Dynamic Programming (Solutions)

1. (a) 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 proceed with this solution as outlined in the notes. 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, as we did in class, 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.

(b) For comparison, what is the solution to the corresponding Fractional Knapsack problem with the same value of $W_{\max}$ (use the greedy method we discussed for this "Fractional" version)?

Solution: Recall that we compute the value indices and take items in order of the value indices to fill up the knapsack. So I have added another line to the table below for the value indices.

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6
v_i	$3\frac{4}{7}$	$7\frac{1}{2}$	$6\frac{2}{3}$	6

So we select items in the order 2,3,4,1.

Doing so will lead us to the Fractional Knapsack solution consisting of items 2, 3, and $\frac{5}{6}$ of item 4, for a total benefit of $15+20+\frac{5}{6}(36) = 65$.

2. Do the same as above (find solutions to both the $\{0,1\}$ and Fractional Knapsack problems) for this collection of items, where $W_{\max} = 11$.

i	1	2	3	4	5
b_i	14	12	15	20	16
w_i	4	3	8	7	3

Here's the table you would get for the $\{0,1\}$ version of this problem:

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	14	14	14	14	14	14	14	14
2	0	0	12	14	14	14	26	26	26	26	26
3	0	0	12	14	14	14	26	26	26	26	27
4	0	0	12	14	14	14	26	26	26	32	34
5	0	0	16	16	16	28	30	30	30	42	42

From this table, you can see that the maximum benefit for the $\{0,1\}$ Knapsack Problem for this set of items is 42. What is the set of items that achieves this benefit? You can find that by tracing backwards through the table. Since $B[5,11] > B[4,11]$ we see that item 5 must be used in the solution.

Then we want to compare $B[4,8]$ and $B[3,8]$. Since they are the same, we conclude that item 4 is not used in the best solution. Continuing in this way, we can determine that the best solution uses items 5, 2, and 1.

For the Fractional case, we again compute the value indices.

i	1	2	3	4	5
b_i	14	12	15	20	16
w_i	4	3	8	7	3
v_i	$3\frac{2}{3}$	4	$1\frac{7}{8}$	$2\frac{6}{7}$	$5\frac{1}{3}$

So we select items (greedily) in the order 5,2,1,4,3.

This will give us the greedy solution for the Fractional case consisting of items 5,2,1 and $\frac{1}{7}$ of item 4, for a total benefit of $42 + \frac{1}{7}(20) = 44\frac{6}{7}$.

3. (This is one of the problems from the sheet I gave out on the first day of class.) Consider a post office that sells stamps in three different denominations, 1p, 7p, and 10p. Design a dynamic programming algorithm that will find the *minimum number* of stamps necessary for a postage value of N pence. (Note that a greedy type of algorithm won't necessarily give you the correct answer, and you should be able to find an example to show that such a greedy algorithm doesn't work.)

   What is the running time of your algorithm?

   Solution: Here I give quite a detailed solution to this problem. Note that I don't necessarily expect you to provide pseudo-code for your proposed solution. Your description should be sufficiently detailed to to give some argument as to why your solution is correct.

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

$$\begin{aligned} S[0] &= 0 \\ S[1] &= 1 \\ S[2] &= 2 \\ S[3] &= 3 \\ S[4] &= 4 \end{aligned}$$

$$\begin{aligned} S[5] &= 5 \\ S[6] &= 6 \\ S[7] &= 1 \\ S[8] &= 2 \\ S[9] &= 3 \quad \text{and} \\ S[10] &= 1. \end{aligned}$$

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 pence, then we can clearly get it by taking one stamp of 1p, and $S[n-1]$ stamps (of appropriate values) to make up the remaining postage of $(n-1)$p. Or if we take a 7p stamp, then we need $S[n-7]$ stamps (of the right values) to make up the rest, and similarly if we take a 10p stamp. This idea is essentially the heart of the dynamic programming algorithm that we use. The pseudo-code is given below:

```
STAMPS(n)
    ▷ We assume here that n ≥ 0.
    ▷ This procedure will determine the minimum number of stamps
    of values 1p, 7p, and 10p to make up total postage of n pence.
    ▷ Base cases
1   Set up the array S starting as above (with the
    values S[i] for i = 0, ..., 10).
2   if n ≤ 10 then
3           return S[n]
4     else
5           for j ← 11 to n do
6                   S[j] = 1 + min{S[j − 1], S[j − 7], S[j − 10]}
7   return S[n]
```

As you can see, there is one "for" loop in the body of the pseudo-code, and this dominates the running time of the algorithm. Thus, finding $S[N]$ takes time which is in $O(N)$.

Now we can find $S[n]$ easily as above, but knowing this doesn't tell us the exact nature of the stamps we need. Right? It would be nice to know that we can make 45p postage with six stamps, *and* that we need one 1p, two 7p, and three 10p 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 63p with nine 7p stamps, *or* with six 10p and three 1p 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 np. For example, we would have $P[45] = (1, 2, 3)$, meaning that we need one 1p, two 7p, and three 10p stamps. In general, if we have $P[n] = (a, b, c)$, then we take a 1p stamps, b 7p stamps, and c 10p stamps

to make np postage. So, for example $P[7] = (0, 1, 0)$, $P[10] = (0, 0, 1)$, and $P[25] = (1, 2, 1)$. (As I said earlier, $P[n]$ isn't really uniquely defined, but that's not important for us. We just want some way, using the minimum *number* of stamps, to get n pence.)

The calculation for $S[n]$ doesn't 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 modified pseudo-code is given below. The addition of vectors in lines 9, 11, and 13 is done component-wise in the usual way, e.g. $(3, 4, 1) + (0, 1, 0) = (3, 5, 1)$. The running time is still $O(n)$.

```
STAMPS(n)
      ▷ We assume here that n ≥ 0.
      ▷ This procedure will determine the minimum number of stamps of values
      1p, 7p, and 10p to make up total postage of n pence. We also return
      a vector, P[n] which will tell us how many of each stamp is necessary.
      ▷ Base cases
 1    Set up the array S starting as above (with the
      values S[i] for i = 0, ..., 10).
 2    Also set up an array P[i] for i = 0, ..., 10. That is P[0] = (0, 0, 0),
      P[1] = (1, 0, 0), P[2] = (2, 0, 0), ..., P[7] = (0, 1, 0), etc.

 3    if n ≤ 10 then
 4            return S[n] and P[n]
 5      else
 6            for j ← 11 to n do
 7                  S[j] = 1 + min{S[j − 1], S[j − 7], S[j − 10]}
                    ▷ Now set P[j] based on the value of S[j].
                       In other words, which stamp did we use?
 8                  if S[j] == (1 + S[j − 1]) then
 9                        P[j] ← P[j − 1] + (1, 0, 0)
10                  elseif S[j] == (1 + S[j − 7]) then
11                        P[j] ← P[j − 7] + (0, 1, 0)
12                  else
13                        P[j] ← P[j − 10] + (0, 0, 1)
14    return S[n] and P[n]
```

4. (This is also from the initial problem sheet that I gave to you.)

   American coins come in five different values, namely 50-cent, 25-cent, 10-cent, 5-cent, and 1-cent coins. Therefore, if we wanted change for 11 cents (or 11¢), we can do this with one 10¢ coin and one 1¢ coin, or two 5¢ coins and one 1¢, or one 5¢ and 6 1¢ coins, or with eleven 1¢ coins. So there are four ways to give change for 11¢.

   How many ways are there to make change for N¢? (And how can you compute this number *efficiently*?)

   We consider the coins in order 1, 5, 10, 25, 50. We compute the number of ways of making change, first using only 1¢ coins, then using both 1¢

and 5¢ coins, and so forth. Doing the computations in this order, we're using the previously computed information to save us some work (in the form of repeated computations).

So if you want, we can consider defining a 2-dimensional array (table) of the form $A[i, n]$ for $i = 1, \ldots, 5$ and $n = 1 \ldots, N$. The index $i = 3$, for example, tells us that we're using the first three coins to make change (i.e. the 1¢, 5¢, and 10¢ coins). Obviously the value n is the amount for which we're making change. We want to find $A[5, N]$.

First of all, we note that $A[1, n] = 1$ for all $n \leq N$. Why? There's only one way to make change for n¢using only pennies. How many ways are there to make change using 1¢ and 5¢ coins? We can find this using the following bit of pseudo-code.

```
1  for j ← 0 to N Do
2          A[2, j] = ⌊j/5⌋ + 1
```

Verify that this number is correct? (Hint: The +1 comes from the fact that you can use no 5¢ coins.)

Then we can find the values of $A[3, n]$ in this fashion

```
1  for j ← 0 to N Do
2          A[3, j] = A[2, j]
3          for k ← 10 to N by 10 Do        ▷ (Increment counter by 10.)
4                  if (j − k >= 0) Then
5                          A[3, j] = A[3, j] + A[2, j − k]
```

Because this pseudo-code has two "for" loops in it, you can see that it runs in time $O(N^2)$. You should also verify that it correctly computes the number of ways of giving change using 1¢, 5¢ and 10¢ coins.

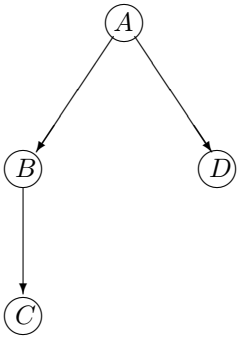
Adding on 25¢ and 50¢ coins is similar to the above pseudo-code, we just increment the counter in the second "for" loop by 25 or 50 as appropriate (and use the appropriate index i in the values of $A[i, n]$).

Overall, this will let us compute the value of $A[5, N]$, the value we want, in time $O(N^2)$.

5. (Adapted from *Algorithm Design* by Jon Kleinberg & Éva Tardos.)

   The current class of students at Sandhurst has an annual picnic (ok, maybe they don't really do this, but for the purposes of this question they do...). This year it happens that the picnic has fallen on a rainy day and the ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism for doing this.

   Each person except for the ranking officer reports to a unique *superior officer*. Thus, the reporting hierarchy can be described as a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we can call v a *direct subordinate* of u. See the picture below, where A is the ranking officer, B and D are direct subordinates of A, and C is a direct subordinate of B.



To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues in this was until everyone is notified. Note that each person in this process will only call direct subordinates, for example A would not be allowed to call C.

We can picture this notification process as divided into rounds. In one round, each person who has already learned of the postponement can call one of his or her subordinates on the phone. The number of rounds it takes for everyone to be notified might depend upon the sequence in which each person calls their direct subordinates. In the given example, it will take only two rounds if A starts by calling B, but will take three rounds if A starts by calling D.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified for a rooted tree T as described above. How can you output a sequence of phone calls (i.e. a schedule for each person) that achieves this minimum number of rounds?

Solution: The recursive type of problems we need to consider naturally come from looking at a node and all of the descendants of that node. In the example above, B has one child, so needs one round to inform its children. If B had three children, then it would need three rounds to inform all of its children.

In general, a vertex wants to inform its children in a "good" order in which to minimize the time needed to inform all of the descendants of itself. Nodes just above leaves will need a number of rounds equal to their children, but vertices with larger heights (closer to the root) will need even more than the number of children, as you need to consider how long is needed to inform the entire subtree rooted at a vertex. You need to consider more complicated trees in your solution method. (Unfortunately, the small tree that I've drawn here might be slightly misleading.)

Let us use a label called $r(v)$ for a vertex v. (I'm using r as I'm thinking that the value tells me how many rounds are necessary until v and all of its descendants have been informed of the news.)

A leaf has label $r(v) = 0$ (if that vertex knows the news, then you need zero rounds until it (and all of its non-existent children) are informed of the news). Nodes just above leaves will have a label

$$r(v) = \text{number of children of } v.$$

Now assume that we have (recursively) calculated values of $r(w)$ for each child of an internal vertex v, and we want to find $r(v)$. (As above, we

know how to find $r(v)$ for vertices with height 1, so we need to consider vertices with height 2 and larger.)

How do we find $r(v)$? We want to inform the children of v in a proper order so that we minimize the value of $r(v)$. Intuitively (I hope), it seems that we want to inform the child, w, of v having largest value of $r(w)$, so that w learns the message first, and then can begin to inform its children as early as possible (while v continues to inform the remaining children, keeping in mind that this process is "parallel" in the sense that each vertex that knows the message can inform one of its children every round).

So, with this in mind, suppose that v has k children, which I will call $w_1, w_2, \ldots, w_k$, and further suppose that I have labeled the children of v so that

$$r(w_1) \geq r(w_2) \geq \cdots \geq r(w_k). \tag{1}$$

(Note that we can have each child w_i inform its parent of the value of $r(w_i)$ when it has completed computing this value, and then v can sort the $r(w_i)$ values once it knows them all.)

Then, v will inform its children in the order $w_1, w_2, \ldots, w_k$. So, child w_i will be informed on round i *after* v becomes informed of the message. Then, the subtree rooted at vertex w_i requires $r(w_i)$ additional rounds until all vertices in that subtree are informed.

Therefore, I claim that

$$r(v) = \max\left\{1 + r(w_1), 2 + r(w_2), 3 + r(w_3), \ldots, k + r(w_k)\right\}. \tag{2}$$

Will Equation (2) actually find the minimum possible value of $r(v)$?

To see that it does, we assume that the values of $r(w_1), \ldots, r(w_k)$ have been correctly computed (recursively). (This should be clear for leaves and for vertices of height 1.) Also, assume that the children of v have been labeled in a manner so that Equation (1) holds.

Then let $i \in \{1, \ldots, k\}$ be a value so that $r(v) = i + r(w_i)$ holds (i.e. w_i is a child that defines the value of $r(v)$). What would happen if we were to swap the value of $r(w_i)$ with some other value $r(w_j)$ for $i \neq j$? (In other words, could I somehow lower the value of $r(v)$ by rearranging the order in which the children of v are informed?)

If we take $j < i$, then let us compare the two values $i + r(w_j)$ and $i + r(w_i)$ (since vertex w_j is now being informed in round i, and $r(w_j)$ rounds are needed to inform all descendants of w_j). By the ordering of the r values of the descendants of v (Equation (1)), we have

$$r(w_j) \geq r(w_i) \qquad \text{hence,} \quad i + r(w_j) \geq i + r(w_i).$$

Therefore, the value of $r(v)$ wouldn't decrease through such a swap of the vertices w_i and w_j.

Suppose, instead, that we swap vertices w_i and w_j for some $j > i$? Then, let us consider the two values $i + r(w_i)$ and $j + r(w_i)$. (Now, vertex w_i is now being informed in round j, and $r(w_i)$ rounds are required to inform all the descendants of w_i.)

Since $j > i$, we have

$$j + r(w_i) > i + r(w_i),$$

so now even more rounds are required to inform w_i and all of that subtree, increasing the value of $r(v)$ if we were to swap the order of informing w_i and w_j!

So swapping the order in which we inform vertices cannot help to reduce the value of $r(v)$ that we find in Equation (2). Thus, Equation (2) computes the correct value of $r(v)$ for an internal vertex v (assuming that v knows the r values of all of its children).

(Note: The argument I have done above is really a proof by induction, although I haven't explicitly use the word "induction" here.)

As for the schedule for informing the nodes, once all of the $r(v)$ values have been computed for all nodes, then assuming that a vertex knows the r values for all its children, it simply informs them in decreasing r value once vertex v receives the news.

Also note that the r values can be computed by basically performing a depth-first search on the graph. Once all descendants of a node v have been visited in the DFS (and each child of v has "reported" its r value to v, then v sorts those r values, and computes the value of $r(v)$ as in Equation (2) and informs its parent).

A depth-first search on a graph with n vertices and m edges can be performed in time $O(m)$. Since a tree has $m = n-1$, and sorting a list of size k can be done in time $O(k \log k)$. If $\Delta =$ max number of children of any node, then we can find all r values in time $O(n\Delta \log \Delta)$, which is the time for a DFS on the tree, together with sorting the (at most Δ) r values at each vertex.

6. Consider the set of weighted intervals given below, where s_i is the start time, f_i is the finish time, and v_i is the value of the interval.

j	s_j	f_j	v_j	$p(j)$
1	0	6	2	0
2	2	10	4	0
3	9	15	6	1
4	7	18	7	1

Solve this instance of the weighted interval scheduling problem, i.e. find a set of (non-conflicting) intervals with maximum total weight.

Solution: Note that the intervals are already sorted by their finish time, and I have added the value $p(j)$ for each interval. Recall that the value $p(j)$ is the largest index of an interval which is compatible with interval j, i.e. interval $p(j)$ could be scheduled together with interval j, but not interval $p(j)+1$ as that would conflict with interval j. (Of course, $p(j) = 0$ if there is no such interval.)

Now recall that $Opt(j)$ was defined to be the best solution that is obtainable using intervals $1, \ldots, j$, where we define $Opt(0) = 0$. Then we have that

$$Opt(j) = \max\{Opt(j-1), v_j + Opt(p(j))\}.$$

Therefore, we have

$$\begin{aligned}
Opt(0) &= 0 \\
Opt(1) &= \max\{Opt(0),\ v_1 + Opt(p(1))\} = 2 \\
Opt(2) &= \max\{Opt(1),\ v_2 + Opt(p(2))\} = \max\{2,\ 4+0\} = 4 \\
Opt(3) &= \max\{Opt(2),\ v_3 + Opt(p(3))\} = \max\{4,\ 6+2\} = 8 \\
Opt(4) &= \max\{Opt(3),\ v_4 + Opt(p(4))\} = \max\{8,\ 7+2\} = 9
\end{aligned}$$

So there is a schedule that gives us a weight (or value) of 9. It's easy enough to see, in this case, that we schedule intervals 1 and 4 to get this value.

In general, if you want to reconstruct the actual schedule that will achieve the maximum value, then you start with the value of $Opt(n)$ (for n intervals) and determine if $Opt(n) = Opt(n-1)$ or if $Opt(n) = v_n + Opt(p(n))$. In the first case, interval n isn't used and then we proceed to the value $Opt(n-1)$ and do the same to continue finding the schedule.

In the second case, interval n is scheduled, and then we proceed to examine task $p(n)$ (and the value of $Opt(p(n))$) to determine the next (i.e. previous) interval to schedule.

7. Do the same as the previous problem for this collection of weighted intervals.

j	s_j	f_j	v_j	$p(j)$
1	1	4	4	0
2	2	6	2	0
3	3	9	1	0
4	5	10	6	1
5	7	14	3	2
6	12	16	1	4
7	9	18	5	3
8	15	20	3	5

Solution: Again, I have added in the values of $p(j)$ for each interval in the table, as the tasks are already sorted by their finishing times.

In this case we have these values:

$$\begin{aligned}
Opt(0) &= 0 \\
Opt(1) &= \max\{Opt(0),\ v_1 + Opt(p(1))\} = \max\{0,\ 4+0\} = 4 \\
Opt(2) &= \max\{Opt(1),\ v_2 + Opt(p(2))\} = \max\{4,\ 2+0\} = 4 \\
Opt(3) &= \max\{Opt(2),\ v_3 + Opt(p(3))\} = \max\{4,\ 1+0\} = 4 \\
Opt(4) &= \max\{Opt(3),\ v_4 + Opt(p(4))\} = \max\{4,\ 6+4\} = 10
\end{aligned}$$

$$
\begin{aligned}
Opt(5) &= \max\{Opt(4),\ v_5 + Opt(p(5))\} = \max\{10,\ 3+4\} = 10 \\
Opt(6) &= \max\{Opt(5),\ v_6 + Opt(p(6))\} = \max\{10,\ 1+10\} = 11 \\
Opt(7) &= \max\{Opt(6),\ v_7 + Opt(p(7))\} = \max\{11,\ 5+4\} = 11 \\
Opt(8) &= \max\{Opt(7),\ v_8 + Opt(p(8))\} = \max\{11,\ 3+10\} = 13
\end{aligned}
$$

So the maximum value obtainable is 13, which comes from scheduling intervals 8, 4, and 1.

8. Suppose you're managing construction of billboards on the Rocky & Bullwinkle Memorial Highway, a heavily traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1 < x_2 < \cdots < x_n$, each in the interval $[0, M]$, specifying their position in miles measured from the western end of the road. If you place a billboard at position x_i, you receive a revenue of $r_i > 0$.

   Regulations imposed by the Highway Department require that no two billboards be within five miles or less of each other. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.

   For example, suppose $M = 20$ and $n = 5$ with

   $$\{x_1, x_2, x_3, x_4, x_5\} = \{6, 7, 12, 13, 14\}$$

   and

   $$\{r_1, r_2, r_3, r_4, r_5\} = \{5, 6, 5, 3, 1\}.$$

   Then the best solution is to place billboards at x_1 and x_3 to achieve a revenue of 10.

   Describe a (dynamic programming) procedure to find a solution for this problem. What is the running time of your procedure (this should be polynomial in n)?

   Hint: As a first step towards the solution, define (similar to the Weighted Interval Scheduling Problem) $e(j)$ to be the easternmost site that is more than 5 miles away from x_j. In other words, if you place a billboard at x_j, then $x_1, x_2, \ldots, x_{e(j)}$ are also valid places to place billboards (subject to the same restriction about distances). Note that computing the $e(j)$ values takes time $O(n)$.

   Solution: The hint contains much of the solution. With the $e(j)$ defined as above, we also define $Opt(j)$ to be the best revenue obtainable using sites $\{x_1, x_2, \ldots, x_j\}$. (Note that we define $Opt(0) = 0$, and $e(j) = 0$ should have an obvious meaning, namely there is no other location smaller than x_j that is compatible with x_j, e.g. if $x_3 = 4$, then by placing a billboard at site x_3, we cannot use x_1 or x_2 as they would be within five miles of x_3. In that case we can define $e(3) = 0$.)

   Then, we see that $Opt(1) = r_1$ and also

   $$Opt(j) = \max\{r_j + Opt(e(j)), Opt(j-1)\}.$$

Why? If we use location x_j, then we combine this with the best solution from sites $x_1, x_2, \ldots, x_{e(j)}$. Otherwise if we don't use x_j then we take the best solution from sites $x_1, x_2, \ldots, x_{j-1}$. Since we're trying to maximize our revenue, we take the better (maximum) of those two options.

9. Longest Increasing Subsequence. (This is a classical dynamic programming problem.) Given a sequence of real numbers $A_1, A_2, \ldots, A_n$, determine a subsequence (not necessarily contiguous) of maximum length in which the values form a *strictly increasing sequence.*

   For example, given the sequence of integers

   $$-3, 4, 14, 0, -1, 3, 5, 2, 5, 10$$

   the subsequence consisting of $4, 5, 10$ is an increasing subsequence, as is $-3, 0, 3, 5, 10$. Your task is to find a longest subsequence so that the values are strictly increasing. (There might be repeated numbers in the original sequence, but there can be no repeats in a longest subsequence since it must be strictly increasing.)

   Hint: Define $L(j)$ to be the length of the longest increasing subsequence that ends at (and includes) position j. How can you find $L(j)$ in terms of the values $L(i)$ for $i < j$? Once you know the $L(j)$ values, what's the answer to the original problem? And how long does it take to compute all of the $L(j)$ values?

   Solution: As suggested above, we define $L(j)$ to be the length of longest increasing subsequence that ends at (and includes) position j. If we are able to compute these $L(j)$ values, then the solution to our problem is equal to $L = \max\{L(1), L(2), \ldots, L(n)\}$. Why? Because the longest increasing subsequence must end at (and include) one of the positions $1, 2, \ldots, n$.

   We note that $L(1) = 1$, since the single value A_1 is an "increasing" subsequence. Also, $L(2) = 1$ or 2, depending upon whether $A_1 \geq A_2$ or $A_1 < A_2$ (recall that the sequence must be strictly increasing).

   If we have computed $L(1), L(2), \ldots, L(i)$ for some i, how do we use these values to find $L(i+1)$? We compute

   $$L(i{+}1) = 1{+}\max\{L(j) : A_j < A_{i+1}\}, \text{ or possibly } L(i{+}1) = 1 \text{ (see below).}$$

   In other words, check all A_j, find those that are less than A_{i+1}. We do this because if $A_j < A_{i+1}$, then we can increase the longest subsequence that ends at A_j by one more value (namely, adding A_{i+1} to that subsequence). Note that if there are no values A_j that are less than A_{i+1}, then we define $L(i+1) = 1$ (as for $L(1)$).

   Computing $L(i+1)$ requires us to check all of the values $A_1, \ldots, A_i$, i.e. $O(i)$ time. So, in total, computing all of the $L(i)$ values takes $O(1{+}2{+}3{+}\cdots{+}n) = O(n^2)$ time. Then finding the value $L = \max\{L(1), L(2), \ldots, L(n)\}$ takes $O(n)$ time. So finding the value L takes $O(n^2)$ time.

10. Balanced Partition. Suppose that you have a set, $A = \{a_1, \ldots, a_n\}$, of n integers, each of which is between 0 and K (inclusive). Your goal is to find

a partition of A into two sets S_1 and S_2 (so $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$) that *minimizes*

$$|\text{sum}(S_1) - \text{sum}(S_2)|,$$

where $\text{sum}(S_i)$ equals the sum of the values in the set S_i.

Hint: Define $V(i, j) = 1$ if there is some subset of $\{a_1, \ldots, a_i\}$ that sums up to the value j (i.e. some collection from the first i integers sums up to j), otherwise set $V(i, j) = 0$.

How many different values of $V(i, j)$ do you need to compute (in terms of n and K)? How can you compute the $V(i, j)$ values if you know all of the $V(i-1, j)$ values? How can you use the collection of $V(i, j)$ values to answer the original question (and find the minimum value of $|\text{sum}(S_1) - \text{sum}(S_2)|$)?

Solution: Since there are n different numbers, each of which is at most K, the sum of all of the n numbers is at most nK (i.e. j can range from 0 to nK for each value of i). Hence, there are at most $O(n^2K)$ values $V(i, j)$ to compute. (This is effectively the running time to find a solution to this Balanced Partition problem.)

For $i = 1$, we have $V(1, a_1) = 1$ and $V(1, j) = 0$ for $j \neq a_1$.

For $i \geq 2$, how can we find $V(i, j)$ given that we have all of previously computed values (i.e. all of the $V(1, j), V(2, j), \ldots, V(i-1, j)$ values for all j)?

First of all, we have $V(i, j) = 1$ if $V(i-1, j) = 1$ or if $V(i-1, j-a_i) = 1$. (Why?) Otherwise, $V(i, j) = 0$.

So after computing all of the $V(i, j)$ values, what do we do?

Let $T = \frac{1}{2}\sum_{i=1}^{n} a_i$. Then T is half of the sum of all the values in the set A. If T is an integer, we can first check to see if $V(n, T) = 1$. If this is the case, then there is a partition S_1, S_2 such that $|\text{sum}(S_1) - \text{sum}(S_2)| = 0$. (Why?)

If T is not an integer, or if T is an integer but $V(n, T) = 0$, then what do we do? For each value j with $V(n, j) = 1$, this means that there are two sets S_1, S_2 such that $\text{sum}(S_1) = j$ and $\text{sum}(S_2) = 2T - j$. For those particular sets, we have $|\text{sum}(S_1) - \text{sum}(S_2)| = |j - (2T - j)| = |2j - 2T| = |2T - 2j|$.

We're trying to minimize that value, so we compute

$$\min\{|2T - 2j| : V(n, j) = 1 \text{ and } j \leq T\}$$

to find this minimum value. (We need only consider the values $j \leq T$ in the minimum because of the symmetry of the problem, i.e. there is a set S_1 with $\text{sum}(S_1) = j$ if and only if there is a set S_2 with $\text{sum}(S_2) = 2T - j$. Furthermore, we would likely consider the values $V(n, j)$ in decreasing order of j starting from T (or the closest integer to T if it is a fraction) since we want to minimize the difference.)

This last part of computing the minimum takes time $O(T) \in O(n^2K)$, so overall the whole algorithm can run in time $O(n^2K)$.