CMPT 705: Design and Analysis of Algorithms **Spring 2008**

Chapter 5 — Divide and Conquer

Lecturer: Binay Bhattacharya *Scribe: Chris Nell*

5.1 Introduction

Given a problem P with input size n, $P(n)$, we define a divide and conquer algorithm as one which solves P according to the following paradigm:

1. Split P into a set of m smaller sub-problems, P'_i, $i = [1, .., m]$, each of which takes input of size $\alpha_i n$, where $\alpha_i < 1$ for all i.
2. Recursively solve each of the P'_i subproblems
3. Merge the individual subproblem solutions to obtain a solution for P.

Complexity analysis of such algorithms typically proceeds by first deriving a recurrence relation for a single iteration of the algorithm, and then using one of several techniques to infer the overall running time. For recurrence relations taking a particular common form, the Master Theorem may be used to immediately determine a bound on this running time.

A commonly-studied example of a divide and conquer algorithm is merge sort, which is described in detail in most undergraduate algorithm analysis textbooks. For this algorithm, one can derive the following recurrence relation:

$$\text{where } T(n) = \text{ time to sort } n \text{ elements},$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The Master Theorem can be used to show that this recurrence results in an $O(n \log n)$ running time for merge sort.

An point of note is that the above recurrence relation makes an implicit assumption about n; namely, that it is a perfect power of two. We can write a more general recurrence as:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn$$

However, this recurrence has the same asymptotic running time as the simplified one presented. In general, simplified recurrences which do not affect asymptotic complexity, such as those obtained by treating n as evenly-divisible, are cleaner to analyze than their more explicit forms.

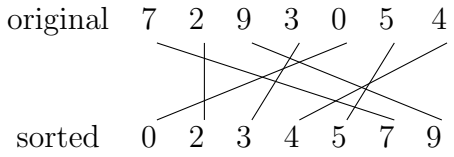
5.2 Problem - Counting Inversions

5.2.1 Problem Description

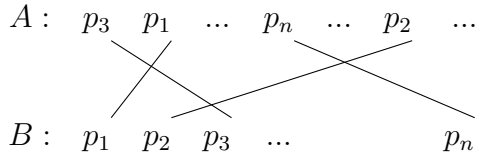
Given a sequence S of n numerical values, $S =< a_1, a_2, ..., a_n >$, we say that elements a_i and a_j of S realize an inversion if $i < j$, but $a_i > a_j$.

Our problem is to find and count all possible inversions present in S. Clearly, an $O(n^2)$ algorithm can be achieved by checking all possible (a_i, a_j) pairs. Thus, we seek a more efficient $O(n \log n)$ solution.

We may gain some insight to the problem by considering an example. Given a specific sequence, we can list its elements in sequence, and again in sorted order. We then draw an edge between corresponding elements of the two lists. The number of inversions in S is the number of intersections between these edges. This is illustrated below for $S =< 7, 2, 9, 3, 0, 5, 4 >$:



The number of inversions in a sequence can be considered a measure of its sortedness. It can also be used as a distance metric to compare two sequences, an interpretations with real-world application. For example, several services (such as Amazon.com and Netflix) use a similar metric to provide product recommendations to customers. In this example, imagine that customers A and B provide a ranked list of product preferences, and consider that there are n elements in common between these lists, $\{p_1, p_2, ..., p_n\}$. If we consider B's ranking to define the sorted order, we can quantify the similarity of their preferences as the number of inversions in A's list, as illustrated:



If by this measure, A and B are found to have similar preferences, and if B has ranked highly some products that A has not ranked, then those products may be confidently recommended to A.

5.2.2 Divide & Conquer Approach

Design of a divide and conquer algorithm for solving the inversion counting problem closely parallels that of merge sort:

1. Split S into two equal halves L and R:

L	R
$n/2$	$n/2$

2. Recursively count the inversions in each of L and R, ensuring that both L and R end up sorted.

3. Find the number of inversions occurring between L and R (that is, inversions pairs consisting of one element from L and one from R). This number, plus the number of inversions in each of L and R, is the desired solution.

Step 3 is realized by merging the sorted lists L and R, to reconstruct S. This results in S also being sorted, as required. The standard merge algorithm (as used in merge sort) may be used, with a single augmentation: whenever an element from R is merged back into to S, the inversion count is increased by the number of un-merged elements remaining in L.

Clearly step 3 occurs in linear time, as merge is linear. As such, we can write a recurrence relation for this algorithm:

$$\text{where } T(n) = \text{ time to count the inversions among } n \text{ elements,}$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Just as with merge sort, the Master Theorem can be used to show that this recurrence results in an $O(n \log n)$ running time.

It is interesting to consider the improvement realized by using the merge algorithm in Step 3. If we instead run a binary search over R for each element in L in order to find all inversion pairings, our recurrence would become:

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2}\log n + O(n)$$

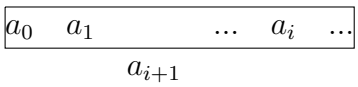
One can show that this recurrence leads to an $O(n \log^2 n)$ upper bound, substantially worse than $O(n \log n)$.

5.2.3 Non-Recursive Approach

The inversion counting problem can also be solved without using divide and conquer techniques, by extending insertion sort in the following way:

- Whenever an element from S is inserted into the sorted array S', all elements to its right in S' represent unique inversions. By maintaining a running total count of the number of these elements, we count the number of inversions in S. Since S is an array, we can determine this number in constant time.

In the example below, all elements to the right of a_{i+1}, the element being inserted, are the result of inversions in S.



However, since insertion into an array is linear due to data movement, this algorithm runs in $O(n^2)$ time, just as insertion sort does. If instead of a list we use an augmented balanced binary search tree, we improve our insertion operations to $O(\log n)$, and thus the running time of the algorithm to $O(n \log n)$, equal to the recursive solution.

Note that the binary tree must be augmented such that every node stores the number of elements in the subtree under it. This allows the number of elements to the right of an inserted element to be counted in the following manner:

- Along the path from the root to the location of the inserted node, every time a left child is explored, the value stored at the right child is added to the inversion count.

This amounts to a constant amount of work per visited node added to the usual tree insertion operation. Similarly, updating the values stored at nodes as a result of insertion also adds a constant amount of work per node to the normal insertion operation.

5.3 Problem - Long Integer Multiplication

5.3.1 Problem Description

Consider a two n-bit binary integers, a and b, where we assume that $n = 2^k$ for some k:

$$a = a_0 a_1 ... a_{\frac{n}{2}-1} a_{\frac{n}{2}} ... a_{n-1}$$
$$b = b_0 b_1 ... b_{\frac{n}{2}-1} b_{\frac{n}{2}} ... b_{n-1}$$

Here, a_i represents the i^{th} bit of integer a; note that integers of length less than n can be represented by padding the most significant digits with 0's.

For example, if $a = 1101$ and $b = 101$, we compute $c = ab$ as:

a	=		1	1	0	1
b	=			1	0	1
			1	1	0	1
		0	0	0	0	
	1	1	0	1		
1	0	0	0	0	0	1

As illustrated, the normal integer multiplication method requires $O(n^2)$ time. We seek a faster algorithm.

5.3.2 First Divide & Conquer Algorithm

As usual with divide and conquer algorithms, we divide the input into halves:

$$x_1 = a_1 a_2 ... a_{\frac{n}{2}-1}$$
$$x_0 = a_{\frac{n}{2}} ... a_{n-1}$$
$$y_1 = b_1 b_2 ... b_{\frac{n}{2}-1}$$
$$y_0 = b_{\frac{n}{2}} ... b_{n-1}$$

Thus, we have:

$$a = x_0 + 2^{\frac{n}{2}} x_1$$
$$b = y_0 + 2^{\frac{n}{2}} y_1$$

And from this, we can calculate:

$$c = ab = x_0 y_0 + 2^{\frac{n}{2}} (x_1 y_0 + x_0 y_1) + 2^n x_1 y_1$$

From the above equation for c, it is clear that we must recursively perform four multiplications: $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, and $x_0 y_0$. Once these multiplications are performed, three additions and two bit-shift operations are needed to combine the solutions; these can occur in linear time. Thus, we arrive at the recurrence relation:

$$\text{where } T(n) = \text{ time to multiply two } n\text{-bit binary integers,}$$
$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

By the Master Theorem, this recurrence is in $O(n^{\log_2 4}) = O(n^2)$, the same as the non-recursive solution; we have not yet made any improvement. However, we now investigate a method to reduce the number of recursive multiplications required from four to three.

5.3.3 Revised Divide & Conquer Algorithm

In order to reduce the number of recursive multiplications needed from four to three, we employ a trick. Observe that:

$$(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + c_0 y_1 + x_0 y_0$$
$$(x_1 + x_0)(y_1 + y_0) - x_0 y_0 - x_1 y_1 = x_1 y_0 + x_0 y_1$$

We can use this to rewrite the equation for c as:

$$c = ab = x_0 y_0 + 2^{\frac{n}{2}} [(x_1 + x_0)(y_1 + y_0) - x_0 y_0 - x_1 y_1] + 2^n x_1 y_1$$

Thus, we only need to recursively calculate $(x_1 + x_0)(y_1 + y_0)$, $x_0 y_0$, and $x_1 y_1$. These, along with additions and bit-shifts done in linear time, result in the recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

By the Master Theorem, this recurrence is in $O(n^{\log_2 3}) \approx O(n^{1.59})$, significantly better than our previous algorithms.

5.4 Problem - Matrix Multiplication

5.4.1 Problem Description

We now present a recursive method of performing square matrix multiplication analogous to that presented for integers. Recall that the usual method for multiplying two $n \times n$ matrices:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & & & \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

$$C = A \times B = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & & & \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

$$c_{ij} = \begin{pmatrix} a_{i1} & a_{i2} & \dots & a_{in} \end{pmatrix} \begin{pmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{pmatrix}$$

It is clear that each of the n^2 elements of C requires n unique multiplications of elements from A and B; thus, this is an $O(n^3)$ algorithm. Again, we seek to improve this bound using a divide & conquer approach.

5.4.2 First Divide & Conquer Algorithm

Here, we divide each of our input matrices into four $\frac{n}{2} \times \frac{n}{2}$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where, for example:

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1\frac{n}{2}} \\ a_{21} & a_{22} & \dots & a_{2\frac{n}{2}} \\ \vdots & & & \\ a_{\frac{n}{2}1} & a_{\frac{n}{2}2} & \dots & a_{\frac{n}{2}\frac{n}{2}} \end{pmatrix}$$

This allows us to write:

$$C = A \times B = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Clearly, in order to compute the elements of C, we need to perform eight unique matrix multiplications. Additionally, in order to merge these solutions we must perform additions on on all elements of four $\frac{n}{2} \times \frac{n}{2}$ matrices; this takes $O(n^2)$ time. These lead to the recurrence relation:

$$\text{where } T(n) = \text{ time to multiply two } n \times n \text{ matrices,}$$
$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

By the Master Theorem, this is in $O(n^{\log_2 8}) = O(n^3)$. Again, we must perform some algebraic tricks to achieve a more efficient algorithm.

5.4.3 Revised Divide & Conquer Algorithm

We aim to perform matrix multiplication using seven, rather than eight, recursive multiplications. In order to do this, we define the following seven matrices:

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} + A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} + A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Note the symmetry between P_1 and P_4, P_2 and P_3, and the similarity between P_5 through P_7. Using these seven matrices, we can determine everything necessary to calculate C[1]:

$$\begin{aligned} C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

[1]Verification of these relationships is left as an exercise to the reader.

Thus, with only seven matrix multiplications (and eighteen additional quadratic additions/subtractions), we can perform matrix multiplication with the recurrence relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

By the Master Theorem, this recurrence is in $O(n^{\log_2 7} \approx O(n^{2.8})$, a distinct improvement over the brute force $O(n^3)$ algorithm.. This technique for matrix multiplication is known as *Strassen Multiplication*.

5.4.4 Comments

Continuing research along these lines on efficient matrix multiplication has yielded an $O(n^{2.376})$ algorithm (Coppersmith & Winograd, 1987), the best currently known. An open conjecture exists stating that matrix multiplication should be possible in $O(n^{2+\epsilon})$ for any choice of ϵ. However, other issues become increasingly important as the asymptotic bounds on matrix multiplication complexity are tightened.

It should be clear that efficient matrix multiplication methods are inherently more complex than brute force methods. In addition to making the associated algorithms more difficult to implement, this complexity can lead to a loss of precision due to round-off error from the many addition/subtraction operations introduced (recall that adding two n-bit integers generates $n+1$ bits of precision). However, this loss of precision is less significant than that from multiplicative operations (since multiplying two n-bit integers generates $2n$ bits of precision).

5.5 Problem - Polynomial Multiplication

5.5.1 Problem Description

Given two polynomials of degree $n-1$, defined by their n coefficients

$$A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1}$$

We seek the product

$$C(x) = A(x)B(x) = c_0 + b_1 x + c_2 x^2 + ... + c_{2n-2} x^{2n-2}$$

For example, we want to be able to calculate[2]

$$(2x^2 + 5)(x^2 + 2x + 3) = 2x^4 + 4x^3 + 11x^2 + 1 - x + 15$$

Specifically, we seek all $2n - 1$ coefficients which together determine their product. We know that c_j, the coefficient of C's j^{th}-order term, is given by

$$c_j = a_0 b_j + a_1 b_{j-1} + ... + a_j b_0$$

A straightforward implementation of this formula will require $O(n^2)$ steps to determine C. As before, we seek a more efficient solution via divide and conquer.

5.5.2 First Divide & Conquer Algorithm

Consider first the intuitive divide and conquer approach; we divide $A(x)$ into two polynomials of order $\frac{n}{2} - 1$:

$$\begin{aligned} A(x) &= \left(a_0 + a_1 x + ... + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}\right) + x^{\frac{n}{2}} \left(a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + ... + a_{n-1} x^{\frac{n}{2}-1}\right) \\ &= A_1(x) + x^{\frac{n}{2}} A_2(x) \end{aligned}$$

Similarly, we divide $B(x) = B_1(x) + + x^{\frac{n}{2}} B_2(x)$. Then we have:

$$\begin{aligned} C(x) &= A(x)B(x) \\ &= A_1(x)B_1(x) + x^{\frac{n}{2}} \left[A_1(x)B_2(x) + x^n A_2(x)B_2(x) + A_2(x)B_1(x)\right] \end{aligned}$$

At first glance, it appears we need to perform four recursive multiplications, plus linear time additions and solution merging, for a total runtime of $O(n^{\log_2 4}) = O(n^2)$. If we employ the same trick used in Section 5.3.3 for fast integer multiplication, we can improve this bound to $O(n^{\log_2 3})$ by computing $[A_1(x) + A_2(x)][B_1(x)B_2(x)]$ instead of $A_1(x)B_2(x) + A_2(x)B_1(x)$. However, we are looking for an even more efficient solution - we seek an $O(n \log n)$ algorithm.

[2]We can consider both terms to be polynomials of equal degree by padding the lower-degree polynomial with zero-coefficient higher order terms as necessary.

5.5.3 Motivation for a Second D&C Strategy

From algebra, we know that a polynomial of degree k is uniquely determined by $k+1$ points. This provides us with the key idea motivating our next algorithm: we will generate these points, and from them find the associated polynomial.

We first define $m = 2n-1$, the number of points needed to specify C. For convenience, we increase m as necessary to ensure it is a power of 2, and then we rewrite $A(x)$ and $B(x)$ as polynomials of degree $m-1$ by adding higher-order terms with zero coefficients:

$$A(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1} + 0x^n + 0x^{n+1} + ... + 0x^{m-1}$$
$$B(x) = b_0 + b_1 x + ... + b_{n-1}x^{n-1} + 0x^n + 0x^{n+1} + ... + 0x^{m-1}$$

Our high-level approach is then:

1. Carefully choose m points $\{x_0, x_1, ..., x_{m-1}\}$ in $O(m)$ time
2. Evaluate $A(x_j)$ and $B(x_j)$ for all x_j
3. Compute $C(x_j) = A(x_j)B(x_j)$ for all x_j, in $O(m)$ operations
4. Interpolate these points to obtain the coefficients c_j of $C(x)$

It is clear that steps 1 and 3 can be done in linear time; we need to develop an efficient method for performing steps 2 and 4.

First note that we can make step 2 somewhat more efficient by computing the polynomials A and B in nested fashion; for example:

$$A(x_i) = a_0 + x_j(a_1 + x_j(a_2 + x_j(a_3... + x_j(a_{m-1})...)))$$

This requires $O(m)$ time per x_j; we can do better.

In order to execute step 4 in general, we must be able to solve the associated system of equations:

$$C(x) = c_0 + c_1 x + c_2 x^2 + ... + c_{m-1}x^{m-1}$$

$$\begin{pmatrix} C(x_0) \\ C(x_) \\ \vdots \\ C(x_{m-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & ... & x_0^{m-1} \\ 1 & x_1 & x_1^2 & ... & x_1^{m-1} \\ \vdots & & & & \\ 1 & x_{m-1} & x_{m-1}^2 & ... & x_{m-1}^{m-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix}$$

Solving this requires being able to invert the central matrix, which we will call F. Doing so directly requires $O(n^3)$ time. Instead, we motivate a particular selection of m points x_j which will end up allowing this inversion to be done much more easily.

5.5.4 Evaluating $A(x)$ at m Points

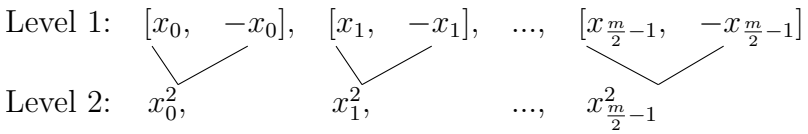
In an effort to apply divide and conquer techniques to this subproblem, we observe that we can further rewrite $A(x)$ as:

$$\begin{aligned} A(x) &= (a_0 + a_2x^2 + ... + a_{n-2}x^{n-2}) + (a_1x + a_3x^3 + ... + a_{m-1}x^{m-1}) \\ &= (a_0 + a_2(x^2) + a_4(x^2)^2 + ... + a_{m-2}(x^2)^{\frac{m}{2}-1}) \\ &+ x(a_1 + a_3(x^2) + a_5(x^2)^2 + ... + a_{m-1}(x^2)^{\frac{m}{2}-1}) \\ &= A_{even}(x^2) + xA_{odd}(x^2) \end{aligned}$$

Here, we have divided one polynomial of degree m into two of degree $\frac{m}{2}$. Further, if we choose our points x_j such that they occur in pairs $\pm x_j$, we only need to evaluate $A_{even}(x_j^2)$ and $A_{odd}(x_j^2)$ once for each pair:

$$\begin{aligned} A(+x_j) &= A_{even}(x_j^2) + x_iA_{odd}(x_j^2) \\ A(-x_j) &= A_{even}(x_j^2) - x_iA_{odd}(x_j^2) \end{aligned}$$

At the first level of recursion, we see that in order to generate the m values of $A(x_j)$ required, we need to perform just two recursive evaluations per pair, one for each of A_{even} and A_{odd}:



Thus, at the first level of recursion, we seem to have the recurrence:

where $T(m) =$ time to generate values of $A(x)$ at m distinct points,

$$T(m) = 2T\left(\frac{m}{2}\right) + O(m)$$

Unfortunately, at the second level of recursion we no longer have $\pm x_j$ pairings as we did at the first level, so our recurrence relation fails to hold.

We need to develop this idea further, so that pairing holds at every level of recursion. This requires the introduction of complex-valued x_j.

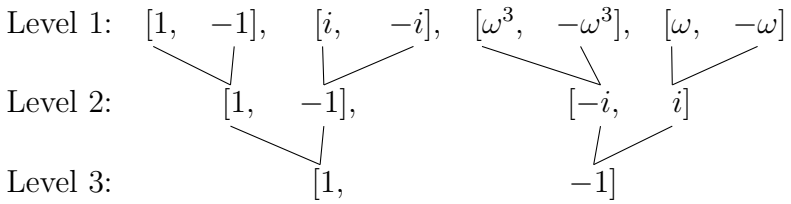
Consider, for example, the case where $m = 8$. Where $i^2 = -1$, we define $\omega = 0.707 + 0.707i$; in general, we define $\omega = \cos(2\pi/m) + i\sin(2\pi/m)$. With this choice of ω we obtain:

$$\begin{aligned}
\omega^0 &= 1 \\
\omega^1 &= 0.707 + 0.707i \\
\omega^2 &= i \\
\omega^3 &= \omega\omega^2 = -0.707 + 0.707i \\
\omega^4 &= (\omega^2)^2 = i^2 = -1 \\
\omega^5 &= \omega\omega^4 = -\omega \\
\omega^6 &= \omega^2\omega^4 = -i \\
\omega^7 &= \omega^4\omega^3 = -\omega^3 \\
\omega^8 &= (\omega^4)^2 = 1
\end{aligned}$$

Thus we see that $\omega^8 = 1$; we say that ω as defined is the 8^{th} *root of unity* (in general, ω will be the m^{th} root of unity). We will let

$$x_j = \omega^j$$

for each of the $m = 8$ values of x_j at which we generate values of $A(x)$. Using these re-defined values of x_j, at the each level of recursion we have the following pairings:



As illustrated, in the second and third levels of recursion (after each of the initial values is squared and the results squared again) we maintain $\pm\, x_j$ pairings. Thus, at each level we are able to manage with only two recursive multiplications, preserving our $T(m) = 2T\left(\frac{m}{2}\right) + O(m)$ recurrence through all levels of recursion. By the Master Theorem, this shows we are able to generate m points of $A(x)$ in $O(m \log m)$ time.

To better understand the behavior of ω^j, we consider the corresponding plots on the complex plane (where we we plot the real component of ω^j on the x-axis, and the complex component on the y-axis), shown in Figure 5.1. From these plots, we see that ω^j corresponds to a $2\pi i/8$ rotation about the complex unit circle. Further, we see that multiplication by a factor of ω has the effect of rotating an additional $2\pi/8$. This plot also allows us to geometrically understand the definition of ω: $\cos(2\pi/m)$ and $\sin(2\pi/m)$ are the x- and y-components respectively of a unit line segment rotated by $2\pi/m = 360°/m$.

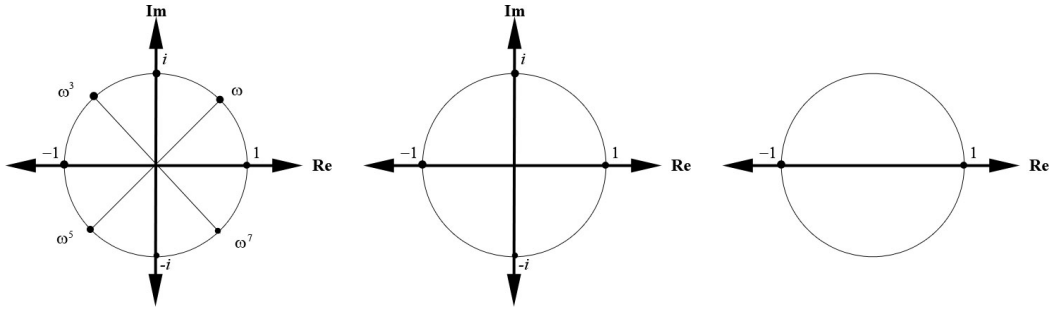
If we implement the strategies discussed for generating points from $A(x)$, we arrive at Algorithm 1. This algorithm implements what is also known as a *Fast Fourier Transform (FFT)*.

5.5.5 Interpolating to find $C(x)$

To this point, we have established an $O(m \log m)$ algorithm for generating m points from the polynomial $A(x)$; we do the same for $B(x)$. We can then use these points to generate m points from $C(x)$:

$$C(\omega^j) = A(\omega^j)B(\omega^j)$$

Thus, we have generated m points from $C(x)$ in $O(m \log m)$ time. However, we still need to find the coefficients c_j of $C(x)$, which is what we are ultimately



(a) First level of recursion
(b) Second level of recursion (after squaring (a))
(c) Third level of recursion (after squaring (b))

Figure 5.1. Plot of ω^j for j from 0 to 7, where $\omega = \cos(2\pi/m) + i\sin(2\pi/m)$ and $m = 8$.

Algorithm 1 The Divide and Conquer FFT Algorithm

FFT(A**,** m**,** ω**)**

Require: A is an array of length m, corresponding to the m coefficients of the $m-1$-degree polynomial $A(x)$. We assume m is a power of 2.

Require: ω is the primitive m^{th} root of unity ($\omega^m = 1$)

```
if m = 1 then
  return A[0]
else
  A_even ← (A[0], A[2], ..., A[m − 2])
  A_odd ← (A[1], A[3], ..., A[m − 1])
  V_even ← FFT(A_even, m/2, ω^2)
  V_odd ← FFT(A_odd, m/2, ω^2)
end if
for j = 0, 1, ..., m/2 − 1 do
  V[j] ← V_even[j] + ω^j V_odd[j]
  V[j + m/2] ← V_even[j] − ω^j V_odd[j]
end for
return V
```

seeking. As discussed in Section 5.5.3, this requires us to solve the system of equations:

$$\begin{pmatrix} C(\omega^0) \\ C(\omega^1) \\ \vdots \\ C(\omega^{m-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{m-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \ldots & \omega^{2\cdot(m-1)} \\ \vdots & & & & \\ 1 & \omega^{(m-1)} & \omega^{2\cdot(m-1)} & \ldots & \omega^{(m-1)\cdot(m-1)} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix}$$

Here, the column vector containing m values of $C(x_j = \omega^j)$ is what we have just generated, and the matrix is F for our choices of x_j. As discussed, we must invert F in order to do this. However, our particular choice of ω makes this inversion easy[3]:

[3] See text for algebraic details.

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \frac{1}{m} \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega^{-1} & \omega^{-2} & ... & \omega^{-(m-1)} \\ 1 & \omega^{-2} & \omega^{-2\cdot 2} & ... & \omega^{-2\cdot(m-1)} \\ \vdots & & & & \\ 1 & \omega^{-(m-1)} & \omega^{-2\cdot(m-1)} & ... & \omega^{-(m-1)\cdot(m-1)} \end{pmatrix} \begin{pmatrix} C(\omega^0) \\ C(\omega^1) \\ \vdots \\ C(\omega^m) \end{pmatrix}$$

Indeed, since the form of this matrix is known *a priori*, there is no need to build these matrices at all in order to find $C(x)$. Instead, we consider the implied polynomial defined by:

$$D(x) = C(1) + C(\omega)x + C(\omega^2)x^2 + ... + C(\omega^{m-1})x^{m-1}$$

Note that this polynomial has been defined such that:

$$\begin{aligned} D(1) &= c_0 \\ D(\omega^{-1}) &= c_1 \\ D(\omega^{-2}) &= c_2 \\ &\vdots \\ D(\omega^{-(m-1)}) &= c_{m-1} \end{aligned}$$

All we need to do in order to obtain the c_j values we are looking for is evaluate $D(x)$ at these specific m points. Moreover, it is easy to verify that just as using points $x_j = \omega^j$ allowed convenient $\pm$ pairings at each level of recursion, so too does using points $x_j = \omega^{-j}$. Thus, we can re-use our recursive FFT algorithm to calculate these in $O(m \log m)$ time as well!

Note: although the points x_j we have chosen are in general complex, the values c_j determined using them are nonetheless real-valued. That this is true should be clear by observation that the coefficients of both $A(x)$ and $B(x)$ are all real-valued; so too must be the coefficients of $C(x)$.

5.5.6 Applications of FFTs

There are many real-world applications of FFTs. Perhaps one of the most common is in the digitization and transmission of analog signals, such as sound recordings. The signal is assumed to be a polynomial of some predefined degree $m - 1$ (typically very large), and m discrete samples at times

determined by ω are taken. These samples, rather than the complete signal, are transmitted; the recipient simply uses the FFT algorithm to reconstruct the original signal.

Perhaps surprisingly, another application of FFTs is in the multiplication of two n-bit integers in $O(n \log n)$ time - more efficient than the $O(n^{\log_2 3})$ algorithm developed in Section 5.3.

To see how this is done, consider that we can rewrite the two n-bit integers A and B as:

$$a = a_0 + a_1 2 + a_2 2^2 + ... + a_{n-1} 2^{n-1}$$
$$b = b_0 + b_1 2 + b_2 2^2 + ... + b_{n-1} 2^{n-1}$$

Written in this form, these integers naturally evoke the polynomials:

$$A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1}$$

From these polynomials, we have $A(2) = a$, and $B(2) = b$. Thus, if we seek $c = ab$, we can instead use the FFT algorithm to determine $C(x) = A(x)B(x)$ in $O(n \log n)$ time, and then evaluate the result at $x = 2$ to get c.

In addition to being efficient, this algorithm extends naturally to integers with bases other than 2. For example, if a and b are decimal integers, then we can evaluate c using exactly the method outlined above, except we evaluate $C(10)$ instead of $C(2)$ to obtain our final answer.

5.6 The Skyline Problem

5.6.1 Problem Description

Consider a budget traveler looking to stay in a hotel by the beach. Naturally, hotels very near the beach tend to be more expensive than ones a bit further away. Our traveler is thus faced with the problem of finding the best hotel (i.e., the hotel closest to the beach) for a given cost.

If we plot the hotels' price vs. distance from the beach, it becomes evident that only the hotels towards the lower right need be considered (see Figure 5.2). Identifying these candidate hotels is an instance of what is known as the *skyline problem.*[4]

[4]Outside the field of data mining, the skyline problem is also sometimes known as the *maximal vector problem.*

More formally, given a set of points

$$S = \{p_1, ..., p_n\}$$

where

$$p_i = (p_i(x), p_i(y)) \in \Re^2$$

we say that a point p_j *dominates* p_k if:

1. $p_j(x) \geq p_k(x)$, and
2. $p_j(y) \geq p_k(y)$

If a point $p_j \in S$ does not dominate any point in $S - \{p_j\}$, then we say that p_j is a *skyline point* with respect to the set S. Thus, we can restate the skyline problem as the problem of finding all points of a set S which do not dominate any other point of S.

5.6.2 First Divide & Conquer Algorithm

A straightforward application of the divide and conquer methodology results in the following algorithm:

1. Divide S along the median $x-$coordinate into two equal halves, S_{left} and S_{right}.
2. Recursively solve the skyline problem for S_{left} and S_{right}. We can assume the skyline points are reported in sorted order, because of our use of the median as the dividing line (pivot).

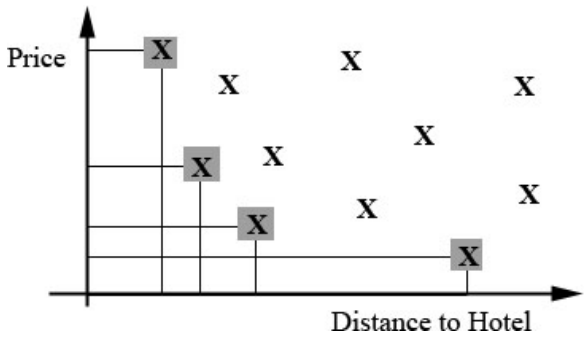


Figure 5.2. Plot of hotel price vs. distance to the hotel from the beach. Skyline points are indicated by boxes.

3. Determine the skyline points of S_{right} which don't dominate any points in S_{left}.
4. Output all skyline points of S_{left} and those maximal points of S_{right} identified in Step 3.

In the worst case, this relation has the following recurrence:

where $T(n) =$ time to find the maximal points of a set of n elements,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \in O(n \log n)$$

An example of a worst-case situation is when all points of S are skyline points. However, this is not the unique worst case for our algorithm; indeed, if all the points in S_{left} are maximal with respect to S, and if all the points in S_{right} are skyline points with respect to S_{right} but none with respect to S_{left}, we again achieve worst-case performance. The difference between these two worst cases is while the all the points identified in the former end up in the solution, in the latter all of the points we've worked to identify from S_{right} end up being discarded. Clearly there is a significant waste of computational effort occurring; this motivates our second attempt at an efficient solution.

5.6.3 Revised Divide & Conquer Algorithm

In order to minimize wasted computation, we observe that only those points in S_{right} which don't dominate points in S_{left} need be considered (Figure 5.3).

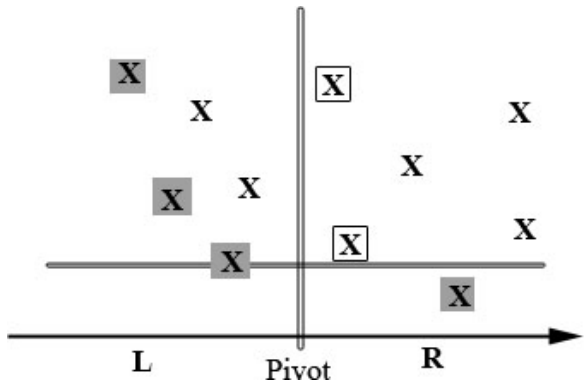


Figure 5.3. Illustration of recursion partitioning; only points in $R = S_{right}$ which do not dominate any point in $L = S_{left}$ need be considered.

Further, if we know the skyline points of S_{left} we can identify this candidate subset of S_{right} efficiently. This leads us to our second algorithm:

1. Divide S along the median $x-$coordinate into two equal halves, S_{left} and S_{right}.
2. Recursively find the skyline points of S_{left}. We assume these points are reported in sorted order; let r^* be the skyline point with the smallest y-value.
3. In $O(n)$ time, eliminate all points $p_j \in S_{right}$ which satisfy $p_j(y) \geq r^*(y)$. Let S'_{right} be the set that results from this operation.
4. Recursively find the skyline points of S'_{right}.
5. In linear time, combine and output all maximal points of S_{left} and S'_{right}.

This revised algorithm has the same recurrence, and thus the same worst-case running time, as our original algorithm. The unique worst case occurs when all points of S are skyline points. In all other cases, we expect to see an improvement in performance. To see this, we analyze the algorithm as a function of output size:

where $T(n,h) =$ time to find h skyline points of a set of n elements,

$$T(n,h) = T\left(\frac{n}{2}, l\right) + T\left(\frac{n}{2}, h-l\right) + O(n) \in O(n \log h)$$

Here, l represents the number of maximal points found in S_{left}.

We prove by induction the upper bound of $O(n \log h)$ given this recurrence. From the definition of big O:

$$\begin{aligned} T(n,h) &\leq c\frac{n}{2} \ln l + c\frac{n}{2} \ln(h-l) + bn \\ &\leq c\frac{n}{2} \left(\ln l + \ln(h-l)\right) + bn \end{aligned}$$

(Note that we are free to evaluate these logarithms at any base; a change will only affect the choice of constant c. We choose the natural logarithm $\ln = \log_e$ for convenience in the following step.)

Next, we find the value of l which maximizes the right hand side of this inequality by differentiating with respect to l and setting equal to zero:

$$\frac{\partial}{\partial l}\left(\ln l + \ln(h-l)\right) = \frac{1}{l} - \frac{1}{h-l} = 0$$
$$l = \frac{h}{2}$$

One can show that $l = \frac{h}{2}$ is a maximum of $\ln l - \ln(h-l)$, so then the worst case is:

$$\begin{aligned} T(n,h) &\leq c\frac{n}{2}\ln\frac{h}{2} + c\frac{n}{2}\ln\frac{h}{2} + bn \\ &= cn\ln h - cn\ln 2 + bn \\ &= cn\ln h - n(c\ln 2 - b) \\ &\leq cn\ln h, \text{ for } c > \frac{b}{\ln 2} \end{aligned}$$

Let's strengthen our intuition about the operation of this algorithm. Consider the case where $l = 1$. Clearly in this case, $T(n,1) \in O(n)$, since we only need to recurse on $O(\frac{n}{2})$ points at each iteration. In total, then, we recurse on $n + \frac{n}{2} + \frac{n}{4} + ... + 1 = 2n$ points, so the overall algorithm would be $O(n)$ in this case. This represents a best-case; as l increases, performance approaches the established $O(n \log h)$ bound.

5.6.4 Marriage Before Conquest

The main idea presented here is that one should be careful to avoid wasted effort when designing divide and conquer algorithms; one should not act blindly. If it is possible to take advantage of the recursive solution to one part of a problem in order to solve another, it is wise to do so. In order to distinguish from the standard divide and conquer technique, this variation is sometimes called *Marriage Before Conquest*.

5.7 The Closest Point Problem

This problem is discussed in the text. As such, coverage here is abridged.

5.7.1 Problem Description

Given a set of points

$$S = \{p_1, ..., p_n\}$$

where

$$p_i = (p_i(x), p_i(y)) \in \Re^2$$

and the usual distance metric

$$d(p, q) = \sqrt{(p(x) - q(x))^2 + (p(y) - q(y))^2}$$

we define the *Closest Point Problem* as the problem of finding the pair of points $s, s' \in S$ for which $d(s, s')$ is minimal. In other words, we search for the pair of points in S which are closest together.

5.7.2 Divide & Conquer Algorithm

Our algorithm follows general divide and conquer principles:

1. Partition S into two halves L and R, using the median x-coordinate as a pivot.

2. Recursively find the minimum pair in each of L and R. Take the minimum of these, and let $\delta = d$, the distance between this minimum pair.

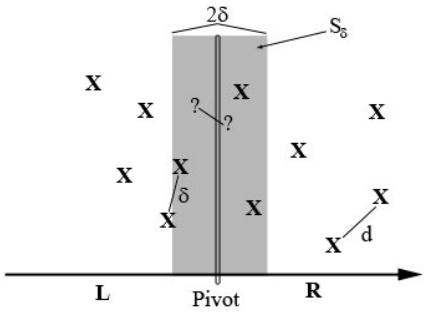


Figure 5.4. Illustration of recursion partitioning. δ represents the shortest distance found between 2 points wholly within L or R. The shaded region corresponds to S_δ, the subset of interest for Step 3 of the algorithm.

3. Determine whether there is a pairing of one point $s_L \in L$ and $s_R \in R$ with distance $d(s_L, s_R) < \delta$; if so, return them, otherwise, return the pair found in Step 2

Of these steps, 1 and 2 are straightforward, but Step 3 requires closer consideration. First, observe that if there is indeed a pair (s_L, s_R) with $d(s_L, s_R) < \delta$, then each of these points necessarily lies within δ of the pivot. Thus, we define $S_\delta \subseteq S$ as all points in S with x-coordinate within δ of the pivot line. L_δ and R_δ are simply the right and left partitions of S_δ, analogous to L and R with respect to S. This situation is illustrated in Figure 5.4.

We now need to search L_δ for s_L and R_δ for s_R. Let us examine S_δ more closely; in particular, we consider a $2\delta \times 2\delta$ slice of S_δ (Figure 5.5). Based on this, we make a key observation:

Claim There can be no more than one point per $\frac{\delta}{2} \times \frac{\delta}{2}$ square in S_δ, or 16 points per $2\delta \times 2\delta$ slice.

Corollary If s_L and $s_R \in S_\delta$ have the property that $d(s_L, s_R) < \delta$, then s_L and s_R are within 15 positions of each other in the y-sorted list of S_δ.

Inspection of Figure 5.5 should convince the reader that these are true. Given this, we see that for each point in S_δ we can perform 15 $d(s, s')$ calculations in constant time, thus finding the minimal pair in S_δ in linear time - *if* S_δ is y-sorted. Assuming this is possible, we obtain the recurrence relation:

$$T(n) = 2T(\frac{n}{2}) + O(n) \in O(n \log n)$$

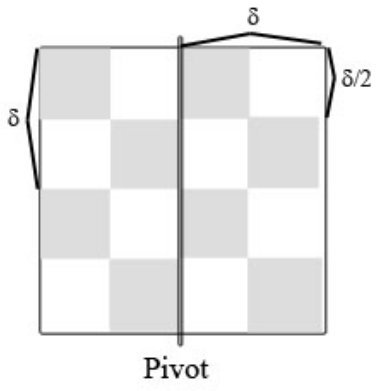


Figure 5.5. $2\delta \times 2\delta$ detail of S_δ. Each square can contain at most one point.

It remains to be shown how we can ensure access to a y-sorted list of points in S_δ, without exceeding the $O(n)$ bound of the merge step.

This problem has a simple solution. We take not one but two lists of points of S as parameters to our algorithm: S_x which is x-sorted, and S_y which is y-sorted. These can be produced in $O(n \log n)$ time via usual sorting techniques before the main algorithm commences.

Given S_x and S_y, the corresponding L_x, L_y, R_x, and R_y needed for recursive calls can be generated in linear time. L_x and R_x are obtained by partitioning, exactly as L and R were in the initial description of the algorithm. Similarly, L_y and R_y are obtained by scanning through S_y from top to bottom, and partitioning the points into two sets based on the x-coordinates. Since this partitioning preserves the relative ordering of elements in S_x and S_y, the resultant lists will also be x- and y-sorted.

We can now generate L_δ and R_δ in y-sorted order in linear time, simply by choosing only those points from L_y and R_y respectively which have x-coordinates within δ of the pivot. Finally, we obtain S_δ in y-sorted order by merging L_δ and R_δ in $O(n)$ time.

Thus, we have described an algorithm for solving the Closest Point Problem in $O(n \log n)$ time.