Exercises, chapter 11, solutions

---

1. Zipf's law predicts that for every $i \in \{1, 2, \ldots, n\}$, the ith most common word in a corpus of n words is $P(i) = \frac{1}{i} \cdot \frac{1}{H_n}$, where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. The value of n is not given explicitly in the problem (and actually, we don't need it here), but we are told that $P(4) = 0.03000049\ldots$ Combining $P(3) = \frac{1}{3} \cdot \frac{1}{H_n}$ and $P(4) = \frac{1}{4} \cdot \frac{1}{H_n}$ gives $P(3) = \frac{1}{3} \cdot 4 \cdot P(4) \approx 0.0400$, rounded to four decimal places.

2. (a) Here is one possible solution:

```
HIRE-ASSISTANT(n)
  best = 0          // candidate 0 is a least-qualified dummy candidate
  for i = 1 to n
      interview candidate i
      if candidate i is better than candidate best
          best = i
          hire candidate i
```

   (b) The probability that exactly one secretary is hired is the probability that the candidate who is interviewed first is the best candidate, which is $\frac{1}{n}$.

   (c) The probability that exactly n secretaries are hired is the probability that the candidates are interviewed in order of increasing qualifications. Out of the $n!$ different orderings of the n candidates, there is one such ordering, so the probability is $\frac{1}{n!}$.

3. It would be better to use an array in this case because it will make the searching process faster. Recall that in each of its iterations, the binary search technique needs to access the middle element in a specified range in order to compare it to the search key. In an array, the element at any given position can be accessed immediately by indexing, but in a (standard) linked list, one needs to start at the first element and follow pointers from one element to the next until the correct position is reached, and this may take a much longer time.

4. To check if a particular position i in A contains a peak, we compare the value of $A[i-1]$ to $A[i]$ and the value of $A[i]$ to $A[i+1]$, which takes $O(1)$ time. To avoid checking all positions in A, proceed as follows.

   - Look at the middle element $A[mid]$; if it's a peak then we are done.
   - Otherwise, if the left neighbor of $A[mid]$ is larger than $A[mid]$ then the left half of A must have a peak and we can recurse on that part of A (and we can ignore the right half).
   - Otherwise, the right neighbor of $A[mid]$ is larger than $A[mid]$, so the right half of A must have a peak and we can recurse on that part of A (and we can ignore the left half).

   The above observations lead to the following algorithm. The initial call is FIND_PEAK$(A, 0, n-1)$.

```
FIND_PEAK(A, low, high)
    mid = low + ⌊(high − low)/2⌋
    if A[mid − 1] ≤ A[mid] and A[mid] ≥ A[mid + 1] then return mid
    elseif A[mid − 1] > A[mid] then return FIND_PEAK(A, low, mid − 1)
    else  // A[mid] < A[mid + 1]
        return FIND_PEAK(A, mid + 1, high)
```

   Time complexity analysis: The same as binary search, i.e., $O(\lg n)$. On each recursion level, we are reducing the search space by making the part of the array that is being searched half as large.

   Remark: More formally, the time complexity of both the binary search algorithm and FIND_PEAK can be expressed by the recurrence $T(n) = T(n/2) + O(1)$, which has the solution $T(n) = O(\lg n)$.