Reductions, Recursion and Divide and Conquer

Lecture 10
Tuesday, February 19, 2019
Part I

Brief Intro to Algorithm Design and Analysis
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

1. implements the primitive instructions
2. allows for an automated implementation of the entire algorithm by keeping track of state
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.
Models of Computation vs Computers

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

2. Computer: an actual *physical device* that implements a very specific model of computation.

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

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

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

Informal description:

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

Sorting: input is an array of $n$ numbers

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

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

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

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

1. For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two $n$-digit numbers, primality etc.

2. Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by $2^k$ where $k$ is word length.

3. Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.
Models used in class

In this course when we design algorithms:

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

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

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

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

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

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

We will broadly see three types of problems.

1. **Decision Problem**: Is the input a YES or NO input?
   Example: Given graph $G$, nodes $s, t$, is there a path from $s$ to $t$ in $G$?
   Example: Given a CFG grammar $G$ and string $w$, is $w \in L(G)$?

2. **Search Problem**: Find a solution if input is a YES input.
   Example: Given graph $G$, nodes $s, t$, find an $s$-$t$ path.

3. **Optimization Problem**: Find a best solution among all solutions for the input.
   Example: Given graph $G$, nodes $s, t$, find a shortest $s$-$t$ path.
Given a problem $P$ and an algorithm $A$ for $P$ we want to know:

- Does $A$ correctly solve problem $P$?
- What is the asymptotic worst-case running time of $A$?
- What is the asymptotic worst-case space used by $A$?

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

“for all $n$ and for all inputs $I$ of size $n$, $A$ 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
Part II

What is a good algorithm?
What is a good algorithm?

Running time...

“No, Thursday’s out. How about never—is never good for you?”
What is a good algorithm?

Running time...

<table>
<thead>
<tr>
<th>Input size</th>
<th>$n^2$ ops</th>
<th>$n^3$ ops</th>
<th>$n^4$ ops</th>
<th>$n!$ ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0 secs</td>
<td>0 secs</td>
<td>0 secs</td>
<td>0 secs</td>
</tr>
<tr>
<td>20</td>
<td>0 secs</td>
<td>0 secs</td>
<td>0 secs</td>
<td>16 mins</td>
</tr>
<tr>
<td>30</td>
<td>0 secs</td>
<td>0 secs</td>
<td>0 secs</td>
<td>3 · 10^9 years</td>
</tr>
<tr>
<td>100</td>
<td>0 secs</td>
<td>0 secs</td>
<td>0 secs</td>
<td>never</td>
</tr>
<tr>
<td>8000</td>
<td>0 secs</td>
<td>0 secs</td>
<td>1 secs</td>
<td>never</td>
</tr>
<tr>
<td>16000</td>
<td>0 secs</td>
<td>0 secs</td>
<td>26 secs</td>
<td>never</td>
</tr>
<tr>
<td>32000</td>
<td>0 secs</td>
<td>0 secs</td>
<td>6 mins</td>
<td>never</td>
</tr>
<tr>
<td>64000</td>
<td>0 secs</td>
<td>0 secs</td>
<td>111 mins</td>
<td>never</td>
</tr>
<tr>
<td>200,000</td>
<td>0 secs</td>
<td>3 secs</td>
<td>7 days</td>
<td>never</td>
</tr>
<tr>
<td>2,000,000</td>
<td>0 secs</td>
<td>53 mins</td>
<td>202.943 years</td>
<td>never</td>
</tr>
<tr>
<td>$10^8$</td>
<td>4 secs</td>
<td>12.6839 years</td>
<td>$10^9$ years</td>
<td>never</td>
</tr>
<tr>
<td>$10^9$</td>
<td>6 mins</td>
<td>12683.9 years</td>
<td>$10^{13}$ years</td>
<td>never</td>
</tr>
</tbody>
</table>
Reduction

Reducing problem $A$ to problem $B$:

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

Reducing problem $A$ to problem $B$:

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

**Q:** How do you hunt a blue elephant?

**A:** With a blue elephant gun.
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.
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.
Problem: Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```python
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$
        for $j = i + 1$ to $n$
                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:

```plaintext
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$ do
        for $j = i + 1$ to $n$ do
                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:

```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: $O(n^2)$*
UNIQUENESS: Distinct Elements Problem

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

Naive algorithm:

\[
\begin{align*}
\text{DistinctElements}(A[1..n]) \\
\quad \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{for } j = i + 1 \text{ to } n \text{ do} \\
\quad \quad \quad \text{if } (A[i] = A[j]) \\
\quad \quad \quad \quad \text{return YES} \\
\quad \text{return NO}
\end{align*}
\]

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

**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.
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.
Reduction to Sorting

**DistinctElements**($A[1..n]$)

Sort $A$

for $i = 1$ to $n - 1$ do

if ($A[i] = A[i + 1]$) then

return YES

return NO

Running time: $O(n)$ plus time to sort an array of $n$ numbers

**Important point:** algorithm uses sorting as a *black box*

Advantage of naive algorithm: works for objects that cannot be “sorted”. Can also consider hashing but outside scope of current course.
Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

1. **Positive direction:** Algorithm for $B$ implies an algorithm for $A$
2. **Negative direction:** Suppose there is no “efficient” algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)

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.
Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

1. **Positive direction:** Algorithm for $B$ implies an algorithm for $A$
2. **Negative direction:** Suppose there is no “efficient” algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)

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

1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
2. If there is no $o(n \log n)$ time algorithm for Distinct Elements problem then there is no $o(n \log n)$ time algorithm for Sorting.
Maximum Independent Set in a Graph

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

Some independent sets in graph above:
Maximum Independent Set Problem

Input  Graph $G = (V, E)$
Goal  Find maximum sized independent set in $G$
Maximum Weight Independent Set Problem

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

Goal  Find maximum weight independent set in $G$
Weighted Interval Scheduling

**Input** A set of jobs with start times, finish times and *weights* (or profits).

**Goal** Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!
**Weighted Interval Scheduling**

**Input** A set of jobs with start times, finish times and *weights* (or profits).

**Goal** Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

![Diagram showing overlapping and non-overlapping intervals with weights.](attachment:diagram.png)
**Question:** Can you reduce Weighted Interval Scheduling to Max Weight Independent Set Problem?
Weighted Circular Arc Scheduling

**Input**  A set of arcs on a circle, each arc has a *weight* (or profit).

**Goal**  Find a maximum weight subset of arcs that do not overlap.
Reductions

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

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

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

\( n \) calls to the sub-routine for interval scheduling
**Question:** Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

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

MaxWeightIndependentArcs(arcs $C$)

```plaintext
cur-max = 0
for each arc $C \in C$ do
    Remove $C$ and all arcs overlapping with $C$
    $w_C = \text{wt of opt. solution in resulting Interval problem}$
    $w_C = w_C + \text{wt}(C)$
    cur-max = max{cur-max, $w_C$}
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!

MaxWeightIndependentArcs(arcs $C$)

- $\text{cur-max} = 0$
- **for** each arc $C \in C$ **do**
  - Remove $C$ and all arcs overlapping with $C$
  - $w_C = \text{wt of opt. solution in resulting Interval problem}$
  - $w_C = w_C + \text{wt}(C)$
  - $\text{cur-max} = \max\{\text{cur-max, } w_C\}$
- **end for**

**return** $\text{cur-max}$

$n$ 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!

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

\( n \) calls to the sub-routine for interval scheduling
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

Reduction: reduce one problem to another

Recursion: a special case of reduction

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

Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.

1. For termination, problem instances of small size are solved by some other method as base cases
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

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

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

\[
T(n) = 2T(n - 1) + 1 \quad n > 1 \quad \text{and} \quad T(1) = 1
\]
Recursive Algorithm

\begin{algorithm}
\textbf{Hanoi}($n$, src, dest, tmp):
\hspace{1em} \textbf{if} ($n > 0$) \textbf{then}
\hspace{2em} \textbf{Hanoi}($n - 1$, src, tmp, dest)
\hspace{2em} Move disk $n$ from src to dest
\hspace{2em} \textbf{Hanoi}($n - 1$, tmp, dest, src)
\end{algorithm}

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

\[ T(n) = 2T(n - 1) + 1 \quad n > 1 \quad \text{and} \quad T(1) = 1 \]
Recursive Algorithm

Hanoi\((n, \text{src}, \text{dest}, \text{tmp})\):

\[
\text{if } (n > 0) \text{ then}
\]

\[
\text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest})
\]

Move disk \(n\) from src to dest

\[
\text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})
\]

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

\[T(n) = 2T(n - 1) + 1 \quad n > 1 \quad \text{and } T(1) = 1\]
\[ T(n) = 2T(n - 1) + 1 \]
\[ = 2^2 T(n - 2) + 2 + 1 \]
\[ = \ldots \]
\[ = 2^i T(n - i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \]
\[ = \ldots \]
\[ = 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \]
\[ = 2^{n-1} + 2^{n-2} + \ldots + 1 \]
\[ = (2^n - 1)/(2 - 1) = 2^n - 1 \]
Part IV

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

**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.
Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

**Approach**

1. Break problem instance into smaller instances - divide step
2. **Recursively** solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

**Question:** Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.
Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.
Input  Given an array of $n$ elements
Goal  Rearrange them in ascending order
1. Input: Array $A[1 \ldots n]$
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 $m = \lceil n/2 \rceil$
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 $A[m + 1 \ldots n]$
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
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
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.

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

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

\[
\begin{array}{cccccccc}
A & G & H & I & L & M & O & R & S & T \\
\end{array}
\]
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.

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

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

\[ \text{AGLORHIMST} \]
\[ \text{AGHILMORS} \]
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.

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

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

\[
\text{\underline{MergeSort}(A[1..n])}: \\
\quad \text{if } n > 1 \\
\quad \quad m \leftarrow \lfloor n/2 \rfloor \\
\quad \quad \text{MergeSort}(A[1..m]) \\
\quad \quad \text{MergeSort}(A[m+1..n]) \\
\quad \text{Merge}(A[1..n], m)
\]

\[
\text{Merge}(A[1..n], m):
\]
\[i \leftarrow 1; \quad j \leftarrow m + 1\]
\[\text{for } k \leftarrow 1 \text{ to } n\]
\[\quad \text{if } j > n\]
\[\quad \quad B[k] \leftarrow A[i]; \quad i \leftarrow i + 1\]
\[\quad \text{else if } i > m\]
\[\quad \quad B[k] \leftarrow A[j]; \quad j \leftarrow j + 1\]
\[\quad \text{else if } A[i] < A[j]\]
\[\quad \quad B[k] \leftarrow A[i]; \quad i \leftarrow i + 1\]
\[\quad \text{else}\]
\[\quad \quad B[k] \leftarrow A[j]; \quad j \leftarrow j + 1\]
\[\text{for } k \leftarrow 1 \text{ to } n\]
\[A[k] \leftarrow B[k]\]
Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $k$ the following hold:

- $B[1..k]$ contains the smallest $k$ elements of $A$ correctly sorted.
- $B[1..k]$ contains the elements of $A[1..(i−1)]$ and $A[(m+1)..(j−1)]$.
- No element of $A$ is modified.
Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $k$ the following hold:

- $B[1..k]$ contains the smallest $k$ elements of $A$ correctly sorted.
- $B[1..k]$ contains the elements of $A[1..(i - 1)]$ and $A[(m + 1)..(j - 1)]$.
- No element of $A$ is modified.
Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $k$ the following hold:

- $B[1..k]$ contains the smallest $k$ elements of $A$ correctly sorted.
- $B[1..k]$ contains the elements of $A[1..(i - 1)]$ and $A[(m + 1)..(j - 1)]$.
- No element of $A$ is modified.
Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $k$ the following hold:

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

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

\[ T(n) = T([n/2]) + T([n/2]) + cn \]

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
**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
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 \textit{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

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

MergeSort: \( n \) is a power of 2

1. Unroll the recurrence. \( T(n) = 2T(n/2) + cn \)

2. Identify a pattern. At the \( i \)th level total work is \( cn \).
Recursion Trees

MergeSort: \( n \) is a power of 2

1. Unroll the recurrence. \( T(n) = 2T(n/2) + cn \)

2. Identify a pattern. At the \( i \)th level total work is \( cn \).
Recursion Trees

MergeSort: $n$ is a power of 2

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

2. Identify a pattern. At the $i$th level total work is $cn$.

3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.
Recursion Trees

MergeSort: \( n \) is a power of 2

1. Unroll the recurrence. \( T(n) = 2T(n/2) + cn \)

2. Identify a pattern. At the \( i \)th level total work is \( cn \).

3. Sum over all levels. The number of levels is \( \log n \). So total is \( cn \log n = O(n \log n) \).
Recursion Trees

An illustrated example...
Recursion Trees

An illustrated example...

Work in each node
Recursion Trees

An illustrated example...

![Recursion Tree Diagram]

Work in each node
Recursion Trees
An illustrated example...

\[
\log n \left\{ \begin{array}{c}
\begin{array}{c}
\frac{cn}{2} + \frac{cn}{2} = cn \\
\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn \\
\vdots \\
\end{array}
\end{array} \right. 
\]

\[= \quad cn \]

Chan, Har-Peled, Hassanieh (UIUC) CS374 Spring 2019
Recursion Trees

An illustrated example...

\[
\begin{align*}
\log n \quad \left\{ \begin{array}{c}
\frac{cn}{2} + \frac{cn}{2} \\
\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} \\
\vdots
\end{array} \right. \\
\end{align*}
\]

\[= cn \log n = O(n \log n)\]
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 [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.
Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.
Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.
Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.
Quick Sort: Example

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
Let $k$ be the rank of the chosen pivot. Then,

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

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

2. If \( k = \lceil n/2 \rceil \) then
\[
T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n).
\]
Then, \( T(n) = O(n \log n) \).

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

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 V

Binary Search
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: $T(n) = T(⌊n/2⌋) + O(1)$. $T(n) = O(\log n)$.

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 $A$?

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

- 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 $n/2^k$
Binary Search in Sorted Arrays

Input  Sorted array $A$ of $n$ numbers and number $x$
Goal   Is $x$ in $A$?

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

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
Problem: shortest paths in a graph.

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 $B$?

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?
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 VI

Solving Recurrences
Two general methods:

1. Recursion tree method: need to do sums
   - elementary methods, geometric series
   - integration

2. Guess and Verify
   - guessing involves intuition, experience and trial & error
   - verification is via induction
Consider $T(n) = 2T(n/2) + n/\log n$ for $n > 2$, $T(2) = 1$.

Construct recursion tree, and observe pattern. $i$th level has $2^i$ nodes, and problem size at each node is $n/2^i$ and hence work at each node is $n/2^i / \log n/2^i$.

Summing over all levels

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

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

$$= n \sum_{j=1}^{\log n} \frac{1}{j} = nH_{\log n} = \Theta(n \log \log n)$$
Consider $T(n) = 2T(n/2) + n/\log n$ for $n > 2$, $T(2) = 1$.

2 Construct recursion tree, and observe pattern. $i$th level has $2^i$ nodes, and problem size at each node is $n/2^i$ and hence work at each node is $n/2^i / \log n$.

3 Summing over all levels

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

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

\[
= n \sum_{j=1}^{\log n} \frac{1}{j} = nH_{\log n} = \Theta(n \log \log n)
\]
Consider $T(n) = T(\sqrt{n}) + 1$ for $n > 2$, $T(2) = 1$.

What is the depth of recursion?

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

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, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$. 
Recurrence: Example II

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

2. What is the depth of recursion?
   \[ \sqrt{n}, \sqrt[4]{n}, \sqrt[8]{n}, \ldots, O(1). \]

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

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

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

Chan, Har-Peled, Hassanieh (UIUC)
Consider $T(n) = \sqrt{n} T(\sqrt{n}) + n$ for $n > 2$, $T(2) = 1$.

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

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

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 $T(n) = \Theta(n \log n)$.
Recurrence: Example IV

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

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

3. Total work in any level is at most \( n \). Total work in any level without leaves is exactly \( n \).

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

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