ITU University, Copenhagen

# Karatsuba Algorithm using Sums of Differences

Advanced Algorithms: Supervisor: Philip Bille

Elly Nkya
12/19/2007

Contents

# Chapter 1 Goal of the project

Mathematical operations are central part of many algorithms. In order to achieve faster computational speeds, these operations need to be efficient. Integer multiplication is one of those operations. Classrooms everywhere teach the classic method of multiplying each digit on one number with a digit of the other resulting in a complexity of $O(n^2)$ .

Karatsuba introduced a method that required $O(n^{1.58})$ . His method used Divide & Conquer.

In this project, a similar algorithm is introduced. The algorithm uses the intuition of using Divide and Conquer as in the Karatsuba, but this time, squares of differences are utilized instead. Various optimizations are then discussed and implemented to improve its computational efficiency. The end effect is a faster algorithm which improves in the constant k on $O(n^{1.58}) = kn^{1.58}$

# Chapter 2 Karatsuba Multiplication Algorithm

Before we go into discussion of this other method we will briefly describe the essence of the Karatsuba.

Given two digit numbers $x, y$ each of size $n$, we can rewrite them using a integer $m < n$ (choosing $m = \frac{n}{2}$) such that

$$x = x_1 10^m + x_2$$

$$y = y_1 10^m + y_2$$

Hence multiplying the two together and expanding,

$$xy = (x_1 10^m + x_2)(y_1 10^m + y_2)$$

$$= x_1 y_1 10^{2m} + (x_1 y_2 + x_2 y_1) 10^m + x_2 y_2$$

So in order to solve $xy$ requires computing four sub-products (i.e. $x_1 y_1,\ x_1 y_2,\ x_2 y_1,\ x_2 y_2$) , two shift operations (i.e. $10^{2m}, 10^m$ ,and two additions.

To compute the sub products (i.e. $x_1 y_1,\ x_1 y_2,\ x_2 y_1,\ x_2 y_2$) the same method is applied again to each one of the terms. Splitting a problem and recombining it again to solve it is known as Divide and Conquer. The method of solve sub problems using the same method as the original problem is known as recursion.

In its current form, the worst case running time $T(n)$, where $n$ here is the number of digits, can be expressed as

$$T(n) \leq 4T\left(\frac{n}{2}\right) + c(n)$$

When n = 2,

$$T(n) \leq c$$

$4T(\frac{n}{2})$ represents the recursive splitting of the problem into four smaller sub problems each of size $\frac{n}{2}$, and $c(n)$ represents the additions, required for merging once the four sub problems have been computed. The second formula, $T(n) \leq c$, represents case when only two numbers remain. At this point a simple multiplication is performed without need of recursion.

It can be shown that $T(n) \leq 4T\left(\frac{n}{2}\right) + c(n) = O(n^2)$. So all this resulted in the same complexity as the “classroom” multiplication.

However, Karatsuba further observed that, $x_1 y_2 + x_2 y_1$ (the middle terms) could be re-written as follows

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

Hence the product $xy$ could be fully expressed as follows

$$\boldsymbol{xy = x_1y_1 10^{2m} + \big((x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2\big)10^m + x_2y_2}$$

**Equation 2-1**

So now solving $xy$ consists of computing **three instead of four** sub products (i.e. $x_1y_1$, $x_2y_2$, $(x_1 + x_2)(y_1 + y_2)$ ), two shift operations, four additions and two subtractions. Additions, subtraction are linear operations. Shifting is constant time.

In this new form, the worst case running time $T(n)$, where $n$ here is the number of digits, can be expressed as

$$T(n) \leq 3T\left(\frac{n}{2}\right) + c(n)$$

When n = 2,

$$T(n) \leq c$$

It can be shown that $T(n) \leq 3T\left(\frac{n}{2}\right) + c(n) = O\left(n^{\log_2 3}\right) \approx O(n^{1.58})$. In this form, the method became asymptotically better than the "classroom" multiplication. It is also worth mentioning that this method only is better when two number of equally large size are being computed.

Below we will introduce two known mathematical relations which will be central to our discussions.

# Chapter 3 Square of a Difference

$$\boxed{(\boldsymbol{a + b})^2 = \boldsymbol{a^2 + 2ab + b^2}}$$

**Equation 3-1**

# Chapter 4 Square of a Sum

$$\boxed{(\boldsymbol{a - b})^2 = \boldsymbol{a^2 - 2ab + b^2}}$$

**Equation 4-1**

# Chapter 5 Computing the product of two numbers using the Square of a Difference

We can rewrite the square of a difference equation by rearranging as follows.

$$\boldsymbol{ab} = \frac{\boldsymbol{a^2 + b^2 - (a - b)^2}}{\boldsymbol{2}}$$

**Equation 5-1**

This form now allows us to compute the product of two numbers by computing:

- **SQUARES:** 3 squaring operations on integers of size up to n ,
- **ADDITIONS:** 1 addition operation on an integer of size up to 2n,
- **SUBTRACTIONS:** 2 subtractions one being in on integer of size up to n and the other up to 2n, finally,
- **DIVISION:** 1 division by two on an integer of size up to n

In this form, the worst case running time $T(n)$, where $n$ here is the number of digits, can be expressed as

$$T(n) \leq 3T\left(\frac{n}{2}\right) + c(n)$$

When $n = 1$,

$$T(n) \leq c$$

$3T(\frac{n}{2})$ represents the recursive splitting of the problem into three smaller sub problems each of size $\frac{n}{2}$, and $c(n)$ represents the additions, subtraction, and division described above. The second formula, $T(n) \leq c$, the represents case when only two numbers remain. At this point a simple square is performed without need of recursion.

It can be shown that $T(n) \leq 3T\left(\frac{n}{2}\right) + c(n) = O\left(n^{\log_2 3}\right) \approx O(n^{1.58})$. Hence the worst case running time equals that of the Karatsuba method described earlier.

The following algorithm describes the process.

```
ComputeProduct(a, b)
  aMinb ← a - b
  aSquared ← ComputeSquare(a)
  bSquared ← ComputeSquare(b)
  aMinbSquared ← ComputeSquare(aMinb)
  productAB ← DivideByTwo(aSquared + bSquared – aMinBSquared)
  return productAB
```

**Algorithm 5-1**

The complexity of the multiplication of is now reduced to complexity of computing the square of a number.
We will now discuss the computing the square of a number using the square of a difference.

# Chapter 6 Computing the Square of a Number using the Square of a Difference and Square of a Sum

Given a number $a$ each of size $n$, we can rewrite it using by choosing integer $m < n$ , preferably choosing $m = \frac{n}{2}$. Hence

$$a = a_1 10^m + a_2$$

*Examples*

Given $a = 56789$
then $n = 5$ ,choose $m = 3$ therefore $a_1 = 56$, and $a_2 = 789$

Given $a = 1234567890$
then $n = 10$ , choose $m = 5$ therefore $a_1 = 12345$, and $a_2 = 67890$

Hence the $a^2$ can be expressed as,

$$a^2 = (a_1 10^m + a_2)^2$$

Expanding using the sum of squares gives,

$$a^2 = a_1^2 10^{2m} + 2a_1 a_2 10^m + a_2^2$$

**Equation 6-1**

Rearranging the sum of differences relation $2a_1 a_2$ can be expressed as,

$$2a_1 a_2 = a_1^2 + a_2^2 - (a_1 - a_2)^2$$

We can substitute the term $2a_1 a_2$ in the equation (1) to get

$$\boldsymbol{a^2 = a_1^2 10^{2m} + (a_1^2 + a_2^2 - (a_1 - a_2)^2) 10^m + a_2^2}$$

**Equation 6-2**

Now we have arrived into expressing $a^2$ in terms three square operations $a_1^2, a_2^2, (a_1 - a_2)^2$ two shift operations $10^m, 10^{2m}$ and three additions and two subtractions. The square can then be solved recursively and applying the same derived formula.

*Example*

Given $a = 56789$
then $n = 5$ , choose m = 3, $a_1 = 56$, and $a_2 = 789$,
hence
$56789^2 = 56^2 \times 10^5 + (56^2 + 789^2 - (56 - 789)^2) \times 10^3 + 789^2$

This form now allows us to compute the square of two numbers by computing:

- **SQUARES:** 3 squaring operations on integers of size up to n / 2 ,
- **ADDITIONS:** 1 addition operation on integers of size up to 2n,
- **SUBTRACTIONS:** 2 subtractions one being in on integer of size up to n and the other up to 2n, finally,

- **SHIFTS**: 2 shifts

Addition and subtraction are linear while shifting can be done in constant time. Hence the Square remains the bottleneck. But we shall see in the next section we can use divide and conquer to further simplify the problem.

This form splits result in a worst case running time of $T(n) \leq 3T\left(\frac{n}{2}\right) + c(n) = O\left(n^{\log_2 3}\right) \approx O(n^{1.58})$. Hence the worst case running time equals that of the Karatsuba method described earlier.

Asymptotically the result looks the same. In further later discussion we will discussion we analyze whether using the sums of difference can lead to better results in $k$, part of $O(n^{1.58}) = kn^{1.58}$

But first we unfold the algorithm for computing squares using the sums of differences.

The algorithm above takes 6 parameters as input;

- result[] contains the result of squaring stored as an array of digits,
- a[] represent the number to be squared also stored as an array of digits,
- fromResult and toResult represent where in the result array the square of the number in a[] has to be stored, and
- from and to represent the number in a[1..n] that need to be squared.

```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  if from == to then
    if a[to] = 1 then
      result[toResult] = 1
    else if a[to] = 2 then
      result[toResult] = 4
    else if a[to] = 3 then
      result[toResult] = 9
    else if a[to] = 4 then
      result[fromResult] = 1
      result[toResult] = 6
    else if a[to] = 5 then
      result[fromResult] = 2
      result[toResult] = 5
    else if a[to] = 6 then
      result[fromResult] = 3
      result[toResult] = 6
    else if a[to] = 7 then
      result[fromResult] = 4
      result[toResult] = 9
    else if a[to] = 8 then
      result[fromResult] = 6
      result[toResult] = 4
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  split = Ceiling((from + to) / 2)
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  aMinB = ComputeDifference(a, from, split, to)
  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  result = ComputeSquare(result, a, splitResult, toResult, split, to)
  resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
aMinusB.Length - 1)
  result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
  return result
```

**Algorithm 6-1**

## *Example*

Given $a = 82^2$ ,

then $result$ would initially be an array of twice the size of the input. ComputeSquare would then split $a$ in half, in this case 8 and 2. As shown below,

| a | 8 | | 2 | |
|---|---|---|---|---|
| result | | | | |

$aMinB = 8 - 2 = 6$ would then be computed and separate array to computes aMinbResult would be created as follows,

| aMinb | 6 | |
|---|---|---|
| aMinResult | 3 | 6 |

The square of $8^2$ would next be computed and placed as follows:

| a | 8 | | 2 | |
|---|---|---|---|---|
| result | 6 | 4 | | |

Similarly $2^2$would be computed and placed as follows:

| a | 8 | | 2 | |
|---|---|---|---|---|
| result | 6 | 4 | 0 | 4 |

Then $6^2$ would be computed and placed as follows:

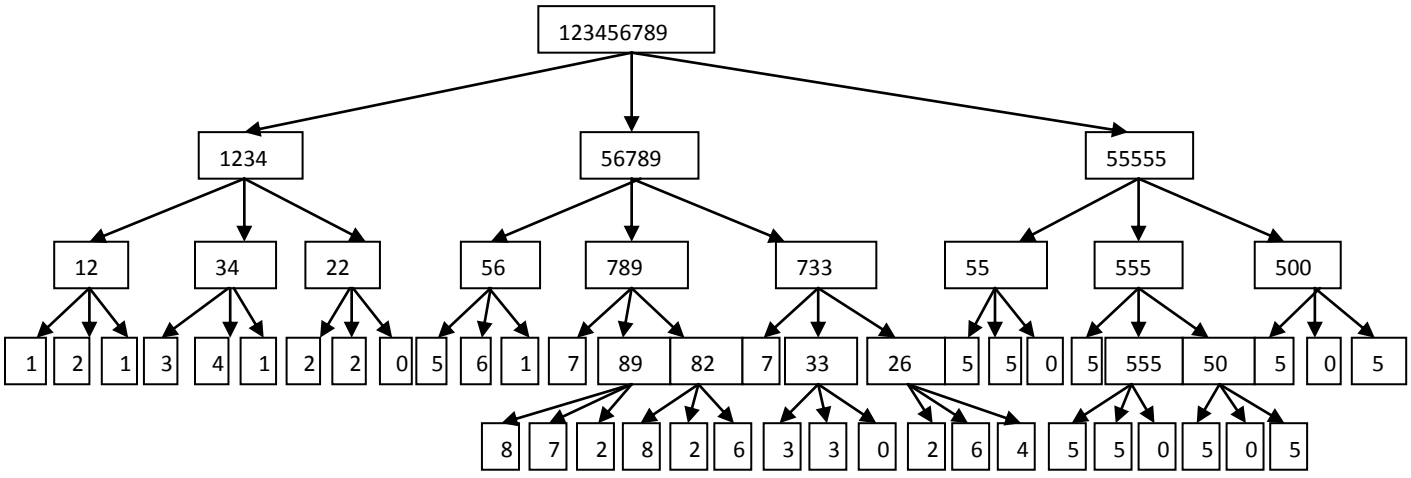
| aMinb | 6 | |
|---|---|---|
| aMinResult | 3 | 6 |

Finally, a merge $aMinResult$ into $result$ using the formula $(a_1^2 + a_2^2 - (a_1 - a_2)^2)10^m$ would be performed, yield the finally result,

| a | 8 | | 2 | |
|---|---|---|---|---|
| result | 6 | 7 | 2 | 4 |

## *Example*

Here $123456789^2$ is being computed, using the formula, three squares need to be computed. The tree below shows how this can be done. The left leaf represents the term used to compute $a_1^2$, the middle leaf represents the term used to compute $a_2^2$, and the right leaf represents the term used to compute $(a_1 - a_2)^2$.



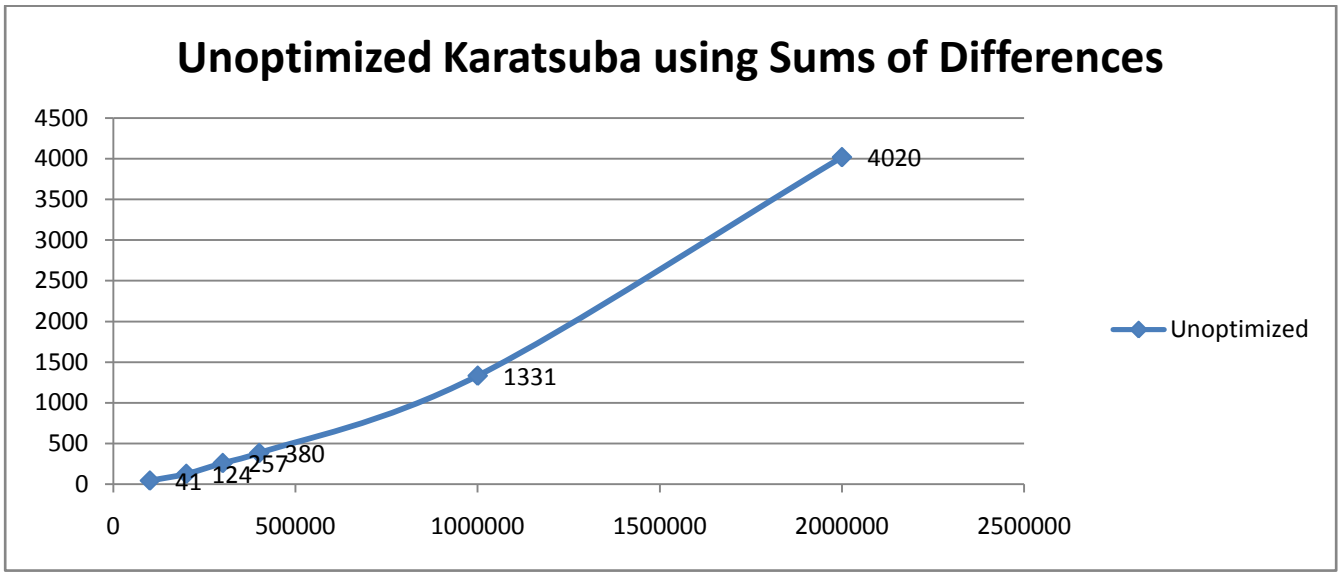
### *Empirical Results*

These results were obtained when running the actual implementation of the algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 41 |
| 200000 | 124 |
| 300000 | 257 |
| 400000 | 380 |
| 1000000 | 1331 |
| 2000000 | 4020 |

### *Conclusion*

Solving $(300000, 257)$ using $T(n) = kn^{1.58}$ when we get $k = \frac{257}{300000^{1.58}} \cong 5.7 \times 10^{-7}$, which agrees when (400000, 380). Therefore the empirical results agree with theoretical analysis in asymptotic bounds

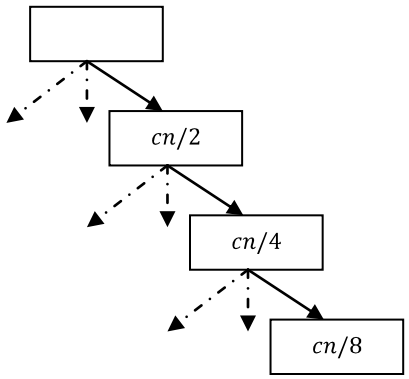


# Chapter 7 Space Complexity Optimization

As the algorithm described above illustrated, the result of the left and middle leafs of the tree are both stored in one array of size $2n$. However the right leaf which stores the result of $aMinB$ uses a separate array. In this section we will analyze the worst case space complexity of the algorithm based on this observation.

Given a large integer of size n therefore the left and middle sub-leaf would require an array of size $2n$. Computing the right sub tree would require a bit more for each recursion it makes on its right sub leafs.

To illustrate visually, below, is a diagram that shows the storage of the right sub-leaf. At each level, during the processing of the right sub-leaf a new array is created of half the size of the level above it. This continues happening until $n = 1$.




The height of the tree is $\log_2 n$, and observing that successive storage sizes are separated by a factor of $1/2$, then summing the space usage $= \frac{cn}{2} + \frac{cn}{4} + \frac{cn}{8} + \ \ldots \ = cn\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ \ldots\right) \leq cn$

Hence the space complexity is therefore worst case $O(n)$

# Chapter 8 Time Complexity Optimization

Raw as it is now the complexity of squaring is exactly like the Karatsuba. Besides being simple, does using sums of differences

$$\boxed{a^2 = a_1^2 10^{2m} + (a_1^2 + a_2^2 - (a_1 - a_2)^2)10^m + a_2^2}$$

imply better time complexity than the textbook form of the Karatsuba?

$$\boxed{xy = x_1 y_1 10^{2m} + \big((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2\big)10^m + x_2 y_2}$$

The un-optimized algorithm using the square of difference presented earlier is equivalent in many ways to the Karatsuba. So we will analyze and present various aspects that have potential of improving it whilst still keeping it simple.

## Optimizing on the number of primitive operations required

We saw that on the original Karatsuba, each level required, solving three multiplications, two shift operations, four additions and two subtractions.

In the un-optimized sum of difference method, it requires solving three square operations, two shift operation and three additions and three subtractions (the extra subtraction comes about as a result on computing $a_1 - a_2$, because some instances would require $a_2 - a_1$ instead.

The amount of primitive operations are the same for both methods, i.e. four additions and two subtractions on the Karatsuba, and three additions and three subtractions on the one using sums of differences.

We see however that the only one of the subtraction in $a_1 - a_2$ gets to be unused. Hence we can attempt an improvement, instead of computing them both all the time we can compute both when it is absolutely necessary.

### *Modified Algorithm*

```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  if from == to then
    if a[to] = 1 then
      result[toResult] = 1
    …
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  split = Ceiling((from + to) / 2)
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  aMinB = ComputeDifference(a, split, to, from, split - 1)
  if not aMinBIsCorrect
    aMinB = ComputeDifference(a, from, split - 1, split, to)
  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  result = ComputeSquare(result, a, splitResult, toResult, split, to)
  resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
aMinusB.Length - 1)
  result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
  return result
```

**Algorithm 8-1**

Here the only change is the replacement of the statement

```
aMinB = ComputeDifference(a, from, split, to)
```

with

```
aMinB = ComputeDifference(a, split, to, from, split - 1)
if not aMinBIsCorrect
  aMinB = ComputeDifference(a, from, split - 1, split, to)
```

*Examples*

Given $a = 5678$

$a_1 = 56$, and $a_2 = 78$,

The first algorithm would compute both $a_1 - a_2$ and $a_2 - a_1$ and return $a_2 - a_1$

The second would attempt compute $a_1 - a_2$ fail then attempt $a_2 - a_1$

Given $a = 7856$

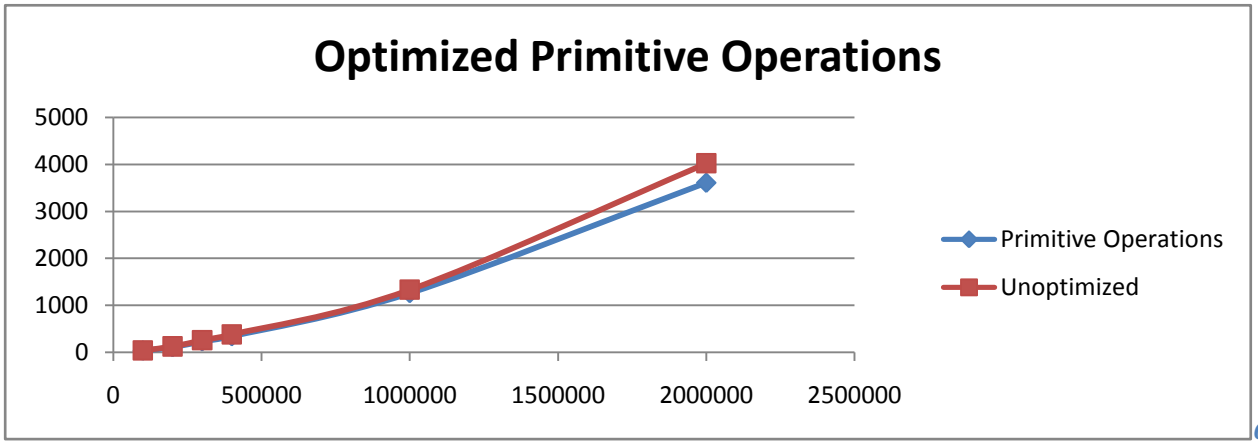
The second algorithm would need only one attempt of $a_1 - a_2$ to succeed.

### *Empirical Results*

These results were obtained when running the actual implementation of the algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 38 |
| 200000 | 114 |
| 300000 | 236 |
| 400000 | 345 |
| 1000000 | 1263 |
| 2000000 | 3603 |

The graph below presents a comparison between optimizing primitive operation and the optimized version




*Co*

### *Conclusion*

Reducing the amount of primitive reduces the time spent but not significantly within the asymptotical bounds.

## Optimizing using $2a_1a_2$

The un-optimized algorithm uses the formula $(a^2 = a_1^2 10^{2m} + (a_1^2 + a_2^2 - (a_1 - a_2)^2)10^m + a_2^2)$ to compute the square of all numbers of size $n > 1$. Is that really necessary though? The answer is no. There are cases when we can resort to more efficient alternatives.

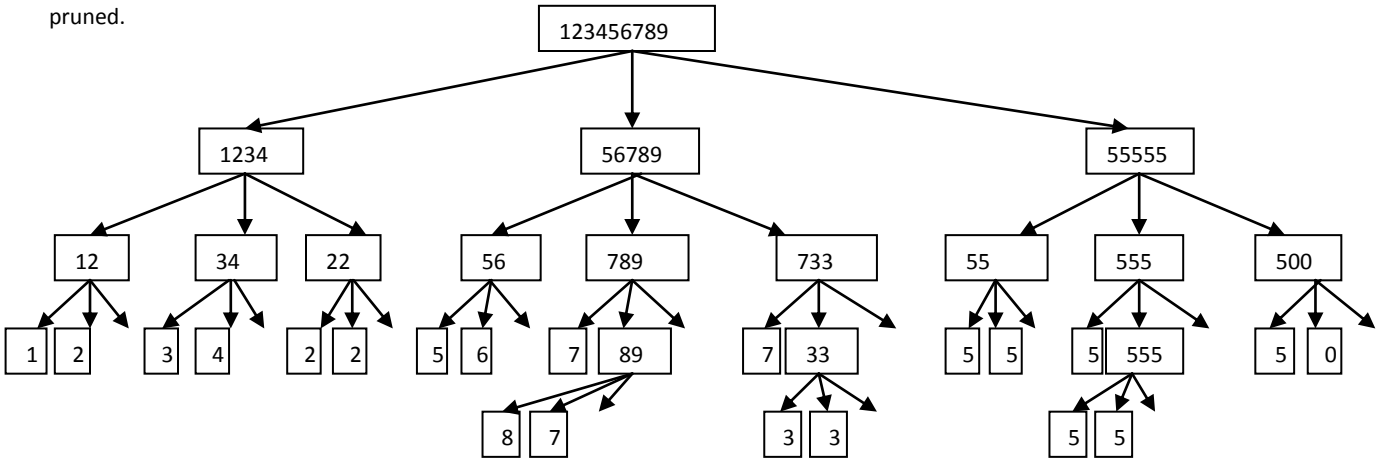
### *Pruning the tree*

The answer lies in the simpler form of the equation above i.e. $a^2 = a_1^2 10^{2m} + 2a_1a_2 10^m + a_2^2$. In this form, the product $2a_1a_2$ can be computed linearly in the cases when $a_1$, or $a_2$ have size 1. Hence instead of resorting a third square we directly compute $2a_1a_2$.

> *Example*
>
> Given $a = 82^2$ ,
>
> Currently we would have to compute $82^2 = 8^2\ 10^2 + \left(8^2 + 2^2 - (8 - 2)^2\right)10 + 2^2$
>
> But instead we can compute $82^2 = 8^2\ 10^2 + 2 \times 8 \times 2\ 10 + 2^2$

## *Example*

In chapter 6, an example of computing $123456789^2$ was given. It required computing 58 squares. Here the same tree can now be pruned at the bottom resulting in the need of computing only 37 squares (represented by each box). Note also whenever pruning occurs, the need to ComputeDifference is also eliminated. Note that the leaves that lead to nowhere are the ones that have been pruned.

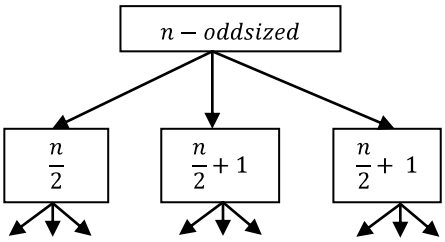


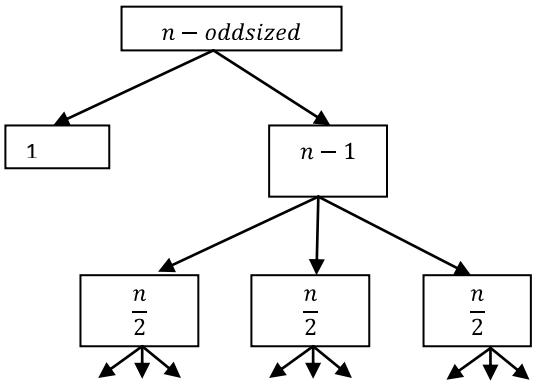

### *Balancing the tree*

Another improvement we can make it to balance the leaves. We observe in many places where split an odd sized number we obtain three numbers of unequal size, with one of being one size smaller than the other two.

> *Example*
>
> Given $a = 12345^2$ ,
>
> We choose $m = 3$, hence we obtain $a_1 = 12$, $a_2 = 345$, and $a_2 - a_1 = 333$
>
> Notice $a_1$ is one size smaller than other 2

An alternative can be to make the tree even-sized by first splitting it with the left sub-tree having being one-digit number. Then continue as always.




The worst case running time for tree in the left is $T_{left-tree}(n) = 2T\left(\frac{n}{2}+1\right) + T\left(\frac{n}{2}\right) + cn,$

The worst case running time for the right tree becomes $T_{right-tree}(n) = c + T(n-1) + cn = c + 3T\left(\frac{n}{2}\right) + c(n-1) + cn = 3T\left(\frac{n}{2}\right) + 2cn$, bearing in mind, multiplication is worse

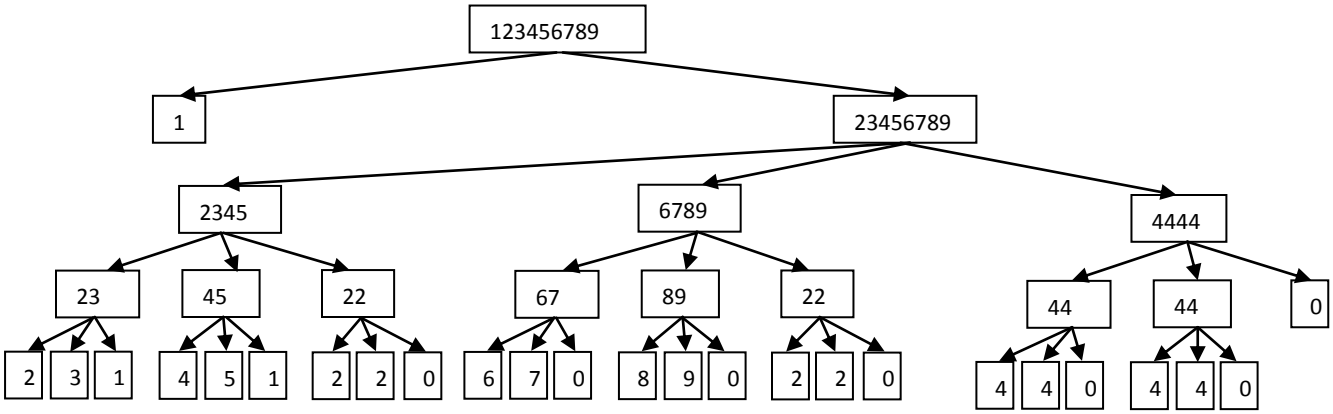
$$T_{left-tree} - T_{right-tree} = 2T\left(\frac{n}{2}+1\right) - 2T\left(\frac{n}{2}\right) - cn \geq 0 \; for \; large \; n$$

Therefore

$$T_{left-tree}(n) > T_{right-tree}(n)$$

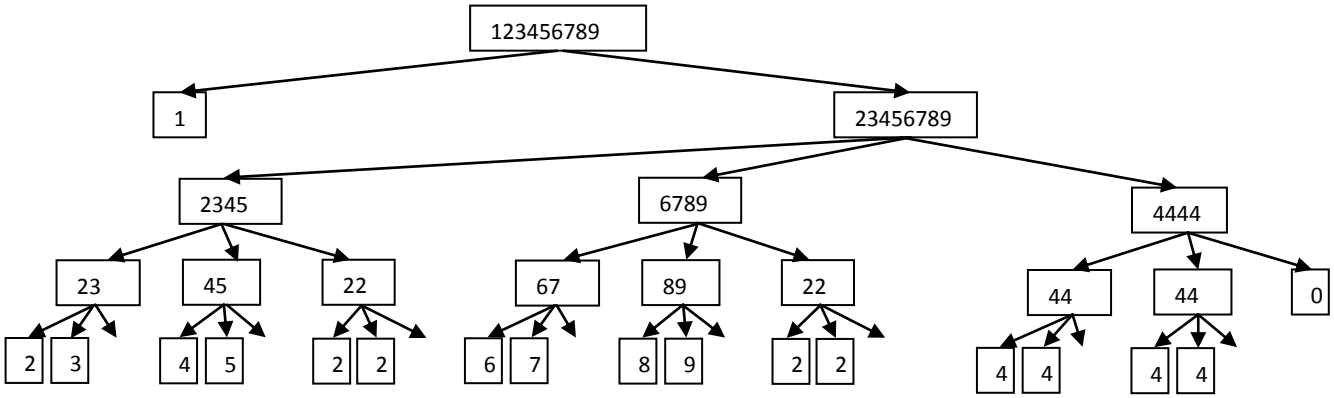
## *Example*

We now balance $123456789^2$ below. Compared to the one in chapter 6 which required computing 58 squares, this one only requires 39 squaring operations




## *Example*

We now prune and balance $123456789^2$ to obtain the combined effect below. The tree below now requires only 31 operations.




*Modified Algorithm*

```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  if from == to then
    if a[to] = 1 then
      result[toResult] = 1
    …
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  split = Ceiling((from + to) / 2)
  if (split - from) < (to - split + 1)
    split = from + 1;
  aIsOneDigit = split == from + 1
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  if not aIsOneDigit
    aMinB = ComputeDifference(a, from, split, to)
  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  result = ComputeSquare(result, a, splitResult, toResult, split, to)

  if aIsOneDigit
    result = Plus2AB(result, a, a[from], to, split, toResult - aMinB.Length)
  else
    resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
      aMinusB.Length - 1)
    result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
return result
```

**Algorithm 8-2**

- The change here is the addition of the statement which handles the balancing mechanism.

  ```
  if (split - from) < (to - split + 1)
     split = from + 1;
  ```

- Pruning is implemented and maintained by the introduction of the `aIsOneDigit = split == from + 1` branching to avoid ComputeDifference as follows

```
if not aIsOneDigit
  aMinB = ComputeDifference(a, from, split, to)
```

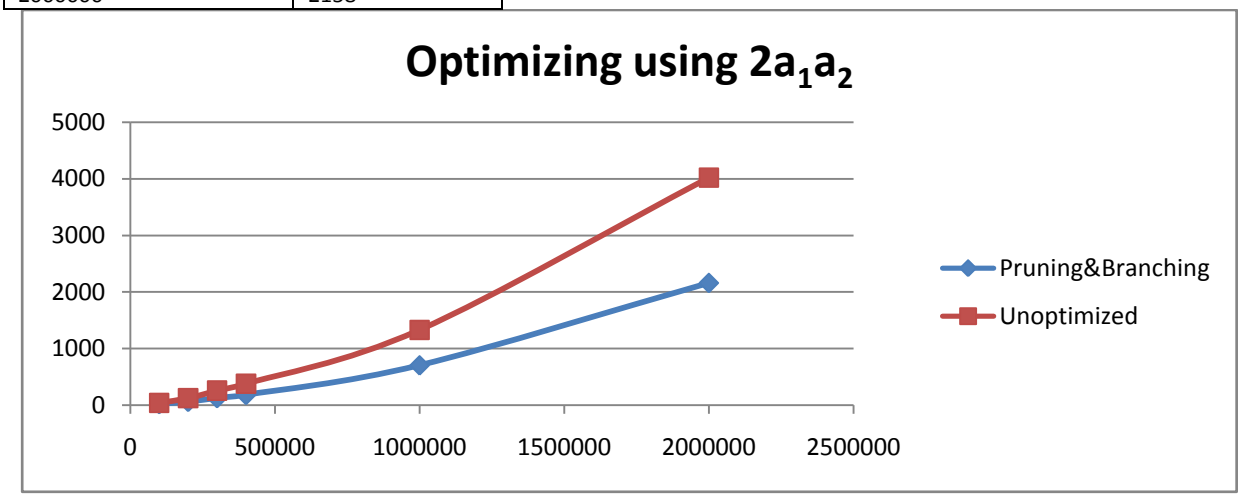
Computing Plus2AB instead squaring the aMinB and Merging through

```
if aIsOneDigit
  Plus2AB(ref result, a, a[from], to, split, toResult - aMinB.Length)
else
  resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
    aMinusB.Length - 1)
  result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
```

### *Empirical Results*

These results were obtained when running the actual implementation of the algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 21 |
| 200000 | 60 |
| 300000 | 126 |
| 400000 | 182 |
| 1000000 | 704 |
| 2000000 | 2158 |




### *Conclusion*

The gain observed was significant in altering the value of $k$.

## Optimizing using $a_2 - a_1$

We observed in the previous section that $a^2 = a_1^2 10^{2m} + 2a_1a_2 10^m + a_2^2$ could be used to compute the $a^2$ instead of the longer form, if that is, $2a_1a_2$ can be computed linearly. In section, we will that we can in fact use special cases of $a_1 - a_2$ to determine whether is possible.

### *Case* $a_1 - a_2 = 0$

The first special case is when $a_1 = a_2$. We therefore see that

$$a^2 = a_1^2 10^{2m} + 2a_1a_2 10^m + a_2^2 = a_1^2 10^{2m} + 2a_1^2 10^m + a_1^2$$

This lead to the observation that only $a_1^2$ need be computed, shift and merged to compute $a^2$.

The worst case running time becomes $T(n) = T\left(\frac{n}{2}\right) + cn$, whenever this situation occurs.

### *Case* $a_1 - a_2 = k10^p$

Suppose $a_1 - a_2 = k10^p$, where k is an integer $10 > k < 10$ and p is positive integer, then once we have computed $a_1^2$ we can linearly compute $a_2^2$, using the following relation

$$a_2^2 = (a_1 - k10^p)^2$$

Which when expanded,

$$a_2^2 = a_1^2 - 2a_1k10^p + k^2 10^{2p}$$

In this form, $a_1^2$is already known, $2a_1k$ can be computed linearly since k is a single digit integer, $k^2$ can also be computed in constant time since it is single digit square, $10^p$ and $10^{2p}$ are shift operations and thus negligible. Hence $a_2^2$ has a worst case $T(n) = 2T\left(\frac{n}{2}\right) + cn = O(n \log_2 n)$ since only two squares need be computed and merged.

*Example*

Given $a = 4444^2$ ,

We choose $m = 2$, hence we obtain $a_1 = 44$, $a_2 = 44$, and $a_2 - a_1 = 0$

Thus we only need to compute $44^2$, and the rest can be performed linearly to obtain $4444^2$

Given $a = 101131100131^2$

We choose $m = 3$, hence we obtain $a_1 = 101131$, $a_2 = 100131$, and $k = 1$, $p = 3$

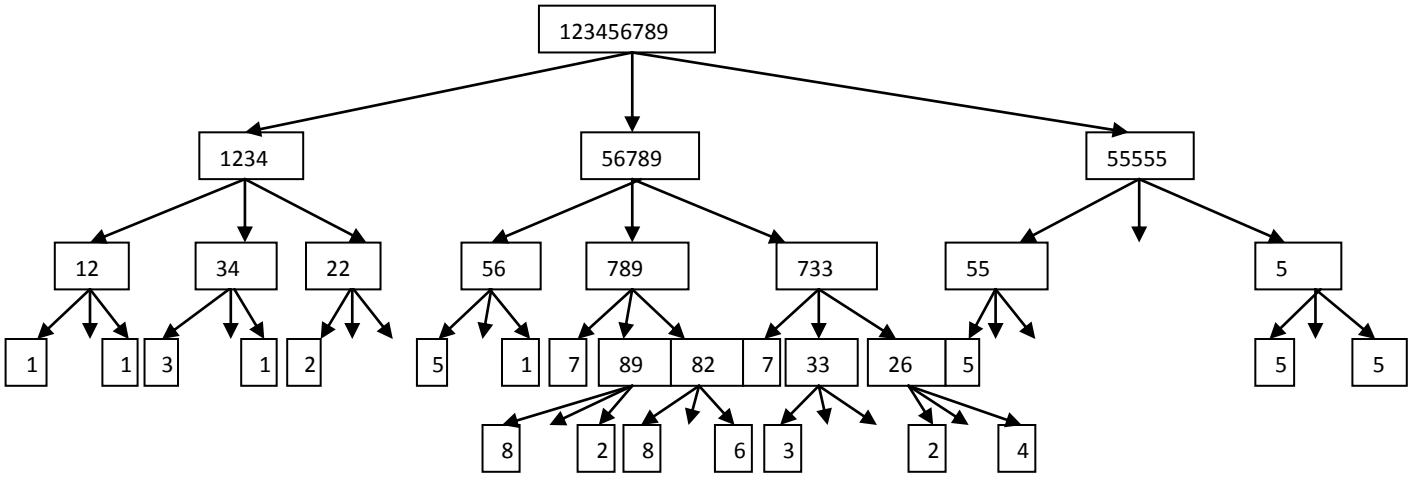
Hence

$$a_2^2 = 100131 = 101131^2 - 2 \times 1 \times 101131 \times 10^3 + 1^2 \times 10^{2\times3}$$

Hence once $101131^2$ is computed, $100131^2$ can be solved linearly.

## *Example*

In chapter 6, an example of computing $123456789^2$ was given. It required computing 58 squares. Here the same tree can now be optimized using information based from computing $a_1 - a_2$. This results in the need of computing only 35 squares (represented by each box). Note that the leaves that lead to nowhere are the ones that have been pruned.




### *Modified Algorithm*

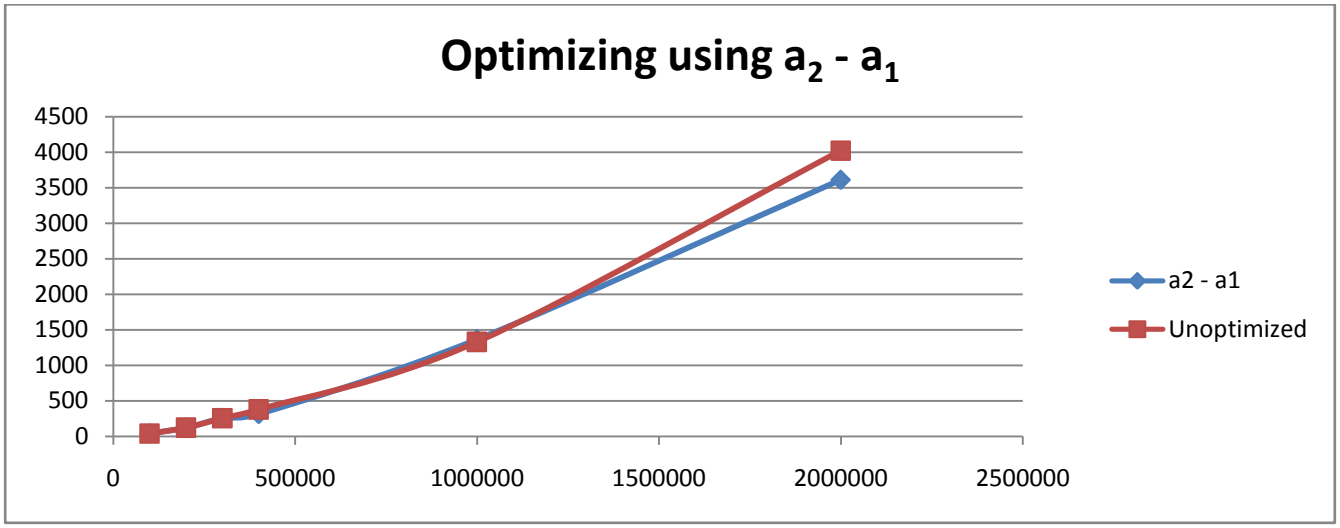
```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  if from == to then
    if a[to] = 1 then
      result[toResult] = 1
    …
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  split = Ceiling((from + to) / 2)
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  aMinB = ComputeDiff(a, from, split - 1, split, to, aMinBFrom, aMinBTo, aIsLarger)
  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  if (aMinBTo < aMinBFrom)
    CopyResultAToBAnd2AB(result, fromResult, splitResult - 1, toResult)
    return result
  else if (aMinBTo == aMinBFrom)
    result = ComputeResultAToBLinearly(result, a, fromResult, splitResult - 1, toResult,
      from, split - 1, aIsLarger, aMinB[aMinBTo], aMinB.Length - aMinBFrom - 1)
  else
    result = ComputeSquare(result, a, splitResult, toResult, split, to)

  resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
aMinusB.Length - 1)
  result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
  return result
```

**Algorithm 8-3**

### *Empirical Results*

These results were obtained when running the actual implementation of the algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 41 |
| 200000 | 124 |
| 300000 | 257 |
| 400000 | 315 |
| 1000000 | 1365 |
| 2000000 | 3609 |




### *Conclusion*

The results seem to suggest the now many number would be benefit from this optimization. Even though there better result as the number tended to be large, the gap was not significant.

## Optimizing using Memoization

Squaring using Karatsuba Method always result in the the tree sub-branching three ways everytime. Hence while squaring a large number the lower level of tree tend to become very dense and therefore a lot of number end up being re-computed.

### *Example*

Suppose we given a 1 million digit number. Using Karatsuba we split the number and half and end up with 3-half a million sized number, then split those to end-up with 9-quarter a million digit number and so on, this will continue for $\log_2 1000000 = 19$ levels. We illustrate this in table below.

| Level of Tree | Numbers to Square | Number of Digits in each Number |
|---|---|---|
| 0 | 1 | 1000000 |
| 1 | 3 | 500000 |
| 2 | 9 | 250000 |
| 3 | 27 | 125000 |
| 4 | 81 | 62500 |
| 5 | 243 | 31250 |
| 6 | 729 | 15625 |
| 7 | 2187 | 7812 |
| 8 | 6561 | 3906 |
| 9 | 19683 | 1953 |
| 10 | 59049 | 976 |
| 11 | 177147 | 488 |
| 12 | 531441 | 244 |
| 13 | 1594323 | 122 |
| 14 | 4782969 | 61 |
| 15 | 14348907 | 30 |
| 16 | 43046721 | 15 |
| 17 | 129140163 | 7 |
| 18 | 387420489 | 3 |
| 19 | 1162261467 | 1 |

At level 17, $10^8$ numbers of size 7 need to be computed, this show that from level and lower a lot of redundant computation occur.

Could we do something about this redundancy?

A solution could be to store pre-computed squares for number sizes are likely to contain redundancy, such that, whenever we encounter a number, we can just retrieve it and write it out in linear time. Storing pre-computed values that can be readily retrieved and used is what is known as Memoization.

Dictionary data structures exist, that provide functionality to store a value using a key, and retrieve in constant time with that key. However they become in-efficient when too many records are stored. We therefore use this structure to store a fairly large number and its corresponding square. For instance we can store all five digit number with their corresponding squares.

This should change our running time formula to

$$T(n) \leq 3T\left(\frac{n}{2}\right) + c(n)$$

When n = 5,

$$T(n) \leq c$$

We see the running becomes constant at a higher, but the asymptotic complexity still remains the same.

### *Modified Algorithm*

```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  if from == to then
    if a[to] = 1 then
      result[toResult] = 1
    …
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  if (to – from <= 5)
    Key = GenerateKeyFromArray(a[], from, to)

    if MemoizedSquares.TryGetValue(Key, squareResult)
      for i = squareResult.Length – 1; i >= 0; i–
        result[toResult] = squareResult[i]
        toResult–
      return result

  split = Ceiling((from + to) / 2)
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  aMinB = ComputeDifference(a, from, split, to)
  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  result = ComputeSquare(result, a, splitResult, toResult, split, to)
  resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
aMinusB.Length - 1)
  result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
  return result
```

**Algorithm 8-4**

The change to the algorithm is the addition of the lines

```
  if (to – from <= 5)
    Key = GenerateKeyFromArray(a[], from, to)

    if MemoizedSquares.TryGetValue(Key, squareResult)
      for i = squareResult.Length – 1; i >= 0; i–
        result[toResult] = squareResult[i]
        toResult–
      return result
```

Here the algorithm assume square for number upto 5 digit have be memorized in a dictionary called `MemoizedSquares`. To retrieve the square a key has to be constructed using

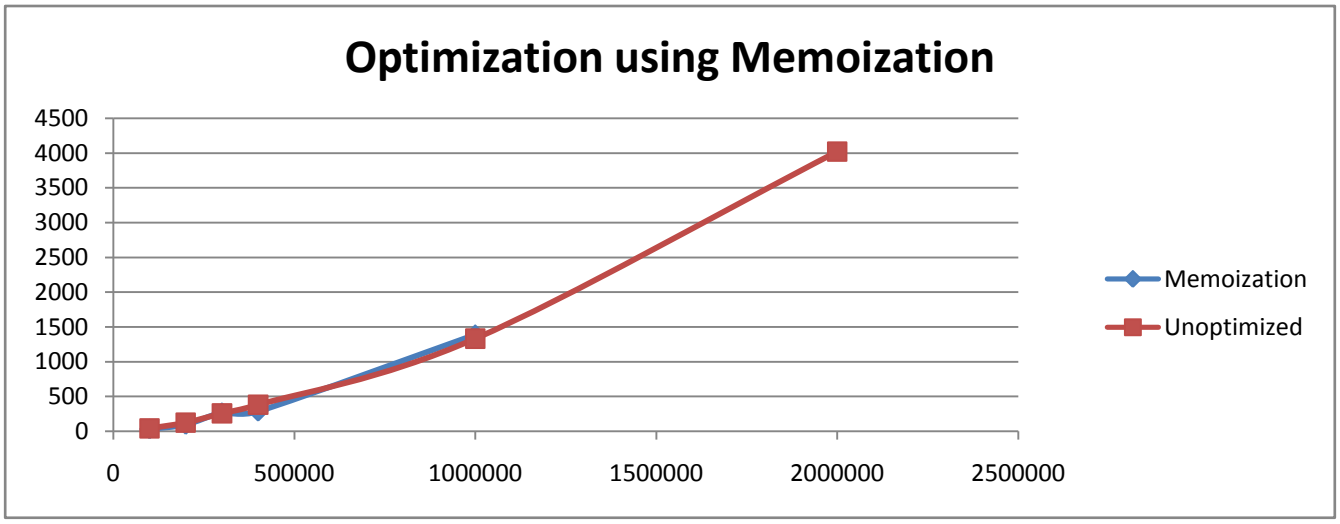
```
Key = GenerateKeyFromArray(a[], from, to)
```

Thereafter the square is retrieved in constant time from the dictionary and written back linearly into result.

### *Empirical Results*

These results were obtained when running the actual implementation of the combined algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 31 |
| 200000 | 94 |
| 300000 | 265 |
| 400000 | 284 |
| 1000000 | 1389 |
| 2000000 | |



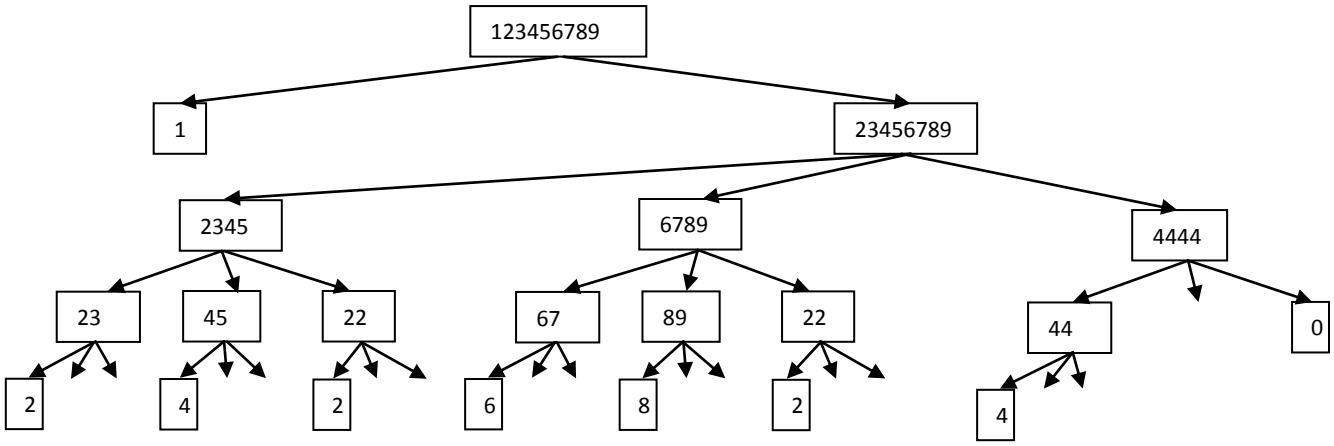

### *Conclusion*

No gain was seen in the empirical result even though the theoretical analysis seemed very promising. The loss in gain can be attributed to the implementation of the dictionary. Even thought retrieval is claimed to be in constant time, it may still be that the constant is still too high. Also generation of the key used for retrieval may be too costly. In our implementation, the key was a string representation of the number. To overcome these cost the solution may be to store even large square, but then again, the density of redundancy also drops.

## Combined Effect

In this section we present the final, algorithm, which consists of putting together all optimization that we discussed earlier into one algorithm. We exclude memorization, due to the poor results that were obtained from it.

To illustrate the effect of combining all the methods, the tree below represents computing $123456789^2$




We notice that now 21 numbers need to be squared. This compared to 58 squares on the unoptimized looks promising

### *Modified Algorithm*

```
ComputeSquare(result[], a[], fromResult, toResult, from, to)
  while (a[from] = 0) and (from < to)
   from++
  while (a[to] = 0) and (from < to)
   to—

  if from == to
    if a[to] = 1 then
      result[toResult] = 1
    …
    else if a[to] = 9 then
      result[fromResult] = 8
      result[toResult] = 1
    return result

  split = Ceiling((from + to) / 2)
  if (split - from) < (to - split + 1)
    split = from + 1;
  aIsOneDigit = split == from + 1
  splitResult = toResult - 2 * (to - split + 1) + 1

  int[] aMinusB = new int[to - split + 1]
  int[] resultAMinB = new int[2 * aMinB.Length]

  if not aIsOneDigit
    aMinB = ComputeDifference(a, split, to, from, split – 1,
        aMinBFrom, aMinBTo, aIsLarger)
    if not aMinBIsCorrect(aMinB)
      aMinB = ComputeDifference(a, from, split - 1, split, to,
          aMinBFrom, aMinBTo, aIsLarger)

  result = ComputeSquare(result, a, fromResult, splitResult - 1, from, split - 1)
  if aMinBTo < aMinBFrom
    CopyResultAToBAnd2AB(result, fromResult, splitResult - 1, toResult)
    return result

  if (aMinBTo == aMinBFrom) and not aIsOneDigit
    result = ComputeResultAToBLinearly(result, a, fromResult, splitResult - 1, toResult,
        from, split - 1, aIsLarger, aMinB[aMinBTo], aMinB.Length - aMinBFrom - 1)
  else
    result = ComputeSquare(result, a, splitResult, toResult, split, to)

  if aIsOneDigit
    result = Plus2AB(result, a, a[from], to, split, toResult - aMinB.Length)
  else
    resultAMinB = ComputeSquare(resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
        aMinusB.Length - 1)
    result = Merge2AB(result, resultAMinB, fromResult, splitResult, toResult)
  return result
```

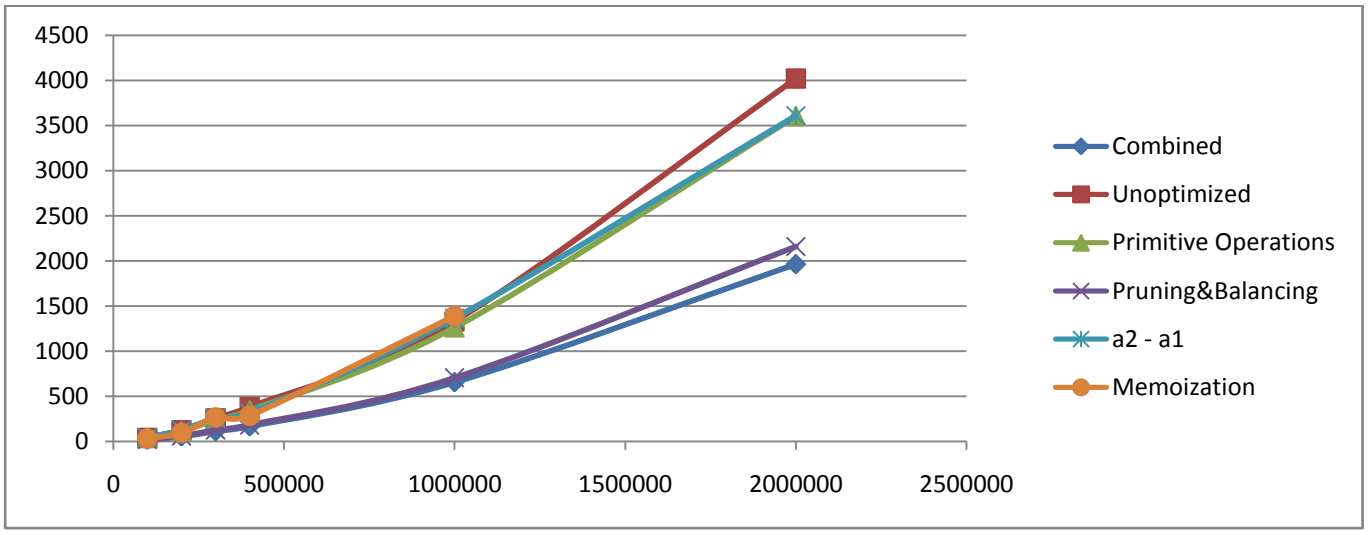
**Algorithm 8-5**

### *Empirical Results*

These results were obtained when running the actual implementation of the combined algorithm. The numbers producing the results were randomly generated.

| Size of number to square | Result (seconds) |
|---|---|
| 100000 | 18 |
| 200000 | 55 |
| 300000 | 113 |
| 400000 | 167 |
| 1000000 | 656 |
| 2000000 | 1963 |

The graph below shows running time of the all methods that were implemented. The combined method is also included for comparison. Also note that the combined does not include memorization.




### *Conclusion*

The combined effect was clearly much better than the un-optimized version of the algorithm. Its worst case running still shows a trend in $O(n^{1.58})$ with its constant very much improved. Pruning and Balancing were major contributors in lowering the constant $k$. Memoization is the only method that excluded in the combined effect.

# Chapter 9 Conclusion

The goal of this project was to demonstrate that multiplying number can also be done efficiently using the Karatsuba Algorithm with Sums of Differences. The results obtained from attempting to optimize squaring based on the properties of Sums of Difference ranged from insignificant (primitive and $a_2 - a_1$), good (branching and pruning), and worse (memorization).

Encouragingly, the methods that improved the algorithm were also very simple, keeping the algorithm still simple and easy to understand.

The positive results also seemed to reinforce the idea that, since squares have powerful properties, it seems promising to investigate for more of those that may improve the worst case running time even more significantly in the area of Large Integer Multiplication.

# Chapter 10 Appendix

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DodomaAlgorithm
{
  using System;
  using System.Collections.Generic;
  using System.Text;
  using System.Collections;

  class Program
  {
    static Dictionary<String, byte[]> squares = new Dictionary<String, byte[]>();
    static int counter = 0;

    static void Main(string[] args)
    {
       // This commented out part is used for the memozation optimizition
       /*
       for (Int64 i = 100000; i < 1000000; i++)
       {
         int inputPositionA = 0;
         StringBuilder inputA = new StringBuilder(i.ToString());
         byte[] numberA = new byte[inputA.Length];
         for (inputPositionA = 0; inputPositionA < inputA.Length; inputPositionA++)
         {
           numberA[inputPositionA] = Convert.ToByte(inputA.ToString(inputPositionA,
               1));
         }
         int count = 0;
         squares.Add(i.ToString(), Integer_BinarySquare(numberA, ref inputA, ref
           count));
       }
       */
      // ComputeSquares(new StringBuilder("326926509420845"));
      // The following part generates a random number and the call ComputesSquares
      Random randomNumber = new Random();
      counter = 0;
      //squares.Clear();

      for (int j = 0; j < 1; j++)
      {
        StringBuilder largeRandomNumber = new StringBuilder();
        for (int i = 0; i < 40000; i++)
        {
          int value = randomNumber.Next(11111, 99999);
          largeRandomNumber.Append(value);
        }
        ComputeSquares(largeRandomNumber);
      }
      //ComputeFactors1(new StringBuilder("56784487"));
      Console.Read();
    }

    static void ComputeSquares(StringBuilder Input)
    {
```

```
  int inputPosition = 0;
  byte[] numberA = new byte[Input.Length];
  inputPosition++;
  for (inputPosition = 0; inputPosition < Input.Length; inputPosition++)
  {
    numberA[inputPosition] = Convert.ToByte(Input.ToString(inputPosition, 1));
  }

  DateTime startTime = DateTime.Now;
  Console.WriteLine(startTime);

  int count = 0;
  byte[] a = Integer_BinarySquare(numberA, ref Input, ref count);

  counter = count;
  DateTime stopTime = DateTime.Now;
  Console.WriteLine(stopTime);
  Console.WriteLine("operations =" + count + ";");
  TimeSpan duration = stopTime - startTime;
  Console.WriteLine(inputPosition);
  Console.WriteLine(duration);
}

static byte[] Integer_BinarySquare(byte[] a, ref StringBuilder aString, ref int count)
{
  byte[] result = new byte[2 * a.Length];
  byte[] input = new byte[a.Length];
  count = 0;
  ComputeSquaresBinaryMethod(ref result, a, 0, 2 * a.Length - 1, 0, a.Length - 1, ref
    count);
  return (result);
}

static void ComputeSquaresBinaryMethod(ref byte[] result, byte[] a, int fromResult, int
  toResult, int from, int to, ref int count)
{
  // This is the CORE PART of the algoritm to compute squares.

  while ((a[from] == 0) && (from < to))
  {
    from++;
    fromResult = fromResult + 2;
  }
  while ((a[to] == 0) && (from < to))
  {
    to--;
    toResult = toResult - 2;
  }
  if (to < from)
    return;
  if (from == to)
  {
    switch (a[to])
    {
      case 0:
        break;
      case 1:
        result[toResult] = 1;
        break;
      case 2:
```

```
      result[toResult] = 4;
      break;
    case 3:
      result[toResult] = 9;
      break;
    case 4:
      result[fromResult] = 1;
      result[toResult] = 6;
      break;
    case 5:
      result[fromResult] = 2;
      result[toResult] = 5;
      break;
    case 6:
      result[fromResult] = 3;
      result[toResult] = 6;
      break;
    case 7:
      result[fromResult] = 4;
      result[toResult] = 9;
      break;
    case 8:
      result[fromResult] = 6;
      result[toResult] = 4;
      break;
    case 9:
      result[fromResult] = 8;
      result[toResult] = 1;
      break;
  }
  //count++;
  return;
}
//  This part is can be commented out if memoization has to be used.
 /*
 if (to - from == 5)
 {
   StringBuilder sq = new StringBuilder();

   for (int i = from; i <= to; i++)
   {
     sq.Append(a[i]);
   }
   byte[] square = null;
   if (squares.TryGetValue(sq.ToString(), out square))
   {
     for (int i = square.Length - 1; i >= 0; i--)
     {
       result[toResult] = square[i];
       toResult--;
     }
     return;
   }
 }
 */
int split = (int)Math.Ceiling((double)(from + to) / 2);
int toSplitPlus1 = to - split + 1;
if ((split - from) < toSplitPlus1)
```

```
  {
    split = from + 1;
    toSplitPlus1 = to - split + 1;
  }
  bool aIsOneDigit = split == from + 1;
  int splitResult = toResult - 2 * toSplitPlus1 + 1;

  int twoABIndex = toResult - toSplitPlus1;
  int toA = split - 1;
  int toResultA = splitResult - 1;
  byte[] aMinB = null;

  //new byte[to - split + 1];
  int aMinBFrom = 0;
  int aMinBTo = 0;
  bool aIsLarger = true;
  if (!aIsOneDigit)
  {
    aMinB = ComputeDiff(a, split, to, from, toA, out aMinBFrom, out aMinBTo, ref
        aIsLarger /* ref InputAMinB,*/ /* ref count */);
    aIsLarger = !aIsLarger;
    if (aMinB == null)
    {
      aMinB = ComputeDiff(a, from, toA, split, to, out aMinBFrom, out aMinBTo, ref
          aIsLarger /* ref InputAMinB,*/ /* ref count */);
    }
  }
  ComputeSquaresBinaryMethod(ref result, a, fromResult, toResultA, from, toA, /* ref
    InputA,*/ ref count);


  if (aMinBTo < aMinBFrom)
  {
    CopyResultAToBAnd2AB(ref result, fromResult, toResultA, toResult, twoABIndex, ref
      count);
    return;
  }

  if (!aIsOneDigit && (aMinBTo == aMinBFrom))
  {
    ComputeResultAToBLinearly(ref result, a, fromResult, toResultA, toResult, from, toA,
      aIsLarger, aMinB[aMinBTo], aMinB.Length - aMinBFrom - 1, ref count);
  }
  else
  {
    ComputeSquaresBinaryMethod(ref result, a, splitResult, toResult, split, to, ref
      count);
  }

  if (aIsOneDigit)
    Plus2AB(ref result, a, a[from], to, split, twoABIndex, ref count);
  else
  {
    byte[] resultAMinB = new byte[2 * aMinB.Length];
    ComputeSquaresBinaryMethod(ref resultAMinB, aMinB, 0, resultAMinB.Length - 1, 0,
      aMinB.Length - 1, ref count);
    Merge(ref result, resultAMinB, fromResult, splitResult, toResult, twoABIndex, ref
      count);
  }
}
```

```
static void CopyResultAToB(ref byte[] result, int fromResultA, int toResultA, int
  toResultB, ref int count)
{
  // count++;
  int indexBForASq = toResultB;
  for (int j = toResultA; j >= fromResultA; j--)
  {
    result[indexBForASq] = result[j];
    indexBForASq--;
  }
}

static void CopyResultAToBAnd2AB(ref byte[] result, int fromResultA, int toResultA, int
  toResultB, int twoABIndex, ref int count)
{
  // count++;
  int subAddB = 0;
  int subAdd2AB = 0;
  int remB = 0;
  int rem2AB = 0;

  while (true)
  {
    subAddB = result[toResultB] + result[toResultA] + remB;
    subAdd2AB = result[twoABIndex] + result[toResultA] * 2 + rem2AB;
    remB = 0;
    rem2AB = 0;

    if (subAddB > 9)
    {
      subAddB = subAddB - 10;
      remB = 1;
    }
    if (subAdd2AB > 9)
    {
      if (subAdd2AB > 19)
      {
        subAdd2AB = subAdd2AB - 20;
        rem2AB = 2;
      }
      else if (subAdd2AB > 9)
      {
        subAdd2AB = subAdd2AB - 10;
        rem2AB = 1;
      }
    }
    result[toResultB] = (byte)subAddB;
    result[twoABIndex] = (byte)subAdd2AB;

    if (fromResultA == toResultA)
      break;
    twoABIndex--;
    toResultB--;
    toResultA--;
  }

  while (rem2AB != 0)
  {
    twoABIndex--;
```

```
      subAdd2AB = result[twoABIndex] + rem2AB;
      rem2AB = 0;
      if (subAdd2AB > 9)
      {
        subAdd2AB = subAdd - 10;
        rem2AB = 1;
      }
      result[twoABIndex] = (byte)subAdd2AB;
    }
    while (remB != 0)
    {
      toResultB--;
      subAddB = result[toResultB] + remB;
      remB = 0;
      if (subAddB > 9)
      {
        subAddB = subAdd - 10;
        remB = 1;
      }
      result[toResultB] = (byte)subAddB;
    }
  }

  static void ComputeResultAToBLinearly(ref byte[] result, byte[] a, int fromResultA, int
    toResultA, int toResultB, int fromA, int toA, bool aIsLarger, int k, int offset, ref
    int count)
  {
    //count++;
    int indexBForASq = toResultB - 2 * offset;
    switch (k)
    {
      case 1:
        result[indexBForASq] = 1;
        break;
      case 2:
        result[indexBForASq] = 4;
        break;
      case 3:
        result[indexBForASq] = 9;
        break;
      case 4:
        result[indexBForASq - 1] = 1;
        result[indexBForASq] = 6;
        break;
      case 5:
        result[indexBForASq - 1] = 2;
        result[indexBForASq] = 5;
        break;
      case 6:
        result[indexBForASq - 1] = 3;
        result[indexBForASq] = 6;
        break;
      case 7:
        result[indexBForASq - 1] = 4;
        result[indexBForASq] = 9;
        break;
      case 8:
        result[indexBForASq - 1] = 6;
        result[indexBForASq] = 4;
        break;
```

```
    case 9:
      result[indexBForASq - 1] = 8;
      result[indexBForASq] = 1;
      break;
  }

  int rem = 0;
  int subAdd = 0;
  indexBForASq = toResultB;
  int startAt = toResultA;
  if (fromResultA > toResultA)
    fromResultA = fromResultA + 0;
  while (true)
  {
    if ((result[indexBForASq] == 0) && (rem == 0))
    {
      result[indexBForASq] = result[toResultA];
    }
    else
    {
      subAdd = result[indexBForASq] + result[toResultA] + rem;
      rem = 0;
      if (subAdd > 19)
      {
        subAdd = subAdd - 20;
        rem = 2;
      }
      else if (subAdd > 9)
      {
        subAdd = subAdd - 10;
        rem = 1;
      }
      result[indexBForASq] = (byte)subAdd;
    }
    if (toResultA == fromResultA)
      break;
    toResultA--;
    indexBForASq--;
  }
  while (rem != 0)
  {
    indexBForASq--;
    subAdd = result[indexBForASq] + rem;
    rem = 0;
    if (subAdd > 9)
    {
      subAdd = subAdd - 10;
      rem = 1;
    }
    result[indexBForASq] = (byte)subAdd;
  }
  //count++;
  indexBForASq = toResultB - offset;
  if (aIsLarger)
  {
    k = -k;
    Minus2AB(ref result, a, k, toA, fromA, indexBForASq, ref count);
  }
  else
  {
```

```
    Plus2AB(ref result, a, k, toA, fromA, indexBForASq, ref count);
  }
}

static void Merge(ref byte[] result, byte[] resultC, int fromA, int fromB, int toB, int
  fromResultIndex, ref int count)
{
  int aIndex = fromB - 1;
  int bIndex = toB;
  int cIndex = resultC.Length - 1;
  int subResultIndex = fromResultIndex - fromB;
  int resultIndex = fromResultIndex;
  int subAdd = 0;
  int rem = 0;
  int[] subResultB = new int[fromResultIndex - fromB + 1];
  bool useSubResultB = false;

  for (int i = resultC.Length; i > 0; i--)
  {
    //count++;
    if (!useSubResultB)
    {
      useSubResultB = bIndex == fromResultIndex;
      if (useSubResultB)
        bIndex = subResultB.Length - 1;
    }
    if (subResultIndex >= 0)
      subResultB[subResultIndex] = result[resultIndex];

    if (aIndex < fromA)
    {
      if (useSubResultB)
        subAdd = result[resultIndex] + subResultB[bIndex] - resultC[cIndex] + rem;
      else
        subAdd = result[resultIndex] + result[bIndex] - resultC[cIndex] + rem;
    }
    else
    {
      if (useSubResultB)
        subAdd = result[resultIndex] + result[aIndex] + subResultB[bIndex] -
          resultC[cIndex] + rem;
      else
        subAdd = result[resultIndex] + result[aIndex] + result[bIndex] - resultC[cIndex]
          + rem;
    }
    rem = 0;
    if (subAdd > 19)
    {
      rem = 2;
      subAdd = subAdd - 20;
    }
    else if (subAdd > 9)
    {
      rem = 1;
      subAdd = subAdd - 10;
    }
    else if (subAdd < 0)
    {
      rem = -1;
      subAdd = subAdd + 10;
```

```
    }

    result[resultIndex] = (byte)subAdd;
    aIndex--;
    bIndex--;
    cIndex--;
    resultIndex--;
    subResultIndex--;
  }
  while (rem != 0)
  {
    //count++;
    subAdd = result[resultIndex] + rem;
    rem = 0;
    if (rem > 10)
    {
      rem = 1;
      subAdd = subAdd - 10;
    }
    if (rem < 0)
    {
      rem = -1;
      subAdd = subAdd + 10;
    }
    result[resultIndex] = (byte)subAdd;
    resultIndex--;
  }
}

static byte[] ComputeDiff(byte[] a, int fromA, int toA, int fromB, int toB, out int
  aMinBFrom, out int aMinBTo, ref bool aIsLarger /* ref int count */)
{
  byte[] aMinB = new byte[(toA - fromA - (toB - fromB) > 0 ? toA - fromA + 1 : toB -
    fromB + 1)];
  int indexAminB = aMinB.Length - 1;
  int subMin = 0;
  aMinBTo = aMinB.Length - 1;
  aMinBFrom = 0;

  aIsLarger = true;
  int rem = 0;
  while (true)
  {
    subMin = aMinB[indexAminB];
    bool useA = toA >= fromA;
    bool useB = toB >= fromB;
    if (useA && useB)
    {
      subMin = subMin + a[toA] - a[toB] + rem;
    }
    else if (useA)
    {
      subMin = subMin + a[toA] + rem;
    }
    else if (useB)
    {
      subMin = subMin - a[toB] + rem;
    }
    rem = 0;
    if ((subMin < 0) && (indexAminB >= 0))
```

```
    {
      subMin = subMin + 10;
      rem = -1;
    }

    aMinB[indexAminB] = (byte)subMin;

    if (indexAminB == 0)
    {
      if (rem != 0)
        aMinB = null;
      break;
    }
    toA--;
    toB--;
    indexAminB--;
  }
  if (aMinB == null)
    return aMinB;
  while ((aMinB[aMinBFrom] == 0) && (aMinBFrom < aMinBTo))
  {
    aMinBFrom++;
    //count++;
  }
  while ((aMinB[aMinBTo] == 0) && (aMinBFrom < aMinBTo))
  {
    aMinBTo--;
    //count++;
  }
  if ((aMinBFrom == aMinBTo) && (aMinB[aMinBFrom] == 0))
  {
    aMinBTo = -1;
  }
  return aMinB;
}

static void Plus2AB(ref byte[] result, byte[] a, int valueA, int toB, int fromB, int
  fromResultIndex, ref int count)
{
  int resultIndex = fromResultIndex;
  int remainder = 0;
  int indexB = toB;
  int subMultiply = 0;
  if (valueA == 0)
    return;
  valueA = 2 * valueA;
  //if (fromA == split - 1)
  {
    while (true)
    {
      // count++;
      if (indexB >= fromB)
      {
        subMultiply = valueA * a[indexB] + remainder + result[resultIndex];
      }
      else
        subMultiply = remainder + result[resultIndex];

      remainder = 0;
      if (subMultiply < 100)
```

```
{
  if (subMultiply < 10)
  {
    //result[resultIndex] = subMultiply;
    remainder = 0;
  }
  else if (subMultiply < 20)
  {
    subMultiply = subMultiply - 10;
    remainder = 1;
  }
  else if (subMultiply < 30)
  {
    subMultiply = subMultiply - 20;
    remainder = 2;
  }
  else if (subMultiply < 40)
  {
    subMultiply = subMultiply - 30;
    remainder = 3;
  }
  else if (subMultiply < 50)
  {
    subMultiply = subMultiply - 40;
    remainder = 4;
  }
  else if (subMultiply < 60)
  {
    subMultiply = subMultiply - 50;
    remainder = 5;
  }
  else if (subMultiply < 70)
  {
    subMultiply = subMultiply - 60;
    remainder = 6;
  }
  else if (subMultiply < 80)
  {
    subMultiply = subMultiply - 70;
    remainder = 7;
  }
  else if (subMultiply < 90)
  {
    subMultiply = subMultiply - 80;
    remainder = 8;
  }
  else if (subMultiply < 100)
  {
    subMultiply = subMultiply - 90;
    remainder = 9;
  }
}
else if (subMultiply < 110)
{
  subMultiply = subMultiply - 100;
  remainder = 10;
}
else if (subMultiply < 120)
{
  subMultiply = subMultiply - 110;
```

```
        remainder = 11;
      }
      else if (subMultiply < 130)
      {
        subMultiply = subMultiply - 120;
        remainder = 12;
      }
      else if (subMultiply < 140)
      {
        subMultiply = subMultiply - 130;
        remainder = 13;
      }
      else if (subMultiply < 150)
      {
        subMultiply = subMultiply - 140;
        remainder = 14;
      }
      else if (subMultiply < 160)
      {
        subMultiply = subMultiply - 150;
        remainder = 15;
      }
      else if (subMultiply < 170)
      {
        subMultiply = subMultiply - 160;
        remainder = 16;
      }
      else if (subMultiply < 180)
      {
        subMultiply = subMultiply - 170;
        remainder = 17;
      }
      else if (subMultiply < 190)
      {
        subMultiply = subMultiply - 180;
        remainder = 18;
      }
      else
      {
        subMultiply = subMultiply - 190;
        remainder = 19;
      }
      result[resultIndex] = (byte)subMultiply;
      indexB--;
      resultIndex--;
      if ((indexB < fromB) && (remainder == 0))
        break;
    }
  }
}

static void Minus2AB(ref byte[] result, byte[] a, int valueA, int toB, int fromB, int
  fromResultIndex, ref int count)
{
  int resultIndex = fromResultIndex;
  int remainder = 0;
  int indexB = toB;
  int subMultiply = 0;
  valueA = 2 * valueA;
```

```
while (true)
{
  // count++;
  if (indexB >= fromB)
  {
    subMultiply = valueA * a[indexB] + remainder + result[resultIndex];
  }
  else
    subMultiply = remainder + result[resultIndex];

  remainder = 0;
  if (subMultiply >= -10)
  {
    subMultiply = subMultiply + 10;
    remainder = -1;
  }
  else if (subMultiply >= -20)
  {
    subMultiply = subMultiply + 20;
    remainder = -2;
  }
  else if (subMultiply >= -30)
  {
    subMultiply = subMultiply + 30;
    remainder = -3;
  }
  else if (subMultiply >= -40)
  {
    subMultiply = subMultiply + 40;
    remainder = -4;
  }
  else if (subMultiply >= -50)
  {
    subMultiply = subMultiply + 50;
    remainder = -5;
  }
  else if (subMultiply >= -60)
  {
    subMultiply = subMultiply + 60;
    remainder = -6;
  }
  else if (subMultiply >= -70)
  {
    subMultiply = subMultiply + 70;
    remainder = -7;
  }
  else if (subMultiply >= -80)
  {
    subMultiply = subMultiply + 80;
    remainder = -8;
  }
  else if (subMultiply >= -90)
  {
    subMultiply = subMultiply + 90;
    remainder = -9;
  }
  else if (subMultiply >= -100)
  {
    subMultiply = subMultiply + 100;
    remainder = -10;
```

```
    }
    else if (subMultiply >= -110)
    {
      subMultiply = subMultiply + 110;
      remainder = -11;
    }
    else if (subMultiply >= -120)
    {
      subMultiply = subMultiply + 120;
      remainder = -12;
    }
    else if (subMultiply >= -130)
    {
      subMultiply = subMultiply + 130;
      remainder = -13;
    }
    else if (subMultiply >= -140)
    {
      subMultiply = subMultiply + 140;
      remainder = -14;
    }
    else if (subMultiply >= -150)
    {
      subMultiply = subMultiply + 150;
      remainder = -15;
    }
    else if (subMultiply >= -160)
    {
      subMultiply = subMultiply + 160;
      remainder = -16;
    }
    else if (subMultiply >= -170)
    {
      subMultiply = subMultiply + 170;
      remainder = -17;
    }
    else if (subMultiply >= -180)
    {
      subMultiply = subMultiply + 180;
      remainder = -18;
    }
    else if (subMultiply >= -190)
    {
      subMultiply = subMultiply + 190;
      remainder = -19;
    }

    result[resultIndex] = (byte)subMultiply;
    indexB--;
    resultIndex--;
    if ((indexB < fromB) && (remainder == 0))
      break;
  }
}


static byte[] ComputeProduct(StringBuilder InputA, StringBuilder InputB)
{
  if (InputA.Length > InputB.Length)
  {
```

```
  StringBuilder inputC = InputA;
  InputA = InputB;
  InputB = inputC;
}

int inputPositionA = 0;
byte[] numberA = new byte[InputA.Length];
inputPositionA++;
for (inputPositionA = 0; inputPositionA < InputA.Length; inputPositionA++)
{
  numberA[inputPositionA] = Convert.ToByte(InputA.ToString(inputPositionA, 1));
}

int inputPositionB = 0;
byte[] numberB = new byte[InputB.Length];
inputPositionB++;
for (inputPositionB = 0; inputPositionB < InputB.Length; inputPositionB++)
{
  numberB[inputPositionB] = Convert.ToByte(InputB.ToString(inputPositionB, 1));
}

DateTime startTime = DateTime.Now;
//Console.WriteLine(startTime);
int count = 0;
byte[] a = Integer_BinarySquare(numberA, ref InputA, ref count);
byte[] b = Integer_BinarySquare(numberB, ref InputB, ref count);

byte[] aMinB = new byte[(InputA.Length > InputB.Length ? InputA.Length :
  InputB.Length)];
byte[] result = new byte[InputA.Length + InputB.Length];
bool enterA = true;
int indexA = 0;
int indexB = 0;
for (int i = 0; i < result.Length; i++)
{
  if (enterA)
  {
    result[i] = numberA[indexA];
    indexA++;
    enterA = indexA < numberA.Length;
  }
  else
  {
    result[i] = numberB[indexB];
    indexB++;
  }
}
int aMinBFrom = 0;
int diffFromA = 0;
int diffToA = numberA.Length - 1;
int diffFromB = numberA.Length;
int diffToB = result.Length - 1;
int aMinBTo = 0;
bool aIsLarger = true;
StringBuilder InputC = new StringBuilder();
ComputeDiff(result, diffFromA, diffToA, diffFromB, diffToB, out aMinBFrom, out
  aMinBTo, ref aIsLarger /* ref InputC,*/ /* ref count */);
byte[] c = Integer_BinarySquare(aMinB, ref InputC, ref count);

indexA = a.Length - 1;
```

```
indexB = b.Length - 1;
int indexC = c.Length - 1;
int rem = 0;
for (int i = result.Length - 1; i >= 0; i--)
{
  //count++;
  int subResult = rem;
  rem = 0;
  if (indexA >= 0)
    subResult = subResult + a[indexA];
  if (indexB >= 0)
    subResult = subResult + b[indexB];
  if (indexC >= 0)
    subResult = subResult - c[indexC];
  if (subResult > 9)
  {
    subResult = subResult - 10;
    rem = 1;
  }
  if (subResult < 0)
  {
    subResult = subResult + 10;
    rem = -1;
  }
  result[i] = (byte)subResult;
  indexA--;
  indexB--;
  indexC--;
}

// Divide by two
rem = 0;
int subDivide = 0;
for (int i = 0; i < result.Length; i++)
{
  //count++;
  subDivide = rem + result[i];
  rem = 0;
  switch (subDivide)
  {
    case 1:
      subDivide = 0;
      rem = 10;
      break;
    case 2:
      subDivide = 1;
      break;
    case 3:
      rem = 10;
      subDivide = 1;
      break;
    case 4:
      subDivide = 2;
      break;
    case 5:
      rem = 10;
      subDivide = 2;
      break;
    case 6:
      subDivide = 3;
```

```
            break;
          case 7:
            rem = 10;
            subDivide = 3;
            break;
          case 8:
            subDivide = 4;
            break;
          case 9:
            rem = 10;
            subDivide = 4;
            break;
          case 10:
            subDivide = 5;
            break;
          case 11:
            rem = 10;
            subDivide = 5;
            break;
          case 12:
            subDivide = 6;
            break;
          case 13:
            rem = 10;
            subDivide = 6;
            break;
          case 14:
            subDivide = 7;
            break;
          case 15:
            rem = 10;
            subDivide = 7;
            break;
          case 16:
            subDivide = 8;
            break;
          case 17:
            rem = 10;
            subDivide = 8;
            break;
          case 18:
            subDivide = 9;
            break;
          case 19:
            rem = 10;
            subDivide = 9;
            break;
        }
        result[i] = (byte)subDivide;
      }
      return result;
    }
  }
}
```

# Chapter 11 Deliverable 1

## Question 1

Give an O(n(m + r) time algorithm for the SR-problem for S, R, and T. Your algorithm should solve the problem in O(n(m + r) time.

## Solution

Let V be the result string obtained from searching string T for pattern S and replacing occurrences with string R. The length of T, S and R being n, m, and r.

The algorithm can described as:

```
i, j = 0                  // (1)
while (i < n) do          // (2)
  if (i < n - m)          // (3)
    j = 0                 // (4)
    while (j < m) and (i + j < n) and (S[j] = T[i + j]) do  // (5)
      if (j == m - 1)     // (6) Match found of S in T.
        V = V + R         // (7) Result string V updated with R.
        i = i + m - 1     // (8) Search in T shift m places.
      else                // (9)
        j = j + 1         // (10)
  if (j < m - 1)          // (11)S not found in T
    V = V + T[i]          // (12)Character in T append to V when S notmatched
  i = i + 1               // (13)Shift outer loop.
```

1. The outerloop (line 2) can shift up to a maximum **n**-times, that is, if there is no match of S in T (line 6 is never satisfied)
2. The next inner loop (line 5) can shift up to a maximum **m-times**, that is if string S is matches in T (line 6 is satisfied)
3. The replacing part (line 7) shifts upto a maximum r-times to append string R into V.

   Hence this algorithm takes O(n(m + r)). The algoritm never reaches this bound however because statement (1) and (2) rely on the opposite condtion (6) to occur.

## Question 2

Find the best-case and worst-case running time of your algorithm and discuss which properties of the input may result in the best-case and worst-case performance, respectively.

## Solution

As can be observed from the algorithm above, the *best-case* running-time of the algoritm is when it avoids going into branch (5), that is for character of string S does not occur in the string T in the first n – m places.

This would result in a best-case performance of O(n).

The *worst-case* running time would be encountered when condition (5) is satified.

The two worst-case scenario can be derived depending on the size of m and r:

1. r is small: Then it is better to allow condition (5) to always be met be never condition(6). The worst-case therefore arise when m = n / 2 and T contains matches upto the m – 1 character.
   This would give rise to:
   $$\Theta\big(n(m+r)\big) = \Theta\left(n\left(\frac{n}{2}+0\right)\right) = \Theta(n^2)$$

2. r is very large: Then it is better to always allow condition (5) and (6) to always be satified. The worst-case would therefore arise when m = 1 and T contains only character found in S.
   $$\Theta\big(n(m+r)\big) = \Theta(n+nr) = \Theta(nr)$$

## Question 3

Suppose that you only want to search and replace complete words in the document. How does this affect the problem? Design an algorithm for this variant.

The following algorithm searches and replaces words only. The testWord flag is introduced to indicate when a word has begun so that a test should spring into action. Otherwise string V just copies the characters in linear time.

### Solution

```
i, j = 0,                  // (1)
testWord = true            // (2)
while (i < n) do           // (3)
  if (i < n - m)           // (4)
    j = 0                  // (5)
    while (j < m) and (i + j < n) and (S[j] = T[i + j]) and testWord do //(6)
      if (j == m - 1) and (T[i + 1] = ' ') // (7)Match found of word in S in T.
        V = V + R          // (8) Result string V updated with R.
        i = i + m - 1      // (9) Search in T shift m places.
      else                 // (10)
        testWord = false   // (11)
        j = j + 1          // (12)
  if (j < m - 1)               // (13)S not found in T
    V = V + T[i]           // (14)Character in T append to V when S notmatched
    testWord = T[i + 1] = ' ';// (15) or any other line terminating chars.
  i = i + 1                // (16)Shift outer loop.
```

1. The outerloop (line 2) remains unaffected
2. The next inner loop (line 6) is modified is check whether the word needs testing and can shift up to a maximum **m-times**, that is if string S is matches in T (line 6 is satisfied)
3. The replacing part (line 7) shifts upto a maximum r-times to append string R into V.

## Question 4

The overlap of two strings A and B, denoted overlap(A,B) is the longest suffix of A that is a prefix of B. For instance, Overlap(ababa, ababc) = aba. Suppose that you

are given a data structure for S that for any $i$ and $\alpha \epsilon \Sigma$ supports the following query in constant time:

$$OverlapLength(i, \alpha): Return\ the\ length\ of\ the\ overlap\ between\ S[1..i]\alpha\ and\ S.$$

Use this data structure to design a faster algorithm for the SR-problem.

### Solution

The algorithm can described as:

```
i, j = 0                    // (1)
while (i < n) do            // (2)
  if (i < n - m)            // (3)
    while (j < m) and (i + j < n) and (S[j] = T[i + j]) do  // (4)
      if (j == m - 1)       // (5) Match found of S in T.
        V = V + R           // (6) Result string V updated with R.
      else                  // (7)
        j = j + 1           // (8)
  if (j > 0)                          // (9)
    ovl = overlapLength(S[1..j], S) // (10)
    if (j < m - 1)                    // (11)
      V = V + T[i..i + j - ovl]// apppend in V the chars to be skipped by i
    i = i + j - ovl - 1   // shift i (13)
  else                  // (14)
    V = V + T[i]        // (15)Character in T append to V when S notmatched
  i = i + 1             // (16)Shift outer loop.
```

The above algoritm is very much similar to the one solved in Question 1. The key distinction in the impact of the OverlapLength function. In step 9 to 13, i, the outer loop can be shifted at least once. At the same time testing in S need not resume at the first character. This leads to a dramatic reduction of loops, such the algoritm becomes linear polnomial.It therefore reduces to a O(n + r).

## Question 5

Show how to build a data structure in $O(m^3|\Sigma|)$ time supporting OverlapLength queries for S in constant time.

### Solution

You can construct such a data structure using an array of size m integers.

## Question 6

Combine the data structure from exercise 5 and the algorithm from exercise 4 to give a new algorithm for the S-R problem. Compare the new algoritm with the one from exercise 1.

### Solution

The OverlapLengh array can be constructed once. This can be done right at the beginning of the algorithm of exercise 4.
The combined time will then be $O(m^3|\Sigma|) + O(n + r) = O(m^3|\Sigma| + n + r)$

This is must faster compared to the $O(n(m+r))$ runnning time seen in exercise 1.

# Chapter 12 Deliverable 2

## Question 1

The solution is to use an algorithm that avoid comparing the ”don’t care” characters such that, it shift the matching to the next ”non-don’t care character”.

To be able to do so, the algorithm would have to perform the following:

1. Construct a data structure S’ such that S’ contains the shift needed at point S[i]

```
shift ← 0
position ← 0
for (i ← 1; i++; i < m)
 if s[i] = DontCareChar then
  s’[pos] ← s’[position] + 1
 else
  s’[i] = 0
  position = i
```

### *Example:*

Suppose $s = 1°°°34$ then the $s' = 300000$

2. Implement a naïve search replace but with the following optimization whenever it achieve a partial match it then proceed to perform the next match based on the shift suggested by the data structure s’ that was generated.

### *Example:*

Suppose we have $T = 1111134$ and $s = 1°°°34$.
Therefore $s' = 300000$
When shift = 0,

| T | 1 | 1 | 1 | 1 | 1 | 3 | 4 |
|---|---|---|---|---|---|---|---|
| S | 1 | *o* | *o* | *o* | 3 | 4 | |
| S’ | 3 | 0 | 0 | 0 | 0 | 0 | |

The first character is matched (green), then s’ causes the next character to attempt matching to be shift by three (orange region is skipped), where it fails (red)

When shift = 1,

| T | 1 | 1 | 1 | 1 | 1 | 3 | 4 |
|---|---|---|---|---|---|---|---|
| S | | 1 | *o* | *o* | *o* | 3 | 4 |
| S’ | | 3 | 0 | 0 | 0 | 0 | 0 |

Because of the mismatch the algorithms moves on with next shift.
The first character is matched (green), then s’ causes the next character to attempt matching to be shifted by three (orange region is skipped). It consequently matches the remaining part successfully and replaces it.

## Time-complexity

The first part for creating the data structure s’ would cost linear time in: $O(m)$ as it scans each character in S once.

The second part of the algorithm would cost $O(n(m - k))$ where k is the number of DontCareCharacters. Hence for each shift , there would be m – k attempt to match.

Replacing would cost $O(r * occ)$, where occ is the number of occurrence of S in T.

Hence the total time complexity would be $O(m + n(m - k) + r * occ)$

### Space complexity

The solution would require additional space to store s': $O(m)$

## Question 2

To be able to find subsequence occurrences of S in T, the algorithm would have to scan left to right on T once. For each character in T it would have to compare the the jth character in S that has not yet been matched. Initially j is the index to the first character in S. Whenver a match is encountered, the index of T is recorded and j is shift by one. When the full length of S has been matched, the result of the indexes of T that were matched are then printed and j is reset to the index of the first character again. The next subsequence is then matched.

```
j ← 0
for (i ← 0; i++; i < n)
   if s[j] = T[i]
    result ← i + ';'
    if j  = m – 1
      print "subsequence found = " + result
      j ← 0
   else
      j ← j + 1
```

### Time-complexity

The algorithm above simply goes through each character in T once with cost $O(n)$

If the algorithm had to find each subsequence.

### *Example*,

Given $T = aaaa$, and $S = aa$.

The subsequences would then be every combination i.e. [0,1], [0,2], [0,3], [1,2], [1,3], [2,3]

This give rise to a complexity of $O(n^2)$

## Question 3

An algorithm which included variable length don't care character could be solved as follows:

To be able to do so, the algorithm would have to perform the following:

1. Construct an data structure to store the OverlapLength discussed in deliverable 1 for each substring that occur before a variable length. Let us denote the substring as $s_j$, where j = 1,2,…,k., the k substring contained in S

   *Example:*

ab * c * aa would result in substrings [ab, c, aa], k = 3 and such each would have its OverlapLength data structure created.

2. Use algorithm discussed in Deliverable 1 that uses the OverlapLength, but with the following optimization, whenever substring $s_j$ is fully matched, proceed matching substring $s_{j+1}$ using $\text{OverlapLength}_{j+1.}$ .

### Time-complexity

The algorithm would cost the following:
Time to construct the OverlapLength for each substring + $O(n)$

## Question 4

Given two string of equal length n, and a query the computes the LCP of two strings in constant time, an algorithms to compute the Hamming Distance (i..e number of mismatch between the two strings could be performed as follows)

1. Compute the LCP between the two strings T and S.
2. If the LCP is zero then no matches were found return length of T, denoting no match.
3. If the LCP is nonzero = j, then it must mean that j + 1 character did not match, hence proceed to **Step 1** again using T[j+1...n] and S[j+1..n]. Also note that 1 character was a mismatch.

The algoritm below would therefore compute the HammingDistance.

```
HammingDistance (T, S)
  j ← LCP(T, S)
  if j = 0 return length(T)
  return HammingDistance(T[j + 1..n], S[j+1..n]) + 1
```

### Time-complexity

Step 1 in the algoritm would always cost constant time.
Step 2 would cost time to compute the length of the substring T. This can only occur once during the entire computation. Worst case would be $o(n)$
Step 3 would cost constant time * match worst-case for this would O(n/2) assuming every second character is a mismatch.
Hence the time complexity is upper-bound by $o(n)$

# Chapter 13 Deliverable 3

## Question 1

Design a dynamic programming algorithm to compute the edit distance between S and T.

## Solution

The following algorithm converts a string S into a string T using the following operations (Delete, Insert, Replace, and Match). To do so, an (m + 1) × (n + 1) array is constructed and initialized with values 0..n horizontally and 0..m vertically (see first row and first column below). The horizontal initialization represents the deletion operation required to convert VINTNER string to an empty string.

| | **To String** | |
|---|---|---|
| | | - |
| **From String** | - | 0 |
| | W | 1 |
| | R | 2 |
| | I | 3 |
| | T | 4 |
| | E | 5 |
| | R | 6 |
| | S | 7 |

The vertical initialization represents the insertion operation required to convert the string WRITERS into an empty string.

| | | | **ToString** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | - | V | I | N | T | N | E | R | |
| **FromString** | - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

To fill the remaining m×n cells the immediate top, diagonal, and left cells must be computed. The value filled is the based on cheapest operation possible. Choosing the top value would represent a Deletion, the Left an Insertion, and the Diagonal would either represent a Match (if the character incident to the cell are the same), or it would represent a Replace operation. The cost for each operation is 1 except for the Match which is 0.

The eventual table is shown below. The cell A[m,n] represents the optimal cost of converting the entire From String to the ToString which is the value we seek to find.

| | | | **To String** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | - | V | I | N | T | N | E | R | Delete Direction |
| **From String** | - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| | W | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| | R | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 6 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | I | 3 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | |
| | T | 4 | 4 | 3 | 3 | 3 | 4 | 5 | 6 | |
| | E | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 5 | |
| | R | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 4 | |
| | S | 7 | 7 | 6 | 6 | 6 | 6 | 6 | 5 | |
| | → Insert Direction | | | | | | | | | Match/Replace Direction ↘ |

**Edit-Distance(X,Y)**

```
  Array[0..m,0..n]
  Initialize A[i,0] = i for each i
  Initialize A[0,j] = j for each j
  for j = 1,...,n
    for i = 1,...,m
      A[i,j] = min(A[i-1,j] + 1, A[i,j-1] + 1, A[i-1,j-1] + Compare(X[i],Y[j]))
  return A[m,n]
```

where

**Min(a,b,c) =** a if a > b and a > c
b if b > c
c otherwise

**Compare[a,b]** = 0 if a = b
1 if a <> b

**Correctness**

**Correctness proof is similar to the one in Algorithm Design Chapter 6 (Sequence Alignment) Kleinberg/Tardos**

**Space Complexity**

The algorithm needs to compute (m + 1) * (n + 1) entries hence the space requirement is O(mn). The entry of interest is the A[m,n] hence the algorithm can do without having to store all the values. When entering the jth row, the values required are the j-1th row and the current row.

Below is a visual illustration of the cell value required to be kept a various states of the algorithm. The shaded parts are parts that need to be kept.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Computing this cell | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Computing this cell | | | | | |

Hence the amount of space can actually be optimised to m + 1 = O(m)

**Time Complexity**

Initialization of the array would cost O(m + n)

The two nested loops would cost O(mn)
The functions Min(a,b) and Compare(a,b) cost constant time.

Hence the total cost would be O(m + n) + O(mn) = O(mn)

## Question 2

Suppose that the edit distance between S and T is at most a small number k < n,m that is given as part of input. Use this to improve your algorithm from the previous exercise such that the running time depends on k.

## Solution

Continuing from the discussion, from question 1, the table of value represent the optimal cost of from converting an empty string to an empty string see cell (0,0) to convert string WRITERS to VINTNER see cell (7,7). Traversing from one end of the diagonal to the other diagonal is the cheapest path. But when the strings are not the same this is not the case. Deviating from the diagonal costs an INSERT or DELETE operation.

If it is known that there a no more than k operations, then there cannot exist more than k consecutive DELETE or INSERT operations from the diagonal at any given row. **This observation allows us to discard computing cells that are more than a distance k from the diagonal.**

The algorithm below implements that observation.

```
Edit-Distance(X,Y,k)
  Array[0..m,0..n]
  Initialize A[i,0] = i for each i to k
  Initialize A[0,j] = j for each j to k

  diagPosition = 0
  width = 2 * k
  nextRowStartIndex = 1
  for j = 1,...,n
    diagPosition = diagPosition + 1
    fromIndex = diagPosition - width
    if fromIndex < 0
      fromIndex = 1
    toIndex = diagPosition + width
    if toIndex > m
      fromIndex = m
    for i = fromIndex,...,toIndex
      topValue = A[i-1,j]
      topDiagValue = A[i-1,j-1]
      leftValue = A[i,j-1]

      if (topValue == 0)
        topValue = topDiagValue + 10
      if (leftValue == 0)
        leftValue = topDiagValue + 10

      A[i,j] = min(topValue + 1, leftValue + 1, topDiagValue + Compare(X[i],Y[j]))
```

```
  return A[m,n]

where
Min(a,b,c) = a if a > b and a > c
             b if b > c
             c otherwise

Compare[a,b] = 0 if a = b
               1 if a <> b
```

**Time Complexity**

We therefore need to compute the initialization k postions verital and horizontal
$O(k + k) = O(2k) = O(k)$

In the main algorithm we need to computes 2k + 1 (k distance on each side of the diagonal plus the diagonal itself for m rows. O(2km + 2m) = O(km+ m)

Hence the time complexity = O(k) + O(km + m) = O(k + km + m)

# Question 3

Suppose that the edit distance between S and T is at most a small number k, but we do not know what k is. Design an algorithm for this problem. The running time should also depend on k.

## Solution

If we k is unknown the following can be done. We first assume k = 0, and compute Edit_Distance. If the value of cell A[m,n] = 0 then we stop otherwise we continue incrementing k until A[m,n] <= k. The algorithm below implements this strategy.

```
Edit_Distance_2(S, T, tryK = 0)
{

  result = EditDistance(S, T, tryK)
  if result > tryK
    Edit_Distance_2(S, T, tryK = 0)
  return result
}
```

**Time Complexity:**

Time complexity would vary depending on how k is actually is. But with k small, the repetitions would be more efficient that computing the entire table.

# Question 4

How quickly can you compute the edit distance between S and T if you have n processors which can read and write in the table simultaneously? Give a fast algorithm for this problem.

## Solution

Each processor can be attached to a row. Such that once the row above has computed it cells value, the row just below can also compute its cell value. This will result into parallel computation. Hence there would be n + 1 processor. Processor 1 for the $0^{th}$ row, processor 2 for the $1^{st}$ row,…, processor n + 1 for the $n+1^{th}$ row.

The time complexity will therefore be O(m + 1) = O(m), assuming initialization also the responsibility of the respective processor.

## Question 5

The local edit distance problem is to find a substring S[i..j] of S that minimizes the edit distance between any substring of S and T. Give an O(nm) time algorithm for this problem. Hint the modify the initial conditions for the dynamic program from exercise 1.

### Solution

You can construct such a data structure using an array of size m integers.

## Question 6

Given an error threshold k the search and replace problem with errors (SRE-problem) is defined as the SR problem except that S matches at position i in T if i is the endpoint of a substring of T whose edit distance to S is at most k. Note that there might be more than one substring of T ending at position I and therefore we require that the shortest of these substrings are replaced.

### Solution

# Chapter 15 Bibliography

Kleinberg, J., & Tardos, E. (2006). *Algorithm Design.* Pearson Education, Inc.

# Chapter 16 Table of Equation and Algorithms