CS 31: Algorithms (Spring 2019): Lecture 4

Date: 2nd April, 2019

Topic: Divide and Conquer 2: MaxRangeSubArray, Karatsuba

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to deeparnab@dartmouth.edu.

---

1 Maximum Range Subarray

In this problem, we are given an array $A[1:n]$ of numbers (think integers or reals), and the goal is to find $i < j$ such that $A[j] - A[i]$ is maximized.

> MAXIMUM RANGE SUBARRAY
> **Input:** Array $A[1:n]$ of integers.
> **Output:** Indices $1 \leq i \leq j \leq n$ such that $A[j] - A[i]$ is maximized.
> **Size:** n, the length of A.

Once again, there is a trivial $O(n^2)$ time algorithm; go over all pairs (i,j) and choose the one that maximizes $A[j] - A[i]$. Once again, we think of a divide and conquer algorithm. Suppose we solved the problem on $A[1:n/2]$ and $A[n/2+1:n]$. More precisely, suppose (i_1, j_1) was the MRS for $A[1:n/2]$ and (i_2, j_2) was the MRS for $A[n/2+1:n]$. Clearly both of these are *candidate* or *feasible* solutions for $A[1:n]$.

Are there other candidate solutions? Yes, and these are of the form (i,j) with $i \leq n/2$ and $n/2 < j$. Is it any easier to find such "cross" (i,j) pairs? In this case the answer is a resounding **yes!**: since we are trying to maximize $A[j] - A[i]$, we should choose j which maximizes $A[j]$ in $n/2 < j \leq n$ and choose i such that $A[i]$ is minimized in $1 \leq i \leq n/2$. These are $O(n)$-time operations; a win over $O(n^2)$!

procedure MRS0($A[1:n]$):
    ▷ *Returns $1 \leq i \leq j \leq n$ maximizing $A[j] - A[i]$.*
    if $n = 1$ **then**:
        $(i,j) \leftarrow (1,1)$. ▷ Singleton Array
        return (i,j).
    $m \leftarrow \lfloor n/2 \rfloor$
    $(i_1, j_1) \leftarrow$MRS0($A[1:m]$)
    $(i_2, j_2) \leftarrow$MRS0($A[m+1:n]$)
    $i_3 \leftarrow \arg\min_{1 \leq t \leq m} A[t]$ ▷ Takes $O(m)$ time
    $j_3 \leftarrow \arg\max_{m+1 \leq t \leq n} A[t]$ ▷ Takes $O(m)$ time
    return best among $(i_1, j_1), (i_2, j_2), (i_3, j_3)$. ▷ Takes $O(1)$ time

As in merge-sort and counting inversions, if $T(n)$ is the worst case running time of MRS0, then looking at the running time on the worst array of length n, we get

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + Oa(n)$$

which evaluates to $T(n) = \Theta(n \log n)$. This seems good, but in fact we can actually do better using a similar idea as discussed in counting inversions algorthm: Ask More!

If you "opened up" the recursion tree, you would observe that the $\Theta(n)$ time to compute the max's and the min's in Lines 9 and 10 seems repetitive; the same comparisons are made more than once. This gives an idea of what to ask more for; we want our maximum range sub-array algorithm *also* returns the maximum and minimum of that sub-array. This gives us the next algorithm.

1: **procedure** MRS($A[1:n]$):
2: ▷ *Returns (s, t, i, j) where*

- *$A[j] - A[i]$ is maximized, and*
- *s, t are the indices of the min and max of A, respectively.*

3: **if** $n = 1$ **then**:
4: **return** $(1, 1, 1, 1)$ ▷ Singleton Array
5: $m \leftarrow \lfloor n/2 \rfloor$
6: $(s_1, t_1, i_1, j_1) \leftarrow$MRS($A[1:m]$)
7: $(s_2, t_2, i_2, j_2) \leftarrow$MRS($A[m+1:n]$)
8: $s \leftarrow \arg\min(A[s_1], A[s_2])$ and $t \leftarrow \arg\max(A[t_1], A[t_2])$. ▷ Takes $O(1)$ time
9: $(i, j) \leftarrow$ best solution among $\{(i_1, j_1), (i_2, j_2), (s_1, t_2)\}$.▷ Takes $O(1)$ time
10: **return** (s, t, i, j).

The conquer step in Line 8 takes only $O(1)$ time: the max of the whole array is the max of the maxima in the two halves. Same for the minima. Therefore, the recurrence inequality becomes

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(1)$$

solving which gives us the following.

Theorem 1. The MRS algorithm returns the maximum-range sub-array in $\Theta(n)$ time.

2 Multiplying Polynomials Faster: Karatsuba's Algorithm

Next we consider the problem of multiplying polynomials. The input is the $(n+1)$ coefficients of two univariate degree n polynomials $p(x)$ and $q(x)$ given as $P[0:n]$ and $Q[0:n]$. That is,

$$p(x) = \sum_{i=0}^{n} P[i] \cdot x^i \qquad \text{and} \qquad q(x) = \sum_{j=0}^{n} Q[j] \cdot x^j$$

We desire to output the coefficients the polynomial $r(x) = p(x) \cdot q(x)$. Note that the degree of $r(x)$ is $2n$, and thus the coefficients needs to be stored in an array $R[0 : 2n]$. We also assume that every $P[i], Q[j]$ are "small" numbers and so they can be added and multiplied in $\Theta(1)$ time[1].

An $O(n^2)$ time algorithm follows from the formula for $R[k]$ which is as follows:

$$\forall 0 \leq k \leq 2n, \quad R[k] = \sum_{0 \leq i,j \leq n : i+j=k} P[i] \cdot Q[j] = \begin{cases} \sum_{0 \leq i \leq k} P[i] \cdot Q[k-i] & \text{if } k \leq n \\ \sum_{(k-n) \leq i \leq n} P[i] \cdot Q[k-i] & \text{if } n < k \leq 2n \end{cases} \tag{1}$$

Do you see this? By the way, in signal processing this has another name. The array $R[0 : 2n]$ is called the ***convolution*** of the two arrays $P[0 : n]$ and $Q[0 : n]$. The above formula gives a $O(n^2)$-time algorithm to compute the convolution.

We now show how Divide-and-Conquer gives a faster algorithm.

Remark: *The story goes that in the early 1960s the famous Russian mathematician Andrei Kolmogorov held a seminar with the objective to show that any algorithm needs $\Omega(n^2)$ to multiply two degree n polynomials. After the first meeting, a young student named **Anatoly Karatsuba** came up with the algorithm we are about to describe. Kolmogorov canceled the remainder of the seminar.*

Let $m = \lceil n/2 \rceil$. Consider the polynomial $p(x)$ and write it as

$$p(x) = p_1(x) + x^m p_2(x) \quad \text{where} \quad p_1(x) = \sum_{i=0}^{m-1} P[i]x^i \quad \text{and} \quad p_2(x) = \sum_{i=0}^{n-m} P[m+i]x^i \tag{2}$$

Similarly write

$$q(x) = q_1(x) + x^m q_2(x) \quad \text{where} \quad q_1(x) = \sum_{j=0}^{m-1} Q[j]x^j \quad \text{and} \quad q_2(x) = \sum_{j=0}^{n-m} Q[m+j]x^j \tag{3}$$

This gives us the following formula for $r(x) = p(x) \cdot q(x)$.

$$\begin{aligned} r(x) &= (p_1(x) + x^m p_2(x)) \cdot (q_1(x) + x^m q_2(x)) \\ &= \Big(p_1(x) \cdot q_1(x)\Big) + x^m \cdot \Big(p_1(x) \cdot q_2(x) + p_2(x) \cdot q_1(x)\Big) + x^{2m} \cdot \Big(p_2(x) \cdot q_2(x)\Big) \end{aligned} \tag{4}$$

Now note that all four polynomials $p_1(x), p_2(x), q_1(x), q_2(x)$ have degree $\leq \lceil n/2 \rceil$. Therefore, (4) implies that $r(x)$ can be computed by recursively multiplying the four pairs $(p_1(x), q_1(x))$, $(p_1(x), q_2(x))$, $(p_2(x), q_1(x))$, and $(p_2(x), q_2(x))$. Subsequently, we need to add these polynomials up, but adding polynomials is a simple $\Theta(n)$ operation.

[1]If they are d-digits, this is what was studied in the Supplemental Problem : Number Theory set – take a look.

To sum, the above recursive algorithm has the following recurrence inequality: $T(n) \leq 4T(\lceil n/2 \rceil) + \Theta(n)$. We apply the Master Theorem and get $T(n) = O(n^2)$. Sigh! Much ado about nothing?

Next comes the Aha! insightful observation. We observe that we really don't need the individual products $p_1(x) \cdot q_2(x)$ and $p_2(x) \cdot q_1(x)$; rather we need just their sum.

Observation 1.

$$p_1(x)q_2(x) + p_2(x)q_1(x) = \Big(p_1(x) + p_2(x)\Big) \cdot \Big(q_1(x) + q_2(x)\Big) - \Big(p_1(x) \cdot q_1(x)\Big) - \Big(p_2(x) \cdot q_2(x)\Big)$$

Therefore, the (4) can be computed using 3 multiplication of polynomials of degree $\lceil n/2 \rceil$. These three are $\Big(p_1(x) \cdot q_1(x)\Big)$, $\Big(p_2(x) \cdot q_2(x)\Big)$, and $\Big((p_1(x) + p_2(x)) \cdot (q_1(x) + q_2(x))\Big)$. After computing this, the polynomial $r(x)$ can be computed using (4) and Observation 1 with $\Theta(1)$ polynomial additions and subtractions. Now, the recurrence inequality governing the above algorithm becomes

$$T(n) \leq 3T(\lceil n/2 \rceil) + \Theta(n)$$

which gives us the following.

Theorem 2. The algorithm KARATMULTPOLY multiplies two n-degree univariate polynomials in $O(n^{\log_2 3}) = O(n^{1.59})$ time.

procedure KARATMULTPOLY($P[0:n], Q[0:n]$): ▷ We want to return $R[0:2n]$.

  if $n = 0, 1$ **then**:

    return $R[0:2n]$ using the naive multiplication

  $m = \lceil n/2 \rceil$.

  ▷ *Recall definitions of $p_1(x), p_2(x), q_1(x), q_2(x)$ from* (2),(3)

  for $0 \le i \le m-1$ **do**

    $P'[i] = (P[i] + P[m+i])$

    $Q'[i] = (Q[i] + Q[m+i])$

  if $n > 2m-1$ **then**: ▷ In which case $n = 2m$ since $m = n/2$ or $m = (n+1)/2$.

    $P'[m] = P[n]$

    $Q'[m] = Q[n]$

  else:

    $P'[m] = Q'[m] = 0$

  ▷ *Now P' has the coefficients of $p_1(x)+p_2(x)$. Q' has the coefficients of $q_1(x)+q_2(x)$.*

  ▷ *Their degrees are $m-1$ or m depending on the parity of n.*

  ▷ *The else statement above forces degree m.*

  $R_1[0:2(m-1)] =$ KARATMULTPOLY $(P[0:m-1], Q[0:m-1])$

  $R_2[0:2(n-m)] =$ KARATMULTPOLY $(P[m:n], Q[m:n])$

  $R_3[0:2m] =$ KARATMULTPOLY $(P'[0:m], Q'[0:m])$

  ▷ *R_1 has the coefficients of $p_1(x) \cdot q_1(x)$*

  ▷ *R_2 has the coefficients of $p_2(x) \cdot q_2(x)$*

  ▷ *R_3 has the coefficients of $(p_1(x)+p_2(x)) \cdot (q_1(x)+q_2(x))$*

  ▷ *Also note that R_1, R_2, R_3 all have length $\le 2m$. We assume they all are $2m$ length by padding 0's.*

  for $0 \le i \le 2m$ **do**:

    $R_4[i] = (R_3[i] - R_1[i] - R_2[i])$

  ▷ *R_4 has the coefficients of $p_1(x) \cdot q_2(x) + p_2(x) \cdot q_1(x)$ and is degree $2m$*

  for $0 \le i \le 2n$ **do**:

    $R[i] = R_1[i] + R_4[i-m] + R_2[i-2m]$

    ▷ *We assume an array 'returns 0' if indexed out of its range. For instance, $R_4[-1]$ returns 0 and $R_1[2n]$ returns 0.*

    ▷ *When you actually code it, you need a few "if" statements to implement the above. A drill will ask you to do this. Please do that – it's super instructive.*

  return $R[0:2n]$