More Asymptotic Notation. Analysis of recursive algorithms.

Abhiram Ranade

January 7, 2016

Summary of last time

Developed a framework for analysing algorithms

- "Time taken by an algorithm" = "Time taken by an algorithm in the worst case as a function of problem size"
- Consider algorithm to be executing on RAM. Pay attentional to the functional form of the expression of the time taken, e.g. whether linear or quadratic etc. in problem size.
- General goal: Try to express time taken by an algorithm as $\theta(\ldots)$.
- θ includes upper bound as well as lower bound.

Used the framework to analyze matrix multiplication algorithm and algorithm for merging.

Outline for today

- Additional asymptotic notation
- Analysis of recursive algorithms
- Divide and conquer algorithms

Additional asymptotic notation

$f = \theta(g)$: "f is between multiples of g for large n"

f is the same as g to within constant factors

f is bounded below and above by g.

Sometimes we may only be able to prove a bound above, or sometimes only a bound below.

In such cases we cannot say $f = \theta(g)$.

We need some different notation.

θ, O, Ω notations

$f = \theta(g)$: there exist real constants $c_1, c_2, n_0 > 0$ such that for all $n \geq n_0$ we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.

$f = O(g)$: there exist real constants $c_2, n_0 > 0$ such that for all $n \geq n_0$ we have $0 \leq f(n) \leq c_2 g(n)$.

$f = \Omega(g)$: there exist real constants $c_1, n_0 > 0$ such that for all $n \geq n_0$ we have $0 \leq c_1 g(n) \leq f(n)$.

$f = \theta(g)$ iff $f = O(g)$ and $f = \Omega(g)$.
$f = O(g)$ iff $g = \Omega(f)$

Intuitively:

- θ means $=$
- O means $\leq$
- Ω means $\geq$

upto constant factors, asymptotically.

Examples

$n^2 + 16n + 23 = O(n^2)$

$n^2 + 16n + 23 = O(n^3)$
$\because\ n^2 + 16n + 23 \leq (1 + 16 + 23)n^2 \leq 40n^3$ for all n.

$n^2 + 16n + 23 \notin \Omega(n^3)$
Proof: Suppose contrary.
For some c, all $n \geq n_0$ we have $cn^3 \leq n^2 + 16n + 23 \leq 40n^2$
Thus $n \leq 40/c$. So not true for all $n \geq n_0$

Simplifying inside θ, O, Ω

$$O(n^2 + 2n) = O(n^2)$$

Just as for θ

Analysis of recursive algorithms

```
mergesort(x[1..n]){
  if n = 1 return x
  else return merge(mergesort(x[1..n/2]),
                    mergesort(x[n/2+1..n]))
}   // assume n even
```

General analysis strategy:

Let $T(n) =$ worst case time for size n.
If the base case is for size n_0, we know $T(n_0)$ directly.

$T(1) \leq c_1$

Relate $T(n)$ to $T(n'), T(n''), \ldots$ assuming function is recursively called on problems of size $n', n'', \ldots$.

$T(n) = 2T(n/2)+$ merge time.

$T(n) \leq 2T(n/2) + c_2 n$ for some c_2 and $n \geq n_0$.

Try to solve for $T(n)$.

Analysis of mergesort

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + c_2 n \\
&\leq 2(2T(n/4) + c_2 n/2) + c_2 n && \text{Assume } n \text{ power of 2.} \\
&= 4T(n/4) + 2c_2 n \\
&\leq 4(2T(n/8) + c_2 n/4) + 2c_2 n \\
&= 8T(n/8) + 3c_2 n \\
&\leq 2^k T(n/2^k) + kc_2 n \\
&= nT(1) + \log n \cdot c_2 n && \text{Choose } k = \log n. \\
&\leq c_1 n + c_2 n \log n \\
&\leq \max(c_1 + c_2) n \log n \\
&= O(n \log n)
\end{aligned}
$$

By similar analysis we can show $T(n) = \Omega(n \log n)$.
Thus $T(n) = \theta(n \log n)$

Remarks

- The idiom: break the input into two parts, recurse on the parts, and then combine the results of the parts appears very often.
  This kind of recursion is called Divide-and-conquer.
- We assumed $n = 2^k$ keys were being sorted.
  Acceptable because: More laborious analysis using floors and ceilings will get us the same result.
  Also, we can always introduce dummy keys to make $n = 2^k$.
  This will not increase the time by a constant factor, which we are ignoring anyway.
- Idea of substituting the recurrence into itself works often.
- Exercise: Solve $T(n) \leq 3T(n/2) + cn$, $T(1) \leq k$

Another D&C example: integer multiplication

Primary school algorithm for multiplying n bit integers: $\theta(n^2)$ time.

can be improved!

Recursive view:

Inputs: n bit integers A, B.

$A_1 = k$ lsbs of A. $A_2 =$ rest. Similarly B_1, B_2

$A \cdot B = (A_2 2^k + A_1)(B_2 2^k + B_1)$
$= A_2 B_2 2^{2k} + (A_1 B_2 + B_1 A_2) 2^k + A_1 B_1$.

Recurse for multiplications $A_2 B_2, A_1 B_1, \ldots$

Multiplication by 2^k: shift

Choice $k = 1$: $T(n) = T(n-1) +$ Addition time $\leq T(n-1) + cn$.

Thus $T(n) = O(n^2)$.

Choice $k = n/2$: $T(n) \leq 4T(n/2) + cn$

Solution of $T(n) \leq 4T(n/2) + cn$

$$
\begin{aligned}
T(n) &\leq 4T(n/2) + cn \\
&\leq 4(4T(n/4) + cn/2) + cn \\
&= 16T(n/4) + cn(1+2) \\
&\leq 16(4T(n/8) + cn/4) + cn(1+2) \\
&= 64T(n/8) + cn(1+2+4) \\
&\leq 4^k T(n/2^k) + cn(2^k - 1) \qquad n = 2^k \\
&\leq n^2 c' + cn(n-1)
\end{aligned}
$$

Thus $T(n) = O(n^2)$.

But we can do better!

A faster algorithm

Our basic relationship: $A \cdot B = (A_2 2^k + A_1)(B_2 2^k + b_1)$
$= A_2 B_2 2^{2k} + (A_1 B_2 + B_1 A_2) 2^k + A_1 B_1.$
$= A_2 B_2 2^n + (A_1 B_2 + B_1 A_2) 2^{n/2} + A_1 B_1$ For k=n/2

Algorithm:

1. Compute $P_1 = (A_1 + A_2)(B_1 + B_2)$, $P_2 = A_1 B_1$, $P_3 = A_2 B_2$
2. $A \cdot B = P_3 2^n + (P_1 - P_2 - P_3) 2^{n/2} + P_2$

$T(n) \leq 2T(n/2) + T(n/2+1) + T_{add}$ $P_3 :\ n+1$ bit inputs
$T(n/2+1) \leq T(n/2) + c_1 n,\ T_{add} \leq c_2 n$
$\leq 3T(n/2) + cn$
$\leq 3(3T(n/4) + cn/2) + cn = 3^2 T(n/2^2) + cn(1+3/2)$
$\leq 3^k T(n/2^k) + \sum_{i=0}^{i=k-1} (3/2)^i cn$ $n = 2^k$
$\leq 3^{\log_2 n} T(1) + cn \frac{(3/2)^{\log_2 n} - 1}{3/2 - 1}$
$\leq 3^{\log_2 n} T(1) + 2c(3)^{\log_2 n}$

Both terms are $O(3^{\log_2 n}) = O(n^{\log_2 3})$.
$T(n) = O(n^{\log_2 3}) = O(n^{1.58})$
Similarly $T(n) = \Omega(n^{\log_2 3})$.

Recursion trees: a graphical method for solving recurrences

1. Draw the tree representing the recursive calls made by the algorithm. Root = original call.
2. Figure out number of levels.
3. Estimate the work at each level of the tree.
4. Add up the work.

Example: Mergesort.

- Number of levels = $\log n$, where n = size of entire sequence.
- Work at internal nodes $\leq cm$ where m = size of the subsequence generated.
- Total work at any level $\leq cn$
- There are n leaves, and hence total work at the leaves is also $c'n$.

Thus total time $\leq cn \log n + cn = O(n \log n)$.

Unequal size subproblems

Suppose just for fun we recurse on sequences of size $n/3$ and $2n/3$.

What changes in the analysis?

The total work at each level is still $\leq cn$, because total number of keys at each level is $\leq cn$.

The number of levels? Some branches terminate early, others go deep.

The rightmost leaf would be deepest, with number of levels $= \log_{3/2} n$.

Max depth $= \log_{3/2} n$.

Time $\leq cn \log_{3/2} n$

Graphical analysis is easier than algebraic?

Equal sized subproblems better.

Floors and ceilings

If number of keys being sorted is not a power of 2, we will encounter floors and ceilings in the analysis.

The subproblems will not be exactly equal.
Larger problem will have size $\lceil n/2 \rceil$.
Reduction at each step $= \lceil n/2 \rceil / n \leq 2/3$

Equality for $n = 3$.

Thus depth of tree $\leq \log_{3/2} n$.

Work at each level $\leq cn$.

Thus total time $= O(n \log n)$

Note that $\log_a n = \log_2 n / \log_2 a = \theta(\log_2 n)$

Base of the logarithm does not matter if we are ignoring constant factors and the base is a constant.

Master Theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0, b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof: Omitted.

Covers many cases that may arise in practice. Gives insight.

You may memorize this and use it, unless you are explicitly asked not to.

Exercise: Solve $T(n) \leq 7T(n/7) + cn$.

Remarks

- Need to understand carefully which constants can be ignored and which cannot.
  - If the time taken for something is cn, usually the precise value of c does not matter.
  - If something creates c subproblems, the the value of c matters very much!
- Dividing subproblems equally is often more efficient.

Remarks

- Even faster algorithms are possible for integer multiplication.
- Sometimes we may write $T(n) + \theta(n)$ and so on.
  In general $f + \theta(g)$ is to be interpreted a $\{f + g' | g' \in \theta(g)\}$.
- Questions you should ask about a given algorithm
  1. What is an upper bound on the time taken for the worst case? (Answer expected as $O()$).
  2. What is a lower bound on the time taken for the worst case? (Answer expected as $\Omega()$).
  3. Have we found the best possible algorithm?
     This last question is very hard to answer...