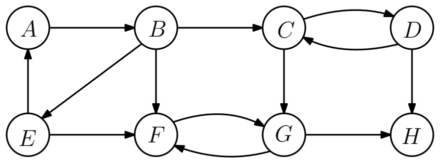
COMP 3804 — Solutions Assignment 3

Question 1: Write your name and student number.

Solution: Erling Haaland, 9

Question 2: Consider the following directed graph:



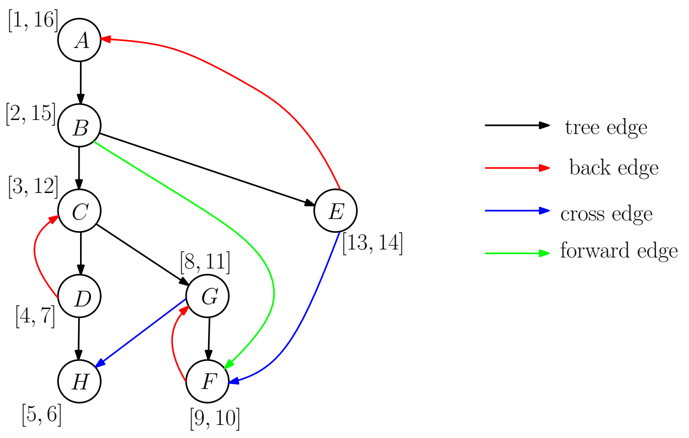
(2.1) Draw the *DFS*-forest obtained by running algorithm DFS. Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the *DFS*-forest, give the *pre*- and *post*-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically first.

(2.2) Draw the *DFS*-forest obtained by running algorithm DFS. Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the *DFS*-forest, give the *pre*- and *post*-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically last.

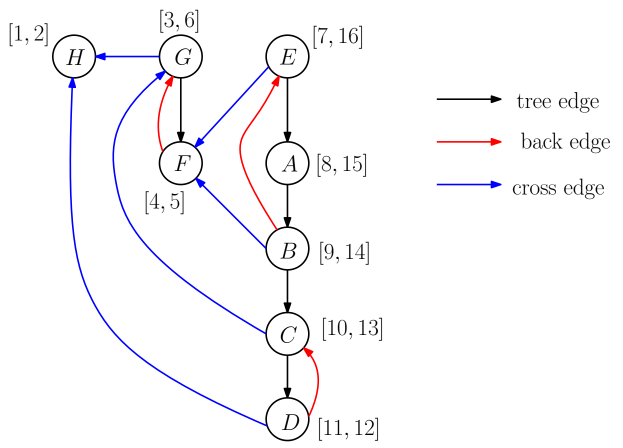
```
Algorithm DFS($G$):
for each vertex $v$
do $visited(v) = false$
endfor;
$clock = 1$;
for each vertex $v$
do if $visited(v) = false$
    then EXPLORE($v$)
    endif
endfor

Algorithm EXPLORE($v$):
$visited(v) = true$;
$pre(v) = clock$;
$clock = clock + 1$;
for each edge $(v, u)$
do if $visited(u) = false$
    then EXPLORE($u$)
    endif
endfor;
$post(v) = clock$;
$clock = clock + 1$
```

Solution: We start with **(2.1)**. In case there is more than one choice, we pick the alphabetically smallest one. Thus, algorithm DFS(G) starts by calling EXPLORE(A). Here is the resulting *DFS*-forest, which consists of one tree:



Next we do **(2.2)**. In case there is more than one choice, we pick the alphabetically largest one. Thus, algorithm DFS(G) starts by calling EXPLORE(H). Here is the resulting *DFS*-forest, which consists of three trees:



Question 3: Let $G = (V, E)$ be a directed graph. After algorithm DFS(G) has terminated, each vertex has a *pre*- and *post*-number. Let u and v be two distinct vertices in V. Prove the following:

- If $pre(u) < pre(v) < post(u)$, then there is a directed path in G from u to v.
- Assume that G is directed and acyclic. If $post(u) < pre(v)$, then there is no directed path in G from u to v.

Solution: We start with the first part. Assume that $pre(u) < pre(v) < post(u)$. Then EXPLORE($v$) is called within EXPLORE(u). Thus, EXPLORE(v) will finish before EXPLORE(u) finishes, and we have $pre(u) < pre(v) < post(v) < post(u)$.

Since we know that EXPLORE(v) is called within EXPLORE(u), there is a sequence $v_1, v_2, \ldots, v_k = v$ of vertices such that:

- EXPLORE(u) calls EXPLORE(v_1); thus, (u, v_1) is an edge.
- EXPLORE(v_1) calls EXPLORE(v_2); thus, (v_1, v_2) is an edge.
- EXPLORE(v_2) calls EXPLORE(v_3); thus, (v_2, v_3) is an edge.
- EXPLORE(v_3) calls EXPLORE(v_4); thus, (v_3, v_4) is an edge.
- Etc.
- EXPLORE(v_{k-1}) calls EXPLORE(v_k); thus, (v_{k-1}, v_k) is an edge.

It follows that

$$u \to v_1 \to v_2 \to v_3 \to \cdots \to v_{k-1} \to v_k = v$$

is a directed path from u to v. (**Exercise:** Convince yourself that this may not be the shortest path from u to v.)

For the second part, we assume that G is directed and acyclic. We also assume that $post(u) < pre(v)$, i.e., EXPLORE(u) finishes before EXPLORE(v) starts. Note that $pre(u) < post(u) < pre(v) < post(v)$.

We refer to $[pre(u), post(u)]$ as the *interval* of vertex u. We are given that u's interval is to the left of v's interval. We want to show that there is no directed path from u to v.

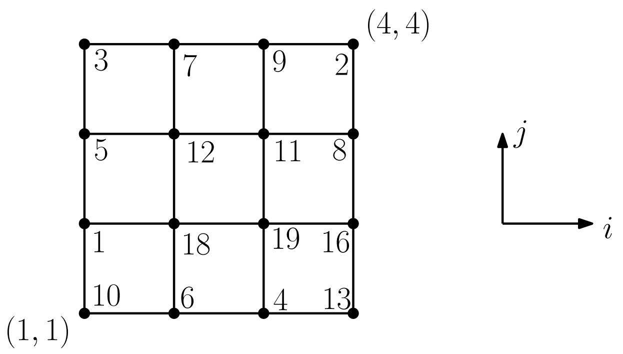
Let a be an arbitrary vertex such that a's interval is to the left of v's interval. Consider any edge (a, b).

- If (a, b) is a tree edge or a forward edge, then b's interval is contained in a's interval. Therefore, b's interval is also to the left of v's interval.
- If (a, b) is a cross edge, then b's interval is to the left of a's interval. Therefore, b's interval is also to the left of v's interval.
- (a, b) cannot be a back edge, because G is acyclic; we have seen this in class.

What does this imply: Vertex u's interval is to the left of v's interval. If we follow an arbitrary directed path, starting at v, then for every vertex a that we visit, a's interval is to the left of v's interval. Therefore, any such path will never reach vertex v. In other words, there is no directed path from u to v.

Question 4: Let $n \geq 2$ be an integer and let G be the $n \times n$ undirected grid graph: The vertices of G are the grid points (i, j) for $1 \leq i \leq n$ and $1 \leq j \leq n$. Each vertex (i, j) has neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$ (provided their coordinates are in the set $\{1, 2, \ldots, n\}$).

Each vertex of this graph stores a number; we assume that all these n^2 numbers are distinct. A vertex is called *awesome*, if the number stored at this vertex is larger than the numbers stored at all of its neighbors. In the example below, $n = 4$ and both vertices $(1, 1)$ and $(3, 2)$ are awesome.



- Prove that there always exists at least one awesome vertex.
- Give an algorithm that finds an awesome vertex in $O(n)$ time. Note that the graph G has n^2 vertices and $\Theta(n^2)$ edges.

Solution: An awesome vertex is actually called a *local maximum*. Why is there always a local maximum:

- First proof: This is obvious, because the largest number is a local maximum.
- Second proof: Use *hill climbing*. Start at an arbitrary vertex. If it is a local maximum, then we are done. Otherwise, move to its largest neighbor. Now repeat this procedure. Since there are only a finite number of vertices, this procedure will terminate. The figure below shows that this procedure may take $\Theta(n^2)$ time if we start at the top left vertex. Every empty cell has a negative number. In this figure, we have switched to matrix notation.

1	2	3	4	5	6
					7
13	12	11	10	9	8
14					
15	16	17	18	19	20
					21

To describe the algorithm, we switch to matrix notation. Thus, we have an $n \times n$ matrix and each cell stores a number. A cell is a local maximum if its entry is larger than the entries in its neighbors.

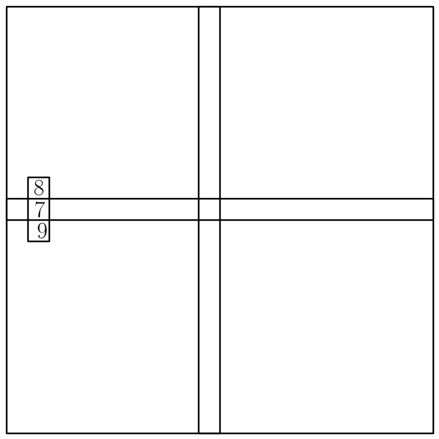
Since recursion is our best friend, let us try to obtain an algorithm whose running time satisfies

$$T(n) = O(n) + T(n/2).$$

Then unfolding, or the Master Theorem, tells us that $T(n) = O(n)$.

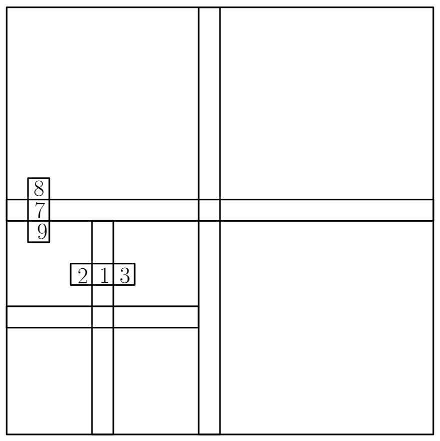
Thus, the algorithm takes as input an $n \times n$ matrix that has n^2 entries, and it finds a local maximum in this matrix. In $O(n)$ time, the algorithm reduces this to computing a local maximum in an $n/2 \times n/2$ matrix. In other words, after spending $O(n)$ time, the algorithm reduces both the number of rows and the number of columns by a factor of two.

How do we do this: The middle row and middle column divide the matrix into four submatrices, each having (about) $n/2$ rows and $n/2$ columns. We refer to the middle row and the middle column as a *cross*. See the figure below.

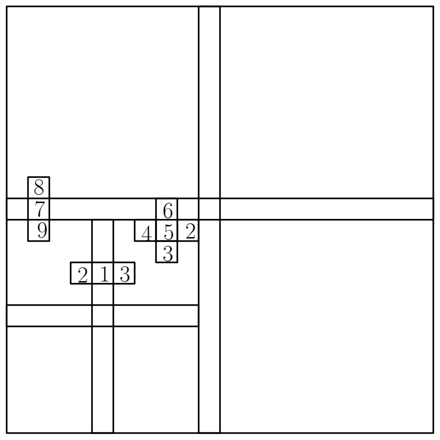


In $O(n)$ time, we scan all entries in this cross and find the largest one, say 7. In $O(1)$ time, we check if this is a local maximum. If it is, then we are done. Otherwise, this largest element cannot be the center of the cross. Assume, as in the figure, that 7 is in the middle row. We know that at least one of its north and south neighbors is larger than 7. In the figure, both are larger. Since $9 > 8$, it makes sense to recurse on the south-west submatrix (excluding the entries that are part of the cross). Note that any local maximum in this submatrix that is at least equal to 9 is also a local maximum in the entire matrix. (However, the same is true for any local maximum in the north-west submatrix that is at least equal to 8.)

Let us make one more step in the south-west submatrix. We scan all entries in its cross and find the largest one, say 1. If this is a local maximum, then we are done, Otherwise, it cannot be the center of the cross. Assume, as in the figure, that 1 is in the column of its cross. We know that at least one of its east and west neighbors is larger than 1. In the example, both are larger.



Since $3 > 2$, it is tempting to recurse in the north-east subsubmatrix. If we do this, then the algorithm will find a local maximum in this subsubmatrix. Is this a local maximum in the entire matrix? It may happen that the local maximum found by the algorithm is 5, as in the figure below. Note that 5 is indeed a local maximum in the north-east subsubmatrix, but it is not a local maximum in the entire matrix. (In the figure, not all numbers are distinct. It is easy to change the numbers so that they are all distinct.)



What we have to do: Recurse in the subsubmatrix that contains the larger of the three numbers 9, 2, and 3. Thus, in the figure, we recurse in the north-west subsubmatrix. Why:

- If we are able to find a local maximum in this subsubmatrix that is at least equal to 9, then it is a local maximum in the entire matrix.

What have we learned:

- In one iteration, we find the neighbors that are larger than the largest element in the cross; these are 2 and 3 in the figure above. We also remember the special number 9. We recurse in the submatrix that contains the largest of these three numbers.

Now we can describe the algorithm. We denote the matrix by M, its rows and columns are numbered $1, 2, \ldots, n$. We denote by $M[\ell, r; \ell', r']$ the submatrix that is bounded by rows ℓ and r, and by columns ℓ' and r'.

- In a generic call, the algorithm gets row numbers ℓ and r, column numbers ℓ' and r', and a number y.
- $r - \ell = r' - \ell'$, possibly plus or minus one. Thus, $M[\ell, r; \ell', r']$ is (almost) square.
- We define the *border* of the submatrix $M[\ell, r; \ell', r']$ to be all cells just outside this submatrix; these are the green cells in the figure below.
- The *precondition* is:
  - Either $y = -\infty$, or y is finite and occurs in $M[\ell, r; \ell', r']$.
  - y is larger than all entries in the border.
- The *postcondition* is: The algorithm returns a local maximum in $M[\ell, r; \ell', r']$ that is at least y.

Note: The precondition implies that the output is a local maximum in the entire matrix.

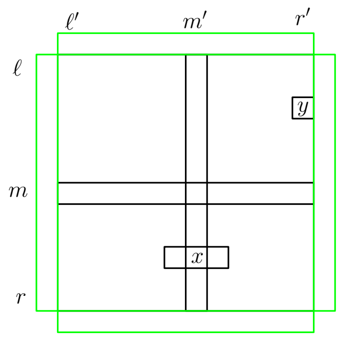
The algorithm does the following:

Base case: At least one of $r - \ell$ and $r' - \ell'$ is at most two.

The algorithm finds, in $O(1)$ time, the largest element in $M[\ell, r; \ell', r']$ and returns it. Note that this largest element is at least y; thus, the postcondition is satisfied.

One iteration in the non-base case: Assume that both $r - \ell$ and $r' - \ell'$ are at least three.

Let $m = \lfloor (\ell + r)/2 \rfloor$ be the middle row and let $m' = \lfloor (\ell' + r')/2 \rfloor$ be the middle column. The algorithm finds the largest element, say x, in the cross of $M[\ell, r; \ell', r']$.



- If x is a local maximum in $M[\ell, r; \ell', r']$: Return x and terminate.

- If x is not a local maximum in $M[\ell, r; \ell', r']$: We know that x is not the center of the cross. Assume that x is in the column of the cross. (The case when x is in the row of the cross is symmetric.) Note that both the west neighbor and the east neighbor of x are in $M[\ell, r; \ell', r']$.
  - If the west neighbor of x is larger than y: Set y to this west neighbor.
  - If the east neighbor of x is larger than y: Set y to this east neighbor.
  - Update ℓ, r, ℓ', and r', such that y is in $M[\ell, r; \ell', r']$. For example, if y is the west neighbor of x, then ℓ changes to $m+1$ and r' changes to $m'-1$.
  - **Exercise:** Convince yourself that the precondition for the next iteration is satisfied.

To find a local maximum in the entire matrix, we call the algorithm with $\ell = 1$, $r = n$, $\ell' = 1$, $r' = n$, and $y = -\infty$.

Remark: If y is finite, then we have to store the two indices i and j such that $M(i, j) = y$. This can easily be added to the algorithm.

It is clear that one iteration takes time proportional to the number of rows in the current submatrix. After this iteration, the number of rows (and columns) is reduced by a factor of two. Thus, the running time $T(n)$ satisfies $T(n) = O(n) + T(n/2)$.

Question 5: Let $G = (V, E)$ be a directed acyclic graph that is given to you using adjacency lists. We say that G is *nice* if for every two distinct vertices u and v, there is a directed path from u to v, or there is a directed path from v to u.

Give an algorithm that decides in $O(|V| + |E|)$ time whether G is nice. As always, justify your answer.
Hint: Find some examples of directed acyclic graphs with four vertices that are nice, and find some examples that are not nice. What property of their topological orderings distinguishes them?

Solution: Let $V = \{v_1, v_2, \ldots, v_n\}$ be an arbitrary topological sort of the graph G.

Claim: G is nice if and only if for all $i = 1, 2, \ldots, n-1$, $(v_i, v_{i+1}) \in E$.

1. Assume there is an index i such that $(v_i, v_{i+1}) \notin E$. Then there is no path from v_i to v_{i+1}, and there is no path from v_{i+1} to v_i. Thus, G is not nice.
2. Assume that for all $i = 1, 2, \ldots, n-1$, $(v_i, v_{i+1}) \in E$. Take two arbitrary distinct vertices u and v. Let i be the index such that $u = v_i$ and let j be the index such that $v = v_j$.
   (a) If $i < j$, then
   $$v_i \to v_{i+1} \to v_{i+2} \to \cdots \to v_{j-1} \to v_j$$
   is a path from u to v.

(b) If $j < i$, then

$$v_j \to v_{j+1} \to v_{j+2} \to \cdots \to v_{i-1} \to v_i$$

is a path from v to u.

From this claim, the algorithm should be clear:

Step 1: Run the topological sorting algorithm that we have seen in class. It runs in $O(|V| + |E|)$ time. This gives the numbering $v_1, v_2, \ldots, v_n$ of the vertices.

Step 2: For each $i = 1, 2, \ldots, n-1$, traverse the adjacency list of vertex v_i and check that you see vertex v_{i+1}. This takes time proportional to

$$\sum_{i=1}^{n-1} (1 + \text{outdegree}(v_i)) = |V| - 1 + |E| \leq |V| + |E|.$$

Question 6: In class, we have seen a data structure for the UNION − FIND problem that stores each set in a linked list, with the header of the list storing the name and size of the set. Using this data structure, any operation FIND(x) takes $O(1)$ time, whereas any operation UNION(A, B, C) takes $O(\min(|A|, |B|))$ time.

Consider the same data structure, except that the header of each list only stores the name of the set (and not the size). Show that, in this new data structure, any operation FIND(x) can be performed in $O(1)$ time, and any operation UNION(A, B, C) can still be performed in $O(\min(|A|, |B|))$ time.

Solution: Each set A is stored in a list:

- The header stores the name of the set.
- Each other node stores one element of the set, a pointer to the next node in the list, and a pointer to the header.
- There are pointers to the header and tail of the list.

For FIND(x), we get a pointer to the node storing x. We follow the pointer to the header of the list and return the name of the set stored at the header. This obviously takes $O(1)$ time.

For UNION(A, B, C), we do the following:

```
$a$ = header of the list storing $A$;
$b$ = header of the list storing $B$;
**while** $a \neq$ tail and $b \neq$ tail
**do** $a = next(a)$;
    $b = next(b)$
**endwhile**;
```

```
if $a$ = tail
then // comment: We know that $|A| \leq |B|$
        add the list storing $A$ at the end of the list storing $B$ and make changes
        as we did in class
else // comment: We know that $|A| > |B|$
       add the list storing $B$ at the end of the list storing $A$ and make changes
       as we did in class
endif
```

In the while-loop, we simultaneously walk along the lists storing A and B. The running time of the loop is determined by the number of steps made by the first of a and b to reach the end of its list. This means that the loop takes $O(\min(|A|, |B|))$ time. As we have seen in class, the rest of the algorithm, in which we perform the actual UNION-operation, also takes $O(\min(|A|, |B|))$ time.

Question 7: A sequence $(b_1, b_2, \ldots, b_k)$ of numbers is called *amazing*, if there is an index i with $1 \leq i \leq k$, such that $(b_1, b_2, \ldots, b_i)$ is increasing and $(b_i, b_{i+1}, \ldots, b_k)$ is decreasing.

Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence of numbers. Give an algorithm that computes, in $O(n^2)$ time, the length $LAS(S)$ of a longest amazing subsequence of S. The numbers in the subsequence are not necessarily consecutive in S.

For example, if

$$S = (10, 22, 9, 33, 21, 50, 41, 60, 80, 1),$$

then $LAS(S) = 7$, because $(10, 22, 33, 50, 60, 80, 1)$ is a longest amazing subsequence.

As always, argue why your algorithm is correct. You my use any result that was presented in class.

Solution: Assume we know an optimal solution $(b_1, b_2, \ldots, b_k)$. Let b_j be the largest number in this solution, and let i be the index such that $b_j = a_i$. Then:

1. $(b_1, b_2, \ldots, b_j = a_i)$ is a longest increasing subsequence (LIS) in $(a_1, a_2, \ldots, a_i)$ that ends with a_i.
2. $(a_i = b_j, b_{j+1}, \ldots, b_k)$ is a longest decreasing subsequence (LDS) in $(a_i, a_{i+1}, \ldots, a_n)$ that starts with a_i.

We define

1. $LIS(i)$ to be the length of an LIS in the sequence $(a_1, a_2, \ldots, a_i)$ that ends with a_i, and
2. $LDS(i)$ to be the length of an LDS in the sequence $(a_i, a_{i+1}, \ldots, a_n)$ that starts with a_i.

Then we have to compute

$$LAS(S) = \max_{i=1,\ldots,n} (LIS(i) + LDS(i) - 1) .$$

The algorithm will first compute all values $LIS(i)$ and then compute all values $LDS(i)$. Once all these values are known, the algorithm will spend $O(n)$ time to compute the value of $LAS(S)$.

We have seen in class how to compute all values $LIS(i)$: Define a directed acyclic graph with vertex set $\{v_0, v_1, \ldots, v_{n+1}\}$. Vertex v_0 stores $-\infty$ as its key, vertex v_{n+1} stores ∞ as its key, and each other vertex v_i stores a_i as its key. There is a directed edge (v_a, v_b) if and only if $a < b$ and the key of v_a is less than the key of v_b. Note that $LIS(i)$ is the length of a longest path from vertex v_0 to vertex v_i minus one. We have seen in class that, using dynamic programming, we can compute these longest path lengths, for all i, in $O(n^2)$ total time.

In a symmetric way, we can compute all values $LDS(i)$ in $O(n^2)$ total time.

Question 8: After having taken many flights with ZoltanJet, Alma is ready to redeem her frequent flyer points: Alma can choose from a wide selection of beers! Each beer costs a certain number of points. Of course, being a beer connoiseur, each beer has a value to Alma. For example, Heineken has a low value, whereas Minerva Stout has a high value. Which beers does Alma choose?

There are n types $B_1, B_2, \ldots, B_n$ of beer. For each i with $1 \leq i \leq n$,

- it costs p_i points to acquire beer B_i,
- the value of beer B_i is equal to v_i.

Alma has P points to spend. We denote the beers that she chooses by the subset $I \subseteq \{1, 2, \ldots, n\}$ of their indices. (For each i, Alma cannot choose more than one bottle of beer B_i.) For example, choosing beers B_3, B_5, B_9 is denoted by $I = \{3, 5, 9\}$.

Since Alma has P points, we must have

$$\sum_{i \in I} p_i \leq P, \tag{1}$$

i.e., the total cost of all beers chosen is at most P. At the same time, Alma wants to maximize the total value of all chosen beers, i.e., choose I such that

$$\sum_{i \in I} v_i \tag{2}$$

is maximized.

To summarize, the input consists of two sequences $p_1, p_2, \ldots, p_n$ and $v_1, v_2, \ldots, v_n$ of positive integers, and a positive integer P. You may assume that each p_i is at most P. The goal is to compute a subset I of $\{1, 2, \ldots, n\}$ such that (1) is satisfied and the summation in (2) is maximized.

Give a dynamic programming algorithm (in pseudocode) that solves this problem in $O(nP)$ time. As always, argue why your algorithm is correct.
Hint: All input values are positive *integers*. Consider $S(i, j)$, which is the value of an optimal solution if Alma chooses beers from the set $\{B_1, B_2, \dots, B_i\}$ and she can spend at most j points.

Solution: We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

Step 1: Show that there is optimal substructure. Assume we know the optimal solution for the entire problem.

1. Assume we do not include beer B_n. Then the value of the solution is the same as the value of the optimal solution where we can choose from the beers $B_1, B_2, \dots, B_{n-1}$ and spend up to P points.

2. Assume we do include beer B_n. Then the value of the solution is equal to v_n plus the largest value we can obtain by choosing from the beers $B_1, B_2, \dots, B_{n-1}$ and spending up to $P - p_n$ points.

Step 2: Set up a recurrence relation. For $i = 0, 1, \dots, n$ and $j = 0, 1, \dots, P$, let $S(i, j)$ be the largest possible value if we can choose from the beers $B_1, B_2, \dots, B_i$ and spend at most j points.

We want to compute the value $S(n, P)$.

1. For $j = 0, 1, \dots, P$, $S(0, j) = 0$.

2. For $i = 0, 1, \dots, n$, $S(i, 0) = 0$.

3. For $i = 1, \dots, n$ and $j = 1, \dots, P$:

   (a) If $p_i > j$, then $S(i, j) = S(i-1, j)$, because we do not have enough points to choose beer B_i.

   (b) If $p_i \le j$, then

$$S(i, j) = \max\left(S(i-1, j), S(i-1, j - p_i) + v_i\right).$$

Step 3: Solve the recurrence, in a bottom-up order.

```
for $j = 0$ to $P$ do $S(0, j) = 0$ endfor;
for $i = 0$ to $n$ do $S(i, 0) = 0$ endfor;
for $i = 1$ to $n$
do for $j = 1$ to $P$
    do if $p_i > j$
        then $S(i, j) = S(i - 1, j)$
        else if $S(i - 1, j) < S(i - 1, j - p_i) + v_i$
             then $S(i, j) = S(i - 1, j - p_i) + v_i$
             else $S(i, j) = S(i - 1, j)$
             endif
        endif
    endfor
endfor;
return $S(n, P)$
```

The first for-loop takes $O(P)$ time, the second for-loop takes $O(n)$ time, whereas the nested for-loops take $O(nP)$ total time.

Exercise: Convince yourself that this running time is not polynomial in the length of the input.