CS 170 Algorithms
Fall 2014 David Wagner

HW 8

Due Oct. 31, 6:00pm

Instructions. This homework is due Friday, October 31st, at 6:00pm electronically. Same rules as for prior homeworks. See `http://www-inst.cs.berkeley.edu/~cs170/fa14/hws/instruct.pdf` for the required format for writing up algorithms.

The hint for all problems on this homework is: dynamic programming.

1. (20 pts.) Subsequence
Design an efficient algorithm to check whether the string $A[1..n]$ is a subsequence of $B[1..m]$, for $n \leq m$. The running time of your solution should be at most $O(nm)$.

Note: the elements in a subsequence do not need to be consecutive (subsequence $\neq$ substring).

2. (20 pts.) Another scheduling problem
You're running a massive physical simulation, which can only be run on a supercomputer. You've got access to two (identical) supercomputers, but unfortunately you have a fairly low priority on these machines, so you can only get time slots on them when they'd otherwise be idle. You've been given information about how much idle computing power is available on each supercomputer for each of the next n one-hour time slots: you can get a_i seconds of computation done on supercomputer A in the ith hour if your job is running on A at that point, or b_i seconds of computation if it running on supercomputer B at that point.

During each hour your job can be scheduled on only one of the two supercomputers. You can move your job from one supercomputer to another at any point, but it takes an hour to transfer the accumulated data between supercomputers before your job can begin running on the new supercomputer, so a one-hour time slot will be wasted where you make no progress.

So you need to come up with a schedule, where for each one-hour time slot your job either runs on supercomputer A, runs on supercomputer B, or "moves" (it switches which supercomputer it will use in the next time slot). If your job is running on supercomputer A for the $i-1$th hour, then for the ith hour your only two options are to continue running on A or to "move." The value of a schedule is the total number of seconds of computation that you get done during the n hours.

You want to find a schedule of maximal value. Design an efficient algorithm to find the value of the optimal schedule, given $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$.

Example: Suppose $n = 4$ and the inputs are given by

i	1	2	3	4
a_i	10	1	5	10
b_i	5	2	3	15

Then the optimal schedule is A, "move", B, B. Its value is $10 + 0 + 3 + 15 = 28$.

3. (20 pts.) Park tours

You're trying to arrange a tour of a national park. The park can be represented as an undirected graph with positive lengths on all of the edges, where the length of an edge represents the time it takes to travel along that edge. You have a sequence of m attractions which you would like to visit. Each attraction is a vertex in the graph. There may be vertices in the graph that are not attractions.

Due to time constraints, you can only visit k of these attractions, but you still want to visit them in the same order as originally specified. What is the minimum travel time you incur?

Design an efficient algorithm for the following problem:

Input: a graph $G = (V, E)$, with lengths $\ell : E \to \mathbb{R}_{>0}$, attractions $a_1, \ldots, a_m \in V$, and $k \in \mathbb{N}$

Output: The length of the shortest path that visits k of the attractions, in the specified order

In other words, we want to find a subsequence of $a_1, \ldots, a_m$ that contains k attractions, and find a path that visits each of those k attractions in that order—while minimizing the total length of that path. More precisely, we want to find the length of the shortest path of the form $a_{i_1} \rightsquigarrow a_{i_2} \rightsquigarrow \cdots \rightsquigarrow a_{i_k}$, where the indices $i_1, \ldots, i_k$ can be freely chosen as long as they satisfy $1 \le i_1 < i_2 < \cdots < i_k \le m$. You don't need to output the path or the subsequence—just the length is enough.

Clarification: Suppose the shortest path from a_1 to a_2 goes through some other attraction, say a_5. That's OK. If the subsequence starts $a_1, a_2, \ldots$, it is permissible for the path from a_1 to a_2 to go through a_5 along the way to a_2, if this is the shortest way to get from a_1 to a_2 (this doesn't violate the ordering requirement, and a_5 doesn't count towards the k attractions).

Revised 10/28 to clarify the problem statement and add the latter 2 paragraphs.

4. (20 pts.) Optimal binary search trees

Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a binary search tree, so that the keyword in the root is alphabetically bigger than all keywords in the left subtree and smaller than all keywords in the right subtree (and this holds for all nodes).

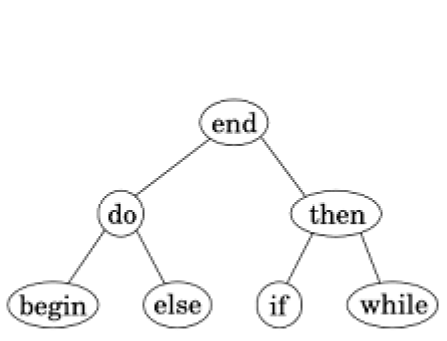
Figure 1 has a nicely-balanced example on the left. In this case, when a keyword is being looked up (for compilation perhaps), the number of comparisons needed is at most three; for instance, in finding "while", only the three nodes "end", "then", and "while" get examined. But since we know the frequency with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word, For the search tree on the left, it is

$$cost = 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) = 2.42$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.



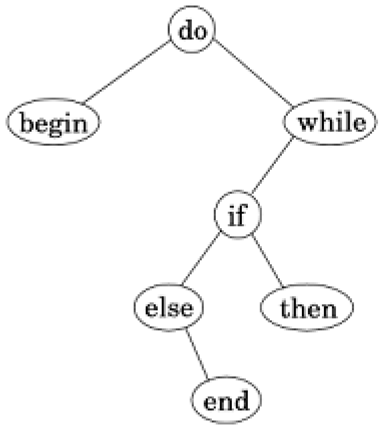


Figure 1: Two binary search trees for the keywords of a programming language

Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

5. **(20 pts.) Beat inference**
   We have an audio track for some song, represented as an array $A[1..n]$, where $A[i]$ represents the loudness at the ith sample. We know the tempo of the song: the time between beats is approximately d samples.

   Our goal is to infer the times at which drum beats occur. Of course, human musicians are not perfect at following the tempo; sometimes the number of samples between two beats will a bit above d, sometimes a bit below d, but it'll generally be close. We do know that the samples when a drum beat occurs tend to be louder than average.

   Based on this, we define a cost function

$$\text{cost}(t_1,\dots,t_m) = \sum_{i=2}^{m}(t_i - t_{i-1} - d)^2 - \sum_{i=1}^{m} A[t_i].$$

   Experiments suggest that if we find the indices $t_1,\dots,t_m$ that minimize $\text{cost}(t_1,\dots,t_m)$, then those represent the times at which the drum beats occurred.

   Design an efficient algorithm for the following problem:

   Input: $A[1..n]$, d, and m.

   Output: $t_1,\dots,t_m$ that minimize $\text{cost}(t_1,\dots,t_m)$, subject to the requirement that $1 \le t_1 \le t_2 \le \cdots \le t_m \le n$.

   The running time of your algorithm should be at most $O(n^2 m)$.

6. **(5 pts.) Optional bonus problem: Image re-sizing**
   (This is an *optional* bonus challenge problem. Only solve it if you want more practice.)

   In this problem you will explore an application of dynamic programming to automatic re-sizing of images. We are given a (greyscale) image $I[1..m][1..n]$ with m rows of pixels and n columns; $I[i,j]$ contains the intensity (greyscale level) of the pixel in the ith row and jth column. We want to shrink this to an image with m rows and $n-1$ columns, i.e., one column narrower.

   We will do this by deleting a *tear* from I. A *tear* is a sequence of pixels that follows a path from the top of the image to the bottom of the image. More precisely, a tear T is a sequence of pixels $T = ((1,x_1),(2,x_2),\dots,(m,x_m))$, such that: (1) for each i, $1 \le x_i \le n$; and, (2) for each i, $|x_{i+1} - x_i| \le 1$. To delete a tear, we delete each pixel in the tear from the image I, so removing a single tear shrinks a $m \times n$-pixel image to a $m \times (n-1)$-pixel image.

We'd like to choose a tear whose removal will be least noticeable to the human eye. Intuitively, one reasonable idea is to choose pixels whose intensity is similar to that of their neighbors (avoiding removal of sharp edges or other kinds of intricate detail). With this idea in mind, we will consider the cost of deleting pixel $I[i,j]$ to be

$$\text{cost}(i,j) = \big|I[i,j-1] - I[i,j]\big| + \big|I[i,j] - I[i,j+1]\big|,$$

and the cost of a tear $T = ((1,x_1),\ldots,(m,x_m))$ to be

$$\text{cost}(T) = \text{cost}(1,x_1) + \cdots + \text{cost}(m,x_m).$$

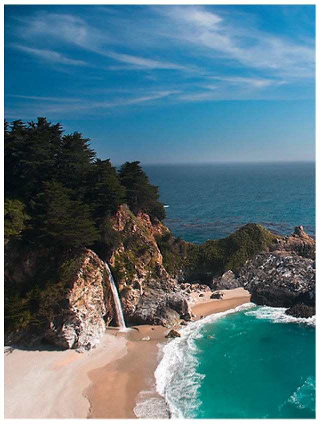
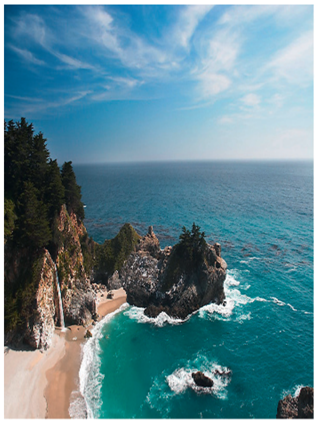
Design an efficient algorithm to find a minimal-cost tear, given the image $I[1..m,1..n]$. Your algorithm should run in $O(nm)$ time.

Comment: This problem introduces you to a powerful technique that was developed by graphics researchers in 2007, for automatic image re-sizing.

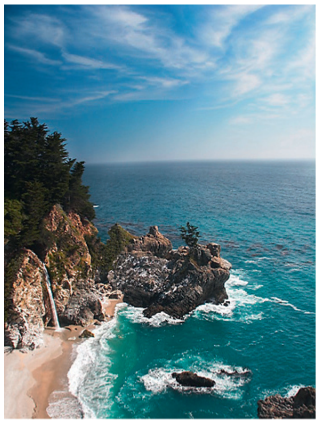
Suppose we have a wide image, and we want to display it on a device with a narrower aspect ratio. We could use "letter-boxing", shrinking the image equally in both the horizontal and vertical dimension; however, letter-boxing leaves black bars at the top and bottom, so you're not getting the benefit of the full size of the display device. We could re-scale the image in just the horizontal dimension, but that changes the aspect ratio and might make people look skinny or introduce other artifacts. We could crop the image, but that might delete important parts of the image. None of these options seem entirely satisfactory. The researchers developed a way to reduce the image width by automatically identifying "less important" parts of the image and deleting them, in a way that avoids the artifacts of the other approaches.

Here is an example. Consider the following original image:

Here it is cropped (left) and re-scaled (right):

Notice how the cropped image omits some important parts of the scene, and the re-scaled version makes everything look too tall and skinny. The researchers' basic approach is to repeatedly find a minimal-cost tear and delete it, until the image is shrunk down to an acceptable aspect ratio. Here is the result of applying their technique:



Notice how this mostly preserves the look of the objects in the scene and retains the most important parts of the scene ("smooshing together" background scenery as needed to re-size the image). The researchers developed a number of powerful extensions, including automatic re-sizing of video, and removing of selected objects from photographic images. The technology has now been integrated into Adobe Photoshop, under the name "content-aware image resizing." Pretty amazing stuff.

Image credits: A. Shamir, S. Avidan.