CSCI 570: Dynamic Programming

1. This is a classic Dynamic Programming problem called PRINTING NEATLY - Consider the problem of neatly printing a paragraph on a printer using a monospace font. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the end of the lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer.
2. Here is another classic Dynamic Programming problem called KNAPSACK. We are given a list of n items; item i has a positive integer w_i weight and a value v_i. You can take a total of W (some positive integer) weight worth of items. Give a dynamic programming algorithm to determine the most valuable subset of the items such that the total weight (of the items in the subset) is at most W.

Dynamic Programming: 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.

- General recursive solution:
- Base Case(s):
- In what order do you memoize the recursive solutions?
- What is the running time of your algorithm?
- How would you modify it to output the subsequence itself?

Example: What is the LCS of the sequences < B D C A B A > and < A B C B D A B > ?

		B	D	C	A	B	A
A							
B							
C							
B							
D							
A							
B							

Dynamic Programming: Offline Optimal Binary Search Trees

In your data structures course, you saw unbalanced binary search trees. These had $O(\log n)$ lookup time under some conditions, but $O(n)$ 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 at most $O(\log n)$ 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.

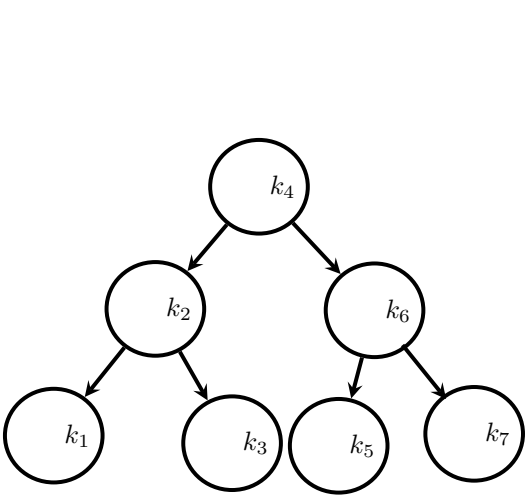
Suppose, though, that before building a binary search tree, we knew exactly which elements were going to be in the tree. If we're likely to look up any given one with equal probability, or if we don't know the likelihood of looking up any given element, we can balance the tree by placing the median element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

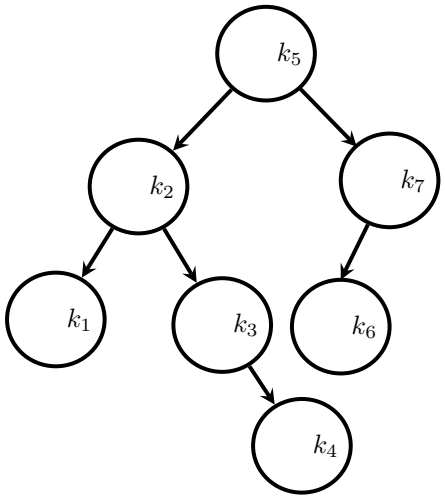
Here's an example with $n = 7$ keys:

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?

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:

- General recursive solution:

- Base Case(s):

- In what order do you memoize the recursive solutions?

- What is the running time of your algorithm?

- How would you modify it to construct the tree itself?