WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.
Case study: Twenty questions

You wish to play the game of twenty questions, in which you are allowed to ask twenty yes/no questions to figure out which object your opponent has selected.

Simplification: Ask 20 yes/no questions to determine an integer in the range from 1 to 100.
Breaking a problem into smaller problems

To solve 20 questions on 1 to 100:

- Ask “Is the number greater than 50?”.
- If the answer is yes, solve 19 questions on 51 to 100.
- If the answer is no, solve 19 questions on 1 to 50.

Key idea: The smaller problems are of the same kind.

How are the smaller problems solved?

To solve \( k \) questions on \( a \) to \( b \):

- Compute \( c = \lfloor \frac{a+b}{2} \rfloor \).
- Ask “Is the number greater than \( c \)?”.
- If the answer is yes, solve \( k - 1 \) questions on \( c + 1 \) to \( b \).
- If the answer is no, solve \( k - 1 \) questions on \( a \) to \( c \).

Key idea: Smaller problems can be solved using recursion.
Paradigm: Divide-and-conquer

View as three steps:

* **Divide** into one or more smaller problems of the same kind.
* **Conquer** the smaller problem(s), possibly recursively.
* **Combine** the solution(s) to the smaller problem(s).

For 20 questions:

* **Divide** by asking about the middle element.
* **Conquer** the smaller problem using recursion.
* **Combine** by returning the solution to the smaller problem.
Solving 20 questions a different way

To solve 20 questions on 1 to 100:

- Ask “Is the number greater than 2?”.
- If the answer is yes, solve 19 questions on 3 to 100.
- If the answer is no, solve 19 questions on 1 to 2.

To solve \( k \) questions on \( a \) to \( b \):

- Ask “Is the number greater than \( a+1 \)?”.
- If the answer is yes, solve \( k - 1 \) questions on \( a+2 \) to \( b \).
- If the answer is no, solve \( k - 1 \) questions on \( a \) to \( a+1 \).
Review of recursive definitions

In a recursive definition, there is at least one base case and at least one recursive case.

In the recursive case, the entity is defined in terms of one or more entities of the same type, closer to the base case.

Example 1: An increasing sequence of numbers is either

- no numbers or
- a number followed by $L$, where $L$ an increasing sequence of numbers and the number is less than the first number in $L$, if any.

Example 2: An increasing sequence of numbers is either

- no numbers or
- an increasing sequence of numbers $L_1$ followed by a number followed by an increasing sequence of numbers $L_2$, where the number is greater than the last number in $L_1$, if any, and less than the first number in $L_2$, if any.
Review of recursive algorithms

Key ideas:

- Base case(s): Solve at least one case non-recursively.
- General case(s): Use one or more function calls on inputs that are closer to the base case.

The two different recursive definitions lead to the two methods discussed for solving twenty questions.
Dealing with recursive functions

Dangers:

- A function call to an input the same size (or the input itself) leads to an infinite loop.
- The absence of a base case means the running never ends.

Running time analysis (to follow in more detail) depends on:

- Size of smaller problem(s)
- Depth of recursion
Generalizing $k$ questions

Search for an item $I$ in an ordered list $L$ using **binary search** (CS 116):

- **Divide** by comparing $I$ to the middle element in $L$. If it matches, you are done. Otherwise, divide $L$ into $L_1$ and $L_2$.
- **Conquer** by recursively running the search on either $L_1$ or $L_2$.
- **Combine** by returning the solution to the smaller problem.
def bin_search(num_list, item):
    if len(num_list) <= 1:
        if len(num_list) == 1 and num_list[0] == item:
            return True
        else:
            return False
    else:
        mid = len(num_list) // 2
        if num_list[mid] == item:
            return True
        elif num_list[mid] > item:
            return bin_search(num_list[:mid], item)
        else:
            return bin_search(num_list[mid+1:], item)
Running time analysis for binary search

Base case: When the length of the list is at most 1, the cost is in $Θ(1)$.

General case: When the length of the list is $n > 1$, the cost is at most the cost of the slice function plus the cost of the function on a list of length $\lceil n/2 \rceil$.

```python
7      mid = len(num_list) // 2
8      if num_list[mid] == item:
9          return True
10     elif num_list[mid] > item:
11         return bin_search(num_list[:mid], item)
12     else:
13         return bin_search(num_list[mid+1:], item)
```
Recurrence relations

The running time of an algorithm on an input of size $n$ can be expressed recursively as a **recurrence relation**, in terms of one or more inputs of size smaller than $n$ and one or more base cases.

The recurrence relation for binary search is:

- $T(1) \in \Theta(1)$
- For $n > 1$, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(n)$.

The cost of $\Theta(n)$ is due to the cost of creating a new, smaller list.

To determine $T(n)$ as a **closed form** (written without reference to another use of $T$), we need to learn how to solve recurrence relations.

We will do so shortly, after considering another use of divide-and-conquer.
Using divide-and-conquer for sorting

Suppose you wish to sort a sequence of numbers by breaking it into smaller sequences of numbers and sorting those.

- **Divide** the sequence into two sequences.
- **Conquer** by sorting the two sequences.
- **Combine** the sorted sequences to determine the solution to the original problem.

Question to answer:

- How do we divide the sequence into two sequences?
- How do we combine the solutions?
Mergesort

We assume that we have a function `merge` that consumes two sorted lists and produces a sorted list containing all the items in the two input lists.

Note: Such a function is developed in Python session 2.

```python
1 def merge_sort(items):
2     if len(items) <= 1:
3         return items
4     else:
5         mid = len(items) // 2
6         first = items[:mid]
7         last = items[mid:]
8         first_sorted = merge_sort(first)
9         last_sorted = merge_sort(last)
10        return merge(first_sorted, last_sorted)
```
Analyzing mergesort

Divide $\Theta(n)$ to create new lists

Conquer The sum of the costs of running mergesort on an input of size $\lceil n/2 \rceil$ and an input of size $\lfloor n/2 \rfloor$.

Combine The cost of merge.

\[
T(1) \in \Theta(1) \\
T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)
\]

```python
def merge_sort(items):
    if len(items) <= 1:
        return items
    else:
        mid = len(items) // 2
        first = items[:mid]
        last = items[mid:]
        first_sorted = merge_sort(first)
        last_sorted = merge_sort(last)
        return merge(first_sorted, last_sorted)
```
Analyzing recursive procedures

1. Express the running time on an input of size $n$ in terms of the running time on one or more inputs of size smaller than $n$.

2. Determine the running time on the base case(s).

3. Solve the recurrence to obtain a closed form expression in order notation.

Note: We will typically use $O()$ notation for our analyses.
Costs of creating new inputs

The “divide” steps of our implementations of binary search and mergesort had a cost of $\Theta(n)$ due to the cost of creating new lists.

More efficient in-place implementations are possible, using the original lists instead of creating new ones.

The recurrence relation for mergesort does not change, due to the cost of the merging.

The recurrence relation for binary search is instead:

\[
T(1) \in \Theta(1)
\]

For $n > 1$, \( T(n) \leq T(\lceil n/2 \rceil) + \Theta(1) \).
Forming recurrences for twenty questions

Splitting into two smaller problems:

- Determine the middle element.
- Ask if the number is greater than the middle element.
- Solve the problem on an input of about half the size.

\[
T(n) \leq T(\lceil n/2 \rceil) + \Theta(1).
\]
\[
T(1) \in \Theta(1)
\]

Splitting into one smaller problem:

- Ask if the number is greater than the second element.
- Solve the problem on an input one smaller.

\[
T(n) \leq T(n-1) + \Theta(1).
\]
\[
T(1) \in \Theta(1)
\]
Methods for solving recurrences

• Iteration method
• Substitution method
• Master theorem method

Aside

Other methods exist, such as using generating functions and characteristic equations.
Iteration method

Step 1 Iteratively expand the recurrence.
Step 2 “Unwind” until the base case is reached.
Step 3 Determine the total.

Example: \( T(n) = T(n - 1) + 5, \ T(1) = 4 \)

Step 1: Iteratively expand the recurrence.

\[
T(n) = T(n - 1) + 5 \\
= T(n - 2) + 5 + 5 \\
= T(n - 3) + 5 + 5 + 5 \\
= T(n - k) + 5k
\]
Iteration method example, continued

Step 2: “Unwind” until the base case is reached.

To be able to convert \( T(n) = T(n - k) + 5k \) into an expression in which there is no \( T \) on the right side:

- Find a value of \( k \) such that \( T(n - k) = T(1) \).
- Solve the equation formed.

We know \( n - k = 1 \) when \( k = n - 1 \) and that \( T(1) = 4 \).

Step 3: Determine the total.

\[
T(n) = T(n - k) + 5k \\
= T(1) + 5(n - 1) \\
= 4 + 5n - 5 \\
= 5n - 1
\]
Second iteration method example

Example: $T(1) = 3; T(n) = T([n/2]) + 1$

Step 1: Iteratively expand the recurrence.

\[
T(n) = T([n/2]) + 1 \\
= T([n/4]) + 1 + 1 \\
= T([n/8]) + 1 + 1 + 1 \\
= T([n/2^k]) + k
\]
Second iteration method example, continued

Step 2: “Unwind” until the base case is reached.

We find a value of \( k \) such that \( \lceil n/2^k \rceil = 1 \).

We know that \( \lceil n/2^k \rceil = 1 \) when \( k = \lceil \log_2 n \rceil \) and that \( T(1) = 3 \).

Step 3: Determine the total.

\[
T(n) \leq T(\lceil n/2^k \rceil) + k \\
\leq T(1) + \lceil \log_2 n \rceil \\
\leq 3 + \lceil \log_2 n \rceil
\]
Substitution method example 1

Example: \( T(n) = 2T(\lceil n/2 \rceil) \), \( T(2) = T(1) = 1 \)

Step 1: Guess an upper bound for \( T(n) \).
\( T(n) \leq n^2 \)

Step 2: Use the guess for values on the right hand side.
Using our guess, for the value \( \lceil n/2 \rceil \), we assume that \( T(\lceil n/2 \rceil) \leq \lceil n/2 \rceil^2 \).
Using \( \lceil n/2 \rceil \leq n/2 \), we obtain the equation \( T(\lceil n/2 \rceil) \leq (n/2)^2 \).

Step 3: Simplify the right hand side to prove the bound.
Since \( T(n) = 2T(\lceil n/2 \rceil) \), we obtain
\( T(n) \leq 2(n/2)^2 = 2(n^2/4) = n^2/2 \leq n^2 \).

Step 4: Check that the bound holds for the base cases.
\( T(2) = 1 \leq 2^2 \)
\( T(1) = 1 \leq 1^2 \)
Master method

For \( T(n) = aT(n/b) + f(n) \), ignoring floors and ceilings, compare \( f(n) \) and 
\[ x = n^{\log_b a}. \]
If \( f(n) \) is “smaller than” \( x \), then \( T(n) \) is in \( \Theta(x) \).
If \( f(n) \) is “bigger than” \( x \), then \( T(n) \) is in \( \Theta(f(n)) \).
If \( f(n) \) and \( x \) are “the same size” then \( T(n) \) is in \( \Theta(x \log n) \).

Aside

The Master Theorem does not cover all cases:
Suppose \( T(n) = aT(n/b) + f(n) \) for constants \( a \geq 1 \) and \( b > 1 \). Then

- If \( f(n) \) is in \( O(n^{\log_b a - \varepsilon}) \) for some constant \( \varepsilon > 0 \), then \( T(n) \) is in \( \Theta(n^{\log_b a}) \).
- If \( f(n) \) is in \( \Omega(n^{\log_b a + \varepsilon}) \) for some constant \( \varepsilon > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) \) is in \( \Theta(f(n)) \).
- If \( f(n) \) is in \( \Theta(n^{\log_b a}) \), then \( T(n) \) is in \( \Theta(n^{\log_b a \log n}) \).
Properties of divide-and-conquer

- Original problem is easily decomposable into subproblems
- Subproblems should be roughly the same size
- Combining solutions shouldn’t be too costly

Divide-and-conquer works well on inputs that can easily be split.

Examples include:

- A set of numbers
- Entries in a grid
- The digits in a number
- A set of points in space
Maximum subtotal

**Maximum subtotal**

**Input:** A sequence of $n$ numbers $a_1$ through $a_n$

**Output:** $\sum_{k=i}^{j} a_k$ that is the largest possible

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>-10</td>
<td>4</td>
<td>-2</td>
<td>16</td>
<td>-5</td>
<td></td>
</tr>
</tbody>
</table>

$i$ = 4

$j$ = 6

Maximum subtotal: 27 / 42
Solving maximum subtotal

**Divide**: divide list into two pieces
**Conquer**: find maximum of left half and maximum of right half
**Combine**: maximum of
  - maximum of left half
  - maximum of right half
  - sum of the maximum suffix of the left half and the maximum prefix of the right half

```
2 -10 4 -2 16 -5
7 7 1 2 3 4 5 6 7
9 16
```

Maximum subtotal: 28 / 42
Maximum subtotal implemented

Full code including helper functions available on course website.

```python
1   def max_subtotal(num_list):
2       if len(num_list) == 1:
3           return num_list[0]
4       else:
5           mid = len(num_list) // 2
6           first = num_list[:mid]
7           last = num_list[mid:]
8           suffix = max_suffix(first)
9           prefix = max_prefix(last)
10          first_max = max_subtotal(first)
11          last_max = max_subtotal(last)
12          return max(suffix + prefix, first_max, last_max)
```

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

$$T(1)$$ is constant time
Selection

**Selection**

**Input:** A set of \( n \) numbers and a value \( j \), \( 1 \leq j \leq n \).

**Output:** A number in the set that is the \( j \)th largest.

Special cases:

- Maximum for \( j = 1 \)
- Minimum for \( j = n \)
- Median for \( n \) odd and \( j = \lceil n/2 \rceil \)
Using divide-and-conquer to solve selection

**Divide**  Split into two sets, bigger and smaller items
**Conquer**  Solve selection on one of the two sets
**Combine**  Return solution to smaller instance

We can split the sets using a **pivot** (one of the items in the set).
If the pivot is the median, we obtain two almost-equal-sized sets.
If the pivot is the minimum or the maximum, the set sizes are imbalanced.
Goal: Find a good pivot quickly.
Using divide-and-conquer to find a good pivot

Algorithm:
- Split items into chunks of five elements each.
- Find the median of each chunk.
- Recursively find the median of the medians.

This leads to a linear-time algorithm.

Beyond the scope of this course:
- Analysis of finding the pivot
- Analysis of using the pivot
Testing paper-cup phones

A group of friends have a new design for paper-cup phones, but a limited budget for string. To economize, you wish to figure out whose bedroom windows are closest.

Simplification: Given a collection of points in the plane, determine which pair is closest.

For points $p_a = (x_a, y_a)$ and $p_b = (x_b, y_b)$, their distance is measured as

$$\text{dist}(p_a, p_b) = \sqrt{(x_a - x_b)^2 - (y_a - y_b)^2}.$$
Formalizing the problem

Exercise: Write a constructive optimization problem.

**CLOSEST PAIR OF POINTS**

**Input:** A collection of $n$ points, where point $p_i = (x_i, y_i)$

**Output:** Points $p_a$ and $p_b$ in the collection, $a \neq b$, such that $\text{dist}(p_a, p_b)$ is minimized.
Using divide-and-conquer for closest pair of points

Divide  Find dividing line to divide into two groups of equal size (by x value)
Conquer  Find closest pair in each group
Combine  ??

But what if the closest pair has one in each group?
Matrices

A **matrix** is a way of storing data in a rectangular form. Common uses of matrices include storing information about images, graphs, multiple equations, and probabilities.

An $r \times c$ matrix has $r$ rows and $c$ columns, such as in the examples below:

- A $3 \times 2$ matrix:
  \[
  \begin{pmatrix}
  1 & 0 \\
  0 & 1 \\
  1 & 1
  \end{pmatrix}
  \]

- A $1 \times 2$ matrix:
  \[
  (a \ b)
  \]

- A $3 \times 1$ matrix:
  \[
  \begin{pmatrix}
  c \\
  d \\
  e
  \end{pmatrix}
  \]

Many problems are solved by various calculations using matrices, such as multiplying matrices.
Multiplying matrices

You can multiply two matrices if the number of columns in the first matrix is equal to the number of rows in the second matrix.

\[
\begin{pmatrix}
1 & 2 \\
3 & 4 \\
5 & 6
\end{pmatrix}
\begin{pmatrix}
7 & 8 & 9 & 10 \\
11 & 12 & 13 & 14
\end{pmatrix}
= 
\begin{pmatrix}
29 & 32 & 35 & 38 \\
65 & 72 & 79 & 86 \\
101 & 112 & 123 & 134
\end{pmatrix}
\]

The entry in row \(i\) and column \(j\) of the result is found by taking the sum of the pair-wise products of the values in row \(i\) of the first matrix and the values in column \(j\) of the second matrix.

\[
\begin{pmatrix}
1 \cdot 7 + 2 \cdot 11 & 1 \cdot 8 + 2 \cdot 12 & 1 \cdot 9 + 2 \cdot 13 & 1 \cdot 10 + 2 \cdot 14 \\
3 \cdot 7 + 4 \cdot 11 & 3 \cdot 8 + 4 \cdot 12 & 3 \cdot 9 + 4 \cdot 13 & 3 \cdot 10 + 4 \cdot 14 \\
5 \cdot 7 + 6 \cdot 11 & 5 \cdot 8 + 6 \cdot 12 & 5 \cdot 9 + 6 \cdot 13 & 5 \cdot 10 + 6 \cdot 14
\end{pmatrix}
\]

To multiply an \(a \times b\) and a \(b \times c\) matrix, the result will be an \(a \times c\) matrix. Each of the entries require \(b\) pairs of values to be multiplied. The total cost will be \(a \times b \times c\).
Using divide-and-conquer to multiply matrices

Matrix multiplication

Input: Two \( n \times n \) matrices \( A \) and \( B \)

Output: The product of \( A \) and \( B \)

Idea: View a matrix as a matrix of smaller matrices.

\[
\begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

To multiply \( A \) and \( B \):

\[
\begin{pmatrix}
A_0 & A_1 \\
A_2 & A_3
\end{pmatrix}
\begin{pmatrix}
B_0 & B_1 \\
B_2 & B_3
\end{pmatrix} =
\begin{pmatrix}
A_0B_0 + A_1B_2 & A_0B_1 + A_1B_3 \\
A_0B_0 + A_2B_2 & A_0B_1 + A_2B_3
\end{pmatrix}
\]
Using divide-and-conquer to solve TSP

**Divide**  Split into two or more smaller graphs
**Conquer**  Solve the problem on the smaller graphs
**Combine**  Build a tour from smaller tours

Issues to consider:
- Splitting a graph into smaller graphs
- Building a tour from smaller tours
Splitting a graph into smaller graphs

Trees:
- Removing a vertex breaks a tree into smaller trees.
- It is possible to obtain pieces of size at most $n/2$.

Planar graphs:
- Such a graph can be drawn on the plane without crossing edges.
- There is a small set of vertices whose removal results in pieces of size at most $2n/3$.

Cheapest directed cycle on weighted directed planar graphs:
- Break the graph into smaller graphs by vertex set $S$.
- Find the cheapest directed cycles in each of the smaller graphs.
- Use Dijkstra’s algorithm to find the cheapest cycle in $S$ starting from each vertex in $S$.
- Return the cheapest of the cycles found.
Comparing paradigms

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Exhaustive search</th>
<th>Greedy</th>
<th>Divide-and-conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subproblems</td>
<td>All</td>
<td>Some</td>
<td>Some</td>
</tr>
<tr>
<td>Applicability</td>
<td>Wide</td>
<td>Narrow</td>
<td>Medium</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Properties of divide-and-conquer algorithms:

- Works for problems that can be decomposed
- Balance size of subproblems to reduce running time
- Running time is determined using a recurrence
Module summary

Topics covered:

- Case study: Twenty questions
- Paradigm: Divide-and-conquer
- Review of recursive definitions and algorithms
- Binary search
- Recurrence relations
- Sorting
- Iteration method
- Substitution method
- Master theorem method
- Maximum subtotal
- Selection
- Finding the closest pair of points
- Matrix multiplication
- TSP
- Comparing paradigms