         } // end while
         return isQueenPlaced;
    } // end placeQueens
```

The details are left to the reader.

2 Recursion in the Definition of Languages

Another use of recursion is to define infinite sets of various types. There are always at least two rules. The first is analogous to a base case in an induction proof, and can be called the *basis clause* or simply the *basis*. The second is the *inductive clause*. For example, the set of *natural numbers*, denoted $\mathbb{N}$, can be defined by the following recursive definition:

1. $0 \in \mathbb{N}$.
2. $n \in \mathbb{N} \Longrightarrow n+1 \in \mathbb{N}$

Repeated application of Rule 2 generates the set of all natural numbers.

Implicit in any definition of a set is that the set contains nothing but what the definition places into it. This does not have to be stated explicitly. The fact that 1.5 is not a natural number is because it is not placed into the set by either of Rules 1 or 2. Some authors make this rule explicit and call it the *extremal clause*.

A more interesting set of numbers is defined by this recursive definition:

1. $0 \in \mathbb{A}$
2. $n \in \mathbb{A} \Longrightarrow 2n+1 \in \mathbb{A}$

If you apply Rule 2 repeatedly, you will see that this set consists of the numbers 0, 1, 3, 7, 15, 31, and so on, which is the set $\{2^n - 1 | n \in \mathbb{N}\}$.

These two examples show that recursive definitions generate the numbers that are in the set by repeated application of the rules.

2.1 Grammars

Sets of words are called *languages*. Every language has an underlying *alphabet*, which is the set of symbols used to form the words. Just as sets of numbers can be generated by recursive definitions, so can sets of words. A recursive definition that defines a language is called a *grammar*. There are various conventions for the notation used in grammars. We will not follow the conventions exactly. As long as the notation is well-defined, it does not matter what it looks like.

2.2 Syntax of Grammars.

A *rule* in a grammar is of the form

```
variable = replacement_string
```

which means that the variable on the left can be replaced by the replacement string on the right. The replacement string is a sequence of variables and/or symbols of the underlying alphabet. When a variable appears in the replacement string, it is enclosed in angle brackets (<>) to distinguish it from the non-variables in the replacement. Grammars usually have a designated start variable, which is the one that appears on the left-hand side of the rule and is the first rule to be applied. Some books use a special letter such as S to denote this, but here it is enough to use a symbol that is self-explanatory. When a variable can be replaced by more than one replacement string, we use a vertical bar as a symbol meaning "or". An example of a grammar over the alphabet consisting of the letters `a`, `b`, and `c` is

```
1. PAL      = a <PAL> a | b <PAL> b  | c <PAL> c
2. PAL      = a | b | c
3. PAL      =
```

Rule 2 specifies that `PAL` can be replaced by any of the letters `a`, `b`, and `c`. Rule 3 specifies that `PAL` can be replaced by an empty string (also called the **null string**.) What words are in this language? The null string is in this language, as are "a", "b", and "c". By applying Rule 1 and then Rule 3, "aa", "bb", and "cc" are in it too. For example, to get "aa' we write

```
PAL => a<PAL>a => aa
```

By using the first two rules like this we get nine words: "aaa", "aba", "aca", "bab", "bbb", "bcb", "cac", "cbc", and "ccc." What do these words have in common? They are all the same when read forward or backward. A **palindrome** is a word that is the same read forwards and backwards, such as "radar" or "madam." This language is the language of all palindromes over the alphabet consisting of the letters `a`, `b`, and `c`. This is not a proof of this claim, although the claim is true. Proving that a grammar generates a particular set is beyond the scope of these notes; toprove this you need to show that every word that is generated by it is a palindrome and that every palindrome has a "derivation" from the start symbol `PAL` of this grammar.

Exercise 1. Write a recursive function which, when given a C string *s*, returns true or false depending on whether it is a palindrome.

2.3 Genetics

The word palindrome in the context of genetics has a slightly definition than this. A **DNA string**, also called a **DNA strand**, is a finite sequence consisting of the four letters `A`, `C`, `G`, and `T` in any order[1]. The four letters stand for the four **nucleotides**: **adenine**, **cytosine**, **guanine**, and **thymine**. Nucleotides, which are the molecular units from which DNA and RNA are composed, are also called *bases*. Each nucleotide has a **complement** among the four: `A` and `T` are complements, and `C` and `G` are complements. Complements are chemically-related in that when they are close to each other, they form **hydrogen bonds** between them. For example, the complement of `TGGC` is `ACCG`, and the complement of `TCGA` is `AGCT`. Notice that this last string has the property that its complement is the same as the string when read backward.

A sequence of nucleotides is **palindromic** if the complement read right to left is the same as the string read from left to right. For example, the DNA string `TGCAACGCGTTGCA` is palindromic because the complement is `ACGTTGCGCAACGT`, which when read backwards is the original string.

Exercise 2. Write a recursive function that, when given a DNA string *s*, returns true or false depending on whether *s* is palindromic. Note that this is different from the preceding exercise.

[1]Some sources use lowercase while others use uppercase. It does not matter.

2.4 Infix Expressions

An important class of languages relevant to the programmer are the languages of *algebraic expressions*. Intuitively, an algebraic expression is an expression made up of constants and/or variables upon which the operations of addition, subtraction, multiplication, and division are applied. Parentheses are used to change the order of evaluation in the standard form of these expressions, which are known as *infix expressions*, because the operator is always in between its operands.

A grammar that generates the set of all infix expressions whose operands are single digit numbers is:

```
IE       = <IE> <operator> <IE>
IE       = ( <IE> )
IE       = <token>
operator = + | - | * | /
token    = 0 | 1 | 2 | 3 | ... | 9
```

Examples of words in this language (with spaces added for ease of reading):

```
1 + 8 / 3 - ( 4 - 5 ) * 6
5 + 5 + 4 * 3 * (3 - (2 - (9 + 2) / 2) )
```

A grammar that generates the set of all such expressions whose operands are valid C++ identifiers and/or numeric literals requires more rules; this is a simplification.

2.5 Prefix Expressions

Unparenthesized infix expressions are ambiguous in the sense that, unless a precedence is established for the order in which the operators should be applied, an expression could have more than one value. For example,

```
6 + 4 * 3
```

can be interpreted as $6 + (4 * 3) = 18$ or as $(6 + 4) * 3 = 30$ depending upon whether the addition or multiplication takes place first. Operators are given precedence to disambiguate these expressions, and parentheses are used to change the order of evaluation. However, there are unambiguous ways to write algebraic expressions.

A *prefix expression* is one in which the operator precedes its two operands, as in

1. `+ab`
2. `*+abc`
3. `+/ab*-cde`

The first expression is the same as the infix `a+b`. The second applies `*` to the operand `+ab` and the operand `c`, which means that it is equivalent to `(a+b)*c`. The third applies `+` to the operand `/ab` and the operand `*-cde`, which is in turn is `*` applied to `(-cd)` and `e`, which means that it is equivalent to `(a/b)+(c-d)*e`. The general rule is that the operator is applied to the two operands that immediately follow it. The operands may themselves be prefix expressions containing operators, so this procedure is applied recursively.

Prefix expressions are unambiguous under the rules by which they are evaluated. The language of prefix expressions whose operands are single lowercase letters is defined by the following grammar:

```
prefix     = <identifier>
           | <operator> <prefix> <prefix>
operator   = + | - | * | /
identifier = a | b | c | ... | x | y | z
```

Notice that there are no parentheses in these expressions. This leads to recursive algorithms for recognizing and for evaluating prefix expressions. The grammar tells us that a prefix expression is either a single identifier, or it is an operator followed by two prefix expressions. The recognition algorithm should look at the next character, and if

- it is an identifier, it is a prefix, and
- if it is an operator, it has to be followed by two prefix expressions.

The problem is finding where the first prefix ends and the second begins. The key observation is that if you add any characters to the end of a valid prefix expression, you break it – it is no longer a prefix expression. This implies that if we scan across a string and we find the end of a prefix expression, this must be the only end. For example, if we scan `+d*bc`, starting at the `+` character, then the character immediately after `+`, i.e. the `d`, must be the start of the first operand, which is a prefix expression. Since `d` is a prefix expression all by itself, we know it ends there; no other characters can be part of the first operand of `+`. Similarly, when we scan further and see the `*` we begin to look for two more prefix expressions. If we find that the first ends at the `b`, we know that the next character (the `c`) is the start of the second prefix expression.

Another example:

```
+/ab-cd
```

If this is a prefix it is of the form $+E_1E_2$ where E_1 and E_2 are both prefix expressions. Since E_1 begins with /, it is of the form $/E_3E_4$ where $E_3 = a$ and $E_4 = b$. Also, E_2 is of the form $-E_5E_6$ where E_5 =c and $E_6 = d$. The key is therefore to write a function that finds the end of a prefix expression.

2.5.1 Recognizing Prefix Expressions

An algorithm to find the end of a prefix expression:

```
int end_of_prefix(const string & prefix_str, const int first)
{
    int last = prefix_str.length() - 1; // index of last character in string
    if (first < 0 || first > last )     // first is out of range
        return -1;

    char ch = prefix_str[first];        // get character at position first
    if ( is_identifier(ch) )            // if an identifier
        return first;                   // return first since it is also the
                                        // end of its own prefix
    else if ( is_operator(ch))   {
        // recursive call to find end of prefix that starts at the character
        // immediately after first
        int first_end = end_of_prefix(prefix_str, first + 1);

        // check if the call was able to find an end
        // Return of -1 means it failed
        if (first_end > -1)
```

```
                // It succeeded, so return the end of the prefix after it, which
                // starts at first_end+1
                return end_of_prefix(prefix_str, first_end + 1);
            else return -1;
        }
        else
            return -1;
    }
```

Given the preceding algorithm, it is trivial to check whether a string is a prefix expression:

```
    bool is_prefix(string str)
    {
        last_char = end_of_prefix(str, 0);
        return ( last_char >= 0 && last_char == str.length() -1);
    }
```

2.5.2 Evaluating Prefix Expressions

A recursive algorithm that evaluates prefix expressions:

```
    // This algorithm modifies its argument in the process of evaluating it.
    float evaluate_prefix(string & prefix_str)
    {
        char ch = prefix_str[first];        // get character at position first

        // Delete first character from prefix_str;
        prefix_str = prefix_str.substr(first+1);
        if ( is_identifier(ch) )
            return value of the identifier;

        // if the character is an operator, then
        else if ( is_operator(ch))   {
            op = ch;
            operand1 = evaluate_prefix(prefix_str);
            operand2 = evaluate_prefix(prefix_str);
            return operand1 op operand2 ;
        }
```

2.6 Postfix Expressions

Another form of algebraic expression that is also unambiguous in the sense described above is called a ***postfix expression***. In postfix, the operator follows immediately after its operands. The table below shows the prefix expressions from above with their infix and postfix equivalences.

Prefix	**Infix**	**Postfix**
+ab	a+b	ab+
*+abc	(a+b)*c	ab+c*
+/ab*-cde	(a/b)+((c-d)*e)	ab/ cd-e*+

Postfix expressions are used by many calculators. They are also used when a compiler generates assembly code from higher-level code. A *postfix expression* over the alphabet of single lowercase letter operands and the standard operators is defined by the following grammar:

```
postfix     = <identifier>
            |  <postfix> <postfix> <operator>
operator   = + | - | * | \
identifier = a | b | c | ... | x | y | z
```

Here are a few more examples of postfix expressions and their equivalent infix expressions:

Postfix	**Equivalent Infix**
`a b + c *`	`(a + b) * c`
`a b c d e - - - -`	`a - (b - (c - (d - e)))`
`a b * c d * e f * + -`	`(a * b)- ((c * d) + (e * f))`

Like the prefix grammar, this leads to recursive algorithms for recognizing and for evaluating postfix expressions. When we cover stacks, we will see non-recursive algorithms developed using stacks that evaluate postfix expressions and convert infix to postfix.