

Pre-lecture brain teaser

We talked a lot about languages representing problems. Consider the problem of adding two numbers. What language class does it belong to?

ECE-374-B: Lecture 9 - Recursion, Sorting and Recurrences

Lecturer: Nickvash Kani

University of Illinois at Urbana-Champaign

Pre-lecture brain teaser

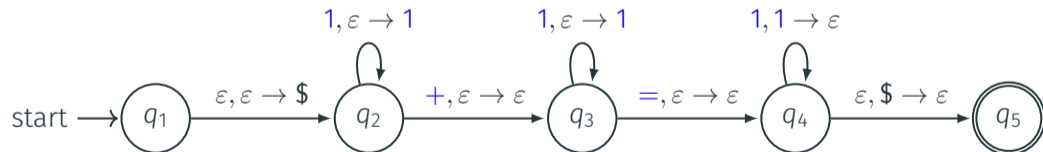
We talked a lot about languages representing problems. Consider the problem of adding two numbers. What language class does it belong to?

Pre-lecture brain teaser

Let's say we are adding two unary numbers.

$$3 + 4 = 7 \rightarrow 111 + 1111 = 1111111 \quad (1)$$

Seems like we can make a PDA that considers

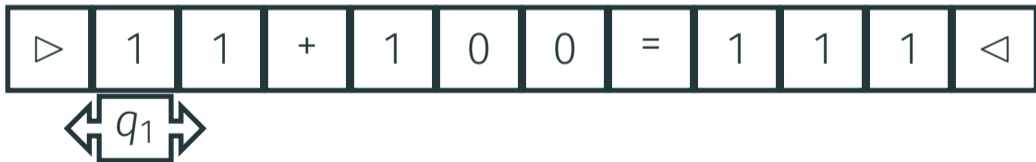


Pre-lecture brain teaser

What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \quad (2)$$

At least context-sensitive b/c we can build a finite Turing machine which takes in the encoding

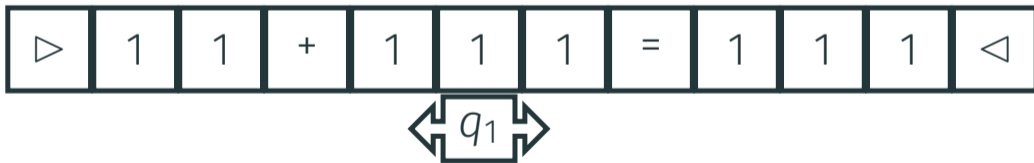


Pre-lecture brain teaser

What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \quad (3)$$

Computes value on left hand side



Pre-lecture brain teaser

What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \quad (4)$$

And compares it to the value on the right..



New Course Section: Introductory algorithms

Brief intro to the RAM model

Algorithms and Computing

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

A computer is a device that

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

Models of Computation vs Computers

- Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
- 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

- Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
- 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 Turing Machine model.

Unit-Cost RAM Model

Informal description:

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

Example

Sorting: input is an array of n numbers

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

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

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

What is an algorithmic problem?

What is an algorithmic problem?

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 $f(w)$.

Types of Problems

We will broadly see three types of problems.

- **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 $w \in L(G)$?
- **Search Problem:** Find a solution if input is a YES input.
Example: Given graph G , nodes s, t , find an s - t path.
- **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.”

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

Reductions: Reducing problem A to
problem B :

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

- **Positive direction:** Algorithm for B implies an algorithm for A
- **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

- **Positive direction:** Algorithm for B implies an algorithm for A
- **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

- An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
- 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.

Recursion as self reductions

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a smaller instance of itself
- self-reduction

Recursion

Reduction: reduce one problem to another

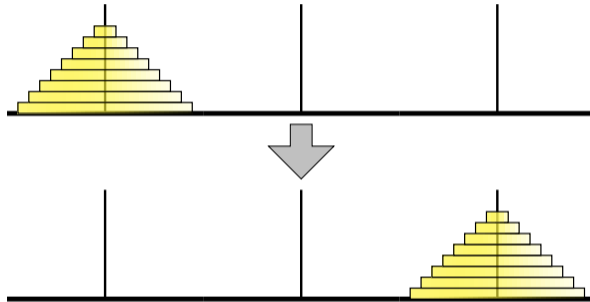
Recursion: a special case of reduction

- reduce problem to a smaller instance of itself
- self-reduction
- Problem instance of size n is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as base cases

Recursion

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

Tower of Hanoi



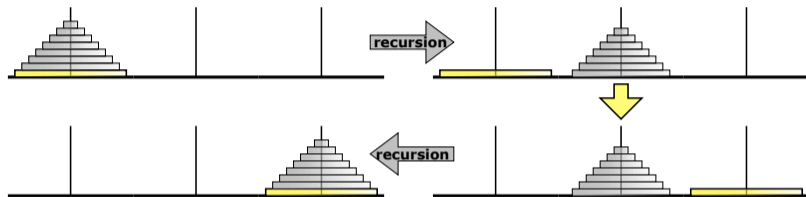
The Tower of Hanoi puzzle

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

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

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

Merge Sort

Sorting

Input Given an array of n elements

Goal Rearrange them in ascending order

MergeSort

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

A L G O R I T H M S

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

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

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

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

- Use a new array C to store the merged array
- 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

- Use a new array C to store the merged array
- 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

- Use a new array C to store the merged array
- 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

- Use a new array *C* to store the merged array
- 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

- Use a new array *C* to store the merged array
- 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

- Use a new array C to store the merged array
- 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

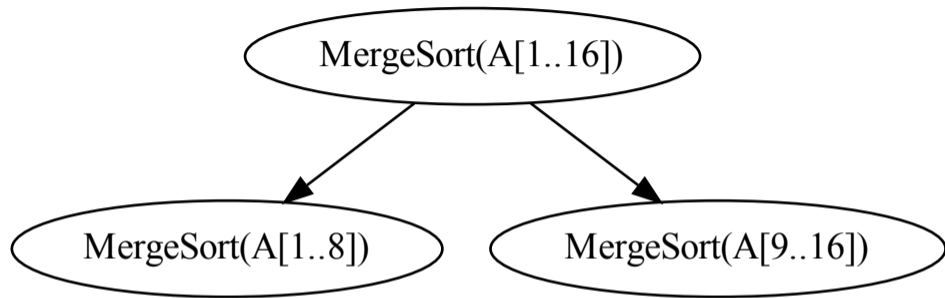
- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical.

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

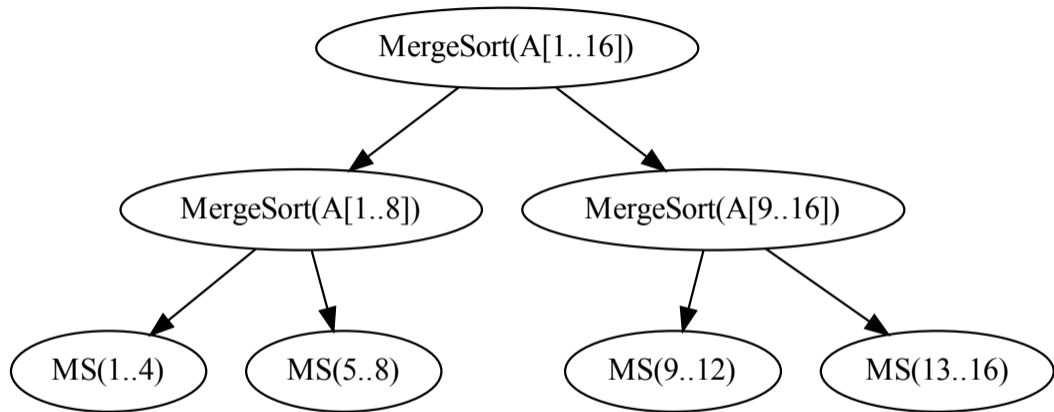
```
MERGE(A[1..n], m):  
   $i \leftarrow 1; j \leftarrow m + 1$   
  for  $k \leftarrow 1$  to  $n$   
    if  $j > n$   
       $B[k] \leftarrow A[i]; i \leftarrow i + 1$   
    else if  $i > m$   
       $B[k] \leftarrow A[j]; j \leftarrow j + 1$   
    else if  $A[i] < A[j]$   
       $B[k] \leftarrow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

Running time analysis of merge-sort: Recursion tree method

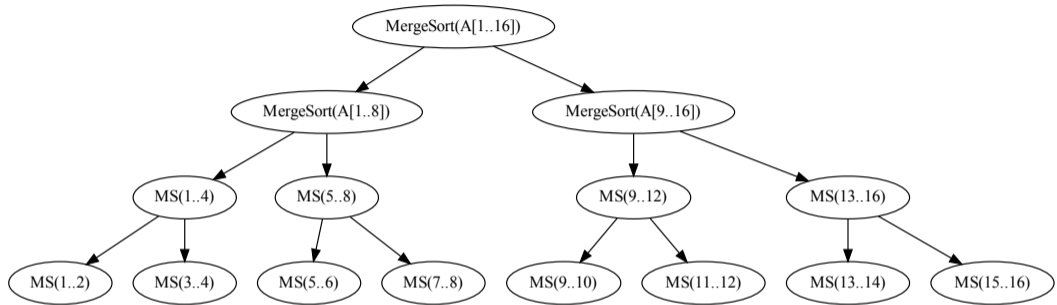
MergeSort(A[1..16])



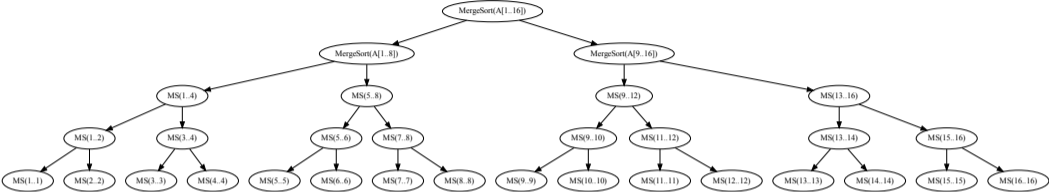
Recursion tree



Recursion tree



Recursion tree

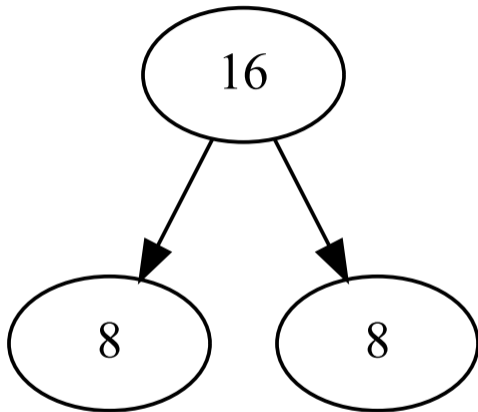
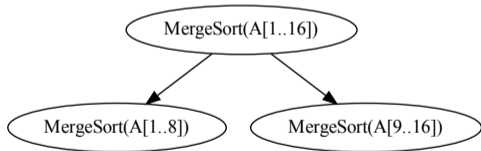


Recursion tree: subproblem sizes

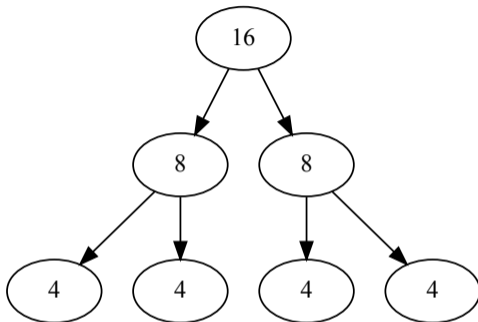
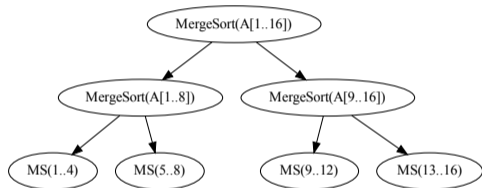
MergeSort(A[1..16])

16

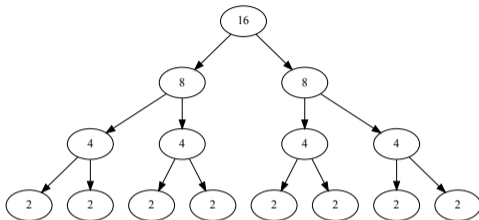
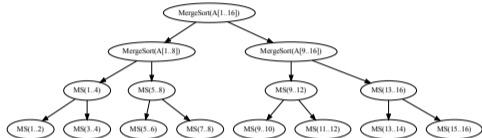
Recursion tree: subproblem sizes



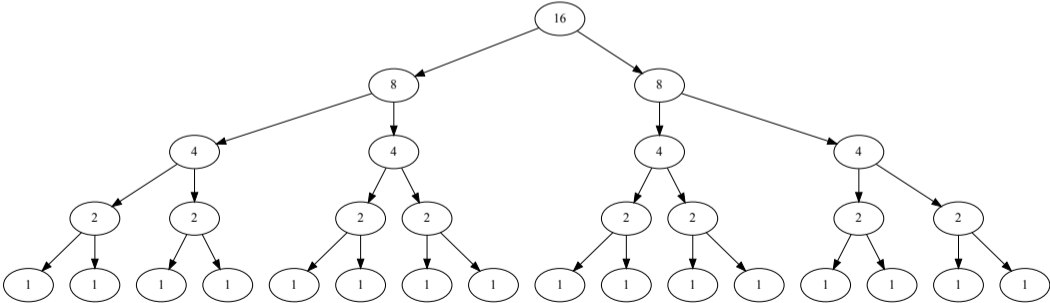
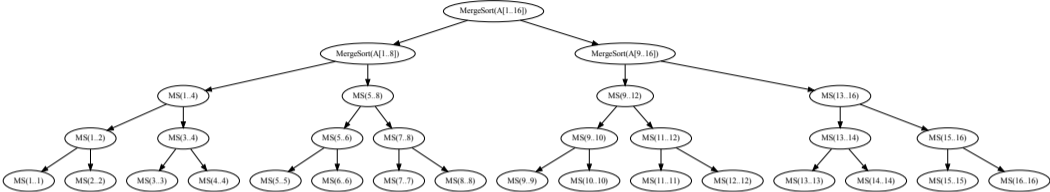
Recursion tree: subproblem sizes



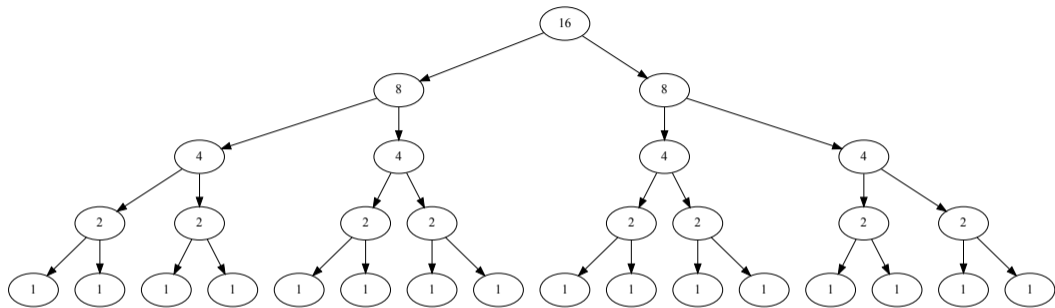
Recursion tree: subproblem sizes



Recursion tree: subproblem sizes



Recursion tree: Total work?



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 $T(n) = \Theta(f(n))$.

- $T(n) = O(f(n))$ - upper bound
- $T(n) = \Omega(f(n))$ - lower bound

Solving Recurrences: Some Techniques

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

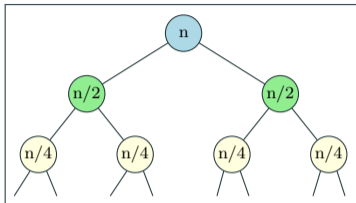
Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- **Recursion tree method** — imagine the computation as a tree
- **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.”

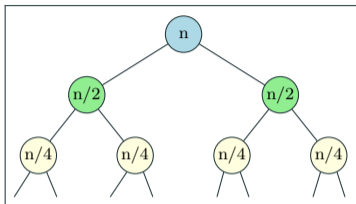
Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Recursion Trees : MergeSort: n is a power of 2



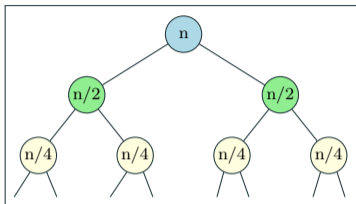
- Unroll the recurrence. $T(n) = 2T(n/2) + cn$

Recursion Trees : MergeSort: n is a power of 2



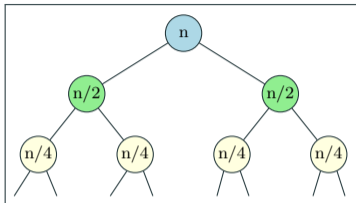
- Unroll the recurrence. $T(n) = 2T(n/2) + cn$
- Identify a pattern.

Recursion Trees : MergeSort: n is a power of 2



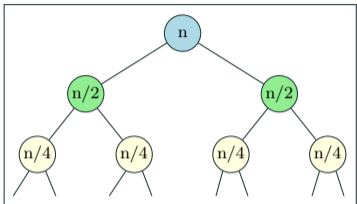
- Unroll the recurrence. $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the i level total work is cn .

Recursion Trees : MergeSort: n is a power of 2



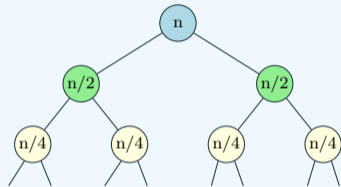
- Unroll the recurrence. $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the i level total work is cn .
- Sum over all levels.

Recursion Trees : MergeSort: n is a power of 2

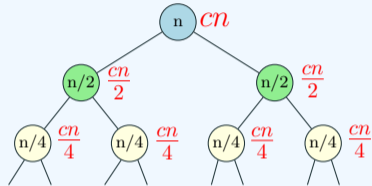


- Unroll the recurrence. $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the i level total work is cn .
- Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.

Recursion Trees

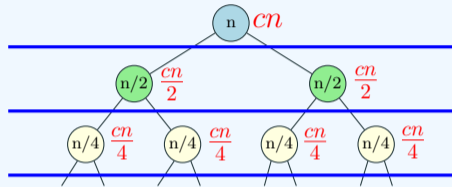


Recursion Trees



Work in each node

Recursion Trees



Work in each node

Recursion Trees

$$\log n \left\{ \begin{array}{l} \text{-----} = cn \\ \frac{cn}{2} + \frac{cn}{2} = cn \\ \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn \\ \vdots \\ \text{-----} = cn \end{array} \right.$$

Recursion Trees

$$\begin{array}{l} \log n \left\{ \begin{array}{l} \text{-----} = cn \\ \frac{cn}{2} + \frac{cn}{2} = cn \\ \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn \\ \vdots \\ \text{-----} = cn \end{array} \right. \\ = cn \log n = O(n \log n) \end{array}$$

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?

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[[(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 $n/2^k$