CSC 373 - Algorithm Design, Analysis, and Complexity **Summer 2014**

Assignment 1: Solutions

Please follow the instructions provided on the course website to submit your assignment. You may submit the assignments in pairs. Also, if you use **any** sources (textbooks, online notes, friends) please cite them for your own safety.
You can use those data-structures and algorithms discussed in CSC263 (e.g. merge-sort, heaps, etc.) and in the lectures by stating their name. You do not need to provide any explanation or pseudo-code for their implementation. You can also use their running time without proving them: for example, if you are using the merge-sort in your algorithm you can simply state that merge-sorts running time is $\mathcal{O}(n \log n)$.

Every time you are asked to design an efficient algorithm, you should provide both a short high level explanation of how your algorithms works in plain English, and the pseudo-code of your algorithm similar to what we've seen in class. State the running time of your algorithm with a brief argument supporting your claim. You must prove that your algorithm finds an optimal solution!

1 ~~Lo~~Lallapalooza!

It's summer! It's music fests season! This year at Lallapalooza, they're giving a free trip (everything included) to the 2015 Lallapalooza in ...SPACE! $^{\text{what?}}$ The winner will be drawn at random. To get tickets for the draw, you must attend performances.

Lallapalooza[1] is a music festival for emerging artists. In order to encourage people to attend their performance, artists give free tickets for the draw. Your goal is of course to collect as many tickets as possible.

Each performance lasts h hours, and you get v tickets if you attend. However, if you leave during a performance, you get a fraction of those tickets. For instance, suppose a performance last 5 hours and hands in 10 tickets in total. If you attend 3 hours only of the performance, you collect $3 * 10/5 = 6$ tickets.

There are k stages set up. You're given a schedule of all the performances ahead of time. Each performance P_i has (s_i, h_i, v_i), the start time, the duration and the profit where $s_i, h_i, v_i \in \mathbb{N}$. You can assume that the fraction you get (if any) is always an integer. You can also assume that moving from one stage to another takes an insignificant amount of time[2]. Devise an efficient algorithm that maximizes the number of tickets you will put into the draw. Give a high-level description of how your algorithm works, state the running time of your algorithm and prove that it is optimal.

Input : A list of n performances $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ where each P_i is represented with the tuple (s_i, h_i, v_i).
Output : The maximum possible number of tickets for the draw.

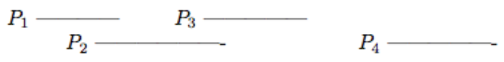
Solution

We use a greedy algorithm to solve this problem as follows: For every performance P_i, we compute its density $\tilde{v}_i = \frac{v_i}{h_i}$. We then "flat" sort the start and finish time of the performances in nondecreasing order; and greedily select the performance with the highest density at every step.

[1] One day, it'll be big...
[2] Yay, teleportation

Let's first consider an example, then formalize the algorithm.

Suppose we have the following input: $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$ and $\tilde{v}_1 = 2, \tilde{v}_2 = 3, \tilde{v}_3 = 4, \tilde{v}_4 = 2$



The "flat" sort gives us:

$$s_1 < s_2 < f_1 < s_3 < f_2 < f_3 < s_4 < f_4$$

Where s_i and f_i represent the start and finish time of every performance P_i. We will construct an AVL tree, with a max pointer as follows: Scan the flat list. Every time you process an element (s_i or f_i), check where the max pointer is in the tree:

- If the max is null, push a node with the density of that element to the tree.
- If the max is not null, compute the time difference between the current element you're processing and the previous one, and add to your solutions the number of tickets of that time difference * the largest density in the tree (i.e., *max*).
- If the element you're processing is a finish time, i.e. f_i, update the number of tickets if necessary and remove any corresponding node from the tree.

The proper balancing is performed after every insetion and deletion. Let S be our solution. And we need a *last* variable to store the last *time* we processed.

We process the flat list one element at a time:

- s_1 first, max is null, so you push a node into the AVL tree with value $\tilde{v}_1 = 2$ and max points at this node. Update $last = s_1$
- Then you process s_2, max is not null. You update S to add $(s_2 - last)$ * max = $(s_2 - last)$*$\tilde{v}_1$. Then you push $\tilde{v}_2$ into the tree. Since $\tilde{v}_2 > \tilde{v}_1$, you update max to point to $\tilde{v}_2$. Update $last = s_2$.
- Then you process f_1. You update S to add $(f_1 - last)$ * max = $(f_1 - last)$ * $\tilde{v}_2$. Since we hit f_1, we remove $\tilde{v}_1$ from the tree. max still points to $\tilde{v}_2$ at this time. Update $last = f_1$.
- Then you process s_3. You update S to add $(s_3 - last)$*max = $(s_3 - last)$*$\tilde{v}_2$. Then you push $\tilde{v}_3$ into the tree. Since $\tilde{v}_3 > \tilde{v}_2$, you update max to point to $\tilde{v}_3$. Update $last = s_3$.
- Then you process f_2. You update S to add $(f_2 - last)$*max = $(f_2 - last)$*$\tilde{v}_3$. Since we hit f_2, we remove $\tilde{v}_2$ from the tree. max still points to $\tilde{v}_3$. Update $last = f_2$.
- Then you process f_3. You update S to add $(f_3 - last)$*max = $(f_3 - last)$*$\tilde{v}_3$. Since we hit f_3, we remove $\tilde{v}_3$ from the tree. max is null, update *last* to 0. (When max is null, we reset *last* to 0).
- Then you process s_4. max is null. so you push a node into the tree for $\tilde{v}_4$ and max points to it. Update $last = s_4$.
- Then you process f_4. You update S to add $(f_4 - last)$*max = $(f_4 - last)$*$\tilde{v}_4$. Since we hit f_4. we remove $\tilde{v}_4$ from the tree and max is null again and we update *last* to 0.

Then we're done. Few important key points: we need the AVL tree to keep the tree balanced. This way inserts and deletes takes $\mathcal{O}\log(n)$. The flat sorting is on a list of $2n$ elements and thus takes $\mathcal{O}(n\log n)$ time. We loop $2n$ times (2 ends points for each interval) in order to process the flat sorted list. Therefore the run time is $\mathcal{O}(n\log n)$.

The optimality proof is a simple contradiction. Suppose there exists an optimal solution O such that $S < O$ and let i be the first iteration in the algorithm where we computed a different number (a lower number) of tickets to add to our solution. Let P_j be the performance whose tickets we gained (by the algorithm) at iteration i, and P_k the performance O selected. This implies that *max* was pointing to the node with density $\tilde{v}_j$, but $\tilde{v}_k > \tilde{v}_j$ which contradicts how *max* is computed. Thus at step i, *max* was also pointing at $\tilde{v}_k$.

Algorithm 1 Max-Tickets

```
 1: Initialize $\mathcal{T}$ the AVL tree, $max$, $last$ and $S$
 2: Sort the endpoints of $\mathcal{P}$ in nondecreasing order.   ▷ Let $\mathcal{L} = \{a_1, a_2, ..., a_{2n}\}$ denote the flat sorted list.
 3: for i=1 ... n do
 4:     $\tilde{v}_i \leftarrow \frac{v_i}{h_i}$
 5: end for
 6: for i=1 ... 2n do
 7:     if $a_i$ is an $s_j$ for some $P_j$ then
 8:         Push $\tilde{v}_j$ into $\mathcal{T}$ and balance $\mathcal{T}$.
 9:         if $max$ is null then
10:             $max = \tilde{v}_j$ and $last = a_i$.
11:         else
12:             $S = S + (a_i - last) * max$
13:             if $\tilde{v}_j > max$ then
14:                 $max = \tilde{v}_j$
15:             end if
16:             $last = a_i$
17:         end if
18:     else $a_i$ is an $f_j$ for some $P_j$
19:         $S = S + (a_i - last) * max$
20:         Remove $\tilde{v}_j$ from $\mathcal{T}$ and balance $\mathcal{T}$.
21:         Update $max$ if necessary                ▷ In case $max$ was the node we just deleted
22:         if $max$ is null then
23:             $last = 0$
24:         end if
25:     end if
26: end for
27: return $S$
```

2 Ew Mud!

One of the reasons Lallapalooza is not very popular is because of its location in MudVille. We have a mud problem in MudVille. When running for mayor, you promised the people that if they elect you, you will pave the roads so they don't have to walk through mud anymore! AND tourists won't be so reluctant to come to Lallapalooza anymore. The girls are happy they don't have to walk with their heels in the mud! The business owners are happy people will make it to their stores more often! And old people are happy they won't drown in mud anymore! You won the elections! But when you got to office, you realized you have a low budget. As an honest *cough* politician, you decided to keep your promise (sort of) and pave the roads that connect only the main destinations (subway stations, banks, malls, Lallapalooza plaza, etc...). As long as the MudVillers can get from one main destination to another they're happy!

The city council gave you a map of Mudvillle with a list of n locations that need to be reached. In addition, you also have access to a list of all the potential roads that connect these locations, plus the cost of paving each road. Clearly, there could be more than one way to get from one location to another. Paving a road will cost c dollars, where c is always a positive integer.

Input : A list of k roads to be paved $\{r_1, r_2, ..., r_k\}$, a list of k costs $\{c_1, c_2, ..., c_k\}$ where c_i is the price you pay to pave road r_i. A map $\mathcal{M}$ of n locations $\{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n\}$ where each r_i connects two locations $\mathcal{L}_a, \mathcal{L}_b$ in $\mathcal{M}$.
Output : The total minimum cost to pave the roads to connect the n locations.

Devise an algorithm to solve this problem efficiently. Give a high-level description of how your algorithm works, state the time complexity of your algorithm, and prove that your algorithm is optimal.

Solution

We reduce this problem to an instance of minimum spanning tree. We constrcut an edge-weighted graph $G(V, E, w)$ where $V = \{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n\}$, $E = \{r_1, r_2, ..., r_k\}$ and for every $r_i \in E, w(r_i) = c_i$. Constructing the graph takes $\mathcal{O}(n + k)$.

It is easy to see that a minimum spanning tree of G will indeed produce the lowest cost to connect all n vertices, i.e., n locations. We can choose to run either Prim's or Kruskal's algorithm to compute the MST. If we choose Kruskal's algorithm, we get an overall running time of $\mathcal{O}(k \log n)$.

3 Let The Paving Begin!

You've selected the roads to pave, and chose a company to do the job: CementGiants. CementGiants doesn't pave roads the regular way, they purchase gigantic cement blocks that they cut if necessary and lay them down on the road. You can assume that each road is a straight line of length l_i. The cement blocks have a fixed length $\mathcal{CL}$, and $l_i < \mathcal{CL}$ for all i.

There are $j \leq k$ roads to pave. Once you give your list of road lengths $\mathcal{RL} = \{l_1, l_2, ..., l_j\}$ to CementGiants, they lay the blocks in the order of this list. That is, road r_a with length l_a is paved iff all the roads r_b for $b \leq a$ have already been paved.

Since $l_i < \mathcal{CL}$, every time they cut a block there is always a remaining unused piece of length $\mathcal{CL} - l_i$. CementGiants' goal is **not** to minimize the total number of remaining pieces, nor the number of blocks used in total. Their goal is to minimize the length difference between these remaining pieces. In other words, they want their remaining blocks to be of equal length as much as possible so they can reuse them for some other project. For instance, suppose the list of road lengths is $\mathcal{RL} = \{1, 3, 2, 2\}$ and $\mathcal{CL} = 5$:

- You could either use one block for each road, your remaining pieces would be of length 4, 2, 3, 3 respectively.
- You could use one block for the first road, one for the 2nd and 3rd, and one more block for the last road with remaining pieces of length 4, 0, 3 respectively.
- You could use the first block to cover both the first and last road (total of 3), and use a second block to cover the second and third roads (total of 5). Although this minimizes the number of blocks used

(2), it is not a valid operation since we are changing the order in which the roads are paved.

- The best solution would be to use one block for the first two roads and one for the last 2. The remaining blocks are both of length 1 each.

Input : The value $\mathcal{CL}$ and a **fixed** list of j road lengths $\mathcal{RL} = \{l_1, l_2, ..., l_j\}$ where $l_i < \mathcal{CL}$ for all i.
Output : A grouping $(l_1, ..., l_{i_1}), (l_{i_1+1}, ..., l_{i_2}), ..., (l_{i_{s-1}+1}, ..., l_{i_s})$ that minimizes the difference between the remaining blocks, where $l_{i_s} = l_j$

Note that this difference can be expressed as $\sum\limits_{r=0}^{s-1}(\mathcal{CL} - l_{i_r+1} - l_{i_r+2} - ... - l_{i_{r+1}})^2$.
Clearly $l_{i_r+1} + l_{i_r+2} + ... + l_{i_{r+1}} \leq \mathcal{CL}$ for all $0 \leq r \leq s-1$.

Construct an efficient algorithm which solves the above problem. Give a high-level description of how your algorithm works, state the time complexity of your algorithm, and prove that your algorithm is optimal.

Modification
For simplicity, let's use the following formula for minimization: $\sum_{r=0}^{s-1}(S^* - S_r)^2$

Where $S_r = l_{i_r+1} + l_{i_r+2} + ... + l_{i_{r+1}}$ and $S^* = min_r(\mathcal{CL} - S_r)$, $0 \leq r \leq s-1$ (that is, S^* is the S_r that minimizes $(\mathcal{CL} - S_r)$)

Solution
Before attacking this problem using the formula above, let's consider the case when we are minimizing the difference with respect to a fixed length $\mathcal{L}$. Now the input is a fixed ordering $\mathcal{RL} = \{l_1, L-2, ..., l_j\}$ and we want a grouping $(l_1, ..., l_{i_1})(l_{i_{i+1}}, ..., l_{i_2})...(l_{i_{s-1}}, l_{i_s} = l_j)$ that minimizes the difference $\sum\limits_{r=0}^{s-1}(\mathcal{L} - l_{i_r+1} - l_{i_r+2} - ... - l_{i_{r+1}})^2$. We will solve this problem using Dynamic Programming.

We use $M[k]$ to denote the value of the optimal solution to the problem with the first k input. So in general:

$$M[i] = \min_{1 \leq k < j} M[k] + (\mathcal{L} - l_{k+1} - l_{k+2} - ... - l_i)^2 \text{ For } 1 \leq i \leq j \text{ and } (l_{k+1} - l_{k+2} - ... - l_i) \leq \mathcal{L}$$

In other words, we consider every possible last subgroup; and the base case is just $M[0] = 0$ since there are no groupings to consider. We use induction to prove this recursive specification.

Proof. For the base, $M[0] = 0$; and suppose the grouping is optimal for the first $k-1$ groupings. Now consider the k^{th} grouping.
We know that for the first $k-1$ groupings must be optimal, otherwise we can replace them with a better solution. In other words $(l_1, ..., l_{i_1}), (l_{i_1+1}, ..., l_{i_2}), ..., (l_{i_{k-1}+1}, ..., l_{i_k})$ must be optimal, now consider the k^{th} grouping. Since $M[k]$ tries all possible groupings for the last subgroup and select the one that minimizes the error, it follows that it is optimal as well. □

The algorithm below takes a fixed length block $\mathcal{L}$ and a fixed ordering $(l_1, ..., l_n)$ and computes the values of $M[i]$.

Algorithm 2 Fixed-Length

```
1: $M[0] \leftarrow 0$
2: for i=1 ... n do
3:     $M[i] \leftarrow M[i-1] + (\mathcal{L} - l_i)^2$
4:     for k = i -2, i-3, ..., 1 do and while $(l_{k+1} - l_{k+2} - ... - l_i) \leq \mathcal{L}$
5:         if $M[k] + (\mathcal{L} - l_i - ... - l_{k+1})^2 < M[i]$ then
6:             $M[i] \leftarrow M[k] + (\mathcal{L} - l_i - ... - l_{k+1})^2$
7:         end if
8:     end for
9: end for
10: return $M[n]$
```

With the two nested loops, we get a run time of $\mathcal{O}(n^2)$. Now let's go back to our original problem, we are trying to minimize the error with respect to S^* as defined above. But since S^* can vary, we cannot really use the formula for the fixed length $\mathcal{L}$.

So what do we know for sure? We know that whatever S^* we'll end up using, S^* will be a subset of $l_1, ..., l_n$ of **consecutive** l_i's. So we ask the question: How many possible groupings of consecutive l_i's can we have? $\mathcal{O}(n^2)$! Why? Consider the fixed ordering $l_1, l_2, ..., l_n$, we have n choices to place the first bracket (, and at most n choices for the closing bracket). In other words, there are n^2 possible choices of S^*, so we just try them all and select the one that minimizes the error.

Algorithm 3 Variant-Length

```
1: $k \leftarrow 0$
2: for i=1...n do
3:     for j=i...n do
4:         if $(l_i + l_{i+1} + ... + l_j \leq \mathcal{CL})$ then
5:             $S^* = (l_i, l_{i+1}, ..., l_j)$
6:             $E[k] \leftarrow$ Fixed-Length$(S^*, l_1, l_2, ..., l_n)$
7:             $k++$
8:         end if
9:     end for
10: end for
11: return $\min_k E[k]$                ▷ Return the $E[k]$ with with the smallest error
```

The correctness follows from the correctness of the Fixed-Length algorithm, and the fact that we are trying all possible choices for S^*. The two nested loop take $\mathcal{O}(n^2)$ and each iteration takes $\mathcal{O}(n^2)$, therefore the total run time is of $\mathcal{O}(n^4)$.

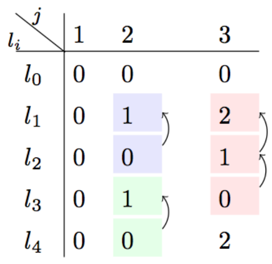
Another way to approach the problem is to construct a table with all possible values of S^* up to $\mathcal{CL}$; and check all possible groupings with respect to these values. This will therefore give us an $(n+1) \times \mathcal{CL}$ table which is not guaranteed to be polynomial in n.

Let T be an $(n+1) \times \mathcal{CL}$ table, where for a given entry $T[i,j]$, i refers to the length of the block l_i, and j refers to the size of S^* (i.e. $S^* = j$). We set $T[0,j] = 0$ and in general:

$$T[i,j] = \min(T[i-1,j] - l_i, j - l_i)$$

Where $T[i-1,j] - l_i = \infty$ if $T[i-1,j] - l_i < 0$, and similarly for $j - l_i$. Notice that the formula above choses the minimum error between (1.) adding l_i to the same grouping as l_{i-1} or (2.) starting a new grouping with l_i at the beginning.

Let's consider the example below, to see how this works. Suppose we have the following input: $\mathcal{CL} = 3$ and $l_1 = l_2 = l_3 = l_4 = 1$. There are two optimal groupings here, either $(l_1, l_2)(l_3, l_4)$ and $(l_1)(l_2)(l_3)(l_4)$. The table is constructed as follows:



l_i \ j	1	2	3
l_0	0	0	0
l_1	0	1	2
l_2	0	0	1
l_3	0	1	0
l_4	0	0	2

Notice that the pointers show how the l_i's are grouped together. Every value of j is a possible value for S^*, we just try all of them up to $\mathcal{CL}$.

To compute the total error for a given S^*, i.e. for a given j, it suffices to sum up the values of the tails of the pointers plus any cell that is left on its own. Meaning, for $j = S^* = 1$, we get an error of $0+0+0+0+0 = 0$. For $j = S^* = 2$, we get $0 + 0 + 0 = 0$, and for $j = S^* = 3$, we get $2 + 0 + 0 = 2$. We use the pointers to reconstruct the groupings.

This is really just trying all possible groupings, for all possible S^*'s. Not an elegant algorithm, and definitely not polynomial in n since $\mathcal{CL} \gg n$ (possibly).

4 Ole Ola!

The World Cup starts in less than a month!!! You're the head of the soccer[3] division at SportsToday and want to send your journalists to cover all the games, but going to Brazil is expensive, and with the World Cup around the corner, all the prices will skyrocket.

The schedule is out! In reality, games don't overlap, but in our 373 world they do. Suppose some matches overlap. You're given the entire schedule ahead of time which includes the start and end time of each match, and want to send as few journalists as possible to cover all the games.

Define formally the input and output of the problem. Devise an efficient algorithm that minimizes the number of journalists you will send to Brazil. Give a high-level description of how your algorithm works, state the running time of your algorithm and prove that it is optimal.

Solution

We reduce this problem to an instance of interval partitioning. Our input is a set $\mathcal{I} = \{I_1, I_2, ..., I_n\}$ of intervals where every interval $I_i = (s_i, f_i)$ has a start and finish time s_i and f_i respectively. Every I_i represents a match.

The interval partitioning problem asks for a partition $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ of the intervals into a minimum

[3]It's called football!

number k of partition classes, such that intervals inside partition class P_i are pairwise nonoverlapping. This problem is also known as Interval Coloring, where the intervals of every P_i are assigned the same color. Every partition class is a color class; and for our problem, every color class (i.e. every subset of nonoverlapping intervals) will represent the list of matches assigned to a journalist.

We will use a greedy algorithm to solve this problem, by first sorting the intervals by nondecreasing early start time, and then greedily assigning every interval I to the first "available" partition class; i.e. to the minimum i such that $P_i \cup \{I\}$ is compatible.

Algorithm 4 Matches Partitioning

Input: A set $\mathcal{I} = \{I_1, I_2, ..., I_n\}$ of n intervals where $I_i = (s_i, f_i)$ and $s_i < f_i$ for all $1 \leq i \leq n$.
Output: A minimum coloring of $\mathcal{I}$.

```
1:  Sort $\mathcal{I}$ by nondecreasing start time: $s_1 \leq s_2 \leq ... \leq s_n$
2:  $k \leftarrow 1$                                        ▷ Minimum number of colors
3:  $P_k \leftarrow \emptyset$                              ▷ The matches to be covered by the $k^{th}$ journalist
4:  for i=1 ... n do
5:      if $I_i$ is compatible with $P_j$ for some $1 \leq j \leq k$ then
6:          color($I_i$) = $\min_{1 \leq j \leq k} (I_i$ is compatible with $P_j)$
7:          $P_j \leftarrow P_j \cup \{I_i\}$
8:      else
9:          $k \leftarrow k + 1$
10:         $P_k \leftarrow \{I_i\}$
11:     end if
12: end for
13: $\mathcal{P} = \{P_1, P_2, ..., P_k\}$
14: return $\mathcal{P}, k$
```

Claim: The algorithm above returnes the minimum number of colors to partition $\mathcal{I}$.

Before proving the above claim, we ask the following question: Given a list $\mathcal{L}$ of intervals, what is the minimum number of colors **necessary** to color $\mathcal{L}$?
It is **at least** the size of $\mathcal{A} \subseteq \mathcal{L}$, the **largest** subset of $\mathcal{L}$ such that all the intervals in $\mathcal{A}$ are pairwise intersecting. That is, if all the intervals in $\mathcal{A}$ intersect, then we need at least $|\mathcal{A}|$ colors to color them.

Proof. We'll prove our claim by contradiction. Let k be the number of colors returned by our algorithm, and let $h < k$ be the minimum number of colors used in an optimal solution O.

Consider I_i, the first interval that used color k (lines 8-10 in the algorithm). By our algorithm, I_i overlaps with a subset of intervals of $\{I_1, I_2, ..., I_{i-1}\} \subset \mathcal{I}$ that uses $k-1$ colors; since otherwise the algorithm would've assigned I_i a color $j < k$ (lines 5-7). Therefore I_i must intersect with at least $k-1$ intervals in $\mathcal{I}$, and thus we need a k^{th} color to color I_i which contradicts the optimality of O with uses $h < k$ colors. □

The run time of the algorithm is dominated by the sorting, which takes $\mathcal{O}(n \log n)$ time, since determining intersection takes linear time. Thus the algorithm runs is $\mathcal{O}(n \log(n))$ time.

5 The Faster The Better

Your journalists in Brazil send you daily predictions of which teams (countries) will end up in the semi final. SportsYesterday is a competitor news channel. They have their journalists in place too and get their daily predictions. The rest of the planet can only find out about what is happening in Brazil through the site predictions.com. All news channels submit their predictions there! The news channel to broadcast the predictions first gets more followers and thus makes more money. You got today's predictions: Brazil, Argentina, Spain and Germany.

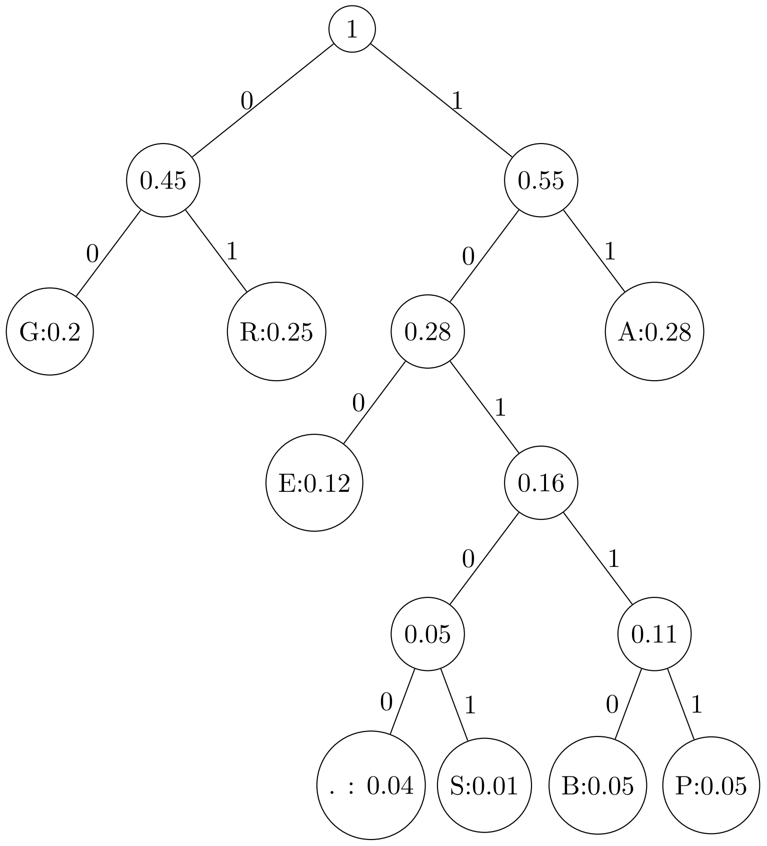
At SportsYesterday, they use fixed length encoding to transfer all their messages, you can do better! You are given the frequencies of the following symbols in the English alphabet and you want to submit the following message: "BRA.ARG.SPA.GER".

A	0.21
B	0.04
E	0.09
G	0.15
P	0.04
R	0.19
S	0.01
.	0.03

Normalize these frequencies and use your probability distribution to produce a Huffman Tree. What would be the encoding of your prediction message "BRA.ARG.SPA.GER"?

Solution
The normalized frequency of a symbol α is : $p(\alpha) = \frac{f(\alpha)}{\sum\limits_{x \in \Sigma} f(x)}$. We produce the following table (the normalized frequencies are rounded):

α	$f(\alpha)$	$p(\alpha)$
A	0.21	0.28
B	0.04	0.05
E	0.09	0.12
G	0.15	0.2
P	0.04	0.05
R	0.19	0.25
S	0.01	0.01
.	0.03	0.04
Total	0.76	1



The encoding of each symbol is (trace the path from the root to the symbol):

A	11
B	10110
E	100
G	00
P	10111
R	01
S	10101
.	10100

The encoding of the string BRA.ARG.SPA.GER is :

10110 01 11 10100 11 01 00 10100 10101 10111 11 10100 00 100 01

I just added space so it's easy to read. Technically, there shouldn't be any.