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

Instructor: Abhishek Kumar Umrawal

February 20, 2024

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?

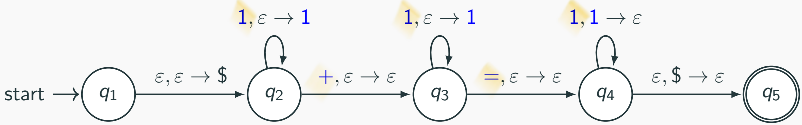
Pre-lecture brain teaser

tick marks

Let's say we are adding two unary numbers.

$$3+4=7 \rightarrow \boxed{111+1111=1111111} \tag{1}$$

Seems like we can make a PDA that considers



context-free !

Pre-lecture brain teaser

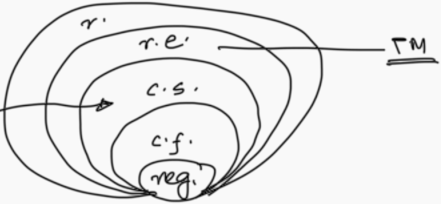
What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \tag{2}$$

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

$\triangleright$	1	1	+	1	0	0	=	1	1	1	$\triangleleft$

q_1

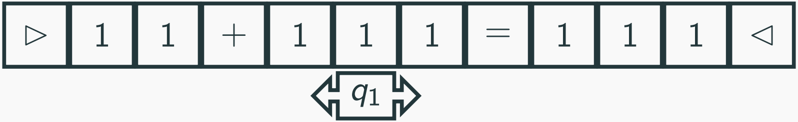


Pre-lecture brain teaser

What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \tag{3}$$

Computes value on left hand side



Pre-lecture brain teaser

What if we wanted add two binary numbers?

$$3 + 4 = 7 \rightarrow 11 + 100 = 111 \tag{4}$$

And compares it to the value on the right..

$\triangleright$	1	1	+	1	1	1	=	1	1	1	$\triangleleft$

q_1

turingmachine.io

New Course Section: Introductory algorithms

Learning Objectives

At the end of the lecture, you should be able to understand

- the idea of an algorithm and algorithmic problems,
- how to reduce a problem into another,
- the design and analysis of recursive algorithms, and
- some example recursive algorithms for sorting and searching.

Brief intro to the Random Access Machine (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

E.g. Turing Machine

- 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

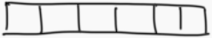
In this course: design algorithms in a high-level model of computation.

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.

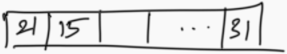
Unit-Cost RAM Model

51 → binary string.

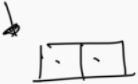


Informal description:

- 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



$A[1] = 15$



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 $f : \Sigma^* \to \Sigma^*$ over strings of a finite alphabet.

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

$w \in \Sigma^*$

E.g. Adding two numbers.

num1 + num2 = num3.

f(num1, num2) = num3

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.

input

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."

$O(f(n))$

$T(n)$ is said to be $O(f(n))$ if

$T(n) \leq c f(n)$ for some $c > 0$

"constant"

E.g. $T(n) = n^2$

Check if $T(n) = O(n^2)$!

$T(n) \leq c \cdot n^2$

Take $c = 1 \Rightarrow T(n) \leq n^2$; $T(n) = n^2$ (Given)

Check if $T(n) = O(n^5)$!

$T(n) \leq c n^5$

$n^2 \leq c n^5$ (?) $c = 1$

$\Rightarrow T(n) = O(n^5)$

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

Reduction

Reducing problem A to problem B:

- Algorithm for A uses algorithm for B as a black box.

Reduction

Reducing problem A to problem B:

- 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.

Reduction

Reducing problem A to problem B:

- 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 its trunk shut until it turns blue, and then shoot it with the blue elephant gun.

Reduction

Reducing problem A to problem B:

- 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. ($100)

Q: How do you hunt a red elephant?

A: Hold its trunk shut until it turns blue, and then shoot it with the blue elephant gun.

($200)

($100)

Q: How do you shoot a white elephant?

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

($500)

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: $O(n^2)$

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
```

$= O(n) +$ Time taken in sorting

$\hookrightarrow O(n \log n)$

$= O(n) + O(n \log n)$

How?

$= O(n \log n)$

$\boxed{O(n^2) \geq O(n \log n) \text{ because } \log n \leq n}$

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)

$O(n^2)$ $O(n \log n)$

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

(RIY)

Recursion: a special case of reduction

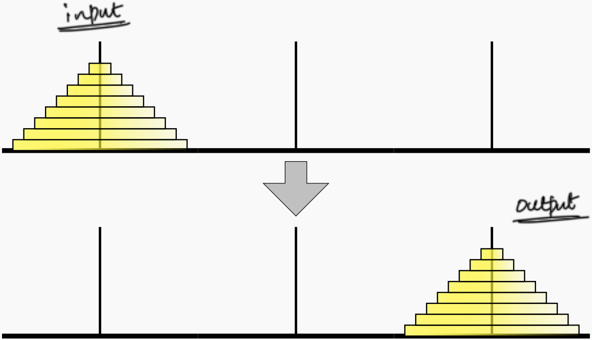
- reduce problem to a smaller instance of itself
- self-reduction

- 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



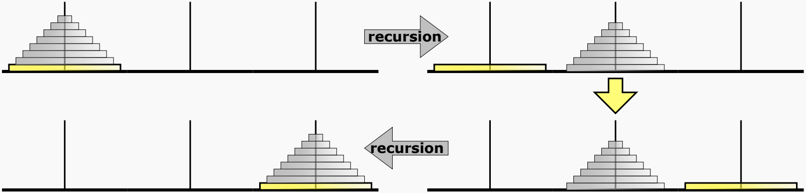
The Tower of Hanoi puzzle

Move stack of n disks from peg 1 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)
```

$$T(n) = T(n-1) + \text{Move disk} + T(n-1)$$

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

" Recursive Algo " →

```
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

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

Recurrence relation

$$T(n) = 2T(n-1) + 1 \qquad n > 1$$
$$T(1) = 1$$

$$T(n) = 2[2T(n-2) + 1] + 1$$

$$T(n) = 2^2 T(n-2) + 2 + 1$$
$$= 2^2[2T(n-3) + 1] + 2 + 1 = 2^3 T(n-3) + 2^2 + 2^1 + 1^0$$

Analysis

Try it!

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

Geometric series

$T(n) = O(2^n)$

Merge Sort

Sorting

Input Given an array of n elements

Goal Rearrange them in ascending order

MergeSort

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

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

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 $m = \lfloor n/2 \rfloor$

$$A \ L \ G \ O \ R \qquad I \ T \ H \ M \ S$$

MergeSort

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

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

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

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

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

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

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 $m = \lfloor n/2 \rfloor$

$$A \ L \ G \ O \ R \qquad I \ T \ H \ M \ S$$

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

$$A \ G \ L \ O \ R \qquad H \ I \ M \ S \ T$$

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 $m = \lfloor n/2 \rfloor$

$$A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$$

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

"Recursion"

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

4. Merge the sorted arrays

"Recursively"

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

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

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

$$A$$

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

$$A \; G \; L \; O \; R \qquad H \; I \; M \; S \; T$$

$$A \; G$$

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

$$A \; G \; L \; O \; R \qquad H \; I \; M \; S \; T$$

$$A \; G \; H$$

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

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

$A\ G\ H\ I$

n $\boxed{O(n)}$

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

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

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

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

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

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)
```

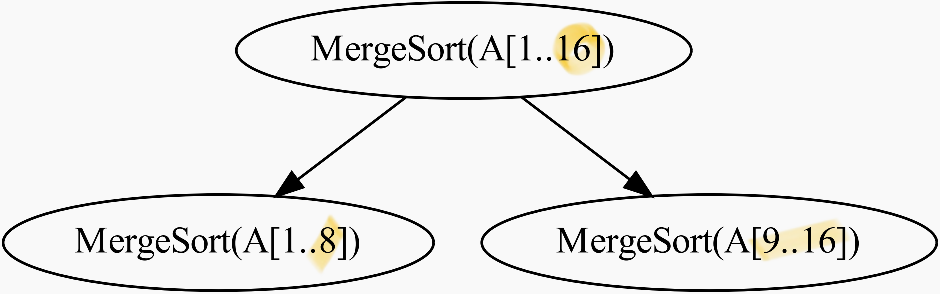
```
Merge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1

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

Running time analysis of merge-sort: Recursion tree method

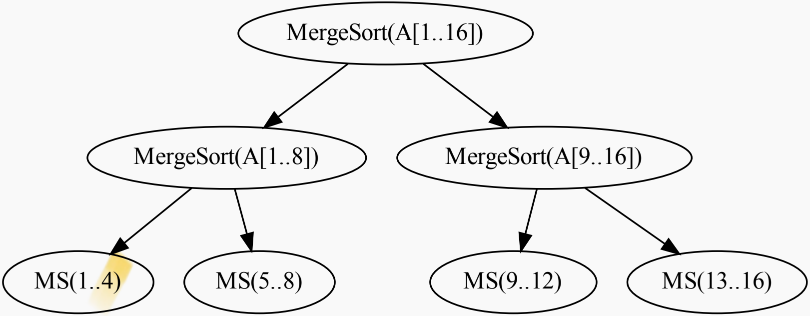
Recursion tree

MergeSort(A[1..16])

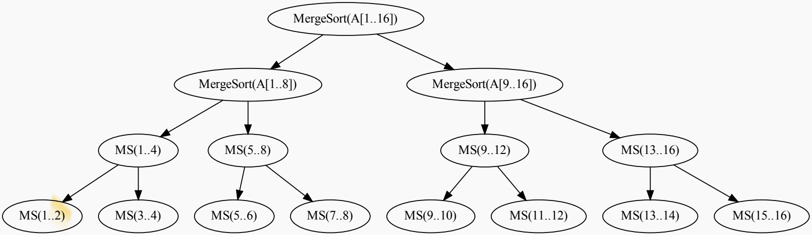
Recursion tree



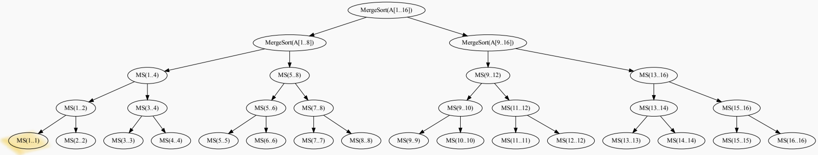
Recursion tree



Recursion tree



Recursion tree



$\boxed{n} \rightarrow 16$ div by 2

8 " O

4 "

2 "

1

$\boxed{4}$ steps

$4 = \log_2 16$

$2^4 = 16$

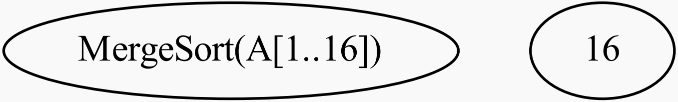
n /2

/2

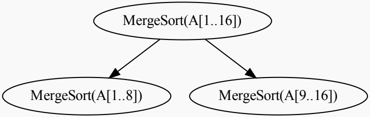
$\sqrt{n}$? (X)

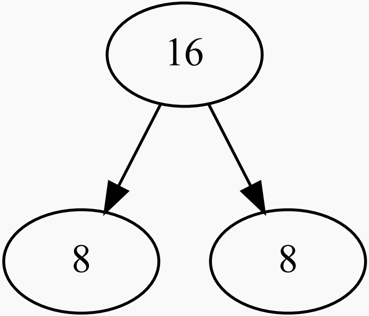
$\log_2 n$

Recursion tree: subproblem sizes

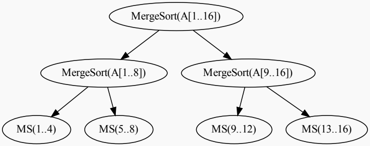


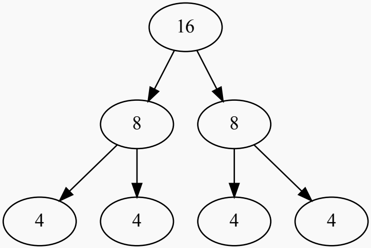
Recursion tree: subproblem sizes



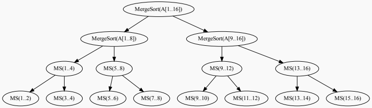


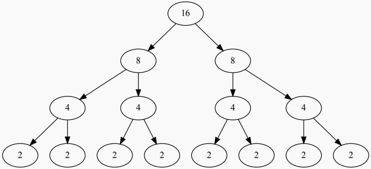
Recursion tree: subproblem sizes



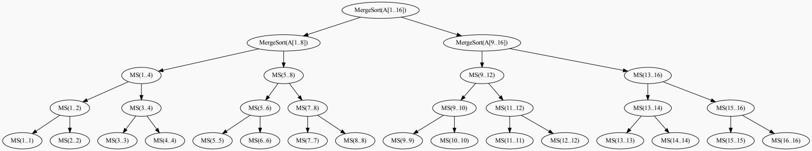


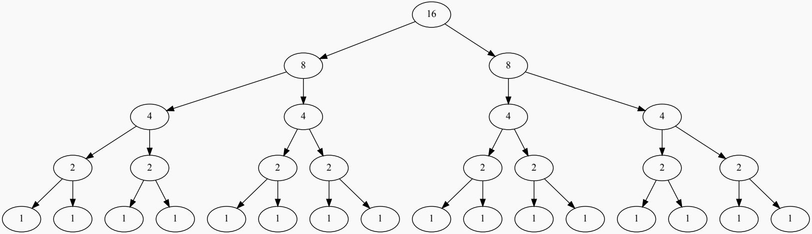
Recursion tree: subproblem sizes



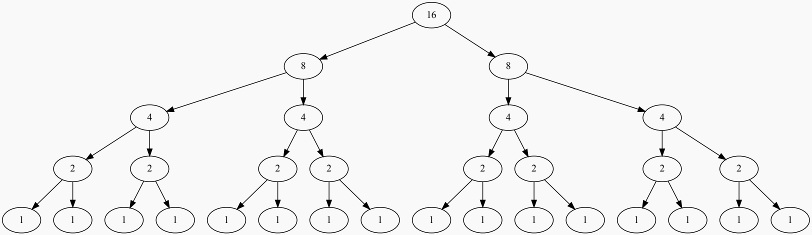


Recursion tree: subproblem sizes





Recursion tree: Total work?



Running Time

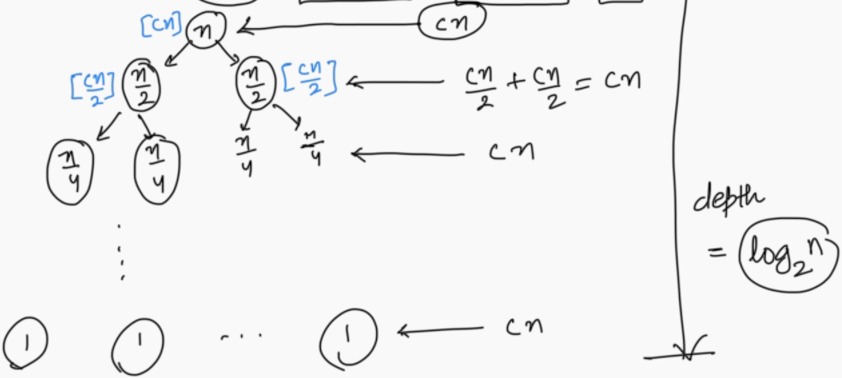
$T(n)$: time for merge sort to sort an n element array

Running Time

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

Merge!

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



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

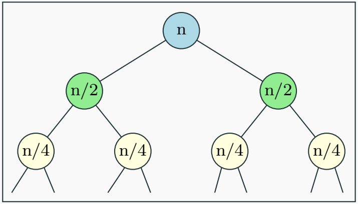
Solving Recurrences: Some Techniques

- 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!

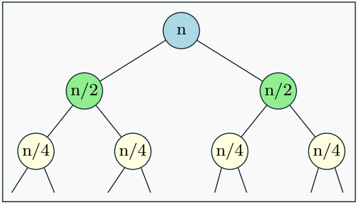
Recursion Trees : MergeSort: n is a power of 2



- Unroll the recurrence.

  $T(n) = 2T(n/2) + cn$

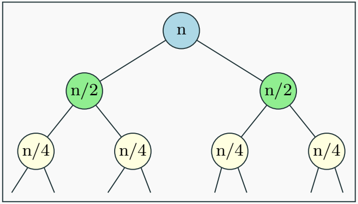
Recursion Trees : MergeSort: n is a power of 2



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

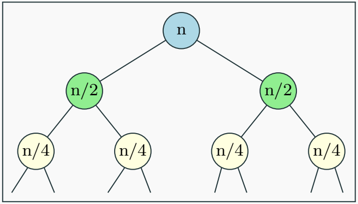
.

Recursion Trees : MergeSort: n is a power of 2



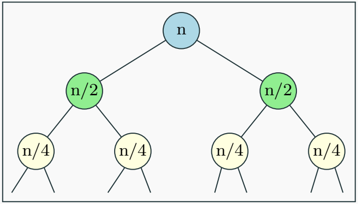
- Unroll the recurrence.
  $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the ith level total work is cn.

Recursion Trees : MergeSort: n is a power of 2



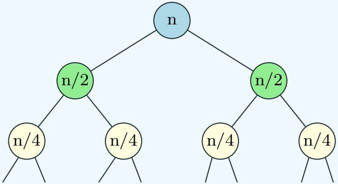
- Unroll the recurrence.
  $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the ith level total work is cn.
- Sum over all levels.

Recursion Trees : MergeSort: n is a power of 2

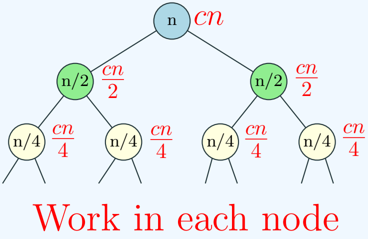


- Unroll the recurrence.
  $T(n) = 2T(n/2) + cn$
- Identify a pattern. At the ith level total work is cn.
- Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.

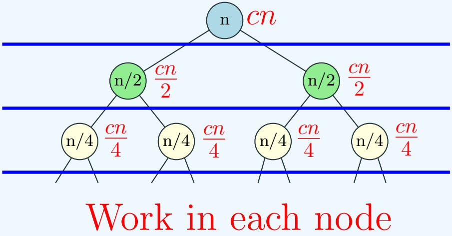
Recursion Trees



Recursion Trees



Recursion Trees



Recursion Trees

$$\log n \left\{ \begin{array}{ccc} cn & & = cn \\ \hline \frac{cn}{2} + \frac{cn}{2} & & = cn \\ \hline \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} & & = cn \\ \hline \vdots & & \vdots \\ \hline & & = cn \end{array} \right.$$

Recursion Tree:

$$cn = cn$$

$$\frac{cn}{2} + \frac{cn}{2} = cn$$

$$\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn$$

$$\vdots$$

$$= cn$$

$\log n$ levels; $\log_2 n$

$$= cn \log n = O(n \log_2 n)$$

$T(n) = O(n \log n)$!

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

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?

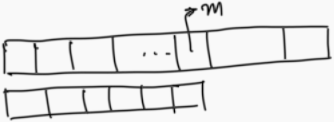
Binary Search

(R1X)

Binary Search in Sorted Arrays

Input Sorted array A of n numbers and number x

Goal Is x in A?



$m = x$ 

$m > x$

$m < x$

$$n \downarrow \frac{n}{2} \downarrow \frac{n}{4} \downarrow \vdots\ 1$$

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 $(A[\lfloor (a+b)/2 \rfloor + 1..b], x)$

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)
```

DIY

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

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