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.
## Comparing problems

### Number 1 odd

**Input:** A sequence of numbers  
**Output:** Yes or no, answering “Is the number 1 odd?”

### First three odd

**Input:** A sequence of numbers  
**Output:** Yes or no, answering “Are the first three numbers in the sequence all odd?”

### All odd

**Input:** A sequence of numbers  
**Output:** Yes or no, answering “Is each number in the sequence odd?”
## Comparing problems, fixed

<table>
<thead>
<tr>
<th><strong>Number 1 odd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A sequence of numbers</td>
</tr>
<tr>
<td><strong>Output:</strong> Yes or no, answering “Is the number 1 odd?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>First three odd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A sequence of numbers of length at least three</td>
</tr>
<tr>
<td><strong>Output:</strong> Yes or no, answering “Are the first three numbers in the sequence all odd?”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>All odd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A nonempty sequence of numbers</td>
</tr>
<tr>
<td><strong>Output:</strong> Yes or no, answering “Is each number in the sequence odd?”</td>
</tr>
</tbody>
</table>
Comparing more problems

**Any Odd**
- **Input:** A nonempty sequence of numbers
- **Output:** An odd number in the sequence, or “None” if none exists

**Maximum**
- **Input:** A nonempty sequence of numbers
- **Output:** A number in the sequence with maximum value
Comparing problems

Goals:

• Classify problems into rough categories.
• Choose a category for a problem based on the best possible algorithm that solves the problem.
• Express running time as a function of the input size.
• Find a representative number for all inputs of a particular size.
• Express algorithms in a hardware- and software-independent manner.
Questions to answer

1. How can we express algorithms in a hardware- and software-independent manner?
2. How can we form rough categories?
3. How can a single number represent all inputs of a given size?
4. How is input size measured?
5. How can we use the categories to simplify analysis?
6. How do we know if an algorithm is the best possible?
Q1: How can we express algorithms in a hardware- and software-independent manner?

Factors that can have an impact on the running time:

- the algorithm that is used,
- the choice of programming language,
- the way the algorithm is coded,
- the efficiency of the compiler,
- the choice of machine,
- the speed of the operating system,
- the speed of the hardware.
Expressing and analyzing algorithms

- Express an algorithm in a way that does not depend on the details (e.g., Python’s powerful list methods not found in other languages).
- Use the algorithm description as a way to get a rough estimate of the running time that can be compared to other algorithm ideas.

Key idea

No need to code the algorithm, or even choose the programming language.
Models of computation

A model of computation defines a set of operations and the resources they require (e.g. time, space).

When the model is weaker than the actual program
If time $t$ suffices for the model to accomplish a task, then the actual program can accomplish the task in time $t$ or less.

The time taken by the model is an upper bound on the time taken by the program.

Key idea: Use a weak model to find an upper bound.

When the model is stronger than the actual program
If the model needs at least time $t$ to accomplish a task, then the actual program needs time $t$ or more.

The time needed by the model is a lower bound on the time needed by the program.

Key idea: Use a strong model to find a lower bound.
Example models of computation

General purpose:

- **Turing machine**: Count steps as reading and writing to cells on an infinite tape, accessed by a tape head that can move between adjacent cells
- **Random Access Machine (RAM)**: Count steps allowing access to nonadjacent cells

Problem specific:

- The **decision tree** is used for comparison-based algorithms.
- For handling of very large numbers, arithmetic operations have a cost proportional to the space needed to store the value (logarithmic number of bits for a binary number); this is a **log cost model**.
Pseudocode

Instead of a formal model of computation, we use pseudocode:

- describes the algorithm at a higher level than in a RAM
- assumes RAM operations have constant cost, such as assignment of a value to a variable, use of a variable, moving to another point in the program, simple arithmetic and Boolean operations, and so on
- roughly approximates the cost of an actual program

Advantages of pseudocode:

- We can prove the correctness and running time of our algorithms (close enough to model of computation).
- We can easily translate our algorithms to programs (close enough to code).
Course pseudocode requirements

In this course:

- pseudocode you **read** will adhere to the guidelines in the resource available on the course website, and
- pseudocode you **write** can be in Python, without powerful list methods (but you might be doing more work than necessary).

Example features:

- Use of functions with name and arguments in parentheses
- Branching and looping like in Python
- Use of keywords in capitals so you can distinguish it from Python

For analysis, make sure you only use built-in Python functions for implementation for which costs are provided (see website for document).

Note: Sometimes describing algorithms in sentences will be adequate for analysis. Always check to see what is required.
Pseudocode for TSP (evaluation)

Assumes: Existence of functions Vertices, Orderings, and Cost.

TSP(Graph)

INPUT: A graph Graph

OUTPUT: The smallest cost of a tour

1  Vertices ← Vertices(Graph)
2  Tours ← Orderings(Vertices)
3  Min ← Cost(Tours[0])
4  for each Tour in Tours
5      Current ← Cost(Tour)
6      if Current < Min
7         Min ← Current
8  return Min

Note: Use of left arrow for assignment, italics for pseudocode, boldface for reserved words, capitalization for variables, and all capitals for functions
Pseudocode template for exhaustive search (constructive)

Assumes: Existence of function $\text{Possibilities}$

$\text{Exhaustive}(I)$

$\text{INPUT:} \quad \text{An input } I$

$\text{OUTPUT:} \quad \text{A solution to } I$

1. $\text{Choices} \leftarrow \text{Possibilities}(I)$
2. $\text{for each Option in Choices}$
3. $\text{determine value for Option}$
4. $\text{compare all values}$
5. $\text{return best possibility}$
Q2: How can we form rough categories?

Constant  1, .5, 10, 7.6, 201
Logarithmic  \( \log_2 n, (\log_3 n)/5, 4\log_2 n - 6 \)
Linear  \( n, 5n, n/3, 4n + 2 \)
Quadratic  \( n^2, 5n^2 - 45n, n^2/2 + 6n - 34 \)
Exponential  \( 2^n, 2^n + n^6 + 12 \)

Key ideas:

- Determine group membership without determining details of a function.
- If two functions are in different groups, we have enough information to compare them.
- If functions are in the same group, we need more information to compare them.
Defining categories cleanly

Idea:

- Express each category in terms of a simple function.
- Simple functions do not have multiplicative factors or additive terms.

<table>
<thead>
<tr>
<th>Simple functions</th>
<th>Not simple functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.5, 10, 7.6, 201</td>
</tr>
<tr>
<td>( \log n )</td>
<td>((\log_3 n)/5), (4 \log_2 n - 6)</td>
</tr>
<tr>
<td>( n )</td>
<td>(5n, n/3, 4n+2)</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>(5n^2 - 45n, n^2/2 + 6n - 34)</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>(2^n + n^6 + 12)</td>
</tr>
</tbody>
</table>
Categories and memberships

- For a simple function \( g(n) \), you can write \( f(n) \in \Theta(g(n)) \) to show that \( f(n) \) is in the category for \( g(n) \).
- Members of \( \Theta(g(n)) \) have \( g(n) \) as the dominant term, and can have multiplicative factors and smaller additive terms.
- Since \( \log_a x = \log_b x / \log_b a \), constant base multiplicative factor from \( \log_2 \) (written log).

**Categories**

\[ \Theta(1) \]
\[ \Theta(\log n) \]
\[ \Theta(n) \]
\[ \Theta(n^2) \]
\[ \Theta(2^n) \]

**Members of categories**

\[ 1, .5, 10, 7.6, 201 \]
\[ \log n, (\log_3 n)/5, 4\log_2 n - 6 \]
\[ n, 5n, n/3, 4n + 2 \]
\[ n^2, 5n^2 - 45n, n^2/2 + 6n - 34 \]
\[ 2^n, 2^n + n^6 + 12 \]
Describing functions using partial information

What if you have only partial information about a function $f(n)$, such as a lower bound or an upper bound?

Very roughly speaking (formal definitions to follow):

- $f(n)$ is in $O(g(n))$ ("Big O") means there is an upper bound on $f(n)$, based on $g(n)$.
- $f(n)$ is in $\Omega(g(n))$ ("Big Omega") means there is a lower bound on $f(n)$, based on $g(n)$.
- $f(n)$ is in $\Theta(g(n))$ ("Theta") means that $f(n)$ is in $O(g(n))$ and that $f(n)$ is also in $\Omega(g(n))$. 
Asymptotic notation (or order notation) is a way of quickly comparing options without getting mired in insignificant details; asymptotic refers to the classification of functions by their behaviour as the size of the input increases towards infinity.

\( f(n) \) is in \( O(g(n)) \) if there is a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq cg(n) \) for every \( n \geq n_0 \).

\( f(n) \) is in \( \Omega(g(n)) \) if there is a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq cg(n) \) for every \( n \geq n_0 \).

\( f(n) \) is in \( \Theta(g(n)) \) if there are real constants \( c_1 > 0 \) and \( c_2 > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( c_1 g(n) \leq f(n) \leq c_2 g(n) \) for every \( n \geq n_0 \).
Using asymptotic notation

Describe algorithms using sentences such as:

- “The running time of the algorithm is in $O(g(n))$.”
- “These lines of the algorithm can be executed in $O(g(n))$ time.”

Notes:

- Here $g(n)$ is something simple like 1, $\log n$, $n$, not 5, $3\log n + 5$, or $n/4 - \log \log n + 21$.
- We don’t give the specific $c$ (multiplicative factor) or $n_0$ (minimum size of value for comparison).
- An upper bound on an algorithm is an upper bound on the complexity of a problem, but a lower bound on an algorithm is not necessarily a lower bound on the complexity of a problem.
Suppose we have algorithms $A$ and $B$ with running times $f(n)$ and $g(n)$.

- If we know that $f(n)$ is in $O(n)$ and $g(n)$ is in $O(2^n)$, do we know that $A$ is a better choice than $B$?
- If we know that $f(n)$ is in $\Omega(n)$ and $g(n)$ is in $\Omega(2^n)$, do we know that $A$ is a better choice than $B$?
- If we know that $f(n)$ is in $\Theta(n)$ and $g(n)$ is in $\Theta(2^n)$, do we know that $A$ is a better choice than $B$?
- If we know that $f(n)$ is in $\Theta(n)$ and $g(n)$ is in $\Theta(n)$, are they equally good choices?
Comparing order notation on multiple variables

- \( f(n, m) = n^2 + 5n + 3m + 1 \) is in \( \Theta(n^2 + m) \)
- \( g(n, m) = 5mn + m - n + 3 \) is in \( \Theta(mn) \)
- \( h(n, m) = n \log n + 5n + 3m^2 + m \log m \) is in \( \Theta(n \log n + m^2) \)

Using multiple variables

- Using knowledge about relative values of variables is OK.
- Making assumptions about relative values is NOT OK.

Examples:
- If \( m \in \Theta(n^3) \), then \( f(n, m) \) is in \( \Theta(n^3) \).
- If \( m \in \Theta(n) \), then \( f(n, m) \) is in \( \Theta(n^2) \).
Q3: How can a single number represent all inputs of a given size?

Algorithm A

Algorithm B

Algorithm C

Algorithm D
Choosing the average case

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( l_1 )</th>
<th>( l_2 )</th>
<th>( l_3 )</th>
<th>( l_4 )</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>62.5</td>
</tr>
<tr>
<td>C</td>
<td>120</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>45</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>40</td>
<td>80</td>
<td>40</td>
<td>50</td>
</tr>
</tbody>
</table>
Considering other probability distributions

**Average case:** The value of $f(k)$ is the sum over all inputs $I$ of size $k$ of the probability of $I$ multiplied by the running time of the algorithm on input $I$.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>$Pr[I_1]$</th>
<th>$Pr[I_2]$</th>
<th>$Pr[I_3]$</th>
<th>$Pr[I_4]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_1$</td>
<td>1/10</td>
<td>3/10</td>
<td>3/10</td>
<td>3/10</td>
</tr>
<tr>
<td>$\delta_2$</td>
<td>7/10</td>
<td>1/10</td>
<td>1/10</td>
<td>1/10</td>
</tr>
<tr>
<td>$\delta_3$</td>
<td>3/10</td>
<td>3/10</td>
<td>1/10</td>
<td>3/10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$l_3$</th>
<th>$l_4$</th>
<th>$\delta_1$</th>
<th>$\delta_2$</th>
<th>$\delta_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>73</td>
<td>31</td>
<td>59</td>
</tr>
<tr>
<td>C</td>
<td>120</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>90</td>
<td>51</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>40</td>
<td>80</td>
<td>40</td>
<td>52</td>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>
Choosing the best case or the worst case

**Best case:** The value of $f(k)$ is the fastest running time of the algorithm on any input of size $k$.

**Worst case:** The value of $f(k)$ is the slowest running time of the algorithm on any input of size $k$. 
Relationships among the cases

- Worst case, best case, and average case are all functions; each can be described with any of $O$, $\Omega$, and $\Theta$.
- Each function is formed from a “representative value” chosen for each input size.
- For any input size, the representative value for best case is less than or equal to the representative value for average case which in turn is less than or equal to the representative value for worst case.
- The functions can be equal, if the same representative values are chosen.

Using worst case

To provide a guarantee, we will use worst case as our standard for comparison of algorithms and problems.
Q4: How is input size measured?

The size of the input is the number of bits used in the encoding. The number of bits depends on the number of data items and their types. We use the following assumptions:

- A set: The size of the set
- A sequence (string, Python list, Python tuple): The length of the sequence
- A grid: The product of the two dimensions
- A tree: The number of vertices in the tree
- A graph: The number of vertices and the number of edges

Notes:

- We are implicitly using order notation.
- In a tree, the number of edges is one smaller than the number of vertices.
- The costs of operations on grids and graphs are each defined in terms of two variables, not just one.
- For very large data items, the size of the data items will also be included in the size measurement.
Q5: How can we use the rough categories to simplify analysis?

Using asymptotic notation means:

- Smaller terms do not matter
  - The dominant term is all that matters.
  - Compute a sum by keeping just the dominant term.
- Constant factors do not matter
  - Simple functions are all that matters.
  - Represent a constant number of steps with the same cost by a single step with that cost.

Recipe for calculating costs:

1. Break the lines into blocks.
2. Determine the cost of each block.
3. Compute the sum of the costs.
Breaking lines into blocks

Use the structure of the program.

Costs of types of blocks:

- **Sequential lines:** The sum of the cost of the lines
- **Branching:** The cost of evaluating conditions plus the maximum cost of any branch
- **Looping:** The sum of the costs of each iteration (iteration management and body of the loop)
- **Recursion:** To be discussed later in the course

Note: These basic ideas give only a rough upper bound on cost; sometimes more refined analysis is needed.
Distinguishing constant-time steps

Constant-time steps:

- Assigning a value to a variable
- Using a variable
- Using an arithmetic or Boolean operation or a comparison
- Determining which branch to take
- Returning a value using `return`

Not constant-time:

- Many Python list operations such as sorting, map, filter, and reduce
- Many Python dictionary operations

Costs for analysis will be provided either in the costs document on the course website or specified in the assignment or exam question.
Example using simple Python

def is_odd(num):
    return num % 2 == 1

print(is_odd(5))
Analyzing loops

Looping: The sum of the costs of each iteration (iteration management and body of the loop).

Common situation 1: The cost of the loop body is the same for each iteration and the cost of iteration management is in $\Theta(1)$.
Cost for situation 1: Product of the number of iterations and the cost of each iteration.

Common situation 2: The cost of the loop body is in $\Theta(i)$ for iteration $i$ and the cost of iteration management is in $\Theta(1)$.
Cost for situation 2: $\Theta(n^2)$ for $n$ the number of iterations.

Intuition: Arithmetic series $\sum_{i=1}^{n} i = n(n+1)/2$, as discussed in the math session.

Note: In general, analysis can be much more complex.
Example using pseudocode

**Nested(Num)**

**INPUT:** A positive integer Num

**OUTPUT:** A value calculated using nested loops

1. \[ Sum \leftarrow 0 \]
2. \[ \text{for } \text{Stage from 1 to Num} \]
3. \[ \text{for } \text{Count from 1 to Stage} \]
4. \[ Sum \leftarrow Sum + (\text{Stage-Count})^3 \]
5. \[ \text{return } Sum \]
Example using Python lists

def is_odd(num):
    return num % 2 == 1

def list_all_odd(num_list):
    new_list = []
    for item in num_list:
        if is_odd(item):
            new_list.append(item)
    return new_list
class Grid:
    ""
    Attributes: entries is a list of lists of entries
                rows is a positive integer (number of rows)
                cols is a positive integer (number of columns)
    ""
    def convert_grid_int(self):
        rows = self.rows
        cols = self.cols
        for row in range(rows):
            for col in range(cols):
                old_value = self.access(row, col)
                new_value = int(old_value)
                self.enter(row, col, new_value)

Note: Cost should be given as a function of $r$, the number of rows, and $c$, the number of columns.
Example of use of graphs.py

Discussed in Python session 1.

```python
from graphs import *

def tour_cost(graph, perm):
    total = 0
    length = len(perm)
    for index in range(length-1):
        new_weight = 
        graph.edge_weight(perm[index], perm[index+1])
        total = total + new_weight
    total = total + graph.edge_weight(perm[length-1], perm[0])
    return total
```

Note: Cost should be given as a function of $n$, the number of vertices in graph, and $m$, the number of edges in graph.
Using \( f(n) \) when \( n \) is an integer

Although we express running time as \( f(n) \), where \( n \) is the size of the input, the values of \( f(n) \) are really only defined for integer values of \( n \).

If we are splitting an input of size \( n \) into two pieces, we cannot assume each will be of size \( n/2 \), as \( n/2 \) might not be an integer.

We can assume the sizes of the two pieces are:

- \( \lfloor n/2 \rfloor \) (the floor of \( n/2 \))
- \( \lceil n/2 \rceil \) (the ceiling of \( n/2 \))

Whether \( n \) is even or odd, \( \lfloor n/2 \rfloor + \lceil n/2 \rceil = n \).

See the math guide for more information on floors and ceilings.
Q6: How do we know if an algorithm is the best possible?

The **complexity** of a problem is a way of describing the worst-case cost of the best algorithm for the problem.

An **upper bound** of $O(f(n))$ on a problem means that there exists an algorithm correctly solving the problem that in the worst case runs in time $O(f(n))$.

A **lower bound** of $\Omega(g(n))$ on a problem means that any algorithm that correctly solves the problem must use $\Omega(g(n))$ time in the worst case.

Note: An upper bound on an algorithm is an upper bound on the complexity of a problem. A lower bound on an algorithm is not necessarily a lower bound on the complexity of a problem.

A **complexity class** is a set of problems that can be solved with specified worst-case bounds on resources using specific models of computation.

We will discuss lower bounds in Module 6.
How to compare algorithms

Compare the same types of functions (worst-case, average-case, or best-case).

Do not set values of variables.

If functions are very different (e.g. linear vs. constant):
- No need to determine exact functions.
- Save work by getting rough notion of functions.

If functions are very close (e.g. both linear):
- It is not enough to use rough notion of functions.
- Both linear does not mean neither is better.
- More precise counting is necessary.
Summary: Analyzing algorithms

- Describe your algorithm using English or pseudocode in a way that makes the analysis clear.
- If you’re using Python as a shorthand, make sure that you are only using built-in functions with running times that have been given in class. Write pseudocode so that analysis is easy.
- For Python lists, strings, and tuples use only the operations specified on the handout, and use the running times provided.
- Figure out how the running time depends on the size of the input.
- Express running time using Θ notation for worst-case behaviour.
For many search problems, the worst case cost of an exhaustive search algorithm can be computed as $A + BC$ where:

- $A =$ cost of computing all possibilities
- $B =$ cost of trying one possibility
- $C =$ number of possibilities to try

Often $A$ is dominated by $BC$. 
Analyzing exhaustive search algorithms for optimization problems

For many optimization problems, the worst case cost of an exhaustive search algorithm can be computed as \( A + BC + D \) where:
- \( A \) = cost of computing all feasible solutions
- \( B \) = cost of calculating the value of one feasible solution
- \( C \) = number of feasible solutions to try
- \( D \) = comparing the values to determine the best

Often \( A \) and \( D \) are dominated by \( BC \).
Counting feasible solutions: All subsets

The number of subsets of \( n \) items of any size is the same as the number of bit vectors of length \( n \), where a bit vector is a sequence of 0’s and 1’s.

To interpret a bit vector as a subset, view each position in the sequence as corresponding to an item in the set. The subset for a bit vector consists of the items with 1’s in their positions.

The number of bit vectors is calculated as two choices for the first position multiplied by two choices for the second position and so on, or \( 2^n \) in total.
Counting feasible solutions: All subsets of a fixed size

A **combination** is a way of selecting items from a collection, where the order does not matter. The number of $k$-combinations of $n$ elements is the **binomial coefficient** ("$n$ choose $k$")

$$\binom{n}{k} = \frac{n(n-1) \cdots (n-k+1)}{k(k-1) \cdots 1}$$

You can generate subsets using $k$ nested loops, or by using the Python module *itertools*. Both methods will be covered as exercises in the first Python review session.

**Fact to remember for this course**

$$\binom{n}{k} \in \Theta(n^k)$$
Counting feasible solutions: All orderings

A permutation is an ordering of elements. The number of permutations of $n$ elements is written $n!$.

**Fact to remember for this course**

$n! \in O(n^n)$
Generating all orderings of vertices

Python has a module `itertools` that contains the following useful functions:

- `itertools.permutations`
- `itertools.combinations`

Python guide 1 gives more information on how to use these functions.

In the code fragment below, `graph` is a Graph:

```python
from graphs import *
vertices = graph.vertices()
print(list(itertools.permutations(vertices,len(vertices))))
```

Note: Use `list` to convert to a list.
Implementing exhaustive search for TSP

Uses tour_cost; discussed in Python session 1.

```python
from graphs import *
import itertools

def exhaustive_tsp(graph):
    vertices = graph.vertices()
    perms = itertools.permutations(vertices, len(vertices))
    all_orders = list(perms)
    min_cost = tour_cost(graph, all_orders[0])
    best_order = all_orders[0]
    for order in all_orders[1:]:
        new_cost = tour_cost(graph, order)
        if new_cost < min_cost:
            min_cost = new_cost
            best_order = order
    return best_order
```

Analyzing exhaustive search for TSP

Cost is $A + BC + D$

- $A =$ cost of computing all feasible solutions ($O(n^n)$)
- $B =$ cost of calculating the value of one feasible solution ($\Theta(n)$)
- $C =$ number of feasible solutions to try ($O(n^n)$)
- $D =$ comparing the values to determine the best ($O(n^n)$)

Total cost: $O(n^{n+1})$
Module summary

Topics covered:

- Comparing problems
- Expressing algorithms using models of computation and pseudocode
- Forming rough categories using asymptotic (order) notation
- Establishing upper and lower bounds
- Choosing among worst case, average case, and best case
- Measuring input size
- Simplifying algorithm analysis
- Determining complexity of problems
- Summary: Analyzing algorithms
- Analyzing exhaustive search
- Counting feasible solutions
- Generating all orderings
- Implementing and analyzing exhaustive search for TSP