Lectures 10, 11, 12

1. Analysis of algorithms: time, space requirements
   (i) Empirical analysis: checking running times, the choice of input data (actual data, random data, perverse data)
   (ii) Theoretical analysis: average case performance, worst case performance

2. Analysis of Selection Sort
   (i) Selection Sort requires $\frac{N(N-1)}{2}$ operations of comparing numbers when applied to an array of size $N$.
   (ii) Implementations (Lectures 7, 8, 9)
       Array implementation is empirically superior.
       Linked list implementation is empirically inferior, but still allows minor improvements (it is possible to eliminate traversal from list_swap function and to avoid swap operation when it is not necessary; the first improvement requires modifying list_max function, one needs to pass the address of k-th node to function list_swap, the second improvement is clear).

3. Examples of recursive algorithms in pseudocode
   (i) Selection Sort
       SelectionSort(list L, integer k)
       {
         if(k==1)
         {
           Sorting is complete;
         }
         else
         {
           Find the index i of the maximal item between 1 and k;
           Exchange L[i] and L[k];
           SelectionSort(L,k-1)
         }
       }
   (ii) Binary Search
       BinarySearch(list L, integer i, integer j, item x)
       {
         if(i>j)
         {
           Write(“Not Found”);
         }
       }
else
{
    Find the index k of the middle item m in the list L[i]-L[j];
    if(x==m)
    {
        Write("Found");
    }
    else if(x<m)
    {
        BinarySearch(L,i,k-1,x);
    }
    else
    {
        BinarySearch(L,k+1,j,x);
    }
}

4. Computational complexity of Binary Search algorithm
   If the number of elements inside the list equals n, then the algorithm requires
   \([\log_2 n]+1\) operations in its worse case.

5. Several examples of C++ implementations of recursive functions
   (i) Factorial
       int Factorial(int n)
       {
           if(n==0)
               return 1;
           else
               return n*Factorial(n-1);
       }
   (ii) Writing decimal numbers digit by digit
        void write_vertical(int number)
        {
            if(number<10)
                cout << number << endl;
            else
            {
                write_vertical(number/10);
                cout << number%10 << endl;
            }
        }
(iii) Decimal to Binary

```cpp
void Dec_to_Bin(int n)
{
    int m, r;

    if(n>0)
    {
        r = n % 2;
        m = n / 2;
        Dec_to_Bin(m);
        cout << r;
    }
}
```

(iv) Fibonacci (a very inefficient implementation)

```cpp
int Fibonacci(int n)
{
    if(n == 0 || n == 1)
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

(v) Pascal’s triangle

```cpp
int C(int n, int k)
{
    if(k == 0 || k == n)
        return 1;
    else
        return C(n-1, k-1) + C(n-1, k);
}
```

6. Two algorithms based on stacks: checking balanced parentheses, evaluating arithmetic expressions

(i) Balanced Parentheses: a bool type function based on stacks

```cpp
bool is_balanced(const string& expression)
{
    Stack<char> store;
    int i;
    char next;
    char discard;
    bool failed = false;
```
for(i=0;i<expression.length() && !failed;i++)
{
    next=expression[i];
    if(next==')')
        store.push(next);
    else if(next==')' && !store.is_empty())
        discard=store.pop();
    else if(next==')' && store.is_empty())
        failed=true;
}
return (store.is_empty() && !failed);
}

(ii) Read and Evaluate: a double type function based on stacks
Functions peek, ignore, isdigit, strchr are explained in the textbook on pages 351,352.

double read_and_evaluate(istream& ins)
{
    Stack<double> numbers;
    Stack<char> operations;
    double number;
    char symbol;

    while(ins.peek() != '\n')
    {
        if(isdigit(ins.peek())||ins.peek()=='.')
        {
            ins >> number;
            numbers.push(number);
        }
        else if(strchr("+-*/",ins.peek())!=NULL)
        {
            ins >> symbol;
            operations.push(symbol);
        }
        else if(ins.peek()==')')
        {
            ins.ignore();
            evaluate_stack_tops(numbers,operations);
        }
        else
            ins.ignore();
    }
    return numbers.pop();
}
void evaluate_stack_tops(Stack<double>& numbers, Stack<char>& operations) {
    double op1, op2;
    op2 = numbers.pop();
    op1 = numbers.pop();
    switch (operations.pop())
    {
        case '+': numbers.push(op1 + op2);
            break;
        case '-': numbers.push(op1 - op2);
            break;
        case '*': numbers.push(op1 * op2);
            break;
        case '/': numbers.push(op1 / op2);
            break;
    }
}