Lab 2: Algorithm Design and Performance – Complexity

1 Operation counting: Assignments and Comparisons

1.1 Background

We often analyze the number of basic operations of a given algorithm to evaluate the time complexity of its performance. There are different kinds of calculations that can be considered basic operations, depending on what the problem is (for example, comparisons, memory accesses, assignments, additions, or multiplications), or any combination of these. In this lab, we are going to run some experiments focusing on two types of basic operations that are `assignment` and `comparison` to analyze the complexity of some familiar algorithms.

For example, we might require an algorithm to calculate the sum of squares of integers from 1 to `n`, with

- `Input`: a integer – `n`
- `Output`: sum of squares – `sum`

, which is presented as follows:

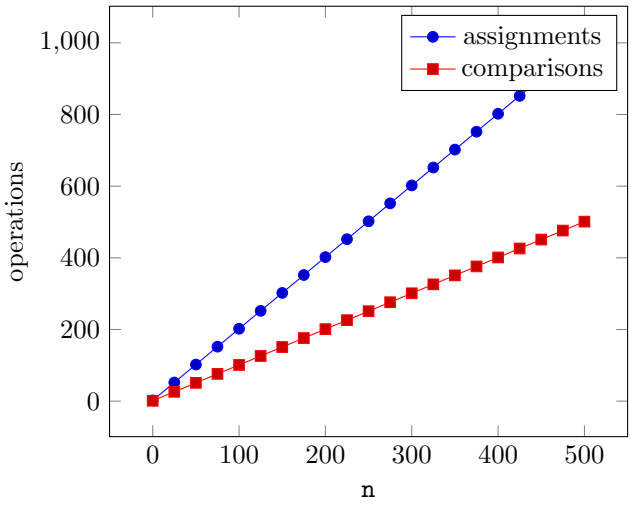
```
1   // func: calculate the sum of integer squares [1,n]
2   int squaresum(int n) {
3       int i = 1;
4       int sum = 0;
5
6       while (i <= n) {
7           sum += i*i;
8           i += 1;
9       }
10      return sum;
11  }
```

Based on theoretical calculation, we do know that this algorithm is $O(n)$ (aka. linear order) whose the numbers of assignments and comparisons are $2n + 2$ and $n + 1$, respectively. The following version of this algorithm will prove the theoretical conclusion above by practically counting two basic operations (assignments and comparisons) with the number of distinct integers input `n`:

```
1   // func: calculate the sum of integer squares [1,n] with assign and compare counting
2   int squaresum(int n, int &count_assign , int &count_compare) {
3       count_assign = 0;
4       count_compare = 0;
5
6       int i = 1; ++count_assign;
7       int sum = 0; ++count_assign;
8
9       while (++count_compare && i <= n) {
10          sum += i*i; ++count_assign ;
11          i += 1; ++count_assign;
12      }
13      return sum;
14  }
```

Visualization By implementing the `squaresum` function with the input integer `n` from 0 to 500, we collect the result as below:

n	assignments	comparisons
0	2	1
25	52	26
50	102	51
75	152	76
100	202	101
125	252	126
150	302	151
175	352	176
200	402	201
225	452	226
250	502	251
275	552	276
300	602	301
325	652	326
350	702	351
375	752	376
400	802	401
425	852	426
450	902	451
475	952	476
500	1,002	501



1.2 Now your turn!

Requirements For each problem, students are required to:

- Modify the code fragment by adding assignment and comparison counting statements
- Implement this algorithm with diferent inputs `n` from 0 to 500
- Make the statistical analysis (table, chart) to prove the theoretical hypothesis

Problem Set

(A)

```
int somesum(int n) {
    int sum = 0, i = 1, j;
    while (i <= n) {
        j = n - i;
        while (j <= i*i) {
            sum = sum + i*j;
            j += 1; }
        i += 1; }
    return sum; }
```

(B)

```
int squaresum_recursion(int n) {
    if (n < 1) return 0;
    else
        return n*n + squaresum_recursion(n-1); }
```

2 Algorithm Design

Any problem can be solved by more than one solution, though some of them are better optimized than others. Your task in this section contains advanced requirements. You have to examine problems, devise two solutions (one is/should be better than another), and then analyze the complexity of your solutions utilizing the method mentioned in part 1.

Requirements For each problem, you are required to:

- Give two algorithms for solving the problem. The first algorithm can be the first thing you think of that works. The second one should be substantially different from the first and should improve upon the first one (i.e. it should use fewer operations). You may think of the best solution first, it is then your job to find a worse solution to compare it to!
- Count the numbers of assignment and comparison operations in your solutions by running them with different inputs
- Make the statistical analysis (table, chart) and compare two solutions

Problem Set

(A) `Majority Element`

- `Input:` An array of integers A, of length n
- `Output:` An integer k such that k appears in more than half of the positions of A if such a k exists. Otherwise "No"
- `Hint:` For the second algorithm you could consider trading space for time

(B) `Greatest common divisor`

- `Input:` Two positive integers u and v, where u > v
- `Output:` The greatest number that divides both u and v
- `Hint:` if you get stuck for a second algorithm, look up Euclid's algorithm

(C) `Word Cloud Problem*`[1] You may have seen word clouds in the media. Some examples are here. They visually represent the important words in a speech or written article. The words that get used most frequently are printed in a larger font, whilst words of diminishing frequency get smaller fonts. Usually, common words like "the", and "and" are excluded. However, we consider the problem of generating a word cloud for all the words. A key operation to generate a word cloud is:

- `Input:` A list W of words, having length n (i.e. n words)
- `Output:` A list of the frequencies of all the words in W, written out in any order, e.g. the=21, potato=1, toy=3, story=3, head=1
- `Hint:` For the second algorithm you could sort the words first. You may assume that this can be done efficiently. For your efficiency calculation, just count the basic operations used after the sorting

(D) `See Saw Problem*` You are running a summer camp for children and are put in charge of the See Saw. This is most fun when the two sides of the See Saw are balanced but you have the issue that children are different weights. Your solution is to give each child a hat of varying weight to balance them out. To help in this task you need to devise an algorithm that takes the weights of two children and the weights of the available hats and works out which hats to give each child.

- `Input:` A pair of numbers (left, right) and list of hat weights W of length n
- `Output:` A different pair of numbers (i, j) such that left + W[i] = right + W[j] or "not possible" if there are no such i and j

[1] You might find better solutions after learning about sorting algorithms