CS29003 Algorithms Laboratory

Assignment No: 0

Last date of submission: 20–July–2017

This assignment is a practical demonstration of the usefulness of characterizing the running times of algorithms in the asymptotic notation. In particular, you will gain an experience how much an exponential-time algorithm can be slower than a polynomial-time algorithm, even for small inputs.

Consider all strings of length n and with symbols coming from the set $\{a,b,c\}$. We are interested in only those of these strings, which contain the pattern (substring) *abc* once and only once. Let $T(n)$ denote the count of these strings. Your task is to compute $T(n)$, given n as input.

Method 1: This method is based on exhaustive search and has exponential running time (to be precise, $O(n3^n)$ running time). The idea is to generate all of the 3^n strings over $\{a,b,c\}$, and to count those strings that contain the pattern *abc* once and only once. Proceed as follows.

Write a function $isvalid(A,n)$ that, given a string A over $\{a,b,c\}$ and its length n as inputs, decides whether the string contains one and only one occurrence of the pattern *abc*. If so, it returns 1, otherwise it returns 0. For example, upon input *babcaab*, the function should return 1, whereas it should return 0 for the inputs *abacbca* and *abccabc*.

Now, write a function $exhsearch(n)$ that implements the exhaustive-search strategy outlined above. It initializes a count to zero. It then generates all the length-n strings one by one, and for each string so generated, it calls $isvalid()$ to find out whether the generated string is of the desired type. If so, it increments the count by one. When all length-n strings are exhaustively handled, it returns the accumulated count. The question is how to generate all ternary strings of length n. This can be done recursively or iteratively.

Here is a recursive strategy. The recursive function assigns $A[0]=a$, $A[0]=b$, and $A[0]=c$ one by one. After each of these assignments, it makes a recursive call for assigning $A[1]$. Each such recursive call assigns $A[1]=a$, $A[1]=b$, and $A[1]=c$, and after each assignment makes a recursive call for assigning $A[2]$, and so on. When all of $A[0],A[1],A[2],\ldots,A[n-1]$ are assigned values, $isvalid()$ is called.

You may also generate the 3^n strings iteratively. Start with a^n, and keep on incrementing the string to the lexicographically next string, and stop after c^n is generated. For example, for $n=3$, this is the sequence of generation of the 27 strings: *aaa*, *aab*, *aac*, *aba*, *abb*, *abc*, *aca*, *acb*, *acc*, *baa*, *bab*, *bac*, *bba*, and so on.

Method 2: For obtaining the count $T(n)$, no strings need to be generated. We can formulate the problem mathematically. A closed-form expression for $T(n)$ may be difficult to arrive at, but formulas involving summations can perfectly describe $T(n)$ and lead to efficient (polynomial-time) algorithms. Let us introduce an $O(n^3)$-time algorithm.

Let $S(i)$ denote the number of strings over $\{a,b,c\}$ of length i containing no occurrence of the pattern *abc*. The desired count $T(n)$ corresponds to strings in which there is a prefix of some length i not containing *abc*, followed by the occurrence of *abc*, followed in turn by a suffix of length $n-i-3$ not containing *abc*. It therefore follows that

$$T(n)=\sum_{i=0}^{n-3} S(i)S(n-i-3). \tag{1}$$

So it suffices to compute $S(0),S(1),\ldots,S(n-3)$ for obtaining $T(n)$. For computing $S(i)$ for $i \geqslant 3$, first notice that the total number of strings of length i is 3^i. Subtract from this count, the counts of strings containing *abc* starting at indices $0,1,2,\ldots,i-3$, and having arbitrary symbols at the remaining $i-3$ positions. This gives the initial formula

$$S(i)=3^i-(i-2)3^{i-3}. \tag{2}$$

This formula is correct only for small values of i. Complete the formula.

Write a function *calculate*(n) that first computes $S(0), S(1), \ldots, S(n-3)$ using the corrected Eqn (2) and subsequently $T(n)$ using Eqn (1). The value of $T(n)$ so computed is to be returned.

The *main*() function:

- Read n from the user. Restrict only to small values of n (like $n \leqslant 18$).
- Call *exhsearch*(n) and record its running time. It is up to you whether you implement the recursive strategy or the iterative strategy. If you have time, implement both. Print the return value and the time taken.
- Call *calculate*(n) and record its running time. Print the return value and the time taken.

Sample output

```
n = 15

+++ Exhaustive search: Iterative
    Count = 4923536, Time taken: 4.271564 sec

+++ Exhaustive search: Recursive
    Count = 4923536, Time taken: 4.446513 sec

+++ Polynomial time calculations
    Count = 4923536, Time taken: 0.000005 sec
```

Submit a single C/C++ source file. Do not use global/static variables.

Appendix: How to Measure Running Time

```
#include <time.h>

clock_t c1, c2;
double runtime;

c1 = clock();
/* Beginning of code whose running time you want to measure */
...
/* End of code whose running time you want to measure */
c2 = clock();

runtime = (double)(c2 - c1) / (double)CLOCKS_PER_SEC;
printf("Running time = %lf seconds\n", runtime);
```