Design & Analysis of Algorithms

The Big O

Lecture 20
Upper-bounds: Big O

- $T(n)$ has an upper-bound that grows “like” $f(n)$
  
  \[ T(n) = O(f(n)) \]

  \[ \exists c, k > 0, \forall n \geq k, \ 0 \leq T(n) \leq c \cdot f(n) \]

- $T(n) = \Theta(f(n))$ if $T(n)=O(f(n))$ and $f(n)=O(T(n))$
Recursion

Given an array \( L \), find max among numbers between positions \( \text{start} \) and \( \text{end} \) (inclusive)

\[
\text{findmax} (L, \text{start}, \text{end}) \{
    \text{if} (\text{start} == \text{end}) \text{ return } L[\text{start}]
    \text{else} \{
        \text{mid} = \left \lfloor \frac{(\text{start}+\text{end})}{2} \right \rfloor
        \text{ x = findmax}(L,\text{start},\text{mid})
        \text{ y = findmax}(L,\text{mid}+1,\text{end})
        \text{ if } (x>y) \text{ return } x
        \text{ else return } y
    \}
\}
\]

e.g. \text{findmax}(L,1,6)

Correctness by strong induction:
Induct on the size of the problem.
i.e., the length of the list, \( n = |\text{end}-\text{start}+1| \)

How about the running time?
Recursion

Given an array \( L \), find max among numbers between positions \( \text{start} \) and \( \text{end} \) (inclusive)

\[
\text{findmax}(L, \text{start}, \text{end}) \ {\}
\ 
\text{if (start == end)}
\ 
\text{return } L[\text{start}]
\ 
\text{else} \ {\}
\ 
\text{mid } = \lfloor (\text{start}+\text{end})/2 \rfloor
\ 
x = \text{findmax}(L, \text{start}, \text{mid})
\ 
y = \text{findmax}(L, \text{mid}+1, \text{end})
\ 
\text{if } (x>y) \text{ return } x
\ 
\text{else return } y
\ 
\}
\}

e.g. \text{findmax}(L, 1, 6)

Recursion structure:
A full binary rooted tree with \( n \) leaves
(Not important that the split was into almost equal parts)
Recursion

Given an array $L$, find max among numbers between positions $start$ and $end$ (inclusive)

\[
\text{findmax} \ (L, \ start, \ end) \ {\{ \\
\text{if} \ (start == end) \ \\
\quad \text{return} \ \ L[start] \\
\text{else} \ {\{ \\
\quad \text{mid} = \ \lfloor (start+end)/2 \rfloor \\
\quad x = \ \text{findmax} \ (L,\ start,\ mid) \\
\quad y = \ \text{findmax} \ (L,\ mid+1,\ end) \\
\quad \text{if} \ (x>y) \ \text{return} \ x \\
\quad \text{else} \ \text{return} \ y \\
\}}}
\]

Time $T(n)$ taken by $\text{findmax}(L,a,a+n-1)$?

Recursion tree: $c_1$ on each leaf and $c_2$ on each internal node

\[
T(n) = T( \lfloor n/2 \rfloor ) + T( \lceil n/2 \rceil ) + c_2
\]

$T(n) = O(\text{number of nodes})$

$T(n) = O(n)$
Question

Time taken by `find3max(L, a, a+n)` is

A. $\Theta(n)$
B. $\Theta(n \log n)$
C. $\Theta(n^{3/2})$
D. $\Theta(n^3)$
E. None of the above

```
find3max(L, st, en) {
    if (st == en)
        return L[st]
    else {
        mid1 = st + \lfloor (en-st+1)/3 \rfloor
        mid2 = st + 2*\lfloor (en-st+1)/3 \rfloor
        x = find3max(L, st, mid1)
        y = find3max(L, mid1+1, mid2)
        z = find3max(L, mid2+1, en)
        if (x \geq y \land x \geq z) return x
        if (y \geq x \land y \geq z) return y
        if (z \geq x \land z \geq y) return z
    }
}
```

$T(n) = \Theta(\#\text{nodes in a full ternary rooted tree with } n \text{ leaves}) = \Theta(n)$
Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list

$T(n) = 2T(n/2) + \text{time to merge}$
Merging Two Sorted Lists

Maintain the invariant that a list K has a prefix of the final merged list. X₁, X₂ have the rest of L₁, L₂.
- Base case: K=empty, X₁=L₁, X₂=L₂
- Inductively, move the smaller of first(X₁) and first(X₂) to the end of K
- Terminating condition: Both X₁ and X₂ are empty

Time taken (as a function of n = |L₁|+|L₂| )?

- When finished K has n elements
- Each element gets added to K exactly once
- Each iteration adds exactly one element to K (in O(1) time)
- T(n) = O(n)

merge (L₁, L₂ : ascending lists) {
    K = empty-list; X₁ = L₁; X₂ = L₂;
    while (X₁ not empty or X₂ not empty) {
        if (X₂ empty)
            x = pop(X₁)
        else if (X₁ empty)
            x = pop(X₂)
        else if ( first(X₁) ≤ first(X₂) )
            x = pop(X₁)
        else
            x = pop(X₂)
        append(K,x)
    }
    return K
}
Merge Sort

- Sorting by divide-and-conquer
- Split the list into two (unless a single element)
- Sort each list recursively
- Merge the sorted lists into a single sorted list

$$T(n) = 2T(n/2) + \text{time to merge}$$

$$T(n) = 2T(n/2) + c \cdot n$$

- Contribution from each level: $O(n)$
- Depth of recursion: $O(\log n)$
- $T(n) = O(n \log n)$
Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
- Objects can be compared with each other (using a total ordering)

Simple idea:
- Check if desired object = middle one in the list
- If not, comparing with the middle one lets you see if it could be in the left half or the right half of the list (since the list is sorted)
- Recursively search in that half

Depth of recursion, for an n element list ≤ \( \lceil \log_2 n \rceil \)
Zeroing in on the answer by shrinking the range by half each time

Traversing an implicit binary tree

Nodes contain the mid-elements of the range under them

At each node compare the desired object with the object at the node
**Binary Search**

- Alternate use: to approximately find a root of a **continuous** function
  - Needs two points $x_1, x_2$, st. $f(x_1) \leq 0$ and $f(x_2) \geq 0$
  - Can maintain this invariant, while shrinking $|x_1-x_2|$ exponentially
  - Continuous $\rightarrow$ this interval will have a root
  - May miss some 0s if function is not monotonous, but will find some other

- Contrast with finding a 0 in an array of values $[f(1),f(2),...,f(n)]$ (no continuity!)
  - If array not sorted, we may miss a 0, and there may not be another one!

- Faster methods exploit value/slope (not just sign)
Binary Search

Example: finding (up to required precision) the square root of a number \(n>1\) (using only comparison and multiplication)

Initial range: \([0,n]\) (say)

How to compare \(\sqrt{n}\) with middle element \(m\)?

- compare \(n\) and \(m^2\)
A General Solution
(a.k.a. “Master Theorem”)

\[ T(n) = a \cdot T(n/b) + c \cdot n^d \] (and \( T(1)=1 \).\n\( a \geq 1, b > 1 \) integer, \( c > 0, d \geq 0 \) real.)

Say \( n = b^k \) (so only integers encountered)

\[ \text{#levels} = \log_b n = k \]

\[ T(n) = O( n^d \left( 1 + \left( \frac{a}{b^d} \right) + \ldots + \left( \frac{a}{b^d} \right)^k \right) ) \]

If \( a = b^d \), contribution at each level = \( n^d \). \( T(n) = O(n^d \cdot \log n) \)

If \( a < b^d \): \( 1 + \left( \frac{a}{b^d} \right) + \left( \frac{a}{b^d} \right)^2 + \ldots = O(1) \). \( T(n) = O(n^d) \)

If \( a > b^d \): \( \left( \frac{a}{b^d} \right)^k [1 + \left( \frac{b^d}{a} \right) + \left( \frac{b^d}{a} \right)^2 + \ldots ] = O((a/b^d)^k) = a^k / n^d \)

\[ T(n) = O(a^k) = O(2^{k \cdot \log a}) = O(2^{\log n \cdot \log a / \log b}) = O(n^{\log_b a}) \]
Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them
- e.g. Addition with carry: each operation (takes 2 numbers and a carry bit, and gives a number and a new carry bit) works on single digit numbers
- To add two n-digit numbers: $O(n)$ operations
- As fast as possible: need to at least read all the digits
- (Remember: the number $N$ has $n=O(\log N)$ digits)
Multiplication of two large (binary) numbers

First attempt: \( x = x_0 + 2 \times x_1 \), where \( x_1 \) has one digit less

Similarly, \( y = y_0 + 2 \times y_1 \). So \( x \cdot y = x_0y_0 + 2 (x_0y_1 + x_1y_0) + 4x_1y_1 \).

\[ T(n) = T(n-1) + O(n) \quad \text{(and} \ T(1)=O(1)). \quad \text{So} \quad T(n) = O(n^2) \]

Can we do better by dividing the problem differently?

\( x = x_0 + 2^{n/2} \times x_1 \) where \( x_0, x_1 \) have \( n/2 \) digits each

(assuming \( n \) is a power of 2)

\( x \cdot y = x_0y_0 + 2^{n/2}(x_0y_1 + x_1y_0) + 2^n x_1y_1 \), where all 4 products are of \( n/2 \) digit numbers (mult. by a power of 2 and addition take \( O(n) \) time)

\[ T(n) = 4T(n/2) + \Theta(n). \quad \text{Still} \quad T(n)=\Theta(n^2). \]

Can we do better?
Big Number Arithmetic

Multiplication of two large numbers

\[ x = x_0 + 2^{n/2} x_1 \] where \( x_0, x_1 \) have \( n/2 \) digits each

(assuming \( n \) is a power of 2)

\[ x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1 \]

\[ = x_0y_0 + 2^{n/2}[ (x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1 ] + 2^n x_1y_1 \]

Only 3 multiplications (and reusing products). All of them on numbers about \( n/2 \) digits each

\[ T(n) = 3T(n/2) + O(n). \quad T(1) = O(1). \]

\[ a > b^d, \text{ where } a=3, b=2, d=1 \]

\[ T(n) = O(n^{\log_2 3}) = O(n^{1.585\ldots}) \]

Karastuba's Algorithm

Can do better, but more involved. Recently: \( O(n \log n) \), but with a very large constant.
Fast Matrix Multiplication

Multiplication of two large square matrices

Suppose we write $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

Then, $AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$, where $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$

Cost of multiplying two $n \times n$ matrices (assuming unit cost for both addition and multiplication)?

- $T(n) = 8T(n/2) + cn^2$
- $T(n) = n^\log \frac{8}{\log 2} = n^3$

Same as the naïve algorithm, computing each of the $n^2$ terms of $C$ using $O(n)$ operations

**Strassen's algorithm**: 7 smaller matrix multiplications instead of 8

- $T(n) = 7T(n/2) + cn^2 \implies T(n) = O(n^{\log_2 7}) = O(n^{2.81})$