Chapter 2– Solutions

January 31, 2007

2.2. Consider $b^{\lceil \log_b n \rceil}$.

2.3. a)

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + cn = \cdots = 3^k T\left(\frac{n}{2^k}\right) + cn\sum_{i=0}^{k-1}\left(\frac{3}{2}\right)^i = \\ &= 3^k T\left(\frac{n}{2^k}\right) + 2cn\left(\left(\frac{3}{2}\right)^k - 1\right) \end{aligned}$$

For $k = \log_2 n$, $T(\frac{n}{2^k}) = T(1) = d = O(1)$. Then:

$$T(n) = dn^{log_2 3} + 2cn\left(\frac{n^{log_2 3}}{n} -\right) = \Theta(n^{log_2 3})$$

as predicted by the Master theorem.

b) $T(n) = T(n-1) + c = \cdots = T(n-k) + kc$. For $k = n$, $T(n) = T(0) + nc = \Theta(n)$.

2.4. a) This is a case of the Master theorem with $a = 5, b = 2, d = 1$. As $a > b^d$, the running time is $O(n^{\log_b a}) = O(n^{\log_2 5}) = O(n^{2.33})$.

b) $T(n) = 2T(n-1) + C$, for some constant C. $T(n)$ can then be expanded to $C\sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$.

c) This is a case of the Master theorem with $a = 9, b = 3, d = 2$. As $a = b^d$, the running time is $O(n^d \log n) = O(n^2 \log n)$.

2.5. a) $T(n) = 2T(n/3) + 1 = \Theta(n^{\log_3 2})$ by the Master theorem.

b) $T(n) = 5T(n/4) + n = \Theta(n^{\log_4 5})$ by the Master theorem.

c) $T(n) = 7T(n/7) + n = \Theta(n \log_7 n)$ by the Master theorem.

d) $T(n) = 9T(n/3) + n^2 = \Theta(n^2 \log_3 n)$ by the Master theorem.

e) $T(n) = 8T(n/2) + n^3 = \Theta(n^3 \log_2 n)$ by the Master theorem.

f) $T(n) = 49T(n/25) + n^{3/2}\log n = \Theta(n^{3/2}\log n)$. Apply the same reasoning of the proof of the Master Theorem. The contribution of level i of the recursion is

$$\left(\frac{49}{25^{3/2}}\right)^i n^{3/2} \log\left(\frac{n}{25^{3/2}}\right) = \left(\frac{49}{125}\right)^i O(n^{3/2}\log n)$$

Because the corresponding geometric series is dominated by the contribution of the first level, we obtain $T(n) = O(n^{3/2}\log n)$. But, $T(n)$ is clearly $\Omega(n^{3/2}\log n)$. Hence, $T(n) = \Theta(n^{3/2}\log n)$.

g) $T(n) = T(n-1) + 2 = \Theta(n)$.

h) $T(n) = T(n-1) + n^c = \sum_{i=0}^{n} i^c + T(0) = \Theta(n^{c+1})$.

i) $T(n) = T(n-1) + c^n = \sum_{i=0}^{n} c^i + T(0) = \frac{c^{n+1}-1}{c-1} + T(0) = \Theta(c^n)$.

j) $T(n) = 2T(n-1) + 1 = \sum_{i=0}^{n-1} 2^i + 2^n T(0) = \Theta(2^n)$.

k) $T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$, where $k \in Z$ such that $n^{\frac{1}{2^k}}$ is a small constant b, i.e. the size of the base case. This implies $k = \Theta(\log\log n)$ and $T(n) = \Theta(\log\log n)$.

2.6. The corresponding polynomial is $\frac{1}{t_0}\left(1 + x + x^2 + \cdots + x^{t_0}\right)$.

2.7. For $n \neq 0$ and $\omega = e^{\frac{2\pi}{n}}$:

$$\sum_{i=0}^{n-1} \omega^i = \frac{1-\omega^n}{1-\omega} = 0$$

$$\prod_{i=0}^{n-1} \omega^i = \omega^{\sum_{i=0}^{n-1} i} = \omega^{\frac{n(n-1)}{2}}$$

The latter is 1 if n is odd and -1 if n is even.

2.8. a) The appropriate value of ω is i. We have $\mathrm{FFT}(1,0,0,0) = (1,1,1,1)$ and $\mathrm{FFT}(1/4,1/4,1/4,1/4) = (1,0,0,0)$.

b) $\mathrm{FFT}(1,0,1,-1) = (1,i,3,-i)$.

2.9. a) Use $\omega = i$. The FFT of $x+1$ is $\mathrm{FFT}(1,1,0,0) = (2, i+1, 0, i-1)$. The FFT of x^2+1 is $\mathrm{FFT}(1,0,1,0) = (2,0,2,0)$. Hence, the FFT of their product is $(4,0,0,0)$, corresponding to the polynomial $1+x+x^2+x^3$.

b) Use $\omega = i$. The FFT of $2x^2+x+1$ is $\mathrm{FFT}(1,1,2,0) = (4, -1+i, 2, -1-i)$. The FFT of $3x+2$ is $\mathrm{FFT}(2,3,0,0) = (5, 2+3i, -1, 2-3i)$. The FFT of their product is then $(20, -5-i, -2, -5+i)$. This corresponds to the polynomial $6x^3+7x^2+5x+2$.

2.10. $\left(-20, \frac{137}{3}, -\frac{125}{4}, \frac{25}{3}, -\frac{3}{4}\right)$.

2.11. For $i, j \leq \frac{n}{2}$:

$$(XY)_{ij} \quad = \quad \sum_{k=1}^{n} X_{ik}Y_{kj} = \sum_{k=1}^{n/2} A_{ik}E_{kj} + \sum_{k=1}^{n/2} B_{ik}G_{ki} = (AE+BG)_{ij}$$

A similar proof holds for the remaining sectors of the product matrix.

2.12. Each function call prints one line and calls the same function on input of half the size, so the number of printed lines is $P(n) = 2P(\frac{n}{2}) + 1$. By the Master theorem $P(n) = \Theta(n)$.

2.13. a) $B_3 = 1$, $B_5 = 2$, $B_7 = 5$. Any full binary tree must have an odd number of vertices, as it has an even number of vertices that are children of some other vertex and a single root. Hence $B_{2k} = 0$ for all k.

b) By decomposing the tree into the subtrees rooted at the children of the root:

$$B_{n+2} = \sum_{i=1}^{n+1} B_i B_{n+1-i}$$

c) We show that $B_n \geq 2^{\frac{n-3}{2}}$. Base case: $B_1 = 1 \geq 2^{-1}$ and $B_3 = 1 \geq 2^0$. Inductive step: for $n \geq 3$ odd:

$$B_{n+2} = \sum_{i=1}^{n+1} B_i B_{n+1-i} \geq \frac{n+2}{2} 2^{\frac{n-5}{2}} \geq 2^{2+\frac{n-5}{2}} = 2^{\frac{n-1}{2}}$$

.

2.14. Sort the array in time $O(n \log n)$. Then, in one linear time pass copy the elements to a new array, eliminating the duplicates.

2.15. The simplest way to implement `split` in place is the following:

```
function split($a[1,\cdots,n], v$)
store= $1$
for $i = 1$ to $n$:
    if $a[i] < v$:
        swap $a[i]$ and $a$[store]
        store = store $+1$
for $i =$store to $n$:
    if $a[i] = v$:
        swap $a[i]$ and $a$[store]
        store = store $+1$
```

The first for loop passes through the array bringing the elements smaller than v to the front, so splitting the array into a subarray of elements smaller than v and one of elements larger or equal to v. The second for loop uses the same strategy on the latter subarray to split into a subarray of elements equal to v and one of elements larger than v. The body of both for loops takes constant time, so the running time is $O(n)$.

2.16. It is sufficient to show how to find n in time $O(\log n)$, as we can use binary search on the array $A[1,\cdots,n]$ to find any element x in time $O(\log n)$. To find n, query elements $A[1], A[2], \cdots, A[2^i], \cdots$, until you find the first element $A[2^k]$ such that $A[2^k] = \infty$. Then, $2^{k-1} \leq n < 2^k$. We can then do binary search on $A[2^{k-1}, \cdots,, 2^k]$ to find the last non-infinite element of A. This takes time $O(\log(2^k - 2^{k-1}) = O(\log n)$.

2.17. First examine the middle element $A[\frac{n}{2}]$. If $A[\frac{n}{2}] = \frac{n}{2}$, we are done. If $A[\frac{n}{2}] > \frac{n}{2}$, then every subsequent element will also be bigger than its index since the array values grow at least as fast as the indices. Similarly, if $A[\frac{n}{2}] < \frac{n}{2}$, then every previous element in the array will be less than its index by the same reasoning. So after the comparison, we only need to examine half of the array. We can recurse on the appropriate half of the array. If we continue this division until we get down to a single element and this element does not have the desired property, then such an element does not exist in A. We do a constant amount of work with each function call. So our recurrence relation is $T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

2.18. As in the $\Omega(n \log n)$ lower bound for sorting on page 52 we can look at a comparison-based algorithm for search in a sorted array as a binary tree in which a path from the root to a leaf represents a run of the algorithm: at every node a comparison takes place and, according to its result, a new comparison is performed. A leaf of the tree represents an output of the algorithm, i.e. the index of the element x that we are searching or a special value indicating the element x does not appear in the array. Now, all possible indices must appear as leaves or the algorithm will fail when x is at one of the missing indices. Hence, the tree must have at least n leaves, implying its depth must be $\Omega(\log n)$, i.e. in the worst case it must perform at least $\Omega(\log n)$ comparisons.

2.19. a) Let $T(i)$ be the time to merge arrays 1 to i. This consists of the time taken to merge arrays 1 to $i-1$ and the time taken to merge the resulting array of size $(i-1)n$ with array i, i.e. $O(in)$. Hence, for some constant c, $T(i) \leq T(i-1) + cni$. This implies $T(k) \leq T_1 + cn \sum_{i=2}^{k} i = O(nk^2)$.

b) Divide the arrays into two sets, each of $k/2$ arrays. Recursively merge the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is $k = 1$, when no merging needs to take place. The running time is given by $T(k) = 2T(k/2) + O(nk)$. By the Master theorem, $T(k) = O(nk \log k)$.

2.20. We keep $M + 1$ counters, one for each of the possible values of the array elements. We can use these counters to compute the number of elements of each value by a single $O(n)$-time pass through the array. Then, we can obtain a sorted version of x by filling a new array with the prescribed numbers of elements of each value, looping through the values in ascending order. Notice that the $\Omega(n \log n)$ bound does not apply in this case, as this algorithm is not comparison-based.

2.21. a) Suppose $\mu > \mu_1$. If we move μ to the left by an infinitesimal amount ϵ, the distance to all $x_i < \mu$ decreases by ϵ, whilst the distance to all points $x_i > \mu$ increases by ϵ. Because $\mu > \mu_1$, there are more points to the left of μ than to the right, so that the the shift by ϵ causes the sum of distances to decrease. The same reasoning can be applied to the case $\mu < \mu_1$.

b) We use the second method:

$$\sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2 =$$
$$= \sum_i x_i^2 - 2n\mu_2^2 + n\mu_2^2 + n\mu^2 - 2n\mu\mu_2 + n\mu_2^2 =$$
$$= \sum_i x_i^2 - 2(\sum_i x_i)\mu + n\mu^2 =$$
$$= \sum_i (x_i - \mu)^2$$

Then, $\sum_i (x_i - \mu)^2$ will be minimized when the second addend of the left hand side is minimized, i.e. for $\mu = \mu_2$.

c) The maximizing x_i in $\max_i |x_i - \mu|$ will be the maximum $x_{\max}$ or the minimum $x_{\min}$ of the observations. Then μ must minimize $\max\{x_{\max} - \mu, \mu - x_{\min}\}$, so $\mu = \frac{x_{\min}+x_{\max}}{2}$. This value can be computed in $O(n)$ time by passing through the observations to identify the minimum and maximum elements and taking their average in constant time.

2.22. Suppose we are searching for the kth smallest element s_k in the union of the lists $a[1, \cdots, m]$ and $b[1, \cdots, n]$. Because we are searching for the kth smallest element, we can restrict our attention to the arrays $a[1, \cdots, k]$ and $b[1, \cdots, k]$. If $k > m$ or $k > n$, we can take all the elements with index larger than the array boundary to have infinite value. Our algorithm starts off by comparing elements $a[\lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil]$. Suppose $a[\lfloor k/2 \rfloor] > b[\lceil k/2 \rceil]$. Then, in the union of a and b there can be at most $k - 2$ elements smaller than $b[\lceil k/2 \rceil]$, i.e. $a[1, \cdots, \lfloor k/2 \rfloor - 1]$ and $b[1, \cdots, \lceil k/2 \rceil - 1]$, and we must necessarily have $s_k > b[\lceil k/2 \rceil]$. Similarly, all elements $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[1, \cdots, \lceil k/2 \rceil]$ will be smaller than $a[\lfloor k/2 \rfloor + 1]$; but these are k elements, so we must have $s_k < a[\lfloor k/2 \rfloor + 1]$. This shows that s_k must be contained in the union of the subarrays $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil + 1, k]$. In particular, because we discarded $\lceil k/2 \rceil$ elements smaller than s_k, s_k will be the $\lfloor k/2 \rfloor$th smallest element in this union. We can then find s_k by recursing on this smaller problem. The case for $a[\lfloor k/2 \rfloor] < b[\lceil k/2 \rceil]$ is symmetric. The last case, which is also the base case of the recursion, is $a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$, for which we have $s_k = a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$.

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(2k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $O(\log k)$, which is $O(\log(m + n))$, as we must have $k \leq m + n$ for the kth smallest element to exist.

2.23. a) If A has a majority element v, v must also be a majority element of A_1 or A_2 or both. To find v, recursively compute the majority elements, if any, of A_1 and A_2 and check whether one of these is a majority element of A. The running time is given by $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

b) After this procedure, there are at most $n/2$ elements left as at least one element in each pair is discarded. If these remaining elements have a majority, there exists v among them appearing at least $n/4$ times. Hence, v must have been paired up with itself in at least $n/4$ pairs during the procedure, showing that A contains at least $n/2$ copies of v. The running time of this algorithm is described by the recursion $T(n) = T(n/2) + O(n)$. Hence, $T(n) = O(n)$.

2.24. a) Modify the in place split procedure of 2.15 so that it explicitly returns the three subarrays S_L,S_R,S_v. Quicksort can then be implemented as follows:

```
function quicksort($A[1,\cdots,n]$)
    pick $k$ at random among $1,\cdots,n$
    ($S_L, S_R, S_v$ =split($A[1,\cdots,n], A[k]$)
    quicksort($S_L$)
    quicksort($S_R$)
```

b) In the worst case we always pick $A[k]$ that is the largest element of A. Then, we only decrease the problem size by 1 and the running time becomes $T(n) = T(n-1) + O(n)$, which implies $T(n) = O(n^2)$.

c) For $1 \leq i \leq n$, let p_i be the probability that $A[k]$ is the ith largest element in A and let t_i be the expected running time of the algorithm in this case. Then, the expected running time can be expressed as $T(n) = \sum_{i=1}^{n} p_i t_i$. $A[k]$ is every element of A with the same probability $\frac{1}{n}$, so $p_i = \frac{1}{n}$. Moreover, t_i is at most $O(n) + T(n-i+1) + T(i-1)$, as S_L has at most $n-i+1$ elements and S_R has at most $i-1$. Then, for some constant c:

$$T(n) \leq \frac{1}{n}\left(\sum_{i=1}^{n} O(n) + T(i-1) + T(n-i+1)\right) \leq cn + \frac{1}{n}\sum_{j=0}^{n-1} T(j) + T(n-j)$$

We use induction to show that the recurrence is satisfied by $T(n) \leq bn\log n$ for some choice of constant b. We start by rewriting $T(n)$ as:

$$T(n) \leq cn + \frac{2}{n}\sum_{j=0}^{\frac{n}{2}} T(j) + T(n-j)$$

Then, we substitute the expression from the inductive hypothesis:

$$T(n) \leq cn + \frac{2b}{n}\sum_{j=1}^{\frac{n}{2}} j\log j + (n-j)\log(n-j)$$

Next, we divide the sum as follows:

$$T(n) \leq cn + \frac{2b}{n}\sum_{j=1}^{\frac{n}{4}-1} j\log j + (n-j)\log(n-j) + \frac{2b}{n}\sum_{j=\frac{n}{4}}^{\frac{n}{2}} j\log j + (n-j)\log(n-j)$$

Finally, we can bound each term in the first sum above by $n\log n$ and each term in the second sum by $n\log(\frac{3}{4}n)$:

$$T(n) \leq cn + \frac{b}{2}n\log n + \frac{b}{2}n\log n - \frac{b}{2}n\log\left(\frac{4}{3}\right) = bn\log n + n\left(c - \frac{b}{2}\log\left(\frac{4}{3}\right)\right)$$

This is smaller than $bn\log n$ for $b > \frac{2c}{\log(\frac{4}{3})}$, completing the proof by induction.

2.25 a) The algorithm should be:

```
function pwr2bin($n$)
if $n = 1$: return $1010_2$
else:
    $z$ =pwr2bin($n/2$)
```

```
        return fastmultiply($z, z$)
```

The running time $T(n)$ can then be written as

$$T(n) = T(n/2) + O(n^a)$$

By the Master theorem, $T(n) = O(n^a)$.

b) The algorithm is the following:

```
dec2bin($x$)
if $n = 1$: return binary[$x$]
else:
    split $x$ into two decimal numbers $x_L$, $x_R$ with $n/2$ digits each
    return {fastmultiply(dec2bin($x_L$),pwr2bin($n/2$)) + dec2bin($x_R$)}
```

The running time $T(n)$ is expressed by the recurrence relation

$$T(n) = 2T(n/2) + O(n^a)$$

as both the `fastmultiply` and the `pwr2bin` operations take time $O(n^a)$. By the Master theorem, $T(n) = O(n^a)$, as $a = \log_2 3 > 1$.

2.26 We show how to use an algorithm for squaring integers to multiply integers asymptotically as fast. Let $S(n)$ be the time required to square a n-bit number. We can multiply n-bit integers a and b by first computing $2ab = (a+b)^2 - a^2 - b^2$ and then shifting the results $2ab$ by one bit to the right to obtain ab. This algorithm takes 3 squaring operations and 3 additions and hence has running time $3S(n) + O(n)$. But $S(n) = \Omega(n)$, as any squaring algorithm must at least read the n-bit input. This implies the running time of the multiplication algorithm above is $\Theta(S(n))$, contradicting Professor Lake's claim.

2.27 a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{bmatrix}$$

Hence the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

b) We do get 5 subproblems but they are *not of the same type as the original problem.* Note that we started with a squaring problem for a matrix of size $n \times n$ and three of the 5 subproblems now involve *multiplying* $n/2 \times n/2$ matrices. Hence the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

It now suffices to compute A^2, as its upper left block will contain XY:

$$A = \begin{bmatrix} XY & 0 \\ 0 & XY \end{bmatrix}$$

Hence, the product XY can be calculated in time $O(S(2n))$. If $S(n) = O(n^c)$, this is also $O(n^c)$.

2.28 For any column vector u of length n, let $u^{(1)}$ denote the column vector of length $n/2$ consisting of the first $n/2$ coordinates of u. Similarly, define $u^{(2)}$ to be the vector of the remaining coordinates. Note then that $(H_k v)^{(1)} = H_{k-1}v^{(1)} + H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} + v^{(2)})$ and $(H_k v)^{(2)} = H_{k-1}v^{(1)} - H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} - v^{(2)})$. This shows that we can find $H_k v$ by calculating $(v^{(1)} + v^{(2)})$ and $(v^{(1)} - v^{(2)})$ and recursively computing $H_{k-1}(v^{(1)} + v^{(2)})$ and $H_{k-1}(v^{(1)} - v^{(2)})$. The running time of this algorithm is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n)$, where the linear term is the time taken to perform the two sums. This has solution $T(n) = O(n \log n)$ by the Master theorem.

2.29 a) We show this by induction on n, the degree of the polynomial. For $n = 1$, the routine clearly works as the body of the loop itself evaluates p at x. For $n = k + 1$, notice that the first k iterations of the for loop evaluate the polynomial $q(x) = a_1 + a_2x + a_3x^2 + \cdots a_nx^{n-1}$ at x by the inductive hypothesis. The last iteration of the for loop evaluates then $xq(x) + a_0 = p(x)$ at point x, as required.

b) Every iteration of the for loop uses one multiplication and one addition, so the routine uses n additions and n multiplications. Consider now the polynomial $p(x) = x^n$ for $n = 2^k$. p can be evaluated at x using only $k = \log n$ multiplication simply by repeatedly squaring x. However, Horner's rule still takes n multiplications.

2.30 (a) Observe that taking $\omega = 3$ produces the following powers : $(\omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6) = (3, 2, 6, 4, 5, 1)$. Verify that

$$\omega + \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6 = 1 + 2 + 3 + 4 + 5 + 6 = 21 = 0\ (mod\ 7)$$

(b) The matrix $M_6(3)$ is the following:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{bmatrix}$$

Multiplying with the sequence $(0, 1, 1, 1, 5, 2)$ we get the vector $(3, 6, 4, 2, 3, 3)$.

(c) The inverse matrix of $M_6(3)$ is easily seen to be the matrix

$$6 \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 5 & 4 & 6 & 2 & 3 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 3 & 2 & 6 & 4 & 5 \end{bmatrix}$$

Verify that multiplying these two matrices mod 7 equals the identity. Also multiply this matrix with vector $(3, 6, 4, 2, 3, 3)$ to get the original sequence.

(d) We first express the polynomials as vectors of dimension 6 over the integers mod 7: $(1, 1, 1, 0, 0, 0)$, and $(-1, 2, 0, 1, 0, 0) = (6, 2, 0, 1, 0, 0)$ respectively. We then apply the matrix $M_6(3)$ to both to get the transform of the two sequences. That produces $(3, 6, 0, 1, 0, 3)$ and $(2, 4, 4, 3, 1, 1)$ respectively. Then we just multiply the vectors coordinate-wise to get $(6, 3, 0, 3, 0, 3)$. This is the transform of the product of the two polynomials. Now, all we have to do is multiply by the inverse FT matrix $M_6(3)^{-1}$ to get the final polynomial in the coefficient representation : $(-1, 1, 1, 3, 1, 1)$.

2.31 a) We prove each statement of the rule separately:

- $*$ If a and b are both even, then $\gcd(a, b) = 2\gcd(a/2, b/2)$.
  Proof: $\gcd(a, b) = d$ iff $a = da'$ and $b = db'$ and $\gcd(a', b') = 1$. Since a and b are both even, $2|d$ so $a/2 = a'd/2$ and $b/2 = b'd/2$, and since $\gcd(a', b') = 1$, then $\gcd(a/2, b/2) = d/2$.
- $*$ If a is even and b is odd, then $\gcd(a, b) = \gcd(a/2, b)$.
  Proof: Since b is odd, $\gcd(a, b) = d$ is odd. If an odd $d|a$ then $d|a/2$.
- $*$ If a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
  Proof: On page 20 we proved that for $a \geq b$, $\gcd(a, b) = \gcd(a - b, b)$. Now since $|a - b|$ is even if both a and b are odd, we can just use part 2 to conclude that $\gcd(a, b) = \gcd((a - b)/2, b)$.

b) Consider the following algorithm:

```
function gcd($a, b$)
if $b == 0$:
    return $a$
if $a$ is even:
    if $b$ is even:
        return $2\cdot$ gcd($a/2, b/2$)
    else:
        return gcd($b, a/2$)
else
                    return gcd($b, (a-b)/2$)
```

The algorithm is correct according to part a) of this problem. Each argument is reduced by half after at most two calls to the function, so the function is called at most $O(\log a + \log b)$ times. Each function call takes constant time, assuming the subtraction in the last line takes constant time, so the running time is $O(\log a + \log b)$.

c) More realistically, if we take the subtraction to take time $O(n)$, we have that each call takes at most time $O(n)$ and there are at most $O(\log a + \log b) = O(n)$ calls. So, the running time is $O(n^2)$, which compares favorably to the $O(n^3)$ algorithm of Chapter 1.

2.32 a) Suppose 5 or more points in L are found in a square of size $d \times d$. Divide the square into 4 smaller squares of size $\frac{d}{2} \times \frac{d}{2}$. At least one pair of points must fall within the same smaller square: these two points will then be at distance at most $\frac{d}{\sqrt{2}} < d$, which contradicts the assumption that every pair of points in L is at distance at least d.

b) The proof is by induction on the number of points. The algorithm is trivially correct for two points, so we may turn to the inductive step. Suppose we have n points and let (p_s, p_t) be the closest pair. There are three cases.
If $p_s, p_t \in L$, then $(p_s, p_t) = (p_L, q_L)$ by the inductive hypothesis and all the other pairs tested by the algorithm are at a larger distance apart, so the algorithm will correctly output (p_s, p_t). The same reasoning holds if $p_s, p_t \in R$.
If $p_s \in L$ and $p_t \in R$, the algorithm will be correct as long as it tests the distance between p_s and p_t. Because p_s and p_t are at distance smaller than d, they will belong to the strip of points with x-coordinate in $[x-d, x+d]$. Suppose that $y_s \leq y_t$. A symmetric construction applies in the other case. Consider the rectangle S with vertices $(x-d, y_s), (x-d, y_s+d), (x+d, y_s+d), (x+d, y_s)$. Notice that both p_s and p_t must be contained in S. Moreover, the intersection of S with L is a square of size $d \times d$, which, by a), can contain at most 4 points, including p_s. Similarly, the intersection of S with R can also contain at most 4 points, including p_t. Because the algorithm checks the distance between p_s and the 7 points following p_s in the y-sorted list of points in the middle strip, it will check the distance between p_s and all the points of S. In particular, it will check the distance between p_s and p_t, as required for the correctness of the algorithm.

c) When called on input of n points this algorithm first computes the median x value in $O(n)$ and then splits the list of points into those belonging to L and R, which also takes time $O(n)$. Then the algorithm can recurse on these two subproblems, each over $n/2$ points. Once these have been solved the algorithm sorts the points in the middle strip by y coordinate, which takes time $O(n \log n)$ and then computes $O(n)$ distances, each of which can be calculated in constant time. Hence the running time is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. This can be analyzed as in the proof of the Master theorem. The kth level of the recursion tree will contribute

$t_k = 2^k \frac{n}{2^k}(\log n - k)$. Hence, the total running time will be:

$$\sum_{k=0}^{\log n} t_k = n\log^2 n - n\sum_{k=0}^{\log n} k \leq n\log^2 n - \frac{n}{2}\log^2 n = O(n\log^2 n)$$

.

d) We can save some time by sorting the points by y-coordinate only once and making sure that the split routine is implemented as not to modify the order by y when splitting by x. Sorting takes time $O(n \log n)$, while the time required by the remaining of the algorithm is now described by the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$, which yields $T(n) = O(n \log n)$. Hence, the overall running time is $O(n \log n)$.

2.33 a) Let $M_{ij} \neq 0$. It is sufficient to bound above $\Pr[(Mv)_i = 0]$, as $\Pr[(Mv)_i = 0] \geq \Pr[Mv = 0]$. Now, $(Mv)_i = \sum_{t=1}^n M_{it}v_t$, so $(Mv)_i$ is 0 if and only if:

$$M_{ij}v_j = \sum_{\substack{t=1 \\ t\neq j}}^{n} M_{it}v_t$$

Consider now any fixed assignment of values to all the v_t's but v_j. If, under this assignment, the right hand side is 0, v_j has to be 0. Similarly, if the right hand side is non-zero, v_j cannot be 0. In either case, v_j can only take one of the two values $\{0, 1\}$ and $Pr[(Mv)_i = 0] \leq \frac{1}{2}$.

b) If $AB \neq C$, the difference $M = AB - C$ is a non-zero matrix and, by part a):

$$\Pr[ABv = Cv] = \Pr[Mv = 0] \leq \frac{1}{2}$$

The randomized test for checking whether $AB = C$ is to compare ABv with Cv for a random v constructed as in *a*). To compute ABv, it is possible to use the associativity of matrix multiplication and compute first Bv and then $A(Bv)$. This algorithm performs 3 matrix-vector multiplications, each of which takes time $O(n^2)$, while the final comparison takes time $O(n)$. Hence, the total running time of the randomized algorithm is $O(n^2)$.

2.34 A linear-time algorithm requires dynamic programming. Here we give a divide-and-conquer algorithm running in polynomial time in n.

Let ϕ be the input 3SAT formula exhibiting the special local property. For $1 \leq i \leq j \leq n$, let $\phi(i, j)$ be the 3SAT formula consisting of clauses of ϕ containing only variables $x_i, x_{i+1}, \cdots, x_j$. We can then split ϕ into $\phi(1, \frac{n}{2}$, $\phi(\frac{n}{2} + 1, n)$ and the formula ψ consisting of the remaining clauses, i.e. those containing at least one variable with index $k \leq \frac{n}{2}$ and one with index $k > \frac{n}{2}$. The local property implies that ψ can only contain variables with index in the range $\{\frac{n}{2} - 9, \cdots, \frac{n}{2} + 10\}$. Hence, at most 20 variables appear in ψ. Suppose now we are given an assignment a to the variable of ψ that satisfies ψ.. To check if a can be extended to a satisfying assignment for the whole of ϕ, we can substitute the values of a for their corresponding variables within $\phi(1, \frac{n}{2}$ and $\phi(\frac{n}{2} + 1, n)$ and consider the satisfiability of these. Letting $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2}+1, n)$ be these new formulae, notice that ϕ will be satisfiable by an extension to a if and only if $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2} + 1, n)$ are satisfiable, as the latter have no variables in common. Moreover, both $\phi_a(1, \frac{n}{2}$ and $\phi_a(\frac{n}{2} + 1, n)$ exhibit the same local property as ϕ, so we can recurse on them to verify their satisfiability. We can apply this procedure over all choices of a satisfying ψ to check if any assignment satisfies ϕ. The running time of this algorithm is described by the recursion $T(n) \leq 2^{20} \cdot 2T(n/2) + O(n) = O(n^21)$, as there are at most 2^{20} assignments a, each of which gives rise to two subproblems.