1 Recursion as a topic

Understand the simple programs discussed in class (sum of integers and palindrome). Understand the Tower of Hanoi example from class. Consider writing your own code to solve the Towers:

```c
void moveTowers(unsigned disks, std::stack<unsigned>& src, std::stack<unsigned>& dst, std::stack<unsigned>& tmp) {
    if (disks == 1) {
        dst.push(src.top()); // add disk to dst
        src.pop(); // remove disk from src.
        return;
    }
    moveTowers(disks-1, src, tmp, dst);
    moveTowers(1, src, dst, tmp);
    moveTowers(disks-1, tmp, dst, src);
}

int main() {
    std::stack<unsigned> src{1, 2, 3, 4, 5}, dst, tmp; // put 5 disks into src.
    moveTowers(5, src, dst, tmp); // move disks from src to dst.
}
```

Here we use an `std::stack<unsigned>` to hold the disks, where the disks are labeled 1, ..., n, because we are only allowed to remove disks from the top of the stack.

2 Some algorithms

Refer to the class notes on bubble sort, selection sort, merge sort, binary search, and simple search. I have nothing to add.

3 Function objects

The C++ standard library provides a class `std::function` for wrapping callable targets. A callable target is any entity on which the call operator can be used, such as a function, a callable class, or a lambda. The template parameter is of the form `R(Args...)`, where `R` is the return type of the callable target, and `Args...` is the list of formal parameters.

```c
struct Functor {
    Functor(const double x = 0) : x_(x) {}  
    double operator() (const double y) const {
        return x_ + y;
    }
    double x_;  
};  

double function(const double x, const double y) {
    return x_ + y;
}
```
4 Mutual Recursion

We consider the example mutually recursive function discussed in class. The terminology used in class is that “terms” are expressions that are added or subtracted from one another, and factors are expressions that are multiplied or divided by one another. Suppose one inputs \((-x \cdot 3 + 1.1) \cdot 2 + 2/x + 4\). We decompose this input as follows:

(1) \((-x \cdot 3 + 1.1) \cdot 2 + 2/x + 4\). (Split into three terms.)

(I) \((-x \cdot 3 + 1.1) \cdot 2\) is a term. (Split into two factors.)

(A) \((-x \cdot 3 + 1.1)\) is a factor. (Remove parentheses and consider \(-x \cdot 3 + 1.1\) as terms.)

(i) 0 is a term. (Evaluate as a factor.)

(a) 0 is factor.

(ii) \(x \cdot 3\) is a term. (Split into two factors.)

(a) \(x\) is a factor.

(b) 3 is a factor.

(iii) 1.1 is a term. (Evaluate as a factor.)

(a) 1.1 is a factor.

(B) 2 is a factor.

(II) \(2/x\) is a term. (Split into two factors.)

(A) 2 is a factor.

(B) \(x\) is a factor.

(III) 4 is a term. (Evaluate as a factor.)

(A) 4 is a factor.

Now, let’s trace through how the program runs when parsing the simple expression \(2 \cdot x + 5\). This expression is stored in the input stream \(in\).

(1) \(\text{combineTerms}\) calls \(\text{combineFactors}(\text{in})\).

(2) \(\text{combineFactors}\) calls \(\text{evaluateFactor}(\text{in})\).
(3) `evaluateFactor` peeks at the next char in the stream, sees that it is not a parenthesis, an `x`, or a `+/-`, from which it infers that the next char must be part of a number, which is reads using the `>>` function. It then returns an `std::function` object that simply returns the number that it read.

(4) `combineFactors` peeks at the next character, sees that it is a `*`, reads the `*` from the stream, and calls `evaluateFactor(in)`.

(5) `evaluateFactor` peeks at the next character, sees that it is an `x`, which is reads from the stream and returns a `std::function` that acts as the identity.

(6) `combineFactors` multiplies the `std::function` returning 2 and the `std::function` returning `x`. It then peeks at the next character, sees that it is a `+`, and returns the resulting multiplied expression.

(7) `combineTerms` peeks at the next character, sees that it is a `+`, removes the `+` from the stream, and calls `combineFactors(in)`.

(8) `combineFactors` calls `evaluateFactor(in)`.

(9) `evaluateFactor` peeks at the next character, sees that it is a 4, and returns an `std::function` that simply returns 4.

(10) `combineFactors` peeks at the next character, sees that it is the terminal character, and returns the `std::function` returning 4.

(11) `combineTerms` adds the `std::function` evaluating `2*x` to the `std::function` returning 5. `combineTerms` peeks at the next character, sees that it is the terminal character, and returns the `std::function` evaluating `2*x+5`.

Now, let’s consider the order in which the functions `combineTerms`, `combineFactors`, and `evaluateFactor` are called on the longer expression `(-x * 3 + 1.1) * 2 + 2/x + 4`. When we say function `F` is called “on” the expression `e`, we mean that the function terminates once it reads the last character in the expression, as discussed in the trace of the program on the expression `2 * x + 5` above.

(1) `combineTerms` on the full expression `(-x * 3 + 1.1) * 2 + 2/x + 4`.

   (I) `combineFactors` on `(-x * 3 + 1.1) * 2`.

      (A) `evaluateFactor` on `(-x * 3 + 1.1)`.

         (i) `combineTerms` on `0 - x * 3 + 1.1`. (Negatives are treated as subtraction from 0.)

            (a) 0 is handled as a special case.

            (b) `combineFactors` on `x * 3`.

               (1) `evaluateFactor` on `x`.

               (2) `evaluateFactor` on `3`.

            (c) `combineFactors` on `1.1`.

               (1) `evaluateFactor` on `1.1`.

      (B) `evaluateFactor` on `2`.

   (II) `combineFactors` on `2/x`.

      (A) `evaluateFactor` on `2`.
(B) evaluateFactor on \( x \).

(III) combineFactors on 4.

(A) evaluateFactor on 4.

The steps above are understood to be combined via the operator in the expression, using the labels above:

- \( (1) = (I) + (II) + (III) \).
  - \( (I) = (A)*(B) \).
    * \( (A) = (i) = (a) - (b) + (c) \).
    · \( (a) = (a) \)
    · \( (b) = (1)*(2) \)
    · \( (c) = (c) \)
    * \( (B) = (B) \)
  - \( (II) = (A) + (B) \)
  - \( (III) = (A) \).

5 Time complexity of algorithms

It is important to know how long it takes for an algorithm to finish its task. A slow algorithm is of course a lot less useful than a fast algorithm. Much of the time algorithms are applied to some kind of data, whether that data is numbers in a vector or nodes and edges in a graph. One might guess that an algorithm is likely to take longer to run when the data that it is used on is larger: sorting a vector of size 10 is certainly much faster than sorting a vector of size 1000000. Accordingly, when describing how long algorithms take to run, we are most interested in how long they take to run as a function of the size of the input:

\[
T(n) = \text{time it takes for a given algorithm to run on data of size } n.
\]

However, it is of course true that \( T(n) \) above will be different depending on the machine on which the algorithm is run; i.e., \( T(n) \) is a function of not just \( n \) but the machine itself. To remove this complication, we make the following modification to \( T(n) \):

\[
T(n) = \text{number of operations needed for the algorithm to run on data of size } n.
\]

Here, an operation is loosely defined as one of:

1. a math operation
2. accessing memory
3. calling a simple function (such as a comparison function, etc.)
and none of these operations can be dependent on the size of the input data \( n \). This new definition of \( T(n) \) does not need to reference a particular machine: the same number of operations will be required regardless of how much power a machine has.

Consider the implementation of merge sort below:

```cpp
void mergeSorted(std::vector<int>& v, size_t beg, size_t mid, size_t end) {
    size_t first = beg, second = mid+1, vecIndex = 0;
    std::vector<int> res(end-beg+1);
    while ((first <= mid) && (second <= end)) {
        if (v[first] < v[second]) {
            res[vecIndex++] = v[first++];
        } else {
            res[vecIndex++] = v[second++];
        }
    }
    while(first <= mid) {
        res[vecIndex++] = v[first++];
    }
    while (second <= end) {
        res[vecIndex++] = v[second++];
    }
    size_t resSize = res.size();
    for (vecIndex = 0; vecIndex < resSize ; ++vecIndex) {
        v[beg + vecIndex] = res[vecIndex];
    }
}

void mergeSort(std::vector<int>& v, size_t beg, size_t end) {
    if (beg == end) {
        return;
    }
    size_t mid = (beg+end)/2;
    mergeSort(v, beg, mid);
    mergeSort(v, mid+1, end);
    mergeSorted(v, beg, mid, end);
}
```

If one were to accurately determine the time complexity, one would have to count every operation in the mergeSorted and mergeSort functions: all of the variable definitions, the comparisons, accessing elements of the vector, etc. Generally, one does not care about the details and instead only cares about the functional form of \( T(n) \). For the function mergeSort above, instead of saying that

\[
T(n) = 3 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2 + 2n,
\]

where the numbers accurately count the number of operations (these numbers are not accurate), one instead says

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn + d,
\]

(1)

where \( c \) and \( d \) are unspecified constants.

Also, when one calculates time complexity, it is an upper bound to the number of operators that is actually calculated. Indeed, if one checks if \( v \) is sorted at the beginning of the mergeSort function, then
the algorithm will finish immediately if \( v \) is a sorted vector. When calculating time complexity, we assume that an algorithm runs to completion to determine the longest amount of time an algorithm will take to run for some input.

Because we are interested only in an upper bound, we use the big-Oh notation \( O(f(n)) \). Indeed, one says that \( g(n) = O(f(n)) \) if \( cf(n) \geq g(n) \) for some constant \( c \) and for all \( n \) large enough. In other words, for \( n \) large enough, for some \( c \), \( cf(n) \) is an upper bound to \( g(n) \). Thus, if one says that

\[
T(n) = O(f(n)),
\]

then there is a constant \( c \) such that the time complexity \( T(n) \leq cf(n) \) for large enough \( n \).

For mergesort above, (1) is because

1. Two \( T(n/2) \) from calling \texttt{mergeSort} individually on the two halves of the original length \( n \) vector.

2. \( cn \) because \texttt{mergeSort} has to loop through both halves of the vector to merge them into a length \( n \) vector, so in total it loops through \( \frac{n}{2} + \frac{n}{2} \) elements. The \( c \) is because for each of those \( n \) elements, we assume we must do \( c \) operations.

3. \( d \) holds all of the \( n \)-independent operations, such as checking \texttt{beg==end}, allocating the variables \texttt{first}, \texttt{second}, and \texttt{vecIndex}, etc.

To solve (1), we simply repeatedly plug the equation into itself repeatedly. Indeed,

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn + d
\]

\[
= 2(2T\left(\frac{n}{4}\right) + c\frac{n}{2} + d) + cn + d
\]

\[
= 4T\left(\frac{n}{2^2}\right) + 2cn + (1+2)d
\]

\[
= 4(2T\left(\frac{n}{2^3}\right) + c\frac{n}{4} + d) + 2cn + (1+2)d
\]

\[
= 2^3T\left(\frac{n}{2^3}\right) + 3cn + (1+2+4)d
\]

\[
= \cdots = 2^\log_2(n)T\left(\frac{n}{2^\log_2(n)}\right) + \log_2(n)cn + (1+2+4+\cdots + 2^{\log_2(n)-1})d
\]

\[
= nT(1) + cn\log_2(n) + (2^{\log_2(n)} - 1)d
\]

\[
= nT(1) + cn\log_2(n) + (n-1)d,
\]

where we use that

\[
1 + r + r^2 + \cdots + r^{k-1} = \frac{r^k - 1}{r - 1}.
\]