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

Lecturer: Lale Özkahya

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

Tower of Hanoi: Move the tower from one peg to another without ever putting a larger block on top of a smaller one.
Tower of Hanoi
Tower of Hanoi
Tower of Hanoi
Tower of Hanoi
Tower of Hanoi
Need to be careful about when we cannot invoke the induction ferry: Base Cases
Hanoi Algorithm

\[
\text{HANOI}(n, \ src, \ dst, \ tmp): \\
\quad \text{if } n > 0 \\
\quad \quad \text{HANOI}(n - 1, \ src, \ tmp, \ dst) \\
\quad \quad \text{move disk } n \text{ from } src \text{ to } dst \\
\quad \quad \text{HANOI}(n - 1, \ tmp, \ dst, \ src)
\]
Reduction = Delegation

Sometimes hard to delegate.
Reduction = Delegation

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

- Reg Exp $\rightarrow$ NFA (thompson)
- NFA $\rightarrow$ DFA (subset)
- DFA $\rightarrow$ min DFA (Moore)

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

How do you hunt a blue elephant?
- With the blue elephant gun

How do you hunt a red elephant?
- Hold its trunk until it turns blue, then hunt it with the blue elephant gun

How do you hunt a white elephant?
- Embarrass it till it becomes red. Use algorithm for hunting red elephants.
Reduction = Delegation

Sometimes hard to delegate.

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

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

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

Quicksort:
Sorting

Quicksort:

A G H I L

M

A G H I L

O R S T
Sorting

Quicksort:
Sorting

Quicksort:

\[
\text{QUICKSORT}(A[1..n]):
\]
\[
\quad \text{if } (n > 1)
\]
\[
\quad \text{Choose a pivot element } A[p]
\]
\[
\quad r \leftarrow \text{PARTITION}(A, p)
\]
\[
\quad \text{QUICKSORT}(A[1..r - 1])
\]
\[
\quad \text{QUICKSORT}(A[r + 1..n])
\]
Sorting

**Partition** (linear time):

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

• Divide the input array into two subarrays of roughly equal size

• Recursively merge sort each of the subarrays

• Merge the two newly sorted subarrays into a single sorted array
Sorting

Mergesort:
Sorting

Mergesort:

Algorithm:

A G L O R I T H M S

A G L O R

H I M S T
Sorting

Mergesort:

Need to merge the two subarrays.
Sorting

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

Merge:

One comparison, one recursive call
Sorting

Merge:

ALGORITHM

One comparison, one recursive call
Sorting

Merge:

A L G O R I T H M S

G L O R

H I M S T

A

Where can this recursion break?
Sorting

Merge:

ALGORITHM

HIMST
Sorting

Merge:

A L G O R I T H M S

I M S T

H
Sorting

Merge:

Where can this recursion break?
Sorting

Merge:

\[
\begin{align*}
\text{MERGE}(A[1..n], m): \\
i &\leftarrow 1; \ j \leftarrow m + 1 \\
&\text{for } k \leftarrow 1 \text{ to } n \\
&\quad \text{if } j > n \\
&\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
&\quad \text{else if } i > m \\
&\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
&\quad \text{else if } A[i] < A[j] \\
&\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
&\quad \text{else} \\
&\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
&\text{for } k \leftarrow 1 \text{ to } n \\
&\quad A[k] \leftarrow B[k]
\end{align*}
\]

Loop = recursion

- When writing actual code easier to unfold the recursion
- When proving correctness easier to use induction (=recursion)
Sorting

Mergesort:

\[
\begin{aligned}
\text{MERGESORT}(A[1..n]) : \\
\text{if } n > 1 \\
\quad m &\leftarrow \lfloor n/2 \rfloor \\
\quad \text{MERGESORT}(A[1..m]) \\
\quad \text{MERGESORT}(A[m+1..n]) \\
\quad \text{MERGE}(A[1..n], m)
\end{aligned}
\]

Base cases:
- When size of arrays to merge is 1
- When size of arrays is less than 10 and then brute force
- It doesn’t matter, no need to optimize
Proof of Correctness

- We prove \texttt{MERGE} is correct by induction on $n - k + 1$, which is the total size of the two sorted subarrays $A[i..m]$ and $A[j..n]$ that remain to be merged into $B[k..n]$ when the $k$th iteration of the main loop begins. There are five cases to consider. Yes, five.

  - If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.)
  - If $i \leq m$ and $j > n$, the subarray $A[j..n]$ is empty. Because both subarrays are sorted, the smallest element in the union of the two subarrays is $A[i]$. So the assignment $B[k] \leftarrow A[i]$ is correct. The inductive hypothesis implies that the remaining subarrays $A[i+1..m]$ and $A[j..n]$ are correctly merged into $B[k+1..n]$.
  - Similarly, if $i > m$ and $j \leq n$, the assignment $B[k] \leftarrow A[j]$ is correct, and The Recursion Fairy correctly merges—sorry, I mean the inductive hypothesis implies that the \texttt{MERGE} algorithm correctly merges—the remaining subarrays $A[i..m]$ and $A[j+1..n]$ into $B[k+1..n]$.
  - If $i \leq m$ and $j \leq n$ and $A[i] < A[j]$, then the smallest remaining element is $A[i]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly merges the rest of the subarrays.
  - Finally, if $i \leq m$ and $j \leq n$ and $A[i] \geq A[j]$, then the smallest remaining element is $A[j]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly does the rest.

Always make sanity check when you design algorithm!
Running time

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

• What is the running time $T(n)$ of quicksort?

• $O(n^2)$ time! (If I choose the smallest pivot)

  • $T(n)=O(n)+T(n-1)$

    $= O(n^2)$
Running time of Mergesort

• What is the running time $T(n)$ of mergesort?

• $O(n \log n)$ time!

• $T(n) = 2T(n/2) + O(n)$

• proof by induction if I know answer

• recursion tree!
Running time of Mergesort

Complete binary tree
every leaf is an array of size 1
Running time of Mergesort

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

= non-recursive work
Running time of Mergesort

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

- \( T(n) = 2T(n/2) + O(n) \)

- Total amount of work at level \( k \) = total amount of work at level \( k-1 \) (induction).
Running time of Mergesort

- $T(n) = 2T(n/2) + O(n)$
- Total amount of work = $n \times$ (height of the tree) = $n \log n$
Running time of Quicksort, revisited

- Quicksort runs in time $O(n \log n)$ in practice.

- Quicksort runs in time $O(n \log n)$ on average if the data is randomly permuted.

- Quicksort runs in expected time $O(n \log n)$ if we randomly permute the data first.
Part II

Reductions and Recursion
Reduction

Reducing problem \textbf{A} to problem \textbf{B}:

\begin{itemize}
\item Algorithm for \textbf{A} uses algorithm for \textbf{B} as a \textit{black box}
\end{itemize}
Reduction

Reducing problem A to problem B:

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

Q: How do you hunt a blue elephant?
A: With a blue elephant gun.

Q: How do you hunt a red elephant?
A: Hold his trunk shut until he turns blue, and then shoot him with the blue elephant gun.

Q: How do you shoot a white elephant?
A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.
UNIQUENESS: Distinct Elements Problem

Problem Given an array $A$ of $n$ integers, are there any duplicates in $A$?
UNIQUENESS: Distinct Elements Problem

Problem Given an array \( A \) of \( n \) integers, are there any duplicates in \( A \)?

Naive algorithm:

\[
\text{DistinctElements}(A[1..n])
\]
\[
\text{for } i = 1 \text{ to } n - 1 \text{ do}
\]
\[
\quad \text{for } j = i + 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \text{if } (A[i] = A[j])
\]
\[
\quad \quad \quad \text{return YES}
\]
\[
\quad \text{return NO}
\]
UNIQUENESS: Distinct Elements Problem

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

Naive algorithm:

```plaintext
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:

$O(n^2)$
Problem: Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```plaintext
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]*)

1. Sort *A*
2. 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
  else
    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.
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:

\[
\{A, C, F\}, \quad \{B, D\}, \quad \{E\}
\]
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 $G$

![Graph Image]
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!

![Diagram](image)
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.
**Question:** Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?
Reducions

**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!

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

n calls to the sub-routine for 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!

```plaintext
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
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
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
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

\[
\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}): \\
\text{if } (n > 0) \text{ then} \\
\quad \text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest}) \\
\quad \text{Move disk } n \text{ from src to dest} \\
\quad \text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})
\]

\[T(n):\] time to move \( n \) disks via recursive strategy
Recursive Algorithm

\begin{algorithm}
\textbf{Hanoi}(n, src, dest, tmp):
  \textbf{if} (n > 0) \textbf{then}
  \textbf{Hanoi}(n - 1, src, tmp, dest)
  \text{Move disk } n \text{ from src to dest}
  \textbf{Hanoi}(n - 1, tmp, dest, src)
\end{algorithm}

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

\[
T(n) = 2T(n - 1) + 1 \quad \text{for } n > 1 \quad \text{and } T(1) = 1
\]
\[
T(n) = 2T(n - 1) + 1 \\
= 2^2T(n - 2) + 2 + 1 \\
= \ldots \\
= 2^iT(n - i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
= \ldots \\
= 2^{n-1}T(1) + 2^{n-2} + \ldots + 1 \\
= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
= (2^n - 1)/(2 - 1) = 2^n - 1
\]
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 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.
Input  Given an array of \( n \) elements
Goal  Rearrange them in ascending order
Input: Array $A[1 \ldots n]$
1. **Input**: Array $A[1 \ldots n]$

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

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

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

3. Recursively MergeSort $A[1 \ldots m]$ and $A[m + 1 \ldots n]$
Merge Sort [von Neumann]

**MergeSort**

1. **Input:** Array \( A[1 \ldots n] \)

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

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

4. Merge the sorted arrays

Chandra & Manoj (UIUC) CS374 Fall 2015 35 / 55
Merge Sort [von Neumann]

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

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

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

4. Merge the 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.

\[
\begin{align*}
A & \quad G & \quad L & \quad O & \quad R & \quad H & \quad I & \quad M & \quad S & \quad T \\
\text{A} & \quad \text{G} & \quad \text{H} & \quad \text{I} & \quad \text{M} & \quad \text{O} & \quad \text{R} & \quad \text{S} & \quad \text{T}
\end{align*}
\]
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
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.

\[
\begin{align*}
A & G L O R \\
A & G H I \\
L & M O R S T \\
A & G H I
\end{align*}
\]
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

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

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

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:

\[ \text{induction!} \]

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 \( k \)

- \( B[1..k] \) 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.
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?

```plaintext
Merge (A[1..n], B[1..m])

if $n$ > 0 and $m$ > 0
   if $A[i] < B[j]$
   else
```

Chandra & Manoj (UIUC)
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 \textit{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

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

What do we want as a solution to the recurrence? Almost always only an asymptotically tight bound. That is we want \(f(n)\) such that \(T(n) = \Theta(f(n))\).

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

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 [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
Let $k$ be the rank of the chosen pivot. Then,

$$T(n) = T(k - 1) + T(n - k) + O(n)$$
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, $T(n) = O(n \log n)$. 

Chandra & Manoj (UIUC) CS374 44 Fall 2015 44 / 55
Let \( k \) be the rank of the chosen pivot. Then,
\[
T(n) = T(k - 1) + T(n - k) + O(n)
\]

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

Theoretically, median can be found in linear time.
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, \( T(n) = O(n \log n) \).

3. Theoretically, median can be found in linear time.

4. 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
Input Sorted array $A$ of $n$ numbers and number $x$

Goal Is $x$ in $A$?

**Binary Search in Sorted Arrays**

The algorithm for binary search in sorted arrays is as follows:

1. **Input Sorted array $A$ of $n$ numbers and number $x$**
2. **Goal Is $x$ in $A$?**

**Algorithm**

$$\text{BinarySearch}(A[a..b], x) :$$

- If $(b - a < 0)$ return NO
- $mid = A[\lfloor (a + b) / 2 \rfloor]$
- If $(x = mid)$ return YES
- If $(x < mid)$ return $\text{BinarySearch}(A[a..\lfloor (a + b) / 2 \rfloor - 1], x)$
- Else return $\text{BinarySearch}(A[\lfloor (a + b) / 2 \rfloor + 1..b], x)$

**Analysis:**

$$T(n) = T(\lfloor n / 2 \rfloor) + O(1)$$

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

**Observation:**

After $k$ steps, size of array left is $n / 2^k$
Input Sorted array \( A \) of \( n \) numbers and number \( x \)

Goal Is \( x \) in \( A \)?

```python
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 \( n/2^k \).
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 $v$?

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

1. Given instance $I$, compute upper bound $U(I)$ on best value
2. Compute lower bound $L(I)$ on best value
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
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
2. Compute lower bound $L(I)$ on best value
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 maximum edge length in $G$.
2. Minimum edge length is $L$.
3. $s$-$t$ shortest path length is at most $(n - 1)U$ and at least $L$.
5. $O(\log((n - 1)U - L))$ 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
Consider $T(n) = 2T(n/2) + n/\log n$. 

Construct recursion tree, and observe pattern. 

The $i$th level has $2^i$ nodes, and problem size at each node is $n/2^i$ and hence work at each node is 

$\frac{n}{2^i \log n}$.

Summing over all levels 

$T(n) = \log n - 1 \sum_{i=0}^{\log n} \left( \frac{n}{2^i \log (n/2^i)} \right) = n \log n \sum_{j=1}^\infty \frac{1}{j} = n \log n H_{\log n} = \Theta(n \log \log n)$.
Recurrence: Example I

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

$$T(n) = \sum_{i=0}^{\log n - 1} 2^i \left[ \frac{(n/2^i)}{\log(n/2^i)} \right]$$

$$= \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i}$$

$$= n \sum_{j=1}^{\log n} \frac{1}{j} = n H_{\log n} = \Theta(n \log \log n)$$
Consider \( T(n) = T(\sqrt{n}) + 1 \)

What is the depth of recursion?

\( \sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt[3]{n}}, \ldots, O(1) \).

Number of levels: \( n^{\frac{1}{2^d}} = 2 \) means \( d = \log \log n \).

Number of children at each level is 1, work at each node is 1.

Thus, \( T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n) \).
\[ T(n) = T(\sqrt{n}) + n \]

\[ n + \sqrt{n} + n^{1/4} + \ldots + 2 \]

\[ = O(\ ) \]
Consider $T(n) = T(\sqrt{n}) + 1$

What is the depth of recursion?

$\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt[3]{n}}, \ldots, O(1)$.

Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.

Number of children at each level is 1, work at each node is 1.

Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$. 

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

Using recursion trees: number of levels $L = \log \log n$.

Work at each level? Root is $n$, next level is $\sqrt{n} \times \sqrt{n} = n$. Can check that each level is $n$.

Thus, $T(n) = \Theta(n \log \log n)$.
Consider $T(n) = T(n/4) + T(3n/4) + n$. 

Using recursion tree, we observe the tree has leaves at different levels (a lopsided tree).

Total work in any level is at most $n$. Total work in any level without leaves is exactly $n$.

Highest leaf is at level $\log_4 n$ and lowest leaf is at level $\log_4/3 n$.

Thus, $n \log_4 n \leq T(n) \leq n \log_4/3 n$, which means $T(n) = \Theta(n \log n)$.
Consider $T(n) = T(n/4) + T(3n/4) + n$.

Using recursion tree, we observe the tree has leaves at different levels (a lop-sided tree).

Total work in any level is at most $n$. Total work in any level without leaves is exactly $n$.

Highest leaf is at level $\log_4 n$ and lowest leaf is at level $\log_{4/3} n$.

Thus, $n \log_4 n \leq T(n) \leq n \log_{4/3} n$, which means $T(n) = \Theta(n \log n)$.
Part I

Fast Multiplication
Multiplying Numbers

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

Grade School Multiplication

Compute “partial product” by multiplying each digit of $y$ with $x$ and adding the partial products.

\[
\begin{array}{c}
3141 \\
\times 2718 \\
\hline
25128 \\
3141 \\
21987 \\
6282 \\
\hline
8537238
\end{array}
\]
Time Analysis of Grade School Multiplication

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

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: \((a + bi)\) and \((c + di)\)

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]
A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: \((a + bi)\) and \((c + di)\)

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]

How many multiplications do we need?
A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: 

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]

How many multiplications do we need?

Only 3! If we do extra additions and subtractions. Compute \(ac, bd, (a + b)(c + d)\). Then

\[(ad + bc) = (a + b)(c + d) - ac - bd\]
Divide and Conquer

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

Split each number into two numbers with equal number of digits

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

\[ 1234 \times 5678 = (100 \times 12 + 34) \times (100 \times 56 + 78) \]
\[ = 10000 \times 12 \times 56 \]
\[ + 100 \times (12 \times 78 + 34 \times 56) \]
\[ + 34 \times 78 \]
Divide and Conquer

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

1. $x = x_{n-1}x_{n-2}\cdots x_0$ and $y = y_{n-1}y_{n-2}\cdots y_0$

2. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\cdots x_{n/2}$ and $x_R = x_{n/2-1}\cdots x_0$

3. $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\cdots y_{n/2}$ and $y_R = y_{n/2-1}\cdots y_0$

Therefore

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$

$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

$$T(n) = 4T\left(\frac{n}{2}\right) + 6n \quad T(4) = 1$$
xy = \left(10^{n/2}x_L + x_R\right)\left(10^{n/2}y_L + y_R\right)
= 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R

4 recursive multiplications of number of size \( n/2 \) each plus 4 additions and left shifts (adding enough 0’s to the right)
Time Analysis

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R \]

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

\[ T(n) = 4T(n/2) + O(n) \quad T(1) = O(1) \]
\[
xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)
\]
\[
= 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R
\]

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

\[
T(n) = 4T(n/2) + O(n) \quad T(1) = O(1)
\]

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

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \]

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

\[ T(n) = 4T(n/2) + O(n) \quad T(1) = O(1) \]

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

Can we invoke Gauss’s trick here?
Improving the Running Time

xy = \left(10^{n/2}x_L + x_R\right)\left(10^{n/2}y_L + y_R\right)
\quad = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R

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

T(n) = 3T\left(\frac{n}{2}\right) + n
Improving the Running Time

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^nx_Ly_L + 10^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R \]

Gauss trick: \( x_Ly_R + x_Ry_L = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R \)

Recursively compute only \( x_Ly_L, x_Ry_R, (x_L + x_R)(y_L + y_R) \).
Improving the Running Time

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \]

Gauss trick: \[ x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \]

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

Time Analysis

Running time is given by

\[ T(n) = 3T(n/2) + O(n) \quad \text{and} \quad T(1) = O(1) \]

which means
Improving the Running Time

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R \]

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

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

Time Analysis

Running time is given by

\[ T(n) = 3T(n/2) + O(n) \]
\[ T(1) = O(1) \]

which means \( T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \)
State of the Art

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

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

Conjecture
There is an $O(n \log n)$ time algorithm.
1. Basic divide and conquer: \( T(n) = 4T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^2) \).

2. Saving a multiplication: \( T(n) = 3T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^{1+\log 1.5}) \)
Analyzing the Recurrences

1. Basic divide and conquer: \( T(n) = 4T(n/2) + O(n), \)
   \( T(1) = 1. \) Claim: \( T(n) = \Theta(n^2). \)

2. Saving a multiplication: \( T(n) = 3T(n/2) + O(n), \) \( T(1) = 1. \)
   Claim: \( T(n) = \Theta(n^{1+\log 1.5}). \)

Use recursion tree method:

1. In both cases, depth of recursion \( L = \log n. \)

2. Work at depth \( i \) is \( 4^i n/2^i \) and \( 3^i n/2^i \) respectively: number of children at depth \( i \) times the work at each child.

3. Total work is therefore \( n \sum_{i=0}^{L} 2^i \) and \( n \sum_{i=0}^{L} (3/2)^i \) respectively.
Recursion tree analysis

\[
\begin{align*}
\text{root node: } n \\
\text{subtree 1: } \frac{n}{2} \\
\text{subtree 2: } \frac{n}{3} \\
\text{subtree 3: } \frac{n}{4} \\
\text{subtree 4: } \frac{n}{2} \\
\text{subtree 5: } \frac{3n}{2} \\
\text{subtree 6: } \frac{3n}{3} \\
\text{subtree 7: } \frac{3n}{4} \\
\ldots
\end{align*}
\]

\[\log_2 n\]

\[
\begin{align*}
\left(\frac{3}{2}\right)^d \cdot n \\
\left(\frac{3}{2}\right)^{\log_2 n} \cdot n = n
\end{align*}
\]
Part II

Selecting in Unsorted Lists
**Rank of element in an array**

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

**Definition**

For $1 \leq j \leq n$, element of rank $j$ is the $j$’th smallest element in $A$. 

<table>
<thead>
<tr>
<th>Unsorted array</th>
<th>16</th>
<th>14</th>
<th>34</th>
<th>20</th>
<th>12</th>
<th>5</th>
<th>3</th>
<th>19</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ranks</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Sort of array</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>34</td>
</tr>
</tbody>
</table>
Problem - Selection

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

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

**Median:**  $j = \lceil (n + 1)/2 \rceil$
Problem - Selection

Input  Unsorted array $A$ of $n$ integers and integer $j$
Goal   Find the $j$th smallest number in $A$ (rank $j$ number)

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

Simplifying assumption for sake of notation: elements of $A$ are distinct
Algorithm I

1. Sort the elements in A
2. Pick jth element in sorted order

Time taken = $O(n \log n)$
Algorithm I

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

Time taken = $O(n \log n)$

Do we need to sort? Is there an $O(n)$ time algorithm?
Algorithm II

If $j$ is small or $n - j$ is small then

1. Find $j$ smallest/largest elements in $A$ in $O(jn)$ time. (How?)
2. Time to find median is $O(n^2)$. 
Divide and Conquer Approach

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

Chandra & Manoj (UIUC) CS374 19 Fall 2015 19 / 32
Example

\[ J = \text{arg}\max \]

14, 8, 3, 12, 11, 16, 20, 34, 19

↑

6
Time Analysis

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

Suppose we always choose pivot to be $A[1]$. Say $A$ is sorted in increasing order and $j = n$.
Exercise: show that algorithm takes $\Omega(n^2)$ time.

Chandra & Manoj (UIUC) CS374 Fall 2015
Time Analysis

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

Suppose we always choose pivot to be \(A[1]\).
Time Analysis

1. Partitioning step: \( O(n) \) time to scan \( A \)

2. How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be \( A[1] \).

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

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is approximately in the middle of $A$.

Then $n/4 \leq |A_{\text{less}}| \leq 3n/4$ and $n/4 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

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

How do we find such a pivot?

Randomly? In fact works!

Analysis a little bit later.

Can we choose pivot deterministically?

Chandra & Manoj (UIUC) CS374 22 Fall 2015 22 / 32
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is \textit{approximately} in the middle of $A$

Then $n/4 \leq |A_{\text{less}}| \leq 3n/4$ \textit{and} $n/4 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

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

Implies $T(n) = O(n)$!
A Better Pivot

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

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

Implies $T(n) = O(n)$!

How do we find such a pivot?
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is *approximately* in the middle of $A$.

Then $n/4 \leq |A_{\text{less}}| \leq 3n/4$ and $n/4 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

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

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly?
A Better Pivot

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

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

Implies $T(n) = O(n)$!

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

Suppose pivot is the $\ell$th smallest element where $\frac{n}{4} \leq \ell \leq \frac{3n}{4}$. That is pivot is *approximately* in the middle of $A$

Then $\frac{n}{4} \leq |A_{\text{less}}| \leq \frac{3n}{4}$ and $\frac{n}{4} \leq |A_{\text{greater}}| \leq \frac{3n}{4}$. If we apply recursion,

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

Implies $T(n) = O(n)$!

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

Can we choose pivot deterministically?
Divide and Conquer Approach

A game of medians

Idea

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

\[
\begin{array}{cccccccccccc}
11 & 7 & 3 & 42 & 174 & 310 & 1 & 92 & 87 & 12 & 19 & 15 \\
\end{array}
\]

\[
\frac{n}{3}
\]

15
Choosing the pivot
A clash of medians

1. Partition array $A$ into $\lceil n/5 \rceil$ lists of 5 items each.
   $L_1 = \{A[1], A[2], \ldots, A[5]\}$, $L_2 = \{A[6], \ldots, A[10]\}$, \ldots,
   $L_i = \{A[5i + 1], \ldots, A[5i - 4]\}$, \ldots,
   $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4], \ldots, A[n]\}$.

2. For each $i$ find median $b_i$ of $L_i$ using brute-force in $O(1)$ time.
   Total $O(n)$ time

3. Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

4. Find median $b$ of $B$
Choosing the pivot

A clash of medians

1. Partition array $A$ into $\lceil n/5 \rceil$ lists of 5 items each.

$L_1 = \{A[1], A[2], \ldots, A[5]\}$, $L_2 = \{A[6], \ldots, A[10]\}$, \ldots, $L_i = \{A[5i + 1], \ldots, A[5i - 4]\}$, \ldots, $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4, \ldots, A[n]\}$.

2. For each $i$ find median $b_i$ of $L_i$ using brute-force in $O(1)$ time. Total $O(n)$ time.

3. Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

4. Find median $b$ of $B$

Lemma

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

A storm of medians

\[\text{select}(A, j):\]

Form lists \(L_1, L_2, \ldots, L_{\lceil n/5 \rceil}\) where \(L_i = \{A[5i - 4], \ldots, A[5i]\}\)

Find median \(b_i\) of each \(L_i\) using brute-force

Find median \(b\) of \(B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}\)

Partition \(A\) into \(A_{\text{less}}\) and \(A_{\text{greater}}\) using \(b\) as pivot

\[
\begin{align*}
\text{if} \ (|A_{\text{less}}|) &= j \ \text{return} \ b \\
\text{else if} \ (|A_{\text{less}}|) &> j \\
& \quad \text{return} \ \text{select}(A_{\text{less}}, j) \\
\text{else} \\
& \quad \text{return} \ \text{select}(A_{\text{greater}}, j - |A_{\text{less}}|)
\end{align*}
\]

How do we find median of \(B\)?
Algorithm for Selection
A storm of medians

\[ \text{select}(A, j): \]
Form lists \( L_1, L_2, \ldots, L_{\lceil n/5 \rceil} \) where \( L_i = \{A[5i - 4], \ldots, A[5i]\} \)
Find median \( b_i \) of each \( L_i \) using brute-force
Find median \( b \) of \( B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\} \)
Partition \( A \) into \( A_{\text{less}} \) and \( A_{\text{greater}} \) using \( b \) as pivot
\[ \text{if} \ (|A_{\text{less}}|) = j \ \text{return} \ b \]
\[ \text{else if} \ (|A_{\text{less}}|) > j \]
\[ \text{return} \ \text{select}(A_{\text{less}}, j) \]
\[ \text{else} \]
\[ \text{return} \ \text{select}(A_{\text{greater}}, j - |A_{\text{less}}|) \]

How do we find median of \( B \)?
Algorithm for Selection

A storm of medians

\texttt{select}(A, j):
Form lists \( L_1, L_2, \ldots, L_{\lceil n/5 \rceil} \) where \( L_i = \{A[5i-4], \ldots, A[5i]\} \)
Find median \( b_i \) of each \( L_i \) using brute-force
Find median \( b \) of \( B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\} \)
Partition \( A \) into \( A_{\text{less}} \) and \( A_{\text{greater}} \) using \( b \) as pivot
\textbf{if} \( \lvert A_{\text{less}} \rvert = j \) \textbf{return} \( b \)
\textbf{else if} \( \lvert A_{\text{less}} \rvert > j \)
\textbf{return} \texttt{select}(A_{\text{less}}, j)
\textbf{else}
\textbf{return} \texttt{select}(A_{\text{greater}}, j - \lvert A_{\text{less}} \rvert)

How do we find median of \( B \)? Recursively!
Algorithm for Selection
A storm of medians

\textbf{select}(A, j): \\
Form lists \(L_1, L_2, \ldots, L_{\lceil n/5 \rceil}\) where \(L_i = \{A[5i - 4], \ldots, A[5i]\}\) \\
Find median \(b_i\) of each \(L_i\) using brute-force \\
\(B = [b_1, b_2, \ldots, b_{\lceil n/5 \rceil}]\) \\
\(b = \text{select}(B, \lceil n/10 \rceil)\) \\
Partition \(A\) into \(A_{\text{less}}\) and \(A_{\text{greater}}\) using \(b\) as pivot \\
\textbf{if} \ (|A_{\text{less}}|) = j \ \textbf{return} \ b \\
\textbf{else if} \ (|A_{\text{less}}|) > j \\
\hspace{1cm} \textbf{return} \ \text{select}(A_{\text{less}}, j) \\
\textbf{else} \\
\hspace{1cm} \textbf{return} \ \text{select}(A_{\text{greater}}, j - |A_{\text{less}}|)
Running time of deterministic median selection

A dance with recurrences

\[ T(n) = T(\lceil n/5 \rceil) + \max \{ T(|A_{\text{less}}|), T(|A_{\text{greater}}|) \} + O(n) \]
Running time of deterministic median selection
A dance with recurrences

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

From Lemma,

\[ T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 + 6 \rfloor) + O(n) \]

and

\[ T(n) = O(1) \quad n < 10 \]
Running time of deterministic median selection

A dance with recurrences

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

From Lemma,

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

and

\[ T(n) = O(1) \quad n < 10 \]

Exercise: show that \( T(n) = O(n) \)
Median of Medians: Proof of Lemma

Proposition

There are at least \(\frac{3n}{10} - 6\) elements greater than the median of medians \(b\).

Figure: Shaded elements are all greater than \(b\).
Median of Medians: Proof of Lemma

Proposition

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

Proof.

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

$$3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil - 2 \right) \geq \frac{3n}{10} - 6$$
Median of Medians: Proof of Lemma

**Proposition**

There are at least \( \frac{3n}{10} - 6 \) elements greater than the median of medians \( b \).

**Corollary**

\[ |A_{\text{less}}| \leq \frac{7n}{10} + 6. \]

Via symmetric argument,

**Corollary**

\[ |A_{\text{greater}}| \leq \frac{7n}{10} + 6. \]
Questions to ponder

1. Why did we choose lists of size 5? Will lists of size 3 work?

2. Write a recurrence to analyze the algorithm’s running time if we choose a list of size $k$.

\[ T(n) = T\left(\frac{9n}{10}\right) + n \]
Median of Medians Algorithm

Due to:
“Time bounds for selection”.
Median of Medians Algorithm

Due to:
“Time bounds for selection”.

How many Turing Award winners in the author list?
Median of Medians Algorithm

Due to:
“Time bounds for selection”.

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

1. Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.

2. Recursive algorithms naturally lead to recurrences.

3. Sometimes one can look for certain types of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.