1 Dynamic Programming

Dynamic Programming - The problem-solving process that relies on solving smaller subproblems. There are four general steps to solving a dynamic programming problem:

1. Find the right subproblem to solve. This is often referred to as "the most crucial part of a dynamic programming problem". These subproblems are often one of a few standard choices that are commonly seen in dynamic programming.

- **Linear** The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots x_i$. This can be seen on many problems where subproblems are solved based on previous subproblems. Examples include: The Fibbonacci Sequence, Longest Increasing Subsequence (LIS). There are $O(n)$ subproblems
- **Dual Input** The inputs are $x_1, x_2, \ldots x_n$ and $y_1, y_2, \ldots y_n$ and a subproblem is $x_1, x_2, \ldots x_i$ and $y_1, y_2, \ldots y_j$. Examples include: Longest Common Subsequence (LCS), Knapsack. There are $O(mn)$ subproblems
- **Input Sections** The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_i, x_{i+1}, \ldots x_{j-1}, x_j$. Unlike linear, these problems must consider partial sequences of the input. Examples include: Chain Matrix Multiplication, Edit Length (not discussed in lecture). There are $O(n^2)$ subproblems.
- **Trees** The input is a rooted tree. A subproblem is a rooted subtree. If the tree is not rooted, a root can usually be picked randomly. Examples include: Independent Sets in a tree. There are $O(|V|)$ subproblems.

2. Find a recursive solution. This is how dynamic programming works - problems have their solutions defined in terms of smaller subproblems. There are a few ways that a solution to a subproblem can be defined.

- **Constant time arithmatic** This will access a previous solution (or previous solutions) and perform arithmatic to get the solution to this subproblem. This is seen in the Fibonacci Sequence, but not commonly otherwise. The work done in this subproblem is $O(1)$
- **Constant time comparisson** This will make a choice between possible previous solutions and choose one over the others. This is most commonly choosing the minimum or maximum of only two options. *Example:* In the Knapsack problem, subproblems can be defined in terms of each item. At each subproblem, a choice has to be made - is it better to take this item or not take this item? In these problems, a constant number of options exist that can be analyzed and compared in constant time to find the solution. Thus the work done in this subproblem is $O(1)$
- **Linear time comparisson** This will make a choice between several previous solutions to choose the optimal solution to this subproblem. Problems defined this way will have to check a variable amount of subproblems to make a decision. *Example:* In Chain Matrix Multiplication, a subproblem from index i to j checks all possible ways to group these matrices before returning the minimal solution. These problems may not even make recursive calls to subproblems - The Longest Increasing Subsequence problem doesn't make a call for any subproblems if the ith element is smaller than all previous elements. The work done in these subproblems can, in the worst case, analyze many previous solutions before this solution is known. The work done in this subproblem is $O(n)$

3. Make the recursive solution efficient. You've finished the hard part of dynamic programming. Now you just have to make the recurssion efficient. We've discussed two ways to do this: Iteration and Memoization

- **Iteration:** Although the problems are defined recursively, in iteration, the recursive calls are removed. Make an array (potenially a two dimensional one) to hold solutions to subproblems. Start by filling in the solutions to the base cases. Then iterate through solutions until all subproblems have been solved. Each step in the iteration will access previous solutions, solve the solution for this subproblem, store it in the array, then move to the next subproblem.

  Caution: Make sure when iterating that the subproblems have actually already been computed. For several problems (especially ones involving two dimensinal arrays) the wrong iteration would access solutions that are currently unknown - Chain Matrix Multiplication is an example of this. General rule of thumb: Start iterating close to the base cases

- **Memoization:** Alternatively, the recursive structure of the problem can be kept with the program only computing a solution once. This technique also utilizes an array to store solutions, but this array is a global array with solutions initialized to *null* or a special value to indicate that a solution has not yet been computed. Inside the function, check to see if a the solution has already been found - if it hasn't, then compute it and store it. The benefit of memoization is that the solution for a subproblem can essentially be duplicated instead of replaced with loops and array accessing.

  Memoization faces many challenges in practice ranging from maximum recursion depth to the time needed to make recursive calls, but it's highly efficient on problems where not all subproblems have to be computed (the Knapsack problem can run much faster with memoization).

  Caution: Be careful to make the right calls and array lookups with memoization. The left hand side will store to the global array but the right hand side will actually make a recursive call

4. Analyze the runtime. The runtime of a properly implemented dynamic program will be the number of subproblems times the work done in each subproblem.

2 Basic Probability

For a set of independent events $X_1, X_2, \ldots X_N$, the probability that *all* occur, $p(X_1 \wedge X_2 \ldots X_n) = p(X_1)p(X_2)\ldots p(X_n)$
For a set of disjoint events $X_1, X_2, \ldots X_N$, the probability that *any* occur $p(X_1 \vee X_2 \ldots X_n) = p(X_1) + p(X_2) \ldots + p(X_n)$
Conditional Probability: $A|B$ means the probability that A occurs if B occurs. $p(A|B) = p(A \cap B)/p(B)$
Total Probability: For event A, disjoint partition $B_1 \cup B_2 \ldots \cup B_n = \Omega$, $P(A) = P(A|B_1)P(B_1) + P(A|B_2)p(B_2)\ldots p(A|B_n)p(B_n)$.
Special case for an event B, $P(A) = P(A|B)P(B) + P(A|\bar{B})P(\bar{B})$

3 Linear Programming

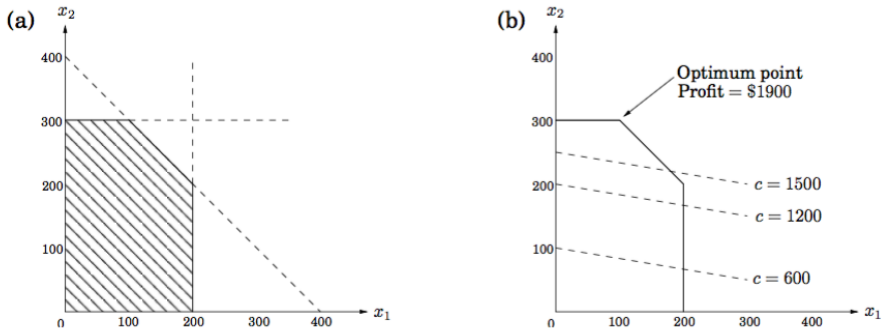
Problems will have some function that must be maximized or minimized, but the values must conform to certain constraints. 2 ways to solve this: Graph and test, matrix version

Graph and Test: Graph the line created by the maximization function, and the lines created by the constraints to get a "feasible area" then test the vertices of the feasible area and find the max result. This works quite well for 2D problems (Only 2 variables) but can get complex for more dimensions.

Matrix: Create an objective matrix C with your maximization function's constants, and matrices A and B with A containing the constants from the constraints (each equation is a seperate row), and B containing the max value from the constraints. If all constraints use the same variables, no filler variables are needed, but if they don't, create filler variables who's constants are 0's for the constraints that don't use them.Once this is done, setup the matrices in the following form:

$max\ C^T x$
$Ax = B$
where x is the matrix of the variables, with a row per variable. Then maximize this function by using the Simplex method.

Examples:



Objective function	$\max\ x_1 + 6x_2$
Constraints	$x_1 \le 200$
	$x_2 \le 300$
	$x_1 + x_2 \le 400$
	$x_1, x_2 \ge 0$

$$\begin{array}{lrcrcrcl} \text{maximize} & 3x_1 & + & 4x_2 & - & 2x_3 & & \\ \text{subject to} & x_1 & + & 0.5x_2 & - & 5x_3 & \le & 2 \\ & 2x_1 & - & x_2 & + & 3x_3 & \le & 3 \\ & & & x_1, & x_2, & x_3 & \ge & 0. \end{array}$$

:'s for slack variables):

$$\begin{array}{rcrcrcrcl} x_1 & + & 0.5x_2 & - & 5x_3 & + & x_4 & = & 2 \\ 2x_1 & - & x_2 & + & 3x_3 & & + x_5 & = & 3. \end{array}$$

:o matrix notation:

$$\begin{bmatrix} 1 & 0.5 & -5 & 1 & 0 \\ 2 & -1 & 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}.$$

$$\begin{bmatrix} 3 \\ 4 \\ -2 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}.$$

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax = b \\ & x \ge 0. \end{array}$$

Simplex Method: idea that the maximization of an LP will always be on the vertices of the feasible region, so test those, and move towards higher vertices to find a solution.

Finding the Dual of an LP: The dual of an LP can be used to flip the problem and confirm that your answer is correct. It can be calculated using the following 7 steps:

$$\max_{x_1\geq 0,\, x_2\leq 0,\, x_3} \quad v_1x_1 + v_2x_2 + v_3x_3 \tag{1}$$
$$\text{such that} \quad a_1x_1 + x_2 + x_3 \leq b_1 \tag{2}$$
$$x_1 + a_2x_2 = b_2 \tag{3}$$
$$a_3x_3 \geq b_3 \tag{4}$$

Step 1. *If necessary, rewrite the objective as a minimization.*

In our case the objective (1) is a maximization, so we rewrite it as

$$\min_{x_1\geq 0,\, x_2\leq 0,\, x_3} -v_1x_1 - v_2x_2 - v_3x_3$$

Step 2. *Rewrite each inequality constraint as a "less than or equal", and rearrange each constraint so that the right-hand side is 0.*

After this step our linear program now looks as follows.

$$\min_{x_1\geq 0,\, x_2\leq 0,\, x_3} \quad -v_1x_1 - v_2x_2 - v_3x_3$$
$$\text{s.t.} \quad a_1x_1 + x_2 + x_3 - b_1 \leq 0 \tag{5}$$
$$x_1 + a_2x_2 - b_2 = 0 \tag{6}$$
$$-a_3x_3 + b_3 \leq 0 \tag{7}$$

Step 3. *Define a non-negative dual variable for each inequality constraint, and an unrestricted dual variable for each equality constraint.*

To constraints (5) and (7) we associate variables $\lambda_1 \geq 0$ and $\lambda_3 \geq 0$ respectively. To constraint (6) we associate λ_2, which is unrestricted.

Step 4. *For each constraint, eliminate the constraint and add the term* (dual variable)*(left-hand side of constraint) *to the objective. Maximize the result over the dual variables.*

If we do this for each constraint (except of course special constraints), and maximize the result over the dual variables, we get

$$\max_{\lambda_1\geq 0,\lambda_2,\lambda_3\geq 0}\ \min_{x_1\geq 0,\, x_2\leq 0,\, x_3} \quad -v_1x_1 - v_2x_2 - v_3x_3$$
$$+ \quad \lambda_1\left(a_1x_1 + x_2 + x_3 - b_1\right) \tag{8}$$
$$+ \quad \lambda_2\left(x_1 + a_2x_2 - b_2\right) \tag{9}$$
$$+ \quad \lambda_3\left(-a_3x_3 + b_3\right) \tag{10}$$

Step 5. *We now have an objective with several terms of the form* (dual variable)*(expression with primal variables), *plus remaining terms involving only* primal variables. *Rewrite the objective so that it consists of several terms of the form* (primal variable)*(expression with dual variables), *plus remainig terms involving only* dual variables.

If we do this to the objective of the previous step, we obtain

$$\max_{\lambda_1\geq 0,\lambda_2,\lambda_3\geq 0}\ \min_{x_1\geq 0,\, x_2\leq 0,\, x_3} \quad -b_1\lambda_1 - b_2\lambda_2 - b_3\lambda_3$$
$$+ \quad x_1(a_1\lambda_1 + \lambda_2 - v_1) \tag{11}$$
$$+ \quad x_2(\lambda_1 + a_2\lambda_2 - v_2) \tag{12}$$
$$+ \quad x_3(\lambda_1 - a_3\lambda_3 - v_3) \tag{13}$$

Step 6. *Remove each term of the form* (primal variable)*(expression with dual variables) *and replace with a cnostraint of the form:*

- *expression* ≥ 0, *if the primal variable is non-negative.*
- *expression* ≤ 0, *if the primal variable is non-positive.*
- *expression* $= 0$, *if the primal variable is unrestricted.*

$$\max_{\lambda_1\geq 0,\lambda_2,\lambda_3\geq 0} \quad -b_1\lambda_1 - b_2\lambda_2 - b_3\lambda_3$$
$$a_1\lambda_1 + \lambda_2 - v_1 \geq 0 \tag{14}$$
$$\lambda_1 + a_2\lambda_2 - v_2 \leq 0 \tag{15}$$
$$\lambda_1 - a_3\lambda_3 - v_3 = 0 \tag{16}$$

Step 7. *If the linear program in step 1 was rewritten as a minimization, rewrite the result of the previous step as a minimization; otherwise, do nothing.*

The result looks as follows. Optionally, the constraints can be also be rearranged in whichever form is most natural.

$$\min_{\lambda_1\geq 0,\lambda_2,\lambda_3\geq 0} \quad b_1\lambda_1 + b_2\lambda_2 + b_3\lambda_3$$
$$a_1\lambda_1 + \lambda_2 \geq v_1 \tag{17}$$
$$\lambda_1 + a_2\lambda_2 \leq v_2 \tag{18}$$
$$\lambda_1 - a_3\lambda_3 = v_3 \tag{19}$$

Once the dual has been found, you can solve it in the same way you'd solve an LP, then use the values determined for the dual variables to check if the maximization value found for the original problem is the same.

Dynamic Programming Homework Solutions

Give an **iterative** dynamic programming algorithm that, given the locations of $m-1$ cuts in a piece of pipe of length n, finds the minimum **cost** of breaking it into m pieces. For convenience, you may want to assume that the input array is of the form $[c_0, c_1, \ldots, c_{m-1}, c_m]$, where $c_0 = 0$ and $c_m = n$. Think carefully about how to define your subproblems, and your final runtime shouldn't be much longer if all lengths were multiplied by 1000.

Solution:

1. The general setup is similar to that of chain-matrix multiplication. Lets say the locations of the m cuts were given to you in an array l.

   Let $CutCost(i,j)$ be the minimum cost of making the $j-i-1$ remaining cuts on the peice of pipe between cuts i and j, if you were given that as one solid piece with the ends already cut off. For notational convenience, if treat positions $l[0] = 0$ and $l[m] = n$ like cuts in our array, then the value we are looking for is $CutCost(0,m)$.

2. Given a segment between cuts i and j, we choose which cut to make first, and we incur a total cost of $l[j] - l[i]$ no matter where we make the cut. What we will be left with is two peices of pipe that we may then have to cut further.

   We might have some intuition about where this first cut should probably be (near the middle, in some sparse area, etc.), but we must ignore that intuition, because the night is dark and full of counterexamples. So to avoid potentially misleading intuition, we instead check all possible choices of the next cut, and choose the best one.

   For any choice of cut x, where $i < x < j$, we have two pieces left, one from i to x, and another from x to j. To pick out the best one, then $CutCost(i,j) = (l[j]-l[i]) + \min_{i<x<j}(CutCost(i,x) + CutCost(x,j))$. Note that to solve $CutCost(i,j)$, we need to know values of $CutCost$ solely on subproblems with a smaller difference between the indices. Thus, in our for loop, we will solve all subproblems of a particular distance between indices before increasing. Also, the base cases can be $CutCost[i,i+1] = 0$, since that piece of pipe no longer needs to be cut.

3. **function** CutCost(Array $l[0\ldots m+1]$)
   Array $CC[0\ldots m+1 \times 0\ldots m+1]$
   for $i = 0$ to m **do**
   $CC[i,i+1] \leftarrow 0$
   ▷ Initialization of values
   for $l = 2$ to $m+1$ **do** ▷ The increasing distance between i and j
   for $i = 0$ to $m+1-l$ **do**
   $CC[i,i+l] \leftarrow l[j] - l[i] + \min_{i<x<j}(CC[i,x] + CC[x,j])$
   return $CC[0,m]$

   We could have also done the iteration in a different way (see the chain matrix multiplication example).

4. We have $O(n^2)$ entries in our array and do $O(n)$ work to solve each entry, resulting an in $O(n^3)$ algorithm.

6. (group) Given two sequences X and Y, let $C(X,Y)$ denote the number of times that X appears as subsequence of Y. For instance, the sequence AB appears 5 times as a subsequence of $ADABACB$. Write an efficient dynamic programming algorithm to calculate $C(X,Y)$. (Hint: Consider the solutions to prefixes of the input as your subproblem.)

Solution:

- Let $C(i,j)$ be the number of occurences of the first i characters of X as a subsequence in the first j characters of Y.

  Let X_i denote the first i characters of the string X and let $X[i]$ denote the i^{th} character (similarly for Y). Let m denote the length of X and let n denote the length of Y.

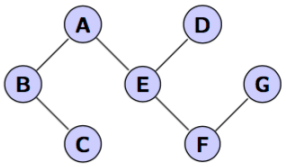
- If the last characters of X_i and Y_j don't match, then the last character of Y_j is not involved in any occurence of X_i, so the count is the same as the number of occurences of X_i in Y_{j-1}. If the last characters do match, then we also include the number of occurences of X_i that involve matching $X[i]$ with $Y[j]$, namely $C(i-1,j-1)$, plus the number that don't, $C(i,j-1)$. So the recurrence is: $C(i,j) = C(i,j-1)$ if $X[i] \neq Y[j]$, and $C(i,j) = C(i,j-1) + C(i-1,j-1)$ if $X[i] = Y[j]$.

  Now we set up the base cases so that your recurrence is correct for small values. $C(X_0, Y_j) = 1$, and $C(X_i, Y_0) = 0$ if $i \geq 1$.

- **function** NumSubsequence(X, Y)
  Array $C[0\ldots m, 0\ldots n]$
  for $i = 1$ to m **do**
  $C[X_i, Y_0] = 0$
  for $j = 0$ to n **do**
  $C[X_0, Y_j] = 0$
  for $i = 2$ to m **do**
  for $j = 2$ to n **do**
  if $X_i = Y_j$ **then**
  $C[X_i, Y_j] = C[X_i, Y_{j-1}] + C[X_{i-1}, Y_{j-1}]$
  else
  $C[X_i, Y_j] = C[X_i, Y_{j-1}]$
  return $C[X_m, Y_n]$

- The running time is $O(mn)$. We have $O(mn)$ entries in our table and it takes $O(1)$ time to compute each entry.

vertex 1



Solution: Assume that the tree is rooted at some node r, otherwise pick one at random to be the root.

There are multiple possible solutions for this problem (at least 4)!

1. Let $VCover(n)$ be minimum cost of a vertex cover of the subtree descending from node n. For a leaf, we set $VCover(n) = 0$.
2. At any node n, we have the choice of including the vertex n, or not including the vertex n. If we include n, we cover all the edges between n and its children. The remaining subtrees descending from n's children still need to be covered, in which case $VCover(n) = 1 + \sum_{x \in n.children} VCover(x)$.

   If we do not include n, then we MUST include every child of n in the vertex cover. This leaves all the subtrees descending from n's gradchildren uncovered. In this case, $VCover(n) = |n.children| + \sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$. We choose the smaller of the two.
3. This is a memoized-ish approach, since we look at children and grandchilden, this is easier. If we used a solution that only every looked at children, then we could have done this in a more tree-search style implementation.

   Global Tree T
   function INIT(node n) ▷ Initializing takes $O(|V|)$ time $n.cover \leftarrow NULL$
   for $c \in n.children$ **do**
   $Init(c)$
   function VCOVER(node n)
   if $n.cover = NULL$ **then** $n.cover \leftarrow \min(1+\sum_{x \in n.children} VCover(x), |n.children|+ \sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$
   return n.cover
   return $Init(T.root)$.
   return $VCOVER(T.root)$.
4. Outside the summations, we do $O(1)$ work per node.

   Inside the summations, at each node n, we do an amount of work proportional to the number of all n's children and all n's grandchildren. This could be large for some nodes and small for others. To count this a different way, think about it backwards.

   Any node n is only ever considered inside a summation by n's parent or n's grandparent. This means $VCover(n)$ is only ever called inside a summation $O(1)$ times per node. Thus, the total work done by all the summations over every call of $VCover$ is $O(|V|)$.

vertex 2

This gives us a total running time of $O(|V|)$. (Recall that in a tree, $|V| = |E| + 1$).

There is an alternate solution that keeps track of two values of $VCOVER$ per vertex, one where you do include the vertex and one where you don't.

There is a third greedy solutions that looks at leaves and works it's way up.

There is a fourth solution that makes use of a connection between independent sets and vertex covers.

You may notice that there are some strong similarities between the independent set algorithm given in class and the vertex cover algorithm here. There's a reason that's clear if you think about it from the perpective of the edges. In an independent set, every edge has 0 or 1 of its end points included in the set, but not 2. In a vertex cover, every edge has 1 or 2 edges included in the set, but not 0. Thus the *complement* of an independent set I is in fact a vertex cover with $n - |I|$ elements, and vice versa! Furthermore the complement of a *maximum* independent set must be a *minimum* vertex cover!

Thus the following is an acceptable solution, if accompanied by something like the above proof of correctness.

"Run the maximum independent set algorithm given in class. Return the complement set of vertices."

Solution:

1. Let $Palin(i, j)$ be the length of the longest palindromic subsequence between (and including) characters i and j. We will use the convention that $Palin(i, i-1) = 0$ (a zero-length string), and that $Palin(i, i) = 1$, since an individual character is a palindrome.

   You could have chosen a different workable solution with a different convention for the base cases.

S 3510
orgia Tech, Spring 2016

Homework 6

Instructor: Prateek Bhakta
Student: YOUR NAME HERE

2. This is a bit like longest common subsequence. Let x be our string. Then if $x[i] = x[j]$, then these endpoints can match with each other, and the length of the longest palindrome is $1 + Palin(i+1, j-1)$.

   If $x[i] \neq x[j]$, then these endpoints are not both in the longest palindrome, and we may eliminate one of them without changing the longest palindrome. We don't know for sure if either of them are used in the longest palindrome so far, we just know it can't be both. Therefore, we should choose $\max(Palin(i+1, j), Palin(i, j-1)$ to cover both cases. Like in chain matrix multiplication or the pipe cutting problem, we can compute these values in order of the increasing *length* of the sequence.
3. **function** LONGESTPALIN(Array $x[1 \ldots n]$)
   Array $Palin[1 \ldots n \times 0 \ldots n]$
   for $i = 1$ to n **do**
   $Palin[i, i] \leftarrow 1$
   $Palin[i, i-1] \leftarrow 0$
   ▷ Initialization of values
   for $l = 1$ to $n - 1$ **do** ▷ Increasing lengths
   for $i = 1$ to $n - l$ **do**
   $j \leftarrow i + l$
   if $x[i] = x[j]$ **then**
   $Palin[i, j] \leftarrow 1 + Palin[i+1, j-1]$
   else
   $Palin[i, j] \leftarrow \max(Palin[i+1, j], Palin[i, j-1])$
   return $MaxL$
4. Each entry of the array takes constant work, and we compute values for about half of the $n(n+1)$ entries, which gives us an $O(n^2)$ algorithm in total.

Solution:

- Let $Change(i, t)$ be the minimum number coins needed to make change for a value of t using the first i coins, or $Null$ if it is impossible.

  Your base cases would be 0 if $t = 0$, and *null* if $t < 0$, and also null if $i = 0$ but $t > 0$.
- We should decide if the best way to make change for t using the first i coins should use a coin with value x_i or not. If it does, then the total number of coins we need is one more than the number needed to get the value $t - x_i$. If it doesn't, then this is a same way to make t using the first $i - 1$ coins. Thus, $Change(i, t) = \min(Change(i-1, t), 1 + Change(i, t - x_i))$.
- **function** CHANGE(A,v)
  Array $S[n \times (v+1)]$
  for $i = 0$ to n **do**
  $S[i, 0] \leftarrow 0$
  for $t = 1$ to v **do**
  $S[0, t] \leftarrow False$ ▷ Base Case Initialization
  for $i = 1$ to n **do**
  for $t = 0$ to v **do**
  $S[i, t] \leftarrow S[i-1, t]$
  if $x_i \leq t$ **then**
  $S[i, x] \leftarrow \min(S[i, x], S[i-1, t - x_t] + 1)$
  return $S[n, v]$
- We have $n \times v$ entries in our array, each of which is computed in $O(1)$ time, which leads to an $O(nv)$ algorithm.