1. (a) The algorithm written in C:

```
int T(int n) {
int sum = 0;
if (n==0 || n==1) return 2;
for(int j=1;j<=n-1;j++)
sum+=T(j)*T(j-1);
return sum;
}
```

Let $N(n)$ denote the number of operations required to calculate $T(n)$, from the algorithm we have:

$$N(n) = \sum_{i=1}^{n-1} 2 * N(i) - N(n-1) - N(0) + 2 * (n-1)$$

$$\geq N(n-1) + N(n-2) \geq 2 * N(n-2)$$
$$= 2 * 2 * N(n-4) = \cdots = 2^{\frac{n}{2}} * N(0) = 2^{\frac{n}{2}}$$

Hence it's exponential.

(b) (b) An algorithm in C:

```
T[0]=T[1]=2;
for(j=2;j<=n;j++)
{ T[j] = 0;
   for(k=1;k<=j;k++)
T[j]+=T[k]*T[k-1];
}
return T[n];
```

(c) An algorithm in C:

```
T[0]=T[1]=2;
for(j=2;j<=n;j++)
T[j]=T[j-1] + T[j-1]*T[j-2];
return T[n];
```

2. The solution to the problem of the shortest common subsequence for three strings is essentially identical to the solution with 2 strings. By treating the $T(i, j, k)$'s as array entries and updating the table in the appropriate way we can get the following $O(n^3)$ time algorithm.

```
For i=0 to m do T(i,0,0)=0

For j=0 to n do T(0,j,0)=0

For k=0 to o do T(0,0,k)=0

For i=1 to m do
    For j= 1 to n do
        For k= 1 to o do
            if a(i) = b(j) = c(k) then T(i,j,k)=T(i-1,j-1,k-1) + 1
            else T(i,j,k)= MAX( T(i, j-1, k), T(i-1, j, k), T(I, j, k-1))
```

3. We present an algorithm to compute the shortest common super-sequence of two strings: Let $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$. Let $M[i, j]$ denote the length of the shortest common super-sequence of string $a_1 \dots a_i$, $b_1 \dots b_j$, for $1 \le i \le m$ and $1 \le j \le n$. Note that the last letter in the shortest super-sequence of $a_1, \dots, a_i$ and $b_1, \dots, b_j$ is either a_i or b_j. If the last letter is a_i then the previous letters are the solution to the subproblem for $a_1, \dots, a_{i-1}$ and $b_1, \dots, b_j$. If the last letter is b_j then the previous letters are the solution to the subproblem for $a_1, \dots, a_i$ and $b_1, \dots, b_{j-1}$. Hence, $M(i, j) =$

$$\min(M(i-1, j-1) + 1 \text{ if} a_i = b_j, M(i-1, j) + 1, M(i, j-1) + 1)$$

. This table update can be embedded in two nested loops, the first that goes from $i = 1$ to m and the second with goes from $j = 1$ to n. This gives an $\Theta(n^2)$ time algorithm. Note that the actual sequence can be computed by following the updates back from $M(m, n)$. Code follows:

```
for $i = 1$ to $m$ do
  for $j = 1$ to $n$ do
    if $a_i = b_j$ then
      $M[i, j] = M[i-1, j-1] + 1$
    else $M[i, j] = \min(M[i-1, j], M[i, j-1]) + 1$
```

4. To appear

5. We present an algorithm to compute the minimum edit distance of two strings. Note that:

   (a) If it were possible to convert $a_1, \dots, a_{m-1}$ into $b_1, \dots, b_n$, one could complete the transformation of A into B by deleting a_m.

(b) If it were possible to convert $a_1, \ldots, a_m$ into $b_1, \ldots, b_{n-1}$, one could complete the transformation by adding b_n to A.

(c) If it were possible to convert $a_1, \ldots, a_{m-1}$ into $b_1, \ldots, b_{n-1}$, one could complete the transformation by replacing a_m with b_n.

Now let $A[i, j]$ be the minimum cost of transforming $a_1, \ldots, a_i$ into $b_1, \ldots, b_j$. The algorithm is:

```
MinumumEditDistance($A, B$)
**for** $i = 1$ to $m$
  **for** $j = 1$ to $n$
    **if** $a_i = b_j$ **then**
      $A[i, j] = A[i-1, j-1]$
    **else** $A[i, j] = \min(A[i-1, j] + 3, A[i, j-1] + 4, A[i-1, j-1] + 5)$
```

Starting from $A[m, n]$, we can trace backwards through the table to determine which operations were performed at each step.

6. To appear.

7. We give a dynamic programming algorithm for the problem of finding the cheapest path on a checkerboard. Number the columns 1 to n from left to right, and number the rows 1 to n from the bottom to the top. Define $Q(a, b)$ to the most profit you can get from moving from square $(1, 1)$ to square (a, b). Then the obvious recursive algorithm is:

   - If $a = 1$ and $b = 1$ then $Q(a, b) = 0$,
   - else if $a = 1$ and $b \neq 1$ then $Q(a, b) = Q(a, b-1) + p((a, b-1), (a, b))$,
   - else if $a \neq 1$ and $b = 1$ then $Q(a, b) = Q(a-1, b) + p((a-1, b), (a, b))$,
   - else if $a \neq 1$ and $b \neq 1$ then $Q(a, b) = \max(Q(a, b-1) + p((a, b-1), (a, b)), Q(a-1, b) + p((a-1, b), (a, b)), Q(a-1, b-1) + p((a-1, b-1), (a, b)))$

   Turning this into an iterative bottom-up array based algorithm we get:

```
For $a = 1$ to $n$ do
    For $a = 1$ to $n$ do
```

   - If $a = 1$ and $b = 1$ then $Q(a, b) = 0$,
   - else if $a = 1$ and $b \neq 1$ then $Q(a, b) = Q(a, b-1) + p((a, b-1), (a, b))$,
   - else if $a \neq 1$ and $b = 1$ then $Q(a, b) = Q(a-1, b) + p((a-1, b), (a, b))$,
   - else if $a \neq 1$ and $b \neq 1$ then $Q(a, b) = \max(Q(a, b-1) + p((a, b-1), (a, b)), Q(a-1, b) + p((a-1, b), (a, b)), Q(a-1, b-1) + p((a-1, b-1), (a, b)))$

Where now the references to Q are array look ups.

8. The array developed is as follows:

	1	2	3	4	5
1	0	200	1200	320	1320
2		0	400	240	640
3			0	200	700
4				0	2000
5					0

In each step, the k is:

M[1][5]	k=4
M[1][4]	k=1
M[2][4]	k=2

Hence the optimal order is $(A_1(A_2(A_3(A_4)))A_5$.

9. We use dynamic programming to formulate an algorithm to compute the minimum polygon triangulation. Note that after the first "cut" is made, the original polygon P will be divided into two new polygons P' and P''. The triangulations P' and P'' are completely independent of each other. The cheapest way to triangulate P is the minimum of:

   (a) For $3 \leq k \leq n-2$, the length of (v_1, v_k) plus the cheapest way to triangulate $P' = \{v_1, \ldots, v_k\}$ plus the cheapest way to triangulate $P'' = \{v_k, \ldots v_n\}$.

   (b) The length of (v_2, v_n) plus the cheapest way to triangulate $P' = \{v_2, \ldots, v_n\}$.

   (c) The length of (v_1, v_{n-1}) plus the cheapest way to triangulate $P' = \{v_1, \ldots, v_{n-1}\}$.

   Create a two-dimensional array A. $A[i,j]$ = minimum weight of triangulation of the polygon defined by $v_i, \ldots, v_j$.

```
**for** $i = 1$ **to** $n-4$ **do**
  **for** $j = i+3$ **downto** $1$ **do**
```

$$\begin{aligned} T[i,j] &= \min\Big(\min_{i+2 \leq k \leq j-2}(d(i,k) + d(k,j) + A[i,k] + A[k,j]), \\ &\quad d(j,i+1) + A[i+1,j], d(i,j-1) + A[i,j-1]\Big) \end{aligned}$$

10. We strengthen the inductive hypothesis to compute not only the balance factor but the height for each node. Begin at the root r.

    Compute-Balance(r)
    {

```
If RChild(n) And LChild(n) == NULL,
    Height(r) = 0
Else
   Height(r) = max[Compute-Balance(LChild(r)),Compute-Balance(RChild(r))]+1

Balance(r) = $|Height(LChild(r)) - Height(RCHild(r))|$
Return Height (r)
}
```

Obviously this algorithm runs in linear time.

11. We present an algorithm to compute the maximum consecutive sum of n integers $x_1, \ldots, x_n$. We define two functions:

    - MCS(i) = Maximum Consecutive Sum of the first i integers.
    - MSS(i) = Maximum Consecutive Sum of the first i integers that uses x_i.

    MSS(i) is computed by:
    MSS(i) = MAX(0, MSS(i-1)+x_i)
    And MCS(n) is computed by:
    MCS(i) = MAX(MCS(i-1), MSS(i-1)+x_i)

    By placing the two assignment statements inside a loop where i runs from 1 to n, we get a linear time algorithm.

12. The input to this problem is a tree T with weights on the edges. The goal is to find the path in T with minimum aggregate weight. An path is a collection of adjacent vertices, where no vertex is used more than once.

    Root the tree at an arbitrary node r, and process the tree in postorder. We generalize the induction hypothesis to compute not only the shortest path in each subtree, but also the shortest path with one endpoint being the root of the subtree. Consider an arbitrary node v with branches to k descendants $w_1, w_2, \ldots, w_k$. For each such node v the algorithm computes the following information:

    - best(v) = the minimum weight of a path for the subtree rooted at v.
    - root-best(v) = the minimum weight of a path ending at v in the subtree rooted v.

    At node v, the algorithm first recursively computes best(w_i) and root-best(w_i) for each descendant subtree . It then computes best(v) and root-best(v) using the following recurrence relations that correspond to the two cases identified above:

$$\text{rootbest}(v) = \min(0, \min_{i=1}^{k}(d(v, w_i) + \text{rootbest}(w_i)))$$

$$\text{best}(v) = \min(\text{rootbest}(v), \min_i(\text{best}(w_i)), \min_{i,j}(\text{rootbest}(w_i)+\text{rootbest}(w_j)+d(v, w_i)+d(v, w_j)))$$

Its not too hard (but not completely trivial) to see that this can be implemented in linear time.

13. We present an algorithm for the clockwise-Towers-of-Hanoi problem: procedure Hanoi(n,A,B,C)

```
    Moves n discs from A to B moving only clockwise.
    Uses C as an "intermediate" peg.
  begin
    if $n = 0$ then
      do nothing
    else if $B$ is one peg clockwise from $A$ then
      Hanoi($n - 1$,$A$,$C$,$B$)
      Move disc $n$ from $A$ to $B$
      Hanoi($n - 1$,$C$,$B$,$A$)
    else
      Hanoi($n - 1$,$A$,$B$,$C$)
      Move disc $n$ from $A$ to $C$
      Hanoi($n - 1$,$B$,$A$,$C$)
      Move disc $n$ from $C$ to $B$
      Hanoi($n - 1$,$A$,$B$,$C$)
  end
```

14. We present the following dynamic programming algorithm to compute the AVL tree with minimum expected access time. The main idea is to strengthen the inductive hypothesis to compute the AVL tree of each height with the minimum expected access time. Define A[i,j,h] as the minimized expected depth for a tree with height h on the keys $K_i, K_{i+1}, \cdots, K_j$. We then get the following recurrence:

$$A[i, j, h] = \min_{i \le r \le j}($$

$$A[i, r-1, h-1] + A[r+1, j, h-2] + \sum_{a=i}^{j} p_a,$$

$$A[i, r-1, h-1] + A[r+1, j, h-1] + \sum_{a=i}^{j} p_a,$$

$$A[i, r-1, h-2] + A[r+1, j, h-1] + \sum_{a=i}^{j} p_a)$$

Here r is the root of the new tree, and p_a is the probability of accessing key K_a. Note that if the tree is going to be of height h then one of its two subtrees must be of height $h-1$ and the other can be of height at most $h-1$. Since the tree is an AVL tree the heights of the two subtrees can differ by at most 1. One can then get code by wrapping this assignment statements in a loop for i from n to 1, a loop for j from i to n, and a loop for h from 0 to n (actually you can replace n here by something like $\sqrt{2}\log n$ if you know that AVL trees are balanced). The final minimum expected depth can be found by taking the minimum over all h of $A[1, n, h]$, and the tree can be recovered the same way that it was for the problem on computing the binary search tree with minimum expected access time.

15. We give dynamic programming algorithm to compute the non-overlapping collection of intervals $I_1, \ldots, I_n$ with maximum measure. We consider the intervals by increasing order of their left endpoints. The main idea is to strengthen the inductive hypothesis to compute the maximum measure non-overlapping collection of intervals with a particular rightmost interval, for all possible choices of rightmost interval. Define $S[i, j]$ as the total length of the longest non-overlapping interval set among the first i intervals such that the right-most interval in this set is the jth one. Then $S[i, j]$ is the maximum over all intervals I_k, such that the right endpoint of I_k is to the left of the left endpoint of I_k, of $S[k, k]$ plus the length of I_j. By wrapping this in a loop for i from 1 to n, and for j from 1 to i, you get the algorithm. The maximum measure of a non-overlapping collection of intervals can then be found by taking the maximum $S[n, j]$ for all j.

16. No solution given.

17. No solution given.

18. We present a solution for the problem of determining if the values of n items can be added and subtract ed in such a way that $\sum_{k=1}^{n}(-1)^{x_k} v_k = L$. We use dynamic programming. As pruning rule we remark that if we have two solutions with the same value we only need to keep one. Here we cannot discard solutions with negative values or with values larger than L since subtraction is allowed. However, if we define $S = \sum_{i=1}^{n} v_i$, we know that any solution will have its value in $\{-S \ldots S\}$, and thus we will have at most $2S$ solutions to keep.

    For $i = 1 \ldots n$ and $j = -S \ldots S$ let $A(i, j)$ be a boolean variable that indicate whether or not there is a solution with the first i objects that has a value equal to j. That is, $A(i,j)$ is true if there is a solution to

$$A(i,j) = \sum_{k=1}^{i} (-1)^{x_k} v_k = j$$

where each x_k is either 1 or 0.

With the initialization $A(0,0) = T$ and $A(0,j) = F$ for $j \neq 0$, we can fill the table row by row using the following rule:

$$\text{if } A(i,j) = \text{ T then} A(i+1, j+v_{i+1}) \text{ and } A(i+1, j-v_{i+1}).$$

That is, if there is a solution for j at level i then there is a solution for $j + v_{i+1}$ and $j - v_{i+1}$ at level $i+1$.

At the end, the entry $A(n, L)$ will indicate whether there is a solution for the problem or not. If you want you can then construct a solution by tracing a "path" back in the table starting at $A(n, L)$.

The size of the table is nS. The time to fill an entry is constant. Therefore the total time is $O(nS)$ which is polynomial in $n + L$ if we assume that S is polynomial in L.

19. We present an algorithm for determining if there is a subset of n items whose total value is $L \bmod n$. We use dynamic programming. As pruning rule we remark that if we have two solutions with the same value modulo n we only need to keep one. Therefore there are at most n solutions to keep at each level.

    For $i = 1 \ldots n$ and $j = 0 \ldots n-1$ let $A(i,j)$ be a boolean variable that indicate whether or not there is a solution with the first i objects that has a value equal to $j \bmod n$. That is, $A(i,j)$ is true if there is a solution to

$$A(i,j) = \left(\sum_{k=1}^{i} x_k v_k \right) \bmod n = j$$

    where each x_k is either 1 or 0.

    With the initialization $A(0,0) = T$ and $A(0,j) = F$ for $j = 1 \ldots n-1$, we can fill the table row by row using the following recurrence relation:

$$A(i,j) = A(i-1, j) \text{ or } A(i-1, (j - v_i) \bmod n).$$

    That is, there is a solution modulo j at level i if there is a solution for j at level $i-1$ or if you can add v_i to a solution at level $i-1$ and get j modulo n.

    At the end, the entry $A(n, L \bmod n)$ will indicate whether there is a solution for the problem or not. If you want you can then construct a solution by tracing a "path" back in the table starting at $A(n, L \bmod n)$.

The size of the table is n^2. The time to fill an entry is constant. Therefore the total time is $O(n^2)$ which is polynomial in $n + L$.

20. We use dynamic programming for the problem of obtaining the maximum value from a subcollection (allowing repetition) of n items, subject to the restriction that the total weight of the set cannot exceed W. As pruning rule we remark that if we have two solutions with the same weight we only need to keep the one with the highest value and that we only need to keep solutions with weight no greater than W.

    For $i = 1 \dots n$ and $j = 0 \dots W$ we compute $A(i, j) =$ the maximum value of an assignment of the i first objects with weight bounded by j. That is,

    $$A(i,j) = \max \sum_{k=1}^{i} x_k v_k \text{ s.t. } \sum_{k=1}^{i} x_k w_k \leq j$$

    where each x_k is a non-negative integer.

    Assuming $A(0, j) = 0$ for $j = 0 \dots W$, we can fill the table row by row using the following recurrence relation:

    $$A(i,j) = \max_{x_i = 0 \dots j/w_i} x_i v_i + A(i-1, j - x_i w_i)$$

    We only need to consider values of x_i in the range $0 \dots j/w_i$ since any higher value would make $j - x_i w_i$ negative. The final solution will be in $A(n, W)$.

    The size of the table is $n \times W$. The time to fill an entry is $O(W)$. Therefore the total time is $O(n \times W^2)$ which is polynomial in $n + W$.

21. I did this one in class

22. We present a dynamic programming algorithm for the problem of determining if there are two subsets of n rubies and emeralds that contain the same number of rubies, the same number of emeralds, and the same total value. Let n_e and n_r be the number of emeralds and rubies among the $n = n_e + n_r$ gems. If any of n_e, n_r or L is odd then clearly the problem has no solution. Otherwise we can determine whether there is a solution using dynamic programming.

    For $i = 0 \dots n$, $j = 0 \dots n_e/2$, $k = 0 \dots n_r/2$, and $\ell = 0 \dots L$ let $A(i, j, k, \ell)$ be a boolean variable that indicate whether or not there is a subset of the first i gems that contains exactly j emeralds and k rubies and has value ℓ. The table is initialized to all `False`, except for $A(0, 0, 0, 0)$ which is initialized to `True`. We can fill the table row by row using the following code:

```
for $i = 1 \dots n$ do
  for $j = 1 \dots n_e/2$ do
for $k = 1 \dots n_r/2$ do
  for $\ell = 0 \dots L/2$ do
    if $A(i-1, j, k, \ell)$ then
      $A(i, j, k, \ell) = \text{True}$
      if $\text{gem}_i = \text{emerald}$ then
        $A(i, j+1, k, \ell + v_i) \leftarrow \text{True}$
      else
        $A(i, j, k+1, \ell + v_i) \leftarrow \text{True}$
```

There is a solution to the problem if at the end $A(n, n_e/2, n_r/2, L/2)$ is `True`. The time taken by the algorithm is $O(n^3 \times L)$ which is polynomial in $n + L$.

23. Solutions for the word-layout problems:

    (a) Each word can either be placed at the end of the current last line or at the beginning of a new line. Therefore, given a layout of the first i words, there are 2 places to put the $i + 1$th word. However, note that the first word can only go at the beginning of the first line. Since there are exactly two options for the placement of each word after the first, there are 2^{n-1} total solutions.

    (b) The ith level of the tree contains all possible layouts of the first i words. The left child of a node v corresponds to the layout achieved by appending the $i+1$th word to the last line of the layout in v. The right child of v corresponds to the layout achieved by beginning a new line with the $i + 1$th word.

    (c) Obviously, we can eliminate all layouts that would place more than L characters on a particular line. More helpfully, if there are two solutions S_1 and S_2 of the first i words such that both S_1 and S_2 have k characters on the last line, we only need to keep the one with the smallest total penalty.

    (d) As pruning rule we remark that if we have two layouts of the first i words that have the same number of characters on the last line we only need to keep the layout with the smaller maximum line penalty. For $i = 0 \dots n$ and $j = 1 \dots L$, let $A(i, j)$ be the smallest maximum line penalty (not counting the last line) of any layout of the first i words that has j characters on the last line. Let w_i be the length of the ith word. Given $A(i-1, j)$ for $j = 1 \dots L$ we can compute $A(i, j)$ by distinguishing three cases:

        i. if $j < w_i$ then $A(i, j) = \infty$ since no layout that ends with the ith word can have less than w_i characters on the last line.

        ii. if $j > w_i$ then $A(i, j) = A(i-1, j-w_i)$ since in this case the ith word is added on the last line, without starting a new one.

   iii. if $j = w_i$ it means that the ith word starts a new line. The penalty depends on the number of characters on the (previous) last line. You want to make the choice that yields the smallest maximum line penalty:

$$A(i, w_i) = \min_{k=1\ldots L} \{\max(A(i-1,k), L-k)\}$$

Using the above relations it is easy to fill the table. At the end the optimal maximum line penalty is given by the minimum of $A(n, i)$, $i = 1 \ldots L$. As usual the actual layout can be reconstructed by tracing the "path" back in the table starting from this final cell.

24. We use dynamic programming to find a subsequence of maximum aggregate cost (note that there is more than one reasonable way to construct a dynamic programming algorithm here). We use the pruning method. Let level l of the tree consist of all subsequences of T that match the first l letters in the pattern P. The pruning rule is that if you have two solutions on the same level are of the same length and end at the same letter in T, then you can prune the one of lesser cost. So let $A[l, s]$ be the maximum cost of a subsequence of T equal to $p_1, \ldots, p_l$, where the last letter of this subsequence is t_s. We then get the following code

```
for $l = 1 \ldots n$ do
    for $s = 1 \ldots n$ do
      if $A(l, s)$ is defined then
        for $r = s + 1 \ldots n$ do
          if $t_r = p_{l+1}$ then
            if $A[l+1, r] = \min(A[l+1, r], A[l, s] + c_r)$
```

25. We use dynamic programming to compute the optimal solution to the two taxi cab problem. We first note that when p_i is serviced one of the taxi is in p_i while the other taxi is in one of the location $p_0 \ldots p_{i-1}$, where p_0 denotes the origin. Therefore at stage i we only need to keep one solution (the best one) for each possible location $p_0 \ldots p_{i-1}$ of the second taxi.

    For $i = 1 \ldots n$ and $j = 0 \ldots i-1$ let $A(i, j)$ be the minimum cost of a routing that serves the points $p_1 \ldots p_i$ and leaves one taxi in p_i and the other in p_j. Given the best solutions for stage i we can compute the solutions for stage $i+1$ by considering the following two possibilities:

    (a) the taxi that was in p_i is used to serves p_{i+1} and the other taxi remains where it was. That is, for $j = 0 \ldots i-1$,

$$A(i+1, j) = |p_i p_{i+1}| + A(i, j).$$

    (b) the taxi in p_i remains at p_i, and the other taxi serves p_{i+1}. Only the solution with minimum cost is kept.

$$A(i+1, i) = \min_{k=0\ldots i-1} |p_k p_{i+1}| + A(i, k),$$

Using the above relations it is easy to fill the table. At the end the minimum routing cost is given by the minimum of $A(n, i)$, $i = 0 \ldots n-1$. The actual routing can be reconstructed by tracing the “path” back in the table starting from the final cell.

Since there are n stages and each stage takes $O(n)$ time, the total time is $O(n^2)$ which is polynomial in n.

26. I did this one in class.