EECS 376 Discussion 2

Sec 27: Th 5:30-6:30 DOW 1017

IA: Eric Khiu

Slide deck available at course drive/Discussion/Slides/Eric Khiu

Re(?) Introduction

- Eric **Khiu** (pronounced as Q)
  - Don’t get confused with Eric **Song** (Sec 28 Th 3:30 EECS 1003)
- He/him
- Junior, Math & CS
- From Malaysia (a small country in southeast Asia)
- New here, just took this class last semester
- OH: Tu 5:00-7:30 MLB B116
- E-mail: erickhiu@umich.edu
  - Would be nice if you could start the subject with [EECS 376]

Agenda

- Potential Method
- Divide and Conquer
- Master Theorem
- Worksheet Problems (if time)

Potential Method

Goal: Review the potential function from lecture with more examples

[Lecture Notes]

Potential Method

- We use potential method to show that a complex algorithm **halts** in a **finite** number of steps
- Big idea: Potential function maps the current "*state*" of the algorithm to a nonnegative real number (Still far away from halting? Almost halts? Halt on this iteration?)
- Consider the following while loop. How can we interpret the potential $s = x$?

  while $(x > 0)$ **do**

  print(x)

  $x \leftarrow x - 1$

- Number of iterations left before exiting the while loop

From Halting Condition to Potential Function

- Consider:

  foo(str):

  $i \leftarrow 0$

  while $i < 376$ **do**

  print(str)

  $i \leftarrow i + 1$

- What is the halting condition?
  - $i \geq 376$
- How convert the halting condition to a potential function that have a finite lower bound?
  - Example: $s = 376 - i$

Does s have to decrease every iteration?

- Consider the following algorithm. Is $s = x$ a valid potential function?

  $x \leftarrow 376$

  while $x > 0$ **do**

      if x is even **then**

          $x \leftarrow x + 1$

      else

          $x \leftarrow x - 2$

- Observe that x decrease by 1 **every 2 iterations**
- Takeaway: As long as there is a constant "interval" such that the potential is guaranteed to decrease, it is a valid potential function

Does s have to decrease by 1 on every interval?

- No! It can be 2, 10, 1/2, …
- Consider the following algorithm:

$x \leftarrow 1$

while $x > 0$ **do**

$\quad x \leftarrow x/2$

- x decreases on every iteration, Does the potential $s = x$ prove that it halts in *finite* time?
- If interested, look up Zeno's paradox
- Takeaway: we just need the decrement to be a fixed positive constant

Worksheet Problem 3

For each algorithm, either prove that it must halt by giving a suitable potential function, or give an example sequence of inputs for which the algorithm would run forever.

(a)

$x \leftarrow \text{input}()$
$y \leftarrow \text{input}()$
while $x > 0$ and $y > 0$ **do**
    $z \leftarrow \text{input}()$
    if z is even **then**
        $x \leftarrow x - 1$
        $y \leftarrow y + 1$
    else
        $y \leftarrow y - 1$

(b)

$x \leftarrow \text{input}()$
$y \leftarrow \text{input}()$
while $x > 0$ and $y > 0$ **do**
    $z \leftarrow \text{input}()$
    if z is even **then**
        $x \leftarrow x - 1$
        $y \leftarrow y + 1$
    else
        $y \leftarrow y - 1$
        $x \leftarrow x + 1$

Note: input() returns a user-specified positive integer

Worksheet Problem 3a Solution

(a)

```
x ← input()
y ← input()
while x > 0 and y > 0 do
    z ← input()
    if z is even then
        x ← x − 1
        y ← y + 1
    else
        y ← y − 1
```

Example answer: $s = 2x + y$

- s decreases by 1 on each iteration
  - If z is even: $s = 2x + y \rightarrow 2(x - 1) + (y + 1) = 2x + y - 1$
  - If z is odd: $s = 2x + y \rightarrow 2x + y - 1$
- s cannot be lower than zero
  - When $s = 0$, at least one of x or y must be 0 or less, in which case the function exits the while loop and halts
  - We've shown that s decreases by 1 on every iteration, so it must pass through 0
- s always decreases by 1 and the function halts when $s = 0$, so the function will halt on all inputs

Worksheet Problem 3b Solution

(b)

$x \leftarrow \text{input}()$
$y \leftarrow \text{input}()$
while $x > 0$ and $y > 0$ **do**
    $z \leftarrow \text{input}()$
    if z is even **then**
        $x \leftarrow x - 1$
        $y \leftarrow y + 1$
    else
        $y \leftarrow y - 1$
        $x \leftarrow x + 1$

- Notice that if z alternates between even and odd, then the values of x and y will never go to zero
- The function will not halt in this case
- Example: $x = 376$, $y = 376$, and $z = 0, 1, 0, 1, \ldots$

Divide and Conquer

Goal: Become comfortable with the structure of divide and conquer algorithms and writing the recurrence relation for DC algorithms

Lecture Notes

Divide and Conquer Intro

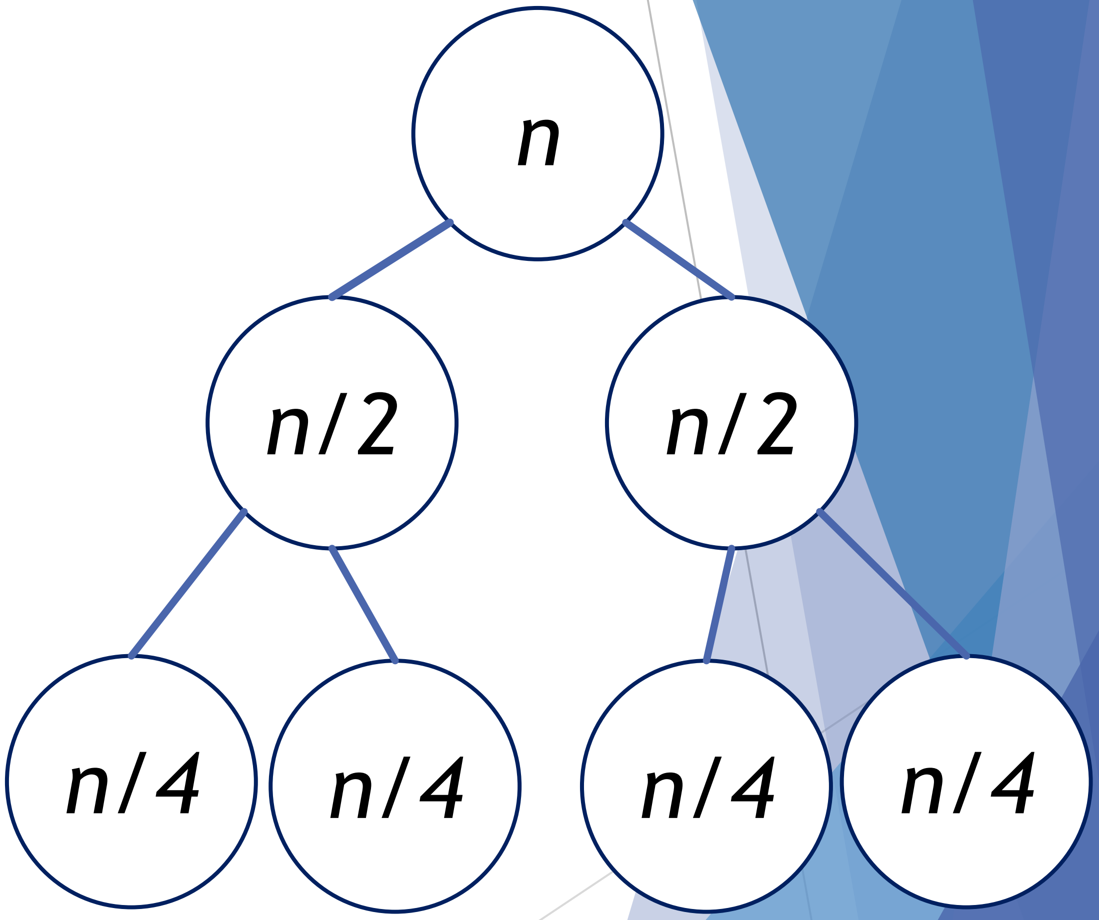
- Big idea:
  - Divide: Divide a problem into smaller versions of the same problem
  - Conquer: Combine the results from those subproblems
- A divide and conquer algorithm usually consists of the following components:
  - Base case
  - Dividing the problems
  - Recursive calls
  - Combining results
- Example: Merge Sort

Algorithm $MergeSort(A[1..n]$: array of n integers) :
  If $n = 1$ **return** A
  $m := \lfloor n/2 \rfloor$
  $L := MergeSort(A[1..m])$
  $R := MergeSort(A[m+1..n])$
  Return $merge(L, R)$

MergeSort: Intuition

Algorithm $MergeSort(A[1..n]$: array of n integers) :
  If $n = 1$ **return** A
  $m := \lfloor n/2 \rfloor$
  $L := MergeSort(A[1..m])$
  $R := MergeSort(A[m+1..n])$
  Return $merge(L, R)$

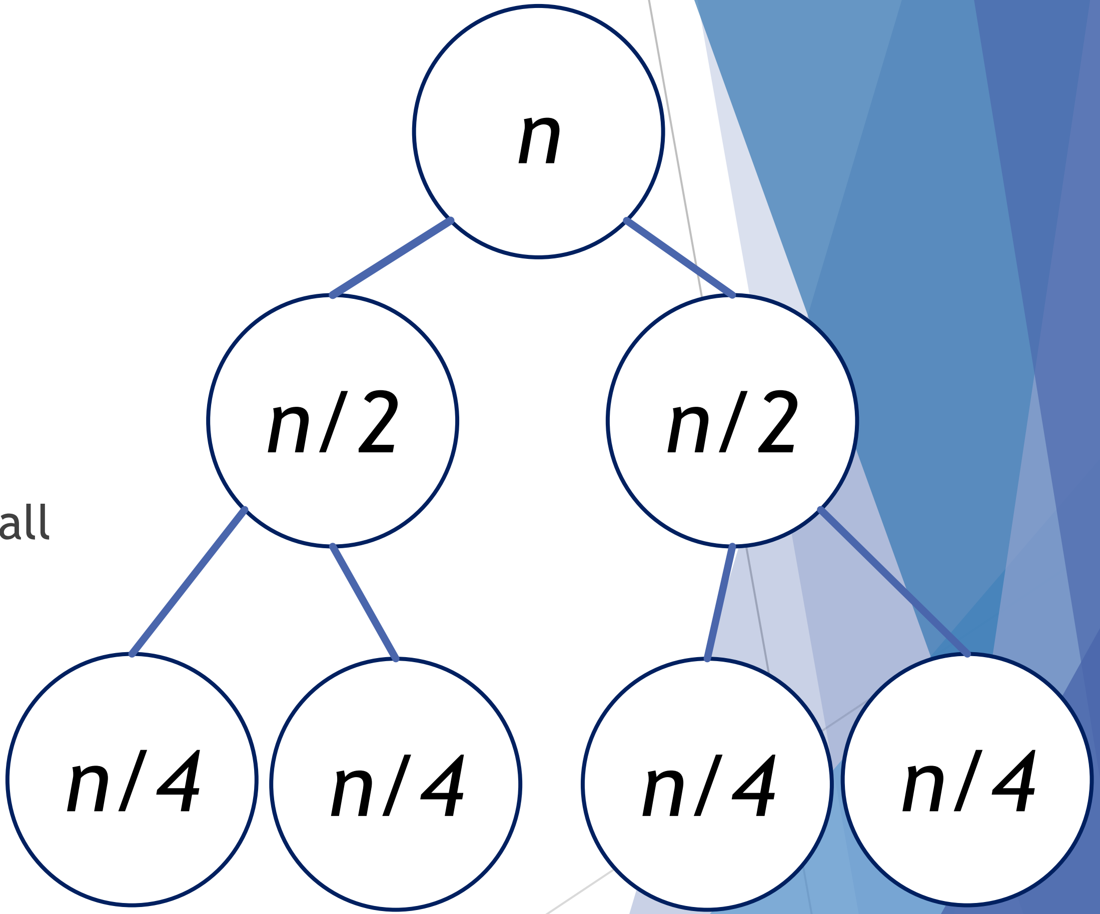
- What can you say about the number of subproblems on each recursive call?
  - Double of the previous
- What about the size of each subproblem?
  - Half of the previous
- Recurrence Relation: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$



MergeSort: Tree Analysis (optional)

Algorithm $MergeSort(A[1..n]$: array of n integers) :
  If $n = 1$ **return** A
  $m := \lfloor n/2 \rfloor$
  $L := MergeSort(A[1..m])$
  $R := MergeSort(A[m+1..n])$
  Return $merge(L, R)$

- Total Runtime = # recursive calls × work per recursive call
- Number of recursive call, d (or the "depth" of the tree)
  - Reach base case when the size of subproblem is 1
  - Size of subproblem is halved every recursive call
  - Solve for $\frac{n}{2^d} = 1 \Rightarrow d = \log_2 n = O(\log n)$
- Work per recursive call: $O(n)$
- Total runtime: $O(n) \cdot O(\log n) = O(n \log n)$



Divide and Conquer: General Form

- Consider an arbitrary divide-and-conquer algorithm that breaks a problem of size n into:
  - k smaller subproblems where $k \geq 1$
  - Each subproblem is of size n/b, where $b > 1$
  - The cost of splitting and combining results is $O(n^d)$ where $d \geq 0$
- This algorithm has the following recurrence

$$T(n) = kT\left(\frac{n}{b}\right) + O(n^d)$$

- Tree analysis is a good tool to analyze the runtime (optional for this class, but good to know!)
- Alternatively, we can apply the Master Theorem for runtime analysis

Master Theorem

Goal: understand the recurrence form required by the Master Theorem, and practice applying it to analyze algorithm complexity

Lecture Notes

Master Theorem

- For the recurrence relation $T(n) = kT\left(\frac{n}{b}\right) + O\left(n^d\right),\ k \geq 1,\ b > 1,\ d \geq 0$

$$T(n) = \begin{cases} O(n^d) & \text{if } k/b^d < 1 \\ O(n^d \log n) & \text{if } k/b^d = 1 \\ O(n^{\log_b k}) & \text{if } k/b^d > 1 \end{cases}$$

- If we replace O with Θ in the recurrence, then the closed form solution is tight
- Master Theorem also holds if the first term is of the form $kT\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$ or $kT\left(\left\lceil\frac{n}{b}\right\rceil\right)$

MergeSort with Master Theorem

Algorithm $MergeSort(A[1..n]$: array of n integers) :
  If $n = 1$ **return** A
  $m := \lfloor n/2 \rfloor$
  $L := MergeSort(A[1..m])$
  $R := MergeSort(A[m+1..n])$
  Return $merge(L, R)$

$$T(n) = \begin{cases} O(n^d) & \text{if } k/b^d < 1 \\ O(n^d \log n) & \text{if } k/b^d = 1 \\ O(n^{\log_b k}) & \text{if } k/b^d > 1 \end{cases}$$

- Our recurrence is $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- $\frac{k}{b^d} = \frac{2}{2^1} = 1$
- By the Master Theorem: $T(n) = O(n \log n)$

Master Theorem with Log Factors

- The Master Theorem generalizes to recurrences with a log factor in the combination term

$$T(n) = kT\left(\frac{n}{b}\right) + \mathrm{O}(n^d \log^w n) \quad \text{where } k \geq 1, b > 1, d \geq 0, w \geq 0.$$

$$T(n) = \begin{cases} \mathrm{O}(n^d \log^w n) & \text{if } k/b^d < 1 \\ \mathrm{O}(n^d \log^{w+1} n) & \text{if } k/b^d = 1 \\ \mathrm{O}(n^{\log_b k}) & \text{if } k/b^d > 1 \end{cases}$$

What happens if w = 0?

Worksheet Problem 2

```
function SlowSort(A[1, 2, ..., n]) // n is length of A
    SlowSort(A[1, ..., ⌊n/2⌋]) // sort both halves of the array recursively
    SlowSort(A[⌊n/2⌋ + 1, ..., n])
    if A[⌊n/2⌋] > A[n] then // largest item in first half is greater than largest in the second
        swap A[⌊n/2⌋] and A[n] // put largest item in the unsorted array at the end
    SlowSort(A[1, ..., n − 1]) // sort the entire array minus one element recursively
    return
```

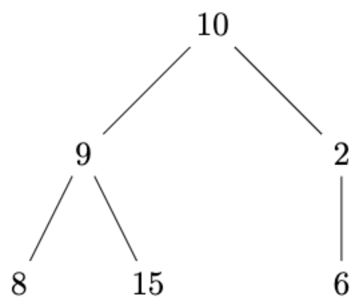
- What is the recurrence relation of SlowSort?

$$2 \cdot T\left(\frac{n}{2}\right) + T(n-1) + O(1)$$

- Can we apply Master Theorem here?
  - No, can't handle $T(n-1)$

Worksheet Problems

Worksheet Problem 5



In this example: 10, 15, and 6 are the local maxima.

5. **Divide and Conquer**

A complete binary tree is a binary tree in which every level, except possibly the last is completely filled and all nodes in the last level are as far left as possible.

Consider a complete binary tree $T = (V, E, r)$ rooted at r where each vertex is labelled with a distinct integer. A vertex $v \in V$ is called a *local maximum* if the label of v is greater than the label of each of its neighbors.

Suppose you are given such a tree where the labelling is given implicitly, i.e., the only way to determine the label of the vertex v is to visit v and query for the vertex label. Provide an algorithm that computes a local maximum of T with using $O(\log(|V|))$ vertex label queries.

Worksheet Problem 5 Intuition

Suppose you are given such a tree where the labelling is given implicitly, i.e., the only way to determine the label of the vertex v is to visit v and query for the vertex label. Provide an algorithm that computes a local maximum of T with using $O(\log(|V|))$ vertex label queries.

- Intuition: Start at the root, check if we're at a local maxima, recurse into a child node if not
- Hint: Consider cases where:
  - Root node does not have any children
  - Root node is greater than both its children
  - Root node is smaller than at least one of its children
- For correctness analysis: How many vertices does the algorithm queries at each level of the tree?

Worksheet Problem 5 Solution

Input: Complete, rooted, vertex labelled, binary tree $T = (V, E, r)$
Output: Local maximum v^*

```
function ComputeLocalMaximum(T = (V, E, r))
    if the label of r is greater than both of its children's or r has no children then
        return r
    else
        Let r' be a child of of r with a label greater than r
        Let T' be the complete, rooted, vertex labelled, binary tree induced by r'
        Compute ComputeLocalMaximum(T' = (V', E', r'))
```

- Correctness
  - The algorithm only recurses into children greater than the root, assuring that the parent is always less than the root under consideration
  - Both children are checked and a node is returned iff it is less than both, so only local maxima will be returned by this function
- $O(\log|V|)$ vertex label queries
  - At each level in the tree, the algorithm queries at most 3 vertices
  - The depth of a complete binary tree is $\log|V|$, so we has $O(\log|V|)$ queries

Worksheet Problem 1: Write a recurrence relation describing the time complexity of MajorityElement and apply the Master Theorem to find a closed-form solution

1. **Divide and Conquer**

   Given an array A of n integers, where n is a power of 2, a *majority element* of A is an element in A that appears strictly more than $\frac{n}{2}$ times. The algorithm MajorityElement defined below finds the majority element of A if it exists. If A has a majority element, MajorityElement will return that element. Otherwise, MajorityElement will return $\emptyset$.

function $\textsc{MajorityElement}(A[1, 2, \ldots, n])$
    if $n = 1$ **then return** $A[1]$
    $x \leftarrow \textsc{MajorityElement}(A[1, \ldots, \lfloor \frac{n}{2} \rfloor])$
    $y \leftarrow \textsc{MajorityElement}(A[\lfloor \frac{n}{2} \rfloor + 1, \ldots, n])$
    if $x \neq \emptyset$ **then**
        iterate over A, counting the number of occurrences of x
        if the number of occurrences of x in A is $> \frac{n}{2}$ **then return** x
    if $y \neq \emptyset$ **then**
        iterate over A, counting the number of occurrences of y
        if the number of occurrences of y in A is $> \frac{n}{2}$ **then return** y
    return $\emptyset$

$$T(n) = kT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d) & \text{if } k/b^d < 1 \\ O(n^d \log n) & \text{if } k/b^d = 1 \\ O(n^{\log_b k}) & \text{if } k/b^d > 1 \end{cases}$$

Worksheet Problem 1a Solution

function $\textsc{MajorityElement}(A[1,2,\ldots,n])$
    if $n = 1$ **then return** $A[1]$
    $x \leftarrow \textsc{MajorityElement}(A[1,\ldots,\lfloor \frac{n}{2} \rfloor])$
    $y \leftarrow \textsc{MajorityElement}(A[\lfloor \frac{n}{2} \rfloor + 1,\ldots,n])$
    if $x \neq \emptyset$ **then**
        iterate over A, counting the number of occurrences of x
        if the number of occurrences of x in A is $> \frac{n}{2}$ **then return** x
    if $y \neq \emptyset$ **then**
        iterate over A, counting the number of occurrences of y
        if the number of occurrences of y in A is $> \frac{n}{2}$ **then return** y
    return $\emptyset$

- $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- $k = 2, b = 2, d = 1$
- Master theorem gives us $O(n \log n)$

$$T(n) = kT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d) & \text{if } k/b^d < 1 \\ O(n^d \log n) & \text{if } k/b^d = 1 \\ O(n^{\log_b k}) & \text{if } k/b^d > 1 \end{cases}$$

Worksheet Problem 1b

(b) Show the correctness of the MAJORITYELEMENT algorithm by proving the following statement:

If z is a majority element of an array A of n integers, then z must be a majority element of at least one of the subarrays $A[1, \ldots, \frac{n}{2}]$ and $A[\frac{n}{2}+1, \ldots, n]$.

Majority Element: appears strictly more than $\frac{n}{2}$ times in an array of size n

Worksheet Problem 1b Solution

(b) Show the correctness of the MAJORITYELEMENT algorithm by proving the following statement:
If z is a majority element of an array A of n integers, then z must be a majority element of at least one of the subarrays $A[1, \ldots, \frac{n}{2}]$ and $A[\frac{n}{2} + 1, \ldots, n]$.

- Let z be some element of A, and for the sake of contradiction, assume z is neither a majority element of $A\left[1, \ldots, \frac{n}{2}\right]$ nor $A\left[\frac{n}{2} + 1, \ldots, n\right]$
- If it is not a majority of $A\left[1, \ldots, \frac{n}{2}\right]$, it must occur $\leq \frac{n}{2} \cdot \frac{1}{2} = \frac{1}{4}$ times
- We can apply the same logic to $A\left[\frac{n}{2} + 1, \ldots, n\right]$, so the total occurrences of z are at most $\frac{n}{4} + \frac{n}{4} = \frac{n}{2}$
- This is a contradiction, as we've assumed z to be a majority element
- We conclude that for z to be a majority element of A, it must be a majority element of at least $A\left[1, \ldots, \frac{n}{2}\right]$ or $A\left[\frac{n}{2} + 1, \ldots, n\right]$