Program Structure and Algorithms

Sid Nath

Lecture 3

Agenda

- Administrative
  - Please email me any time you need extra help!!!

- Lecture

Administrative

- Quiz 1 today
  - 30min (in-class)
  - Topic: Growth of functions, basic data structure operations

- Any qs on HW1?

- Lecture
  - More on sorting
  - Recurrences
  - Divide and Conquer
  - Binary search

QuickSort

- Pick a pivot element
- Create two subproblems
  - Elements smaller than the pivot is the first subproblem
  - Elements larger than the pivot is the second subproblem
- Sort each subproblem
- Merge

QuickSort

- Pick a pivot element
- Create two subproblems
  - Elements smaller than the pivot is the first subproblem
  - Elements larger than the pivot is the second subproblem
- Sort each subproblem
- Merge

QUICKSORT(A, p, r)
  if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r)
      QUICKSORT(A, p, q–1)
      QUICKSORT(A, q+1, r)

p: The starting index
r: The ending index
q: The pivot index

Initial call: QUICKSORT(A, 1, n)

QuickSort

- Pick a pivot element
- Create two subproblems
  - Elements smaller than the pivot is the first subproblem
  - Elements larger than the pivot is the second subproblem
- Sort each subproblem
- Merge

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q–1)
      QUICKSORT(A, q+1, r)
```

Initial call: QUICKSORT$(A, 1, n)$

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
FOR j = p TO r − 1 DO
    IF A[j] ≤ x THEN
        i = i + 1
        Exchange A[i] and A[j]
    FI
OD
Exchange A[i + 1] and A[r]
RETURN i + 1
```

Step1: Select a Pivot

7	**2**	**1**	**6**	**8**	**5**	**3**	**4**

- At random
- First element
- Last element (let's use this, so 4 is the pivot)
- Middle element

Step2: Create Two Subproblems

7	**2**	**1**	**6**	**8**	**5**	**3**	**4**

2	**1**	**3**	**4**	**8**	**5**	**7**	**6**

- Starting from the first element, compare each element to the pivot
  - Create a pointer pointing to the index of array when it is larger than the pivot
  - If element is smaller than the pivot, swap the indices of elements, pointer points to the index after swap
- When pointer is at (n-1) index, change it to the last index

Partitioning Step

Init: $i = -1, j = \ low = 0, high = 7, pivot = 4$

$low \leq j \leq high - 1; 0 \leq j \leq 6$

7	2	1	6	8	5	3	4

Step1: j = 0; Is A[0] < pivot? No

Step2: j = 1; Is A[1] < pivot? Yes → i+=1

i = 0, swap(A[i], A[j]), i.e., swap(7, 2)

2	7	1	6	8	5	3	4

Step3: j = 2; Is A[2] < pivot? Yes → i+=1

i = 1, swap(A[1], A[2]), i.e., swap(7,1)

2	1	7	6	8	5	3	4

Step4: j = 3; Is A[3] < pivot? No

Step5: j = 4; Is A[4] < pivot? No

Step6: j = 5; Is A[5] < pivot? No

Step7: j = 6; Is A[6] < pivot? Yes → i+=1

i = 2, swap(A[2], A[6]), i.e., swap(7,3)

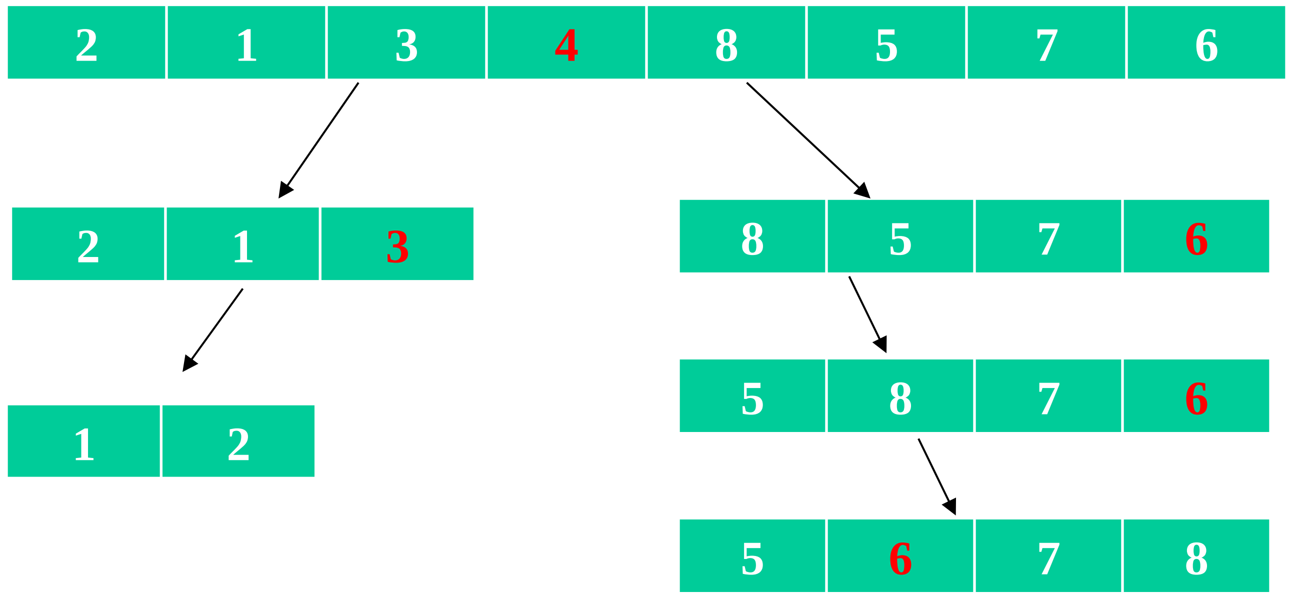
2	1	3	6	8	5	7	4

Step8: End: i+=1, swap(A[i], A[high])

i = 3, swap(A[3], A[7]), i.e., swap(6,4)

2	1	3	4	8	5	7	6

Step3: Sort Subproblems (Recursively)



Running Time Analysis

- T(n) = T(k) + T(n-k-1) + O(n) (to partition)

- Best-case: 2T(n/2) + O(n) (pivot ~ middle)
- Avg-case: T(n/9) + T(9n/10) + O(n)
- Avg and best-case: O(n log n)

- Worst-case: $\mathrm{T}(1) + \mathrm{T}(\mathrm{n} - 1) + \mathrm{O}(n) = O(n^2)$
  - Typically when pivot is the largest/smallest element
  - Array is sorted / reverse sorted
  - All elements are the same

Python Implementation

```python
def quickSort(arr, start, end):
    if start < end:
        # Pick the last element as pivot
        pivot = end
        i = start - 1
        for j in range(start, end):
            if arr[j] <= arr[pivot]:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[pivot] = arr[pivot], arr[i + 1]
        p = i + 1
        quickSort(arr, start, p - 1)
        quickSort(arr, p + 1, end)

arr = [5, 4, 3, 2, 1]
quickSort(arr, 0, len(arr) - 1)
for i in range(len(arr)):
    print(arr[i], end=" ")
```

Sorting so far…

- Comparison sorting
  - The only operation to decide the order of keys is comparison of pairs of keys.
  - Following are comparison sorts: ***insertion sort***, selection sort, ***merge sort (next class)***, **quicksort**, heapsort (once we study binary trees).

- How fast can we sort?
  - Is there a lower bound?
  - Can we do it in linear time?
- $\Omega(n)$ to examine all the input
- $\Omega(n \log n)$ is a lower bound for comparison sorts

Sorting Algorithms

- Insertion sort: incremental
- Quick sort: recursive
- Non-comparison-based linear-time sorting
  - Counting sort
  - Radix sort

Counting Sort

- Sorts the elements of an array by counting the #occurrences of each unique element in the array
- E.g., {4, 2, 2, 8, 3, 3, 1}
- Steps
  - Find max; max = 8
  - Init an array “count” of length (max + 1) with all elements as 0
  - Store the count of each element at their respective index in the count array;

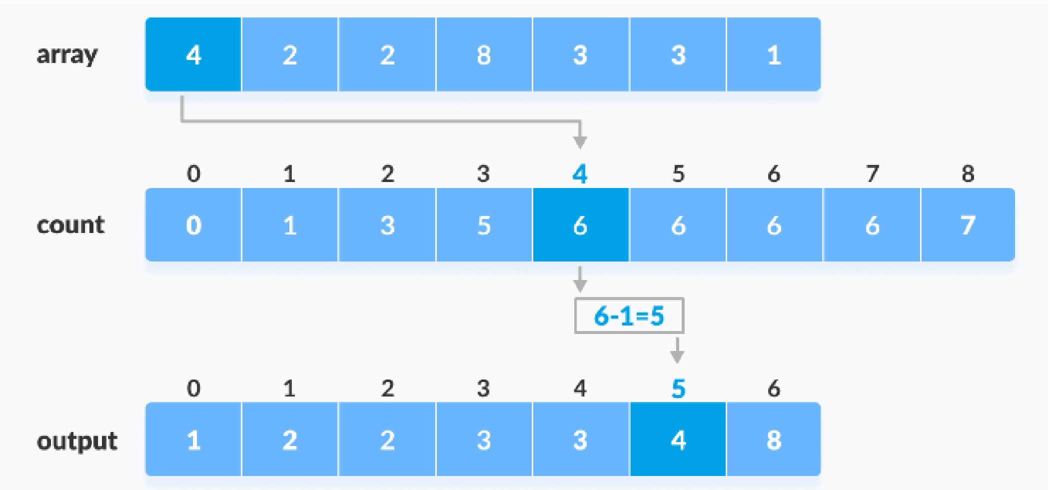
0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

  - Store cumulative sum of elements of the count array

0	1	3	5	6	6	6	6	7
0	1	2	3	4	5	6	7	8

Counting Sort

- Steps
  - Find the index of each element of input array in the count array. Place the element at the index calculated. Decrease count by 1 in the count array



- Runs in linear time O(n) // if we assume some value for MAX and true max <= MAX

Radix Sort

- Not a comparison-based sorting algorithm
- Uses digits (in some radix, e.g., base 10) of integer keys from least to most significant digits
- E.g., {121, 432, 564, 23, 1, 45, 788}

1	2	1
0	0	1
4	3	2
0	2	3
5	6	4
0	4	5
7	8	8

Sorted by units digit

0	0	1
1	2	1
0	2	3
4	3	2
0	4	5
5	6	4
7	8	8

Sorted by tens digit

0	0	1
0	2	3
0	4	5
1	2	1
4	3	2
5	6	4
7	8	8

Sorted by hundreds digit

Radix Sort

- Find the max element in the array. Let X be the #digits in it
  - E.g., max = 788 and has 3 digits
- Iterate over each significant digit from the least one
  - Use any sorting (e.g., counting) technique to sort the digits at each significant place

1	2	1
0	0	1
4	3	2
0	2	3
5	6	4
0	4	5
7	8	8

Solving Recurrences

- Why?
  - Many algorithms use recursions (D/Q, DP)
  - Non-trivial to analyse running time by unrolling the #times a recursive function is called
  - Recurrences provide a general way to analyze running time of algorithms that use recursion

Recurrence--Overview

- A ***recurrence*** is a function and is defined terms of
  - one or more base cases, and
  - itself, with smaller arguments.
- A recurrence could have 0, 1 or more functions that satisfy it
  - Well-defined if at least one function satisfies
  - Ill-defined otherwise

  T(n) = aT(f(n)) + $\theta(g(n) + c)$ => θ(f'(n))

  T(n) ≤ aT(f(n)) + $\theta(g(n) + c)$ => O(f'(n))

- How to solve recurrence
  - substitution method
  - recursion tree method
  - Master method
  - Akra-Bazzi method → not covered

Examples of Recurrences

- An algo that breaks a problem of size n into one subproblem of size n/3 and another of size 2n/3, taking θ(n) to combine
  - $T(n) = T(n/3) + T(2n/3) + \theta(n)$

- An algo that creates one subproblem and it has one element less than the original problem
  - $T(n) = T(n - 1) + \theta(1)$

Substitution method

- Guess the solution.
- Use induction to show that the solution works.
- *Example:* Determine an asymptotic upper bound on $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$.
  - Floor function ensures that $T(n)$ is defined over integers.

  Guess: $T(n) = O(n \log n)$

Inductive step: Assume that $T(n) \le cn \lg n$ for all numbers $\ge n_0$ and $< n$. If $n \ge 2n_0$, holds for $\lfloor n/2 \rfloor \Rightarrow T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$. Substitute into the recurrence:

$$
\begin{aligned}
T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\
&\le 2(c(n/2) \lg(n/2)) + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - cn \lg 2 + \Theta(n) \\
&= cn \lg n - cn + \Theta(n) \\
&\le cn \lg n \,.
\end{aligned}
$$

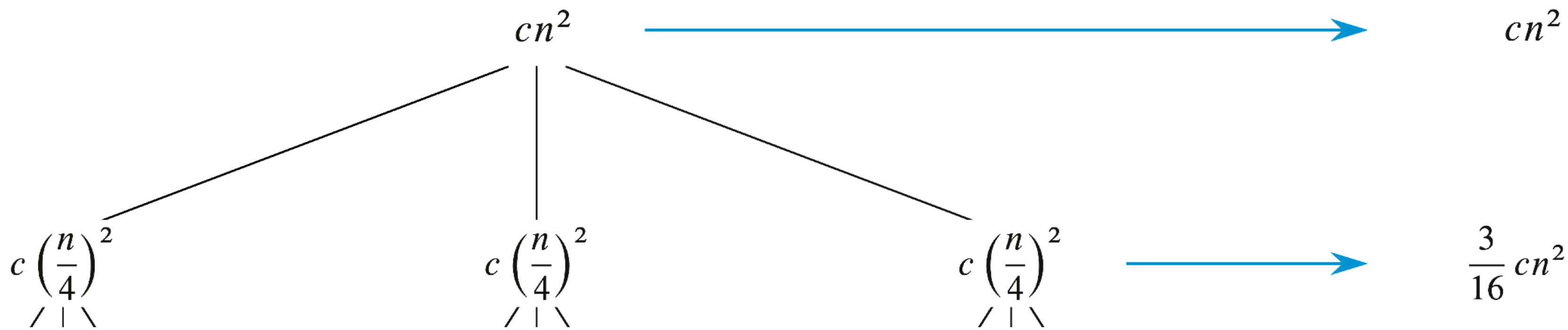
Recursion trees

- Original problem: root with size *n*
- Each non base node has *a* children with size *n/b*
- By summing across each level, the recursion tree shows the cost at each level of recursion
  - Total cost= sum of all levels
- Can be used to generate a guess. Then verify by substitution method.

Example

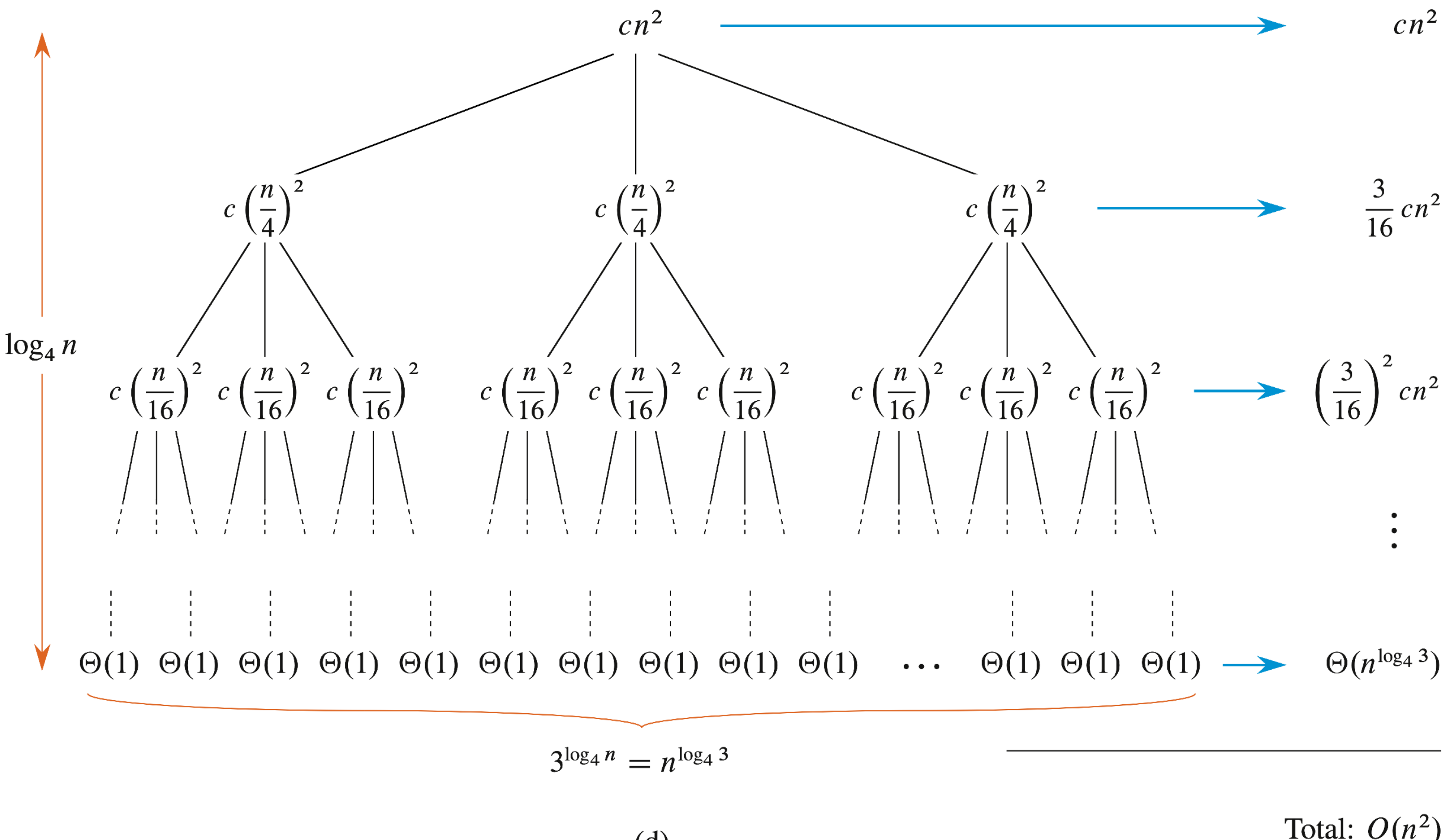
$T(n) = 3T(n/4) + \Theta(n^2)$.

Draw out a recursion tree for $T(n) = 3T(n/4) + cn^2$:



For simplicity, assume that n is a power of 4 and the base case is $T(1) = \Theta(1)$.

Example Contd.



(d)

Recurrence Tree Analysis

- Subproblem size for nodes at depth i is $n/4^i$
  - Base case is when $n/4^i = 1 \rightarrow i = \log_4 n$
- Each node has 3x nodes as previous level, so depth i has 3^i nodes
- Each node at depth i has cost $c\left({}^{n}/_{4^i}\right)^2 \rightarrow 3^i\, c\left({}^{n}/_{4^i}\right)^2 = ({}^{3}/_{16})^i cn^2$ is the total cost at depth i
- Leaf level has depth $\log_4 n$, so #leaves is $3^{\log_4 n} = n^{\log_4 3}$
- Cost of each leaf node $\theta(1)$, so total cost of leaves is $\theta(n^{\log_4 3})$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)\ .
\end{aligned}
$$

Master method –General

Consider $T(n) = aT(n/b) + f(n)$

Theorem 4.1 Master theorem

Case 1: $f(n) = O(n^{\log_b a-\varepsilon}),\ \varepsilon > 0 \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a})$

Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n),\ k \geq 0 \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Case 3: $f(n) = \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon > 0 \quad \Rightarrow \quad T(n) = \Theta(f(n))$

$T(n) = 5T(n/2) + \Theta(n^2)$
$n^{\log_2 5}$ vs. n^2
Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$

$T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
$n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
Use case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

Example

$T(n) = 5T(n/2) + \Theta(n^3)$
$n^{\log_2 5}$ vs. n^3
Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
Check regularity condition (don't really need to since $f(n)$ is a polynomial):
$af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$
Use case 3 $\Rightarrow T(n) = \Theta(n^3)$

$T(n) = 27T(n/3) + \Theta(n^3/\lg n)$
$n^{\log_3 27} = n^3$ vs. $n^3/\lg n = n^3 \lg^{-1} n \neq \Theta(n^3 \lg^k n)$ for any $k \ge 0$.
Cannot use the master method.

Master method – Another Way

This seems easier, but not general

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

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{if } d > \log_b a \quad \text{Case 3} \\ \mathcal{O}(n^d \log n) & \text{if } d = \log_b a \quad \text{Case 2} \\ \mathcal{O}(n^{\log_b a}) & \text{if } d < \log_b a \quad \text{Case 1} \end{cases}$$

Examples

- $T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{\log n}$

- $T(n) = 2T\left(\frac{n}{4}\right) + \Theta(1)$

- $T(n) = 3T\left(\frac{n}{4}\right) + n\ \log n$

- $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

Divide and Conquer (D-Q)

- Basic idea
  - *divide* the problem into 2 or more sub problems
  - *conquer* the sub problems recursively
  - combine sub problems

- Binary search
- Merge sort
- Matrix multiplication
- Find Majority

Binary Search

Given a **sorted** array $A[1{:}n]$ integers, find the position of x (another integer) if it exists in $A[1{:}n]$

Linear search?

- Scan through every element in A starting from index 1
- Return i if $A[i] == x$
- Return -1 if x cannot be found we index is n.

Running time is $O(n)$

Binary Search

Given a **sorted** array $A[1{:}n]$ integers, find the position of x (another integer) if it exists in $A[1{:}n]$

Binary search

- Find middle element mid = lo + (hi – lo)/2
- Return i if $A[mid] == x$
- If $A[mid] < x$, search $A[mid + 1{:}hi]$ (subproblem)
- If $A[mid] > x$, search $A[lo{:}mid - 1]$ (subproblem)
- Return -1 if x cannot be found in subproblem.
- Initially, lo = 1, hi = n

Binary Search – Running time

- In each iteration, we discard one half of the subproblem
- So, the size of each successive subproblem is halfved
  - Let, $n = 2^k$
  - Subproblem sizes: $2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^{k-k} = 1$
- In each iteration we do a constant amount of work
  - Either check for equality or inequality of $A[mid]$ with x
- We can write this down as a recurrence relation
  - $T(n) = T\left(\frac{n}{2}\right) + O(1)$
- Using Master theorem, we can solve this as:
  - $a = 1, b = 2, d = 0, k = 0 \Rightarrow \log a / \log b \ = d \Rightarrow Case\ 2$
  - So, $\Theta\left(n^d \log^{k+1} n\right) = \Theta(\log n)$

Binary Search -- Example

A = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}; x = 23

Iteration #1 (lo = 0, hi = 9, x = 23)

mid = 0 + (9 – 0)/2 = 4
A[4] = 16 < 23, so look in upper half
that is, A[5:9]

Iteration #2 (lo = 5, hi = 9, x = 23)

mid = 5 + (9 - 5)/2 = 7
A[7] = 56 > 23, so look in lower half
that is, A[5:6]

Iteration #3 (lo = 5, hi = 6, x = 23)

mid = 5 + (6 - 5)/2 = 5
A[5] = 23 == 23, so return 5

Lecture 3 summary

- Sorting – Quicksort, Counting and Radix sorts
- Solving recurrences
- D/Q algorithms
- Binary search