Reductions, Recursion and Divide and Conquer

Lecture 08

September 17, 2015

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++. Later we will se a theoretical model called the Turing Machine.

Unit-Cost RAM Model

Informal description:

1. Basic data type is an integer/floating point 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 $\mathbf{n}$-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 $\mathbf{2^k}$ where $\mathbf{k}$ 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. We will discuss Turing Machines later.

What is an algorithmic problem?

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

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

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

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

The domain $\mathbf{D}$ 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~~ output a YES or NO input?
   Example: Given graph $\mathbf{G}$, nodes $\mathbf{s}, \mathbf{t}$, is there a path from $\mathbf{s}$ to $\mathbf{t}$ in $\mathbf{G}$?
   Example: Given a CFG grammar $\mathbf{G}$ and string $\mathbf{w}$, is $\mathbf{w} \in \mathbf{L}(\mathbf{G})$?
2. Search Problem: Find a *solution* if input is a YES input.
   Example: Given graph $\mathbf{G}$, nodes $\mathbf{s}, \mathbf{t}$, find an $\mathbf{s}$-$\mathbf{t}$ path.
3. Optimization Problem: Find a *best* solution among all solutions for the input.
   Example: Given graph $\mathbf{G}$, nodes $\mathbf{s}, \mathbf{t}$, find a shortest $\mathbf{s}$-$\mathbf{t}$ 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 **aysmptotic 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."

Algorithmic Techniques

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

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 $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

UNIQUENESS: Distinct Elements Problem

Problem Given an array $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

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 $\mathbf{A}$ of $\mathbf{n}$ integers, are there any *duplicates* in $\mathbf{A}$?

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)

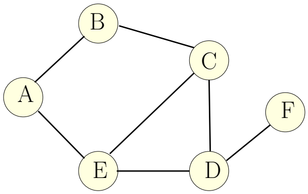
Example: Distinct Elements reduces to Sorting in $\mathbf{O(n)}$ time

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 $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a subset of nodes $\mathbf{S} \subseteq \mathbf{V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u}, \mathbf{v} \in \mathbf{S}$ then $(\mathbf{u}, \mathbf{v}) \not\in \mathbf{E}$.

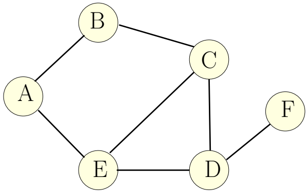


Some independent sets in graph above: {A, C, F}, {B, D}
{D}

Maximum Independent Set Problem

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

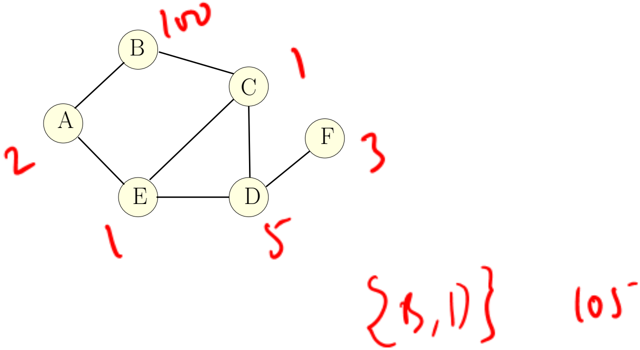
Goal Find maximum sized independent set in $\mathbf{G}$



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 $\mathbf{G}$

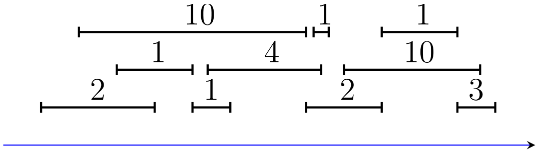


Weighted Interval Scheduling

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!

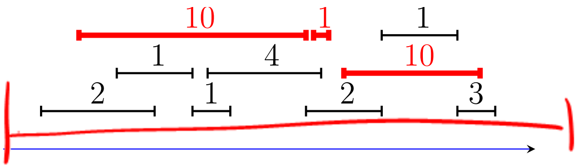


Weighted Interval Scheduling

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!



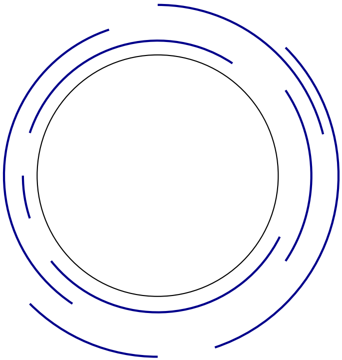
Reduction from Interval Scheduling to MIS

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

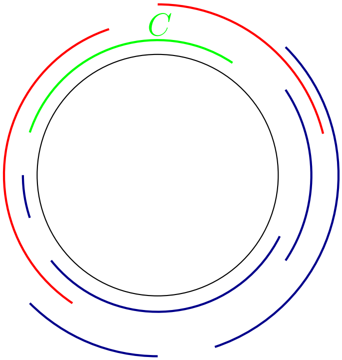
Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

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

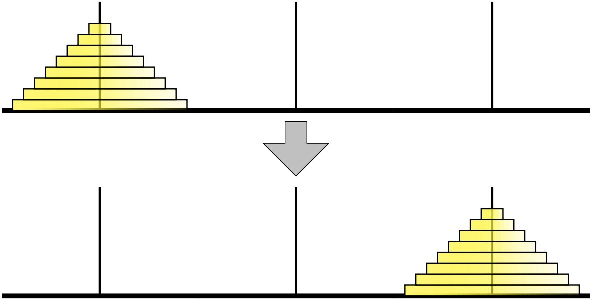
1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

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

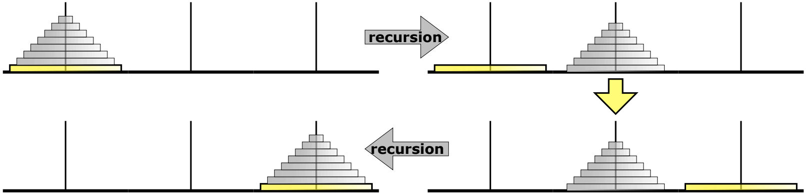
Tower of Hanoi



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

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

Analysis

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

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

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

A L G O R 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

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

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ 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 $\mathbf{n/k}$ 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 $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \le 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.
    1. Theoretically, median can be found in linear time.

Time Analysis

1. Let $\mathbf{k}$ 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.
3. Typically, pivot is the first or last element of array. Then,

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

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

Part IV

Binary Search

Binary Search in Sorted Arrays

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

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

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[⌊(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 **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[⌊(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 **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 $\mathbf{U}$ 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

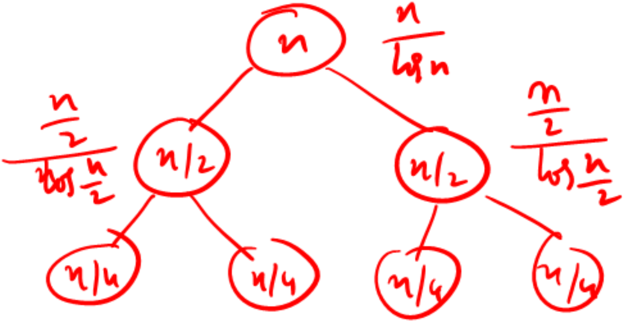
Two general methods:

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

$T(2) = 1$



$n \log \log n$

$$\frac{n}{\log n} + 2 \cdot \frac{\frac{n}{2}}{\log \frac{n}{2}} + 4 \cdot \frac{\frac{n}{4}}{\log \frac{n}{4}} + \cdots$$

$$n\left[\frac{1}{\log n} + \frac{1}{\log n - 1} + \frac{1}{\log n - 2} + \cdots\right]$$

Recurrence: Example I

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

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

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

$T(2) = 1$

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

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

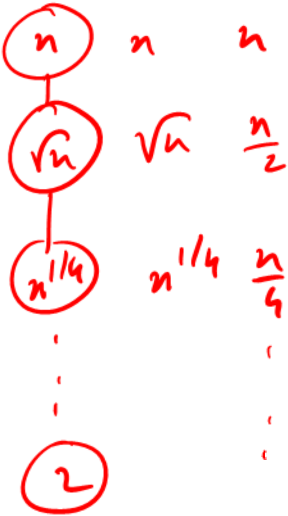
$\frac{1}{2^d} \log n = 1$

$2^d = \log n$

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

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

$$= O(\quad)$$

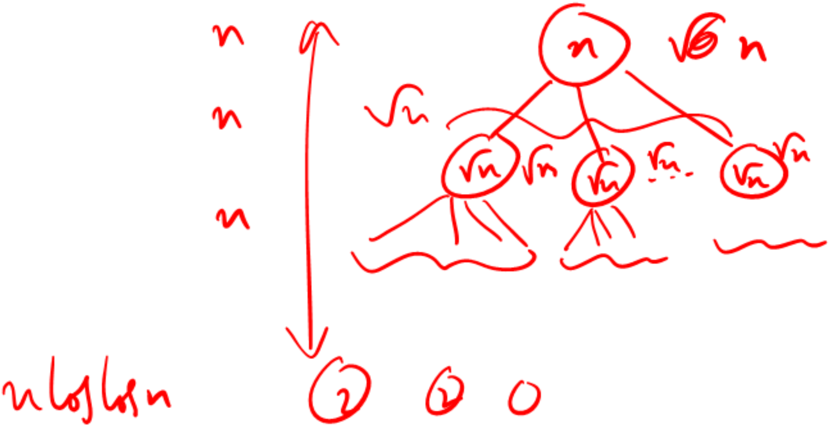


Recurrence: Example II

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

Recurrence: Example III

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

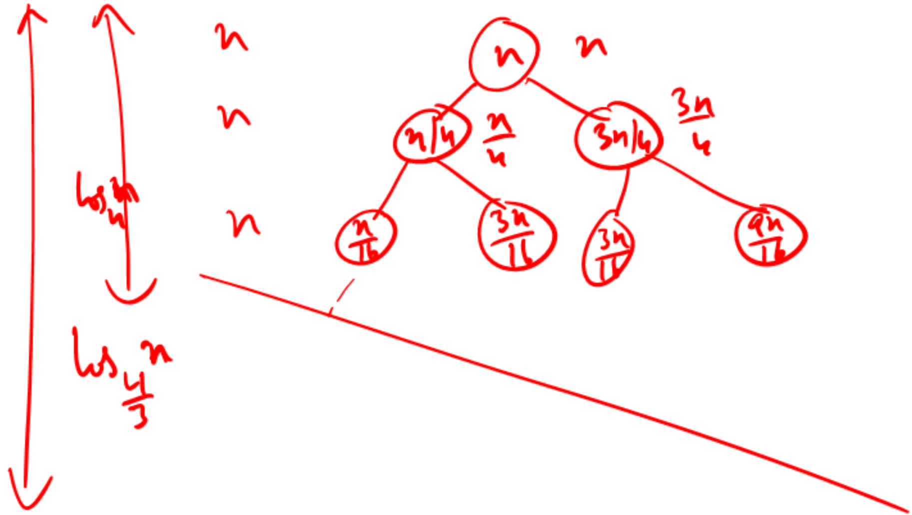


Recurrence: Example III

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



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 $\mathbf{T(n) = \Theta(n \log n)}$