D2: Potential Method and Divide & Conquer

Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

Agenda

- The Potential Method
- Divide and Conquer
- Master Theorem

The Potential Method

Notes

Starter: Halting

- Consider the following algorithms:

A

```
x ← INPUT()
while (x > 0) do
    print(x)
    x ← x − 2
```

C

```
x ← INPUT()
while (x < 376) do
    print(x)
    x ← x + 1
```

B

```
x ← INPUT()
while (x < 0) do
    print(x)
    x ← x − 1
```

D

```
x ← INPUT()
while (x > 0) do
    print(x)
    if x is odd then x ← x + 1
    else x ← x − 2
```

Poll: Which of the algorithm(s) are **guaranteed** to halt **regardless of the input?**

Potential Method

- The **Potential Method** is a useful technique to reason about the number of steps required to run a complex algorithm
- A **potential function** maps the current "*state*" of the algorithm to a nonnegative real number
- We can use potential functions to
  - Analyze whether a program will halt
  - Analyze the time complexity of an algorithm (seen in lecture for Euclid's algorithm)

Potential Method and Halting

A

```
x ←INPUT()
while (x > 0) do
    print(x)
    x ← x − 2
```

- Let $s_i = x$ (the value of x on the i-th iteration)
- The program halts when $s_i = 0$
- Since $s_{i+1} = s_i - 2$, $s_{i+1} < s_i$ for all i and will eventually reach 0
- Therefore, algorithm A will eventually halt

- **Fact:** An algorithm will halt if there exists a decreasing potential function that has a finite lower bound

But wait! Wait if it's increasing?

Fact: An algorithm will halt if there exists a decreasing potential function that has a finite lower bound.

Ambiguous!

C

```
x ←INPUT()
while (x < 376) do
    print(x)
    x ← x + 1
```

- Let $s_i = 376 - x$
- The program halts when $s_i = 0$
- Since $s_{i+1} = s_i - 1$, $s_{i+1} < s_i$ for all i and will eventually reach 0
- Therefore, algorithm C will eventually halt

Discuss: Does s has to decrease on **every** iteration?

Decreasing on Constant Interval

Fact: An algorithm will halt if there exists a decreasing potential function that has a finite lower bound.

Ambiguous!

D

```
x ←INPUT()
while (x > 0) do
    print(x)
    if x is odd then x ← x + 1
    else x ← x − 2
```

- Observe that x decrease by 1 every 2 iterations
- Instead of having 1 "interval" = 1 iteration, we define 1 "interval" = 2 iterations
- Let $s_{2i} = x$ (keep track of value of x every two iterations)
- Finite lower bound = 0
- $s_{2(i+1)} < s_{2i}$ for all $i \Rightarrow$ strictly decreasing

TL; DPA

- We discussed how potential method can be used in halting analysis
- If there exists a potential function that

See back matter

1. strictly decreases by at least some fixed constant c in each constant interval, and
2. is lower-bounded by some fixed value

then the algorithm will eventually halt.

Worksheet Problem 1.1 (if time)

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)

$x \leftarrow$ `INPUT()`

$y \leftarrow$ `INPUT()`

While $x > 0$ **and** $y > 0$ **do**

    $z \leftarrow$ `INPUT()`

    if z is even **then**

        $x \leftarrow x - 1$

        $y \leftarrow y + 1$

    else

        $y \leftarrow y - 1$

(b)

$x \leftarrow$ `INPUT()`

$y \leftarrow$ `INPUT()`

While $x > 0$ **and** $y > 0$ **do**

    $z \leftarrow$ `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 1.1(a) 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: $s = 2x + y \to 2(x-1) + (y+1) = 2x + y - 1$
  - If z is odd: $s = 2x + y \to 2x + y - 1$
- 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 ← 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
        x ← 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

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

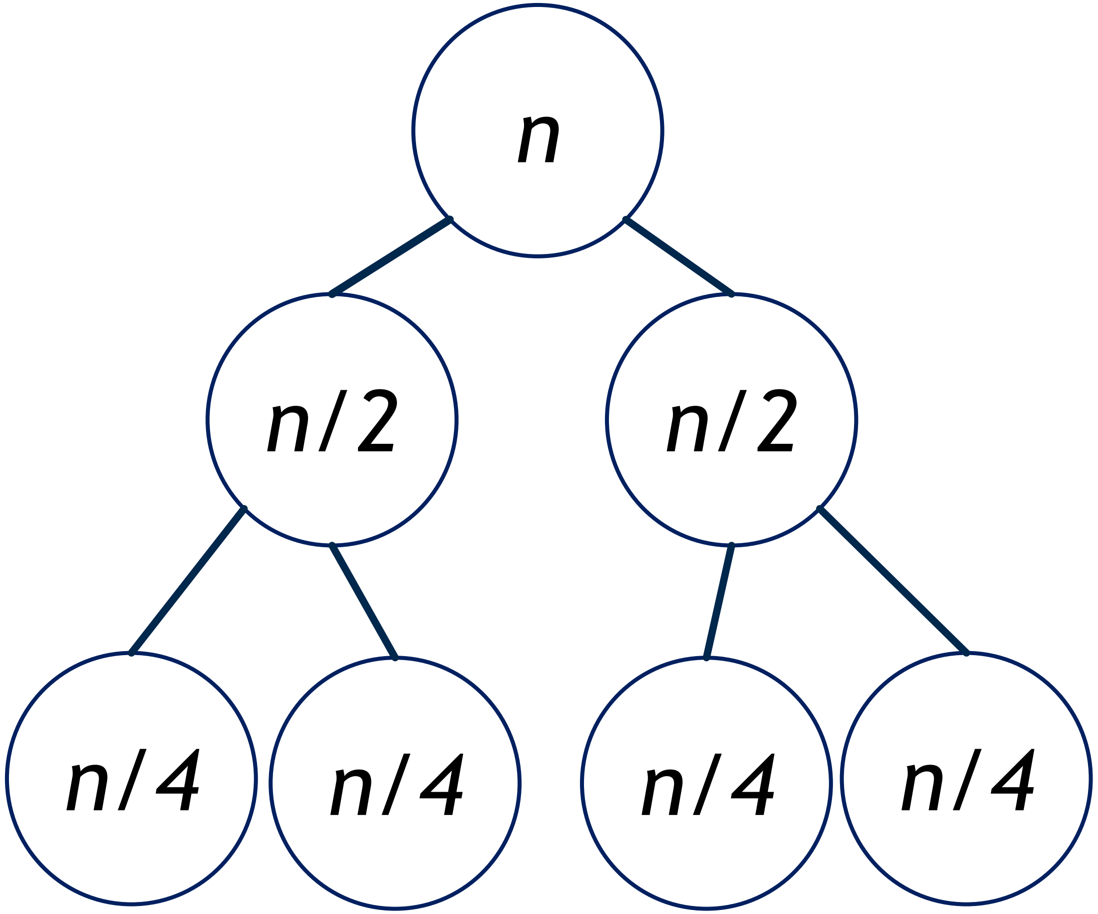
```
MERGESORT (A[1, ..., n]):
    if n = 1 then return A
    m ← ⌊n/2⌋
    L ← MERGESORT(A[1, ..., m])
    R ← MERGESORT(A[m + 1, ..., n])
    return MERGE(L, R) //O(n)
```

Visualizer

MergeSort: Intuition

```
MERGESORT (A[1,...,n]):
    if n = 1 then return A
    m ← ⌊n/2⌋
    L ←MERGESORT(A[1,...,m])
    R ←MERGESORT(A[m+1,...,n])
    return MERGE(L,R) //O(n)
```

- **Q:** 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: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$



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

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

See back matter

- 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

Divide and Conquer Correctness Proof

- Similar idea to prove by induction

Prove by Induction	Correctness Proof for D&C
Prove that $P(0)$ is true	Prove that the base case(s) is/ are correct
Assuming $P(k)$ is true, prove $P(k+1)$ is true	Assuming recursive calls on smaller inputs return correct answer, prove that the current call is correct Extra: Briefly justify you are making recursive calls under correct <u>condition</u> and with correct <u>input</u>

TL; DPA

- We went through the key elements in a divide and conquer algorithm.
- We looked at the general form of divide and conquer in form of recurrence relation.
- We compared proof by induction with the correctness proof for divide and conquer.

Master Theorem

Notes

Master Theorem

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

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

- **Remark:** 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 $kT\left(\left\lceil\frac{n}{b}\right\rceil\right)$
- **Example:** For merge sort we have $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
  - $\frac{k}{b^d} = \frac{2}{2^1} = 1$
  - 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

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

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

WS #3.2: SlowSort

Consider the following sorting algorithm SLOWSORT

```
SLOWSORT (A[1, …, n]):
 m ← ⌊n/2⌋
 SLOWSORT(A[1, …, m])
 SLOWSORT(A[m + 1, …, n])
 if A[m] > A[n] then
     swap A[m] and A[n]
 SLOWSORT(A[1, …, n − 1])
```

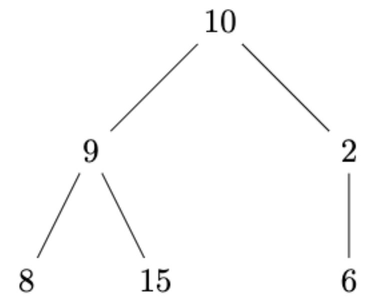
- What is the recurrence relation of SLOWSORT?
  - $2 \cdot T\left(\frac{n}{2}\right) + T(n-1) + O(1)$
- **Q:** Why can't we apply Master Theorem here?
  - Master Theorem can't handle $T(n-1)$

TL; DPA

- We looked at the Master Theorem- an extremely useful tool to analyze runtime of recursion
- It is important to first make sure that Master Theorem is applicable before applying
  - Write the recurrence relation in the form $T(n) = kT\left(\frac{n}{b}\right) + O\left(n^d\right)$ or $T(n) = kT\left(\frac{n}{b}\right) + O\left(n^d \log^w n\right)$
  - Also check the constraints on $k, b, d,$ and w!

Worksheet Problems

WS #2.3: Binary Tree Local Maximum



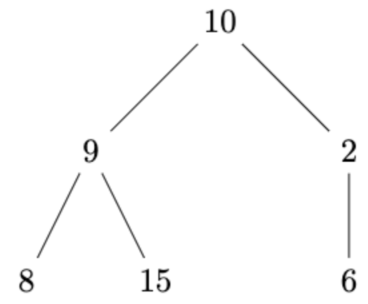
In this example: 10, 15, and 6 are the local maxima.

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 a *local maximum* if the label of v is greater than the label of each of its parent **and** children.

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 returns *a* local maximum of T using $O(\log|V|)$ vertex label queries.

WS #2.3: Binary Tree Local Maximum



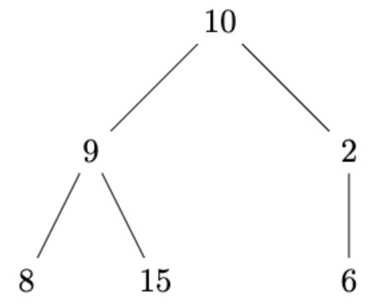
In this example: 10, 15, and 6 are the local maxima.

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 returns *a* local maximum of T using $O(\log|V|)$ vertex label queries.

- Consider:
  - What are the base case(s)?
  - How to divide the problem?
  - When to make the recursive calls?
  - What is the input to the recursive call?

Hint: Consider cases where:
- Root node is greater than both its children
- Root node is smaller than at least one of its children

WS #2.3: Binary Tree Local Maximum



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

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 returns *a* local maximum of T using $O(\log|V|)$ vertex label queries.

- Consider:
  - What are the base case(s)? Only has one node
  - How to divide the problem? Start at root node, check if it's local maxima, recurse into children if not
  - When to make the recursive calls?
  - What is the input to the recursive call?

Hint: Consider cases where:
- Root node is greater than both its children Return root node
- Root node is smaller than at least one of its children Recurse into that child

WS #2.3 Solution

Draft:
Root has no children ⇒ Return root // Base case
Root > both children ⇒ Return root
Root < at least one child ⇒ Recurse into that child

```
CBTLocalMax(T = (V, E, r)):
    if r has no children then return r // Base case
    else if label of r is greater than both its children's then return r
    else
        r' ←child of r with label greater than r
        T' ←complete binary tree rooted at r'
        return CBTLocalMax(T' = (V', E', r')):
```

- Correctness Analysis
  - Base case: If there is only one node, then it is the local maximum
  - If r is greater than both its children, then it is a local maximum and is returned correctly
  - Suppose the algorithm returns a local maximum of a CBT of depth k, since we only recurses into children greater than the root, the parent is always less than the root under consideration. Therefore, by IH, the algorithm returns a local maximum of a CBT of depth $k+1$.

WS #2.3 Solution

Draft:
Root has no children ⇒ Return root // Base case
Root > both children ⇒ Return root
Root < at least one child ⇒ Recurse into that child

```
CBTLocalMax(T = (V, E, r)):
    if r has no children then return r // Base case
    else if label of r is greater than both its children's then return r
    else
        r' ←child of r with label greater than r
        T' ←complete binary tree rooted at r'
        return CBTLocalMax(T' = (V', E', r')):
```

- $O(\log|V|)$ vertex label queries
  - At each level, the algorithm queries at most 3 vertices
  - Depth of a complete binary tree is $\log|V|$, so we has $O(\log|V|)$ queries

WS #2.2: Array Local Minimum

Let $A[1, \ldots, n]$ be an array of n distinct integers, where $n = 2^k$ for some $k \in \mathbb{N}$. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a *local minimum* if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

Devise a divide and conquer algorithm to find *a* local minimum in this array in $O(\log n)$ time.

- Consider:
  - What are the base case(s)?
  - How to divide the problem?
  - When to make the recursive calls?
  - What is the input to the recursive call?

WS #2.2: Array Local Minimum

Let $A[1, \ldots, n]$ be an array of n distinct integers, where $n = 2^k$ for some $k \in \mathbb{N}$. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a *local minimum* if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

Devise a divide and conquer algorithm to find a local minimum in this array in $O(\log n)$ time.

- Consider:
  - What are the base case(s)? Array size = 1
  - How to divide the problem? Start at the middle, check if its local min, recurse into left/ right subarray if not
  - When to make the recursive calls? If at least one neighbor is smaller than the middle element of the array
  - What is the input to the recursive call? If middle > left neighbor: left subarray
    If middle > right neighbor: right subarray

WS #2.2 Solution

Draft:
Array size = 1 ⇒ Return that element // Base case
Middle < both neighbor ⇒ Return middle
Middle > left neighbor ⇒ Recurse into left subarray
Middle > right neighbor ⇒ Recurse into right subarray

```
ArrLocalMin(A = [1, …, n]):
    if n = 1 then return A[1] // Base case
    m ← ⌊n/2⌋
    if A[m] < A[m − 1] and A[m] < A[m + 1] then return A[m]
    else if A[m] > A[m − 1] then
        return ArrLocalMin(A[1, …, m − 1])
    else
        return ArrLocalMin(A[m + 1, …, n])
```

Runtime analysis:

- $T\left(\frac{n}{2}\right) + O(1)$
- Master Theorem says $O(\log n)$

- Correctness Analysis:
  - Base case: If there is only one element in the array, then it is the local minimum
  - If $A[m] < A[m-1]$ and $A[m] < A[m+1]$, then $A[m]$ is a local minimum and is correctly returned
  - Suppose the algorithm correctly return the local minimum of an array of size $n/2$. Since we only consider subarrays where the element after the right/left end is larger, the solution to a subarray is also the solution to the whole array.

WS #2.1: Majority Elements

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 that $\frac{n}{2}$ times. The algorithm MajorityElement defined below finds the majority element of A if it exists, or return $\emptyset$ otherwise.

```
MajorityElement(A[1,...,n]):
    if n = 1 then return A[1]
    m ← ⌊n/2⌋
    x ←MajorityElement(A[1,...,m])
    y ←MajorityElement(A[m+1,...,n])
    if x ≠ ∅ then
        c ←count(x,A) //count occurrence of x in A
        if c > n/2 then return x
    if y ≠ ∅ then
        c ←count(y,A) //count occurrence of y in A
        if c > n/2 then return y
    return ∅
```

What is the recurrence relation of MajorityElement?

- $T(n) = 2T(n/2) + O(n)$

WS #2.1: Majority Elements Proof

Show the correctness of the algorithm by proving the following statement:

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

- Let z be some element of A, and for the sake of contradiction, assume z is neither a majority element of $A\left[1, \dots, \frac{n}{2}\right]$ nor $A\left[\frac{n}{2}+1, \dots, n\right]$
- If it is not a majority of $A\left[1, \dots, \frac{n}{2}\right]$, it must occur $\leq \frac{n}{2} \cdot \frac{1}{2} = \frac{1}{4}$ times
- We can apply the same logic to $A\left[\frac{n}{2}+1, \dots, n\right]$, 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 $A\left[1, \dots, \frac{n}{2}\right]$ or $A\left[\frac{n}{2}+1, \dots, n\right]$

Back Matter

Potential Method: Lower Bound on Decrement

- We established earlier that if there exists a potential function that
  1. strictly decreases by at least some fixed constant c in each constant interval, and
  2. is lower-bounded by some fixed value

  then the algorithm will eventually halt.
- What is the significance of having a lower bound on the decrement? Consider

$x \leftarrow 1$

while $x > 0$ **do**

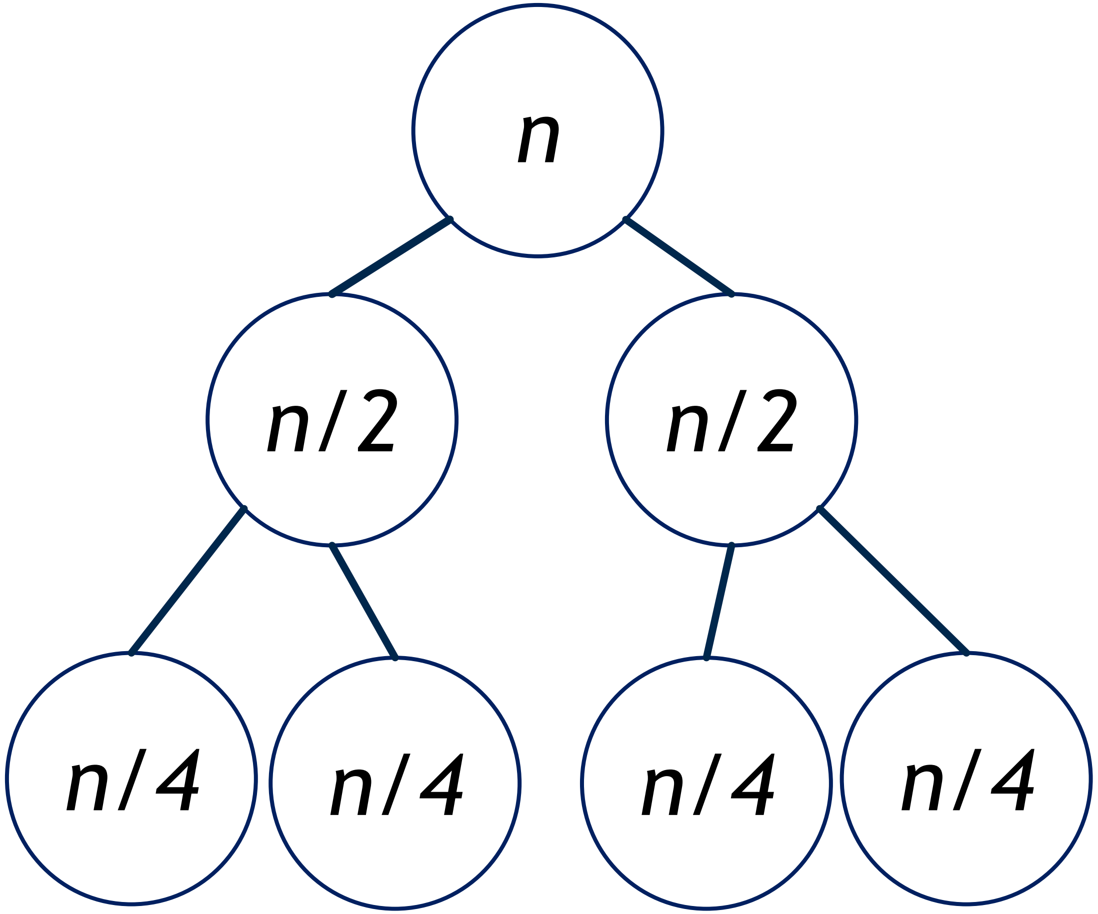
$\quad x \leftarrow x/2$

- x *does* decrease on every iteration, but the potential $s = x$ does not prove that the algorithm halts in *finite* time (Well, practically we will end up in arithmetic underflow so yeah it will halt)
- If interested, look up Zeno's paradox

MergeSort: Tree Analysis (optional)

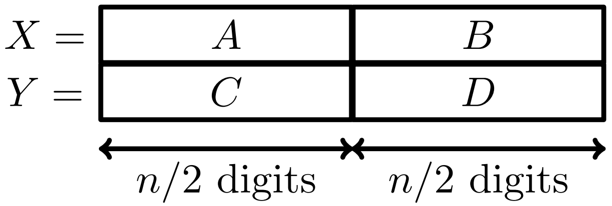
```
MERGESORT (A[1, …, n]):
    if n = 1 then return A
    m ← ⌊n/2⌋
    L ←MERGESORT(A[1, …, m])
    R ←MERGESORT(A[m + 1, …, n])
    return MERGE(L, R) //O(n)
```

- 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: $O(n) \cdot O(\log n) = O(n \log n)$



Karatsuba Algorithm: Big Idea

- We want to multiply two numbers X and Y. Each has n digits. Naïve way: $O(n^2)$
- Rewrite X and Y as follows:
  - $X = A \cdot 10^{\frac{n}{2}} + B$
  - $Y = C \cdot 10^{\frac{n}{2}} + D$



- Expand $X \cdot Y$ as follows:
  - $X \cdot Y = \left(A \cdot 10^{\frac{n}{2}} + B\right)\left(C \cdot 10^{\frac{n}{2}} + D\right) = AC \cdot 10^n + (AD + BC) \cdot 10^{\frac{n}{2}} + BD$
- Observation: The multiplications AC, AD, BC, BD are smaller versions of the original problem- we can use Divide and Conquer!
- **Karatsuba Algorithm:** Clever "preparations" to make the conquer step faster

Karatsuba Algorithm

- The Karatsuba Algorithm for Decimal Multiplication is as follows:
- **Q:** What are the “clever preparations”?
  - $M_1 = AC$
  - $M_2 = BD$
  - $M_3 = (A+B)(C+D)$
- Remember we wanted $AC \cdot 10^n + (AD+BC) \cdot 10^{\frac{n}{2}} + BD$
  - Algebra says $M_3 - M_1 - M_2 = AD + BC$
- Recurrence: $3T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right) = O(n^{1.585})$
  - Better than $O(n^2)$

```
KARATSUBA(x, y): //Assume x ≥ y
    n ←number of digits of x
    if n = 1 then return x · y
    write x as a · 10^{n/2} + b
    write y as c · 10^{n/2} + b
    M_1 ←Karatsuba(a, c)
    M_2 ←Karatsuba(b, d)
    M_3 ←Karatsuba(a + b, c + d)
    return M_1 · 10^n + (M_3 − M_1 − M_2) · 10^{n/2} + M_2
```

Karatsuba Algorithm: Exercise

```
KARATSUBA(x,y): //Assume x ≥ y
    n ←number of digits of x
    if n = 1 then return x · y
    write x as a · 10^{n/2} + b
    write y as c · 10^{n/2} + b
    M_1 ←Karatsuba(a,c)
    M_2 ←Karatsuba(b,d)
    M_3 ←Karatsuba(a + b,c + d)
    return M_1 · 10^n + (M_3 − M_1 − M_2) · 10^{n/2} + M_2
```

Compute 37×76

- $n = 4$ (number of digits)
- $A = 3, B = 7, C = 7, D = 6$

- $M_1 = AC = 3 \cdot 7 = 21$
- $M_2 = BD = 7 \cdot 6 = 42$
- $M_3 = (A + B)(C + D) = (3 + 7)(7 + 6) = 130$

- $37 \times 76 = 21 \cdot 10^2 + (130 - 21 - 42) \times 10 + 42 = 2100 + 67 + 42 = 2812$