

Solutions for Homework Assignment #2

Answer to Question 1. Let D_1 and D_2 be the two databases, each containing n elements. Recall that the median of $2n$ (distinct) elements has exactly n elements larger than it, and exactly $n - 1$ elements smaller than it. Let $\text{QUERY}(t, D_i)$ return the element ranked t -th in the database D_i . Let $m = \lfloor n/2 \rfloor$. By applying $\text{QUERY}(m, D_1)$ and $\text{QUERY}(m, D_2)$ we can find the values of the m -th elements, a_1 and a_2 , of D_1 and D_2 , respectively. Now we claim that by comparing a_1 to a_2 , we can eliminate as potential candidates for the overall median half of the elements of each database. To see why, suppose $a_1 < a_2$ (the case $a_2 < a_1$ is symmetric). Then, none of the first m elements in D_1 (i.e., the elements ranked $1..m$ in D_1) can be the overall median, because there are more than n larger elements in D_1 and D_2 — namely, the elements ranked $m + 1..n$ in D_1 and the elements ranked $m..n$ in D_2 . Similarly, none of the last m elements in D_2 (i.e., the elements ranked $n - m + 1..n$ in D_2) can be the overall median, because there are more than $n - 1$ smaller elements in D_1 and D_2 — namely, the elements ranked $1..m$ in D_1 and the elements ranked $1..n - m$ in D_2 . Thus, with two queries we can eliminate from consideration half the elements of each database. Continuing in the same way, after $\lceil \log_2 n \rceil$ steps we will have eliminated from consideration all but one element in each database. At this point, we can retrieve the remaining two elements from the databases and return the minimum as the median.

This divide-and-conquer algorithm is shown below as the procedure $\text{DB-MEDIAN}(f_1, f_2, k)$. The interpretation of the parameters is that the search for the overall median of the elements in D_1 and D_2 has been narrowed to the elements ranked $f_1..f_1 + k - 1$ in D_1 and the elements ranked $f_2..f_2 + k - 1$ in D_2 . To find the overall median of the elements in the two databases, we call $\text{DB-MEDIAN}(1, 1, n)$.

```

DB-MEDIAN( $f_1, f_2, k$ )
if  $k = 1$  then
    return  $\min(\text{QUERY}(f_1, D_1), \text{QUERY}(f_2, D_2))$ 
else
     $m_1 := f_1 + \lfloor k/2 \rfloor - 1$ ;  $m_2 := f_2 + \lfloor k/2 \rfloor - 1$ ;
    if  $\text{QUERY}(m_1, D_1) < \text{QUERY}(m_2, D_2)$  then return  $\text{DB-MEDIAN}(m_1 + 1, f_2, \lceil k/2 \rceil)$ 
    else return  $\text{DB-MEDIAN}(f_1, m_2 + 1, \lceil k/2 \rceil)$ 

```

The correctness of this algorithm can be shown by proving that for any k , $1 \leq k \leq n$, if f_1, f_2 are positive integers such that $f_1 + k - 1 \leq n$ and $f_2 + k - 1 \leq n$, then $\text{DB-MEDIAN}(f_1, f_2, k)$ returns the median of the $2k$ elements ranked $f_1..f_1 + k - 1$ in D_1 and $f_2..f_2 + k - 1$ in D_2 . This is proved by complete induction on k , where the essential observation is the point we saw earlier: depending on the outcome of the comparison between the elements ranked $\lfloor k/2 \rfloor$ in these two ranges, we can eliminate from consideration half the elements in each range.

Let $Q(n)$ denote the number queries applied to D_1 and D_2 by the call $\text{DB-MEDIAN}(1, 1, n)$. This quantity is described by the recurrence $Q(n) = Q(\lceil n/2 \rceil) + 2$: we make two queries to retrieve the medians of two databases each of size n , and narrow our search to two databases each of size $\lceil n/2 \rceil$. In terms of the parameters of the Master Theorem, we have $a = 1$, $b = 2$ and $d = 0$, i.e., $a = b^d$. Thus, $Q(n) = \Theta(n^d \log n) = \Theta(\log n)$.

Answer to Question 2.

a. The standard algorithm from merging two sorted lists of length n_1 and n_2 takes time proportional to $n_1 + n_2$. The suggested algorithm proceeds in $k - 1$ stages. In stage 1 it merges two lists of length n each, producing a list of length $2n$. In stage i , $1 \leq i < k$, it merges the list produced in stage $i - 1$ (of length $i \cdot n$) and a list of length n , producing a list of length $(i + 1)n$. Thus, for all i such that $1 \leq i < k$, stage

i requires time proportional to $(i + 1)n$. The overall time complexity (over all $k - 1$ stages) is therefore $\Theta(\sum_{i=1}^{k-1} n(i + 1)) = \Theta(nk^2)$.

b. A better algorithm is to (a) divide the k lists into two groups, each consisting of (about) $k/2$ lists; (b) recursively merge the lists in each group; and (c) merge the resulting two lists (each of size $kn/2$ into a single list. The recursion continues until $k = 1$, in which it simply returns the input list (there is nothing to merge).

The time complexity of this algorithm is described by the following recurrence:

$$T(k) = \begin{cases} 2T(k/2) + kn, & \text{if } k > 1 \\ 1, & \text{if } k = 1 \end{cases}$$

(Here we assume, for simplicity, that k is a power of 2. If not, the general term of the recurrence becomes $T(k) = T(\lceil k/2 \rceil) + T(\lfloor k/2 \rfloor) + kn$, and the solution is within a constant factor of the solution for the special case.) Unwinding the recurrence using standard techniques (e.g., from CSCB36) yields $T(k) = \Theta(nk \log k)$, which is better than $\Theta(nk^2)$.

Answer to Question 3. Professor N. O’Bright’s proposed algorithm is wrong. Here is a counterexample. The graph is a “triangle” with nodes a, b , and c , and edges $\{a, b\}$, $\{b, c\}$, and $\{c, a\}$ with weights 100, 1, and 1, respectively. Suppose the initial partition of the set of nodes is into the node sets $V_1 = \{a, b\}$, and $V_2 = \{c\}$. In that case, N. O’Bright’s algorithm will construct a spanning tree consisting of edge $\{a, b\}$, and one of the other two edges. This spanning tree has cost 101, while the spanning tree consisting of $\{b, c\}$ and $\{c, a\}$ has cost 2. Thus, N. O’Bright’s algorithm doesn’t always find the MST.

Answer to Question 4. The product of $1 + x + 2x^2$ and $2 + 3x$ is a polynomial of degree 3. In order to interpolate the product, we must therefore evaluate each of the given polynomials in at least 4 points — conveniently a power of 2. The polynomial multiplication algorithm that uses the FFT proceeds as follows:

- (1) Use the FFT to evaluate each of the given polynomials at the four 4th roots of unity, namely $1, i, -1, -i$. Specifically, for each of the polynomials, compute $\text{FFT}(\vec{a}, i)$, where \vec{a} is the vector of the polynomial’s coefficients padded out with 0s to length four. Note that here we are using i as the primitive 4th root of unity. This computation yields, for each of the given polynomials, a vector of length four that is a value representation (at four points) for each of the given polynomials.
- (2) Multiply component-wise the two vectors that are value representations of the two given polynomials. This yields a vector of length four that is a value representation of their product.
- (3) Use the FFT to interpolate the product. Specifically, compute $\frac{1}{4}\text{FFT}(\vec{v}, -i)$, where \vec{v} is the vector computed in step (2). Note that here we are using as our primitive 4th root of unity the *reciprocal* of the primitive root we used in step (1): $-i = i^{-1}$. This computation yields a vector of length 4; by dividing the entries of this vector by 4 we get the coefficients of the product of the given polynomials.

Recall that applying the FFT to a vector \vec{x} of length 4 using as the primitive root i or, respectively, $-i$ amounts to multiplying by \vec{x} the matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \quad \text{or, respectively,} \quad \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}$$

Thus, the evaluation of polynomial $1 + x + 2x^2$ at the four chosen points yields

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ -1 + i \\ 2 \\ -1 - i \end{pmatrix}$$

Similarly, the evaluation of polynomial $2 + 3x$ at the four chosen points yields

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 2 + 3i \\ -1 \\ 2 - 3i \end{pmatrix}$$

Therefore, a value representation of the product of the two polynomials is

$$\begin{pmatrix} 4 \cdot 5 \\ (-1 + i) \cdot (2 + 3i) \\ 2 \cdot (-1) \\ (-1 - i) \cdot (2 - 3i) \end{pmatrix} = \begin{pmatrix} 20 \\ -5 - i \\ -2 \\ -5 + i \end{pmatrix}$$

Applying the FFT to this vector with primitive root $-i$ yields

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} 20 \\ -5 - i \\ -2 \\ -5 + i \end{pmatrix} = \begin{pmatrix} 8 \\ 20 \\ 28 \\ 24 \end{pmatrix}$$

Dividing each entry by 4 results in the vector $(2, 5, 7, 6)$ of the product's coefficients. Therefore the product of the two given polynomials is $2 + 5x + 7x^2 + 6x^3$ — which can be readily verified.

Answer to Question 5.

a. Let $M_1 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 9 & 16 \end{pmatrix}$. Then $\vec{a} \cdot M_1 = \vec{v}$.

b. As in part (a), we have $\vec{a} \cdot M = \vec{v}$ where $M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 9 & 16 \\ 0 & 1 & 8 & 27 & 64 \\ 0 & 1 & 16 & 81 & 256 \end{pmatrix}$.

So we have $\vec{a} = \vec{v} \cdot M^{-1}$.

So let $M_2 = M^{-1} = \begin{pmatrix} 1 & -25/12 & 35/24 & -5/12 & 1/24 \\ 0 & 4 & -13/3 & 3/2 & -1/6 \\ 0 & -3 & 19/4 & -2 & 1/4 \\ 0 & 4/3 & -7/3 & 7/6 & -1/6 \\ 0 & -1/4 & 11/24 & -1/4 & 1/24 \end{pmatrix}$

c. Our algorithm for multiplying two n -bit numbers A and B is as follows. We assume that n is a power of 3. If $n = 1$, then multiply the two bits in the obvious way. So assume $n > 1$.

Let $a_0, a_1, a_2, b_0, b_1, b_2$ be the $n/3$ -bit numbers obtained from the right, middle, and left thirds of A and B . So we have $A = a_0 + a_1 2^{n/3} + a_2 (2^{n/3})^2$ and $B = b_0 + b_1 2^{n/3} + b_2 (2^{n/3})^2$. Our goal is to compute

$C = A \cdot B = c_0 + c_1 2^{n/3} + c_2 (2^{n/3})^2 + c_3 (2^{n/3})^3 + c_4 (2^{n/3})^4$ where $\vec{c} = [c_0, c_1, c_2, c_3, c_4]$ are the coefficients of the degree-4 polynomial $r(z)$ obtained by multiplying the two degree-2 polynomials $p(z) = a_2 z^2 + a_1 z + a_0$ and $q(z) = b_2 z^2 + b_1 z + b_0$.

We first compute the vectors

$$\begin{aligned}\vec{u} &= (u_0, u_1, u_2, u_3, u_4) = (p(0), p(1), p(2), p(3), p(4)) \quad \text{and} \\ \vec{v} &= (v_0, v_1, v_2, v_3, v_4) = (q(0), q(1), q(2), q(3), q(4))\end{aligned}$$

by computing

$$\vec{u} = \vec{a} \cdot M_1 \quad \text{and} \quad \vec{v} = \vec{b} \cdot M_1.$$

Because this involves a constant number of additions, and multiplications by constants, on numbers of length $O(n)$, the time for this is $O(n)$. Note that the resulting numbers are integers with at most a constant number of bits more than $n/3$.

We next compute

$$\begin{aligned}\vec{w} &= (w_0, w_1, w_2, w_3, w_4) \\ &= (r(0), r(1), r(2), r(3), r(4)) \\ &= (p(0) \cdot q(0), p(1) \cdot q(1), p(2) \cdot q(2), p(3) \cdot q(3), p(4) \cdot q(4))\end{aligned}$$

by recursively computing

$$w_0 = u_0 \cdot v_0, \quad w_1 = u_1 \cdot v_1, \quad w_2 = u_2 \cdot v_2, \quad w_3 = u_3 \cdot v_3, \quad w_4 = u_4 \cdot v_4.$$

This involves 5 multiplications of $n/3$ -bit integers, and takes time $5 \cdot T(n/3)$ (where $T(n)$ is the time for our algorithm to multiply two n -bit integers). Actually, since the numbers we are multiplying recursively are integers with at most a constant number of bits more than $n/3$, the time is $5 \cdot T(n/3) + O(n)$.

We now compute

$$\vec{c} = \vec{w} \cdot M_2$$

where M_2 is the matrix from part (b). We compute the values of \vec{c} as rational numbers with integers in the numerators and denominators. Because all the integers have $O(n)$ bits and we are only performing a constant number of additions and multiplications by constants, this can be done in time $O(n)$, and results in rational numbers with integers in the numerators and denominators with $O(n)$ bits. (In fact, we can assume the denominator is always 24.)

Lastly, we compute

$$C = c_0 + c_1 2^{n/3} + c_2 (2^{n/3})^2 + c_3 (2^{n/3})^3 + c_4 (2^{n/3})^4.$$

This involves some shifting, and a constant number of additions and multiplications by constants on rational numbers with integers in the numerators and denominators with $O(n)$ bits. This can be done in time $O(n)$, resulting in the proper integer result!

Note that we have $T(n) = 5 \cdot T(n/3) + O(n)$. So $T(n) \in O(n^{\log_3 5})$. Since $\log_3 5 < 1.47 < \log_2 3$, we have that this algorithm is asymptotically faster than Karatsuba's.