BBM402-Lecture 2: Recursion: linear-time selection, Karatsuba multiplication

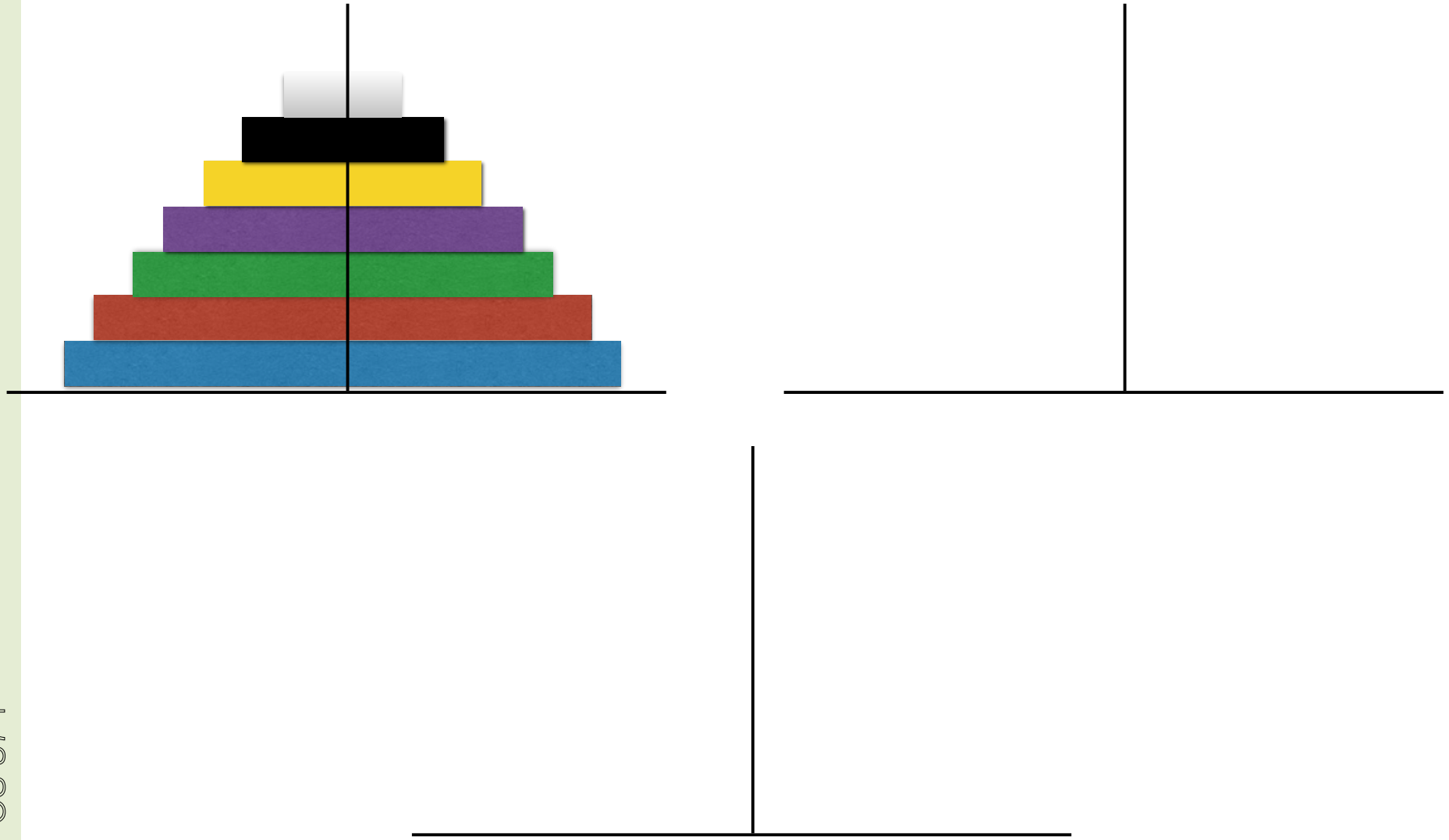
Lecturer: Lale Özkahya

Resources for the presentation:
https://courses.engr.illinois.edu/cs374/fa2016/lectures.html
https://courses.engr.illinois.edu/cs374/fa2015/lectures.html

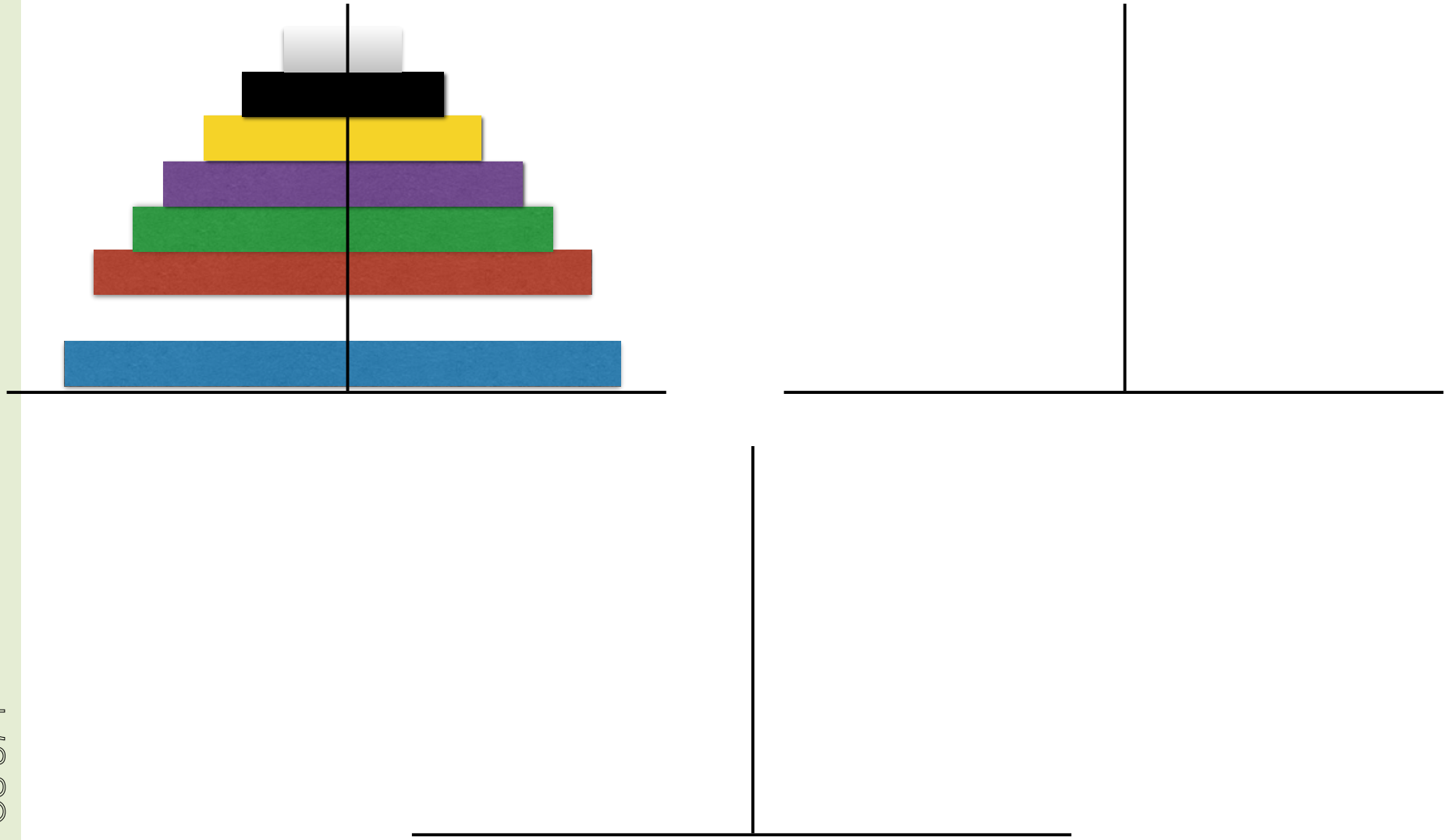
What is Recursion?

Tower of Hanoi: Move the tower from one peg to another without ever putting a larger block on top of a smaller one.

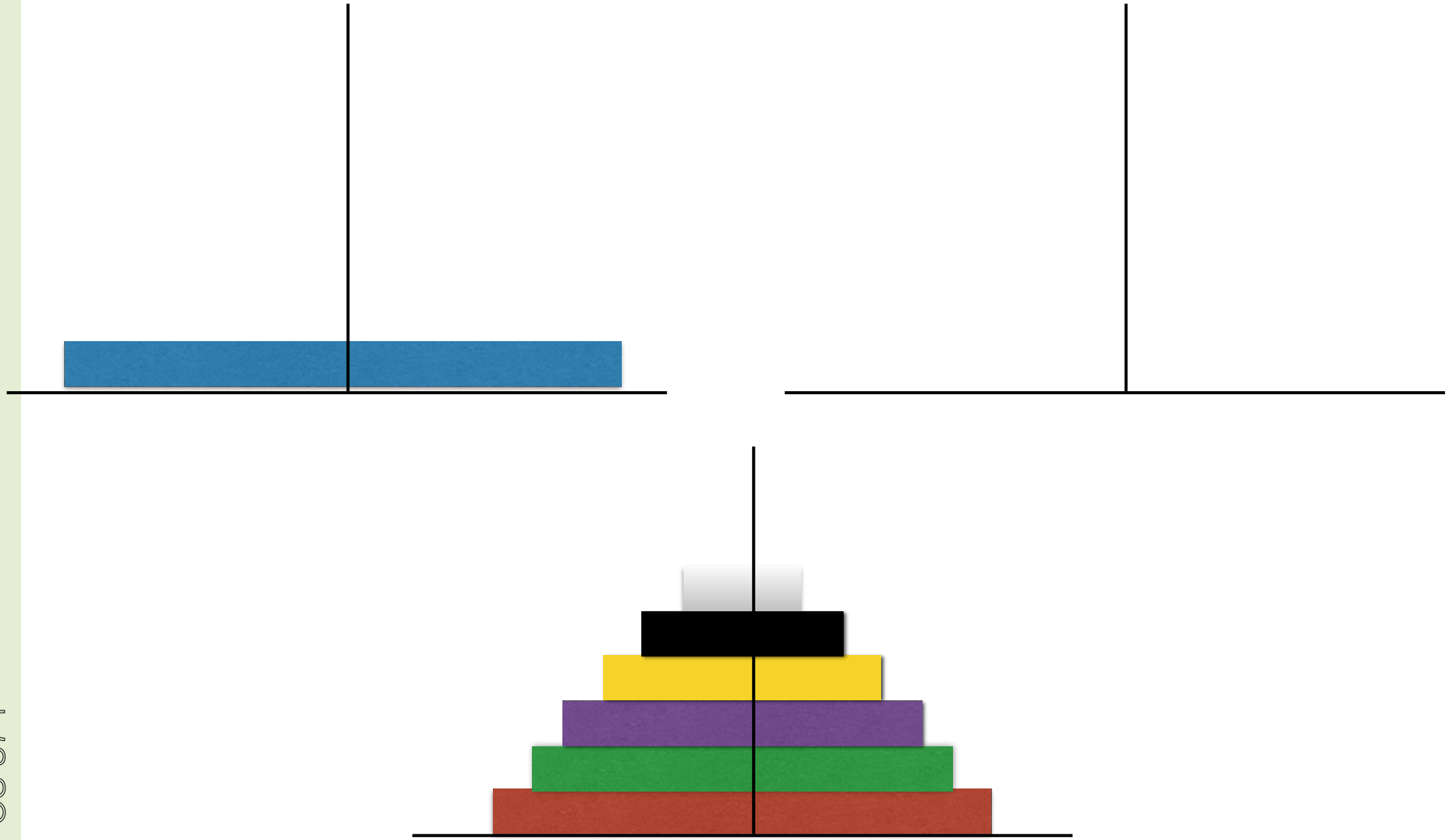
Tower of Hanoi



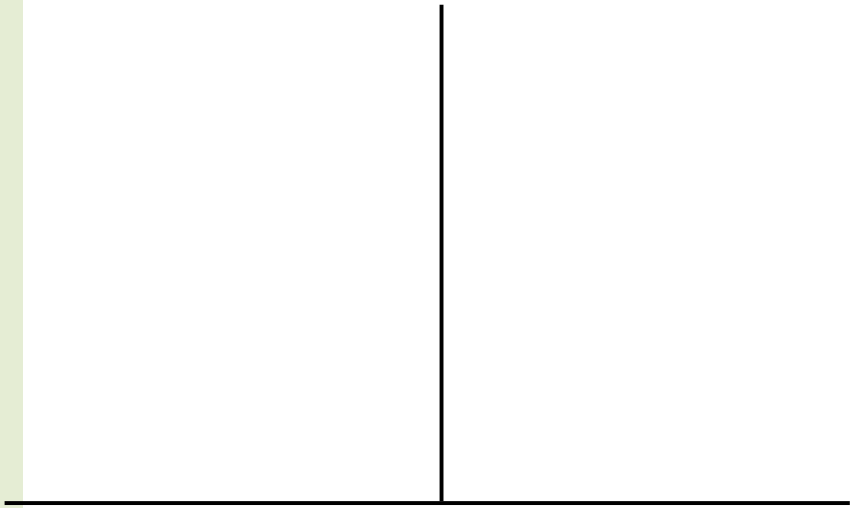
Tower of Hanoi

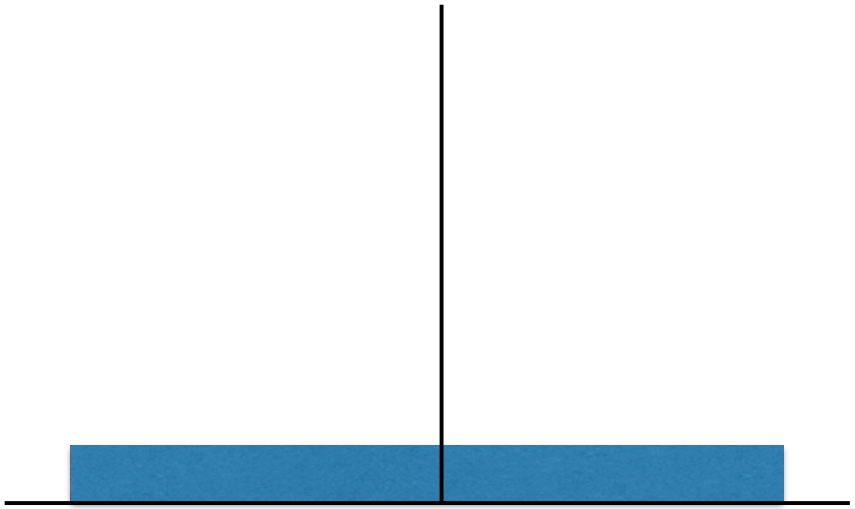


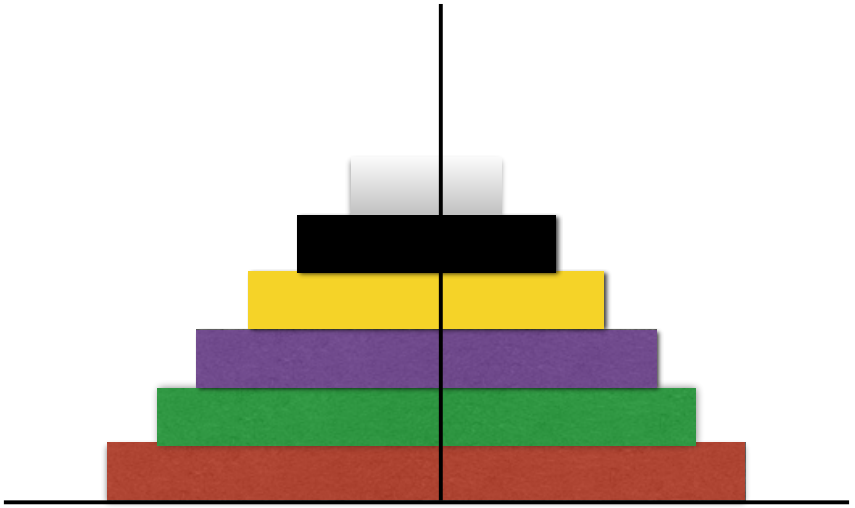
Tower of Hanoi



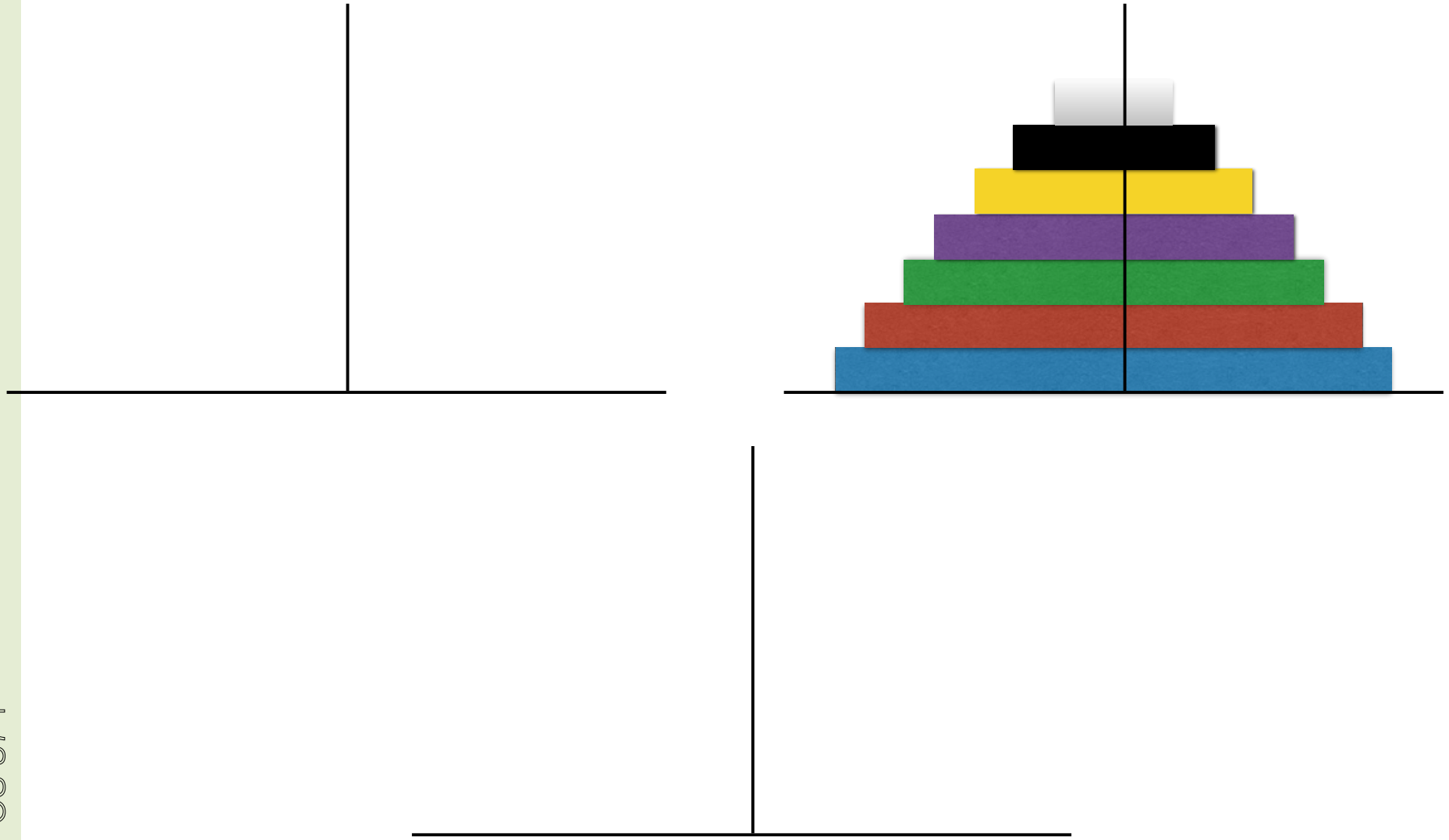
Tower of Hanoi





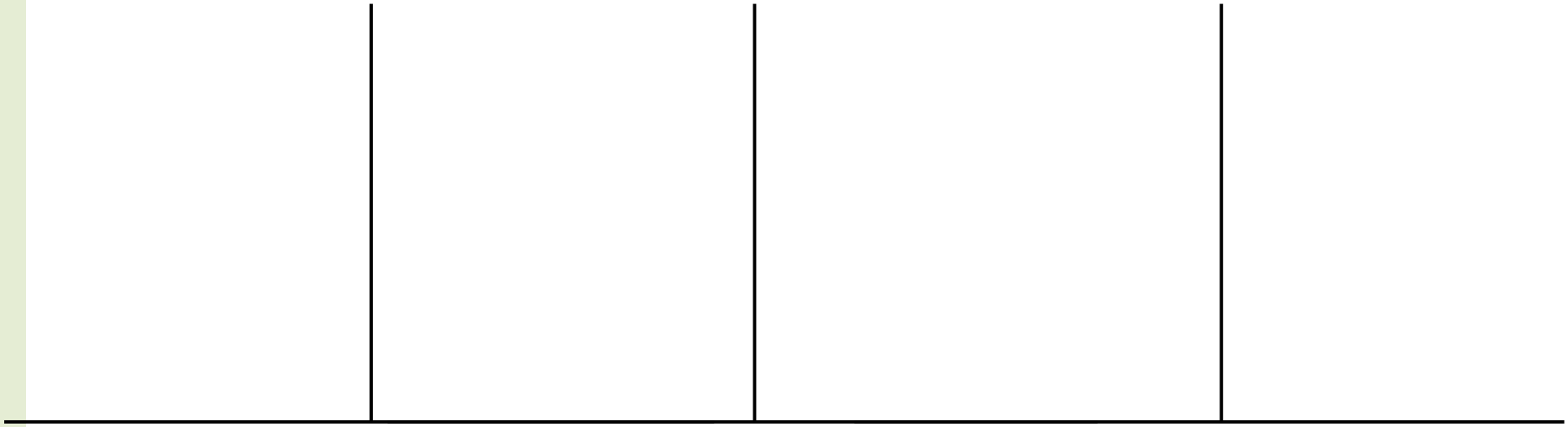


Tower of Hanoi



Base Case?

Need to be careful about when we cannot invoke the induction ferry: Base Cases



Hanoi Algorithm

```
HANOI(n, src, dst, tmp):
    if n > 0
        HANOI(n − 1, src, tmp, dst)
        move disk n from src to dst
        HANOI(n − 1, tmp, dst, src)
```

Reduction = Delegation

Sometimes hard to delegate.

Reduction = Delegation

Say we want to build a minimal DFA from a regular expression

- Reg Exp → NFA (thompson)
- NFA → DFA (subset)
- DFA → min DFA (Moore)

3 Steps. Not important how any of those work, as long as we are guaranteed they work

Reduction = Delegation

How do you hunt a blue elephant?

- With the blue elephant gun

How do you hunt a red elephant?

- Hold its trunk until it turns blue, then hunt it with the blue elephant gun

How do you hunt a white elephant?

- Embarrass it till it becomes red. Use algorithm for hunting red elephants.

Reduction = Delegation

Sometimes hard to delegate.

Recursion even harder to delegate,you have to trust yourself.

Recursion = Delegation to yourself

Recursion is reduction to smaller instances of the SAME problem, which are solved by magic (or fairies, or inductive hypothesis…)

Sorting

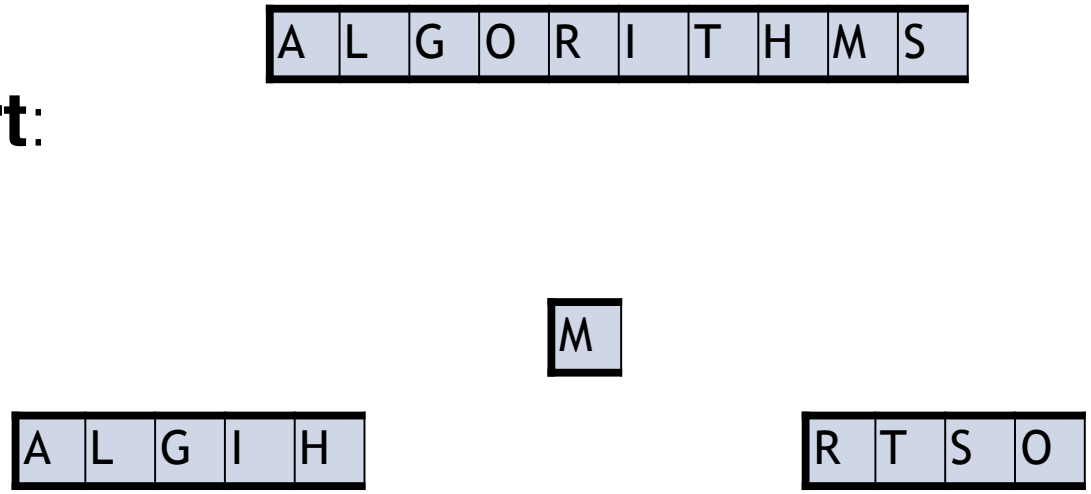
A	L	G	O	R	I	T	H	M	S

Quicksort:

- choose a pivot element from the array
- partition the array into three subarrays: one with elements smaller than pivot, one the pivot itself, one with elements larger than pivot.
- Recursively quick sort the first and last subarray
- How to choose pivot?

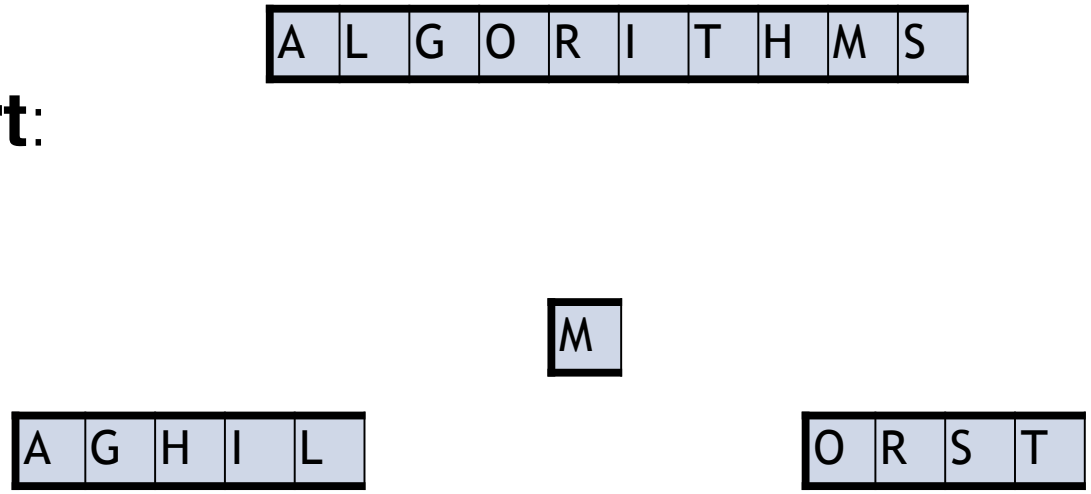
Sorting

Quicksort:



Sorting

Quicksort:



Sorting

A	L	G	O	R	I	T	H	M	S

Quicksort:

A	G	H	I	L	M	O	R	S	T

Sorting

Quicksort:

```
QUICKSORT(A[1..n]):
    if (n > 1)
        Choose a pivot element A[p]
        r ← PARTITION(A, p)
        QUICKSORT(A[1..r − 1])
        QUICKSORT(A[r + 1..n])
```

Sorting

Partition (linear time):

```
PARTITION(A[1..n], p):
    swap A[p] ↔ A[n]
    i ← 0
    j ← n
    while (i < j)
        repeat i ← i + 1 until (i ≥ j or A[i] ≥ A[n])
        repeat j ← j − 1 until (i ≥ j or A[j] ≤ A[n])
        if (i < j)
            swap A[i] ↔ A[j]
    swap A[i] ↔ A[n]
    return i
```

Sorting

A	L	G	O	R	I	T	H	M	S

Mergesort:

- Divide the input array into two subarrays of roughly equal size
- Recursively merge sort each of the subarrays
- Merge the two newly sorted subarrays into a single sorted array

Sorting

A	L	G	O	R	I	T	H	M	S

Mergesort:

A	L	G	O	R

I	T	H	M	S

Sorting

A	L	G	O	R	I	T	H	M	S

Mergesort:

A	G	L	O	R

H	I	M	S	T

Sorting

A	L	G	O	R	I	T	H	M	S

Mergesort:

A	G	L	O	R

H	I	M	S	T

Need to merge the two subarrays.

Sorting

A	L	G	O	R	I	T	H	M	S

Merge:

A	G	L	O	R

H	I	M	S	T

- Compare the first elements of the subarrays
- Write the smallest one in the output array.
- Recursion, now the problem is smaller

Sorting

A	L	G	O	R	I	T	H	M	S

Merge:

A	G	L	O	R

H	I	M	S	T

One comparison, one recursive call

Sorting

Merge:

A	L	G	O	R	I	T	H	M	S

	G	L	O	R

H	I	M	S	T

A									

One comparison, one recursive call

Sorting

Merge:

A	L	G	O	R	I	T	H	M	S

	G	L	O	R

H	I	M	S	T

A									

Where can this recursion break?

Sorting

A	L	G	O	R	I	T	H	M	S

Merge:

H	I	M	S	T

Sorting

Merge:

A	L	G	O	R	I	T	H	M	S

	I	M	S	T

H									

Sorting

A	L	G	O	R	I	T	H	M	S

Merge:

Where can this recursion break?

Sorting

Merge:

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Loop = recursion

- When writing actual code easier to unfold the recursion

- When proving correctness easier to use induction (=recursion)

Sorting

Mergesort:

```
MergeSort(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeSort(A[1..m])
      MergeSort(A[m+1..n])
      Merge(A[1..n], m)
```

Base cases:

- When size of arrays to merge is 1
- When size of arrays is less than10 and then brute force
- It doesn't matter, no need to optimize

Proof of Correctness

- We prove MERGE is correct by induction on $n-k+1$, which is the total size of the two sorted subarrays $A[i..m]$ and $A[j..n]$ that remain to be merged into $B[k..n]$ when the kth iteration of the main loop begins. There are five cases to consider. Yes, five.
  - If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.)
  - If $i \le m$ and $j > n$, the subarray $A[j..n]$ is empty. Because both subarrays are sorted, the smallest element in the union of the two subarrays is $A[i]$. So the assignment $B[k] \leftarrow A[i]$ is correct. The inductive hypothesis implies that the remaining subarrays $A[i+1..m]$ and $A[j..n]$ are correctly merged into $B[k+1..n]$.
  - Similarly, if $i > m$ and $j \le n$, the assignment $B[k] \leftarrow A[j]$ is correct, and The Recursion Fairy correctly merges—sorry, I mean the inductive hypothesis implies that the MERGE algorithm correctly merges—the remaining subarrays $A[i..m]$ and $A[j+1..n]$ into $B[k+1..n]$.
  - If $i \le m$ and $j \le n$ and $A[i] < A[j]$, then the smallest remaining element is $A[i]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly merges the rest of the subarrays.
  - Finally, if $i \le m$ and $j \le n$ and $A[i] \ge A[j]$, then the smallest remaining element is $A[j]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly does the rest.

Always make sanity check when you design algorithm!

Running time

- Number of fundamental operations as a function of input size n
- If array is sorted, then O(n), but we don't care about best case!
- Worst case running time for this class.
- Maybe different in practice, assumptions

Running time of Quicksort

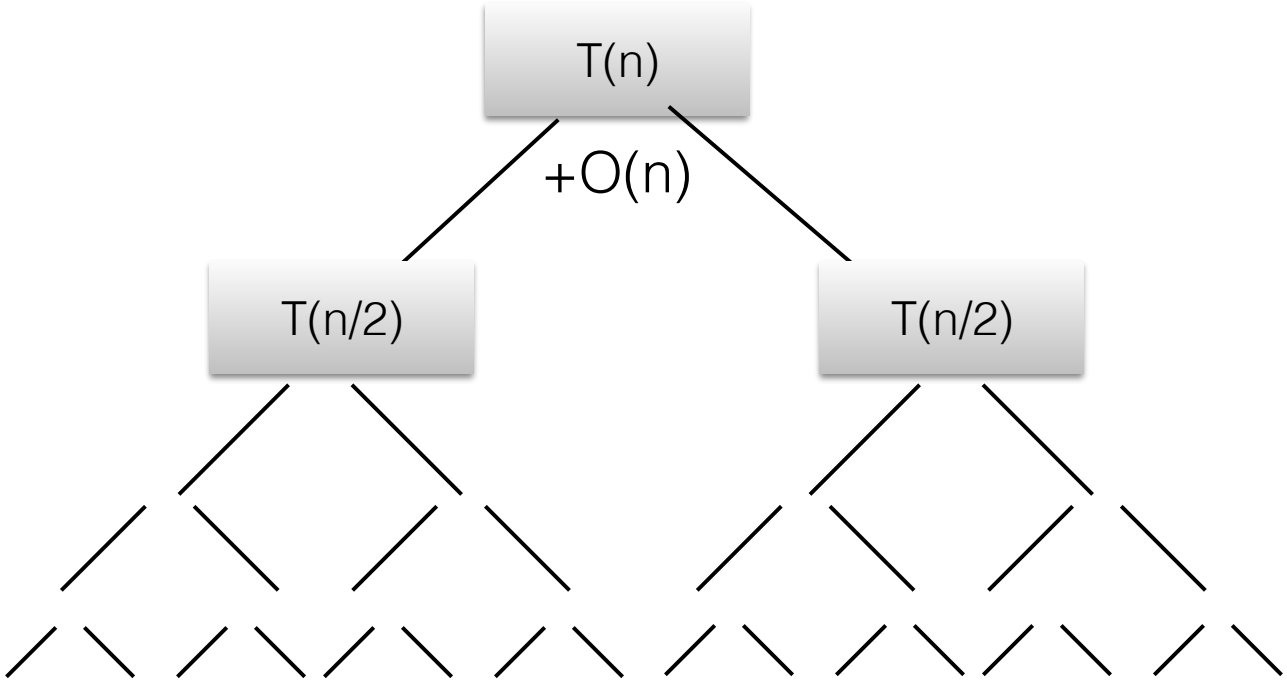
- What is the running time T(n) of quicksort?
- $O(n^2)$ time! (If I choose the smallest pivot)
  - $T(n)=O(n)+T(n-1)$

    $= O(n^2)$

Running time of Mergesort

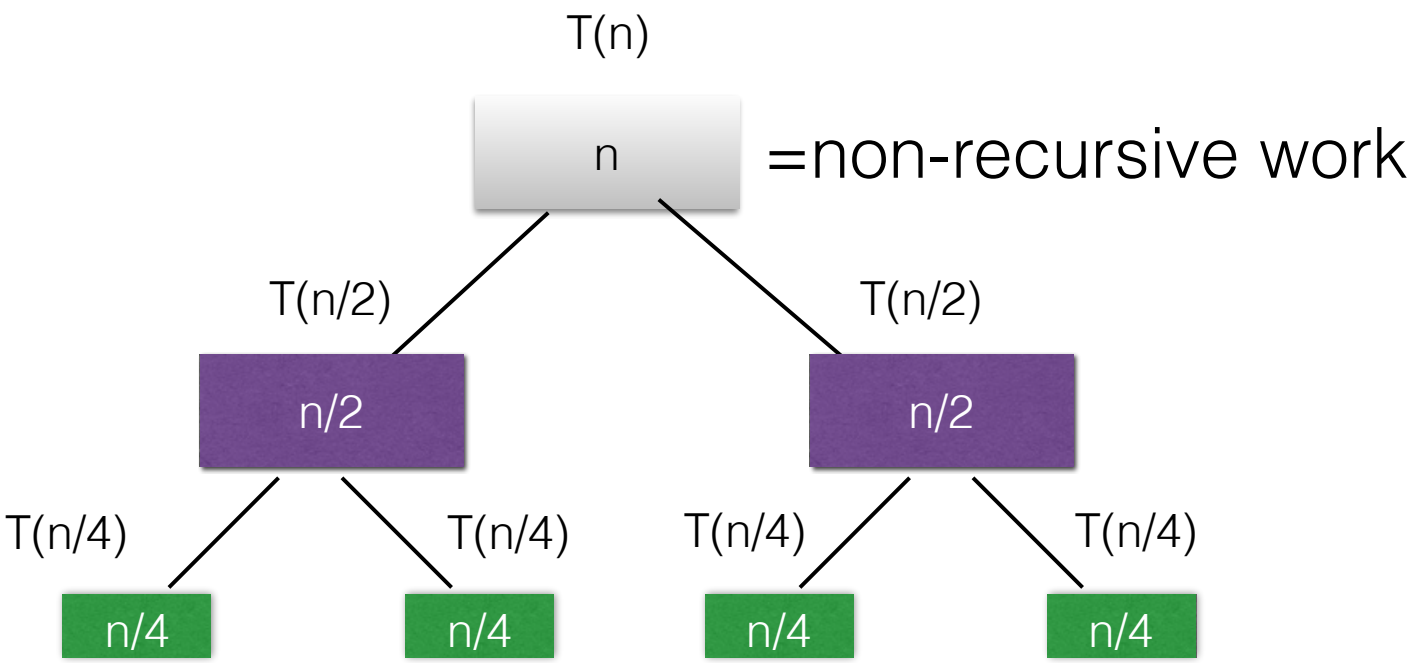
- What is the running time T(n) of mergesort?
- O(nlogn) time!
    - $T(n)=2T(n/2)+O(n)$
    - proof by induction if I know answer
    - recursion tree!

Running time of Mergesort



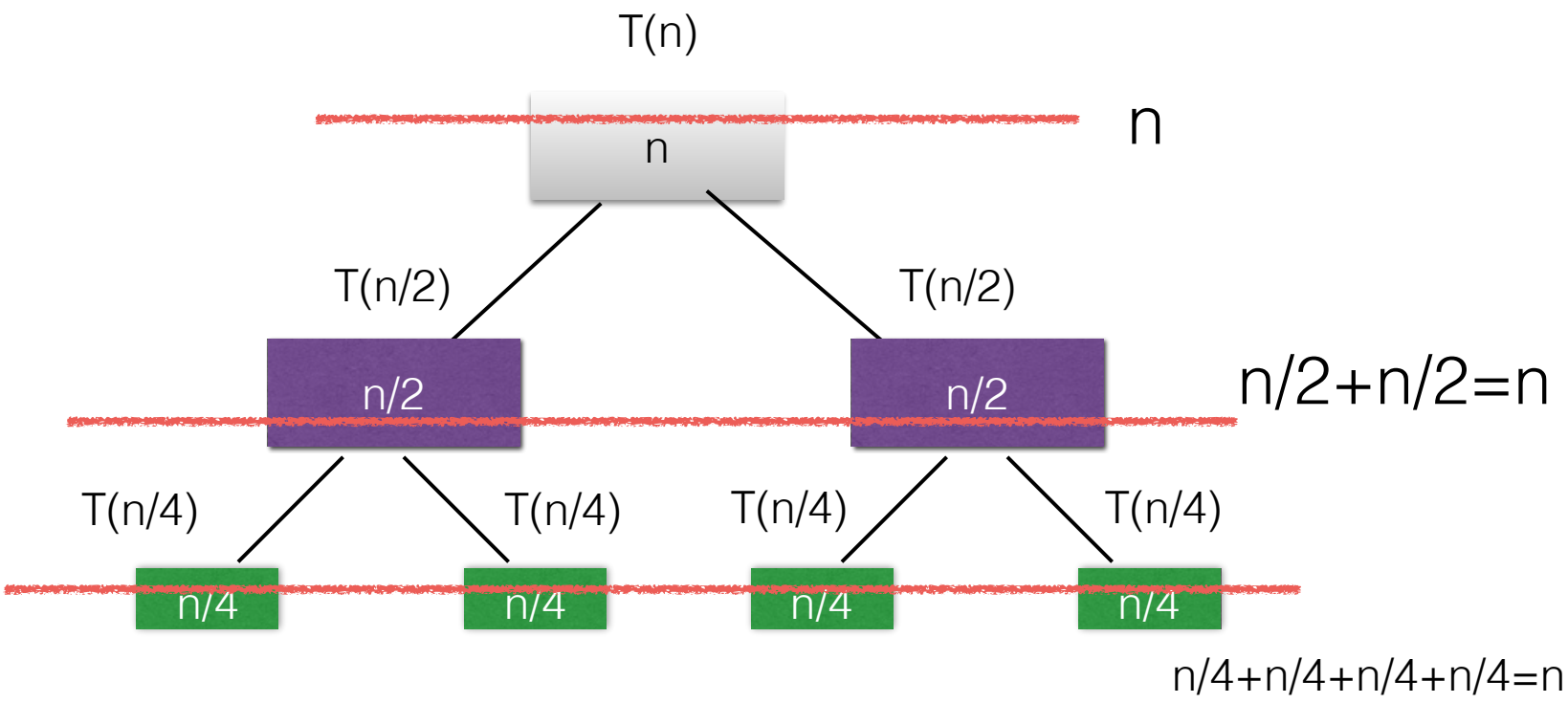
Complete binary tree
every leaf is an array of size 1

Running time of Mergesort



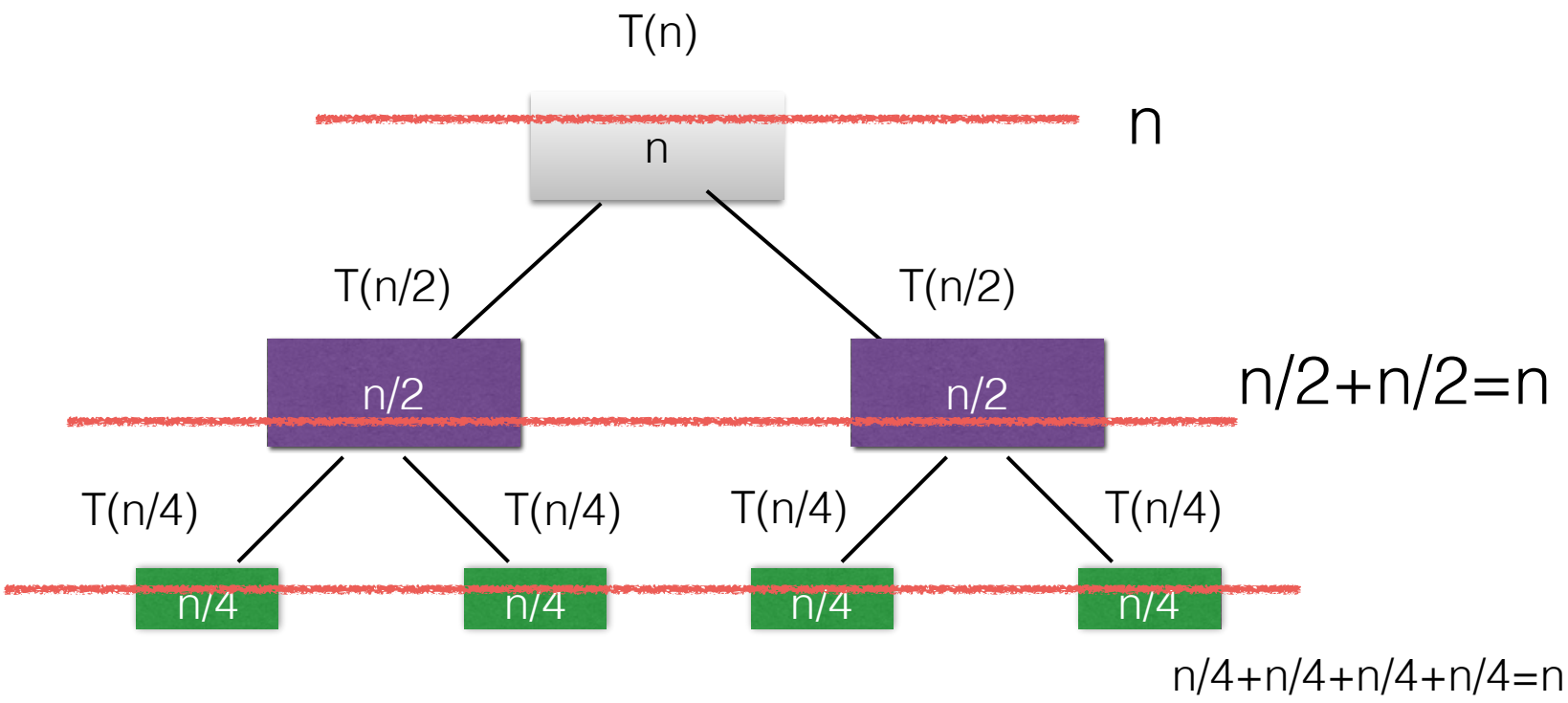
- Leave all the O() till the very end.
- Goal is to sum up all the quantities in all the nodes.

Running time of Mergesort



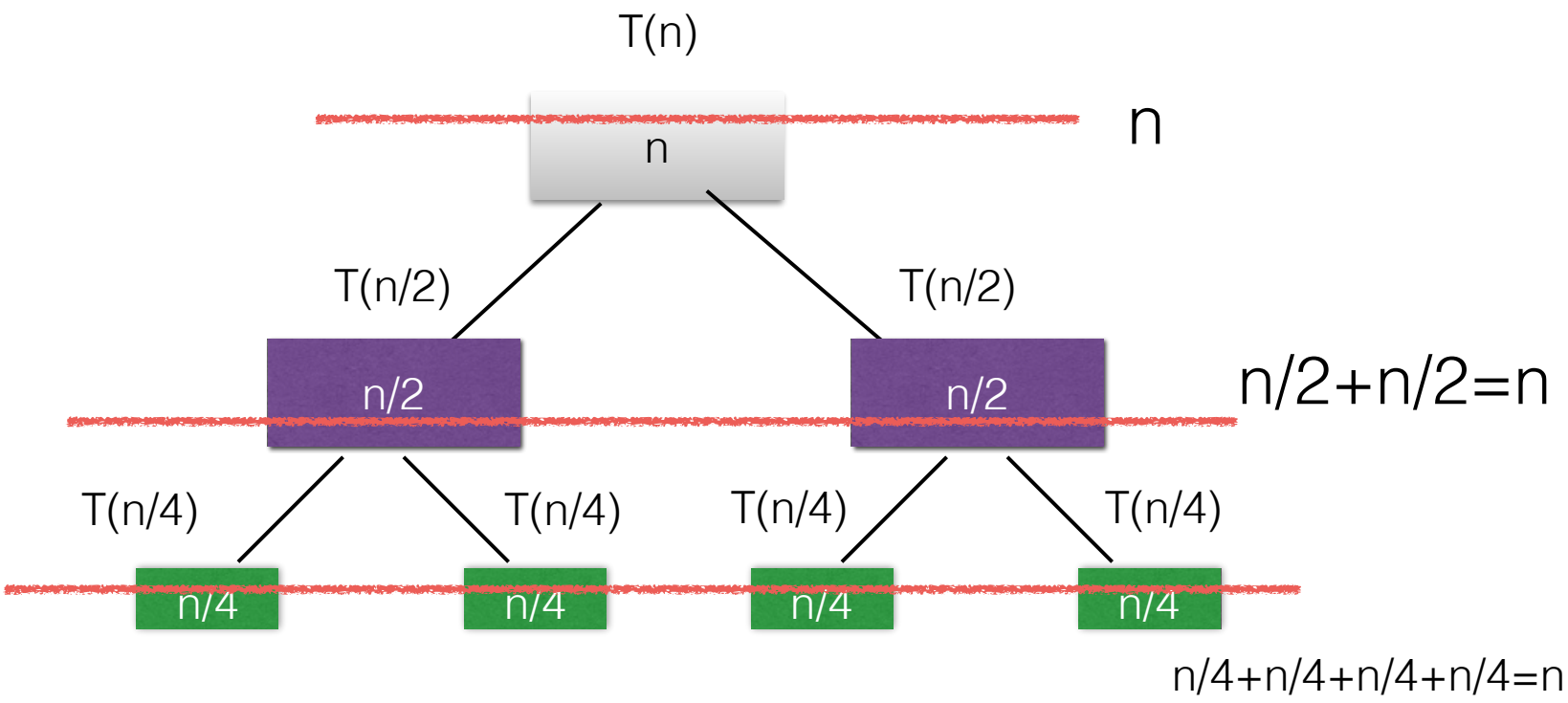
- $T(n)=2T(n/2)+O(n)$
- Solve the recurrence by summing up work at each level

Running time of Mergesort



- $T(n)=2T(n/2)+O(n)$
- Total amount of work at level k= total amount of work at level k-1 (induction).

Running time of Mergesort



- $T(n)=2T(n/2)+O(n)$
- Total amount of work = n x (height of the tree) =n logn

Running time of Quicksort, revisited

- Quicksort runs in time O(n log n) in practice.
- Quicksort runs in time O(n log n) on average if the data is randomly permuted
- Quicksort runs in expected time O(n log n) if we randomly permute the data first.

Part II

Reductions and Recursion

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until he turns blue, and then shoot him with the blue elephant gun.

Q: How do you shoot a white elephant?

A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time:

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time: $O(n^2)$

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a *black box*

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a *black box*

Advantage of naive algorithm: works for objects that cannot be "sorted". Can also consider hashing but outside scope of current course.

Two sides of Reductions

Suppose problem **A** reduces to problem **B**

1. Positive direction: Algorithm for **B** implies an algorithm for **A**
2. Negative direction: Suppose there is no "efficient" algorithm for **A** then it implies no efficient algorithm for **B** (technical condition for reduction time necessary for this)

Two sides of Reductions

Suppose problem **A** reduces to problem **B**

1. Positive direction: Algorithm for **B** implies an algorithm for **A**
2. Negative direction: Suppose there is no "efficient" algorithm for **A** then it implies no efficient algorithm for **B** (technical condition for reduction time necessary for this)

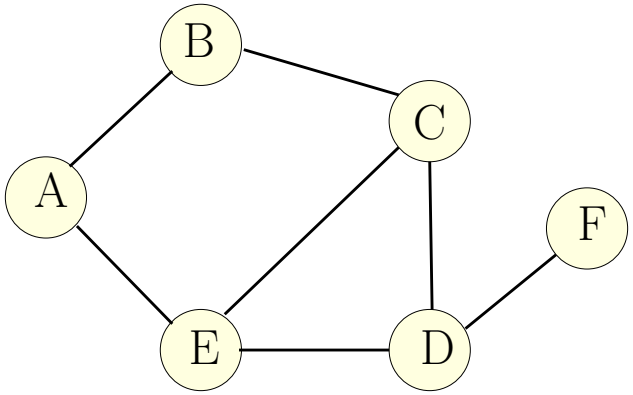
Example: Distinct Elements reduces to Sorting in $\mathbf{O(n)}$ time

1. An $\mathbf{O(n \log n)}$ time algorithm for Sorting implies an $\mathbf{O(n \log n)}$ time algorithm for Distinct Elements problem.
2. If there is *no* $\mathbf{o(n \log n)}$ time algorithm for Distinct Elements problem then there is *no* $\mathbf{o(n \log n)}$ time algorithm for Sorting.

Maximum Independent Set in a Graph

Definition

Given undirected graph $\mathbf{G = (V, E)}$ a subset of nodes $\mathbf{S \subseteq V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u, v \in S}$ then $\mathbf{(u, v) \notin E}$.

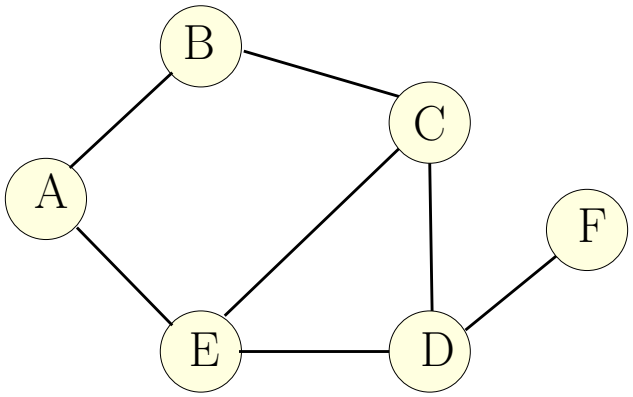


Some independent sets in graph above: {A, C, F}, {B, D}
{D}

Maximum Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$

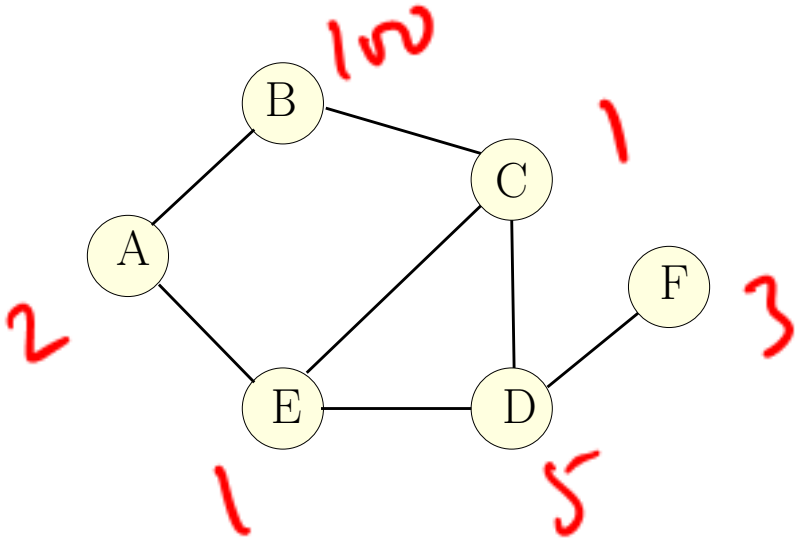
Goal Find maximum sized independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$, weights $\mathbf{w(v) \geq 0}$ for $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$



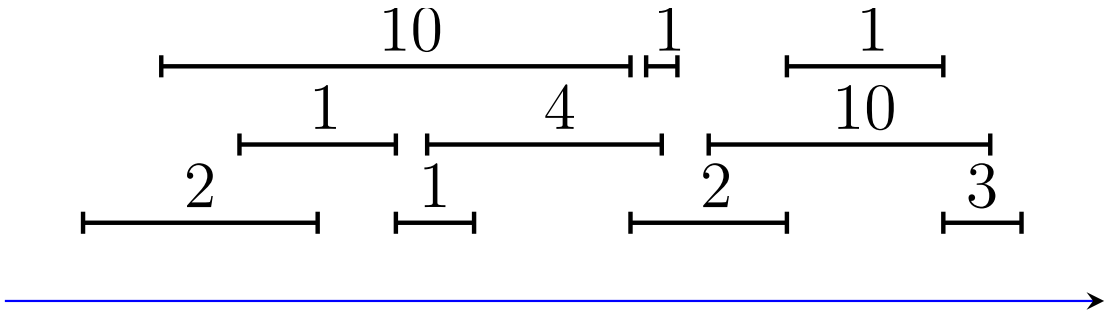
{B, D} 105

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

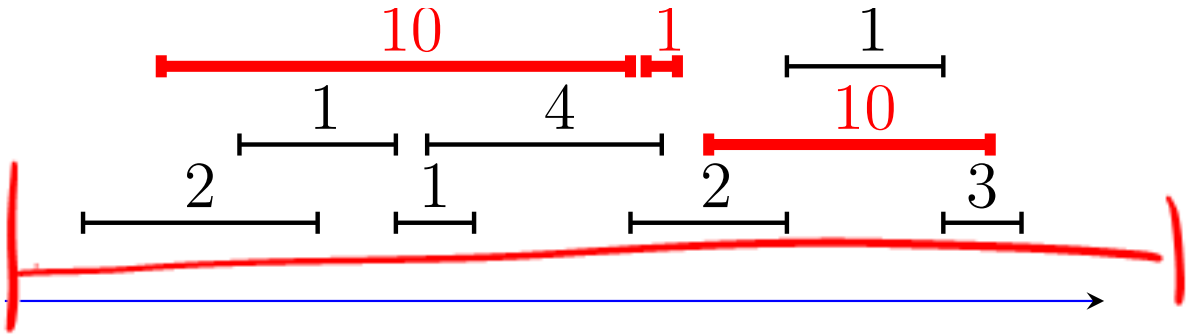


Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!



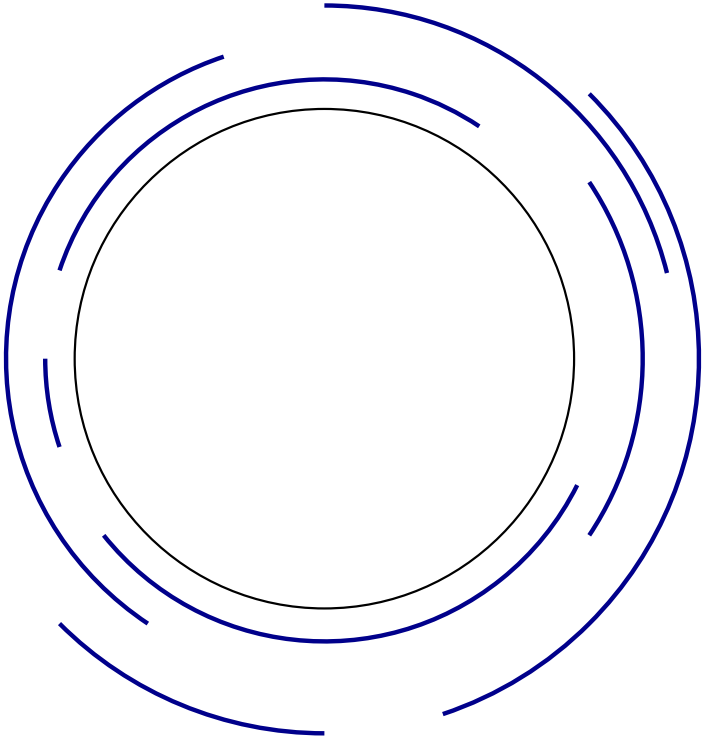
Reduction from Interval Scheduling to MIS

Question: Can you reduce Weighted Interval Scheduling to Max Weight Independent Set Problem?

Weighted Circular Arc Scheduling

Input A set of arcs on a circle, each arc has a *weight* (or profit).

Goal Find a maximum weight subset of arcs that do not overlap.



Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling?

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
MaxWeightIndependentArcs(arcs 𝒞)
    cur-max = 0
    for each arc C ∈ 𝒞 do
        Remove C and all arcs overlapping with C
        w_C = wt of opt. solution in resulting Interval problem
        w_C = w_C + wt(C)
        cur-max = max{cur-max, w_C}
    end for
    return cur-max
```

Reductions

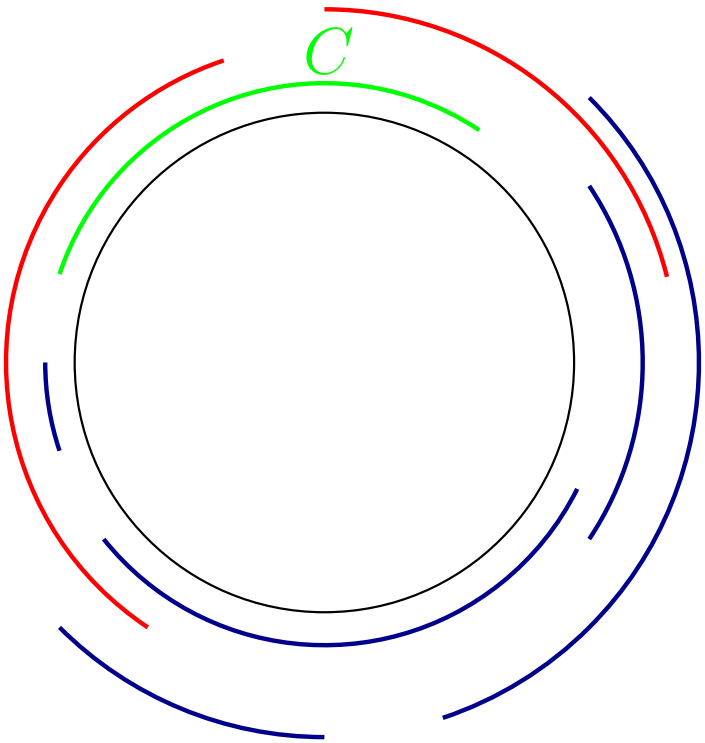
Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
MaxWeightIndependentArcs(arcs 𝒞)
    cur-max = 0
    for each arc C ∈ 𝒞 do
        Remove C and all arcs overlapping with C
        w_C = wt of opt. solution in resulting Interval problem
        w_C = w_C + wt(C)
        cur-max = max{cur-max, w_C}
    end for
    return cur-max
```

n calls to the sub-routine for interval scheduling

Illustration



Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

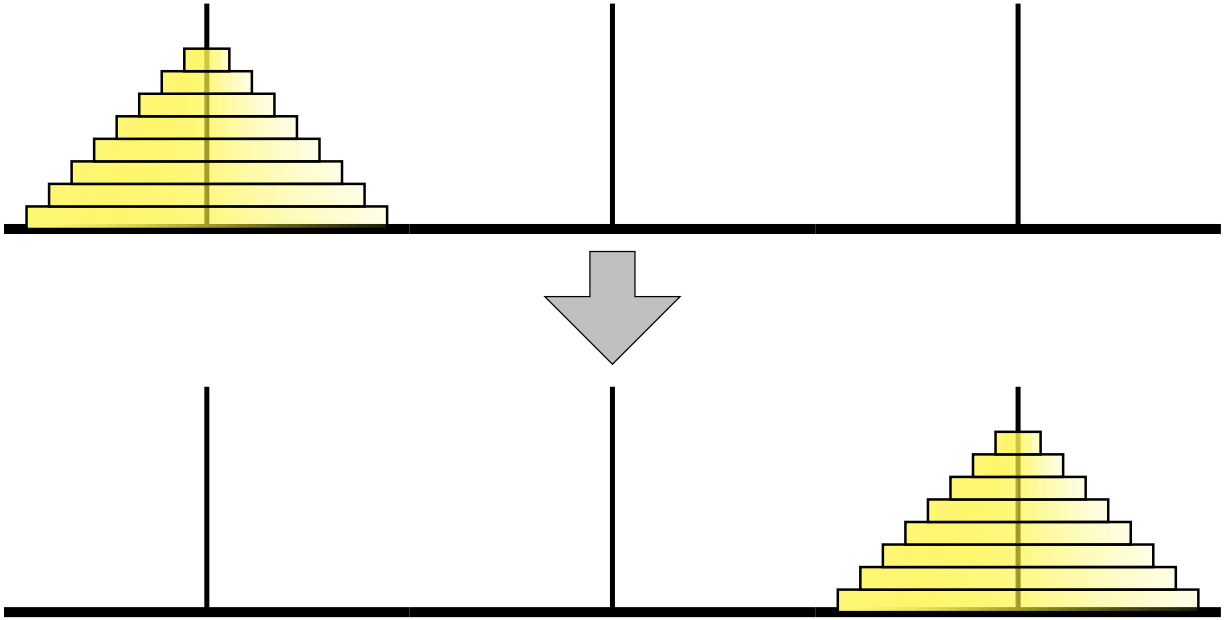
1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size $\mathbf{n}$ is reduced to *one or more* instances of size $\mathbf{n-1}$ or less.
2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. Recursion is a very powerful and fundamental technique
2. Basis for several other methods
   1. Divide and conquer
   2. Dynamic programming
   3. Enumeration and branch and bound etc
   4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

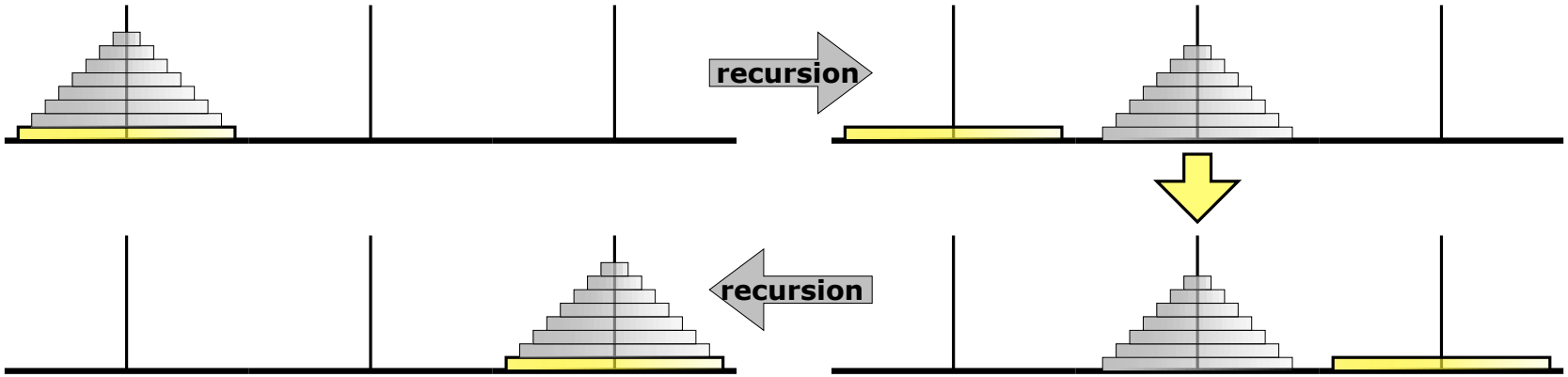
Tower of Hanoi



The Tower of Hanoi puzzle

Move stack of **n** disks from peg **0** to peg **2**, one disk at a time.
Rule: cannot put a larger disk on a smaller disk.
Question: what is a strategy and how many moves does it take?

Tower of Hanoi via Recursion



The Tower of Hanoi algorithm; ignore everything but the bottom disk

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

T(n): time to move **n** disks via recursive strategy

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$\mathbf{T(n)}$: time to move $\mathbf{n}$ disks via recursive strategy

$$\mathbf{T(n) = 2T(n-1) + 1} \qquad \mathbf{n > 1} \qquad \text{and } \mathbf{T(1) = 1}$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Part III

Divide and Conquer

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.

Sorting

Input Given an array of **n** elements

Goal Rearrange them in ascending order

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

A L G O R I T H M S

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

$$\mathbf{A\ L\ G\ O\ R\ I\ T\ H\ M\ S}$$

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

$$\mathbf{A\ L\ G\ O\ R \quad I\ T\ H\ M\ S}$$

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R **H I M S T**

A

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T

A G

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H I

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H I L M O R S T

3. Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical.

Formal Code

```
MergeSort(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeSort(A[1..m])
      MergeSort(A[m+1..n])
      Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Proving Correctness

Obvious way to prove correctness of recursive algorithm:

Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on **n** that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct?

Merge (A[1..n], B[1..m])
~~If~~ If (A[1] < B[1])
return A[1] · Merge (A[2..n], B[1..m])
Else
B[1] Merge (A[1..n], B[2..m])

Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on **n** that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural *loop invariant* that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $\mathbf{n}$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural *loop invariant* that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $\mathbf{k}$ the following hold:

- $\mathbf{B[1..k]}$ contains the smallest $\mathbf{k}$ elements of $\mathbf{A}$ correctly sorted.
- $\mathbf{B[1..k]}$ contains the elements of $\mathbf{A[1..(i-1)]}$ and $\mathbf{A[(m+1)..(j-1)]}$.
- No element of $\mathbf{A}$ is modified.

Running Time

T(n): time for merge sort to sort an **n** element array

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

$$\mathbf{T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn}$$

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

$$\mathbf{T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn}$$

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $\mathbf{f(n)}$ such that $\mathbf{T(n) = \Theta(f(n))}$.

1. $\mathbf{T(n) = O(f(n))}$ - upper bound
2. $\mathbf{T(n) = \Omega(f(n))}$ - lower bound

Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Review notes on recurrence solving.

Recursion Trees

Merge Sort Variant

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $\mathbf{k}$ arrays of size $\mathbf{n}/\mathbf{k}$ each?

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.
    1. Theoretically, median can be found in linear time.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Part IV

Binary Search

Binary Search in Sorted Arrays

Input Sorted array **A** of **n** numbers and number **x**

Goal Is **x** in **A**?

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
BinarySearch(A[a..b], x):
    if (b − a < 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b],x)
```

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
BinarySearch(A[a..b], x):
    if (b − a < 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b],x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

Observation: After k steps, size of array left is $n/2^k$

Another common use of binary search

1. Optimization version: find solution of best (say minimum) value
2. Decision version: is there a solution of value at most a given value **v**?

Another common use of binary search

1. Optimization version: find solution of best (say minimum) value
2. Decision version: is there a solution of value at most a given value $\mathbf{v}$?

Reduce optimization to decision (may be easier to think about):

1. Given instance $\mathbf{I}$ compute upper bound $\mathbf{U(I)}$ on best value
2. Compute lower bound $\mathbf{L(I)}$ on best value
3. Do binary search on interval $\mathbf{[L(I), U(I)]}$ using decision version as black box
4. $\mathbf{O(\log(U(I) - L(I)))}$ calls to decision version if $\mathbf{U(I), L(I)}$ are integers

Example

1. Problem: shortest paths in a graph.
2. Decision version: given $\mathbf{G}$ with non-negative integer edge lengths, nodes $\mathbf{s}, \mathbf{t}$ and bound $\mathbf{B}$, is there an $\mathbf{s}$-$\mathbf{t}$ path in $\mathbf{G}$ of length at most $\mathbf{B}$?
3. Optimization version: find the length of a shortest path between $\mathbf{s}$ and $\mathbf{t}$ in $\mathbf{G}$.

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

Example continued

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

1. Let $\mathbf{U}$ be maximum edge length in $\mathbf{G}$.
2. Minimum edge length is $\mathbf{L}$.
3. $\mathbf{s}$-$\mathbf{t}$ shortest path length is at most $\mathbf{(n-1)U}$ and at least $\mathbf{L}$.
4. Apply binary search on the interval $\mathbf{[L, (n-1)U]}$ via the algorithm for the decision problem.
5. $\mathbf{O(\log((n-1)U - L))}$ calls to the decision problem algorithm sufficient. Polynomial in input size.

Part V

Solving Recurrences

Solving Recurrences

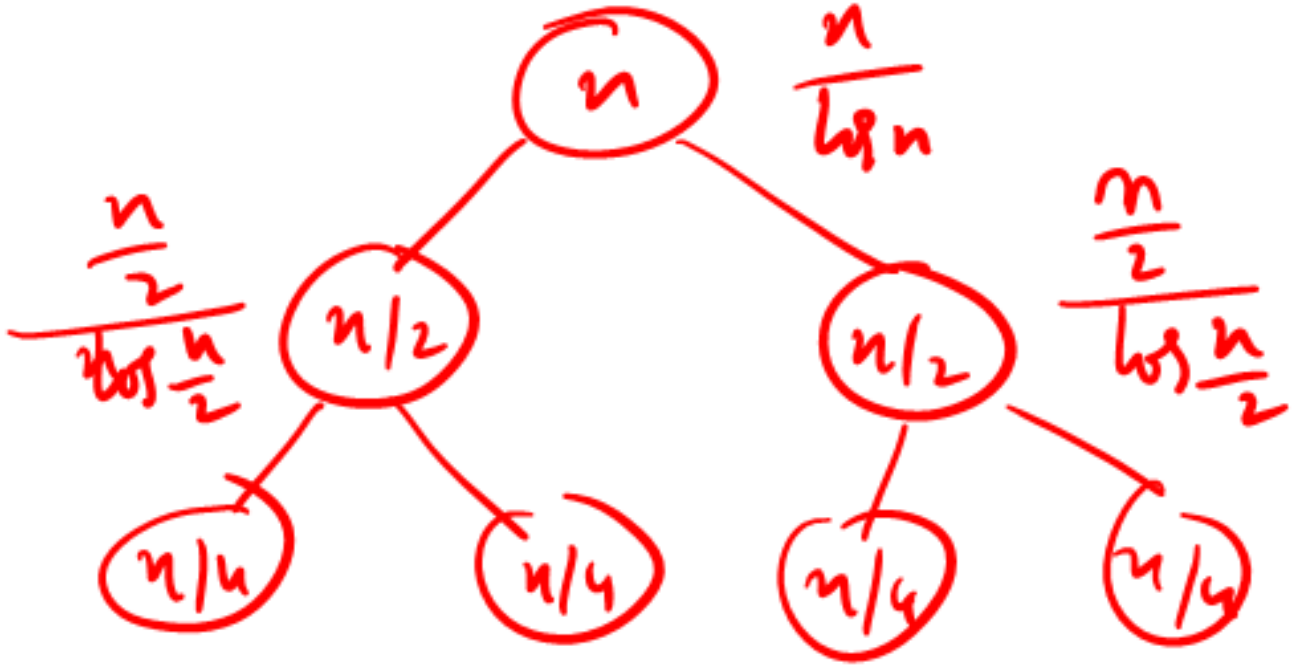
Two general methods:

1. Recursion tree method: need to do sums
   1. elementary methods, geometric series
   2. integration
2. Guess and Verify
   1. guessing involves intuition, experience and trial & error
   2. verification is via induction

Recurrence: Example I

1. Consider $\mathbf{T(n) = 2T(n/2) + n/\log n}$.

$T(2) = 1$



$\underline{\underline{n \log \log n}}$

$$\frac{n}{\log n} + 2 \cdot \frac{\frac{n}{2}}{\log \frac{n}{2}} + 4 \cdot \frac{\frac{n}{4}}{\log \frac{n}{4}} + \cdots \Bigg| \; n\left[\frac{1}{\log n} + \frac{1}{\log n - 1} + \frac{1}{\log n - 2} + \cdots\right]$$

Recurrence: Example I

1. Consider $T(n) = 2T(n/2) + n/\log n$.
2. Construct recursion tree, and observe pattern. ith level has 2^i nodes, and problem size at each node is $n/2^i$ and hence work at each node is $\frac{n}{2^i}/\log\frac{n}{2^i}$.
3. Summing over all levels

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log n - 1} 2^i \left[\frac{(n/2^i)}{\log(n/2^i)}\right] \\
&= \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} \\
&= n \sum_{j=1}^{\log n} \frac{1}{j} = nH_{\log n} = \Theta(n \log\log n)
\end{aligned}
$$

Recurrence: Example II

1. Consider $\mathbf{T(n) = T(\sqrt{n}) + 1}$

$T(2) = 1$

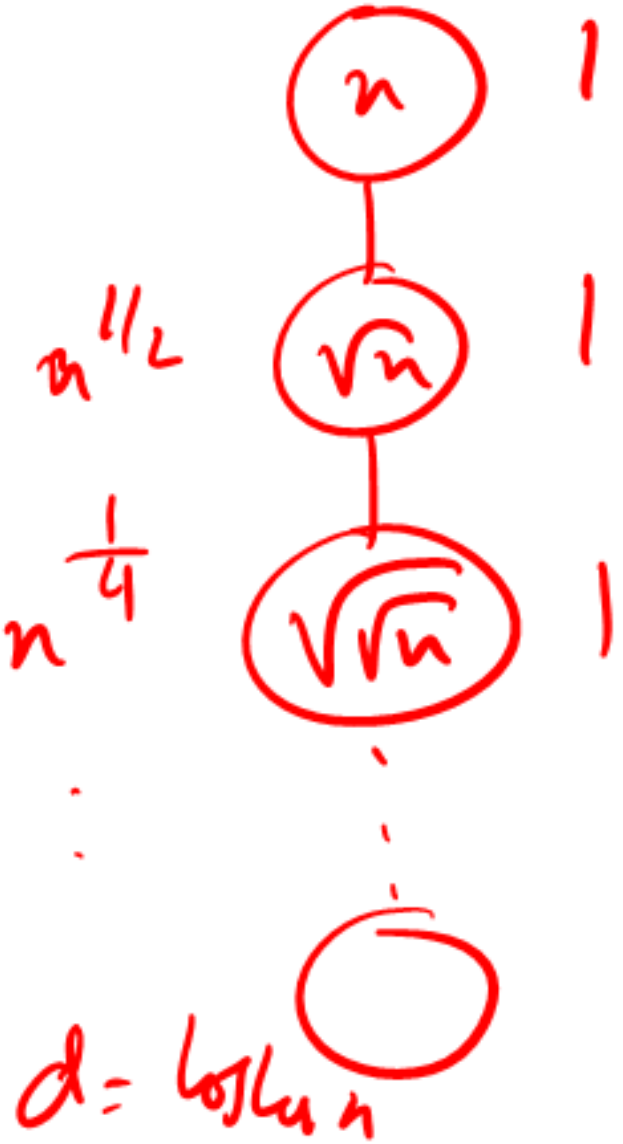
$n^{\frac{1}{2^i}}$ $\quad d$

$n^{\frac{1}{2^d}} = 2$

$\frac{1}{2^d} \log n = 1$

$2^d = \log n$

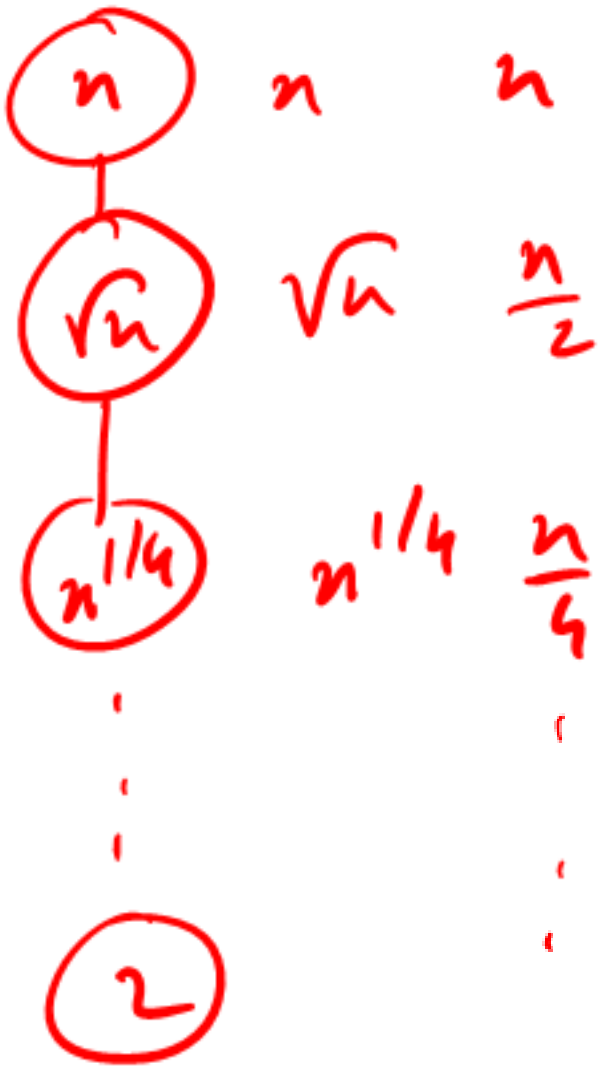
$d = \log\log n$



$$T(n) = T(\sqrt{n}) + n$$

$$n + \sqrt{n} + n^{1/4} + \dots + 2$$

$$= O(\quad)$$

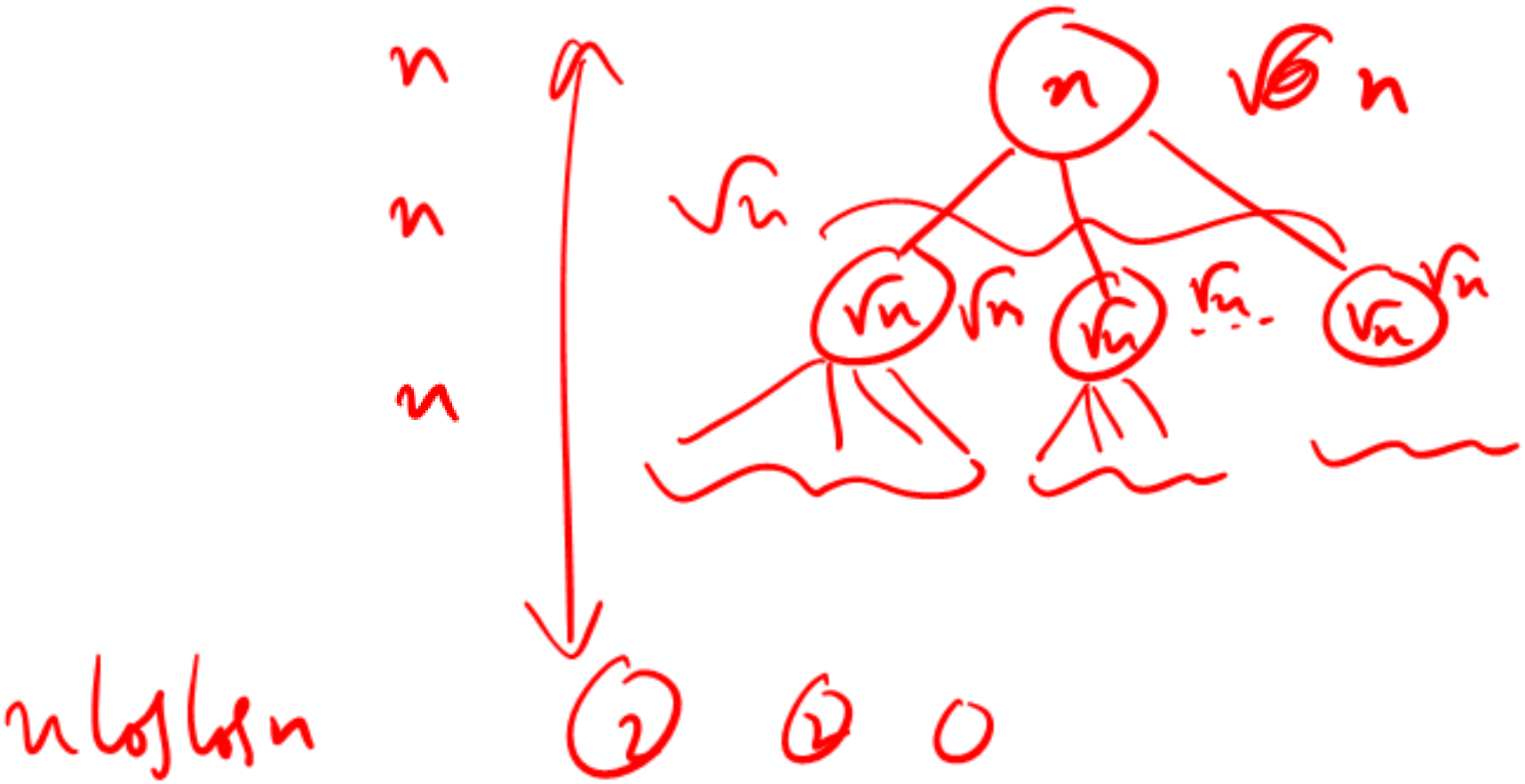


Recurrence: Example II

1. Consider $T(n) = T(\sqrt{n}) + 1$
2. What is the depth of recursion?
   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
3. Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
4. Number of children at each level is 1, work at each node is 1
5. Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.



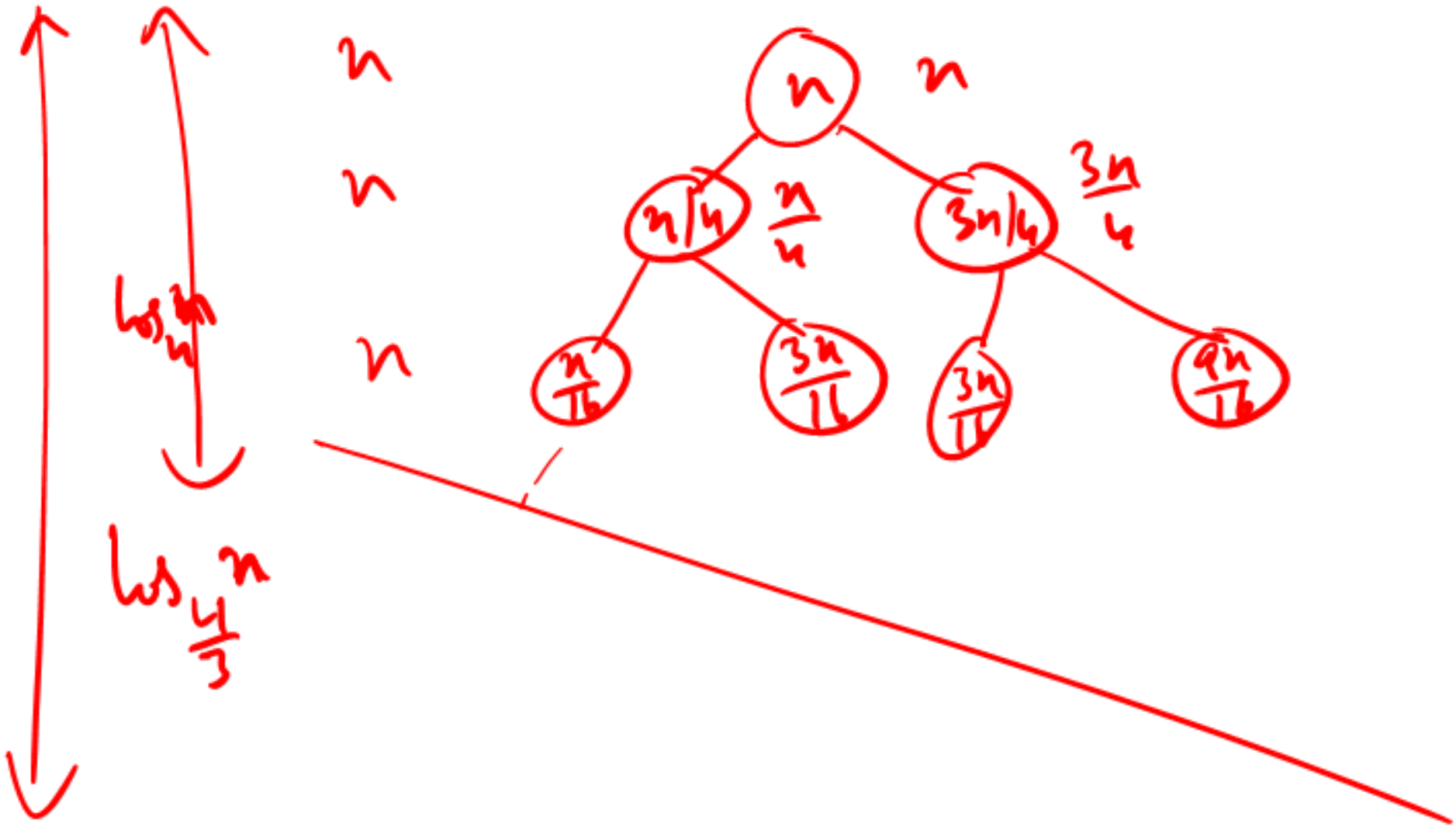
Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.
2. Using recursion trees: number of levels $\mathbf{L = \log \log n}$
3. Work at each level? Root is $\mathbf{n}$, next level is $\mathbf{\sqrt{n} \times \sqrt{n} = n}$. Can check that each level is $\mathbf{n}$.
4. Thus, $\mathbf{T(n) = \Theta(n \log \log n)}$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.

$T(n) = 1 \quad n = 2$



Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$, which means $\mathbf{T(n) = \Theta(n \log n)}$

Part I

Fast Multiplication

Multiplying Numbers

Problem Given two **n**-digit numbers **x** and **y**, compute their product.

Grade School Multiplication

Compute "partial product" by multiplying each digit of **y** with **x** and adding the partial products.

$$
\begin{array}{r}
3141 \\
\times 2718 \\
\hline
25128 \\
3141 \\
21987 \\
6282 \\
\hline
8537238
\end{array}
$$

Time Analysis of Grade School Multiplication

1. Each partial product: $\Theta(n)$
2. Number of partial products: $\Theta(n)$
3. Addition of partial products: $\Theta(n^2)$
4. Total time: $\Theta(n^2)$

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a + bi)}$ and $\mathbf{(c + di)}$

$$\mathbf{(a + bi)(c + di) = ac - bd + (ad + bc)i}$$

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a + bi)}$ and $\mathbf{(c + di)}$

$$\mathbf{(a + bi)(c + di) = ac - bd + (ad + bc)i}$$

How many multiplications do we need?

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a + bi)}$ and $\mathbf{(c + di)}$

$$\mathbf{(a + bi)(c + di) = ac - bd + (ad + bc)i}$$

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute $\mathbf{ac, bd, (a + b)(c + d)}$. Then
$\mathbf{(ad + bc) = (a + b)(c + d) - ac - bd}$

Divide and Conquer

Assume $\mathbf{n}$ is a power of $\mathbf{2}$ for simplicity and numbers are in decimal.

Split each number into two numbers with equal number of digits

1. $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$
2. $x = x_{n-1}\ldots x_{n/2}0\ldots 0 + x_{n/2-1}\ldots x_0$
3. $x_L = 10^{n/2}x_L$ where $x_L = x_{n-1}\ldots x_{n/2}$ and $x_R = x_{n/2-1}\ldots x_0$
4. Similarly $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\ldots y_{n/2}$ and $y_R = y_{n/2-1}\ldots y_0$

Example

$$
\begin{aligned}
1234 \times 5678 &= (100 \times 12 + 34) \times (100 \times 56 + 78) \\
&= 10000 \times 12 \times 56 \\
&\quad +100 \times (12 \times 78 + 34 \times 56) \\
&\quad +34 \times 78
\end{aligned}
$$

Divide and Conquer

Assume $\mathbf{n}$ is a power of $\mathbf{2}$ for simplicity and numbers are in decimal.

1. $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$
2. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\ldots x_{n/2}$ and $x_R = x_{n/2-1}\ldots x_0$
3. $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\ldots y_{n/2}$ and $y_R = y_{n/2-1}\ldots y_0$

Therefore

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

$T(n) = 4T(\frac{n}{2}) + 6n \quad T(4) = 1$

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$\mathbf{T(n) = \Theta(n^2)}$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

$$T(n) = 3T\left(\frac{n}{2}\right) + n$$

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

State of the Art

Schönhage-Strassen 1971: $\mathbf{O(n \log n \log \log n)}$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $\mathbf{O(n \log n 2^{O(\log^* n)})}$ time

Conjecture

There is an $\mathbf{O(n \log n)}$ time algorithm.

Analyzing the Recurrences

1. Basic divide and conquer: $\mathbf{T(n) = 4T(n/2) + O(n)}$, $\mathbf{T(1) = 1}$. **Claim:** $\mathbf{T(n) = \Theta(n^2)}$.
2. Saving a multiplication: $\mathbf{T(n) = 3T(n/2) + O(n)}$, $\mathbf{T(1) = 1}$. **Claim:** $\mathbf{T(n) = \Theta(n^{1+\log 1.5})}$

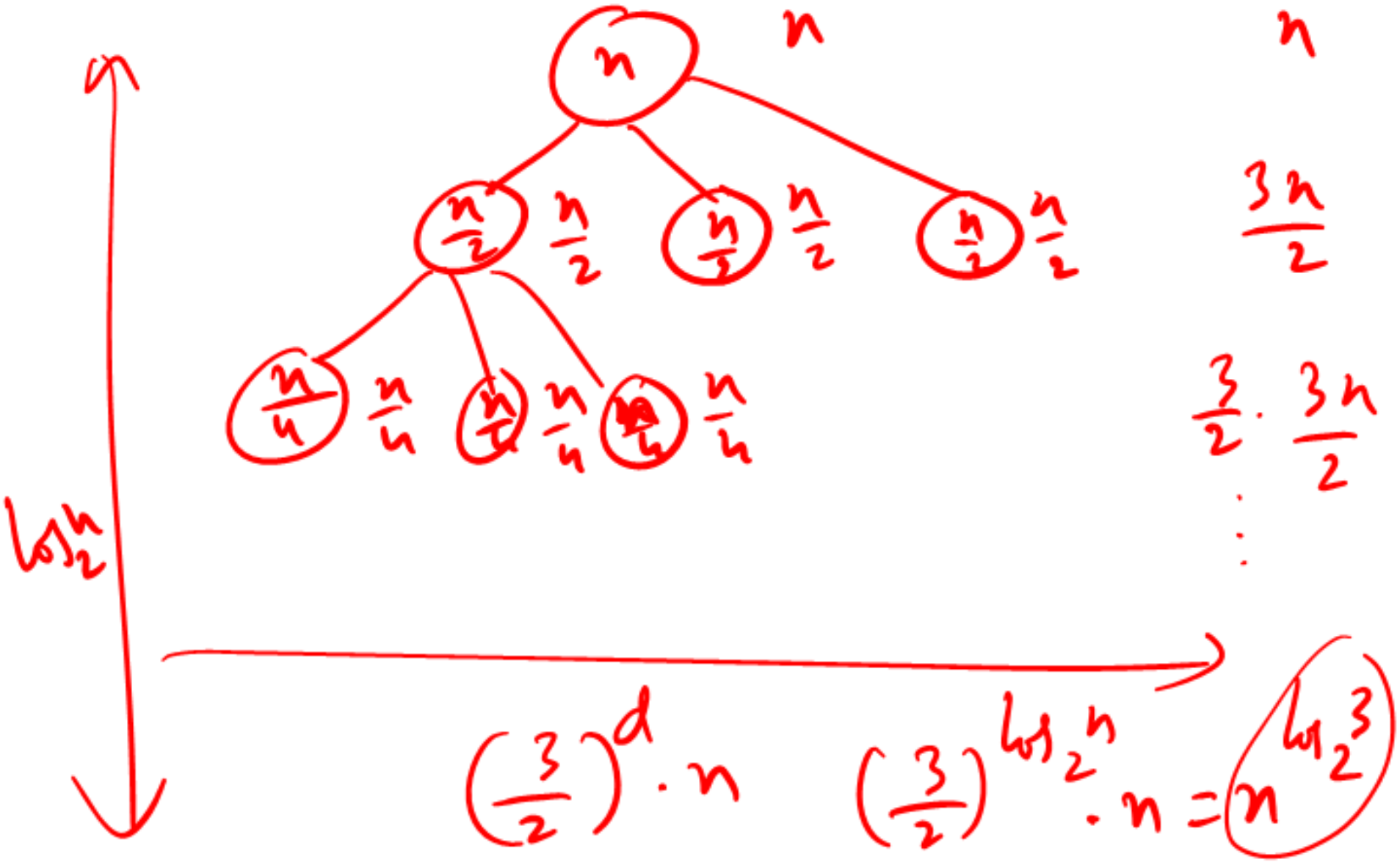
Analyzing the Recurrences

1. Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
2. Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

1. In both cases, depth of recursion $L = \log n$.
2. Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
3. Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Recursion tree analysis



Part II

Selecting in Unsorted Lists

Rank of element in an array

A: an unsorted array of **n** integers

Definition

For $\mathbf{1} \leq \mathbf{j} \leq \mathbf{n}$, element of rank **j** is the **j**'th smallest element in **A**.

Unsorted array	16	14	34	20	12	5	3	19	11
Ranks	6	5	9	8	4	2	1	7	3

Sort of array	3	5	11	12	14	16	19	20	34

Problem - Selection

Input Unsorted array **A** of **n** integers **and** integer **j**

Goal Find the **j**th smallest number in **A** (*rank* **j** number)

Median: $\mathbf{j} = \lfloor (\mathbf{n}+1)/2 \rfloor$

Problem - Selection

Input Unsorted array **A** of **n** integers **and** integer **j**

Goal Find the **j**th smallest number in **A** (*rank* **j** number)

Median: $\mathbf{j} = \lfloor (\mathbf{n}+1)/2 \rfloor$

Simplifying assumption for sake of notation: elements of **A** are distinct

Algorithm I

1. Sort the elements in **A**
2. Pick **j**th element in sorted order

Time taken = $\mathbf{O(n \log n)}$

Algorithm I

1. Sort the elements in **A**
2. Pick **j**th element in sorted order

Time taken = $\mathbf{O(n \log n)}$

Do we need to sort? Is there an $\mathbf{O(n)}$ time algorithm?

Algorithm II

If $\mathbf{j}$ is small or $\mathbf{n - j}$ is small then

1. Find $\mathbf{j}$ smallest/largest elements in $\mathbf{A}$ in $\mathbf{O(jn)}$ time. (How?)
2. Time to find median is $\mathbf{O(n^2)}$.

Divide and Conquer Approach

1. Pick a pivot element $\mathbf{a}$ from $\mathbf{A}$
2. Partition $\mathbf{A}$ based on $\mathbf{a}$.
   $\mathbf{A}_{\mathbf{less}} = \{\mathbf{x} \in \mathbf{A} \mid \mathbf{x} \leq \mathbf{a}\}$ and $\mathbf{A}_{\mathbf{greater}} = \{\mathbf{x} \in \mathbf{A} \mid \mathbf{x} > \mathbf{a}\}$
3. $|\mathbf{A}_{\mathbf{less}}| = \mathbf{j}$: return $\mathbf{a}$
4. $|\mathbf{A}_{\mathbf{less}}| > \mathbf{j}$: recursively find $\mathbf{j}$th smallest element in $\mathbf{A}_{\mathbf{less}}$
5. $|\mathbf{A}_{\mathbf{less}}| < \mathbf{j}$: recursively find $\mathbf{k}$th smallest element in $\mathbf{A}_{\mathbf{greater}}$ where $\mathbf{k} = \mathbf{j} - |\mathbf{A}_{\mathbf{less}}|$.

Example

16	14	34	20	12	5	3	19	11

j = 8

14, 5, 3, 12, 11 16 20 34 19

↑

6

Time Analysis

1. Partitioning step: **O(n)** time to scan **A**
2. How do we choose pivot? Recursive running time?

Time Analysis

1. Partitioning step: **O(n)** time to scan **A**
2. How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be **A[1]**.

Time Analysis

1. Partitioning step: $\mathbf{O(n)}$ time to scan $\mathbf{A}$
2. How do we choose pivot? Recursive running time?

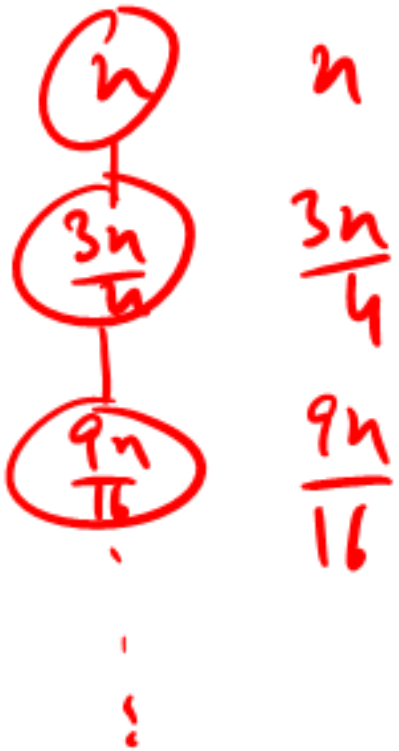
Suppose we always choose pivot to be $\mathbf{A[1]}$.

Say $\mathbf{A}$ is sorted in increasing order and $\mathbf{j = n}$.
Exercise: show that algorithm takes $\mathbf{\Omega(n^2)}$ time

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$T(n) \leq n + T\left(\frac{3n}{4}\right)$$



A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4 \leq \ell \leq 3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4 \leq |A_{less}| \leq 3n/4}$ *and* $\mathbf{n/4 \leq |A_{greater}| \leq 3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

Divide and Conquer Approach

A game of medians

Idea

1. Break input $\mathbf{A}$ into many subarrays: $\mathbf{L_1, \ldots L_k}$.
2. Find median $\mathbf{m_i}$ in each subarray $\mathbf{L_i}$.
3. Find the median $\mathbf{x}$ of the medians $\mathbf{m_1, \ldots, m_k}$.
4. Intuition: The median $\mathbf{x}$ should be close to being a good median of all the numbers in $\mathbf{A}$.
5. Use $\mathbf{x}$ as pivot in previous algorithm.

Example

11	7	3	42	174	310	1	92	87	12	19	15

7 174 87 15

$\frac{n}{3}$

15

Choosing the pivot

A clash of medians

1. Partition array $\mathbf{A}$ into $\lceil \mathbf{n/5} \rceil$ lists of $\mathbf{5}$ items each.
   $\mathbf{L_1 = \{A[1], A[2], \ldots, A[5]\}}$, $\mathbf{L_2 = \{A[6], \ldots, A[10]\}}$, $\ldots$,
   $\mathbf{L_i = \{A[5i+1], \ldots, A[5i-4]\}}$, $\ldots$,
   $\mathbf{L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4, \ldots, A[n]\}}$.
2. For each $\mathbf{i}$ find median $\mathbf{b_i}$ of $\mathbf{L_i}$ using brute-force in $\mathbf{O(1)}$ time.
   Total $\mathbf{O(n)}$ time
3. Let $\mathbf{B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}}$
4. Find median $\mathbf{b}$ of $\mathbf{B}$

Choosing the pivot

A clash of medians

1. Partition array $\mathbf{A}$ into $\lceil \mathbf{n}/\mathbf{5} \rceil$ lists of $\mathbf{5}$ items each.
   $\mathbf{L_1} = \{\mathbf{A[1], A[2], \ldots, A[5]}\}$, $\mathbf{L_2} = \{\mathbf{A[6], \ldots, A[10]}\}$, $\ldots$,
   $\mathbf{L_i} = \{\mathbf{A[5i+1], \ldots, A[5i-4]}\}$, $\ldots$,
   $\mathbf{L_{\lceil n/5 \rceil}} = \{\mathbf{A[5\lceil n/5 \rceil - 4, \ldots, A[n]}\}$.
2. For each $\mathbf{i}$ find median $\mathbf{b_i}$ of $\mathbf{L_i}$ using brute-force in $\mathbf{O(1)}$ time. Total $\mathbf{O(n)}$ time
3. Let $\mathbf{B} = \{\mathbf{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}}\}$
4. Find median $\mathbf{b}$ of $\mathbf{B}$

Lemma

Median of $\mathbf{B}$ *is an* approximate *median of* $\mathbf{A}$. *That is, if* $\mathbf{b}$ *is used a pivot to partition* $\mathbf{A}$, *then* $|\mathbf{A}_{less}| \leq \mathbf{7n/10 + 6}$ and $|\mathbf{A}_{greater}| \leq \mathbf{7n/10 + 6}$.

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

How do we find median of B?

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

How do we find median of $\mathbf{B}$? Recursively!

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    B = [b_1, b_2, ..., b_⌈n/5⌉]
    b = select(B, ⌈n/10⌉)
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

Running time of deterministic median selection

A dance with recurrences

$$\mathbf{T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)}$$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Exercise: show that $T(n) = O(n)$

Median of Medians: Proof of Lemma

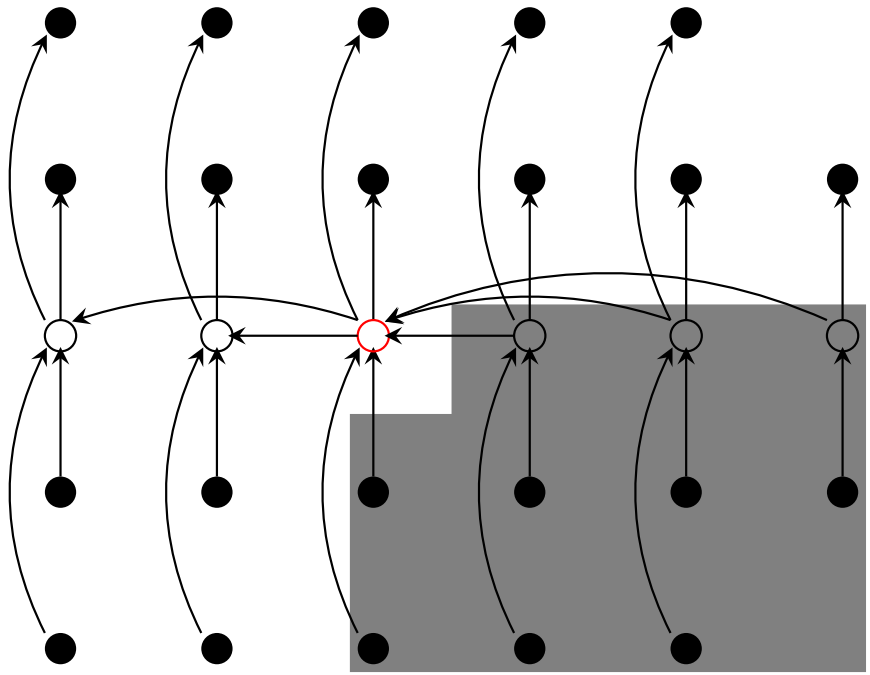


Figure : Shaded elements are all greater than **b**

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* **b**.

Median of Medians: Proof of Lemma

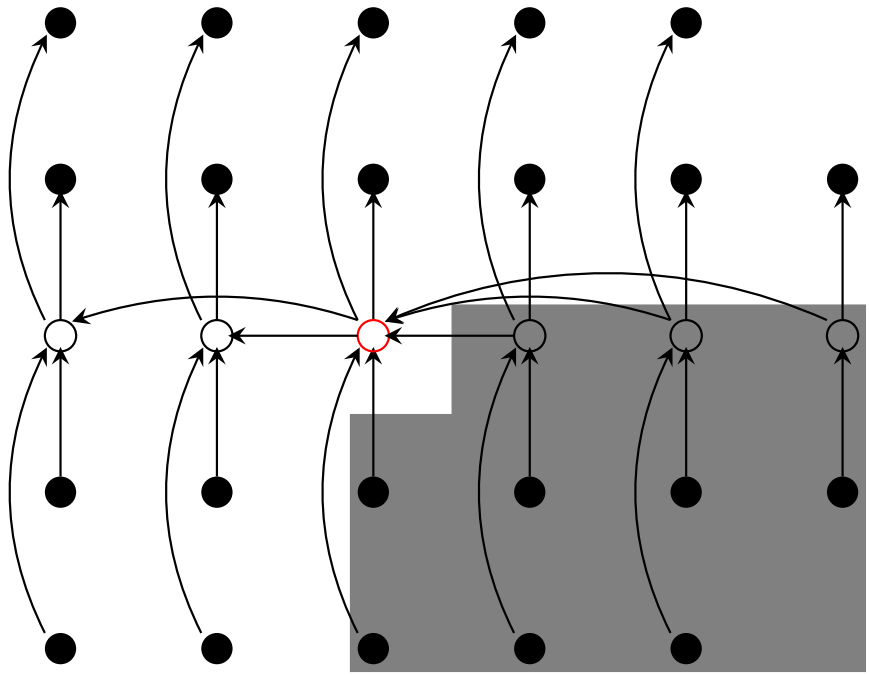


Figure : Shaded elements are all greater than **b**

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* **b**.

Proof.

At least half of the $\lceil \mathbf{n/5} \rceil$ groups have at least 3 elements larger than **b**, except for last group and the group containing **b**. Hence number of elements greater than **b** is:

$$\mathbf{3(\lceil (1/2)\lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6}$$

□

Median of Medians: Proof of Lemma

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* $\mathbf{b}$.

Corollary

$|\mathbf{A}_{less}| \leq \mathbf{7n/10 + 6}$.

Via symmetric argument,

Corollary

$|\mathbf{A}_{greater}| \leq \mathbf{7n/10 + 6}$.

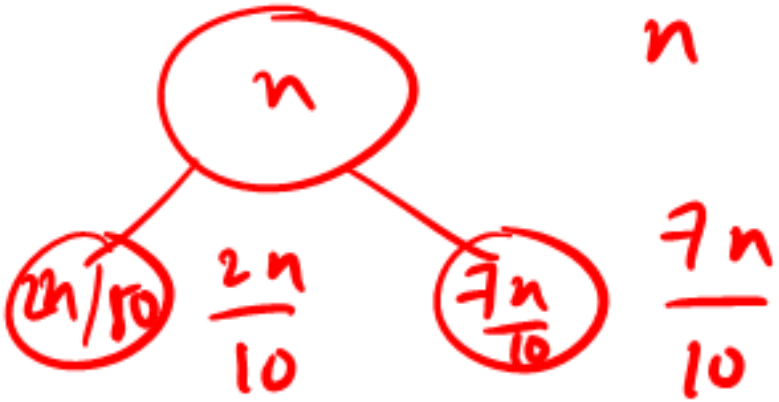
Questions to ponder

1. Why did we choose lists of size **5**? Will lists of size **3** work?
2. Write a recurrence to analyze the algorithm's running time if we choose a list of size **k**.

n

$\frac{9n}{10}$

$\frac{9}{10} \cdot \frac{9}{10} n$



$$T(n) = T\left(\frac{9n}{10}\right) + n$$

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?
All except Vaughn Pratt!

Takeaway Points

1. Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.
2. Recursive algorithms naturally lead to recurrences.
3. Some times one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.