Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

1. [20 points] Consider the following algorithm for doing a postorder traversal of a binary tree with root vertex $root$.

Algorithm 1 Postorder Traversal

```
function POSTORDER(root)
    if root ≠ null then
        POSTORDER(root.left)
        POSTORDER(root.right)
        visit root
    end if
end function
```

Prove that this algorithm run in time $\Theta(n)$ when the input is an n-vertex binary tree.

The easiest way to prove that this algorithm runs in O(n) time is to understand what the algorithm is doing. Given a node, the algorithm checks the left and right child to see if the child exists on either side. The algorithm then recursively checks the children of the left and right child until it finds a null node.

Given that each node in the tree has a parent that is also in the tree, this algorithm will visit each of the n nodes in the tree, as well as find all of the null nodes. By definition, a binary tree with O(n) nodes can have no more than O(n) null nodes. Thus, the running time of the algorithm is O(n) + O(n) = O(n).

Also, by definition, a postorder traversal processes all nodes of a tree by recursively processing all subtrees, then finally processing the root. Thus, since there are n verticies in the graph and "processing" each (or visiting) takes O(1) time, the whole postorder traversal must take O(n) time.

2. [20 points] We define an AVL binary search tree to be a tree with the binary search tree property where, for each node in the tree, the height of its children differs by no more than 1. For this problem, assume we have a team of biologists that keep information about DNA sequences in an AVL binary search tree using the specific weight (an integer) of the structure as the key. The biologists routinely ask questions of the type, "Are there any structures in the tree with specific weight between a and b, inclusive?" and they hope to get an answer as soon as possible. Design an efficient algorithm that, given integers a and b, returns true if there exists a key x in the tree such that $a \leq x \leq b$, and false if no such key exists in the tree. Describe your algorithm in pseudocode and English. What is the time complexity of your algorithm? Explain.

We first note that AVL trees are nothing more than self-balancing binary search trees. Thus we can assume that traversal from the root node to a leaf node will take no more than O(lgn) time.

Using this, what we are going to do is traverse the tree, checking each node we pass as we go. If the node has a value between a and b, we can return true because that is the criteria. If the value is less than a, we know a value between a and b cannot exist in the left subtree since all values in the left subtree are of value less than a as well. Thus we can check the right subtree. Likewise, if the value is greater than b, we know a value between a and b cannot exist in the right subtree since all values in the right subtree are of value greater than b as well. In this case we check the left subtree.

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

We can do this iteratively until we either reach a value between a and b (in which case we return true) or we reach a root node. If we reach a root node we know that there is no value between a and b in the AVL tree, and thus we return false. Each check at a node takes O(1) time, and as we discussed earlier, traversing an AVL tree takes O(lgn) time, thus the algorithm described takes O(lgn) time.

psuedocode:

```
item_exist_between(a, b, root):          // give values for a, b, and the tree root
  node = root                            // initialize the active node to root
  while node != null:                    // iterate until we hit a leaf node
    if node.value >= a && node.value <= b:   // if value is between a and b return true
      return true
    else if node.value < a:              // if value less than a, check right subtree
      node = node.right
    else:                                // if value greater than b, check left subtree
      node = node. left
  return false                           // we hit a leaf node, return false
```

3. Suppose you are consulting for a company that manufactures computer equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected supplysi of equipment (measured in pounds) that has to be shipped by an air freight carrier. Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges at a fixed rate r per pound (so it costs rsi to ship a week's supply si).
- Company B makes contracts for a fixed amount c per week, independent of weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A schedule for the computer company is a choice of air freight company (A or B) for each of the n weeks with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks in time. The cost of the schedule is the total amount paid to companies A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values s1, ..., sn and returns a schedule of minimum cost. For example, suppose r = 1, c = 10, and the sequence of values is

11, 9, 9, 12, 11, 12, 12, 9, 9, 11.

Then the optimal schedule would be to chose company A for the first three weeks, company B for the next block of four contiguous weeks, and then company A for the final three weeks.

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

In an optimal solution, we wither use company A or B for the ith week. For week i, if we choose company A, we pay r*si and behave optimally from week i-1 to week i. If we choose company b, then we pay 4*c, and we behave optimally from week i-4 to week i.

Along this line, we can set up a recurrence such that we minimize cost for the given schedule. For the first three weeks, we have the trivial recurrence formula:

$$Opt(0) = 0$$
$$Opt(1) = s_1 r$$
$$Opt(2) = (s_1 + s_2) r$$
$$Opt(3) = (s_1 + s_2 + s_3) r$$

Where s_n is the amount to be shipped each week, r is the fixed rate per unit weight charged by company A, and Opt(n) designates the optimal cost through that week.

On the fourth week, however, we have a choice whether to book company A or company B for the four week period. This recurrence can be written generally as:

$$Opt(i) = \min (Opt(i-1) + s_i r, Opt(i-4) + 4c)$$

Using this recurrence we can iterate through each value in the list and apply our recurrence formula. The recurrence formula will only have to compare two values, the cost of choosing company A for the week or the cost of choosing company B for the previous 4 weeks. By using dynamic programming and storing the optimized costs for the previous weeks, we can do this in linear time. See the pseudocode below:

```
def create_schedule(weights, r, c):                 // input weight schedule, and r and c from the prob.
 if len(weights) < 4:                               // check if there are < 4 weights -> trivial solution
  return numpy.fill(len(weights), 'A'))             // return a schedule of all company A

 opt_results: []                                    // we use a list of tuples to store the optimized results
 opt_results.append ((weights[0]*r, [A]))           // store cost and schedule for each optimized result
 opt_results.append((opt_results[0][0] + weights[1]*r, [A]))
 opt_results.append((opt_results[1][0] + weights[2]*r, [A]))

 for week_num, w in weights[3:]:                    // iterate over each of the weights after the first 3
  cost_a = w*r + opt_results[week_num-1][0]         // get the total cost if we choose company A this week
  cost_b = 4*c + opt_results[week_num-4][0]         // get the total cost if we choose company B this week
  if cost_a <= cost_b:
   opt_results.append((cost_a, [A]))                // if company A is cheaper add that cost to opt_results
  else:
   opt_results.append((cost_b, [B, B, B, B]))       // if company B is cheaper add that cost to opt_results
```

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

```
 // this is the end of the for loop
 schedule = []                                  // we have to reconstruct the schedule from opt_results
 index = len(weights)-1                         // start at the back of the opt_results and reconstruct
 while index > 0:
   schedule_items = opt_results[index][1]       // get the optimum choice for the index in opt_results
   schedule.prepend(schedule_items)             // put optimum choice at the beginning of the schedule
   index -= len(schedule_items)                 // decrement index

 return schedule                                // this gives schedule [A, A, B, B, B, B, …] as requested
```

This algorithm is split into 4 parts. The first part checks whether the list of weights is less than 4 items long. Given less than 4 weeks of weights the trivial choice is a schedule with just company A ([4, 6, 5] -> [A, A, A]). This can be done in O(1) time since it is a finite number of operations.

The second part prepopulates the first three values of opt_results. Note that opt_results is a list of tuples, with the first value in the tuple being the optimized cost through that week in the schedule. The second value of the tuple is either [A], designating that you should use company A for the week, or [B, B, B, B] for the 4 week period directly before the current week. While it might be tempting to store the full schedule in the second value of the tuple, this will result in exponential growth in memory space, which would be fine for small problems, but could be problematic for larger problems. This part of the algorithm can be done in O(1) time, since it is a finite number of operations.

The third part of the algorithm iterates over the weights not in the first three weeks. We use our recurrence relation above to determine if we should use company A or B. The values of the tuple are the same as in the second part of the algorithm. There is a finite number of operations for each iteration in the for loop, of which there are n-3 iterations, thus this part takes O(n) time, where n is the number of weights to consider.

The fourth part of the algorithm is how we get the schedule. The final value in opt_results has the total cost, but the full schedule is not yet stored. We start at the back of the list and move until we pass week 0. If we encounter [A], we prepend that to the schedule and move back 1 index in opt_results. If we encounter [B, B, B, B], we prepend that to the schedule and move back 4 indexes in opt_results. When we get to a negative index, we know that the schedule has been built. Since there are finite operations in each iteration of the while loop, and the while loop will execute n times worst case, the runtime of part 4 is O(n) time.

The total runtime of this algorithm is O(1) + O(1) + O(n) + O(n) = O(n), which fits the problem specification of a polynomial time algorithm.

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

[20 points] Collaborative Problem: As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the word segmentation problem–inferring likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like “meetateight” and deciding that the best segmentation is “meet at eight” (and not “me et at eight” or “meet ate ight” or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative “quality” of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters x = x1x2...xk, we return a number quality(x). This number can either be positive or negative; larger numbers correspond to more plausible English words. (So quality(“me00) would be positive while quality(“ight00) would be negative.)

Given a long string of letters y = y1y2...yn, a segmentation of y is a partition of its letters into contiguous blocks of letters, each block corresponding to a word in the segmentation. The total quality of a segmentation is determined by adding up the qualities of each of its blocks. (So we would need to get the right answer above provided that quality(“meet”)+quality(“at”)+quality(“eight”) was greater than the total quality of any other segmentation of the string.) Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. Prove the correctness of your algorithm and analyze its time complexity.

We start by examining a string with two letters. For a string with two letters, there are two possible segmentations of the string, one with no spaces, and one with a space in the middle of the two letters. We would like to select the segmentation scheme which maximizes our quality function. Written mathematically, we have the recurrence relation for a string with two letters written as:

$$Opt(2) = \max\{\ quality(s[0]) + \ quality(s[1]), quality(s[0{:}1])\ \}$$

Where s is our string of length 2 (with the same indexing rules as in python, e.g. s[a:b] is the substring in s from index a to index b), quality() is the black box function given in the problem statement, and Opt(2) is the value given by quality() for the optimal segmentation for a string of length 2.

We also note that of course we can replace quality(s[0]) with Opt(1) since the optimal segmentation for a one character string will be the quality() of that one character string. This will be useful later. Rewriting the previous relation gives us:

$$Opt(2) = \max\{\ Opt(1) + \ quality(s[1]), quality(s[0{:}1])\ \}$$

Let’s move on to a string with three characters. For a string with three characters, we note that there are four possibilities for the segmentation. Given the string “cat”, we can segment the following ways:

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

c a t
ca t
c at
cat

In the same order as our segmentations above, let's write our recurrence relation for Opt(3):

$$Opt(3) = \max\{ \\ quality(s[0]) + quality(s[1]) + quality(s[2]), \\ quality(s[0:1]) + quality(s[2]), \\ quality(s[0]) + quality(s[1:2]), \\ quality(s[0:2]) \\ \}$$

We can combine the first two terms, since Opt(2) is already the maximum of the first two characters. Then substituting Opt(1) for quality(s[0]) in the third term, we can rewrite this as:

$$Opt(3) = \max\{ Opt(2) + quality(s[2]), Opt(1) + quality(s[1:2]), quality(s[0:2])$$

Now we are beginning to see a pattern. Given n as the length of the string and i as each integer such that 0≤i<n

$$Opt(n) = \max_{0 \le i < n} \{ \mathrm{Opt(i)} + \mathrm{quality(s[i:n])} \}$$

Note: this assumes that Opt(0) evaluates to 0, which makes some sense given that Opt(0) would be the quality() value of an empty string.

For this recurrence, a string of length n will have n terms to maximize. We see this in the relations that we already wrote out, where Opt(1) was one value, Opt(2) was the maximum of two terms, and Opt(3) was simplified to the maximum of three terms, the obvious prerequisite to this being that we already solved Opt(n-1). Assuming that solving for each term takes O(1) time, then solving Opt(1) will take O(1) time, Opt(2) will take O(1)+O(2) time, Opt(3) will take O(1)+O(2)+O(3) time, etc. We see that this is a polynomial progression of O(n^2).

But perhaps assuming that solving for each term is not a great assumption. Each term takes the form:

Opt(i) + quality(s[i:n])

O(1) is actually probably a pretty good assumption for the Opt(i) term, since this should just be a table lookup with our dynamic programming approach. However, we don't know the runtime of the quality() function call since it is given as a black box in the problem statement. But it might be unwise to assume this is an O(1) call, as it probably related to the length of the string it is analyzing. Since quality() never calls a string longer than size n, we can place an upper bound on the runtime as O(quality(n)) for each term. Since there are O(n^2) terms, as discussed before, the runtime for our algorithm would have an

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

upper bound of O((n^2)*O(quality(n))), which assuming O(quality(n)) is polynomial, gives us a polynomial runtime as discussed in the problem statement.

Thus we should be able to design an algorithm that can complete this task in O((n^2)*O(quality(n))) time! Let's write some pseudocode:

```
def segment(string):
  segmentations = []                                  // store segmented strings in array
  for index, letter in enumerate(string):             // iterate over each letter in the string
    max_quality = -inf                                // set max quality to a low number
    for i in range(index):                            // iterate over each value in Opt(n)
      opt_i = segmentations[i][0]                     // access the quality value from recursion
      quality_substring = quality(string[i:index])    // calculate quality substring
      quality = opt_i + quality_substring             // combining terms
      if quality > max_quality:                       // check if this is the new max quality val
        segmentations[index] = (quality, segmentation)  // save quality and segmentation at index
        max_quality = quality                         // set the new value of max_quality
    // end for loop
  //end for loop
  return segmentations[index][1]                      // index at end of string; return segmentation
```

This algorithm is relatively simple. The algorithm starts by creating an array to store all of our dynamic programming values called segmentations. Each index of the array will store a tuple containing the quality value of the segmentation and the segmentation itself.

Next we iterate over each letter in the string, building the best segmentation for each increasingly large substring. Within the inner for() loop we calculate the quality of each term of the form Opt(i) + quality(s[i:n]), and assess if it is the maximum quality() value that we have seen for this index. If so we store the quality value and the segmentation in the aforementioned segmentations[] array. The outer for() loop iterates over each letter in the string and builds our segmentation[] array at each index in the string.

Once we get to the last letter of the string, it is trivial to return the segmented string. Since index is still the last index in the string we can reuse that value, and to get the segmentation we access index 1 of the tuple.

This algorithm is correct because each iterative call increases the size of the analyzed string by one so the base case will always be reached. The process repeats n times because of the 1 to n for loop, but each time the recursion will terminate because of the increase in length of the analyzed string by 1 with each call. Observing that each iterative call returns the best segmentation of its input (by the invariant)

and the final iterative call contains the entire original string, we conclude that segment() will return the best segmentation of the input and that the algorithm is correct.

This algorithm follows the general process described when creating the recurrence relation, thus can be completed in a maximum of O((n^2)*O(quality(n))) time. Assuming that O(quality(n)) is polynomial, then our algorithm runtime is polynomial as well.

5. [20 points] Suppose you are acting as a consultant for the Port Authority of a small Pacific Rim nation. They are currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port. Here is a basic sort of problem they face.

A ship arrives with n containers of weight w1, w2, ..., wn. Standing on the deck is a set of trucks, each of which can hold K units of weight. (You may assume that K and wi are integers.) You can stack multiple containers in each truck, subject to the weight restrictions of K. The goal is to minimize the number of trucks that are needed to carry all the containers. This problem is NP-complete.

A greedy algorithm you might use for this is the following. Start with an empty truck and begin piling containers 1,2,3,... onto it until you get to a container that would overflow the weight limit. (These containers might not be sorted by weight.) Now declare this truck “loaded” and send it off. Then continue the process with a fresh truck. By considering trucks one at a time, this algorithm may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

(a) [10 points] Give an example of a set of weights and a value for K where this algorithm does not use the minimum number of trucks.

(b) [10 points] Show that the number of trucks used by this algorithm is within a factor of two of the minimum possible number for any set of weights and any value of K.

a) Assume a value K = 10 and a set of weights [1, 4, 6, 5, 4]

Using the greedy algorithm described, we can pile the following containers on trucks 1..n:

1: [1, 4] (the third container with a weight of 6 would overweigh the truck with max weight 10)
2: [6] (the fourth container with a weight of 5 would overweigh the truck)
3: [5, 4] (the final two containers could fit on the third truck

Whereas the optimal solution would be two trucks with container weights as follows:

1: [1, 4, 5]
2: [6, 4]

This is an example of where a set of weights and a value K where the algorithm does not use the minimum number of trucks.

Brian Loughran
Johns Hopkins
Algorithms
4/1/2020

b) Note that the total weight (Wt) is the sum of all of the container weights. In a perfect world, given a maximum truck load of K, we could unload the cargo in Wt/K trips (given that the container weights cooperate). This will provide us with a minimum bound for the number of trucks T that need to be used. Stated algebraically:

$$T \geq \frac{Wt}{K}$$

Next we consider the load of two consecutive trucks produced by the greedy algorithm. We note that by definition, the sum of the capacity of two consecutive trucks must exceed K, otherwise they would both be loaded onto the same truck. In pairing off each truck, it is obvious that at least one of each pair is at least half filled. It is possible (in the case that we have an odd number of trucks) that the last truck will also be less than half filled. Given a number of trucks T, we know that at least (T-1/2) are more than half filled. Thus we know that:

$$\frac{T-1}{2} * K \geq Wt$$

Rearranging we get the number of trucks used by the algorithm to be :

$$T - 1 \leq \frac{2 * Wt}{K}$$

Comparing the Wt/K terms in the two inequalities makes clear that the maximum number of trucks used by the greedy algorithm is within a factor of 2 of the optimal solution.