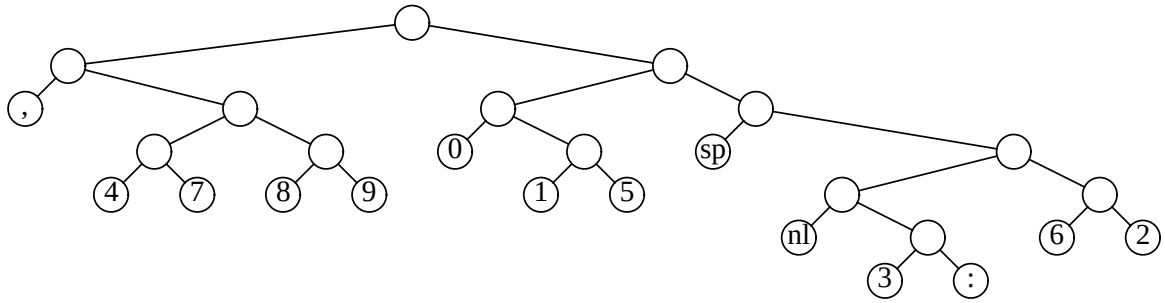
Chapter 10: Algorithm Design Techniques

10.1 First, we show that if N evenly divides P, then each of $j_{(i-1)P+1}$ through j_{iP} must be placed as the i^{th} job on some processor. Suppose otherwise. Then in the supposed optimal ordering, we must be able to find some jobs j_x and j_y such that j_x is the t^{th} job on some processor and j_y is the $t+1^{th}$ job on some processor but $t_x > t_y$. Let j_z be the job immediately following j_x. If we swap j_y and j_z, it is easy to check that the mean processing time is unchanged and thus still optimal. But now j_y follows j_x, which is impossible because we know that the jobs on any processor must be in sorted order from the results of the one processor case.

Let j_{e1}, j_{e2}, ..., j_{eM} be the extra jobs if N does not evenly divide P. It is easy to see that the processing time for these jobs depends only on how quickly they can be scheduled, and that they must be the last scheduled job on some processor. It is easy to see that the first M processors must have jobs $j_{(i-1)P+1}$ through j_{iP+M}; we leave the details to the reader.

10.3



10.4 One method is to generate code that can be evaluated by a stack machine. The two operations are *Push* (the one node tree corresponding to) a symbol onto a stack and *Combine*, which pops two trees off the stack, merges them, and pushes the result back on. For the example in the text, the stack instructions are *Push(s), Push(nl), Combine, Push(t), Combine, Push(a), Combine, Push(e), Combine, Push(i), Push (sp), Combine, Combine.*

By encoding a *Combine* with a 0 and a *Push* with a 1 followed by the symbol, the total extra space is $2N - 1$ bits if all the symbols are of equal length. Generating the stack machine code can be done with a simple recursive procedure and is left to the reader.

10.6 Maintain two queues, Q_1 and Q_2. Q_1 will store single-node trees in sorted order, and Q_2 will store multinode trees in sorted order. Place the initial single-node trees on Q_1, enqueueing the smallest weight tree first. Initially, Q_2 is empty. Examine the first two entries of each of Q_1 and Q_2, and dequeue the two smallest. (This requires an easily implemented extension to the ADT.) Merge the tree and place the result at the end of Q_2. Continue this step until Q_1 is empty and only one tree is left in Q_2.

10.9 To implement first fit, we keep track of bins b_i, which have more room than any of the lower numbered bins. A theoretically easy way to do this is to maintain a splay tree ordered by empty space. To insert w, we find the smallest of these bins, which has at least w empty space; after w is added to the bin, if the resulting amount of empty space is less than the inorder predecessor in the tree, the entry can be removed; otherwise, a *DecreaseKey* is performed.

To implement best fit, we need to keep track of the amount of empty space in each bin. As before, a splay tree can keep track of this. To insert an item of size w, perform an insert of w. If there is a bin that can fit the item exactly, the insert will detect it and splay it to the root; the item can be added and the root deleted. Otherwise, the insert has placed w at the root (which eventually needs to be removed). We find the minimum element M in the right subtree, which brings M to the right subtree's root, attach the left subtree to M, and delete w. We then perform an easily implemented *DecreaseKey* on M to reflect the fact that the bin is less empty.

10.10 *Next fit: 12 bins* (.42, .25, .27), (.07, .72), (.86, .09), (.44, .50), (.68), (.73), (.31), (.78, .17), (.79), (.37), (.73, .23), (.30).

First fit: 10 bins (.42, .25, .27), (.07, .72, .09), (.86), (.44, .50), (.68, .31), (.73, .17), (.78), (.79), (.37, .23, .30), (.73).

Best fit: 10 bins (.42, .25, .27), (.07, .72, .09), (.86), (.44, .50), (.68, .31), (.73, .23), (.78, .17), (.79), (.37, .30), (.73).

First fit decreasing: 10 bins (.86, .09), (.79, .17), (.78, .07), (.73, .27), (.73, .25), (.72, .23), (.68, .31), (.50, .44), (.42, .37), (.30).

Best fit decreasing: 10 bins (.86, .09), (.79, .17), (.78), (.73, .27), (.73, .25), (.72, .23), (.68, .31), (.50, .44), (.42, .37, .07), (.30).

Note that use of 10 bins is optimal.

10.12 We prove the second case, leaving the first and third (which give the same results as Theorem 10.6) to the reader. Observe that

$$\log^p N = \log^p (b^m) = m^p \log^p b$$

Working this through, Equation (10.9) becomes

$$T(N) = T(b^m) = a^m \sum_{i=0}^{m} \left\{ \frac{b^k}{a} \right\}^i i^p \log^p b$$

If $a = b^k$, then

$$\begin{aligned} T(N) &= a^m \log^p b \sum_{i=0}^{m} i^p \\ &= O(a^m m^{p+1} \log^p b) \end{aligned}$$

Since $m = \log N / \log b$, and $a^m = N^k$, and b is a constant, we obtain

$$T(N) = O(N^k \log^{p+1} N)$$

10.13 The easiest way to prove this is by an induction argument.

10.14 Divide the unit square into $N-1$ square grids each with side $1/\sqrt{N-1}$. Since there are N points, some grid must contain two points. Thus the shortest distance is conservatively given by at most $\sqrt{2}/(N-1)$.

10.15 The results of the previous exercise imply that the width of the strip is $O(1/\sqrt{N})$. Because the width of the strip is $O(1/\sqrt{N})$, and thus covers only $O(1/\sqrt{N})$ of the area of the square, we expect a similar fraction of the points to fall in the strip. Thus only $O(N/\sqrt{N})$ points are expected in the strip; this is $O(\sqrt{N})$.

10.17 The recurrence works out to

$$T(N) = T(2N/3) + T(N/3) + O(N)$$

This is not linear, because the sum is not less than one. The running time is $O(N \log N)$.

10.18 The recurrence for median-of-median-of-seven partitioning is

$$T(N) = T(5N/7) + T(N/7) + O(N)$$

If all we are concerned about is the linearity, then median-of-median-of-seven can be used.

10.20 When computing the median-of-median-of-five, 30% of the elements are known to be smaller than the pivot, and 30% are known to be larger. Thus these elements do not need to be involved in the partitioning phase. (Extra work would need to be done to implement this, but since the whole algorithm isn't practical anyway, we can ignore any extra work that doesn't involve element comparisons.) The original paper [9] describes the exact constants in the worst-case bound, with and without this extra effort.

10.21 We derive the values of s and δ, following the style in the original paper [17]. Let $R_{t,X}$ be the rank of element t in some sample X. If a sample S' of elements is chosen randomly from S, and $|S'| = s$, $|S| = N$, then we've already seen that

$$E(R_{t,S}) = \frac{N+1}{s+1} R_{t,S'}$$

where E means expected value. For instance, if t is the third largest in a sample of 5 elements, then in a group of 19 elements it is expected to be the tenth largest. We can also calculate the variance:

$$V(R_{t,S}) = \sqrt{\frac{(R_{t,S})(s-R_{t,S}+1)(N+1)(N-s)}{(s+1)^2(s+2)}} = O(N/\sqrt{s})$$

We choose v_1 and v_2 so that

$$E(R_{v_1,S}) + 2dV(R_{v_1,S}) \approx k \approx E(R_{v_2,S}) - 2dV(R_{v_2,S})$$

where d indicates how many variances we allow. (The larger d is, the less likely the element we are looking for will not be in S'.)

The probability that k is not between v_1 and v_2 is

$$2\int_d^\infty erf(x)\,dx = O(e^{-d^2}/d)$$

If $d = \log^{1/2} N$, then this probability is $o(1/N)$, specifically $O(1/(N \log N))$. This means that the expected work in this case is $O(\log^{-1} N)$ because $O(N)$ work is performed with very small probability.

These mean and variance equations imply

$$R_{v_1,S'} \geq k\frac{(s+1)}{(N+1)} - d\sqrt{s}$$

and

$$R_{v_2,S'} \leq k\frac{(s+1)}{(N+1)} + d\sqrt{s}$$

This gives equation (A):

$$\delta = d\sqrt{s} = \sqrt{s}\log^{1/2}N \qquad \text{(A)}$$

If we first pivot around v_2, the cost is N comparisons. If we now partition elements in S that are less than v_2 around v_1, the cost is $R_{v_2,S}$, which has expected value $k + \delta\frac{N+1}{s+1}$. Thus the total cost of partitioning is $N + k + \delta\frac{N+1}{s+1}$. The cost of the selections to find v_1 and v_2 in the sample S' is $O(s)$. Thus the total expected number of comparisons is

$$N + k + O(s) + O(N\delta/s)$$

The low order term is minimized when

$$s = N\delta/s \qquad \text{(B)}$$

Combining Equations (A) and (B), we see that

$$s^2 = N\delta = \sqrt{s}N\log^{1/2}N \qquad \text{(C)}$$

$$s^{3/2} = N\log^{1/2}N \qquad \text{(D)}$$

$$s = N^{2/3}\log^{1/3}N \qquad \text{(E)}$$

$$\delta = N^{1/3}\log^{2/3}N \qquad \text{(F)}$$

10.22 First, we calculate 12*43. In this case, $X_L = 1$, $X_R = 2$, $Y_L = 4$, $Y_R = 3$, $D_1 = -1$, $D_2 = -1$, $X_L Y_L = 4$, $X_R Y_R = 6$, $D_1 D_2 = 1$, $D_3 = 11$, and the result is 516.

Next, we calculate 34*21. In this case, $X_L = 3$, $X_R = 4$, $Y_L = 2$, $Y_R = 1$, $D_1 = -1$, $D_2 = -1$, $X_L Y_L = 6$, $X_R Y_R = 4$, $D_1 D_2 = 1$, $D_3 = 11$, and the result is 714.

Third, we calculate 22*22. Here, $X_L = 2$, $X_R = 2$, $Y_L = 2$, $Y_R = 2$, $D_1 = 0$, $D_2 = 0$, $X_L Y_L = 4$, $X_R Y_R = 4$, $D_1 D_2 = 0$, $D_3 = 8$, and the result is 484.

Finally, we calculate 1234*4321. $X_L = 12$, $X_R = 34$, $Y_L = 43$, $Y_R = 21$, $D_1 = -22$, $D_2 = -2$. By previous calculations, $X_L Y_L = 516$, $X_R Y_R = 714$, and $D_1 D_2 = 484$. Thus $D_3 = 1714$, and the result is 714 + 171400 + 5160000 = 5332114.

10.23 The multiplication evaluates to $(ac - bd) + (bc + ad)i$. Compute ac, bd, and $(a - b)(d - c) + ac + bd$.

10.24 The algebra is easy to verify. The problem with this method is that if X and Y are positive N bit numbers, their sum might be an $N+1$ bit number. This causes complications.

10.26 Matrix multiplication is not commutative, so the algorithm couldn't be used recursively on matrices if commutativity was used.

10.27 If the algorithm doesn't use commutativity (which turns out to be true), then a divide and conquer algorithm gives a running time of $O(N^{\log_{70}143640}) = O(N^{2.795})$.

10.28 1150 scalar multiplications are used if the order of evaluation is

$$((A_1A_2)(((A_3A_4)A_5)A_6))$$

10.29 (a) Let the chain be a 1x1 matrix, a 1xA matrix, and an AxB matrix. Multiplication by using the first two matrices first makes the cost of the chain $A + AB$. The alternative method gives a cost of $AB + B$, so if $A > B$, then the algorithm fails. Thus, a counterexample is multiplying a 1x1 matrix by a 1x3 matrix by a 3x2 matrix.

(b, c) A counterexample is multiplying a 1x1 matrix by a 1x2 matrix by a 2x3 matrix.

10.31 The optimal binary search tree is the same one that would be obtained by a greedy strategy: *I* is at the root and has children *and* and *it*; *a* and *or* are leaves; the total cost is 2.14.

10.33 This theorem is from F. Yao's paper, reference [58].

10.34 A recursive procedure is clearly called for; if there is an intermediate vertex, *StopOver* on the path from *s* to *t*, then we want to print out the path from *s* to *StopOver* and then *StopOver* to *t*. We don't want to print out *StopOver* twice, however, so the procedure does not print out the first or last vertex on the path and reserves that for the driver.

```
/* Print the path between S and T, except do not print */
/* the first or last vertex.  Print a trailing " to " only. */

void
PrintPath1( TwoDArray Path, int S, int T )
{
        int StopOver = Path[ S ][ T ];
        if( S != T && StopOver != 0 )
        {
                PrintPath1( Path, S, StopOver );
                printf( "%d to ", StopOver );
                PrintPath1( Path, StopOver, T );
        }
}

/* Assume the existence of a Path of length at least 1 */

void
PrintPath( TwoDArray Path, int S, int T )
{
        printf( "%d to ", S );
        PrintPath1( Path, S, T );
        printf( "%d", T ); NewLine( );
}
```

10.35 Many random number generators are poor. The default UNIX random number generator *rand* uses a modulus of the form 2^b, as does the VAX/VMS random number generator. UNIX does, however, provide better random number generators in the form of *random.* The Turbo random number generators, likewise, are also deficient. The paper by Park and Miller [44] discusses the random number generators on many machines.

10.38 If the modulus is a power of two, then the least significant bit of the "random" number oscillates. Thus *Flip* will always return heads and tails alternately, and the level chosen for a skip list insertion will always be one or two. Consequently, the performance of the skip list will be $\Theta(N)$ per operation.

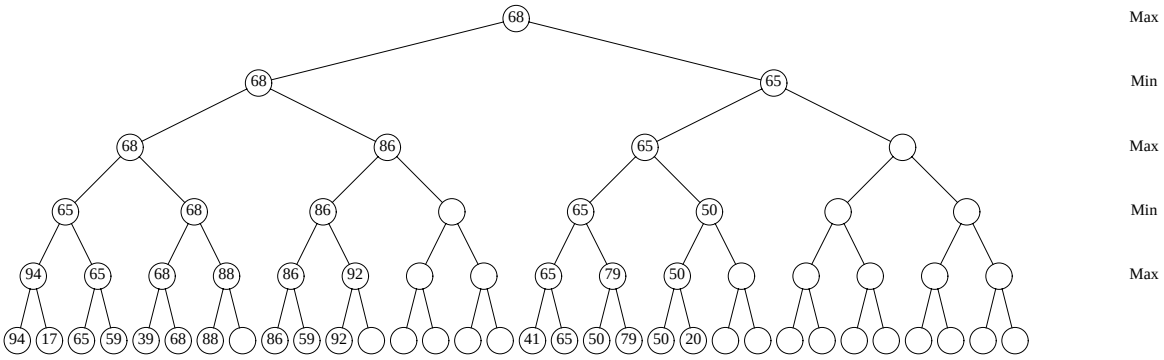
10.39 (a) $2^5 \equiv 32 \ mod \ 341$, $2^{10} \equiv 1 \ mod \ 341$. Since $32^2 \equiv 1 \ mod \ 341$, this proves that 341 is not prime. We can also immediately conclude that $2^{340} \equiv 1 \ mod \ 341$ by raising the last equation to the 34^{th} power. The exponentiation would continue as follows: $2^{20} \equiv 1 \ mod \ 341$, $2^{21} \equiv 2 \ mod \ 341$, $2^{42} \equiv 4 \ mod \ 341$, $2^{84} \equiv 16 \ mod \ 341$, $2^{85} \equiv 32 \ mod \ 341$, $2^{170} \equiv 1 \ mod \ 341$, and $2^{340} \equiv 1 \ mod \ 341$.

(b) If $A = 2$, then although $2^{560} \equiv 1 \ mod \ 561$, $2^{280} \equiv 1 \ mod \ 561$ proves that 561 is not prime. If $A = 3$, then $3^{561} \equiv 375 \ mod \ 561$, which proves that 561 is not prime. $A = 4$ obviously doesn't fool the algorithm, since $4^{140} \equiv 1 \ mod \ 561$. $A = 5$ fools the algorithm: $5^1 \equiv 5 \ mod \ 561$, $5^2 \equiv 25 \ mod \ 561$, $5^4 \equiv 64 \ mod \ 561$, $5^8 \equiv 169 \ mod \ 561$, $5^{16} \equiv 511 \ mod \ 561$, $5^{17} = 311 \ mod \ 561$, $5^{34} \equiv 229 \ mod \ 561$, $5^{35} \equiv 23 \ mod \ 561$, $5^{70} \equiv 529 \ mod \ 561$, $5^{140} \equiv 463 \ mod \ 561$, $5^{280} \equiv 67 \ mod \ 561$, $5^{560} \equiv 1 \ mod \ 561$.

10.41 The two point sets are {0, 4, 6, 9, 16, 17} and {0, 5, 7, 13, 16, 17}.

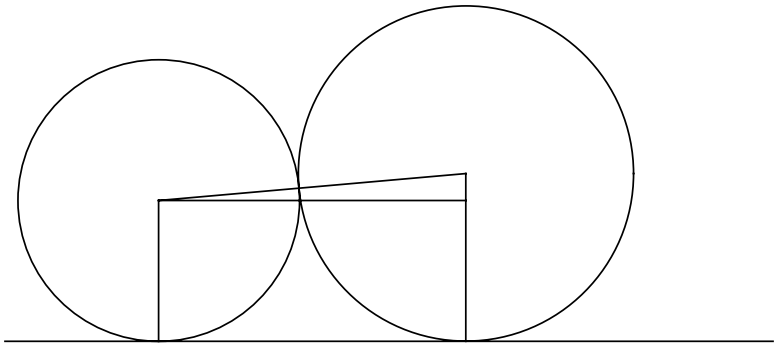
10.42 To find all point sets, we backtrack even if *Found == true,* and print out information when line 2 is executed. In the case where there are some duplicate distances, it is possible that several symmetries will be printed.

10.43



10.44 Actually, it implements both; *Alpha* is lowered for subsequent recursive calls on line 9 when *Value* is changed at line 12. *Beta* is used to terminate the *while* loop when a line of play leads to a position so good that it is obvious the human player (who has already seen a more favorable line for himself or herself before making this call) won't play into it. To implement the complementary routine, switch the roles of the human and computer. Lines that change are 3 and 4 (obvious changes), 5 (replace *Alpha* with *Beta*), 6 (replace **Value <Beta* with **Value >Alpha*), 8 (obvious), 9 (replace *Human · · · *Value ,Beta* with *Comp · · · Alpha ,*Value*), and 11 (obvious).

10.46 We place circles in order. Suppose we are trying to place circle j, of radius r_j. If some circle i of radius r_i is centered at x_i, then j is tangent to i if it is placed at $x_i + 2\sqrt{r_i r s_j}$. To see this, notice that the line connecting the centers has length $r_i + r_j$, and the difference in y-coordinates of the centers is $|r_j - r_i|$. The difference in x-coordinates follows from the Pythagorean theorem.



To place circle j, we compute where it would be placed if it were tangent to each of the first $j-1$ circles, selecting the maximum value. If this value is less than r_j, then we place circle j at x_j. The running time is $O(N^2)$.

10.47 Construct a minimum spanning tree T of G, pick any vertex in the graph, and then find a path in T that goes through every edge exactly once in each direction. (This is done by a depth-first search; see Exercise 9.31.) This path has twice the cost of the minimum spanning tree, but it is not a simple cycle.

Make it a simple cycle, without increasing the total cost, by bypassing a vertex when it is seen a second time (except that if the start vertex is seen, close the cycle) and going to the next unseen vertex on the path, possibly bypassing several vertices in the process. The cost of this direct route cannot be larger than original because of the triangle inequality.

If there were a tour of cost K, then by removing one edge on the tour, we would have a minimum spanning tree of cost less than K (assuming that edge weights are positive). Thus the minimum spanning tree is a lower bound on the optimal traveling salesman tour. This implies that the algorithm is within a factor of 2 of optimal.

10.48 If there are two players, then the problem is easy, so assume $k > 1$. If the players are numbered 1 through N, then divide them into two groups: 1 through $N/2$ and $N/2+1$ though N. On the i^{th} day, for $1 \le i \le N/2$, player p in the second group plays players $((p + i) \bmod N/2) + 1$ in the first group. Thus after $N/2$ days, everyone in group 1 has played everyone in group 2. In the last $N/2-1$ days, recursively conduct round-robin tournaments for the two groups of players.

10.49 Divide the players into two groups of size $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$, respectively, and recursively arrange the players in any order. Then merge the two lists (declare that $p_x > p_y$ if x has defeated y, and $p_y > p_x$ if y has defeated x – exactly one is possible) in linear time as is done in mergesort.

10.50

10.51 Divide and conquer algorithms (among others) can be used for both problems, but neither is trivial to implement. See the computational geometry references for more information.

10.52 (a) Use dynamic programming. Let S_k = the best setting of words w_k, w_{k+1}, ... w_N, U_k = the ugliness of this setting, and l_k = for this setting, (a pointer to) the word that starts the second line.

To compute S_{k-1}, try putting w_{k-1}, w_k, ..., w_M all on the first line for $k < M$ and $\sum_{i=k-1}^{M} w_i < L$. Compute the ugliness of each of these possibilities by, for each M, computing the ugliness of setting the first line and adding U_{m+1}. Let M' be the value of M that yields the minimum ugliness. Then U_{k-1} = this value, and $l_{k-1} = M' + 1$. Compute values of U and l starting with the last word and working back to the first. The minimum ugliness of the paragraph is U_1; the actual setting can be found by starting at l_1 and following the pointers in l since this will yield the first word on each line.

(b) The running time is quadratic in the case where the number of words that can fit on a line is consistently $\Theta(N)$. The space is linear to keep the arrays U and l. If the line length is restricted to some constant, then the running time is linear because only $O(1)$ words can go on a line.

(c) Put as many words on a line as can fit. This clearly minimizes the number of lines, and hence the ugliness, as can be shown by a simple calculation.

10.53 An obvious $O(N^2)$ solution to construct a graph with vertices 1, 2, ..., N and place an edge (v, w) in G iff $a_v < a_w$. This graph must be acyclic, thus its longest path can be found in time linear in the number of edges; the whole computation thus takes $O(N^2)$ time.

Let $BEST(k)$ be the increasing subsequence of exactly k elements that has the minimum last element. Let t be the length of the maximum increasing subsequence. We show how to update $BEST(k)$ as we scan the input array. Let $LAST(k)$ be the last element in $BEST(k)$. It is easy to show that if $i < j$, $LAST(i) < LAST(j)$. When scanning a_M, find the largest k such that $LAST(k) < a_M$. This scan takes $O(\log t)$ time because it can be performed by a binary search. If $k = t$, then x_M extends the longest subsequence, so increase t, and set $BEST(t)$ and $LAST(t)$ (the new value of t is the argument). If k is 0 (that is, there is no k), then x_M is the smallest element in the list so far, so set $BEST(1)$ and $LAST(1)$, appropriately. Otherwise, we extend $BEST(k)$ with x_M, forming a new and improved sequence of length $k+1$. Thus we adjust $BEST(k+1)$ and $LAST(k+1)$.

Processing each element takes logarithmic time, so the total is $O(N \log N)$.

10.54 Let $LCS(A, M, B, N)$ be the longest common subsequence of A_1, A_2, ..., A_M and B_1, B_2, ..., B_N. If either M or N is zero, then the longest common subsequence is the empty string. If $x_M = y_N$, then $LCS(A, M, B, N) = LCS(A, M-1, B, N-1), A_M$. Otherwise, $LCS(A, M, B, N)$ is either $LCS(A, M, B, N-1)$ or $LCS(A, M-1, B, N)$, whichever is longer. This yields a standard dynamic programming solution.

10.56 (a) A dynamic programming solution resolves part (a). Let $FITS(i, s)$ be 1 if a subset of the first i items sums to exactly s; $FITS(i, 0)$ is always 1. Then $FITS(x, t)$ is 1 if either $FITS(x-1, t-a_x)$ or $FITS(x-1, t)$ is 1, and 0 otherwise.

(b) This doesn't show that $P = NP$ because the *size* of the problem is a function of N and $\log K$. Only $\log K$ bits are needed to represent K; thus an $O(NK)$ solution is exponential in the input size.

10.57 (a) Let the minimum number of coins required to give x cents in change be $COIN(x)$; $COIN(0) = 0$. Then $COIN(x)$ is one more than the minimum value of $COIN(x - c_i)$, giving a dynamic programming solution.

(b) Let $WAYS(x, i)$ be the number of ways to make x cents in change without using the first i coin types. If there are N types of coins, then $WAYS(x, N) = 0$ if $x \neq 0$, and $WAYS(0, i) = 1$. Then $WAYS(x, i - 1)$ is equal to the sum of $WAYS(x - pc_i, i)$, for integer values of p no larger than x/c_i (but including 0).

10.58 (a) Place eight queens randomly on the board, making sure that no two are on the same row or column. This is done by generating a random permutation of 1..8. There are only 5040 such permutations, and 92 of these give a solution.

10.59 (a) Since the knight leaves every square once, it makes B^2 moves. If the squares are alternately colored black and white like a checkerboard, then a knight always moves to a different colored square. If B is odd, then so is B^2, which means that at the end of the tour the knight will be on a different colored square than at the start of the tour. Thus the knight cannot be at the original square.

10.60 (a) If the graph has a cycle, then the recursion does not always make progress toward a base case, and thus an infinite loop will result.

(b) If the graph is acyclic, the recursive call makes progress, and the algorithm terminates. This could be proved formally by induction.

(c) This algorithm is exponential.