香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 6: Complexity analysis with recursion and divide-and-conquer

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Review of asymptotic notations
- Steps of worst-case analysis
- Complexity analysis
  - Recursion
  - Divide-and-conquer

Review of asymptotic notations

- Big-Oh definition:
  - $g(n) = O(f(n))$ if and only if there exist positive constants c and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$
- Big-Omega definition:
  - $g(n) = \Omega(f(n))$ if and only if there exists a positive constant c and n_0 such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$
- Big-Theta definition:
  - If $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$, then $g(n) = \Theta(f(n))$

Steps for worst-case analysis

- Step 1: find the worst-case number of basic operations in the algorithm as a function of the input size
  - Example: Linear search & Binary search
    - We counted that the number of basic operations by Linear Search and Binary search are $4n + 4$ and $12 \log_2 n + 17$, respectively

- Step 2: Use Big-Oh and Big-Omega to analyze the algorithm, and derive the Big-Theta if possible
  - We may not be lucky enough to derive that Big-Oh and Big-Omega to be the same
  - Less rigorously, we may also say that an algorithm with $O(\log n)$ time complexity is better than the one with $O(n)$, when we cannot derive that Big-Oh and Big-Omega to be the same

Counting basic operations

Sum_LinearSearch(A, searchnum, sumestimation)

Input: array A, a search number, and a sum estimation
Output: return 1 if the search number exists in A and the sum estimation is exactly the sum of the array, otherwise return 0

```
1  tempsum = 0                                                          O(1)
2  for i = 0 to n-1                                                     O(n)
3     tempsum += A[i]                                                   O(n)
4  findmatch = linear_search(A, searchnum)
5  return findmatch!=-1 and tempsum == sumestimation                    O(1)
```

$O(n)$ by further counting its number of basic operations

How to count basic operations in recursion?

BinarySearch(arr, searchnum, left, right)

1	`if left == right`	$O(1)$
2	`    if arr[left]= searchnum`	$O(1)$
3	`        return left`	$O(1)$
4	`    else`	$O(1)$
5	`        return -1`	$O(1)$
6	`middle = (left + right)/2`	$O(1)$
7	`if arr[middle] = searchnum`	$O(1)$
8	`    return middle`	$O(1)$
9	`elseif arr[middle] < searchnum`	$O(1)$
10	`    return BinarySearch(arr, searchnum, middle+1, right)`	$O(?)$
11	`else`	
12	`    return BinarySearch(arr, searchnum, left, middle -1)`	$O(?)$

Counting basic operations in recursion

- Given input size n, let $g(n)$ be the total # of basic operations executed in BinarySearch in the worst case

BinarySearch(arr, searchnum, left, right)

```
1   if left == right
2       if arr[left]= searchnum
3           return left
4       else
5           return -1
6   middle =(left + right)/2
7   if arr[middle] = searchnum
8       return middle
9   elseif arr[middle] < searchnum
10      return BinarySearch(arr, searchnum, middle+1, right)
11  else
12      return BinarySearch(arr, searchnum, left, middle -1)
```

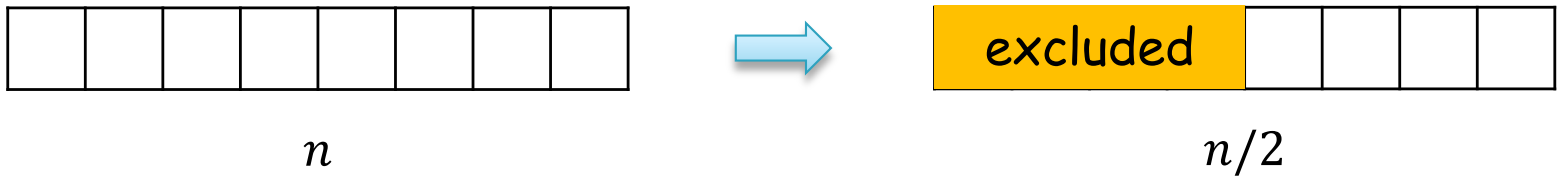
We can still count the number of basic operations for this part

The total number of basic operations executed is a constant independent of the input size n, we can use a to denote this

We either run line 10 or line 12, but not both. What is the number of basic operations that are executed by Line 10 or 12?

Analysis for recursive binary search (i)

- $g(n)$ can be also defined recursively
  - At the beginning, the input size is n
  - After executing a basic operations, we reduce the input size by half
  - Then, we run the recursive binary search with input size $n/2$
    - What is the number of basic operations executed in the worst case by recursive binary search with input size $n/2$?
    - We do not know, but we know it is $g(\frac{n}{2})$ according to our definition



$$g(n) = a + g\left(\frac{n}{2}\right)$$

Analysis for recursive binary search (ii)

- Given $g(n) = a + g\left(\frac{n}{2}\right)$, $g(1) = b$
  - What is $g(4)$ by using a and b to represent?

$$\begin{aligned} g(4) &= g(2) + a \\ &= g(1) + a + a \\ &= 2a + b \end{aligned}$$

  - What is $g(n)$ by using a and b to represent if $n = 2^x$?

$$\begin{aligned} g(n) &= g\left(\frac{n}{2}\right) + a \\ &= g\left(\frac{n}{2^2}\right) + a + a \\ &= g\left(\frac{n}{2^3}\right) + a + a + a \\ &= g\left(\frac{n}{2^4}\right) + a + a + a + a \\ &= \cdots \qquad\qquad x \text{ of them} \\ &= g(1) + \overbrace{a + a + \cdots + a + a}^{} = x \cdot a + b = a \cdot \log_2 n + b \end{aligned}$$

Analysis for recursive binary search (iii)

- How to analyze if $n \neq 2^x$?
  - We can simulate searching on an array of size 2^x, where x is the smallest integer such that $2^x \geq n$

n

$n' = 2^x$

$n' \leq 2n$

- In this case, $g(n) \leq g(2^x)$, and we have that:
  - $g(n) \leq g(2^x) \leq a \cdot x + b$

    $\leq a \cdot \log_2 (2n) + b = a \cdot \log_2 n + (a + b)$
  - $g(n) = O(\log n)$

Selection sort

- Step 1: Scan all the n elements in the array to find the position i_{max} of the largest element $maxnum$

$i_{max} = 4, maxnum = 9$

4	2	3	6	9	5

- Step 2: swap the position of the last one and $maxnum$

4	2	3	6	5	9

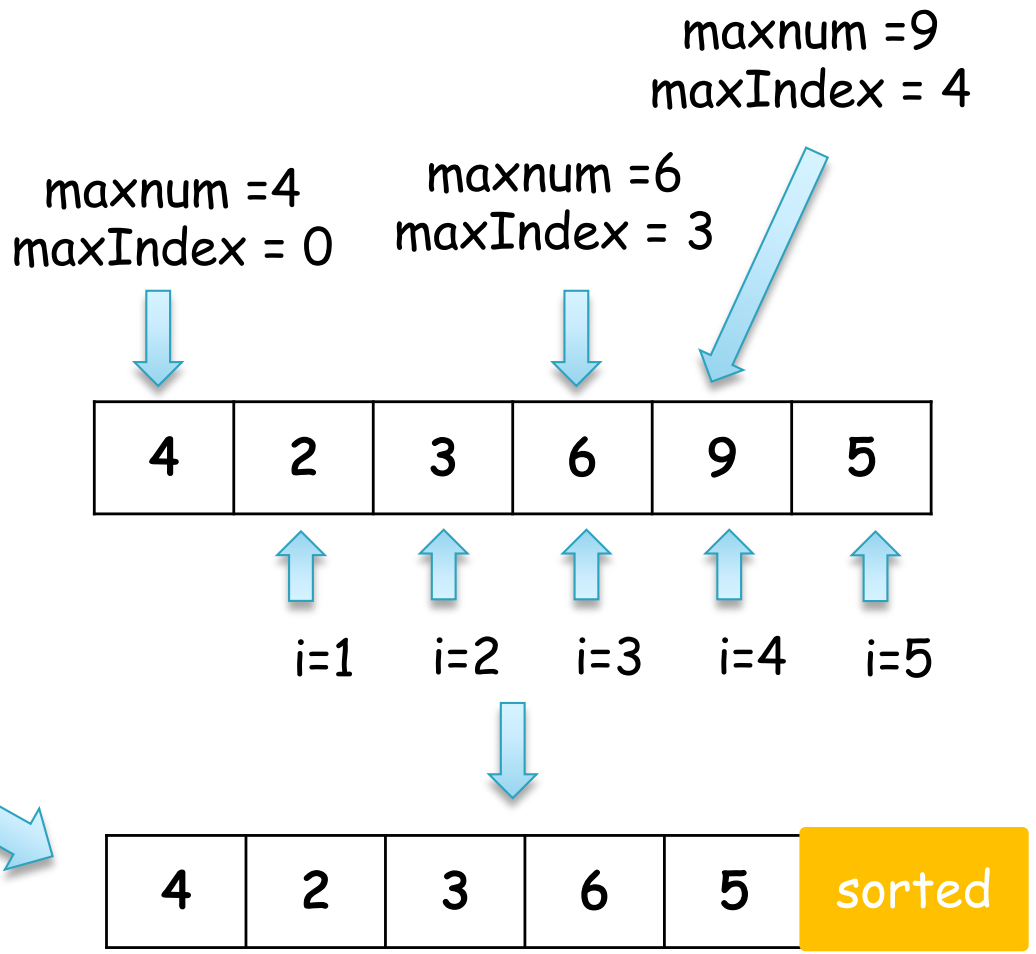
- Step 3: We have a smaller problem: sorting the first n-1 elements

4	2	3	6	5	sorted

Selection sort

SelectionSort(arr, n)

```
1   if n ≤ 1
2     return arr
3   maxnum = arr[0]
4   maxIndex = 0
5   for i = 1 to n -1
6       if maxnum < arr[i]
7           maxnum = arr[i]
8           maxIndex = i
9   arr[maxIndex] = arr[n-1]
10  arr[n-1] = maxnum
11  SelectionSort(arr, n-1)
```



Selection sort: complexity analysis

SelectionSort(arr, n)		# of basic operations
1	`if n ≤ 1`	$O(1)$
2	`  return arr`	$O(1)$
3	`maxnum = arr[0]`	$O(1)$
4	`maxIndex = 0`	$O(1)$
5	`for i = 1 to n -1`	$O(n)$
6	`  if maxnum < arr[i]`	$O(n)$
7	`  maxnum = arr[i]`	$O(n)$
8	`  maxIndex = i`	$O(n)$
9	`arr[maxIndex] = arr[n-1]`	$O(1)$
10	`arr[n-1] = maxnum`	$O(1)$
11	`SelectionSort(arr, n-1)`	$O(?)$

- What is the total # of basic operations from Lines 1-10?
  - O(n)

Selection sort: complexity analysis

- Let $g(n)$ be the total number of basic operations in the worst case. Let $g(1) = b$
  - We have that:
    - $g(n) = g(n-1) + O(n)$
  - We can find constant c such that
    - $g(n) \leq g(n-1) + c \cdot n$

- We have that:
  - $g(n) \leq g(n-1) + c \cdot n \leq g(n-2) + c \cdot n + c \cdot (n-1)$
    $\leq g(n-3) + c \cdot n + c \cdot (n-1) + c \cdot (n-2)$
    $\leq g(1) + c \cdot n + c \cdot (n-1) \cdots + c \cdot 2$
    $\leq c \cdot \frac{n(n+1)}{2} + b$
  - $g(n) = O(n^2)$

In many cases, we only want to have the upper bound of the worst case running time. Deriving its Big-Oh is sufficient.

Practice

- Analyze the time complexity of maxInArray1 algorithm

maxInArray1(arr, n)

1	`if n == 1`	$O(1)$
2	`  return arr[0]`	$O(1)$
3	`else`	$O(1)$
4	`  tempMax = maxInArray1(arr, n-1)`	?
5	`  return max(arr[n-1], tempMax)`	$O(1)$

- Denote $g(n)$ as the number of basic operations executed by maxInArray1 in the worst case when the input size is n
  - For Line 4, it is invoking maxInArray1 itself with an input size of n-1
  - Then its number of basic operations is $g(n-1)$, so $g(n) = g(n-1) + O(1)$
  - Then, there exists some constant c such that $g(n) \leq g(n-1) + c$. Let $g(1) = a$
    - $g(n) \leq g(n-1) + c \leq g(n-2) + 2c \ldots . \leq (n-1)c + a \leq cn + a$
    - $g(n) = O(n)$

Practice (Cont.)

- Analyze the time complexity of maxInArray2 algorithm

maxInArray2(arr, left, right)

1	`if left == right`	$O(1)$
2	`  return arr[left]`	$O(1)$
3	`else`	$O(1)$
4	`  mid = (left+right)/2`	$O(1)$
5	`  maxLeft = maxInArray2(arr, left,middle)`	?
6	`  maxRight = maxInArray2(arr,middle+1,right)`	?
7	`  return max(maxLeft, maxRight)`	$O(1)$

- Define $g(n)$ as the number of basic operations executed by maxInArray2 in the worst case when the input size is n
  - $g(n) = 2g\left(\frac{n}{2}\right) + O(1)$. Let $g(1) = b$
  - When $n = 2^x$, we have that: $g(n) \leq 2g\left(\frac{n}{2}\right) + c \leq 4g\left(\frac{n}{4}\right) + c + 2c \leq 8g\left(\frac{n}{8}\right) + c + 2c + 4c \ldots \leq 2^x \cdot g(1) + c + 2c + 4c + \cdots + 2^{x-1}c \leq bn + cn - c$
  - When $n \neq 2^x$, we can follow similar analysis as Page 7 and show that $g(n) \leq g(n') \leq bn' + n' - c \leq 2bn + 2cn - c$. Thus, $g(n) = O(n)$

Master theorem (big-Oh version)

- A formula for solving many recurrence relations!

- Let $T(n)$ be the running cost depending on the input size n, and we have its recurrence:
  - $T(1) = O(1)$
  - $T(n) \leq a \cdot T(n/b) + O(n^d)$
  - a, b, d are constants such that $a \geq 1, b > 1, d \geq 0$. Then,

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

Note that n/b may not be an integer, but it will not affect the asymptotic behavior of recurrence

Master theorem: intuition

- Recurrence: $T(n) \leq a \cdot T(n/b) + O(n^d)$
- An algorithm that divides a problem of size n into a subproblems, each of size n / b

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

a: number of subproblems (branching factor)
b: factor by which input size shrinks (shrinking factor)
d: need to do $O(n^d)$ work to create subproblems + "merge" their solutions

Master theorem: examples

$$T(n) \leq a \cdot T(n/b) + O(n^d)$$

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

- $g(1) = c_0, g(n) \leq g(n/2) + c$
  - We have that $a = 1, b = 2, d = 0$
  - Since $\log_b a = d$, we know that: $g(n) = O(n^0 \cdot \log n) = O(\log n)$
- $g(1) = c_0, g(n) \leq g(n/2) + c_1 \cdot n$
  - We have that $a = 1, b = 2, d = 1$
  - Since $\log_b a < d$, we know that: $g(n) = O(n^d) = O(n)$

Master theorem: examples

$$T(n) \leq a \cdot T(n/b) + O(n^d)$$

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

- $g(1) = c_0, g(n) \leq 2 \cdot g(n/2) + c_1 \cdot n^{0.5}$
  - We have that $a = 2, b = 2, d = 0.5$
  - Since $\log_b a > d$, we have that: $g(n) = O(n^{\log_b a}) = O(n)$
- $g(1) = c_0, g(n) \leq 2 \cdot g(n/4) + c_1 \cdot \sqrt{n}$
  - We have $a = 2, b = 4, d = 0.5$
  - Since $\log_b a = d$, we have that: $g(n) = O\left(n^d \cdot \log n\right) = O(\sqrt{n} \cdot \log n)$

Practice

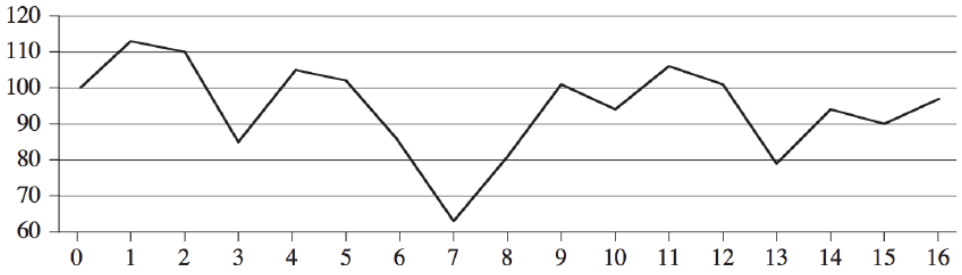
$$T(n) \leq a \cdot T(n/b) + O(n^d)$$

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

- Solve the following recurrence relations
  - $g(1) = c_0, g(n) \leq 8g(n/2) + c_1 \cdot n^2$
  - $g(1) = c_0, g(n) \leq 2g(n/8) + c_1 \cdot n^{\frac{1}{3}}$
  - $g(1) = c_0, g(n) \leq 2g(n/4) + c_1 \cdot n$

Maximum-subarray problem

- Consider to buy a stock given the prediction curve
  - To earn more money, we often "buy low and sell high"



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- How to earn more money?
  - First: choose highest price and go left to find the lowest price
  - Second: choose lowest price and go right to find the highest price
  - Third: try every possible pair of buy and sell dates to find the optimal one, but this will lead to $O(n^2)$ time cost
  - Can we find an optimal solution with smaller time cost?

A transformation

- Find a sequence of days over which the summarized net change is maximum

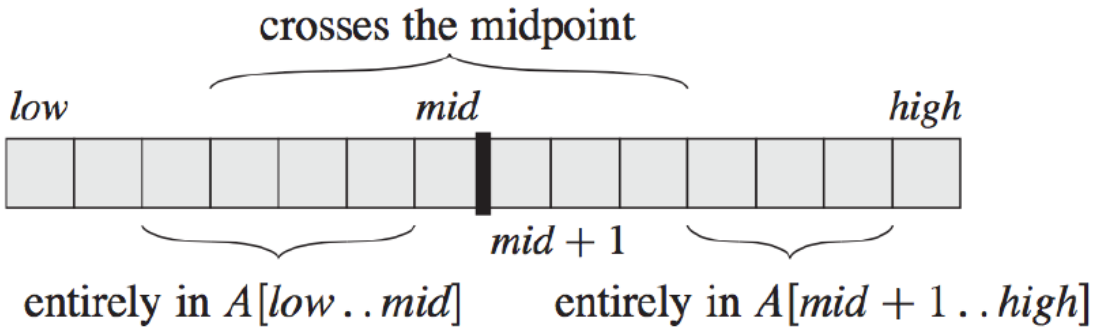
Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Find the nonempty, contiguous subarray of the array A whose values have the largest sum, which is called the max subarray

- The max subarray problem
  - In computer science, it is the task of finding a contiguous subarray with the largest sum, within a given array $A[1...n]$ of numbers
  - It can be extended for multi-dimensional arrays

A divide-and-conquer solution

- Suppose we want to find a max subarray of *A[low,…,high]*
  - The middle point, *mid=(low+high)/2*
- Suppose *A[i,…,j]* is the max subarray of *A[low,…,high]*
- Three situations:
  - *A[i,…,j]* in *A[i,…,mid]*
  - *A[i,…,j]* in *A[mid+1,…,j]*
  - *A[i,…,j]* crosses *mid*

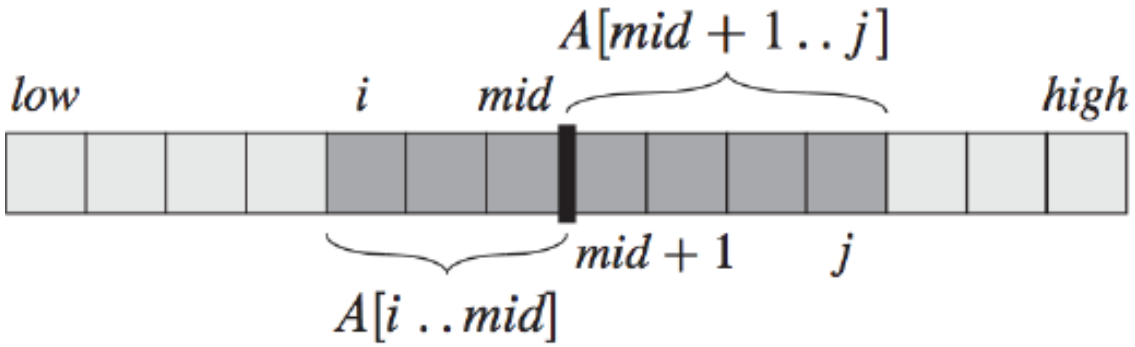


A divide-and-conquer solution

- Solve the subarrays A[low,…,mid] and A[mid+1,…,high] (two sub-problems) recursively, and then the third case low<=i<=mid<j<=high
- Then take the largest of these three
- How to solve the third case?
  - Seems not a sub-problem
  - But the added restriction – crossing the midpoint - helps

A divide-and-conquer solution

- Any subarray crossing the midpoint is itself made of two subarrays A[i,…,mid] and A[mid+1,…,j]
- The restriction fixes one ending point for the first array and one starting point for the second array
- Thus, we just need to find maximum subarrays of the form A[i,…,mid] and A[mid+1,…,j] and then combine them



Find max subarray crossing midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

What is the time cost? Θ(n)

Analyze running time

- The base case, when n = 1, is easy, $T(1) = \Theta(1)$
- Solve two subarrays $2T(n/2)$, solve the subarray crossing midpoint $\Theta(n)$, thus the running time $T(n)=2T(n/2)+\Theta(n)$

- By master theorem, the divide-and-conquer solution takes $\Theta(n\log n)$ time, which is faster than brute-force!

- Question: is there a better solution?
  - Answer: Yes, see Ex.4.1-5

Questions

- Can we use master theorem to solve the following recurrences?
  - $T(n) \leq a \cdot T(n-1) + c$
  - T(n) = T(n/5) + T(7n/10) + n, where T(n) = 1 when $1 \leq n \leq 10$

Recommended reading

- Reading this week
  - Chapter 4, textbook
- Next week
  - List: Chapter 10, textbook

(Materials of back slides are NOT included in the midterm and final exams)

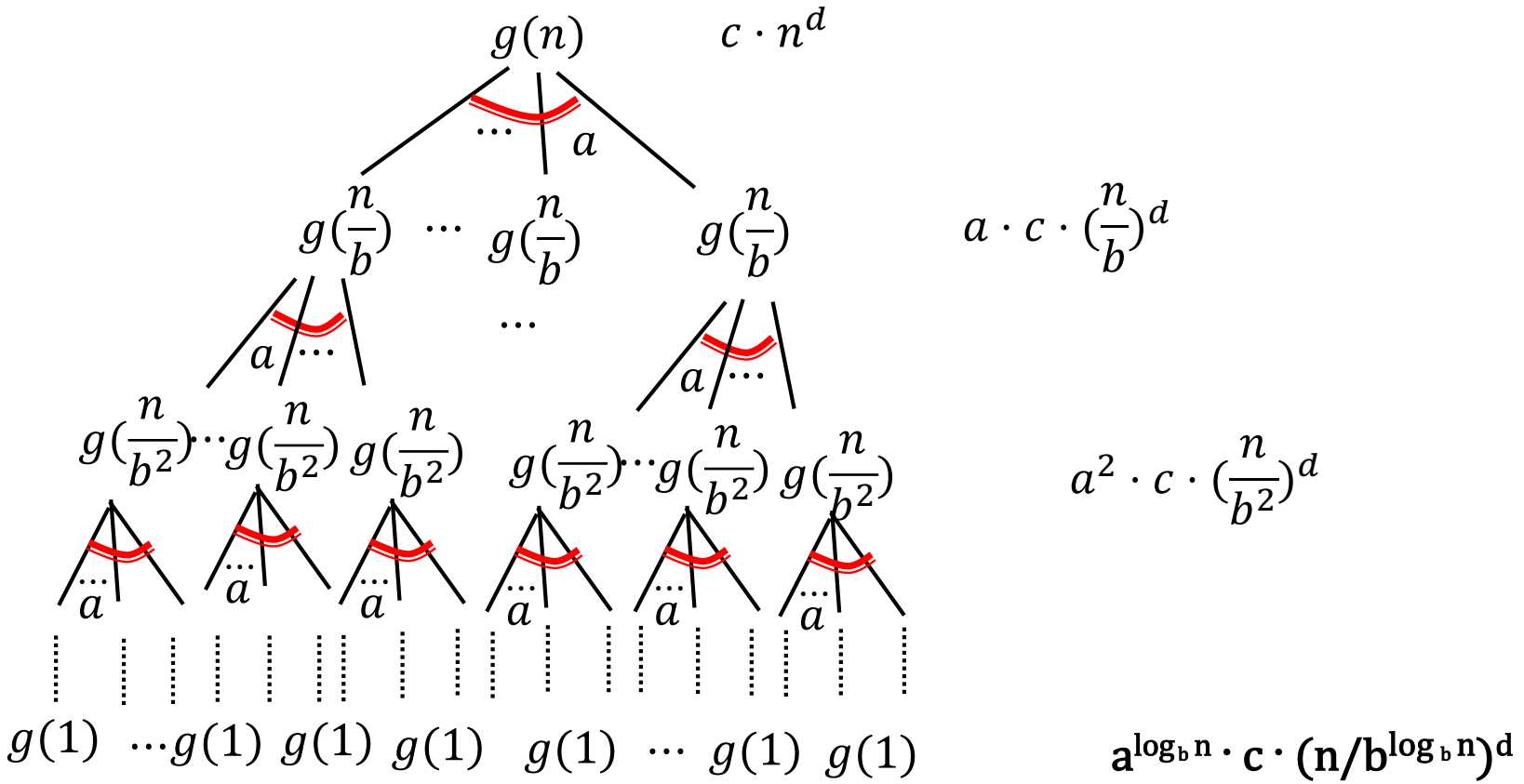
Backup slides

Proof of master theorem

- Preparation:
  - $\log_b n^x = x \cdot \log_b n$
  - $a^{\log_b n} = n^{\log_b a}$, which implies that $b^{\log_b n} = n$
  - For $x > 1$, $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$
  - For $0 < x < 1$, $\sum_{i=0}^{n} x^i \leq \frac{1}{1-x}$
  - For $x = 1$, $\sum_{i=0}^{n} x^i = (n+1)$

Recursion tree of master theorem

- Draw the tree for $T(n) = a \cdot T(n/b) + c \cdot n^d$
- Fill out the table & sum up last column (from t = 0 to t = $\log_b n$)



Recursion tree of master theorem

- Draw the tree for $T(n) = a \cdot T(n/b) + c \cdot n^d$
- Fill out the table & sum up last column (from t = 0 to t = $\log_b n$)

LEVEL	# OF PROBLEMS	SIZE OF EACH PROBLEM	WORK PER PROBLEM	TOTAL WORK AT THIS LEVEL
0	1	n	$c \cdot n^d$	$\mathbf{1 \cdot c \cdot n^d}$
1	a	n/b	$c \cdot (n/b)^d$	$\mathbf{a \cdot c \cdot (n/b)^d}$
...				
t	a^t	n/b^t	$c \cdot (n/b^t)^d$	$\mathbf{a^t \cdot c \cdot (n/b^t)^d}$
...				
$\log_b n$	$a^{\log_b n}$	$n/b^{\log_b n} = 1$	$c \cdot (n/b^{\log_b n})^d$	$\mathbf{a^{\log_b n} \cdot c \cdot (n/b^{\log_b n})^d}$ (This = 1)

Total amount of work:

$$c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

(add work across all levels up, then factor out the c & n^d terms & write in summation form)

Recursion tree of master theorem

- Draw the tree for $T(n) = a \cdot T(n/b) + c \cdot n^d$
- Fill out the table & sum up last column (from t = 0 to t = $\log_b n$)

So $T(n) \leq$ $$c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$$

We can verify that for each of the three cases (**a =, <, or > b^d**), this equation above gives us the desired results:

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

Case 1: $a = b^d$

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

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

$$= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1$$

This is equal to 1!

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

$$= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right)$$

$$= \Theta(n^d \log(n))$$

Case 2: $a < b^d$

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

This is less than 1!

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

$$= c \cdot n^d \cdot [\text{ some constant }]$$

$$= \Theta(n^d)$$

Geometric series with the "multiplier" < 1 & constant!

Case 3: $a > b^d$

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

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

$$= \Theta\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)}\right)$$

$$= \Theta\left(n^{\log_b(a)}\right)$$

This is greater than 1!

The n^d term cancels with $(b^d)^{\log_b n}$! and $a^{\log_b n} = n^{\log_b a}$!

Use the geometric series formula to convince yourself that this is legitimate!