American Journal of Computational Mathematics, **2013, 3, 38-40**
doi:10.4236/ajcm.2013.33B007 Published Online September 2013 (http://www.scirp.org/journal/ajcm)

Reduction in Complexity of the Algorithm by Increasing the Used Memory - An Example

Leonid Kugel, Victor A. Gotlib
The Faculty of Sciences, Holon Institute of Technology (H. I.T), Holon, Israel
Email: leonidk@hit.ac.il, gotlib@hit.ac.il

Received June, 2013

ABSTRACT

An algorithm complexity, or its efficiency, meaning its time of evaluation is the focus of primary care in algorithmic problems solving. Raising the used memory may reduce the complexity of algorithm drastically. We present an example of two algorithms on finite set, where change the approach to the same problem and introduction a memory array allows decrease the complexity of the algorithm from the order $O(n^2)$ up to the order $O(n)$.

Keywords: Algorithm; Complexity Reduction; Memory Usage

1. Introduction

An algorithm efficiency understood as the time of its execution is the focus of primary care in the design and analysis of algorithms ([1, 2]). The lower bond of the execution time of an algorithm directly correlated with the order of complexity of the algorithm. A different approach to the solution to the problem allows sometimes to change the algorithm and to reduce its complexity by introduction an additional memory, for example ([3]). Such a method requires some further analysis of the problem at hand. We illustrate this case with the following example.

2. The Problem

Given two n-digit natural numbers ($n > 0$). One needs to find the number of matching digits at the same positions in both numbers, alongside with overall count of matching digits over the numbers. If a digit has already participated in a matching pair, it is ignored in further encounter. Consider, for example, two numbers 172345 and 287376. The amount of matching identical digits in the equivalent positions is 1 (this is the digit 3). The count of matching identical digits in the various positions is 2 (digit 2 and digit 7). If the same digit in the numbers appears more than once, the count is defined as the minimum number of occurrences in one of the positions of the numbers. For example: 22275 and 86322 specifies the repetition of the number 2 twice.

2.1. The First Approximate

To address the first part of the task (counting the digits on the same positions) we use quite a simple approach: check the number of the units digit in both numbers (the remainder of these numbers divided by 10 gives the number of units in those numbers), and if they are equal, then the corresponding counting variable is increased by 1. Then we "delete" the units digit in both numbers (integer divide by 10). If a match was encountered, the digits are not returned to the original numbers. The performance of this algorithm requires $2O(n)O(1)$ operations. The detailed first approximate algorithm is as follow:

```
*Equals Digits In Position ( num1, num2 )*
count_pos=0, tmp_num1=0, tmp_num2=0
while ( num1> 0 )
if (num1%10 = num2%10 ) {if digits match, we counting them without storing into temporary variables}
count_pos ← count_pos+1
else    {if digits differ, we store them in temporary
    variables}
tmp_num1 ← tmp_num1*10+num1%10
tmp_num2 ← tmp_num2*10+num1%10
{In any case, we delete the "right" units digits}
num1 ← num1 mod 10
num2 ← num2 mod 10
end while
while ( tmp_num1 > 0 ) {restoring original numbers, without replicable digits }
num1 ← num1*10 + tmp_num1%10,
tmp_num1 ← tmp_num1/10
num2 ← num2*10 + tmp_num2%10,
tmp_num2 ← tmp_num2/10
end while
return count_pos
```

```
end algorithm
```

Note that in the above algorithm to the presented problem, it is not significant that the variables *tmp_num*1 and *tmp_num*2 include digits of the original numbers in reverse order. One can just assign as desired.

For the second part of the solution, an auxiliary algorithm-function that receives two parameters: the number and the digit which will be used. The second algorithm checks if the digit is presented in the number. If so, the algorithm deletes its first occurrence from the right (of unit's digit) and returns the number lower by one order, else it returns the number unchanged.

```
DeleteDigit ( num, dig )
   if ( num<10 )
   if ( num==dig ) return 0
   else return num;
   tmp_digit←num % 10; { remainder of division by 10}
   if ( tmp_digit == dig )
   return num div 10
{ return number without the given digit}
       tmp_num← DeleteDigit ( num div 10, dig )
{ check number without units digit (one order lower)}
   if ( tmp_num < num div 10 )
{ digit was deleted, and digit tmp_dig is missing
     Return the missing digit}
   tmp_num ← tmp_num * 10 + tmp_digit
   return tmp_num
   else
   return num
{ return the number without change}
   end if
   end algorithm
```

The number of actions needed to run this algorithm is about $O(n)O(1)$ runtimes in the worst case scenario (see [1, 2]). Using this algorithm, one can count the total number of occurrences of digits in different positions (matched digits in same positions were "deleted" in the first part of the solution)

```
countEqualsRegular ( num1, num2 )
   count_eq ← 0;
   while ( num1 > 0 )
   digit ← num1 % 10
   tmp_num2 ← DeleteDigit ( num2, digit )
   if ( tmp_num2 ≠ num2 )
       {If deletion occurred, count as digits match}
   count_eq ← count_eq + 1
   num1 ← num1 div 10    {deleting reviewed digit}
   return count_eq
   end if
   end while
   end algorithm
```

The number of actions required to perform this algorithm with the auxiliary algorithm will be of order $O(n)O(n)O(1)$. Finally counting, for both parts the runtime will approach $\left(2O(n)+O\left(n^2\right)\right)O(1)$. In other words, the final value of the asymptotic limit of the above functions is of $O\left(n^2\right)$.

A question is, if one can to reduce the order of operations amount in order to solve the problem under consideration (see [3])? To answer is yes, but the using memory should be extended.

2.2. The second Approximate

The idea behind the second solutions is based on the idea of the quick sort method like "counting sort", where we count equal elements in the array, based on the property of the studied values limits. For the first part of the task we construct two auxiliary arrays, which are equal to the length of the given numbers, hence of length *n*. Thus, equal digits on matching positions give us the matching digits on same positions. By presenting the numbers as digits in arrays, we are not obliged to use digits standing on same positions.

For the second part of the problem we use the same idea, but considering the length of the given numbers. We know that all decimal numbers consist of digits from 0 to 9. Thus, creating an auxiliary array of size 10 we can place in it the amount of counted corresponding digits. Checking the corresponding numbers of such arrays for each of these numbers will give us the result of matching numbers ([3, 4]).

```
   CountEquals2 (num1, num2)
   {two arrays of length n for digits of first part of the
solution. Arr_num1 [n], arr_num2 [n], the lengths of our
numbers are given – n.
count_pos←0 count of position matches
   count_eq ←0 count of total matches}
   for ( i ← 1, i ≤ n, i←i+1 )
{ operations of order O(n)*5.
       Counting the number of matches }
   if ( num1%10 = num2%10 )
       count_pos ← count_pos + 1
   else
   {storing the number of units digit in the proper vari-
able }
   arr_num1 [i]←num1%10
   arr_num2[i]←num2%10
   { "deleting" digit of units }
   num1←num1 div 10
   num2←num2 div 10
   end if
   end for
   n ← n – count_pos { the remained amount of digits in
the original numbers. All the digits of our numbers are
stored in the arrays, with the exception of. Those,
matched by position. Create two arrays of 10 elements
each and initiate its values with 0 (digits are not counted
yet.) Then recount the amount of remaining different dig-
```

```
its with}
   {counting arrays - [1], [2] }
   arr_count1={ 0 }, arr_count2={ 0 }
   for (i←1, i ≤ n, i ← i+1 { operations of order O(n)*2 }
   arr_count1[arr_num1[i]]←arr_count1[arr_num1[i] ] +1
   arr_count2 [arr_num2[i]]←arr_count2[arr_num2[i] ]+1
   end for
   {counting matches ([1], [2], [4]) }
   for (i←1, i ≤ n, i ← i+1) { operations of order O(n)*3 }
   if ( arr_count1[i]>0 AND arr_count2[i>0] )
   count_eq←count_eq +min(arr_count1[i], arr_count2[i])
   end for
   return count_pos, count_eq
   end algorithm
```

The number of operations required to perform the algorithm is of an order

$$5O(n)+2O(n)+3O(n)=10\cdot O(n).$$

The final value of the asymptotic function that limits our result from above is $O(n)$ comparing with $O(n^2)$ in the previous versions.

3. Conclusions

For small values of *n*, the second solution may require even more operations than the first solution. There is a range of values of the segment closest to the origin of the function $y=a\cdot n$ while above will limit the function by $y=b\cdot n^2$, where a and b are appropriate constants. However, when *n* tends to infinity (or became sufficiently large), the square function grows faster than linear obviously ([3]).

With this simple example, we wanted to show how the runtimes of different algorithms solving the same problem can be different drastically, when the problem is correctly formulated and the right model and built solution are properly matched.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms", 3rd Edition, MIT Press, Cambridge, MA, 2009.

[2] M. A. Weiss, "Data Structures and Algorithm Analysis in C," Addison-Wesley, 1997.

[3] G. L. Abdulgalimov and L. A. Kugel and N. A. Masimova, "To the Question About Teaching Designing of Information Systems and Data Analysis", Science and Education," No. 9, 2012. pp 81-82 (in Russian). http://elibrary.ru/item.asp?id=18251112

[4] G. L. Abdulgalimov, S. M. Yevstigneev and L. A. Kugel, "Analysis of the Data for Teaching the Basics of Programming", Proceedings of the X National Russian Conference "IT Education in the Russian Federation", 16-18 May 2012, Moscow, Moscow State University, pp 273-275 (in Russian). http://2012.xn----8sbacgtleg3cfdxy.xn--p1ai/upload/IT-EDUCATION-2012-book.pdf