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

Li Jiang
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 two 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 two positive constants 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
    - We counted that the number of basic operations by Linear Search is $4n + 3$
- 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

`tempsum = 0`	$O(1)$
`for i = 0 to n-1`	$O(n)$
`   tempsum += A[i]`	$O(n)$
`findmatch = linear_search(A, searchnum)`	$O(n)$ by further counting its number of basic operations
`return findmatch!=-1 and tempsum == sumestimation`	$O(1)$

How to count basic operations in recursion?

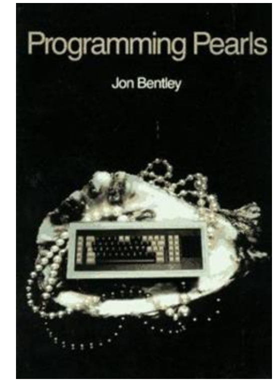
BinarySearch(arr, searchnum, left, right)

```
if left >= right                                          O(1)
    if arr[left] == searchnum                             O(1)
        return left                                       O(1)
    else                                                  O(1)
        return -1                                         O(1)
middle = (left + right)/2                                 O(1)
if arr[middle] == searchnum                               O(1)
    return middle                                         O(1)
elseif arr[middle] < searchnum                            O(1)
    return BinarySearch(arr, searchnum, middle+1, right)  O(?)
else
    return BinarySearch(arr, searchnum, left, middle -1)  O(?)
```

A cautionary tale

- Only 10% of programmers can write binary search!

- Binary search dates back to 1946 (at least)
  - First correct description in 1962
  - Jon Bentley (CMU) wrote the definitive binary search and proved it correct



> *I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found).*

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)
```

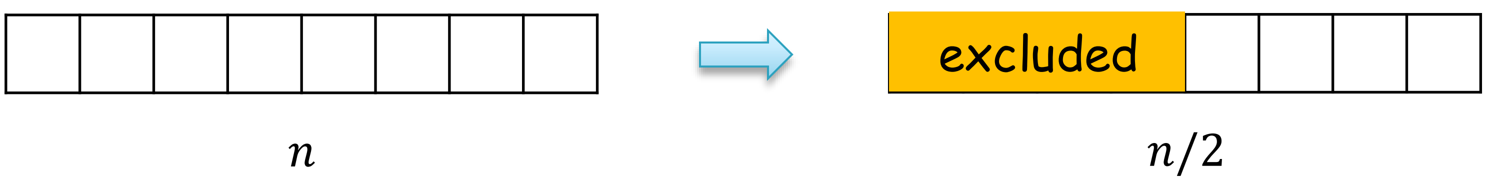
We can still count the number of basic operations for this part

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 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?

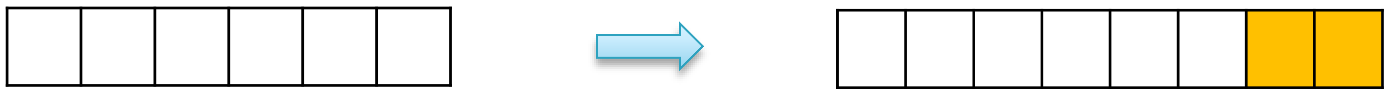
$$\begin{aligned} g(4) &= g(2) + a \\ &= g(1) + a + a \\ &= 2a + b \end{aligned}$$

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

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

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

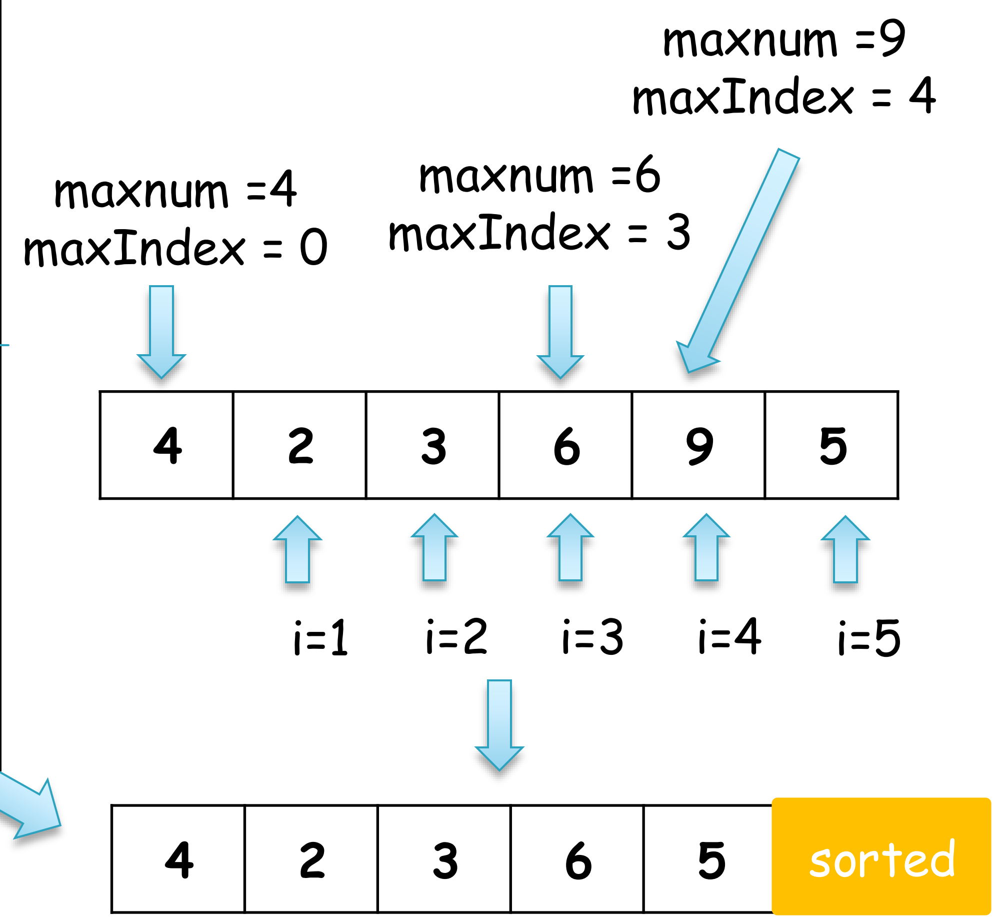
- Step 3: We have a smaller problem: sorting the first n-1 elements

4	2	3	6	5	sorted

Selection sort

SelectionSort(arr, n)

```
if n ≤ 1
  return arr
maxnum = arr[0]
maxIndex = 0
for i = 1 to n -1
    if maxnum < arr[i]
       maxnum = arr[i]
       maxIndex = i
arr[maxIndex] = arr[n-1]
arr[n-1] = maxnum
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 and $g(1) = b$
  - We have that:
    - $g(n) = g(n-1) + O(n)$
  - We can find constant c such that
    - $g(n) \le g(n-1) + c \cdot n$

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

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`	
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)

```
if left == right                                   O(1)
  return arr[left]                                 O(1)
else                                               O(1)
   mid = (left+right)/2                            O(1)
   maxLeft = maxInArray2(arr, left,middle)         ?
   maxRight = maxInArray2(arr,middle+1,right)      ?
   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
  - $g(n) = 2g\left(\frac{n}{2}\right) + O(1)$ and 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 11 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:
  - $T(1) = O(1)$
  - $T(n) \le a \cdot T(n/b) + O(n^d)$
  - a, b, d are constants such that $a \ge 1, b > 1, d \ge 0$. 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

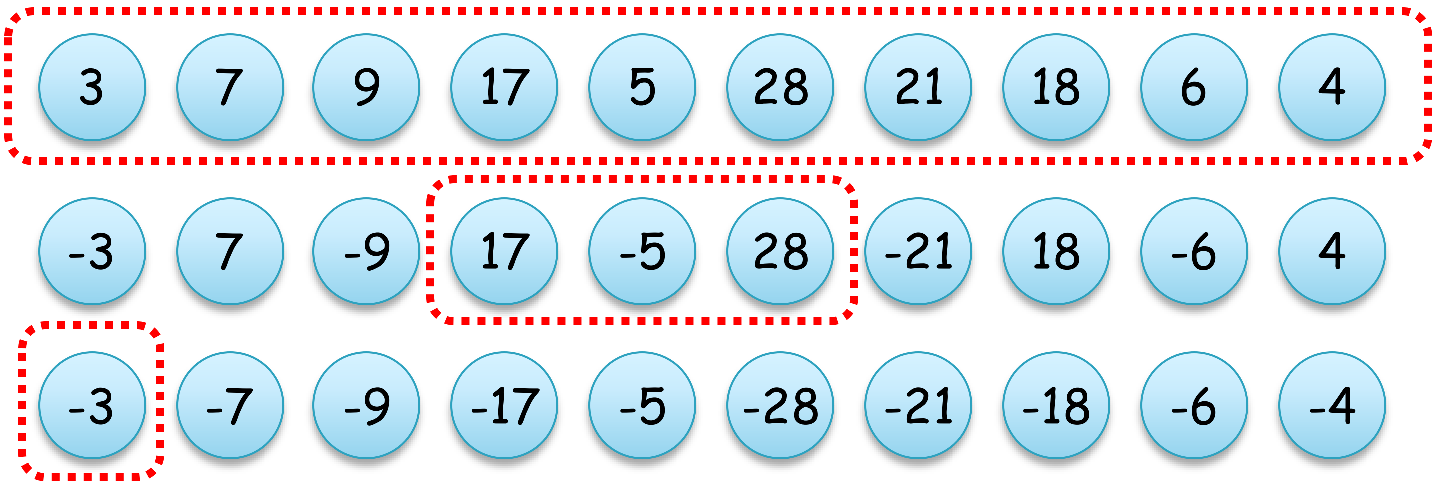
$$T(n) \leq a \cdot T(n/b) + O(n^d)$$

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

- $g(1) = c_0, g(n) \leq 2 \cdot g(n/2) + c_1 \cdot n^{0.5}$
  - 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: $g(n) = O\left(n^d \cdot \log n\right) = O(\sqrt{n} \cdot \log n)$

Maximum subarray problem

- Input: an array of integers A[1], A[2], …, A[n]
- Output: a subarray with the largest sum



Two brute force algorithms

```
MaxSubarray-1(i, j)
  for i = 1,…,n
    for j = i,i+1,…,n
      S[i][j] = A[i] + A[i+1] + … + A[j]

  return Champion(S)
```

$O(n^3)$


```
MaxSubarray-2(i, j)
  R[0] = 0
  for i = 1,…,n
    R[i] = R[i-1] + A[i]

  for i = 1,…,n
    for j = i+1,i+2,…,n
      S[i][j] = R[j] - R[i-1]

  return Champion(S)
```

$O(n^2)$

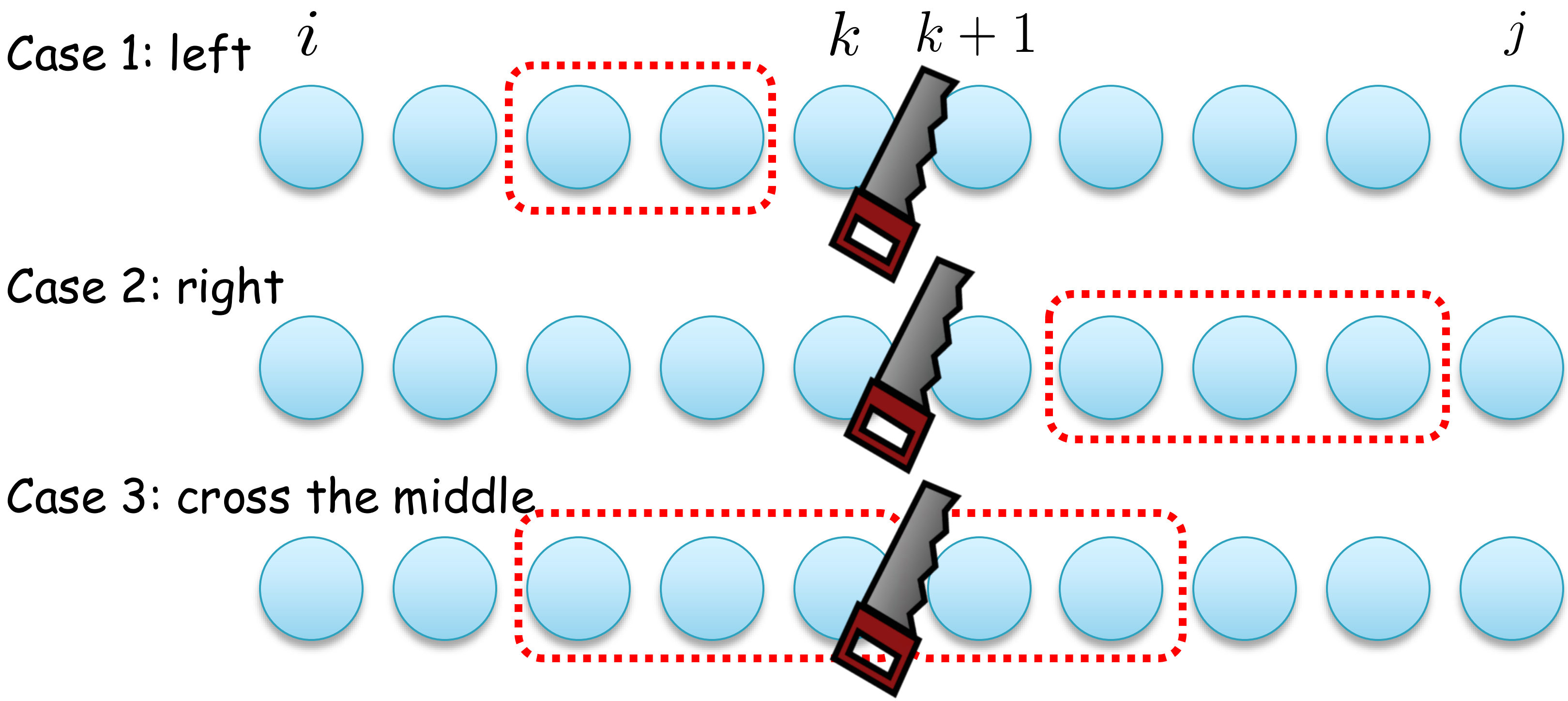
A divide-and-conquer solution

- Base case (n = 1)
  - Return itself (maximum subarray)

- Recursive case (n > 1)
  - Divide the array into two sub-arrays
  - Find the maximum sub-array recursively
  - Merge the results

A divide-and-conquer solution

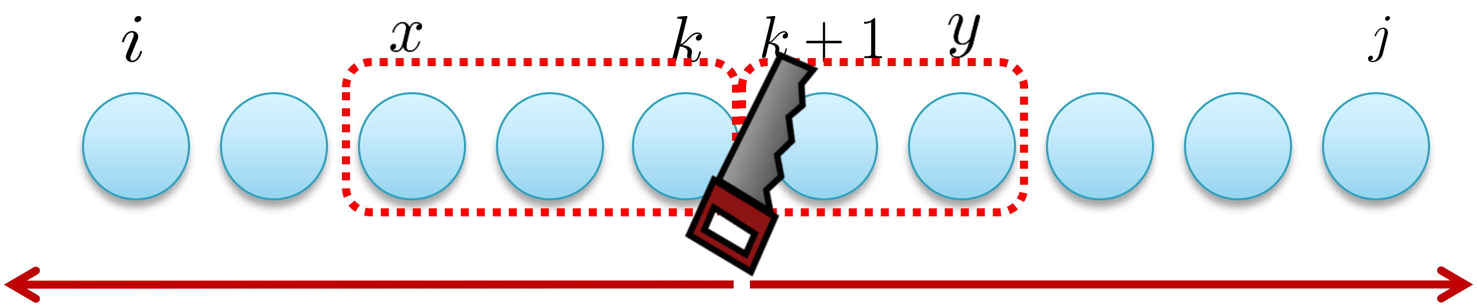
- The maximum subarray for any input must be in one of following cases:



Case 1: MaxSub(A, i, j) = MaxSub(A, i, k)
Case 2: MaxSub(A, i, j) = MaxSub(A, k+1, j)
Case 3: MaxSub(A, i, j) cannot be expressed using MaxSub!

Case 3: cross the middle

- Goal: find the maximum subarray crossing the middle



(1) Start from the middle to find the left maximum subarray

(2) Start from the middle to find the right maximum subarray

The solution of Case 3 is the combination of (1) and (2)

- Observation
  - Left: sum of $A[x \dots k]$ must be the maximum among $A[i \dots k]$
  - Right: sum of $A[k+1 \dots y]$ must be the maximum among $A[k+1 \dots j]$

  Solvable in linear time → $\Theta(n)$

A divide-and-conquer algorithm

```
MaxCrossSubarray(A, i, k, j)
  left_sum = -∞
  sum=0
  for p = k downto i
    sum = sum + A[p]
    if sum > left_sum
      left_sum = sum
      max_left = p

  right_sum = -∞
  sum=0
  for q = k+1 to j
    sum = sum + A[q]
    if sum > right_sum
      right_sum = sum
      max_right = q

  return (max_left, max_right, left_sum + right_sum)
```

$O(k - i + 1)$

$O(j - k)$

$= O(j - i + 1)$

A divide-and-conquer algorithm

```
MaxSubarray(A, i, j)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2
    return (r_low, r_high, r_sum)
  else // case 3
    return (c_low, c_high, c_sum)
```

Divide

Conquer

Combine

A divide-and-conquer algorithm

```
MaxSubarray(A, i, j)
  if i == j // base case                                          O(1)
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)                 T(k - i + 1)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)               T(j - k)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)         O(j - i + 1)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)                                 O(1)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2
    return (r_low, r_high, r_sum)                                 O(1)
  else // case 3
    return (c_low, c_high, c_sum)                                 O(1)
```

Algorithm time complexity

- Divide a list of size *n* into 2 subarrays of size *n/2* $\Theta(1)$
- Recursive case $(n > 1)$ $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$
  - Find **MaxSub** for each subarrays
- Base case $(n = 1)$ $\Theta(1)$
  - Return itself
- Find **MaxCrossSub** for the original list $\Theta(n)$
- Pick the subarray with the maximum sum among 3 subarrays $\Theta(1)$

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases} \Rightarrow T(n) = O(n \log n)$$

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

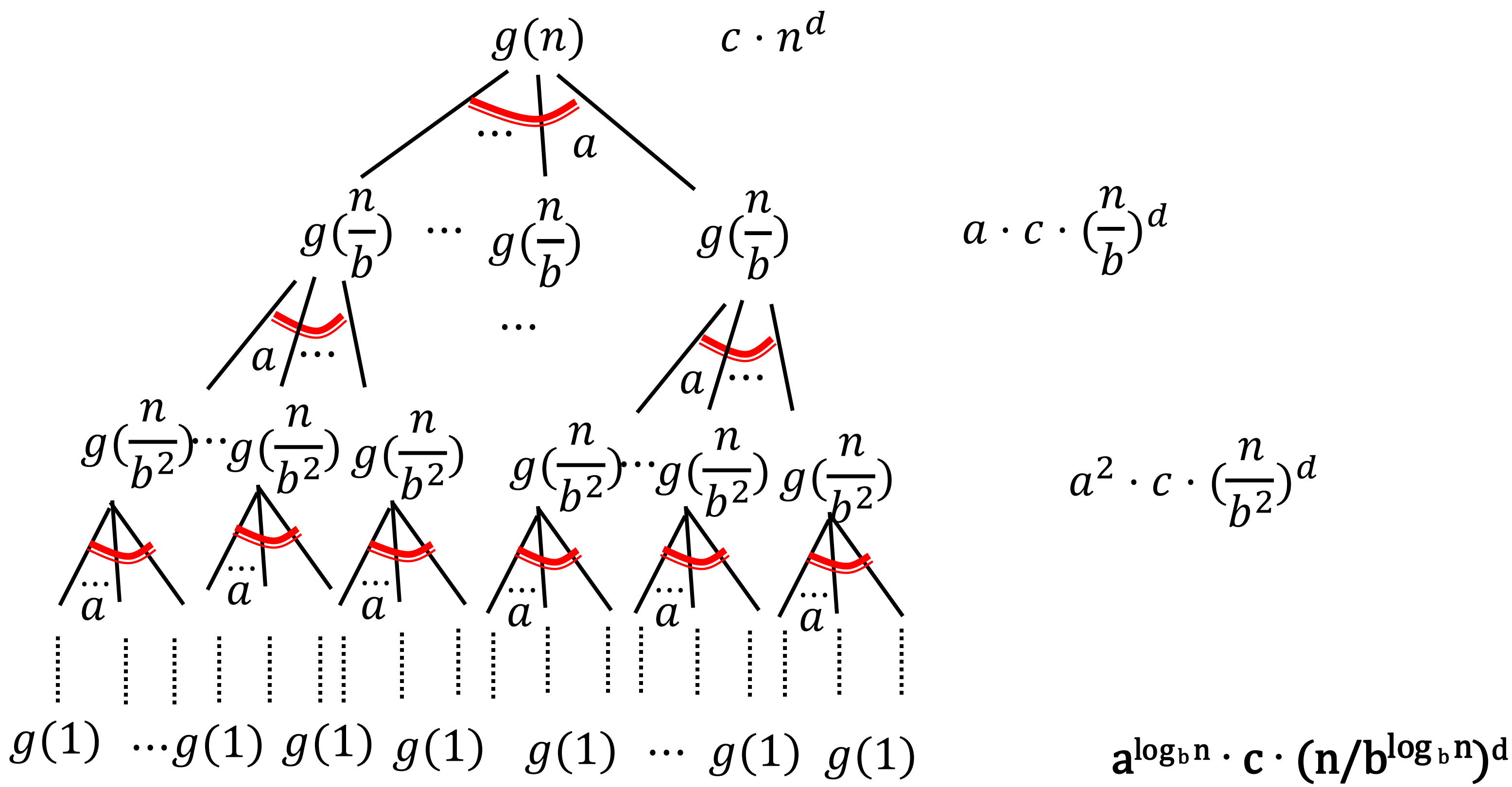
Backup slides

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	$c \cdot (n/b^t)^d$	$\mathbf{a^t \cdot c \cdot (n/b^t)^d}$
		…		
$\log_b n$	$a^{\log_b n}$	$n/b^{\log_b n} = 1$	$c \cdot (n/b^{\log_b n})^d$	$\mathbf{a^{\log_b n} \cdot c \cdot \overbrace{(n/b^{\log_b n})^d}^{\text{This} = 1}}$

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

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: $a = b^d$

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

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

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

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!

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

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

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

Geometric series with the "multiplier" < 1

Case 3: $a > b^d$

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

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

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

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

This is greater than 1!

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

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