CS 3510: Design & Analysis of Algorithms (Section B)

Exam 3 (Dynamic Programming) Study Guide

Professor Abrahim Ladha

1 Recommended Problems:

Here are some recommended problems to do in the textbook! Link To DPV

Questions: 6.3, 6.4, 6.7, 6.20, and 6.26

Additionally, we have some practice problems on Project Euler, which is a website containing challenging programming problems:

Problem 67, Problem 31

Note: These problems are difficult, perhaps more difficult than the exam. They will provide good practice for understanding and designing dynamic programming algorithms. See the pinned piazza post for plenty of practice questions.

2 Topic List:

2.1 Dynamic Programming Concepts

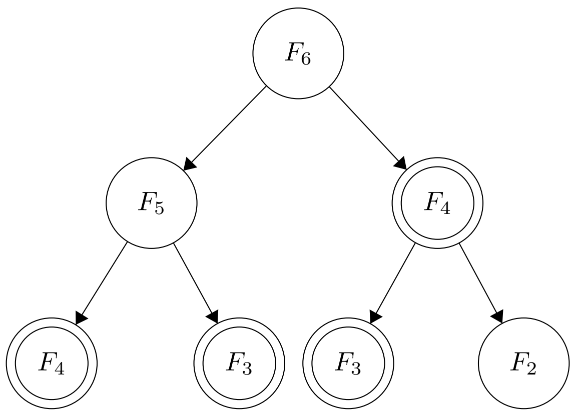
Before we review the algorithms you need to know for the exam, let's first review the concept of Dynamic Programming (DP). DP is a method to optimally solve problems that have a sub-problem structure, since we store results of smaller sub-problems to build up to our solution. We do this creating an array (known as *tabulation*) and populate values corresponding to larger and larger sub-problems. This is also called a **bottom-up** DP approach.

Difference between DP and Divide-and-Conquer
This is common question as both algorithms approach problems similarly: by breaking up the original problem into smaller parts. The key difference between DP and D-and-C is that DP breaks problems up into *overlapping* subproblems, while D-and-C aims to break up problems up into *non-overlapping* subproblems. Think of MergeSort, for example: the subproblems are the left and right half of the array, which do not overlap. If we look at the Fibonacci example below, however, we can see that we find many duplicate subproblems with standard recursion. Dynamic programming solves that issue.

Let's look at simple example of DP: computing the n^{th} Fibonacci number. We have a standard recursive solution to this problem:

```
Fibonacci (n):
    if $n$ == 0 or $n$ == 1:
        return 1
    return Fibonacci($n-1$) + Fibonacci($n-2$)
```

However, expanding the recursive tree shows us that we're computing the same subproblem several times:



Instead of allowing this, we can instead store results of previous Fibonacci numbers in a table as follows:

0	1	2	3	...	n
1	1	2	3	F_i	F_n

At each index i in array T, $T[i]$ represents the i^{th} Fibonacci number. Our base cases are $T[0] = T[1] = 1$, just as in the recursive implementation. However, now we compute $T[i] = T[i-1] + T[i-2]$ which is just an $\mathcal{O}(1)$ operation since we've already precomputed the previous Fibonacci numbers. Therefore computing the n^{th} Fibonacci number takes just $\mathcal{O}(n)$ time in total, which is a huge improvement over the naive recursion algorithm, which runs in exponential time.

2.2 DP Algorithm Requirements

On the exam, we'll be asking you to design and analyze of the efficiency of a dynamic programming algorithm to solve problems, just as on the homeworks. You will see the below steps as individual parts to a question, but just for practice and clarification, remember to do each of these steps when designing DP algorithms:

1. **Define your table and what each entry represents**. For example, what does $T[i]$ represent in the context of the problem, or what does $T[i, j]$ represent?

2. **Define the base cases**. Remember to include all relevant base cases, whether that be a single value or rows of a matrix.

3. **State the DP recurrence and explain why it's correct**. This step should not involve pseudocode - just a mathematical expression that shows how we update the values in the DP table based on previous entries, and a paragraph or two that explains why this correctly computes the result of the current sub-problem.

4. **Explain how we get the return value for the original problem**. This is not always the final cell in the DP table, though, so make sure this is correct.

5. **Analyze the running time of the algorithm**. For this question, you want to focus on the size of the table and the cost of the operation to populate each entry in the table. We multiply those together to get the final efficiency of the algorithm.

2.3 Introductory Problems

Number of Ways: Suppose you have n stair steps. You can leap one, two, or three steps at a time. What is the number of combinations, or the number of ways, that you could reach step n?

Solution: Create an array $T[0 \dots n]$ where $T[i]$ represents the number of ways to reach step i.
Our base cases are $T[0] = 1, T[1] = 1$ and $T[2] = 2$. To reach step n, there was some last step. You either jumped 1, 2, or 3 steps, and you jumped from 1, 2, or 3 steps away. That means that the number of ways to go from 0 to n is the sum of the number of ways to go from 0 to $n-1$, 0 to $n-2$, and 0 to $n-3$. This gives us our recurrence of

$$T[i] = T[i-1] + T[i-2] + T[i-3]$$

Implementation: (Pseudocode)

```
def numways(k):
    initialize dp as table of size n + 1 with 0s
    dp[0] = 0
    dp[1] = 1
    dp[2] = 1

    for i in 3..(n+1):
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]

    return dp[k]
```

Figure 1: Numways algorithm.

The results are as follows:

indices	0	1	2	3	4	5	6	7	8	9
results	1	1	2	4	7	13	24	44	81	149

Runtime: The runtime of a dynamic progrmaming algorithm is the size of the table multiplied by the work done to calculate each cell. In this case that is $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$

Min Operations: Suppose you had two operations:

- add one
- multiply by two

How many operations does it take to get from 0 to some k.
The results are as follows.

indices	0	1	2	3	4	5	6	7	8	9
results	0	1	2	3	3	4	4	5	4	5

It's not monotonic, and it's actually a little hard. Note that for 8 it's a power of two so it takes fewer operations than either of its neighbors. Trivially you could find this in exponential time by trying all possibilities. Let's write our recurrence.
$dp[0] = 0$ because for 0, you need zero operations. $dp[1] = 1$ because for 1, you add one to zero. Now we'll try to multiply by two as much as possible so we'll add 1 if the element is odd and multiply by two if it is even. We get the following overall recurrence.

$$d(i) = \left\{ \begin{array}{ll} d(i-1)+1, & \text{if } i \text{ is odd} \\ d(i/2)+1, & \text{if } i \text{ is even} \end{array} \right\} \tag{1}$$

This might require a proof, but it's simple enough you can likely see why it is what it is. Let's actually write out the algorithm.

```
def minop(k):
    initialize dp as table of size k + 1 with 0s
    dp[0] = 0
    dp[1] = 1

    for i in 2..(k + 1)
        dp[i] = dp[i - 1] + 1
        if i is even
            dp[i] = min(dp[i], dp[i / 2] + 1)

    return dp[k]
```

Figure 2: Minop algorithm.

Minimum is a fairly common tool in dynamic programming problems. Usually you choose between two possible recurrences, and the minimum results in your answer for that element in the table. You could try some greedy approach, but it gets hard to prove if that is optimal.

House Robber: Given an array houses with values $H = [h_1 \dots h_n]$, you want to rob them, but you can't rob two adjacent houses or alarms will go off. What is the maximum amount you can steal? Here's an example: $[2, 7, 9, 3, 1]$ would yield $2 + 9 + 1 = 12$.
Solution: Let's define our array $T[0 \dots n]$ with $T[i]$ =maximum we can steal from houses $0 \dots i$. The base cases are that $T[0] = H[0], T[1] = \max(H[0], H[1])$.
Now let's develop the recurrence. At the next house visited, we can choose to rob or not rob a house. We take the maximum of both scenarios. If we rob the house, than we could not have robbed the previous house, and if we didn't rob the house, then we could have robbed the previous house. This gives us the recurrence

$$T[i] = \max(T[i-2] + H[i], T[i-1])$$

```
def houserobber(H):
    initialize dp as table of size n with 0s
    dp[0] = H[0]
    dp[1] = max(H[0], H[1])

    for i in 2..n
        dp[i] = max(dp[i - 2] + H[i], dp[i - 1])
    return dp[n - 1]
```

Figure 3: House Robber algorithm

For example, given $H = [2, 7, 9, 3, 1]$, $T = [2, 7, 11, 10, 12]$. The runtime is $\mathcal{O}(n)$ since we have a table of size n with $\mathcal{O}(1)$ work at each step.

Grid Paths: Given n, m, what is the number of paths through an $n \times m$ grid from (1, 1) to (n, m) if you can only go down and right? For example, given a 3×3 matrix, there are 6 paths. This can be solved without dynamic programming, just some combinatorics, but let's do it with DP.

Solution: Let's define array $T[1 \dots n][1 \dots m]$ where $T[i][j]$ =the number of paths from (1, 1) to (i, j). The base cases are that $T[1 \dots n][1]$ and $T[1][1 \dots n]$ equal 1. This is essentially the first row and the first column of the matrix.
Now working on the recurrence, for non-base case cells,
the number of ways to reach (i, j) is the sum of the number of ways to reach the cell above it and the cell to its left. This is because you can only come to (i, j) from $(i - 1, j)$ or $(i, j - 1)$.
So, the recurrence relation is:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

Finally, $dp[n][m]$ will give the number of paths from $(1, 1)$ to (n, m).

```
def count_paths(n, m):
   initialize dp as a table of size n*m with 0s

    for i in 0...n:
        dp[i][0] = 1
    for j in 0...m:
        dp[0][j] = 1

    for i in 1...n:
        for j in range(1, m):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[n-1][m-1]
```

Figure 4: House Robber algorithm

The Time and Space complexity is $O(n * m)$

Grid Paths II: Similar question as example 4, Find the number of paths to reach cell (n, m). Additionally, the grid has bombs, denoted by an input *bombs* such that $bombs[i][j] = True$ denoted cell (i, j) has a bomb. Find number of paths from $(0, 0)$ to (n, m) such that no bombs are traversed.
Unlike example 4, this problem cannot be solved with combinatorics and requires dynamic programming.
Let's define array $T[1 \ldots n][1 \ldots m]$ where $T[i][j]$ =the number of paths from (1, 1) to (i, j).
Base Cases:

1. If the starting cell $(1, 1)$ has a bomb, then there are 0 paths.

$$dp[1][1] = \begin{cases} 0 & \text{if } bombs[1][1] = \text{True} \\ 1 & \text{otherwise} \end{cases}$$

2. For the first row and first column:

$$dp[i][1] = \begin{cases} 0 & \text{if } bombs[i][1] = \text{True} \\ dp[i-1][1] & \text{otherwise} \end{cases}$$

$$dp[1][j] = \begin{cases} 0 & \text{if } bombs[1][j] = \text{True} \\ dp[1][j-1] & \text{otherwise} \end{cases}$$

Recurrence Relation:
For all cells not on the first row or first column:

$$dp[i][j] = \begin{cases} 0 & \text{if } bombs[i][j] = \text{True} \\ dp[i-1][j] + dp[i][j-1] & \text{otherwise} \end{cases}$$

This means that if a cell has a bomb, no paths can go through it. Otherwise, the number of paths to a cell is the sum of the paths to the cell above it and the cell to its left.
The solution will be in $dp[n][m]$, which gives the number of paths from $(1, 1)$ to (n, m) without traversing any bombs.

```
def count_paths_with_bombs(n, m, bombs):
   initialize dp as a table of size n*m with 0s

    if bombs[0][0]:
        return 0

    dp[0][0] = 1

    for i in 1...n:
        if not bombs[i][0]:
            dp[i][0] = dp[i-1][0]

    for j in 1...m:
        if not bombs[0][j]:
            dp[0][j] = dp[0][j-1]

    for i in 1...n:
        for j in range(1, m):
            if not bombs[i][j]:
                dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[n-1][m-1]
```

Figure 5: House Robber algorithm

2.4 Longest Increasing Subsequence

Let's move on to Longest Increasing Subsequence (LIS). Here's the problem statement:

Given an array $S = [s_1, s_2, \ldots, s_n]$, find the length of the longest increasing subsequence of S.

A **subsequence** of an array (or string) is a subgroup of elements in the array such that they are positioned in order relative to their original order in the array. Mathematically, a subsequence of S can be written as

$$[S_{i_1}, S_{i_2}, \ldots, S_{i_k}]$$

such that indices $i_1 < i_2 < \cdots < i_k$.

A **substring** is a subsequence where all elements must be contiguous in the original array.

Let's define a DP algorithm for this problem.

1. Consider entry $T[i]$ to represent the length of the LIS of the first i elements that ends at S_i. We will populate the entries in this table from 1 to n.
2. Our base case is $T[0] = 0$ since a subsequence does not exist. We can include $T[1] = 1$ as a simple case since a single element is its own increasing subsequence.
3. Our recurrence is as follows:

$$T[i] = \max_{j<i}\{1 + T[j] \;\text{ if }\; S[i] > S[j]\}$$

   We compute $T[i]$ by looking at all entries of the table prior to index i which are smaller than the value of $T[i]$. This is because we can only append S_i to an increasing subsequence if the last element of the subsequence is smaller than S_i. To find the largest length of the LIS to which we can add S_i, we take the maximum of the valid entries we iterate over and add 1 to the maximum.
4. To return the result of the original problem, we take the maximum of all the entries in T. This is because we do not know which element of S is the final element in the LIS.
5. Let's analyze the runtime of the algorithm. We have a table of n elements, and computing each entry has an upper-bound running time of $\mathcal{O}(n)$, since we traverse through previous indices of the array.

Computing the final result takes $\mathcal{O}(n)$ time since we find the maximum of the array. Therefore the total running time of the algorithm is $\mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$.

An extension of the LIS problem is the following: what if we want to output the LIS (or one of the LIS's), rather than just the length of the LIS?

We can actually backtrack from the maximum entry in T one step at a time such that each backtrack entry $T[j]$ is one less than the previous. For example, if $T[10] = 6$ is the length of the LIS, then we find an index $j < 10$ with $T[j] = 5$, index $k < j$ with $T[k] = 4$, and so on. At each backtrack index i, we add the element S_i to the LIS. If there are multiple backtrack options, we can choose any one of them, as we'll still end up with a valid LIS.

2.5 Longest Common Subsequence

Next up is Longest Common Subsequence, or LCS. The problem statement is as follows:

Given string $X[1...n]$ and $Y[1...m]$, find the length of their longest common subsequence. For example, given

$$X = AABCDDE$$

$$Y = ABDDECEA$$

the LCS would be $ABDDE$, so we would return 5.
Often times, DP with strings as input involves substructures which compute the solution for substrings of the input. Let's look at an example to see if we can find a relation between a smaller and larger sub-problem:

Case 1: Given $LCS(AABCD, ABD) = |ABD| = 3$, can we find $LCS(AABCDD, ABDD)$?

Notice that we added two new characters to both strings which happen to be equal. In this case, it must be that both characters can be added to the LCS of X and Y, so we just add 1 to the previous subproblem. If the characters are not equal however, then we can't necessarily add the character to the LCS. Let's look at this:

Case 2: Given $LCS(AABCDD, ABDE) = 3$, can we find $LCS(AABCDDE, ABDED)$?

Here, the first subproblem is not going to help us arrive at the expected answer of 4 (ABDD), since the new characters are not equal. Therefore we'll need some way to keep track of the largest LCS found so far. This means that **we cannot add one character to both strings at a time** since we'll not have enough information to compute the new subproblem. So we can't actually solve this problem with a one-dimensional DP table; we'll need two dimensions, since we add one character from a string at a time.

Now let's examine the algorithm:

1. Consider entry $T[i, j]$ to represent the length of the LCS of substring $X[: i]$ and substring $Y[: j]$ $(0 \leq i \leq n,\ 0 \leq j \leq m)$
2. Our base cases are the first row and column of the table. Since this corresponds to $i = 0$ or $j = 0$, the LCS of $X[: i]$ and $Y[: j]$ will be 0. Using notation, we have $T[0, j]\forall j = T[i, 0]\forall i = 0$.
3. To define our recurrence, we have two cases:

   Case 1: $X[i] = Y[j]$. Since the new characters are equal, we can always append them to the LCS of the substrings not including both characters. Therefore $T[i, j] = T[i - 1, j - 1] + 1$.

   Case 2: $X[i] \neq Y[j]$. Since the new characters are not equal, we take the the length of the LCS found so far. This could be either of two subproblems we've computed: (1) $LCS(X[: i - 1], Y[: j])$ and (2) $LCS(X[: i], Y[: j - 1])$. Therefore $T[i, j] = \max(T[i - 1, j], T[i, j - 1])$.

   Our final recurrence is as follows:

$$T[i, j] = \begin{Bmatrix} \max\{T[i-1, j], T[i, j-1]\} & \text{if} \quad X[i] \neq Y[j] \\ 1 + T[i-1, j-1] & \text{if} \quad X[i] = Y[j] \end{Bmatrix}$$

4. We return $T[n, m]$ as the solution to the original problem by definition of $T[i, j]$.
5. Let's analyze the runtime of the algorithm. We have a table of $n \times m$ elements, and computing each entry takes just $\mathcal{O}(1)$ time. Returning the solution also takes $\mathcal{O}(1)$ time. Therefore the total running time of the algorithm is $\mathcal{O}(nm)$.

Let's visualize the DP table for the example above. $X = AABCDDE$ has length $n = 7$, $Y = ABDDECEA$ has length $n = 8$. Initializing our table and base cases:

	0	$1(A)$	$2(A)$	$3(B)$	$4(C)$	$5(D)$	$6(D)$	$7(E)$
0	0	0	0	0	0	0	0	0
1(A)	0	*	*	*	*	*	*	*
2(B)	0	*	*	*	*	*	*	*
3(D)	0	*	*	*	*	*	*	*
4(D)	0	*	*	*	*	*	*	*
5(E)	0	*	*	*	*	*	*	*
6(C)	0	*	*	*	*	*	*	*
7(E)	0	*	*	*	*	*	*	*
8(A)	0	*	*	*	*	*	*	*

Now, let's look at the filled in table according to the recurrence. Try doing this yourself if you're confused on how the recurrence allows us to fill in the table.

Note: the cells marked in red are the cells where the characters are equal, so we increment $T[i-1, j-1]$.

	0	$1(A)$	$2(A)$	$3(B)$	$4(C)$	$5(D)$	$6(D)$	$7(E)$
0	0	0	0	0	0	0	0	0
1(A)	0	1	1	1	1	1	1	1
2(B)	0	1	1	2	2	2	2	2
3(D)	0	1	1	2	2	3	3	3
4(D)	0	1	1	2	3	3	4	4
5(E)	0	1	1	2	3	3	4	5
6(C)	0	1	1	2	3	3	4	5
7(E)	0	1	1	2	3	3	4	5
8(A)	0	1	2	2	3	3	4	5

What if we want to output the LCS? We can use a backtracking strategy based on the recurrence. Moving backwards from $T[n, m]$, we find a cell (i, j) where the characters $X[i]$ and $Y[j]$ are equal. We then record the character and move diagonally backwards to $T[i-1, j-1]$ and continue. We can visualize the path below in blue, with the recorded characters shown in orange:

	0	$1(A)$	$2(A)$	$3(B)$	$4(C)$	$5(D)$	$6(D)$	$7(E)$
0	0	0	0	0	0	0	0	0
1(A)	0	1	1	1	1	1	1	1
2(B)	0	1	1	2	2	2	2	2
3(D)	0	1	1	2	2	3	3	3
4(D)	0	1	1	2	3	3	4	4
5(E)	0	1	1	2	3	3	4	5
6(C)	0	1	1	2	3	3	4	5
7(E)	0	1	1	2	3	3	4	5
8(A)	0	1	2	2	3	3	4	5

So we get the final LCS as $ABDDE$.

Note: In lecture, the variants Longest Common Substring and Longest Palindromic Substring were also covered. Please ensure you understand how those problems work as well - the lecture notes is the best resource for this.

2.6 Chain Matrix Multiplication

Let's move on to the chain matrix multiplication problem.

Given a chain of matrices $A_1 \dots A_k$ such that all matrices can be multiplied together, determine the minimum cost of multiplying all k matrices.

Let's define the cost of multiplying two matrices.
Suppose matrix A has dimension $a \times b$, while matrix B has dimension $b \times c$. AB while have dimension $a \times c$. To compute this matrix, we take all c columns of length b in matrix B and compute dot products with a rows of length b in matrix A. Since the dot product is $\mathcal{O}(b)$, the cost of multiplying these two matrices is $\mathcal{O}(abc)$.

We essentially want to find the optimal grouping of individual matrix products so that the overall cost of the product is minimized. For example, if we want to compute the product of four matrices $ABCD$, we have multiple possible groupings: $A(B(CD)), A((BC)D), ((AB)C)D, (AB)(CD), (A(BC))D$. We can already begin to see the overlapping subproblems here; notice that in the groupings $((AB)C)D$ and $(A(BC))D$, we're just multiplying two different groups of ABC with D. We can use this as an idea to develop a DP algorithm!

Our input for this problem are the dimensions of all matrices $A_1 \dots A_k$. **We represent this by having a list of integers $m_0, m_1, m_2, m_3 \dots m_k$ that correspond to the matrix dimensions**. A_1 is of size $m_0 \times m_1$, A_2 is of size $m_1 \times m_2$, and A_k is of size $m_{k-1} \times m_k$.

Let's define our DP algorithm:

1. Consider entry $T[i, j]$ as the minimum cost of computing the product of matrices A_i through A_j, with $i \leq j$. This means that we only fill in half (upper triangle) of a regular 2-d array, since the case where $j < i$ does not apply.

2. Our base cases are when $i = j$, since the cost of computing the product of one matrix is 0. Therefore we fill in $T[1, 1], T[2, 2], \dots T[k, k] = 0$. This corresponds to the diagonal of our 2-d array. We fill in the table by diagonal, moving upwards to smaller and smaller diagonals (see next page if this is unclear).

3. Let's define the recurrence. We will be filling in the table so that when we reach a cell $[i, j]$, we would have already computed the cost of finding all smaller products in between matrices A_i and A_j. Therefore, we need to iterate through each way to divide the product $A_i \dots A_j$ and lookup the values in the table to find the minimum cost. Essentially, we want to find the minimum cost of the product

$$(A_i \dots A_o)(A_{o+1} \dots A_j)$$

   where k ranges from i to $j - 1$. We would have already computed $T[i, o]$ and $T[o + 1, i]$ prior to this entry. We also have to add the cost of computing the product of the two groupings. Since the product of $A_i \dots A_o$ would have dimension $m_{i-1} \times m_o$ and $A_{o+1} \dots A_j$ would have dimension $m_o \times m_j$, the efficiency of the product would be $m_{i-1} m_o m_j$.

   Therefore our recurrence is as follows:

$$T[i, j] = \min_{i \leq o < j} \{T[i, o] + T[o + 1, j] + m_{i-1} m_o m_j\}$$

4. We return $T[1, k]$ as the solution to the original problem by definition of $T[i, j]$.

5. Let's analyze the runtime of the algorithm. We have a table of order n^2, and computing each entry takes just $O(n)$ time since we iterate through potential groupings and find the minimum cost. Returning the solution also takes $\mathcal{O}(1)$ time. Therefore the total running time of the algorithm is $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$.

Why do we need to fill in the table by diagonals?

	0	1	2	3	4
0	0	1	2	3	4
1	■	0	1	2	3
2	■	■	0	1	2
3	■	■	■	0	1
4	■	■	■	■	0

(numbers in cells represent order of filling in table entries)

This is definitely a little weird compared to the way we've filled in the table in LIS and LCS, which is a more natural order of computing entries. This is because of the structure of the subproblems. In LIS and LCS, we always started computing subproblems from index 1 and growing our subarray/substring incrementally. In the matrix multiplication problem, we can't do that, since a grouping can be *anywhere* in the array - it could even be two matrices at the very end of the chain that we need to compute before handling larger subproblems!

This is why we fill in the table by incrementally expanding the difference between i and j until we eventually reach k. We first calculate the efficiency of the product of all neighboring pairs of matrices, then triplets, and so on. This ensures that we find all possibilities of groupings of a larger subproblem before we attempt to compute it.

2.7 Knapsack

The last application of DP relevant to the exam is a set of problems known as **Knapsack**. Knapsack has a generalized form, but there are multiple applications of Knapsack. There are also a couple additional versions of Knapsack, but for this study guide we'll review the one covered in lecture:

Given items $1, 2, \ldots, N$ with values $v_1, v_2, \ldots v_N$ and weights $w_1, w_2, \ldots, w_N$, and a knapsack capacity K, find a subset of items $S \subseteq \{1, 2, \ldots, N\}$ such that we maximize the sum of the values and that the sum of the weights in the subset is at most K.

Essentially, we want to optimally fill our knapsack so that we maximize the value we get while ensuring we don't go over capacity.

Let's look at the algorithm to solve this:

1. T[k, i] represents the maximum value of a subset of items $\{1 \ldots i\}$ with allowed capacity of k. We have $1 \leq i \leq N$ and $1 \leq k \leq K$.

2. Our base cases are when $i = 0$ (which corresponds to no items), or when $k = 0$ (which means we cannot carry any items). Therefore we set $T[0, i] \forall k = T[k, 0] \forall i = 0$.

3. At each step, we can choose to either carry item i in our knapsack, or not carry it (which would correspond to the previous subproblem of $i - 1$, keeping capacity k the same. Therefore we have two choices, and we should find the maximum value of both. If we choose to carry item i, then the remaining capacity will be decreased by the weight of item i. Therefore our recurrence is as follows:

$$T[k, i] = \max\{T[k, i-1], T[k - w_i, i-1] + v_i\}$$

   $T[k, i-1]$ represents not carrying the item, while $T[k - w_i, i-1] + v_i$ represents carrying the item. We've already computed both $T[k, i-1]$ and $T[k - w_i, i-1]$ before reaching this larger subproblem, so this is just an $\mathcal{O}(1)$ operation!

4. We return $T[K, N]$ as the result.

5. Looking at the runtime analysis, we have a table of size KN and a entry computation time of $\mathcal{O}(1)$. Therefore our overall efficiency is $\mathcal{O}(KN)$.

An example of a Knapsack problem is the Subset Sum problem. Consider an array $S = \{S_1 \ldots S_n\}$ and a target K. Can we find a subset of the array S such that the elements in the subset sum to K?

This seems different from the generalized version of Knapsack shown earlier; here we're just outputting a Yes or a No. How does this work?

Knapsack Optimization vs. Knapsack Search
This is the main difference between the generalized version and Subset Sum. The generalized version is known as a Knapsack Optimization problem, since we're trying to maximize the value of the knapsack. However, in the Knapsack Search problem, we have a goal g with which we are trying to build a knapsack such that the sum of the values in the knapsack equals g. However, both utilize similar DP recurrences.

The other, more minor difference is that we don't have weights this time! We only work with the values with no limit K.

Let's look at the changes for the Subset Sum problem:

1. $T[k, i]$ represents whether or not we can reach target k by making a subset of elements $S[: i]$. The entry is a 1 if we can reach the target, or a 0 if we cannot.
2. Base case: when $k = 0$, we store the value 1 as a base case, since we have an empty subset of elements that can sum to 0. When $i = 0$ and $k \neq 0$, we store 0 since we can't make a nonzero target with no elements. Therefore we have
$$T[0, 0] = 1, T[k, 0] \forall k = 0, T[0, i] = 1 \forall i$$
3. Recurrence:
$$T[k, i] = \max\{T[k, i - 1], T[k - S_i, i - 1]\}$$
4. Return $T[K, n]$
5. Efficiency: $\mathcal{O}(Kn)$

Notice that in our recurrence, we don't add the value of the elements, since this is a search problem rather than optimization. Additionally, note the base case $T[0, i] = 1$, since we can form an empty subset of elements to get a target sum of 0.

Subset Sum is also very similar to another classic Knapsack problem, which is **Coin Change**. If we have a set of coin denominations $c_1 \ldots c_i$ and can pick at most one coin of each denomination, can we make change for some value K? As it turns out, the recurrence for this problem is exactly the same as Subset Sum! If we represent each element in the array to represent the values of coins, we can try to find a subset of those values that sum up to our target.

So that's the content you need to know! To best prepare for the exam, review the lectures and ensure you have a good understanding of all the algorithms we've covered in class. To get better at coming up with DP algorithms, the best method is practice! Go through the practice textbook and Project Euler exercises. Feel free to attempt DP textbook problems we haven't listed in the recommended problems section; other problems will definitely be more involved and/or harder, but will really help you get used to coming up with DP algorithms. Please ask any questions you have on Piazza or in TA office hours, and we'd be glad to help. Best of luck on the exam :-)!