

# Reductions, Recursion and Divide and Conquer

Lecture 10

September 27, 2018

# Part I

## Brief Intro to Algorithm Design and Analysis

# Algorithms and Computing

- 1 Algorithm solves a specific problem.
- 2 Steps/instructions of an algorithm are simple/primitive and can be executed mechanically.
- 3 Algorithm has a finite description; same description for all instances of the problem
- 4 Algorithm implicitly may have state/memory

A computer is a device that

- 1 implements the primitive instructions
- 2 allows for an automated implementation of the entire algorithm by keeping track of state

# Models of Computation vs Computers

- ① Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
- ② Computer: an actual physical device that implements a very specific model of computation

**In this course:** design algorithms in a high-level model of computation.

**Question:** What model of computation will we use to design algorithms?

# Models of Computation vs Computers

- ① Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
- ② Computer: an actual physical device that implements a very specific model of computation

**In this course:** design algorithms in a high-level model of computation.

**Question:** What model of computation will we use to design algorithms?

The standard programming model that you are used to in programming languages such as Java/C++. We have already seen the the Turing Machine model.

# Unit-Cost RAM Model

Informal description:

- 1 Basic data type is an integer number
- 2 Numbers in input fit in a word
- 3 Arithmetic/comparison operations on words take constant time
- 4 Arrays allow random access (constant time to access  $A[i]$ )
- 5 Pointer based data structures via storing addresses in a word

# Example

Sorting: input is an array of  $n$  numbers

- 1 input size is  $n$  (ignore the bits in each number),
- 2 comparing two numbers takes  $O(1)$  time,
- 3 random access to array elements,
- 4 addition of indices takes constant time,
- 5 basic arithmetic operations take constant time,
- 6 reading/writing one word from/to memory takes constant time.

We will usually not allow (or be careful about allowing):

- 1 bitwise operations (and, or, xor, shift, etc).
- 2 floor function.
- 3 limit word size (usually assume unbounded word size).

# Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in several important situations so one has to be careful.

- ① For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two  $n$ -digit numbers, primality etc.
- ② Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by  $2^k$  where  $k$  is word length.
- ③ Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.



# Models used in class

In this course when we design algorithms:

- 1 Assume unit-cost **RAM** by default.
- 2 We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.
- 3 Turing Machines (or some high-level version of it) will be the non-cheating model that we will fall back upon when tricky issues come up.

# What is an algorithmic problem?

**Simplest and robust definition:** An algorithmic problem is simply to compute a function  $f : \Sigma^* \rightarrow \Sigma^*$  over strings of a finite alphabet.

Algorithm  $\mathcal{A}$  solves  $f$  if for all **input strings**  $w$ ,  $\mathcal{A}$  outputs  $f(w)$ .

Typically we are interested in functions  $f : D \rightarrow R$  where  $D \subseteq \Sigma^*$  is the domain of  $f$  and where  $R \subseteq \Sigma^*$  is the range of  $f$ .

We say that  $w \in D$  is an **instance** of the problem. Implicit assumption is that the algorithm, given an arbitrary string  $w$ , can tell whether  $w \in D$  or not. Parsing problem! The **size of the input**  $w$  is simply the length  $|w|$ .

The domain  $D$  depends on what **representation** is used. Can be lead to formally different algorithmic problems.

# Types of Problems

We will broadly see three types of problems.

- 1 **Decision Problem:** Is the input a YES or NO input?  
Example: Given graph  $G$ , nodes  $s, t$ , is there a path from  $s$  to  $t$  in  $G$ ?  
Example: Given a CFG grammar  $G$  and string  $w$ , is  $w \in L(G)$ ?
- 2 **Search Problem:** Find a solution if input is a YES input.  
Example: Given graph  $G$ , nodes  $s, t$ , find an  $s-t$  path.
- 3 **Optimization Problem:** Find a best solution among all solutions for the input.  
Example: Given graph  $G$ , nodes  $s, t$ , find a shortest  $s-t$  path.

# Analysis of Algorithms

Given a problem  $P$  and an algorithm  $\mathcal{A}$  for  $P$  we want to know:

- Does  $\mathcal{A}$  **correctly** solve problem  $P$ ?
- What is the **asymptotic worst-case running time** of  $\mathcal{A}$ ?
- What is the **asymptotic worst-case space** used by  $\mathcal{A}$ .

**Asymptotic running-time analysis:**  $\mathcal{A}$  runs in  $O(f(n))$  time if:

“for all  $n$  and for all inputs  $I$  of size  $n$ ,  $\mathcal{A}$  on input  $I$  terminates after  $O(f(n))$  primitive steps.”

# Big-O Caveats

Suppose you have two algorithms,  $A_1$  which is  $O(n \log n)$  and  $A_2$  which is  $O(n^2)$ . Which one should you pick?

# Big-O Caveats

Suppose you have two algorithms,  $A_1$  which is  $O(n \log n)$  and  $A_2$  which is  $O(n^2)$ . Which one should you pick?

- What are the constants? E.g.,  $T_1(n) = 10^{10} n \log n$ ,  
 $T_2(n) = 0.1 n^2$

# Big-O Caveats

Suppose you have two algorithms,  $A_1$  which is  $O(n \log n)$  and  $A_2$  which is  $O(n^2)$ . Which one should you pick?

- What are the constants? E.g.,  $T_1(n) = 10^{10} n \log n$ ,  
 $T_2(n) = 0.1 n^2$
- Are the bounds tight? E.g., Any algorithm  $O(n \log n)$  is also  $O(n^2)$ . (big  $\Theta$  helps avoid these issues)

# Big-O Caveats

Suppose you have two algorithms,  $A_1$  which is  $O(n \log n)$  and  $A_2$  which is  $O(n^2)$ . Which one should you pick?

- What are the constants? E.g.,  $T_1(n) = 10^{10} n \log n$ ,  
 $T_2(n) = 0.1 n^2$
- Are the bounds tight? E.g., Any algorithm  $O(n \log n)$  is also  $O(n^2)$ . (big  $\Theta$  helps avoid these issues)
- Are we optimizing for the worst case? E.g., there exist exponential worst-case time algorithms that are often used in practice.



# Algorithmic Techniques

- Reduction to known problem/algorithm
- Recursion, divide-and-conquer, dynamic programming
- Graph algorithms to use as basic reductions
- Greedy

Some advanced techniques not covered in this class:

- Combinatorial optimization
- Linear and Convex Programming, more generally continuous optimization method
- Advanced data structure
- Randomization
- Many specialized areas

# 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

## October 1

A 374 student wakes up to find a small fire in their room. The student quickly grabs a bucket and extinguishes the fire, then goes to the exam and aces it.

# Reduction

Reducing problem  $A$  to problem  $B$ :

- 1 Algorithm for  $A$  uses algorithm for  $B$  as a black box

## October 1

A 374 student wakes up to find a small fire in their room. The student quickly grabs a bucket and extinguishes the fire, then goes to the exam and aces it.

## November 12

Immediately upon waking, the student lights a small fire in the room and then reuses the algorithm from October 1

# 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)

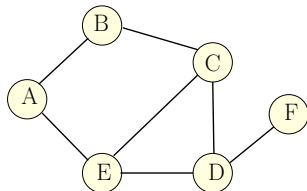
**Example:** Distinct Elements reduces to Sorting in  $O(n)$  time

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

# Maximum Independent Set in a Graph

## Definition

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

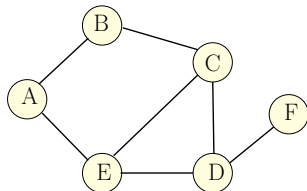


Some independent sets in graph above:

# Maximum Independent Set in a Graph

## Definition

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



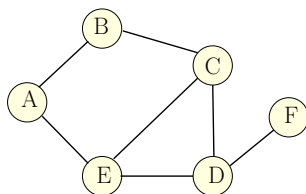
Some independent sets in graph above:  
 $\{A, C, F\}, \{B, E, F\}, \{B, D\}$



# Maximum Independent Set Problem

Input Graph  $G = (V, E)$

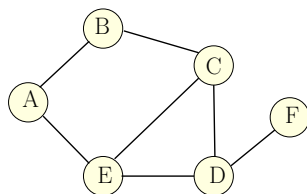
Goal Find maximum sized independent set in  $G$



# Maximum Weight Independent Set Problem

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

**Goal** Find maximum weight independent set in  $G$

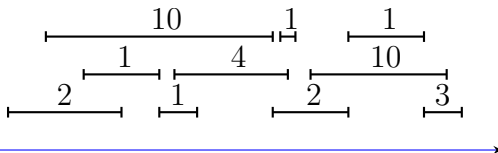


# Weighted Interval Scheduling

**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!

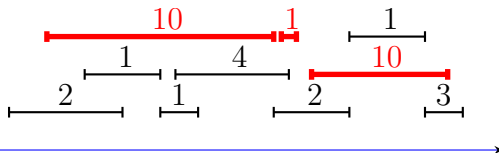


# Weighted Interval Scheduling

**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!

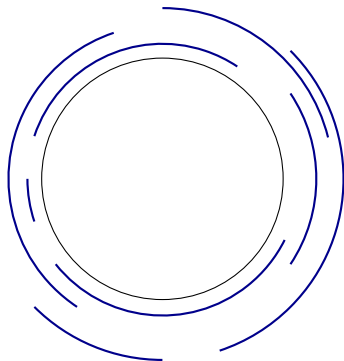


# Reduction from Interval Scheduling to MIS

**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  $\mathcal{C}$ )
  cur-max = 0
  for each arc  $C \in \mathcal{C}$  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 =  $\max\{\text{cur-max}, w_C\}$ 
  end for
  return cur-max
```

# Reductions

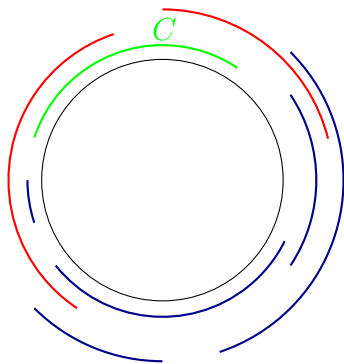
**Question:** Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

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

```
MaxWeightIndependentArcs(arcs  $\mathcal{C}$ )
  cur-max = 0
  for each arc  $C \in \mathcal{C}$  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 =  $\max\{\text{cur-max}, w_C\}$ 
  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

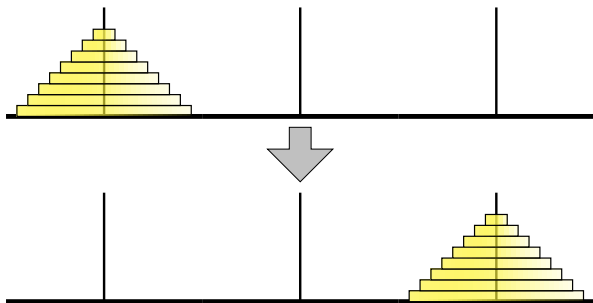
**Recursion:** a special case of reduction

- 1 reduce problem to a smaller instance of itself
  - 2 self-reduction
- 
- 1 Problem instance of size  $n$  is reduced to one or more instances of size  $n - 1$  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



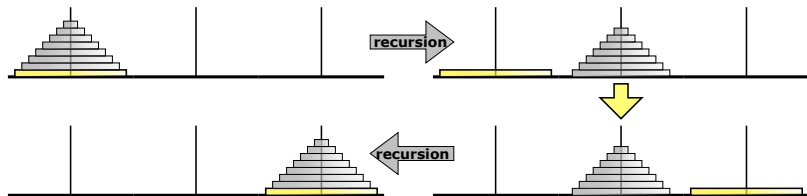
The Tower of Hanoi puzzle

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)
```

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

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

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

# 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  $A[1 \dots n]$

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

# Merge Sort [von Neumann]

## MergeSort

- 1 **Input:** Array  $A[1 \dots n]$

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

- 2 Divide into subarrays  $A[1 \dots m]$  and  $A[m + 1 \dots n]$ , where  $m = \lfloor n/2 \rfloor$

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

# Merge Sort [von Neumann]

## MergeSort

- 1 **Input:** Array  $A[1 \dots n]$

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

- 2 Divide into subarrays  $A[1 \dots m]$  and  $A[m + 1 \dots n]$ , where  $m = \lfloor n/2 \rfloor$

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

- 3 Recursively **MergeSort**  $A[1 \dots m]$  and  $A[m + 1 \dots n]$

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

# Merge Sort [von Neumann]

## MergeSort

- 1 **Input:** Array  $A[1 \dots n]$

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

- 2 Divide into subarrays  $A[1 \dots m]$  and  $A[m + 1 \dots n]$ , where  $m = \lfloor n/2 \rfloor$

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

- 3 Recursively **MergeSort**  $A[1 \dots m]$  and  $A[m + 1 \dots n]$

*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  $A[1 \dots n]$

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

- 2 Divide into subarrays  $A[1 \dots m]$  and  $A[m + 1 \dots n]$ , where  $m = \lfloor n/2 \rfloor$

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

- 3 Recursively **MergeSort**  $A[1 \dots m]$  and  $A[m + 1 \dots n]$

*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.

```
function MERGESORT(A[1..n])  
  if  $n > 1$  then  
     $m \leftarrow \lfloor n/2 \rfloor$   
    MERGESORT(A[1..m])  
    MERGESORT(A[m+1..n])  
    MERGE(A[1..n],m)
```

```
function MERGE(A[1..n], m)  
   $i \leftarrow 1$   
   $j \leftarrow m + 1$   
  for  $k \leftarrow 1$  to  $n$  do  
    if  $j > n$  or ( $i \leq m$  and  $A[i] < A[j]$ ) then  
       $B[k] \leftarrow A[i]$   
       $i \leftarrow i + 1$   
    else  
       $B[k] \leftarrow A[j]$   
       $j \leftarrow j + 1$   
  for  $k \leftarrow 1$  to  $n$  do  
     $A[k] \leftarrow B[k]$ 
```

# 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?

# 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  $n$  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  $k$  the following hold:

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

# Running Time

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

# Running Time

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

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

# Running Time

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

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

What do we want as a solution to the recurrence?

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

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

# 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  $k$  arrays of size  $n/k$  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.

# 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.

# 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.

# Quick Sort: Example

- 1 array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- 2 pivot: 16

# Time Analysis

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

# 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,  $T(n) = O(n \log n)$ .

# 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.

# 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  $A$  of  $n$  numbers and number  $x$

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

```
BinarySearch( $A[a..b]$ ,  $x$ ):  
  if ( $b - a < 0$ ) return NO  
   $mid = A[\lfloor (a + b)/2 \rfloor]$   
  if ( $x = mid$ ) return YES  
  if ( $x < mid$ )  
    return BinarySearch( $A[a.. \lfloor (a + b)/2 \rfloor - 1]$ ,  $x$ )  
  else  
    return BinarySearch( $A[\lfloor (a + b)/2 \rfloor + 1..b]$ ,  $x$ )
```

# Binary Search in Sorted Arrays

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

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

```
BinarySearch( $A[a..b]$ ,  $x$ ):  
  if ( $b - a < 0$ ) return NO  
   $mid = A[\lfloor (a + b)/2 \rfloor]$   
  if ( $x = mid$ ) return YES  
  if ( $x < mid$ )  
    return BinarySearch( $A[a..\lfloor (a + b)/2 \rfloor - 1]$ ,  $x$ )  
  else  
    return BinarySearch( $A[\lfloor (a + b)/2 \rfloor + 1..b]$ ,  $x$ )
```

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  $v$ ?

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

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

# Example

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

**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  $U$  be sum of all edge lengths in  $G$ .
- 2 Let  $L$  be the length of the smallest edge in  $G$ .
- 3  $s$ - $t$  shortest path length is at most  $U$  and at least  $L$ .
- 4 Apply binary search on the interval  $[L, U]$  via the algorithm for the decision problem.
- 5  $O(\log(U - L))$  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  $T(n) = 2T(n/2) + n/\log n$  for  $n > 2$ ,  $T(2) = 1$ .

# Recurrence: Example I

- 1 Consider  $T(n) = 2T(n/2) + n/\log n$  for  $n > 2$ ,  $T(2) = 1$ .
- 2 Construct recursion tree, and observe pattern.  $i$ th 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  $T(n) = T(\sqrt{n}) + 1$  for  $n > 2$ ,  $T(2) = 1$ .

# Recurrence: Example II

① Consider  $T(n) = T(\sqrt{n}) + 1$  for  $n > 2$ ,  $T(2) = 1$ .

② What is the depth of recursion?

$$\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, O(1).$$

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  $T(n) = \sqrt{n}T(\sqrt{n}) + n$  for  $n > 2$ ,  $T(2) = 1$ .

# Recurrence: Example III

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



# Recurrence: Example IV

- 1 Consider  $T(n) = T(n/4) + T(3n/4) + n$  for  $n > 4$ .  
 $T(n) = 1$  for  $1 \leq n \leq 4$ .

# Recurrence: Example IV

- 1 Consider  $T(n) = T(n/4) + T(3n/4) + n$  for  $n > 4$ .  
 $T(n) = 1$  for  $1 \leq n \leq 4$ .
- 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  $n$ . Total work in any level without leaves is exactly  $n$ .
- 4 Highest leaf is at level  $\log_4 n$  and lowest leaf is at level  $\log_{4/3} n$
- 5 Thus,  $n \log_4 n \leq T(n) \leq n \log_{4/3} n$ , which means  
 $T(n) = \Theta(n \log n)$