Recitation 4 - Divide and Conquer, MTF

Sebastian Laudenschlager

sebastian.laudenschlager@colorado.edu

February 9, 2018

Another D&C example

- Suppose we want to multiply two polynomials, e.g. $p(x) = 8x^3 + 7x^2 + 2x + 7$ and $q(x) = 3x^2 + 4x + 5$.
- Standard way?
- $8x^3(q(x)) + 7x^2(q(x)) + 2x(q(x)) + 7q(x)$
- Time complexity?
- Better way $\rightarrow$ Karatsuba's algorithm

Karatsuba

- Clever way of multiplying two polynomials
- Let $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ and $q(x) = b_0 + b_1 x + \ldots + b_n x^n$
- Split $p(x)$ into two parts, so that

$$\begin{aligned} p(x) &= (a_0 + a_1 x + \ldots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}) + (a_{\frac{n}{2}} x^{\frac{n}{2}} + \ldots + a_n x^n) \\ &= p_1(x) + x^{\frac{n}{2}} p_2(x) \end{aligned}$$

- Note that p_1 and p_2 are both polynomials of degree $\frac{n}{2}$.
- Similarly, write $q(x) = q_1(x) + x^{\frac{n}{2}} q_2(x)$.

Karatsuba, cont.

- Now we can write the polynomial multiplication as

$$\begin{aligned} p * q &= (p_1 + x^{\frac{n}{2}} p_2)(q_1 + x^{\frac{n}{2}} q_2) \\ &= (p_1 q_1) + x^{\frac{n}{2}} (p_2 q_1 + p_1 q_2) + x^n (p_2 q_2) \end{aligned}$$

Pseudocode

Pseudocode for multiplying polynomials:

```
def multPoly(p, q, n):
    if n < 1:
        # base case
    (p1, p2) = splitPoly(p)
    (q1, q2) = splitPoly(q)
    r1 = multPoly(p1, q1, n/2) # p1*q1
    r2 = multPoly(p1, q2, n/2) # p1*q2
    r3 = multPoly(p2, q1, n/2) # p2*q1
    r4 = multPoly(p2, q2, n/2) # p2*q2
    r5 = add(r2, r3) # p1*q2 + p2*q1
    r6 = shift(r5, n/2)
    r7 = add(r1, r6)
    r8 = shift(r4, n)
    r9 = add(r7, r8)
    return r9
```

Analysis

- Recursion: Four subproblems of size $\frac{n}{2}$ and $\Theta(n)$ work at each level (splitting, adding, shifting).
- Leads to the following recurrence relation:

$$T(n) = \begin{cases} 4T(\frac{n}{2}) + \Theta(n) & \textit{if} \quad n > 1 \\ \Theta(1) & \textit{if} \quad n \leq 1 \end{cases}$$

- Let's try solving this recurrence via the expansion method.

Improvement

- So this divide and conquer algorithm is no better than the naive method.
- It turns out that a simple change to this algorithm results in much better performance.
- Notice that $p_1 q_2 + p_2 q_1 = (p_1 + p_2)(q_1 + q_2) - p_1 q_1 - p_2 q_2$.
- Thus, in our algorithm pseudocode, instead of computing $r2$ and $r3$, we would instead compute $s = (p_1 + q_1)(p_2 + q_2)$ and then $s - r1 - r4$.
- This reduces the number of recursive subproblems from 4 to 3.
- Impact?
- $\Rightarrow \Theta(n^{\log_2 3}) \approx \Theta(n^{1.5849})$.

Move-To-Front Algorithm

- We are given an alphabet of n symbols $[c_1, c_2, \ldots, c_n]$ and some data to be encoded.
- Goal: encode (rearrange) data to make compression easier.
- Idea:
  - Go through data one element at a time, and for each element encountered, place it at the front of the alphabet.
  - Keep placing elements at the front as they are encountered, until all data has been encoded.

Example

- Let's say we wish to encode the word "arugula" using this method.
- We use the standard alphabet $A = [a, b, c, \ldots, z]$.
- Output will be a list of indices such that we can invert the transformation to retrieve the original data.
  1. a: move to front $\rightarrow A = abcdefghijklmnopqrstuvwxyz$, $output = [0]$
  2. r: MTF $\rightarrow A = rabcdefghijklmnopqstuvwxyz$, $output = [0, 17]$
  3. u: MTF $\rightarrow A = urabcdefghijklmnopqstvwxyz$, $output = [0, 17, 20]$
  4. g: MTF $\rightarrow A = gurabcdefhijklmnopqstvwxyz$, $output = [0, 17, 20, 8]$
  5. u: MTF $\rightarrow A = ugrabcdefhijklmnopqstvwxyz$, $output = [0, 17, 20, 8, 1]$
  6. l: MTF $\rightarrow A = lugrabcdefhijkmnopqstvwxyz$, $output = [0, 17, 20, 8, 1, 13]$
  7. a: MTF $\rightarrow A = alugrbcdefhijkmnopqstvwxyz$, $output = [0, 17, 20, 8, 1, 13, 4]$

Decoding

- Start with same original alphabet.
- Go through the output list and retrieve the element at those indices.
- Move that element to the front just like during the encoding process.

Example, cont.

- $output = [0, 17, 20, 8, 1, 13, 4]$, $A = abcdefghijklmnopqrstuvwxyz$
  1. $A[0] = a \rightarrow A = abcdefghijklmnopqrstuvwxyz$
  2. $A[17] = r \rightarrow A = rabcdefghijklmnopqstuvwxyz$
  3. $A[20] = u \rightarrow A = urabcdefghijklmnopqstvwxyz$
  4. $A[8] = g \rightarrow A = gurabcdefhijklmnopqstvwxyz$
  5. $A[1] = u \rightarrow A = ugrabcdefhijklmnopqstvwxyz$
  6. $A[13] = l \rightarrow A = lugrabcdefhijkmnopqstvwxyz$
  7. $A[4] = a \rightarrow A = alugrbcdefhijkmnopqstvwxyz$
- So we've retrieved our original word.

Why would this be useful?

- It should be clear that after we encode our word, the most frequently used characters should be near the front of the updated alphabet.
- Furthermore, the more frequent a character appears in our data, the more smaller numbers we'll get in the output.
- This should make sense: each time we encounter a character, we move it to the front, thus if we encounter a particular character often, we expect it to be not too far from the from the front, resulting in small output indices.
- How can this help us with compression?
  - Use fewer bits to store characters occurring near the front, which should translate to using fewer bits to store more frequently occurring characters.