Problem 1
Consider the following function computing the Fibonacci number

Algorithm 1 FIBONACCI

```
1: function FIB(n)
2:     if i == 0 or i == 1 then
3:         return i
4:     else
5:         return Fib(i − 1) + Fib(i − 2);
6:     end if
7: end function
```

Draw the tree illustrating the recursive calls for FIBONACCI(5). It turns out that the number of leafs in a call tree for FIBONACCI is equal to the Fibonacci number itself, which is roughly 1.618^n. Write an iterative implementation of FIBONACCI(n) that only uses $O(n)$ time and space.

Solution:

Algorithm 2 FIBONACCI

```
1: function FIB(n)
2:     initialise an array M[0..n]
3:     M[0] ← 0
4:     M[1] ← 1
5:     for i = 2 to n do
6:         M[i] ← M[i − 1] + M[i − 2]
7:     end for
8:     return M[n]
9: end function
```

Problem 2
Show all intermediate steps of the dynamic programming algorithm for the weighted interval scheduling problem, for the following input.

item	1	2	3	4	5	6	7	8	9	10
start	0	1	0	3	2	4	6	2	7	6
finish	2	3	4	5	6	7	8	9	10	11
weight	2	9	6	5	7	11	8	10	4	6

Solution:
The DP recurrence for the weighted interval scheduling problem is:

$$M[j] = \max\left(w_j + M[p(j)], M[j-1]\right),$$

with $M[0] = 0$ as the base case. Recall that $M[j]$ is the optimum value of the subinstance restricted to intervals $I_1, \ldots, I_j$. Let $\mathcal{O}_j$ be the optimum set of items that gives the value $M[j]$.

j	1	2	3	4	5	6	7	8	9	10
s_j	0	1	0	3	2	4	6	2	7	6
f_j	2	3	4	5	6	7	8	9	10	11
w_j	2	9	6	5	7	11	8	10	4	6
$p(j)$	0	0	0	2	1	3	5	1	6	5
$M[j]$	2	9	9	14	14	20	22	22	24	24
$\mathcal{O}_j$	$\{1\}$	$\{2\}$	$\{2\}$	$\{2,4\}$	$\{2,4\}$	$\{2,6\}$	$\{2,4,7\}$	$\{2,4,7\}$	$\{2,6,9\}$	$\{2,6,9\}$

The solution is at $M[10] = 24$ and corresponds to $\mathcal{O}_j = \{2, 6, 9\}$.

Problem 3
Consider the following variant of the Interval Scheduling problem we saw in class. The input is defined by a set of intervals $I_1, \ldots, I_n$. We say that interval $I_i = (s_i, f_i)$ has length $f_i - s_i$. We would like to pick a set of non-intersecting intervals that use as much as possible of the common resource; that is, we want to maximize the sum of the lengths of the scheduled intervals. Design an efficient algorithm for this problem using dynamic programming.

Solution: There is a simple solution to this problem: We define the weight of interval i to be $f_i - s_i$. Now the problem reduces to finding a maximum weight set of non-intersecting intervals, which is a Weighted Interval Scheduling problem that can be solved in $O(n)$ time using the dynamic programming algorithm that we went through in class.

Problem 4
Solve the following instance of the $\{0, 1\}$ Knapsack Problem with four items where the maximum allowed weight is $W_{\max} = 10$.

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

Solution: We define $B[k, w]$ to be the optimal solution that can be obtained using only the first k items, with maximum allowed total weight of w. Here k ranges from 1 to 4, while w ranges from 0 to 10. We also define $B[0, w] = 0$ for all w. Then we have to fill in the table, one row at a time, to find the optimal solution, using the recurrence type of equation that can be found in the class notes. This table will end up having these values:

	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	25	25	25	25
2	0	0	15	15	15	15	15	25	25	40	40
3	0	0	15	20	20	35	35	35	35	40	45
4	0	0	15	20	20	35	36	36	51	56	56

So the maximum benefit obtainable is 56. Since this example is small, we can easily figure out that we want to take items 3 and 4 to obtain this benefit. Otherwise, as also mentioned in class, we can trace back through the table to find this out.

Problem 5

Suppose we are going on a hiking trip along the Great North Walk from Sydney to Newcastle. We have a list of all possible campsites along the way, say there are n possible places where we could camp. (Assume campsite are right off the path.) We want to do this trip in exactly k days, stopping in $k-1$ campsites to spend the night. Our goal is to plan this trip so that we minimize the maximum amount of walking done on any one day. In other words, if our trip involves 3 days of walking, and we walk 11, 14, and 12 kilometers on each day respectively, the cost of this trip is 14. Another schedule that involves walking 11, 13, and 13 kilometers on each day has cost 13, and is thus preferable. The locations of the campsites are specified in advance, and we can only camp at a campsite.

Using dynamic programming, design an efficient algorithm for solving this problem. Your algorithm should run in $O(n^2k)$ time.

Solution: Number all the locations along the trail. We start at location 0 and want to go to location $n+1$; there are n campsites along the way. Let $D(0,i)$ be the distance from 0 to location i; to compute the distance between two locations $j < i$ we use $D(j,i) = D(0,i) - D(0,j)$. The journey takes k days, and we stop at $k-1$ places along the way.
Let $M(i,\ell)$ to be cost of the optimal solution to reach campsite i in ℓ days. To derive a recurrence for this quantity we need to consider which campsite was used before the ℓth, suppose it was j then $M(i,\ell) = \max\{M(j,\ell-1), D(j,i)\}$. For the base case, if we want to reach location i in one day we must walk from 0 to i, thus

$$\begin{cases} M(i,\ell) = \min\limits_{j<i} \Big\{ \max\{M(j,\ell-1), D(j,i)\} \Big\}, \\ M(i,1) = D(0,i). \end{cases} \tag{1}$$

The answer will be in $M(n+1,k)$. The running time is $O(n^2k)$ since the table has size $O(nk)$ and computing each entry takes $O(n)$ time.

Problem 6

Consider a post office that sells stamps in three different denominations, 1c, 7c, and 10c. Design a dynamic programming algorithm that will find the minimum number of stamps necessary for a postage value of n cents. (Note that a greedy type of algorithm wont necessarily give you the correct answer, and you should be able to find an example to show that such a greedy algorithm doesnt work.) What is the running time of your algorithm?

Solution: Let $S[n]$ denote the minimum number of stamps needed to make postage for n cents. We clearly then have

$$S[0] = 0$$
$$S[1] = 1$$
$$S[2] = 2$$
$$S[3] = 3$$
$$S[4] = 4$$
$$S[5] = 5$$
$$S[6] = 6$$
$$S[7] = 1$$
$$S[8] = 2$$
$$S[9] = 3$$
$$S[10] = 1.$$

These cases above should be considered our base cases and we then work from these to compute values of $S[n]$ for higher values of n. The main idea, then is to consider what happens if we use a stamp of a particular value. For example, if we want postage of n cent, then we can clearly get it by taking one stamp of 1c, and $S[n-1]$ stamps (of appropriate values) to make up the remaining postage of $(n-1)c$. Or if we take a 7c stamp, then we need $S[n-7]$ stamps (of the right values) to make up the rest, and similarly if we take a 10c stamp. This idea is essentially the heart of the dynamic programming algorithm that we use.
Set up the array S starting as above (with the values $S[i]$ for $i = 0, \ldots, 10$).
If $n > 10$ then $S[j] = 1 + \min\{S[j-1], S[j-7], S[j-10]\}$.
Thus, finding $S[n]$ takes $O(n)$. Now we can find $S[n]$ easily as above, but knowing this does not tell us the exact nature of the stamps we need. It would be nice to know that we can make 45c postage with six stamps, and that we need one 1c, two 7c, and three 10c stamps to do so. (For some values of n there could certainly be more than one way to do this. For example, we can get 63c with nine 7c stamps, or with six 10c and three 1c stamps.) Well, we can easily do this too with another array called, say, P. Then $P[n]$ will be a vector of length 3 that will tell us what stamps we need to make postage for nc. For example, we would have $P[45] = (1, 2, 3)$, meaning that we need one 1c, two 7c, and three 10c stamps. In general, if we have $P[n] = (a, b, c)$, then we take a 1c stamps, b 7c stamps, and c 10c stamps to make nc postage.
The calculation for $S[n]$ doesnt change. All we do is determine which denomination of postage we use and add it to the appropriate value of $P[n-1], P[n-7]$, or $P[n-10]$. The running time is still $O(n)$.

Problem 7
Let $T(n)$ be the number of different binary search trees on n keys. For example, $T(1) = 1$ and $T(2) = 2$. Come up with a recurrence for $T(n)$. Use dynamic programming to compute $T(n)$ for a given n. Run some experiments to determine the asymptotic growth of $T(n)$.

Solution: The recurrence would be

$$T(n) = \sum_{k=1}^{n} T(k-1)T(n-k) \text{ for } n > 1$$

and $T(0) = T(1) = 1$.
We can think of the base case $n = 0$ as the empty tree and $n = 1$ as the tree with a single node. For $n > 1$ we can condition on the kth key being at the root, where $k = 1, \ldots, n$; in which case the left subtree has $k - 1$ keys (there are $T(k-1)$ possible trees) and the right subtree has $n - k$ keys (there are $T(n-k)$ possible trees), so the total number of possible configuration with k at the root is $T(k-1)T(n-k)$.
If we plot $T(n+1)/T(n)$ we see that the value tends to 4, which suggest that the growth order of $T(n)$ is roughly 4^n.
In fact, $T(n)$ is a well studied series called the Catalan numbers and it is known that $T(n) = \frac{4^n}{n^{1.5}\sqrt{\pi}}(1 + o(1))$.