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

# CSC3100 Data Structures
# Lecture 5: Complexity analysis with recursion and divide-and-conquer


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

# Counting Basic Operations

***Sum_LinearSearch(A, searchnum, sumestimation)***

Input: array $A$, a search number, and a sumestimation
Output: return 1 if the searchnumber exists in $A$ and the sumestimation 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)` | |
| `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)***

```
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(?)
```

# 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)
```

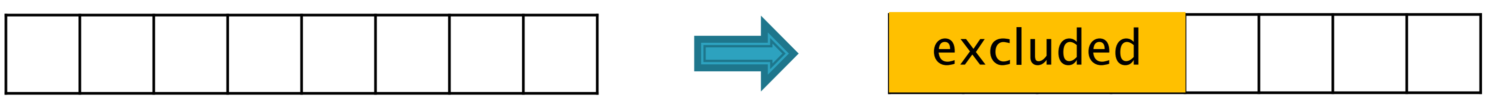
We can still count the number of basic operations for this part

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




$n$ $\qquad\qquad$ $n/2$

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

# 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?

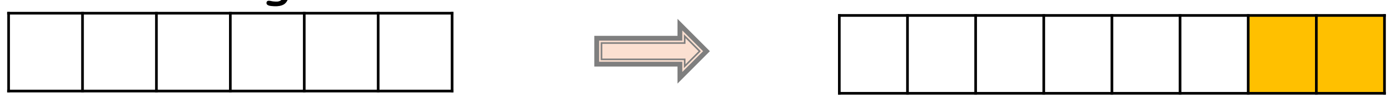
$$\begin{aligned} g(4) &= g(2) + a \\ &= g(1) + a + a \\ &= 2a + b \end{aligned}$$

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

- We will only discuss the big-Oh complexity in the coming lectures

# The Sorting Problem

- Input: a set $S$ of $n$ integers
- Problem: store $S$ in an array such that the elements are arranged in ascending order

| | | | | | |
|---|---|---|---|---|---|
| 4 | 2 | 3 | 6 | 9 | 5 |

⇨

| | | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 9 |

# 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 |

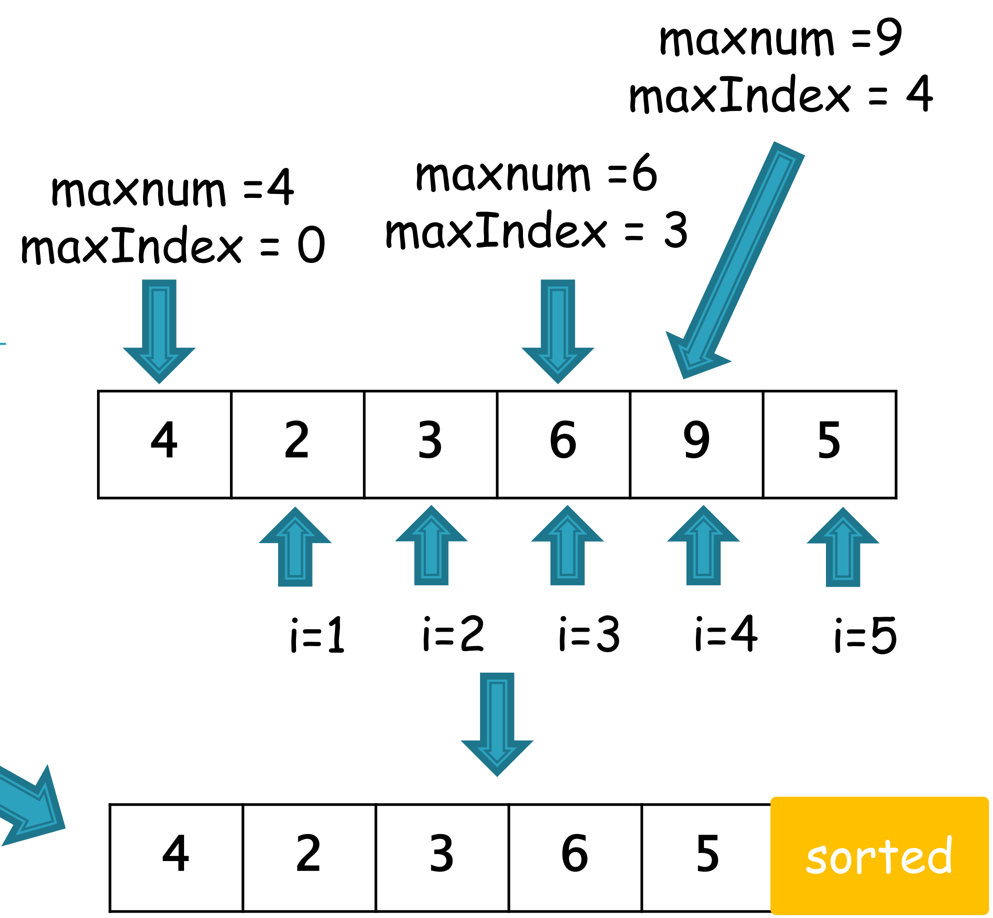
- Step 3: We have a smaller problem: sorting the first n-1 elements

| | | | | | |
|---|---|---|---|---|---|
| 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?
  - $O(n)$

# Analysis for Selection Sort (i)

- 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 the number of basic operations executed by Line 4 is: $g(n-1)$ according to the definition
  - We have that: $g(n) = g(n-1) + O(1)$
  - 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
  - $g(n) = 2g\left(\frac{n}{2}\right) + O(1)$. 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, $g(n) = O(n)$

# Question

- Can we implement the selection sort without using recursion? or in an easier method?
  - If so, how to do it?

# Master Theorem (Big-Oh Version)

- Let $g(n)$ be the running cost depending on the input size $n$, and we have its recurrence:
  - $g(1) = O(1)$
  - $g(n) \leq a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^{\lambda})$
  - With $a, b, \lambda$ to be constants such that $a \geq 1, b > 1, \lambda \geq 0$. Then,
    - If $\log_b a < \lambda$, $g(n) = O(n^{\lambda})$
    - If $\log_b a = \lambda$, $g(n) = O(n^{\lambda} \cdot \log n)$
    - If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$

  - Limitations: cannot be applied to cases like:
    - $g(n) \leq a \cdot g(n-1) + c$

# Master Theorem: Examples

- $g(n) \le a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^\lambda)$
  - If $\log_b a < \lambda$, $g(n) = O(n^\lambda)$
  - If $\log_b a = \lambda$, $g(n) = O(n^\lambda \cdot \log n)$
  - If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$

- $g(1) = c_0, g(n) \le g\left(\left\lceil \frac{n}{2} \right\rceil\right) + c$
  - We have that $a = 1, b = 2, \lambda = 0$
  - Since $\log_b a = \lambda$, we know $g(n) = O(n^0 \cdot \log n) = O(\log n)$

- $g(1) = c_0, g(n) \le g\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_1 \cdot n$
  - We have that $a = 1, b = 2, \lambda = 1$
  - Since $\log_b a < \lambda, g(n) = O\left(n^\lambda\right) = O(n)$

# Master Theorem: Examples

- $g(n) \leq a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^{\lambda})$
  - If $\log_b a < \lambda$, $g(n) = O(n^{\lambda})$
  - If $\log_b a = \lambda$, $g(n) = O(n^{\lambda} \cdot \log n)$
  - If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$

- $g(1) = c_0, g(n) \leq 2 \cdot g\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_1 \cdot n^{0.5}$
  - We have that $a = 2, b = 2, \lambda = 0.5$
  - Since $\log_b a > \lambda$, we have that: $g(n) = O(n^{\log_b a}) = O(n)$

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

# Master Theorem: Practice

- $g(n) \le a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^{\lambda})$
  - If $\log_b a < \lambda$, $g(n) = O(n^{\lambda})$
  - If $\log_b a = \lambda$, $g(n) = O(n^{\lambda} \cdot \log n)$
  - If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$

- 1. $g(1) = c_0, g(n) \le 8 \cdot g\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_1 \cdot n^2$

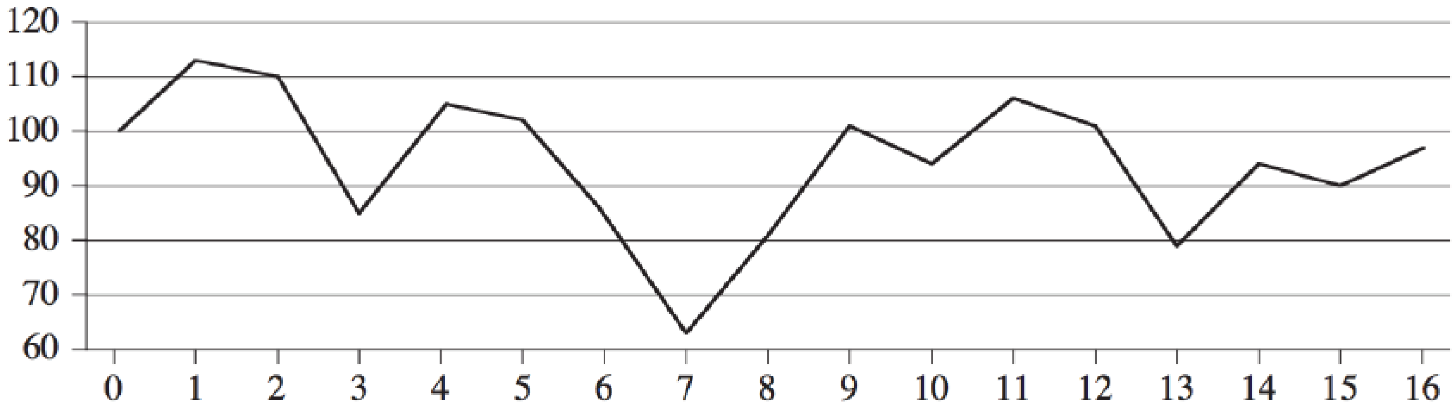
- 2. $g(1) = c_0, g(n) \le 2g\left(\left\lceil \frac{n}{8} \right\rceil\right) + c_1 \cdot n^{\frac{1}{3}}$

- 3. $g(1) = c_0, g(n) \le 2g\left(\left\lceil \frac{n}{4} \right\rceil\right) + c_1 \cdot n$

- 4. Previous practice of MaxInArray2: $g(n) = 2g\left(\frac{n}{2}\right) + O(1), g(1) = b.$
  - Hint: use the fact that $g\left(\frac{n}{b}\right) \le g(\lceil \frac{n}{b} \rceil)$

# Maximum-subarray problem

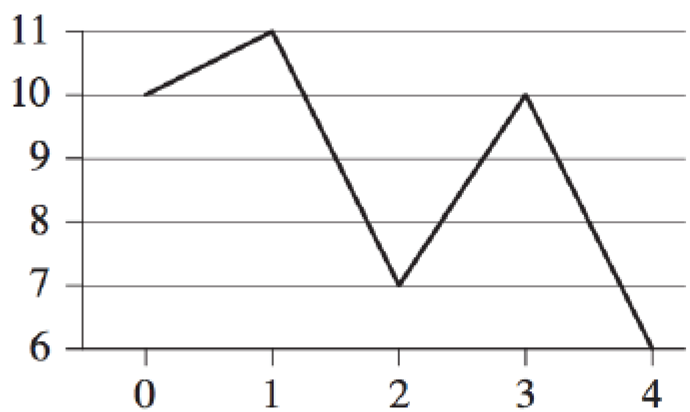
- Consider to buy a stock given the prediction curve - buy low and sell high




| 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 |

# Maximum-subarray problem

- First: choose highest price and go left to find the lowest price
- Second: choose lowest price and go right to find the highest price
- Optimal solution: neither of them




| Day | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Price | 10 | 11 | 7 | 10 | 6 |
| Change | | 1 | −4 | 3 | −4 |

# A brute force solution

- Try every possible pair of buy and sell dates
  - Question: what is the running time of this algorithm for n dates?

- Can we do better? Answer is YES!

# A transformation

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

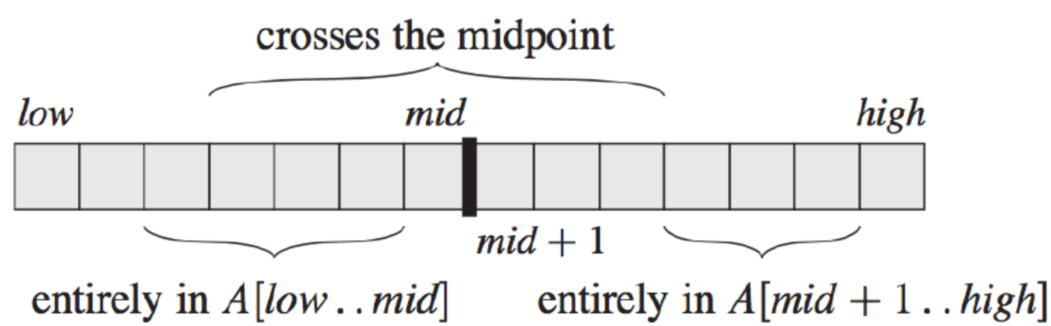
| Day | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| Change | | 13 | −3 | −25 | 20 | −3 | −16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

- Find the nonempty, contiguous subarray of $A$ whose values have the largest sum

- Call this contiguous subarray the maximum subarray

# A divide-and-conquer solution

- Suppose we want to find a maximum subarray of *A[low,…,high]*
- The middle point, *mid=(low+high)/2*
- Suppose *A[i,…,j]* is the maximum 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]* cross *mid*

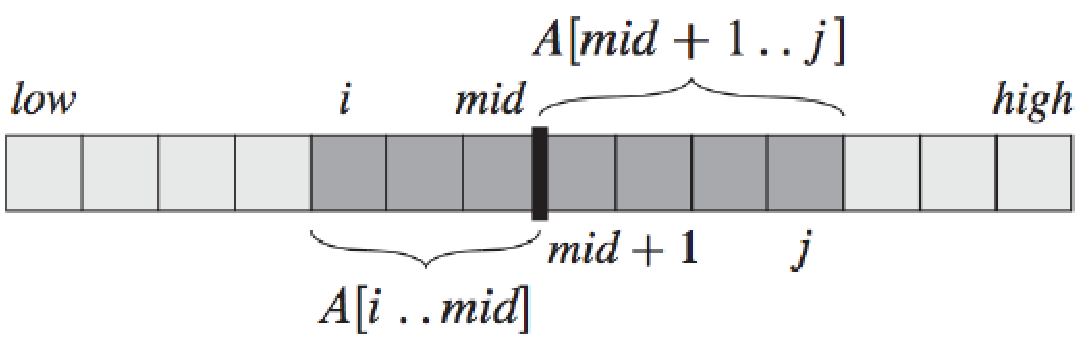


# A divide-and-conquer solution

- Solve the subarrays A[low,…,mid] and A[mid+1,…,high] (two sub-problems) recursively
- 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
- Therefore, we just need to find maximum subarrays of the form A[i,…,mid] and A[mid+1,…,j] and then combine them – which is easy to solve!



# Find max subarray crossing midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Running time $\Theta(n)$

# 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 $T(n)=2T(n/2)+\Theta(n)$

- Same as merge sort $\Theta(n\log n)$
- Faster than brute-force, divide and conquer is very powerful

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

# Recommended reading

- Reading this week
  - Chapter 4, textbook

- Next week
  - Linked List, Stack, and Queue

# Backup slides

# Proof of Master Theorem

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

# Proof of Master Theorem

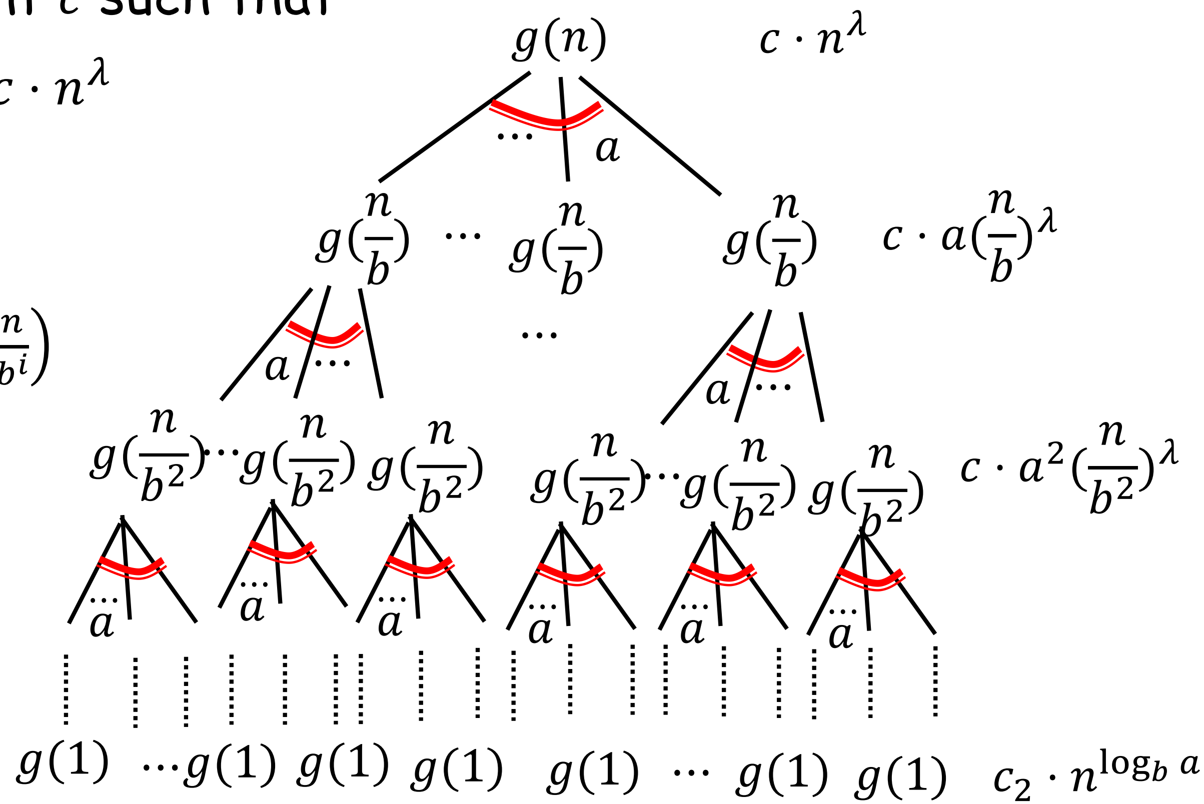
- We first consider the case when $n = b^x$.
  - Since $g(n) \le a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^\lambda)$
  - We can find a constant $c$ such that
    - $g(n) \le a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + c \cdot n^\lambda$

How many $g(1)$?

In level $i$, there are $a^i \; g\left(\frac{n}{b^i}\right)$

$g(1)$ is in level $i$ such that $\frac{n}{b^i} = 1$.

Therefore, there are $a^{\log_b n} \; g(1)$.



# Proof of Master Theorem

- Let $y = \log_b n - 1$
- $g(n) \leq c \cdot n^\lambda + c \cdot a\left(\frac{n}{b}\right)^\lambda + c \cdot a^2\left(\frac{n}{b^2}\right)^\lambda + \cdots + c \cdot a^y\left(\frac{n}{b^y}\right)^\lambda + c_2 \cdot n^{\log_b a}$
- $= c \cdot n^\lambda\left(1 + \frac{a}{b^\lambda} + \left(\frac{a}{b^\lambda}\right)^2 + \cdots \left(\frac{a}{b^\lambda}\right)^y\right) + c_2 \cdot n^{\log_b a}$

- Case 1: $\log_b a < \lambda \Leftrightarrow a < b^\lambda$
  - $g(n) \leq c \cdot n^\lambda \cdot \frac{1}{1-\frac{a}{b^\lambda}} + c_2 \cdot n^{\log_b a}$.
  - $g(n) = O\left(n^\lambda\right)$

- Case 2: $\log_b a = \lambda \Leftrightarrow a = b^\lambda$
  - $g(n) \leq c \cdot n^\lambda \cdot \log_b n + c_2 \cdot n^{\log_b a} = c \cdot n^\lambda \cdot \log_b n + c_2 \cdot n^\lambda$
  - Therefore $g(n) = O(n^\lambda \cdot \log n)$

# Proof of Master Theorem

- Let $y = \log_b n - 1$
- $g(n) \le c \cdot n^\lambda + c \cdot a\left(\frac{n}{b}\right)^\lambda + c \cdot a^2\left(\frac{n}{b^2}\right)^\lambda + \cdots + c \cdot a^y\left(\frac{n}{b^y}\right)^\lambda + c_2 \cdot n^{\log_b a}$
- $= c \cdot n^\lambda\left(1 + \frac{a}{b^\lambda} + \left(\frac{a}{b^\lambda}\right)^2 + \cdots \left(\frac{a}{b^\lambda}\right)^y\right) + c_2 \cdot n^{\log_b a}$

- Case 3: $\log_b a > \lambda \Leftrightarrow a > b^\lambda$
  - $g(n) \le c \cdot n^\lambda \cdot \frac{\left(\frac{a}{b^\lambda}\right)^{y+1} - 1}{\frac{a}{b^\lambda} - 1} + c_2 \cdot n^{\log_b a} = c \cdot n^\lambda \cdot \frac{\left(\frac{a}{b^\lambda}\right)^{\log_b n} - 1}{\frac{a}{b^\lambda} - 1} + c_2 \cdot n^{\log_b a}$
  - $= cn^\lambda \cdot \frac{\frac{a^{\log_b n}}{b^{\lambda \cdot \log_b n}} - 1}{\frac{a}{b^\lambda} - 1} + c_2 \cdot n^{\log_b a} = cn^\lambda \cdot \frac{\frac{n^{\log_b a}}{n^\lambda} - 1}{\frac{a}{b^\lambda} - 1} + c_2 \cdot n^{\log_b a}$
  - $= c \cdot \frac{n^{\log_b a} - n^\lambda}{\frac{a}{b^\lambda} - 1} + +c_2 \cdot n^{\log_b a} \Rightarrow g(n) = O(n^{\log_b a})$

# Proof of Master Theorem

- When $n \neq b^x$. Choose $n' = b^x$ such that $x$ is the smallest integer that $b^x \geq n$.
  - $g(n) \leq g(n')$.
  - It can be verified that the Master Theorem still follows.