Ruolin Li
31764160

CPSC 320 2020S2: Tutorial 8

In the following two problems, the quantities of interest can be defined inductively, that is, using recurrences. See how these definitions can then be used to define efficient algorithms, using either divide and conquer or memoization/dynamic programming.

1 Word segmentation

Some languages are (or used to be) written without spaces between 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 "meetaeight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight", or "meet ate ight", or any of the huge number of even less plausible alternatives). How could we automate this process?

A simple, reasonably effective approach is to find a segmentation that maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that for any string of one or more letters $X[1..k]$, $k \geq 1$, will return a number $Q(X)$. This number can be either positive or negative; larger numbers correspond to more plausible English words. So Q(me) would be positive, while Q(ght) would be negative.

Given a long string of letters $Y[1..n]$, a segmentation of Y is a partition of its letters into contiguous nonempty blocks of letters; each block corresponds 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 get the right answer for the problem above provided that Q(meet) + Q(at) + Q(eight) was greater than the total quality of any other segmentation of the string.

1. Give a recurrence relation for Max-TQ(i): the maximum total quality of any segmentation of a string of letters $Y[1..i]$ for $i \geq 0$.

2. Use the recurrence to get an efficient algorithm for computing Max-TQ, for an input $Y[1..n]$. Assume that a function for computing $Q(X)$ is available, where X is a string of one or more characters.

Tutorial 8

2020/07/31

Q1.

1) MaxTQ(i)

if i=0

maxTQ(i)=0

if i>=1

maxTQ(i)= max{ Q(Y[1...i]),

max{ MaxTQ(Y[j])+ Q(Y[j+1...i]) for j in range 1 to i }

Algorithm for MaxTQ(Y[1...n]):

Create an array Soln[0...n] and initialize all entries with -∞.

Soln[0]=0

return TQHelper(n).

Algorithm TQHelper(i):

if Soln[i] is -∞:

Soln[i]= Q[Y[1...i]]

for j from 1 to i-1:

X= TQHelper(j) + Q(Y[j+1...i])

if Soln[i]< X:

Soln[i]= X

else

return Soln[i]

Work done at internal node : $\sum_{i=0}^{n} i = \frac{n(n-1)}{2}$ $O(n^2)$

Work done at leaf node: constant * # of leaves = n^2. $Q(n^2)$

2 Exponentiation

Given two non-negative integers x and n and a positive integer N, the exponentiation problem is to compute $x^n \bmod N$. For example, if $x = 3, n = 4$, and $N = 15$ then $x^n = 3^4 = 81$ and $81 \bmod 15 = 6$.

1. Fill in the missing parts to describe $x^n \bmod N$ inductively:

$$x^n \bmod N = \begin{cases} 1 \bmod N, & \text{if } n = 0 \\ \underline{\qquad\qquad}, & \text{if } n = 1 \\ (x^{n/2} \bmod N)^2 \bmod N, & \text{if } n > 1 \text{ is even} \\ \underline{\qquad\qquad\qquad\qquad}, & \text{if } n > 1 \text{ is odd.} \end{cases}$$

2. Using this inductive definition, write a recursive algorithm that computes $x^n \bmod N$. You can assume that multiplication and mod operations on n-bit numbers are available, that we can test if a number is even, and so on.

3. Give a recurrence relation that describes an upper bound on the number of *multiplication* operations done by your algorithm as a function of n.

4. Using Θ notation, write the solution to your recurrence relation from part 3.

Q2.

1. $x^n \bmod N = \begin{cases} 1 \bmod N, & \text{if } n=0 \\ x \bmod N, & \text{if } n=1 \\ (x^{n/2} \bmod N)^2 \bmod N, & \text{if } n>1 \text{ is even} \\ (x^{\frac{n-1}{2}} \bmod N)^2 x \bmod N, & \text{if } n>1 \text{ is odd} \end{cases}$

$$x^n \bmod N$$
$$= x^{n-1} \cdot x \bmod N$$
$$= (x^{\frac{n-1}{2}} \bmod N)^2 x \bmod N$$

2. func:

```
if (n = 0)
    return 1 % N
else if (n=1)
    return x % N
                    half = func(x, n/2, N)
else if (n>1 AND n%2 ==0)
    return (half)^2 mod N          O(1)
else if (n>1 AND n%2 != 0)
    return (half)^2 * x mod N      O(1)
```

3. $T(n) = \begin{cases} c, & n=0 \text{ or } n=1 \\ T(\frac{n}{2}) + 2, & n>1 \end{cases}$

$a = 1$ only call func once during recursion

$b = 2$ half of the size (approximately)

$k = 0$

$a = b^k$ $\Theta(n^k \log n) = \Theta(\log n)$