BOARDS

SCHOOL ORGANIZED

TABLE OF CONTENTS

Part

BEING A STUDENT IS HARD

KEEPING TRACK

FINDING HELP

INITIAL PAPER PROTOTYPE

OR HOW TO FAIL MISERABLY

M I S S I N G

UI UX

PROMPTS

FUNCTIONALITY

CONFIRMATIONS

A L O T

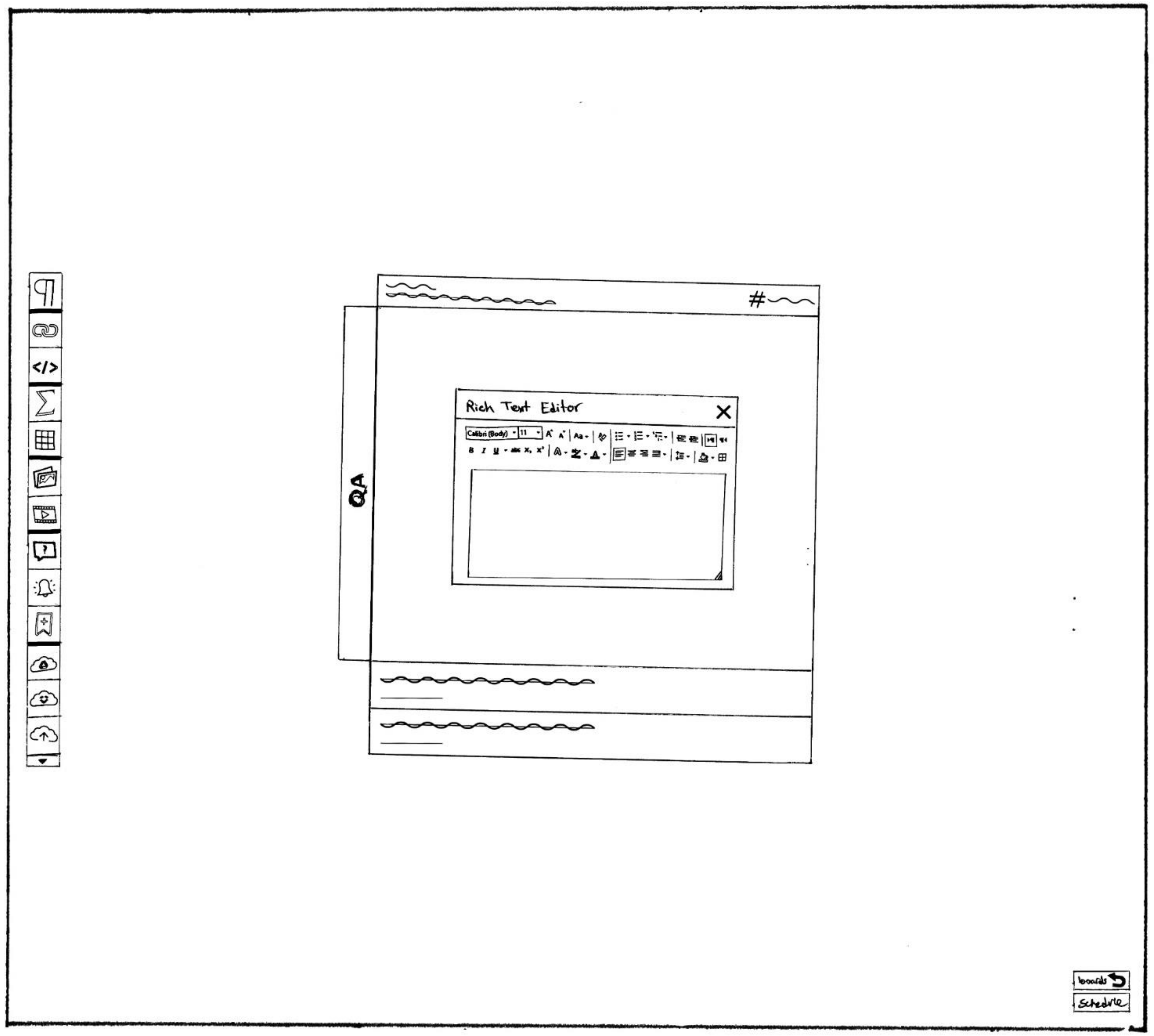
OVERLY COMPLICATED

FAILED TO ADDRESS TASKS

UNORGANIZED

AMBITIOUS

FORGETFUL



A D D

ONE BY ONE

F E A T U R E S

TESTING PROCESS PARTICIPANTS

LEO

Leo, a tech savvy CS student was the first test to be done with our prototype.

CALIN

Communication student. This test was performed in the Marriott Library, due to it being a popular space that students go to study and finish assignments.

ALEX

A student who needs help with optimizing his current study habits. He is a senior computer science student with 15 credit hours

TESTING PROCESS

TASKS

USING BOARDS TO

CREATE A CLASS

ADD AN ALERT

FIND DUE DATE

USING CLASSES TO

FIND HELP ON TOPIC

ADD A REMINDER

CHANGE DUE DATE

FINAL

PROTOTYPE

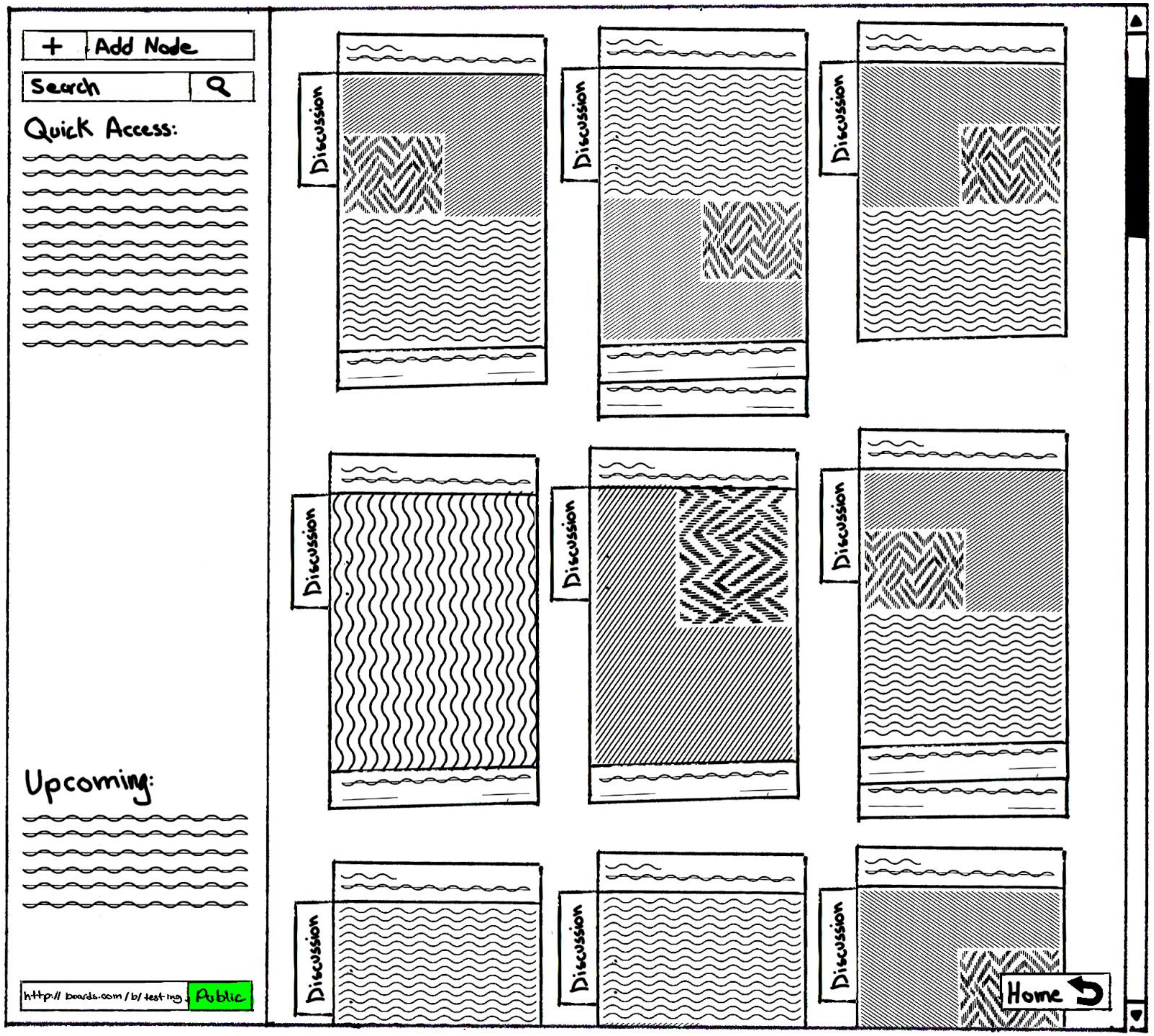
FUNCTIONALITY FIRST

HUGE CHANGES

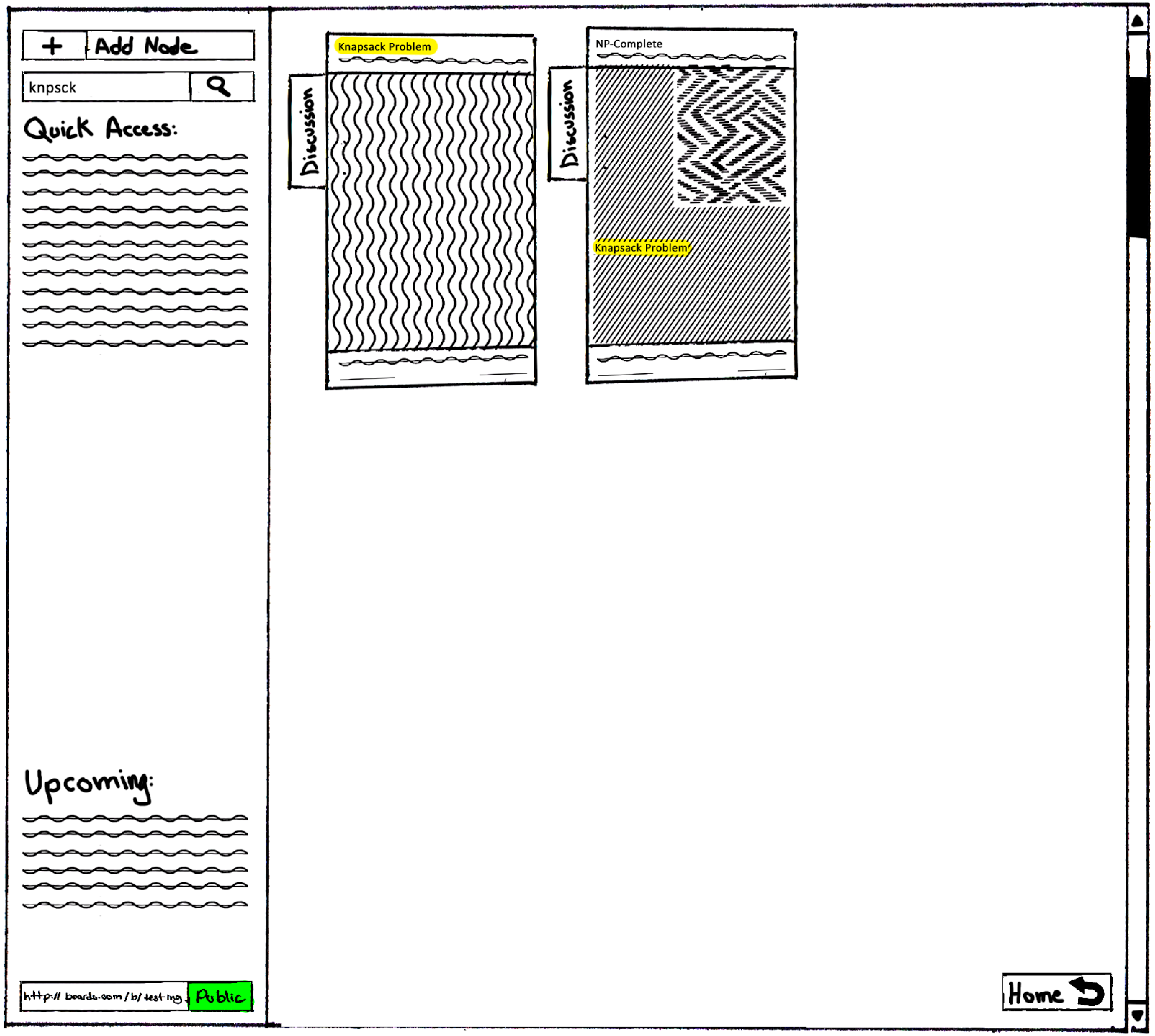
REMOVED FEATURES

SIMPLE & MINIMAL

TASK 1: GETTING HELP



TASK 1: GETTING HELP



TASK 1: GETTING HELP

knapsack problem / np-complete / big-o complexity

Wikipedia:

Knapsack Problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

Continuous Knapsack Problem

In theoretical computer science, the continuous knapsack problem (also known as the fractional knapsack problem) is an algorithmic problem in combinatorial optimization in which the goal is to fill a container (the "knapsack") with fractional amounts of different materials chosen to maximize the value of the selected materials.[1][2] It resembles the classic knapsack problem, in which the items to be placed in the container are indivisible; however, the continuous knapsack problem may be solved in polynomial time whereas the classic knapsack problem is NP-hard.[1] It is a classic example of how a seemingly small change in the formulation of a problem can have a large impact on its computational complexity.

NP-Complete:

In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard. The set of NP-complete problems is often denoted by NP-C or NPC. The abbreviation NP refers to "nondeterministic polynomial time".
Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a consequence, determining whether or not it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems ,,,

Stackoverflow:

Python - Greedy Knapsack with Dictionary input

i am trying to implement a greedy knapsack algorithm in python given the data set below. The output is supposed to be a list of lists, that observes the limit set. In example with the dataset below the output should be :

out = [[C, B, D, A], [Z, F, E]]

Crossover Operation for Binary Encoding

I'm following the genetic algorithm approach to solving the Knapsack problem using DEAP. I understand that they used a direct value encoding scheme rather than a binary representation. The crossover function is as follows:

```
def cxSet(ind1, ind2):
"""Apply a crossover operation on input sets. The first child is the
intersection of the two sets, the second child is the difference of
the
two sets.
"""
```

Multiple inner Knapsack and Fitness Calculation

Scratching my head on this one and I might be thinking it wrong.

Basically it's the Knapsack Problem but modified. You have a set of items with various weights on them and you are to put it in three knapsacks with a capacity of 20 each.

I have codes to initialise all the items randomly in the sacks. That means I can have a sack with more than 20, less than 20, and equal to 20. The problem is that the items are all being added hence my total score is the same for all population making it impossible to mutate.

Google:

Dynamic Programming

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

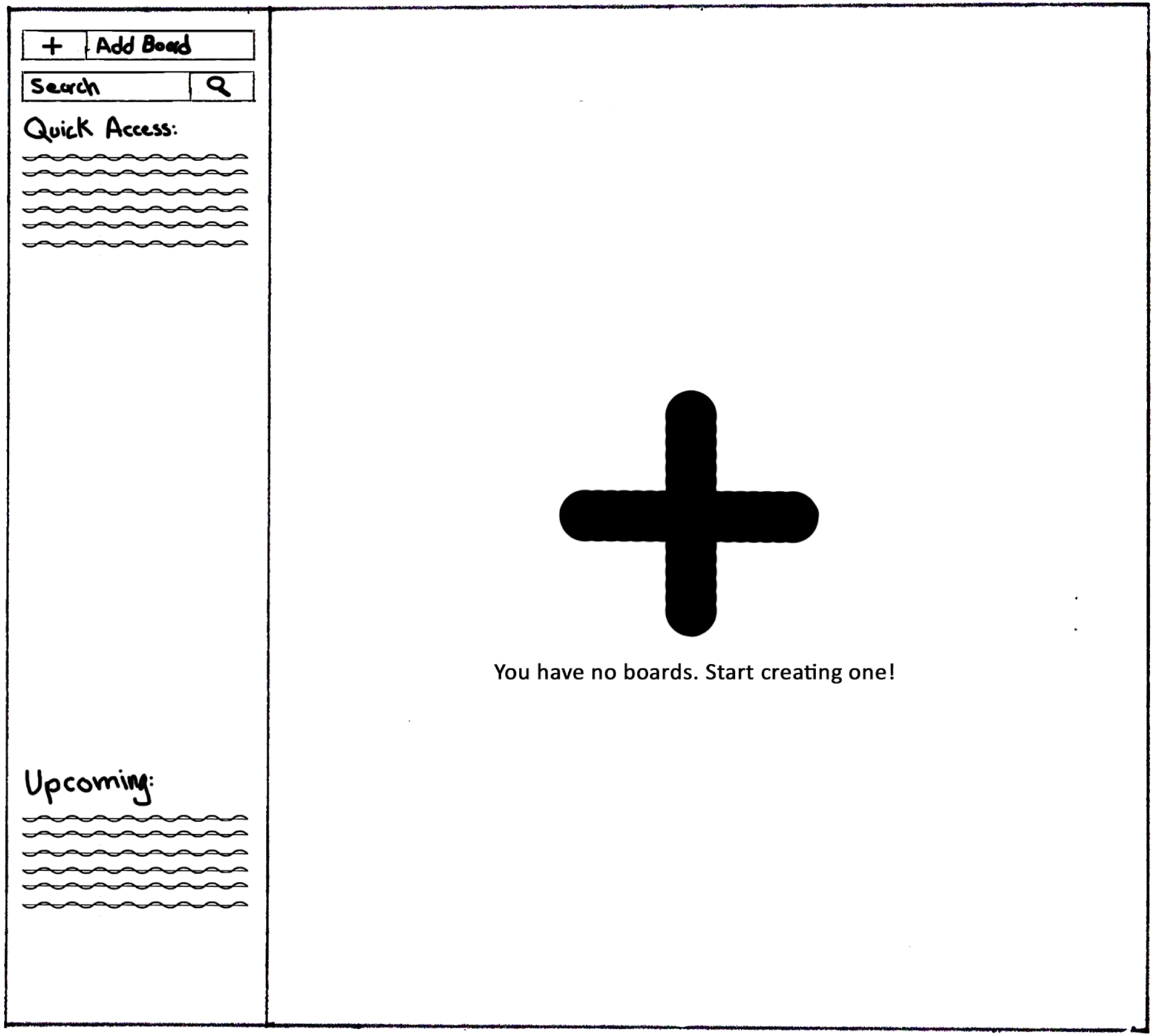
Lecture 13: The Knapsack Problem

Informal Description:
We have computed [illegible] data files that we want to store, and we have available bytes of storage.
File A has size [illegible] bytes and takes [illegible] minutes to recompute.
We want to avoid as much recomputing as possible,
so we want to find a subset of files to store such that
The files have combined size at most [illegible]
The total computing time of the stored files is as
large as possible.
We can not store parts of files, it is the whole file or
nothing.

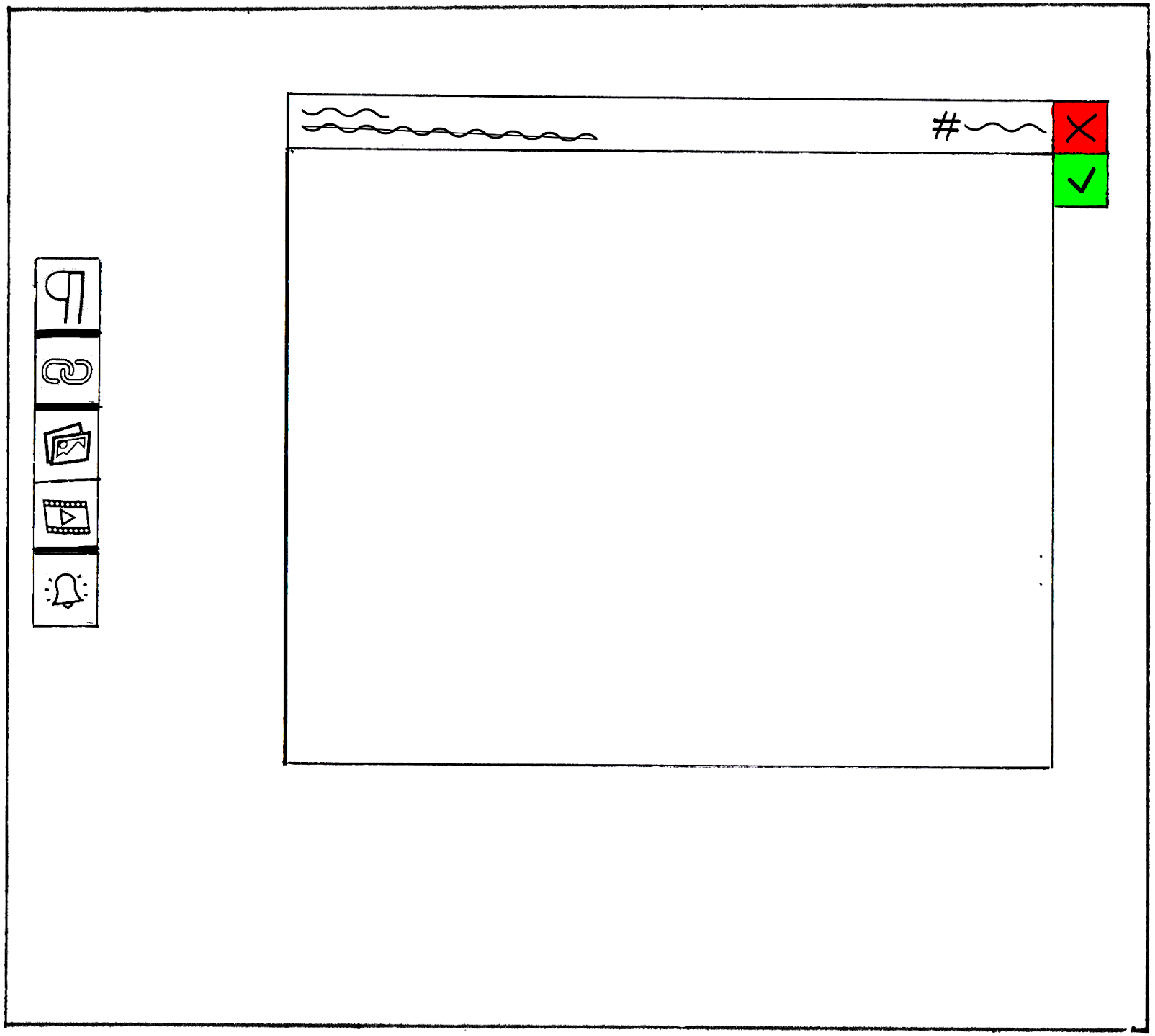
The Knapsack Problem

Suppose we are planning a hiking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip. There are N different item types that are deemed desirable; these could include bottle of water, apple, orange, sandwich, and so forth. Each item type has a given set of two attributes, namely a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type of item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value. What we have just described is called the knapsack problem.

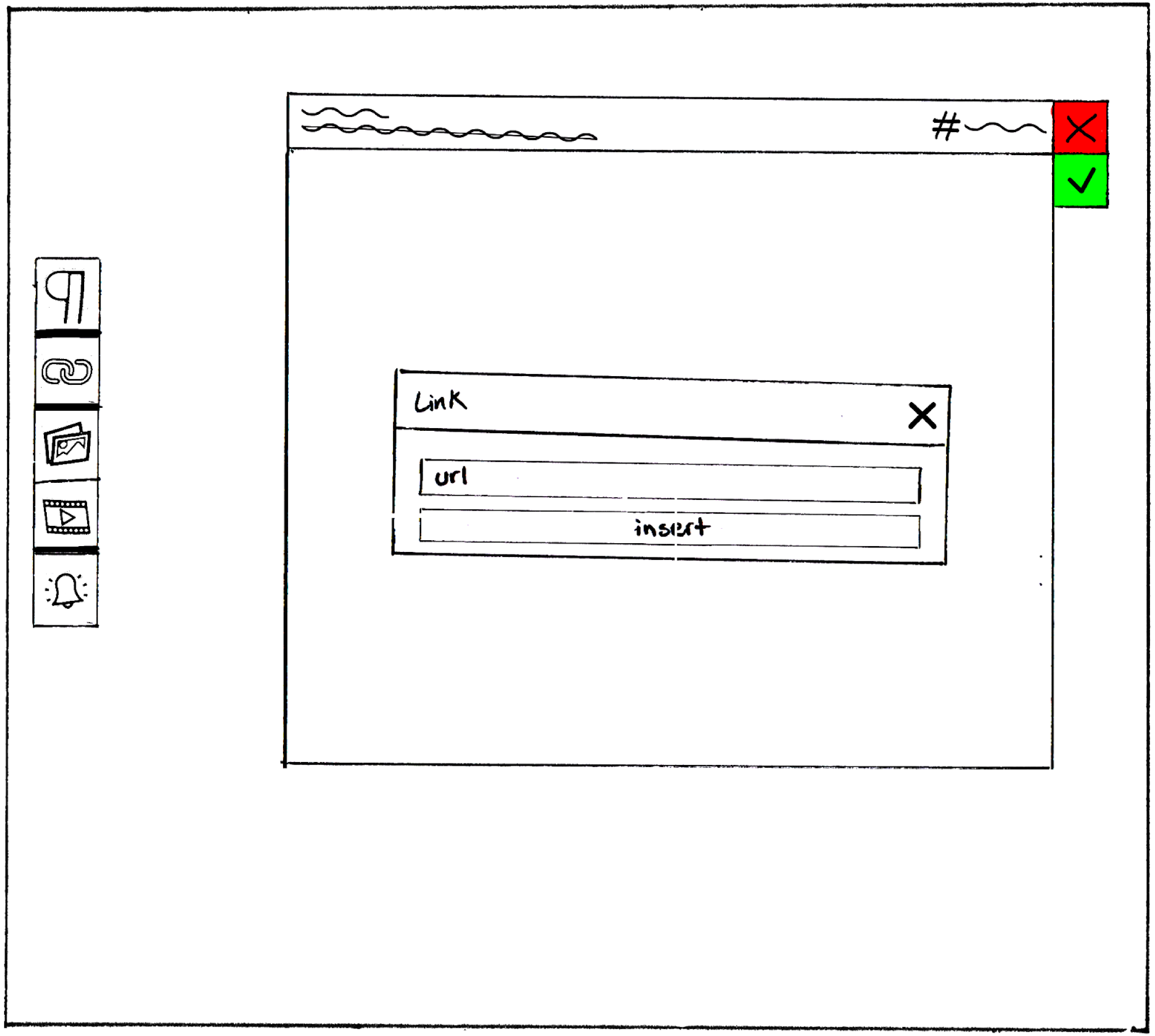
TASK 2: ORGANIZE COURSE MATERIAL



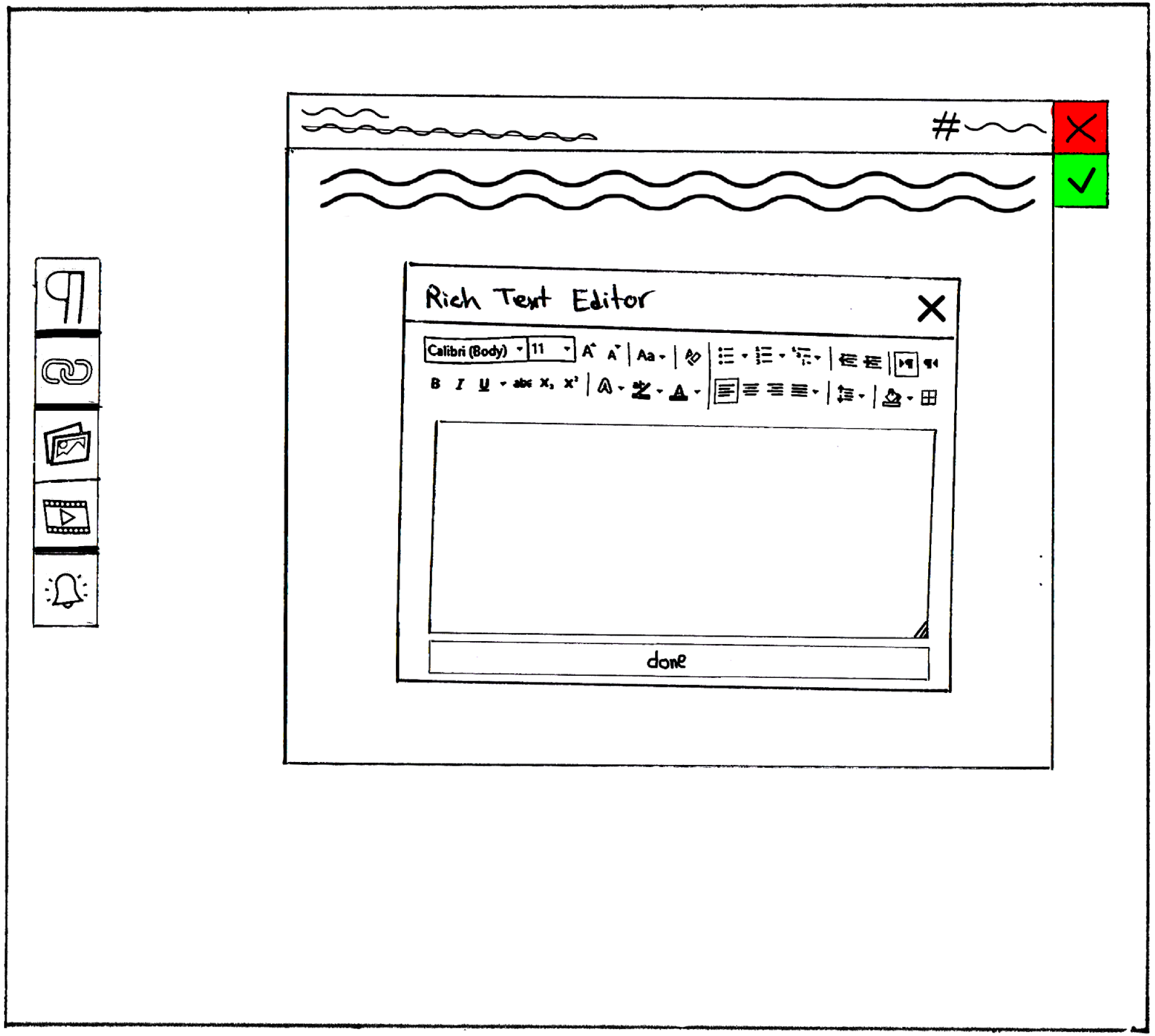
TASK 2: ORGANIZE COURSE MATERIAL



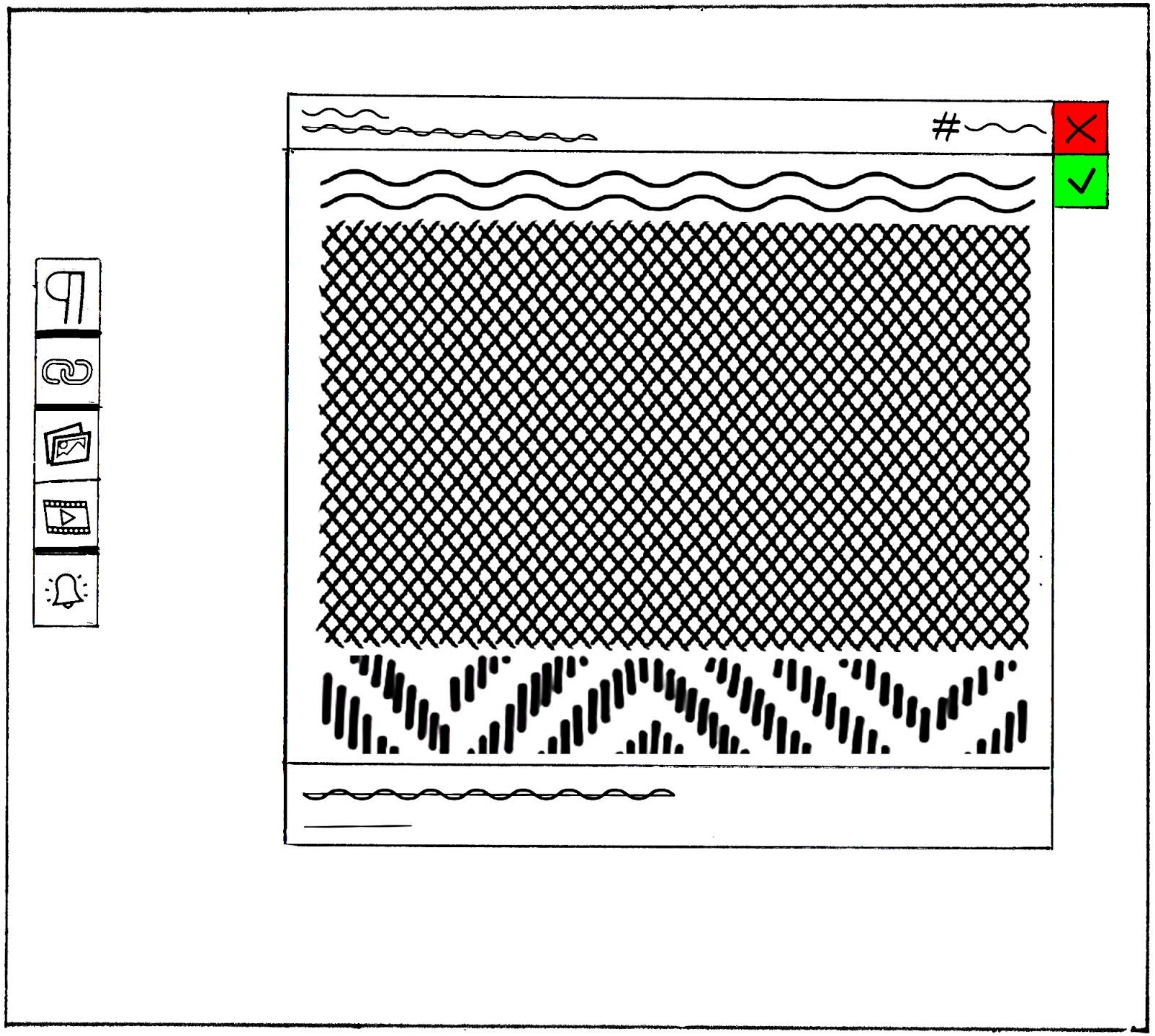
TASK 2: ORGANIZE COURSE MATERIAL



TASK 2: ORGANIZE COURSE MATERIAL



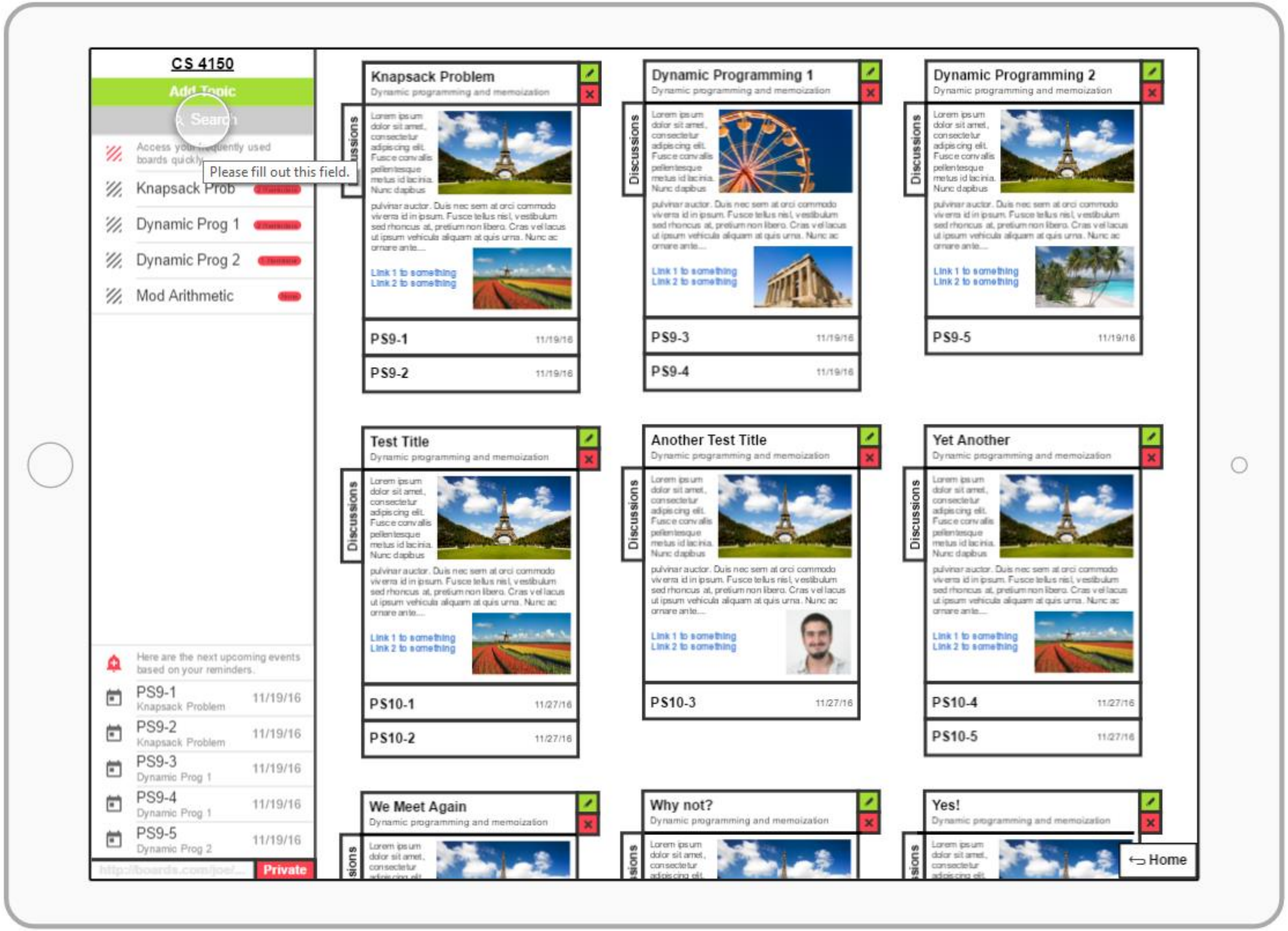
TASK 2: ORGANIZE COURSE MATERIAL



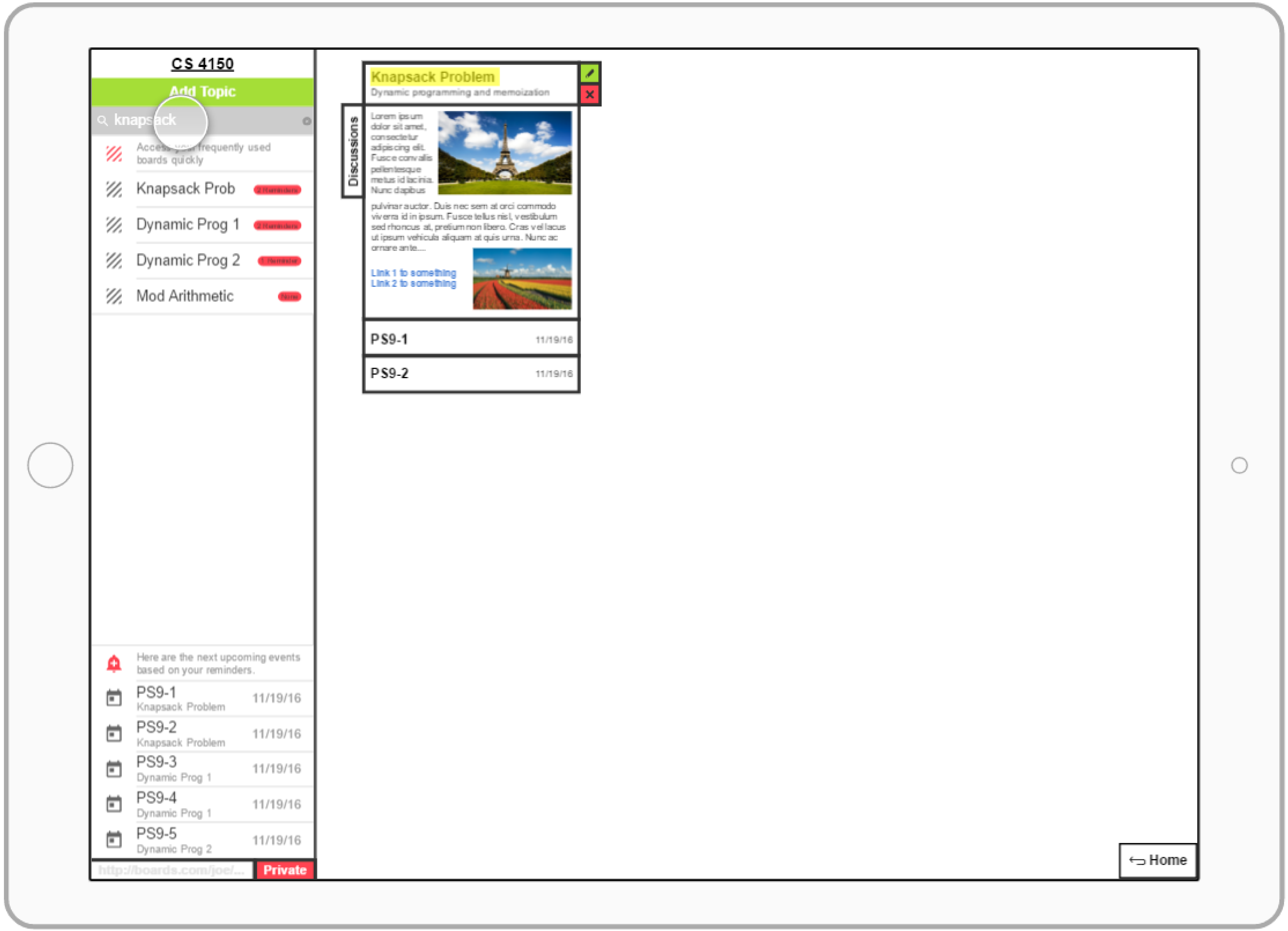
LETS GET DIGITAL

TASK 1: GETTING HELP

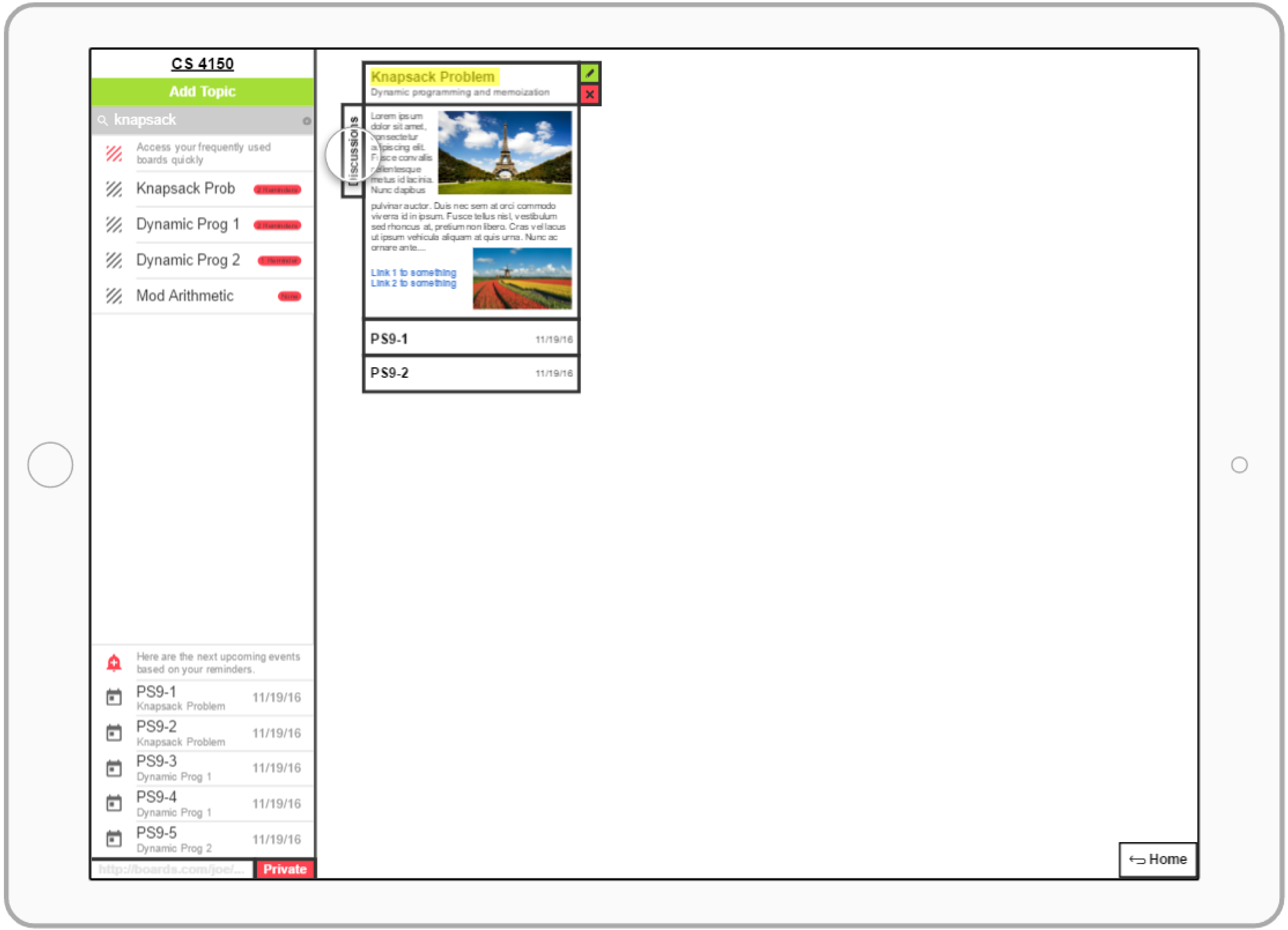
TASK 1: GETTING HELP



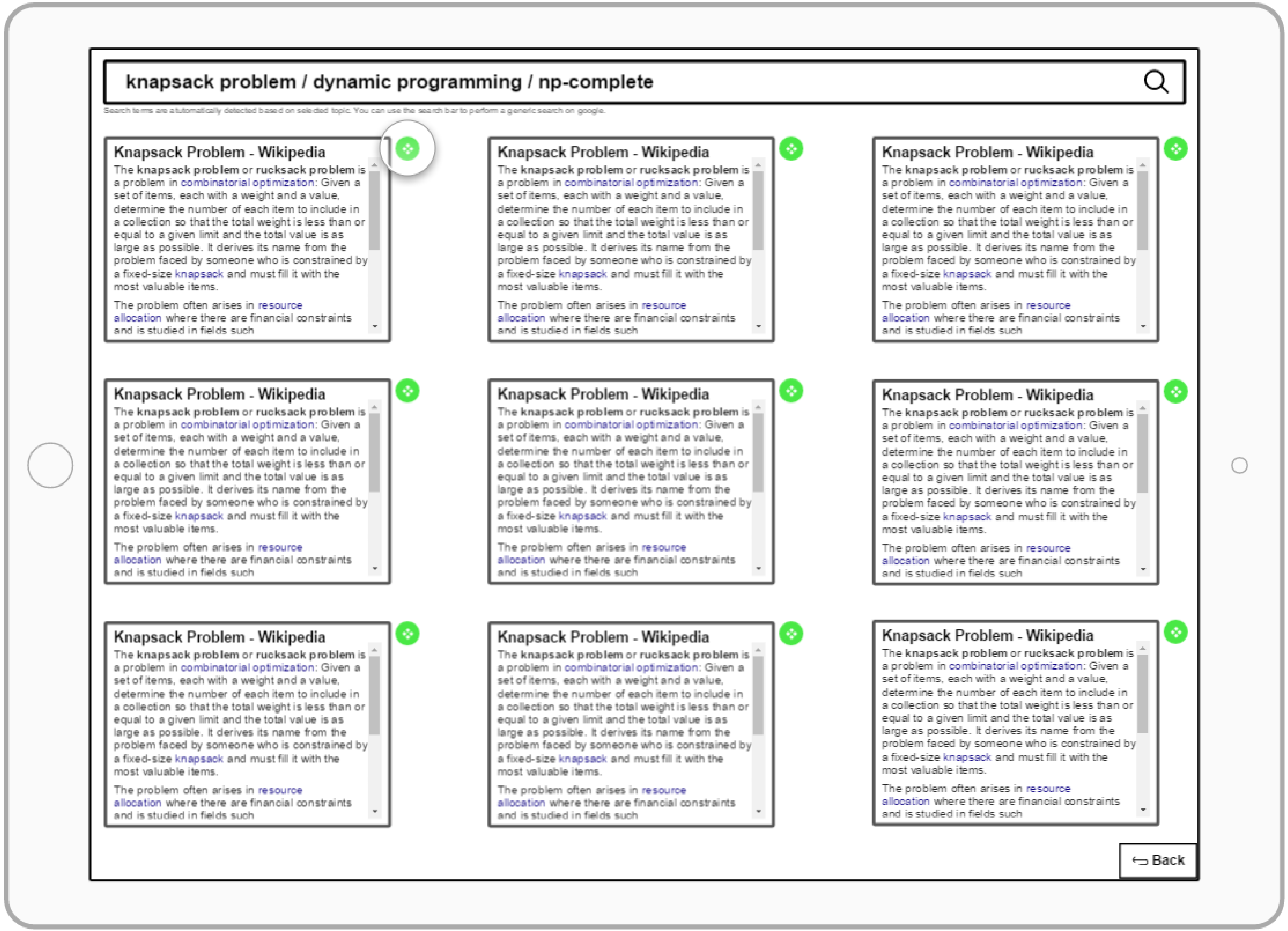
TASK 1: GETTING HELP



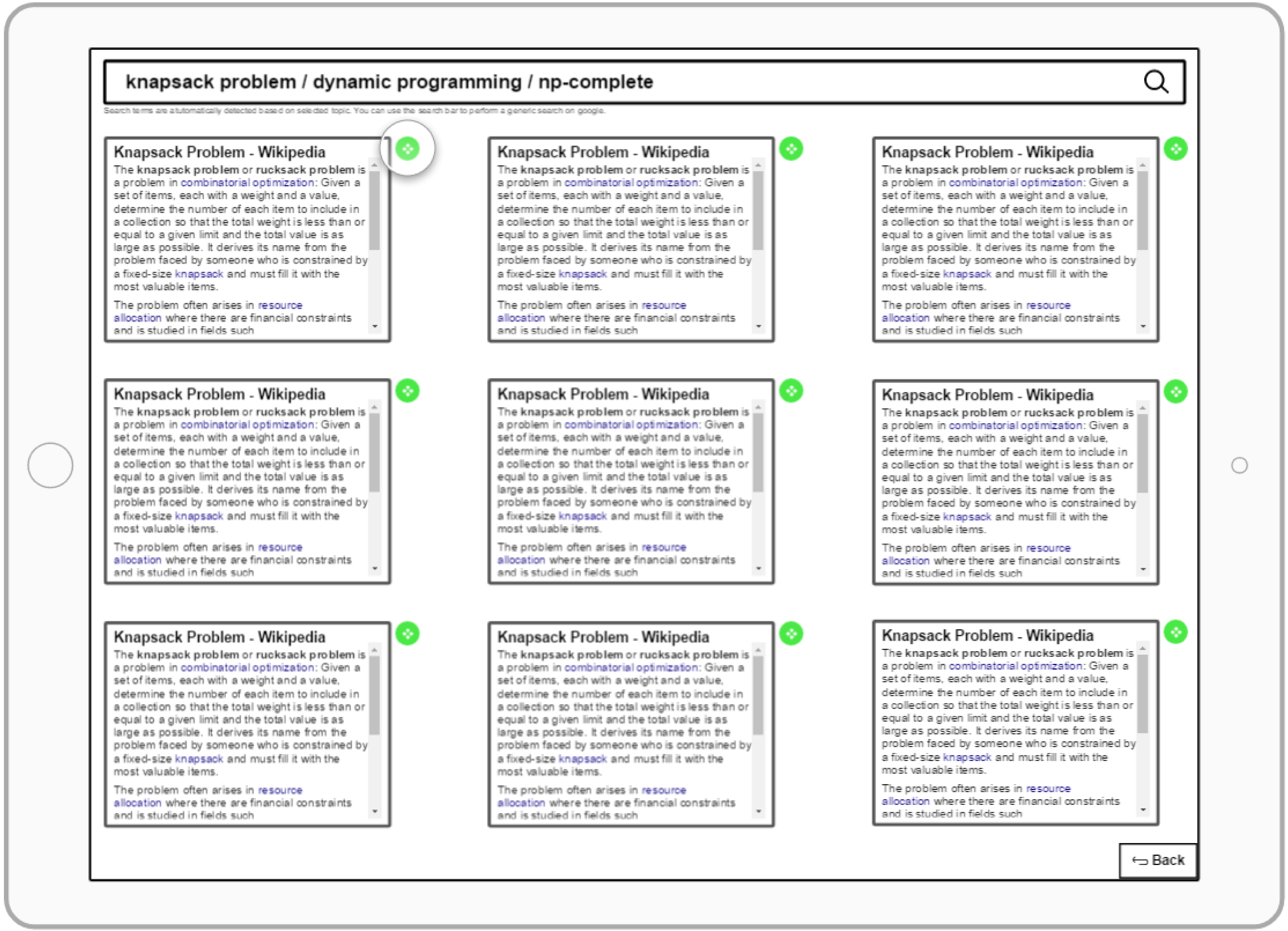
TASK 1: GETTING HELP



TASK 1: GETTING HELP



TASK 1: GETTING HELP



TASK 1: GETTING HELP

Knapsack Problem - Wikipedia

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897.[1] The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956),[2] and refers to the commonplace problem of packing your most valuable or useful items without overloading your luggage.

Contents [hide]

Applications [edit]

A 1998 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithmic problems, the knapsack problem was the 18th most popular and the 4th most needed after kd-trees, suffix trees, and the bin packing problem.[3]

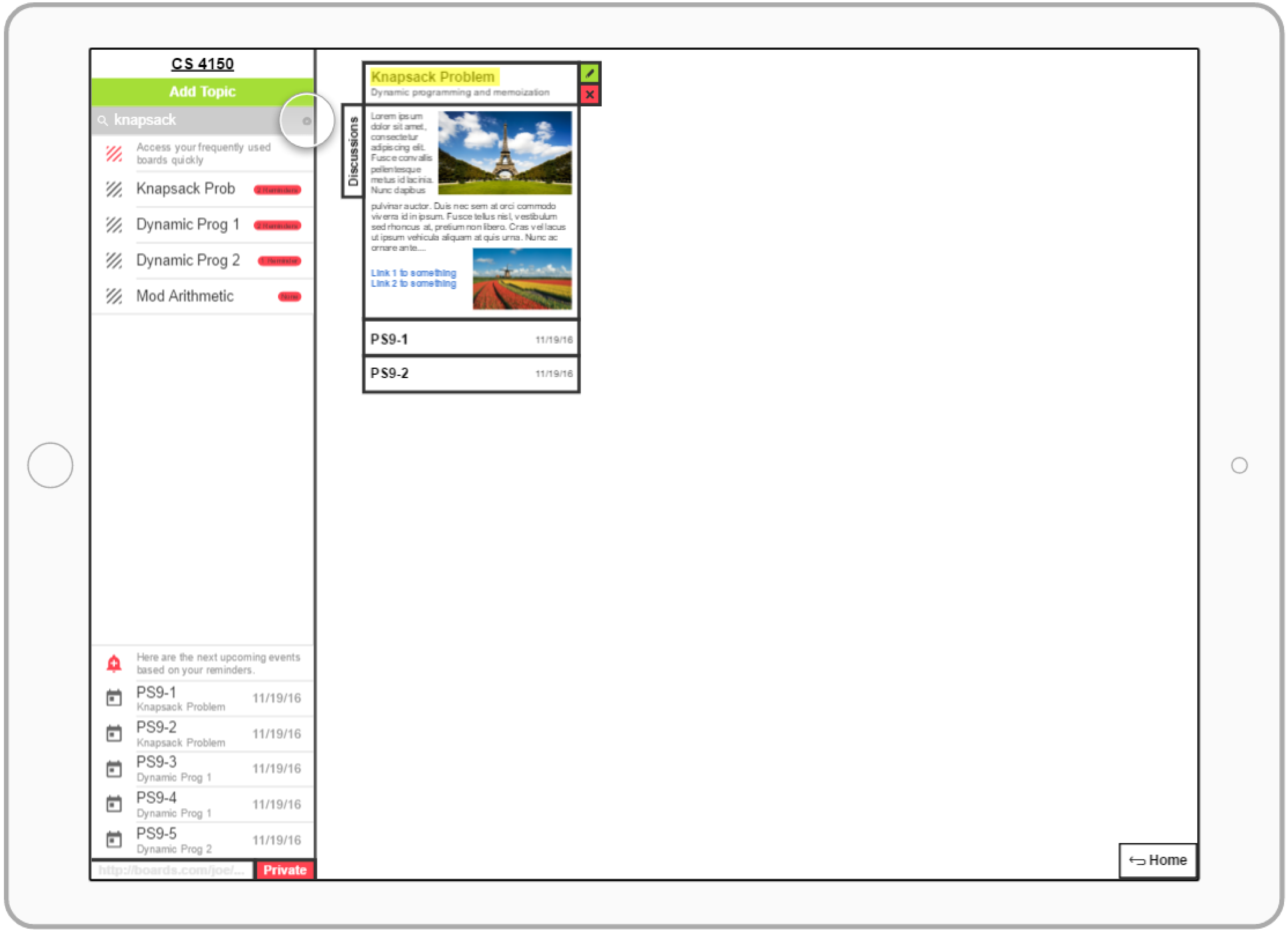
Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials,[4] selection of investments and portfolios,[5] selection of assets for asset-backed securitization,[6] and generating keys for the Merkle–Hellman[7] and other knapsack cryptosystems.

One early application of knapsack algorithms was in the construction and scoring of tests in which the test-takers have a choice as to which questions they answer. For small examples it is a fairly simple process to provide the test-takers with such a choice. For example, if an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a heterogeneous distribution of point values—i.e. different questions are worth different point values— it is more difficult to provide choices. Feuerman and Weiss proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a knapsack algorithm would determine which subset gives each student the highest possible score.[8]

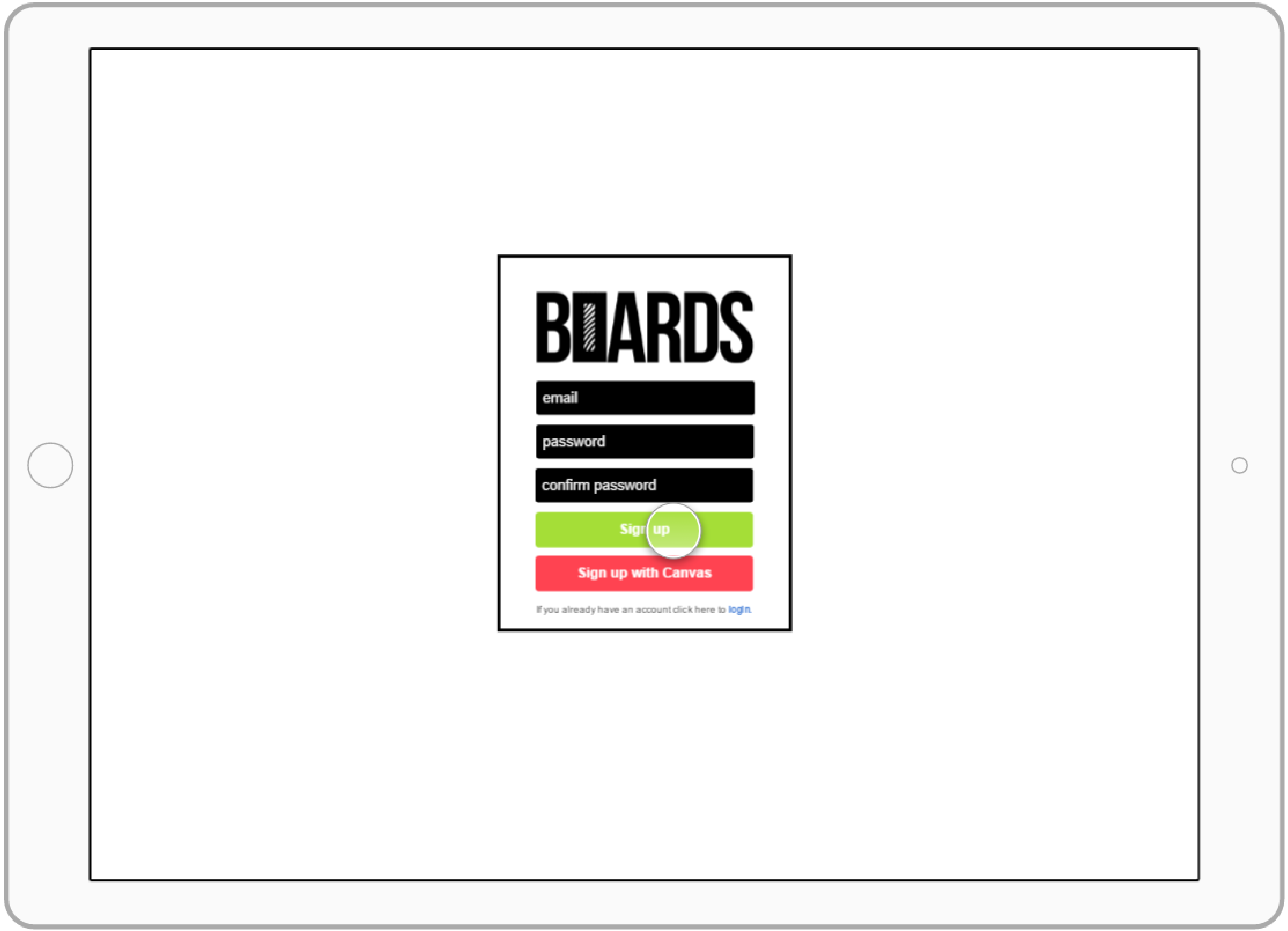
Definition [edit]

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W,

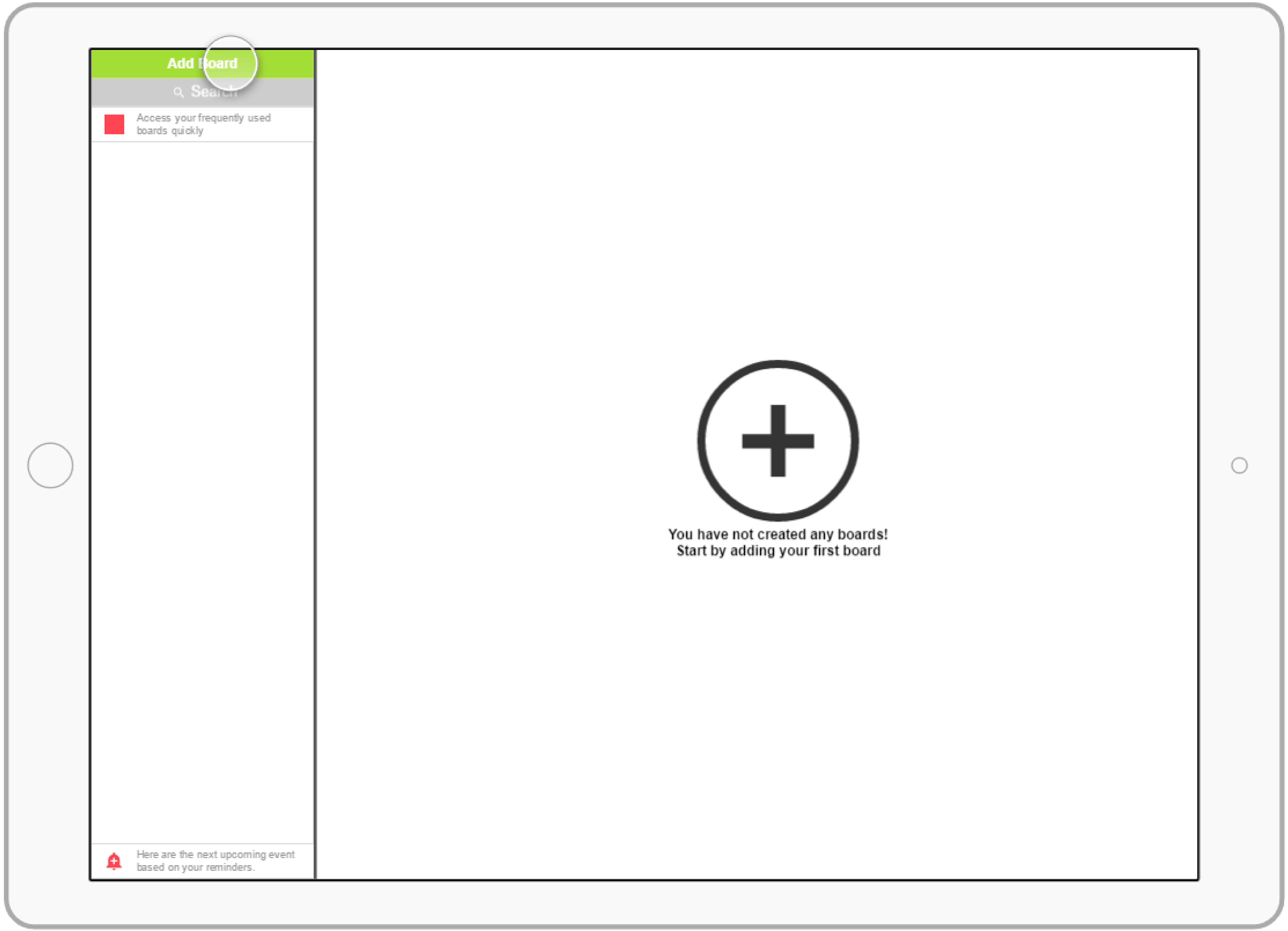
TASK 1: GETTING HELP



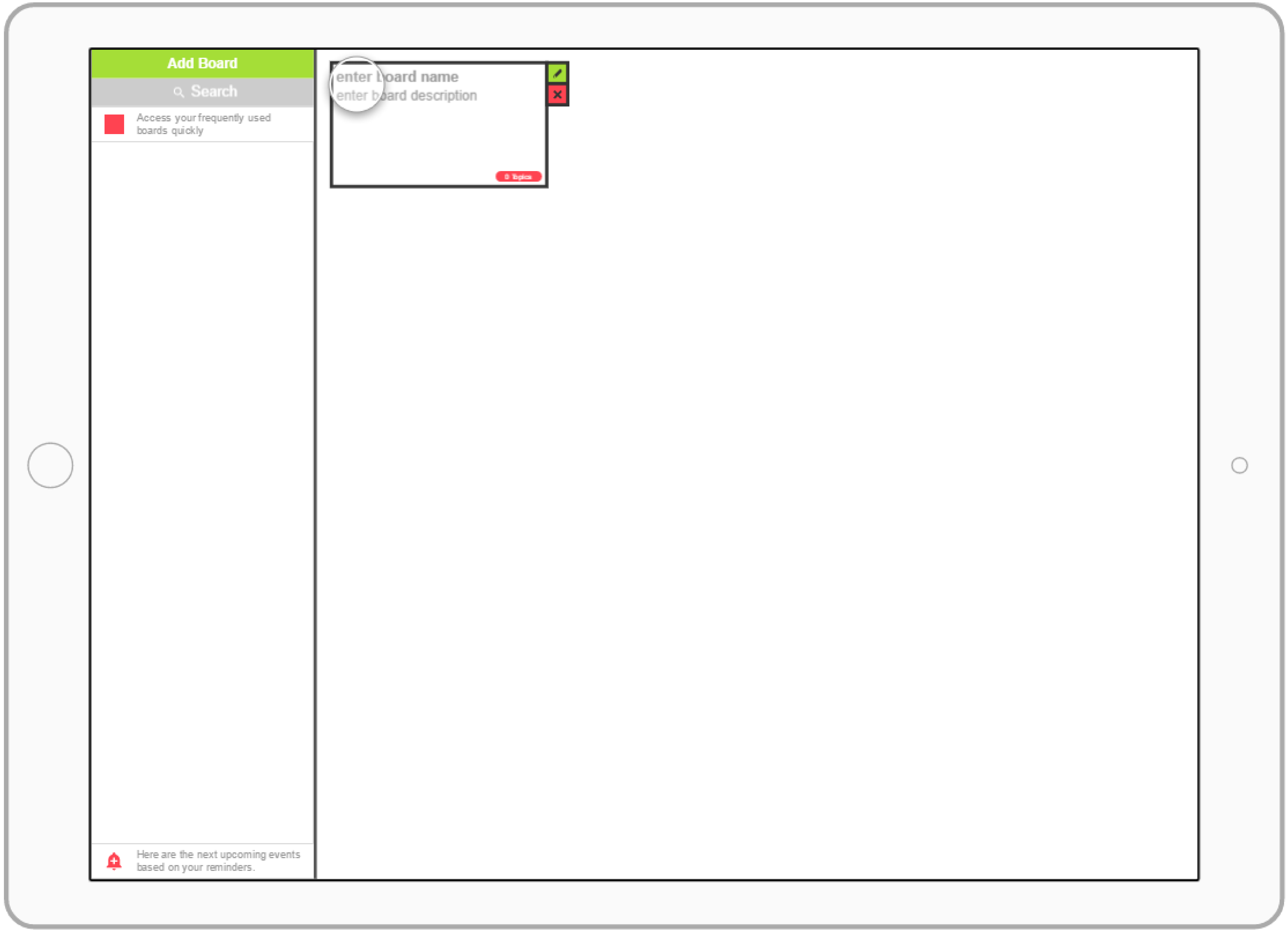
TASK 1: GETTING HELP



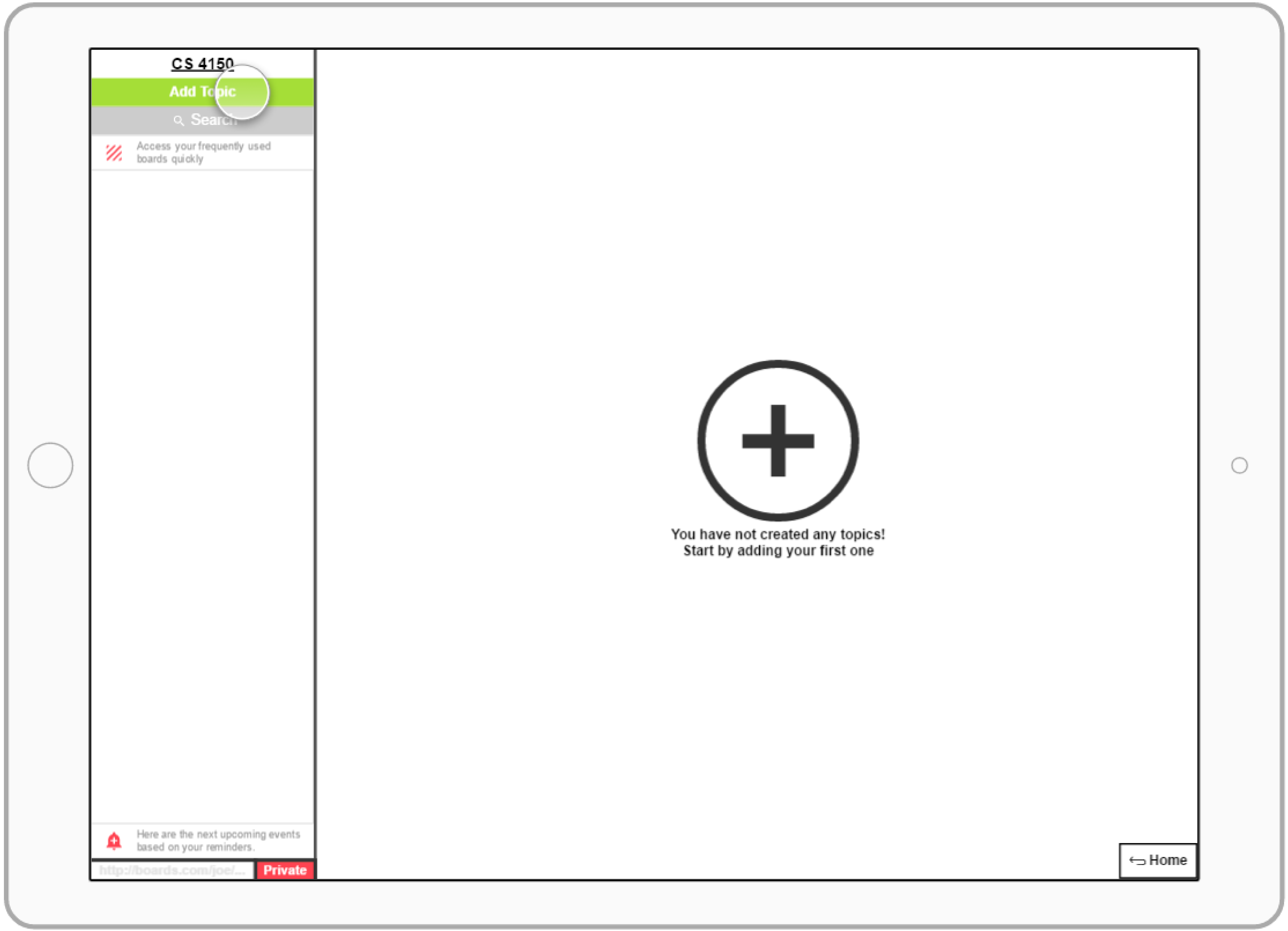
TASK 2: ORGANIZE COURSE MATERIAL



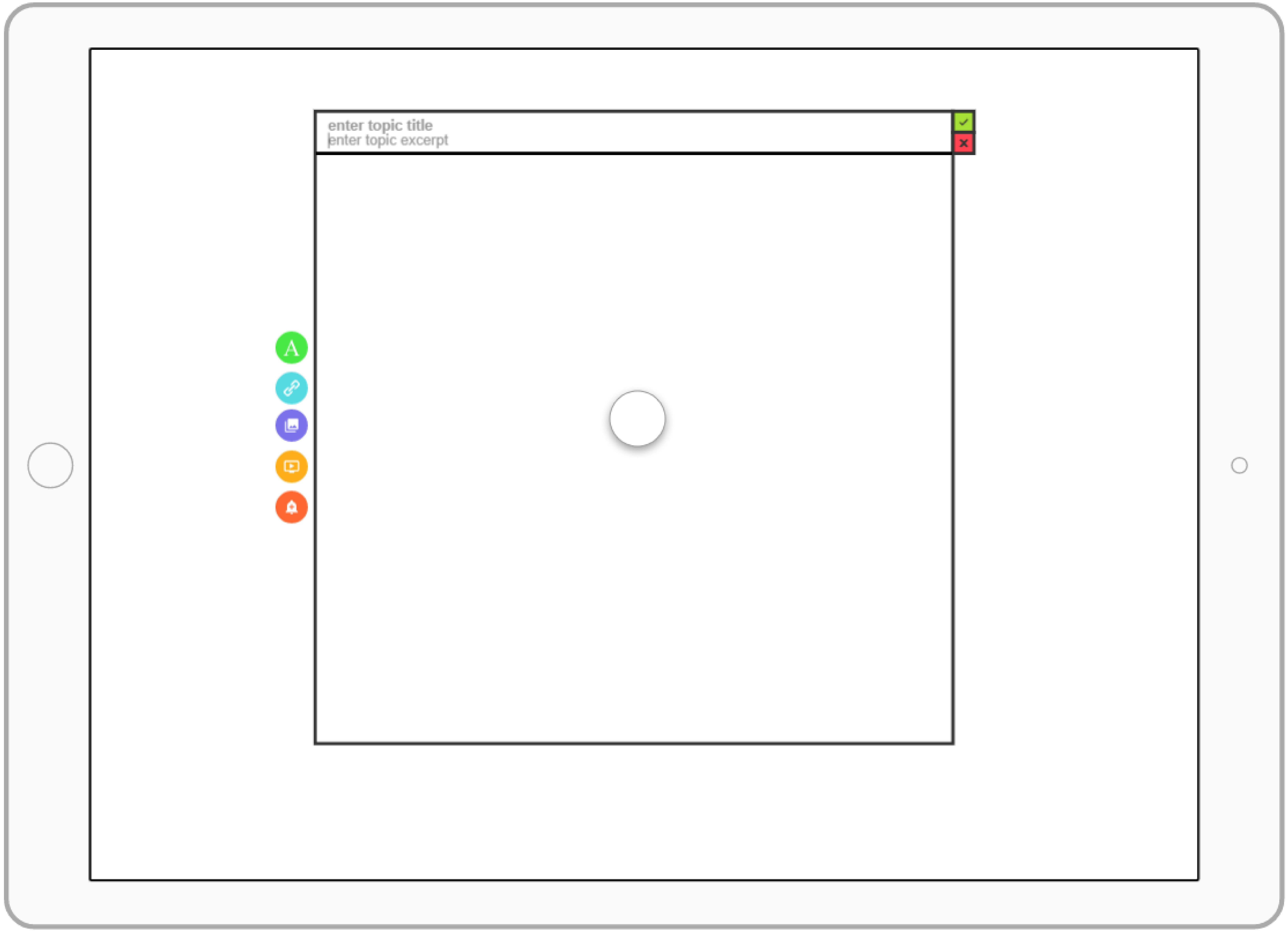
TASK 2: ORGANIZE COURSE MATERIAL



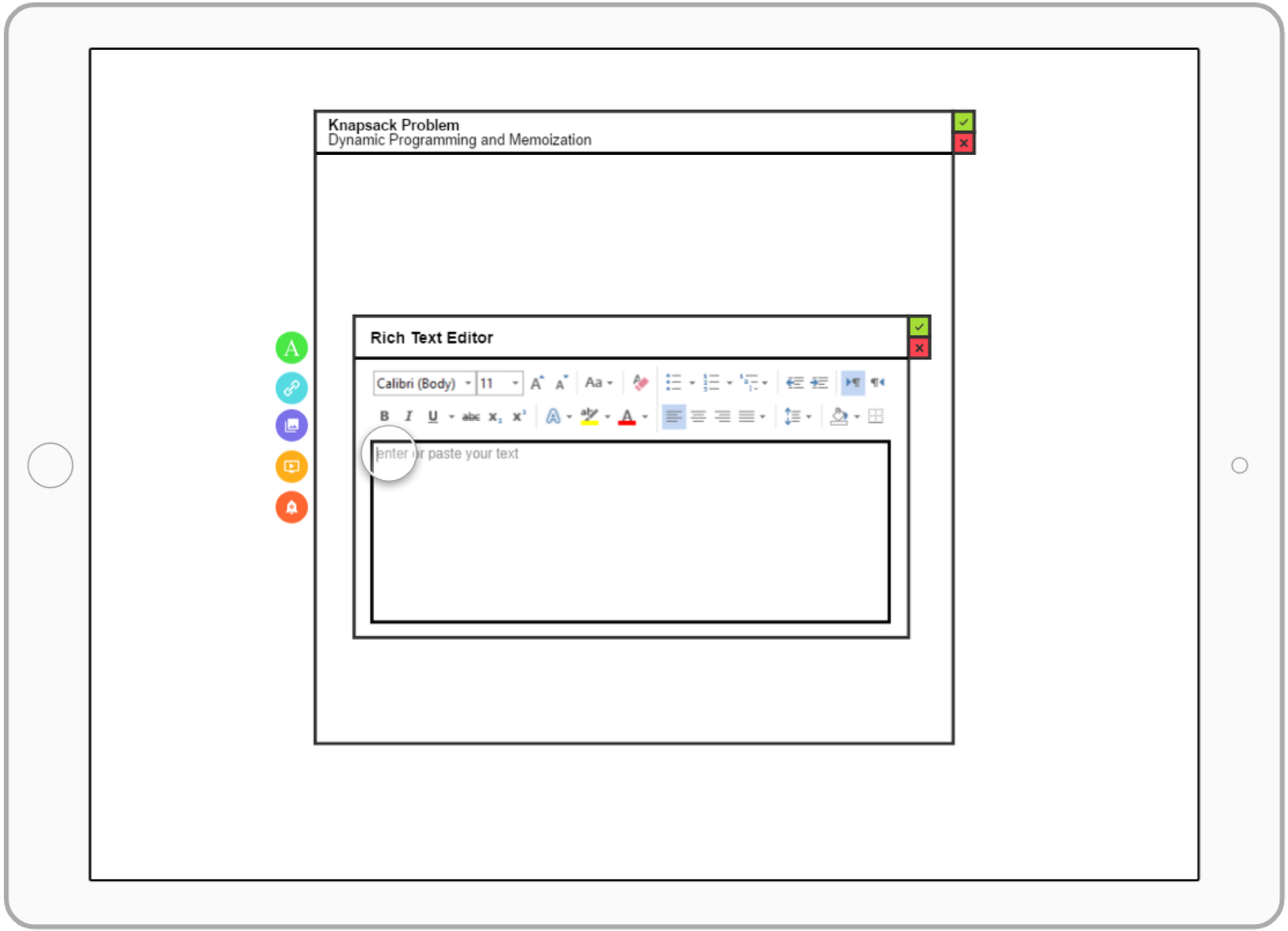
TASK 2: ORGANIZE COURSE MATERIAL



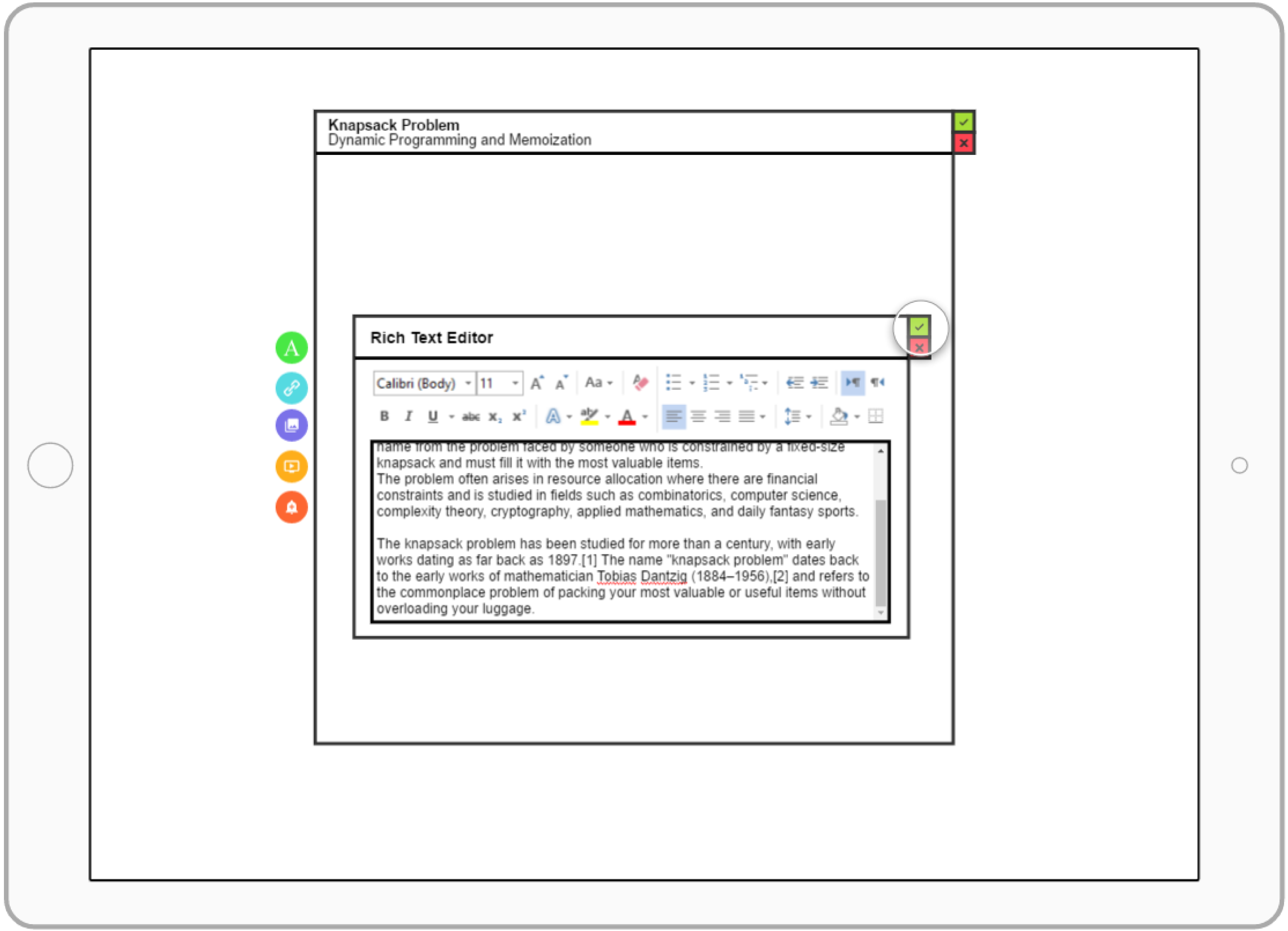
TASK 2: ORGANIZE COURSE MATERIAL



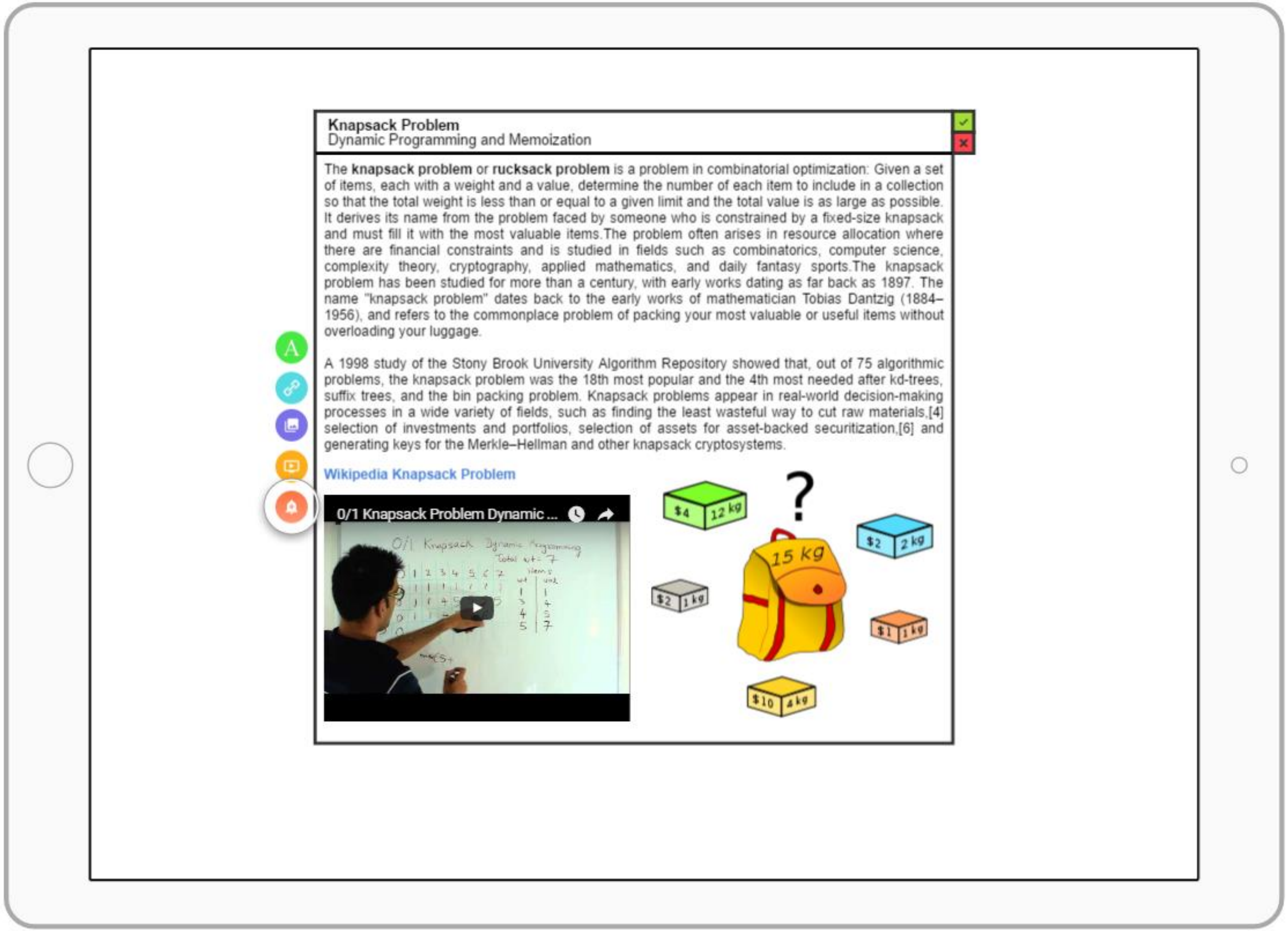
TASK 2: ORGANIZE COURSE MATERIAL



TASK 2: ORGANIZE COURSE MATERIAL



TASK 2: ORGANIZE COURSE MATERIAL



TASK 2: ORGANIZE COURSE MATERIAL

TASK 2: ORGANIZE COURSE MATERIAL

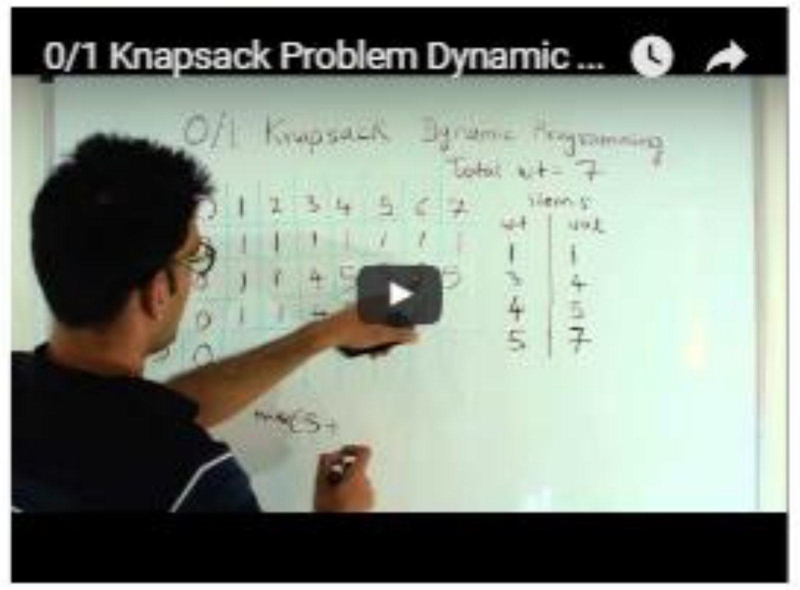
TASK 2: ORGANIZE COURSE MATERIAL

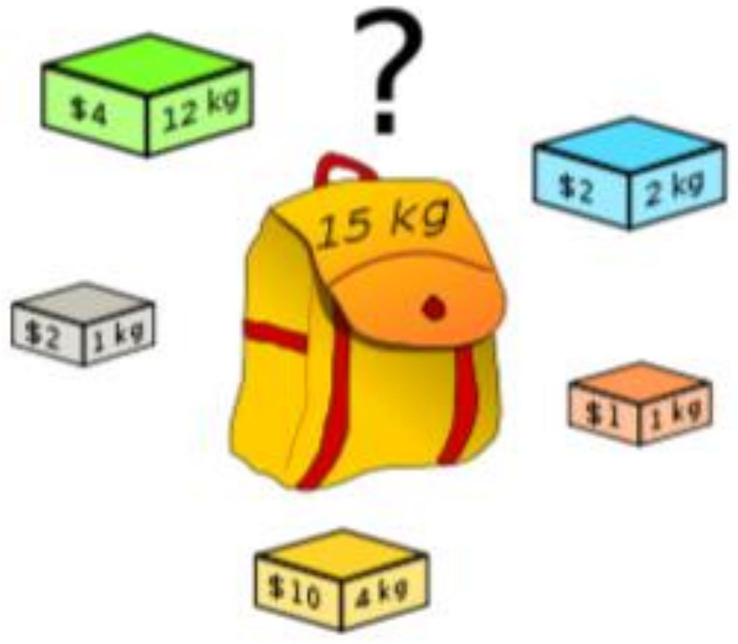
Knapsack Problem
Dynamic Programming and Memoization

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956), and refers to the commonplace problem of packing your most valuable or useful items without overloading your luggage.

A 1998 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithmic problems, the knapsack problem was the 18th most popular and the 4th most needed after kd-trees, suffix trees, and the bin packing problem. Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials,[4] selection of investments and portfolios, selection of assets for asset-backed securitization,[6] and generating keys for the Merkle–Hellman and other knapsack cryptosystems.

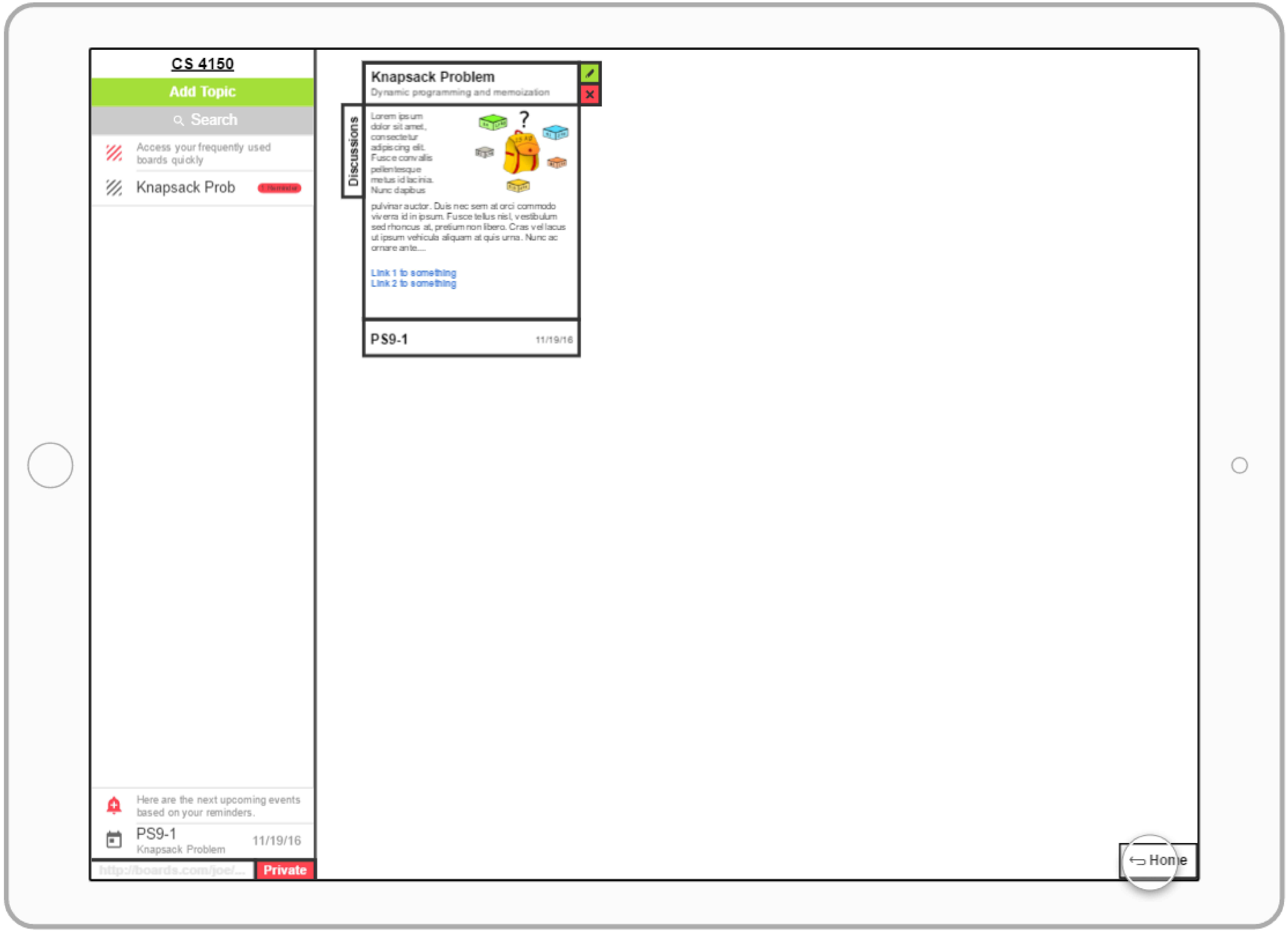
Wikipedia Knapsack Problem





PS9-1
Short description for the reminder module. Short description for the reminder module.
11/19/2016

TASK 2: ORGANIZE COURSE MATERIAL



TASK 2: ORGANIZE COURSE MATERIAL

SUMMARY

Be aware of the scope.

Start small, grow gradually.

Be conscious of your choices.

Testing is never enough.

Modularity saves you time.

Iterate often, yet be smart.

Ask for opinions.

If it’s not broken, don’t fix it.

QUESTIONS?