Divide and Conquer

Andreas Klappenecker

The Divide and Conquer Paradigm

The divide and conquer paradigm is important general technique for designing algorithms. In general, it follows the steps:

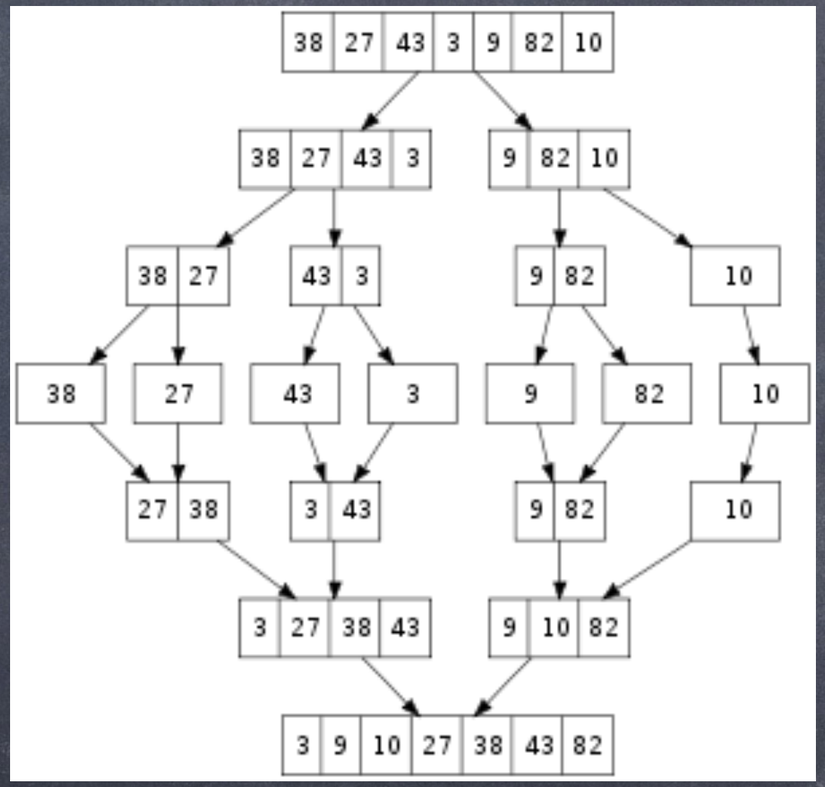
- divide the problem into subproblems
- recursively solve the subproblems
- combine solutions to subproblems to get solution to original problem

Mergesort

Example: Mergesort

- DIVIDE an input sequence of length n into two parts:
  - the initial $\lceil n/2 \rceil$ elements and
  - the final $\lfloor n/2 \rfloor$ elements.
- RECURSIVELY sort the two halves, using sequences with 1 key as a basis of the recursion.
- COMBINE the two sorted subsequences by merging them

Mergesort Example



Example courtesy of wikipedia

Recurrence Relation for Mergesort

Let $T(n)$ be the worst case running time of mergesort on a sequence of n keys

If $n = 1$, then $T(n) = \Theta(1)$ (constant)

If $n > 1$, then

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

Simplified Recurrence Relation

Let $T(n)$ be the worst case running time of mergesort on a sequence of n keys

If $n = 1$, then $T(n) = \Theta(1)$ (constant)

If $n > 1$ and n even, then $T(n) = 2\ T(n/2) + \Theta(n)$

[Indeed, we recursively sort the two subproblems of size $n/2$, and we need $\Theta(n)$ time to do the merge.]

Recurrence Relations

Master Theorem

Consider a recurrence of the form

$$T(n) = a\,T(n/b) + f(n)$$

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)} \log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and $f(n)$ is regular, then $T(n) = \Theta(f(n))$

[$f(n)$ regular iff $af(n/b) \leq cf(n)$ for some constant $c<1$ and all but finitely many n]

Roughly speaking...

Essentially, the Master theorem compares the function f(n) with the function $g(n)=n^{\log_b(a)}$.

Roughly, the theorem says:

a) If $f(n) << g(n)$ then $T(n)=\Theta(g(n))$.

b) If $f(n) \approx g(n)$ then $T(n)=\Theta(g(n)\log(n))$

c) If $f(n) >> g(n)$ then $T(n)=\Theta(f(n))$.

Now go back and memorize the theorem!

Nothing is perfect...

The Master theorem does not cover all possible cases. For example, if

$f(n) = \Theta(n^{\log_b(a)} \log n)$,

then we lie between cases 2) and 3), but the theorem does not apply.

There exist better versions of the Master theorem that cover more cases, but these are harder to memorize.

Idea of the Proof

Let us iteratively substitute the recurrence:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + f(n/b)) + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= \dots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

Idea of the Proof

Thus, we obtained

$$T(n) = n^{\log_b(a)}\, T(1) + \sum a^i\, f(n/b^i)$$

The proof proceeds by distinguishing three cases:

1) The first term in dominant: $f(n) = O(n^{\log_b(a)-\varepsilon})$

2) Each part of the summation is equally dominant: $f(n) = \Theta(n^{\log_b(a)})$

3) The summation can be bounded by a geometric series: $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and the regularity of f is key to make the argument work.

Fast Integer Multiplication

Integer Multiplication

Elementary school algorithm (in binary)

```
      101001 = 41
    x 101010 = 42
--------------------
      1010010
    1010010
+ 1010010
--------------------
11010111010 = 1722
```

Scan second number from right to left. Whenever you see a 1, add the first number to the result shifted by the appropriate number of bits.

Integer Multiplication

- The multiplication of two n bits numbers takes $\Omega(n^2)$ time using the elementary school algorithm.
- Can we do better?
- Kolmogorov conjectured in one of his seminars that one cannot, but was proved wrong by Karatsuba.

Divide and Conquer

- Let's split the two integers X and Y into two parts: their most significant part and their least significant part.
- $X = 2^{n/2}A + B$ (where A and B are n/2 bit integers)
- $Y = 2^{n/2}C + D$ (where C and D are n/2 bit integers)
- $XY = 2^{n}AC + 2^{n/2}BC + 2^{n/2}AD + BD$.

How Did We Do?

- Multiplication by 2^x can be done in hardware with very low cost (just a shift).
- We can apply this algorithm recursively:
- We replaced one multiplication of n bits numbers by four multiplications of n/2 bits numbers and 3 shifts and 3 additions
- $T(n) = 4T(n/2) + cn$

Solve the Recurrence

- $T(n) = 4T(n/2) + cn$
- By the Master theorem, $g(n) = n^{\log_2(4)} = n^2$.
- Since $cn = O(n^{2-\text{epsilon}})$, we can conclude that $T(n) = \Theta(n^2)$.

How can we do better?

- Suppose that we are able to reduce the number of multiplications from 4 to 3, allowing for more additions and shifts (but still a constant number).
- Then $T(n) = 3T(n/2) + dn$
- Master theorem: $dn = O(n^{\log2(3)-\text{epsilon}})$, so we get $T(n) = O(n^{\log2(3)}) = O(n^{1.585})$.

Karatsuba's Idea

Let's rewrite

$XY = 2^n AC + 2^{n/2} BC + 2^{n/2} AD + BD$

in the form

$XY = (2^n - 2^{n/2})AC + 2^{n/2}(A+B)(C+D) + (1-2^{n/2})BD.$

Done! Wow!

Karatsuba in Base 2

Split input X into two parts A and B such that

$X = 2^{n/2}A + B.$

Split input Y into two parts C and D such that

$Y = 2^{n/2}C + D.$

Then calculate AC, (A+B)(C+D), BD.

Copy and shift the results, and add/subtract:

$XY = (2^n - 2^{n/2})AC + 2^{n/2}(A+B)(C+D) + (1-2^{n/2})BD.$

```
def karatsuba(x,y): # variation, now for decimal numbers, base 10
    if len(str(x)) == 1 or len(str(y)) == 1:
        return x*y
    d = max(len(str(x)), len(str(y))) // 2      # ≈half the number of digits
    x_msd, x_lsd = divmod(x, 10**d)             # chop x into ≈d digit chunks
    y_msd, y_lsd = divmod(y, 10**d)             # chop y into ≈d digit chunks
    z_lsd = karatsuba(x_lsd, y_lsd) # multiplication 1
    z_msd = karatsuba(x_msd, y_msd) # multiplication 2
    z_mix = karatsuba( x_msd + x_lsd, y_msd + y_lsd ) # multiplication 3
    return z_msd * 10**(2*d) + (z_mix - z_msd - z_lsd)* 10**d + z_lsd
```

```
Give the value of x = 1234
Give the value of x = 5678
karatsuba(1234,5678)
  karatsuba(34,78)   # Multiplication 1
    karatsuba(4,8)       # Multiplication 1-1
    karatsuba(3,7)       # Multiplication 1-2
    karatsuba(7,15)     # Multiplication 1-3
  karatsuba(12,56) # Multiplication 2
    karatsuba(2,6)       # Multiplication 2-1
    karatsuba(1,5)       # Multiplication 2-2
    karatsuba(3,11)     # Multiplication 2-3
  karatsuba(46,134) # Multiplication 3
    karatsuba(6,4)       # Multiplication 3-1
    karatsuba(4,13)     # Multiplication 3-2
    karatsuba(10,17)   # Multiplication 3-3
      karatsuba(0,7)       # Multiplication 3-3-1
      karatsuba(1,1)       # Multiplication 3-3-2
      karatsuba(1,8)       # Multiplication 3-3-3
The product x*y =  7006652 (recursive Karatsuba)
Compare to  x*y =  7006652 (built in mult)
```