General Introduction: Recursion is a great technique for problem solving, even beyond cases where your teacher required you to use it. However, sometimes implementing a solution recursively can lead to problems if sub-problems are repeated. Let's examine computing the Fibonacci numbers. Recall that $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$. What happens if you implement that recursively? Is there a better way?

Weighted Interval Scheduling

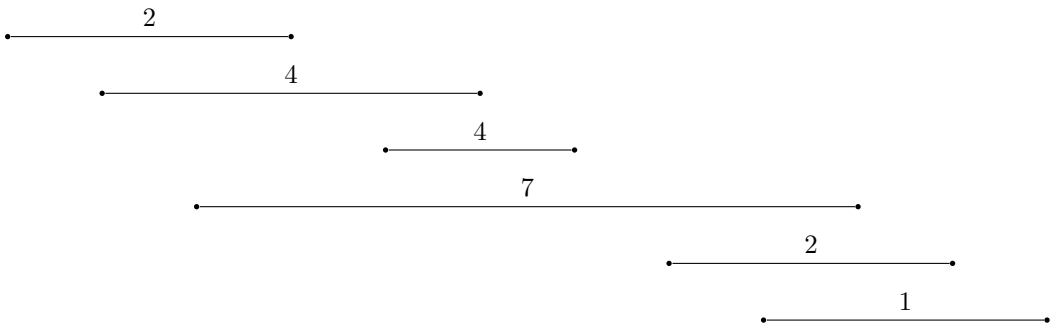
Warm-Up: we are given a set of n intervals, numbered $1 \dots n$, each of which has a start time s_i and a finish time f_i. For each interval, we want to compute a value $p[i]$, which is the interval j with the *latest* finish time f_j such that $f_j \leq s_i$; that is, the last-ending interval that finishes before interval i starts. If no intervals end before interval i begins, then $p[i] = 0$.

Give an $\mathcal{O}(n \log n)$ time algorithm that computes $p[i]$ for all intervals. You may assume that the intervals are already sorted by finish time.

The Big Problem: Fed up after the first two quizzes in CompSci 161, your friend has decided to change majors to one that grades based only on attendance. The only question is which classes your friend should take in Fall quarter. The classes all meet once a day, at different times and lengths, and are worth different amounts of credits. Your friend's goal is to maximize the amount of credits earned in that quarter without having to skip any classes (as this may interfere with passing those classes).

Problem Statement: More formally, we are given a set of n intervals, each of which has a start time s_i, a finish time f_i, and a value v_i. Our goal is to select a subset of the intervals such that no two selected intervals overlap and the total value of those taken is maximized.

Example Input: Please be aware that sample input will not always be provided in CompSci 161; one of the educational objectives is for you to be able to solve a problem in the abstract.



Let's solve this *recursively* (yay!). We will write a function `OPT(`i`)` that returns the optimal number of credits obtainable among intervals (classes) $1 \dots i$. We can then call `OPT(n)` to figure out the optimal number of credits obtainable among all intervals.

The key observation here is that your friend will either take class i or your friend won't take class i.

```
OPT(i)

    // Base Case:



   // If my friend doesn't take class i:

   value_if_not_taken =


   // If my friend takes class i:

   value_if_taken =


   //return something:
```

Should we implement it that way? We now have a recursive solution. Think back to the Fibonacci example earlier in lecture. What will happen if we implement this program this way?

Iterative Solution

We can now move to have a solution that uses no recursive *function calls*. Note that our solution is still conceptually recursive. The iterative solution will also allow us to output which courses to take, not simply the optimal value.

i	$p[i]$	v_i	OPT(p(i))	OPT(p(i)) + v_i	OPT($i-1$)	OPT(i)
0		N/A	N/A	N/A	N/A	0
1		2				
2		4				
3		4				
4		7				
5		2				
6		1				

Dynamic Programming is not about filling in tables.
Dynamic Programming is about smart recursion.

Longest Common Subsequence

Problem Statement: A *subsequence* of a given sequence is just the given sequence with zero or more elements left out. Given two sequences X and Y, we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y. Our goal is to find the maximum length common subsequence.

Examples:

X	Y	LCS
complete	continue	cote
exercise	determine	eerie
surface	character	race
morning	triangle	ring
toward	thousand	toad

As with the previous lecture, let's determine the general recursive solution first. Can you determine something tautological about the LCS of sequences X and Y?

Example: What is the LCS of the sequences < M O R N I N G > and < T R I A N G L E > ?

		M	O	R	N	I	N	G
T								
R								
I								
A								
N								
G								
L								
E								

Edit Distance

The **Edit distance** problem is as follows. We are given two strings (not necessarily of equal length). We want to convert the first string to the other by a sequence of insertions, deletions, and substitutions. The **cost** is the number of operations we perform.

For example, if we want to convert FOOD to MONEY, we could do this:
FOOD $\rightarrow$ MOOD $\rightarrow$ MOND $\rightarrow$ MONED $\rightarrow$ MONEY

One way to visualize this is by alignment:

F	O	O		D
M	O	N	E	Y

We **define** Edit(i, j) to be the **minimum cost** to convert $X[1 \ldots i]$ to $Y[1 \ldots j]$.

What happened in the last column?

		A	L	G	O	R	I	T	H	M
A										
L										
T										
R										
U										
I										
S										
T										
I										
C										

Subset Sum

Problem Statement: Given a set S of n positive integers, as well as a positive integer T, determine if there is a subset of S that sums to exactly T.

Example 1: $S = \{2, 3, 4\}$, $T = 6$, the answer is "yes"

Example 2: $S = \{2, 3, 5\}$, $T = 6$, the answer is "no"

As with all other dynamic programming algorithms, we are going to start with a recursive case and transform it from there. Remember, *dynamic programming is about smart recursion.*

- Find a recursive algorithm to determine if a subset of the first n values in the input adds up to T.

- Finish the process to make this a dynamic programming algorithm, including outputting the subset of the items for the case when the answer is "yes." For example, your output on example one (above) should be "yes, 2, 4" while your output for example two should be "no."

Here are the tables for the Subset Sum examples.

Example 1: $S = \{2, 3, 4\}$, $T = 6$.

	0	1	2	3	4	5	6
{}							
{2}							
{2, 3}							
{2, 3, 4}							

Example 2: $S = \{2, 3, 5\}$, $T = 6$.

	0	1	2	3	4	5	6
{}							
{2}							
{2, 3}							
{2, 3, 5}							

What is the running time of the dynamic programming algorithm you gave for Subset Sum above?

- Suppose we double the size of S, but leave T alone. Will your algorithm scale well?

- Suppose we double the value of T, but leave S alone. Will your algorithm scale well?

One of the key terms here is *pseudo-polynomial*. We will discuss the effect that has on efficiency as the quarter progresses.

Offline Optimal Binary Search Trees

In ICS 46, you saw unbalanced binary search trees. These had $\mathcal{O}(\log n)$ lookup time under some conditions, but $\mathcal{O}(n)$ lookup time in the worst case. You then saw various forms of balanced binary search trees: AVL and Red/Black trees made the "promise" that any given lookup in the tree would take $\mathcal{O}(\log n)$ time. We can't reasonably expect a better worst case, and these are great data structures for the case when elements can be added to the tree arbitrarily and we don't know how often (or even if) we will look up any given element.

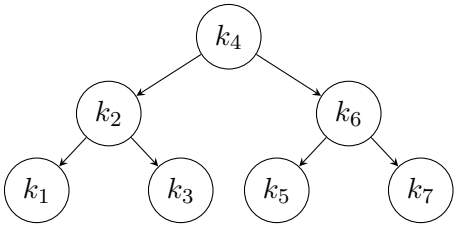
Suppose, though, that before building a binary search tree, we knew exactly which elements were going to be in the tree. If we're likely to look up any given one with equal probability, or if we don't know the likelihood of looking up any given element, we can balance the tree by placing the median element at the root and recursively building trees in this fashion for the left- and right-subtrees.

But what if we also knew the probability that we'd look up any given element once the tree was built? This might not produce an optimal binary search tree in terms of the expected value of the lookup. Suppose we have n keys, $k_1 \dots k_n$, with probabilities $p_1 \dots p_n$ that we will look up the given elements; each probability p_i is positive, and the sum of these is 1.

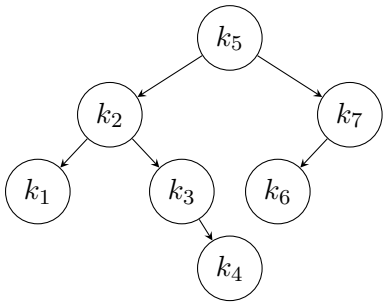
Here's an example with $n = 7$ keys:

i	1	2	3	4	5	6	7
p_i	.13	.21	.11	.01	.22	.08	.24

Here are two possible binary search trees with those keys:



This tree is balanced



This one is less balanced

What is the *expected* lookup cost (in terms of nodes examined) for each of these trees?

Problem Statement: We are given n probabilities, $p_1 \dots p_n$; p_i represents the probability of looking up the ith smallest key once the tree is built. Our goal is to build a binary search tree with the smallest expected lookup cost.
Note that our output must be a *binary search tree*; we cannot reorder the elements.

Check for understanding: Suppose we have computed d_i, the depth within the tree of each node. The root has $d_i = 1$, its children have $d_i = 2$, and so on. What is the expected lookup cost of this tree?

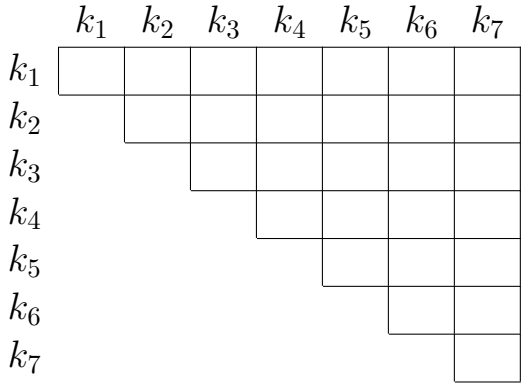
Creating the Dynamic Programming Algorithm

Let's compute OPT(i, j), which is going to be the *cost* of the optimal binary search tree consisting of keys i through j (inclusive). If this is called with $j < i$, we consider this a null tree and return 0 (treat this as a base case).

- Which key(s) can be the root of a binary search tree consisting of keys i through j?

- Suppose key r is the root. What is the cost of the search tree, rooted at r, consisting of keys i through j? You may assume that the left- and right- subtrees of r are constructed optimally.

Let's use that information to create a dynamic programming algorithm. When we're done, we will use that information to construct the tree itself.



The dynamic programming table before any are filled in. Any spaces that will remain unused are not pictured.

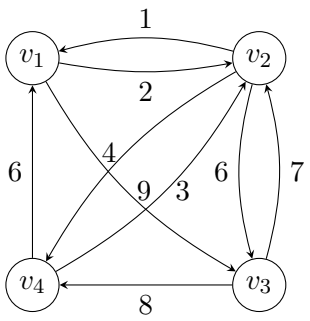
	k_1	k_2	k_3	k_4	k_5	k_6	k_7
k_1	0.13	0.47	0.69	0.72	1.28	1.52	2.12
k_2		0.21	0.43	0.46	1	1.17	1.73
k_3			0.11	0.13	0.47	0.63	1.19
k_4				0.01	0.24	0.4	0.95
k_5					0.22	0.38	0.92
k_6						0.08	0.4
k_7							0.24

The dynamic programming table after the program finishes with the sample input.

Traveling Salesperson Problem

Consider the Traveling Salesperson problem. We are given a simple (not necessarily complete) directed graph. Our goal is to find the Hamiltonian Cycle of lowest total weight.

Example:



Tour	Length
v_1, v_2, v_3, v_4, v_1	22
v_1, v_3, v_2, v_4, v_1	26
v_1, v_3, v_4, v_2, v_1	21

The last tour listed is the optimal one for this input.

A dynamic programming algorithm

In general, we could enumerate every possible tour in time $\mathcal{O}(n!)$, although this is probably a poor idea. Use dynamic programming to produce a better algorithm for this problem. You should not expect to get a polynomial running time for this problem.

Instead, let's use dynamic programming, *allowing* for super-polynomial running time.

Hint 1: Without loss of generality, every tour "begins" and "ends" at v_1. Every such cycle can be thought of as going from v_1 to some v_j, going through zero or more vertices along the way, and then returning to v_1 going through the remaining vertices.

Hint 2: Any subset of vertices can be represented with a bit vector of n bits. The ith bit corresponds to whether or not v_i is included. However, you should focus on the concept, and instead treat this as if your algorithm can take a parameter of a subset of vertices. The detail of how to represent the set is important when you implement the algorithm.