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.
Being lucky

**What I said**
A problem being NP-complete means that it is unlikely that there exists an algorithm that is guaranteed to run in time polynomial in the size of the input and is guaranteed to produce the correct answer.

**What I didn’t say**
- The problem cannot be solved quickly on small instances.
- The problem cannot be solved quickly on special instances.

You might be lucky if:
- The problem instance is small. Exponential in a small number may still be a reasonable running time.
- The problem instance is of a special kind. Try solving **HAMiltonian Cycle** on a tree.
Searching in a methodical way

Example: Generate all sequences of size 3 over \(\{1, 2, 3, 4\}\).

Group sequences by the first number in the sequence, then the second, then the third.
Solutions and partial solutions

For many problems, the solution is formed from the input, such as:

- Ordering of elements
- Arrangement of elements
- Subset of elements
- Partition of elements

A **partial solution** is a partial specification that corresponds to the set of all solutions consistent with the information specified.

For example, 1?? is a partial solution for sequences of size 3 over \( \{1, 2, 3, 4\} \). It corresponds to all the sequences that have 1 in the first position.

Note: Because the input might be a graph and the elements might be vertices, to avoid confusion we refer to **nodes** in the search tree.
Search trees

- The root corresponds to the set of all the solutions.
- A leaf corresponds to a particular solution.
- The sets associated with the children of a node form a partition of the set of solutions associated with the node.
Exploring a search tree

Pro: A search tree contains all feasible solutions.

Con: Constructing the tree is very time-consuming.

Ideas to use:

- Create the tree as you search, forming only the nodes that you need.
- To explore a node, generate and explore the subtree rooted at each possible child.
- When a node has been completely explored, backtrack to continue exploration of its parent.
Using recursion to explore a tree

Base cases:
- Reach a solution
- Reach a partial solution that cannot be extended

General case:
- Recursion is used to form a bigger tree from the smaller trees rooted at the children of a node.
- The first child is explored completely before the second child.
- All children of a node are explored completely before backtracking from the node to its parent.

CS 234 covers a similar type of exploration of graphs, known as depth-first search.

Variants:
- Generate all solutions (return a list of all solutions found)
- Generate one solution (stop when one solution is found)
- Generate the best solution (keep track of the best solution so far)
Paradigm: Backtracking

Ideas:
- View a partial solution as a list.
- Try to extend a partial solution of length $k$ by adding an element in position $k + 1$.
- If extension is possible, process the new node next.
- If extension is not possible, *backtrack* by undoing the last decision.

Recipe for backtracking:
1. Specify a partial solution.
2. Determine how the children of a node are formed.
3. Choose when to backtrack.

Note: In contrast to greedy algorithms, backtracking algorithms do not eliminate solutions as they go, and decisions can be undone.
The implicit backtracking tree

- Each node represents a partial solution.
- The root of the tree is the initial partial solution.
- For a node with partial solution $p$ of length $k$:
  - If there are no candidates for position $k + 1$ in the list, the node is a leaf.
  - Otherwise, the children of a node represent all ways of extending the partial solution from size $k$ to size $k + 1$.

Note: The number of children of a node is equal to the number of candidates for the next position in the list.

Tip
Make sure that all possibilities are covered.
Template to find all solutions

def all_sols(instance):
    return extend([], instance, [])

def extend(part_sol, part_inst, sols_so_far):
    if is_solution(part_sol, part_inst):
        return sols_so_far + [part_sol]
    elif part_inst == None:
        return sols_so_far
    else:
        for option in all_options(part_sol, part_inst):
            sol = form_sol(option, part_sol, part_inst)
            inst = form_inst(option, part_sol, part_inst)
            sols_so_far = extend(sol, inst, sols_so_far)
        return sols_so_far
Observations on the template

Helper functions used by the template:

- **is_solution** determines if the partial solution is a complete solution
- **all_options** generates all ways of forming a child of a node
  - **form_sol** extends a partial solution to a bigger partial solution
  - **form_inst** modifies an instance to fit the bigger partial solution

Coding with the template:

- The first function is a non-recursive **wrapper function** that has the instance as an input.
- The instance might be an object with multiple fields, or multiple parameters might be used.
- In the base cases, a leaf is reached and if it contains a solution, it is added to the collection of solutions so far.
- The function returns a list of all solutions.
Using backtracking to generate sequences

Example: Generate all sequences of size 3 over \{1, 2, 3, 4\}.

1. Specify a partial solution. A sequence of length 0, 1, 2, or 3
2. Determine how the children of a node are formed. For any numbers not already in the solution, create a child by adding one number to the partial solution.
3. Choose when to backtrack. When the sequence has length three

Example: The node with the solution 2?? has as its children 21?, 23?, and 24?.
Forming the helper functions

**is_solution** determines if the length of the partial solution is three

**all_options** generates all values not yet in the partial solution

**form_sol** extends a partial solution to a bigger partial solution by adding a new value

**form_inst** modifies an instance by removing a value from the list of choices

In example code available on the website, notice the use of `copy.copy` to generate a new list.
def one_sol(instance):
    return search([], instance)

def search(part_sol, part_inst):
    if is_solution(part_sol, part_inst):
        return part_sol
    elif part_inst == None:
        return None
    else:
        for option in all_options(part_sol, part_inst):
            sol = form_sol(option, part_sol, part_inst)
            inst = form_inst(option, part_sol, part_inst)
            possible_sol = search(sol, inst)
            if possible_sol != None:
                return possible_sol
Observations on the template

- As before, use a non-recursive wrapper function.
- Here only one solution is returned.
- When a base case is reached, return a solution or None.
- In the general case, stop searching once a solution is found.
- The entire tree need not be searched.
Using backtracking to find one sequence

Example: Generate a sequence in descending order.

Example code is available on the website.

Is there a better way to find a sequence in descending order?
Integer knapsack

Input: A set of $n$ types of objects, where object $i$ has weight $w_i$ and value $v_i$, and a weight bound $W$

Output: A list of integers $c_1, \ldots, c_n$ such that $\sum_{i=1}^{n} c_i \cdot w_i \leq W$ and $\sum_{i=1}^{n} c_i \cdot v_i$ is maximized

Example: $n = 3$, $W = 10$

<table>
<thead>
<tr>
<th></th>
<th>$w_i$</th>
<th>$v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Using backtracking for **Integer Knapsack**

1. Specify a partial solution. A *sequence of objects with possible repeats*
2. Determine how the children of a node are formed. *For any object that fits in the available space, create a child by adding one new object*
3. Choose when to backtrack. *When there is not enough space for any object to be added*

Implementation options:

- Check if an object fits before adding it.
- Add objects that might exceed the capacity, instead checking the total capacity of a “partial solution” when processing it.

Key idea: Various choices can be made on exactly how to define a partial solution and how to process it.
Implicit backtracking tree for **Integer Knapsack**

Key idea: We will come back to the idea of how to fix the problem of having some solutions appear more than once.
Using backtracking for \textit{k-Colouring}

1. Specify a partial solution. \textit{An assignment of up to k colours to a subset of the vertices}
2. Determine how the children of a node are formed. \textit{For an uncoloured vertex, assign one of the k colours to it}
3. Choose when to backtrack. \textit{When all vertices have been coloured}

A colour for a vertex is \textbf{legal} if none of its neighbours has been assigned that colour.

1. Specify a partial solution. \textit{An assignment of up to k colours to a subset of the vertices such that each colour is legal}
2. Determine how the children of a node are formed. \textit{For an uncoloured vertex, assign any one of the k colours that is legal, if any}
3. Choose when to backtrack. \textit{When there is no possible legal colour for any uncoloured vertex}

Key idea: Choosing the second option means we can stop early if there is a vertex that can’t be coloured.
Fine-tuning \textit{k-Colouring}

When forming the child of a node, how do we choose which vertex to colour and which colour to assign?

Key idea: If we don’t pay attention to our choices, we may generate the same solution more than once.

Solution: Order the vertices so that in the child of the root, the first vertex is coloured, in the grandchildren the first two vertices are coloured, and so on.

Key idea: If we are very careful in our choices of ordering of vertices and colours, we might increase our chances of finding a solution sooner.
Using backtracking for optimal colouring

Goal: Find a colouring using the smallest number of colours.

Note: Search time can be improved by avoiding solutions that differ only by the colour given to a set of vertices.

Observations:

- The number of children of a node can be as big as the number of colours used so far plus 1 (a new colour).
- The algorithm can’t stop when a colouring is found, as a better one might exist.
- Backtracking only occurs when all vertices have been coloured (since it is always possible).

Key idea: If we’ve already found a $k$-colouring, stop searching on a branch in which an $(k+1)$st colour is needed.
Template to keep track of the best so far

def find_best(part_sol, part_inst, best_so_far):
    if is_solution(part_sol, part_inst):
        if best_so_far == None:
            return part_sol
        elif sol_value(part_sol) > sol_value(best_so_far):
            return part_sol
        else:
            return best_so_far
    else:
        value = best_so_far
        for option in all_options(part_sol, part_inst):
            sol = form_sol(option, part_sol, part_inst)
            inst = form_inst(option, part_sol, part_inst)
            value = find_best(sol, inst, value)
        return value
Observations on the template

- A non-recursive wrapper function can be used here too.
- Helper function sol_value computes a value of the solution.
- There is no early exit when a solution is found.
Ideas for refining backtracking

Ideas seen:

**Pruning**: Stop search as soon as it is clear that the partial solution cannot lead to a solution.

**Exploiting symmetry**: Avoid revisiting identical partial solutions.

**Comparing to best-so-far**: Stop search when a partial solution cannot lead to a solution better than the best found so far.

A new idea:

For a maximization (minimization) problem, use a **bounding function** to determine the maximum (minimum) value possible in the subset associated with a partial solution.

If the value of the function at a node is smaller (bigger) than the best value calculated so far, stop search from that node and backtrack.
Branch-and-bound

Ideas:

- At each node, compute a bound on value of any children.
- If this bound is worse than best solution we have so far, don’t search farther from this node.

Recipe for branch-and-bound:

1. Determine what to store at each node.
2. Decide how to generate the children of a node.
3. Specify what global information should be stored and updated.
4. Choose a bounding function.
Template for branch-and-bound

Add the following extra case:

```python
elif bound(part_sol, part_inst) < sol_value(best_so_far):
    return best_so_far
```

- A non-recursive wrapper function can be used here too.
- Helper function `bound` computes a bound.
- In comparing bounds, `<` is used for a maximization problem and `>` is used for a minimization problem.
Exploiting symmetry for knapsack

To avoid revisiting identical partial solutions, place an order on the items and never add an item that comes earlier in the ordering than the item just added.

1. Specify a partial solution. A sequence of objects ordered by object number, with possible repeats

2. Determine how the children of a node are formed. For any objects with numbers at least that of the last item in the sequence, create a child by adding one in the order of item number.

3. Choose when to backtrack. When there is not enough space for the object being added
The refined implicit tree
Choosing an order for items

To increase the chances of finding the best solution early, order items so that \( \frac{v_i}{w_i} \) decreases as \( i \) increases.

Example: \( n = 3, W = 10 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w_i )</th>
<th>( v_i )</th>
<th>( \frac{v_i}{w_i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4/3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5/4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>6/5</td>
</tr>
</tbody>
</table>
Using branch-and-bound for Knapsack

1. Determine what to store at each node. A partial solution, its weight, and its value.

2. Decide how to generate the children of a node. Each child is formed by adding a different object, if possible, where the object can be $k$ or greater, where $k$ is the last object added.

3. Specify what global information should be stored and updated. Keep track of the best complete solution so far.

4. Choose a bounding function. Add the value of the solution plus $v_k/w_k$ times the remaining weight, where the last item added was of type $k$.

Note: If only items of types 1 through $k$ have been taken at a given node, then we know that no child of the node can do better than the value of the bounding function.
Branch-and-bound tree for **Knapsack**
Using a search tree for Independent Set

Backtracking ideas:

1. Specify a partial solution. A set of vertices that form an independent set

2. Determine how the children of a node are formed. Add another vertex that has no edges to any of the vertices in the set so far

3. Choose when to backtrack. When no more vertices can be added

Refinements:

- Can we prune useless branches? Yes, we did so by only considering independent sets instead of any sets of vertices
- Is each solution explored only once? Not as currently defined
- Can we stop early if a better solution was already found? Not without extra calculations
Using a search tree for Vertex Cover

Backtracking ideas:

1. Specify a partial solution. A subgraph of the original graph and a set of vertices no longer in the graph that cover all edges that have been deleted.

2. Determine how the children of a node are formed. For a vertex in the subgraph, create one child in which the vertex is added to the vertex cover (and $G'$ is formed by removing $v$ and all incident edges) and create one child in which all neighbours of the node are added to the vertex cover (and $G'$ is formed by removing $v$ and all its neighbours and all incident edges).

3. Choose when to backtrack. When all edges have been covered.
Refining **Vertex Cover**

Refinements:

- Can we prune useless branches? *Not as currently defined*
- Is each solution explored only once? *Yes, as we always partition the remaining options*
- Can we stop early if a better solution was already found? *Yes, we can keep track of the best solution so far and stop if the partial solution is bigger than the solution so far*
Partition

**Input:** A group of $n$ objects, where object $i$ has weight $w_i$;

**Output:** A partition of the objects into groups $A$ and $B$ with minimum difference in the sum of the weights of each part.
Using branch-and-bound for \textbf{Partition}

Recipe for branch-and-bound:

1. Determine what to store at each node. \textit{An assignment of a subset of the objects to groups A and B}
2. Decide how to generate the children of a node. \textit{Each child is formed by adding an unassigned object to either A or B}
3. Specify what global information should be stored and updated. \textit{Keep track of the difference between sums of weights in the best complete solution so far.}
4. Choose a bounding function. \textit{Suppose the sum of weights of all the unassigned objects (x) is less than the difference between the sums of the weights in the two parts (y). Then we use y − x as our value. Otherwise, we use y as our value.}

Note: No bigger change in difference is possible than by putting all the items in the much-less-full part of the partition.
Exploiting nice properties

Idea: Develop fast algorithms under nice conditions, characterized as one or more **parameters** of the problem or the instance.

Examples:

- Vertex cover when the size of the vertex cover is 1. The parameter is the size of the vertex cover.
- Vertex cover when the maximum degree of any vertex in $G$ is 1. The parameter is the maximum degree of any vertex in $G$.

Note: The parameter should not limit the size of instance.

Not allowed: Vertex cover when the size of $G$ is constant. The parameter is the number of vertices in $G$. 
Finding parameterized algorithms

Can we find an algorithm with running time that depends on both the size of $G$ and the parameter?

- Aim for the part dependent on the size of $G$ to be polynomial in the size of $G$.
- Allow the part dependent on the parameter to be exponential (or worse), as this part will be small when the parameter is small.

A fixed-parameter tractable (FPT) algorithm runs in time $O(g(k)n^{O(1)})$, where $k$ is a parameter and $g$ is any function.

Caution: The function $g$ must be a function of $k$ only, and not a function of $n$. 
Using branching to find a fixed-parameter algorithm

**Branching** is one of the common paradigms used for fixed-parameter algorithms.

Idea: Use the parameter to bound the size of the search tree explored in the algorithm.

To obtain a fixed-parameter tractable algorithm:

- Ensure that exploring the bounded tree is sufficient. Show that if no solution is found in the tree, no solution exists.
- Ensure that the cost of creating and exploring the tree is in $O(g(k)n^{O(1)})$, where $k$ is a parameter and $g$ is any function.
Calculating the cost of creating and exploring the tree

Suppose we have a search tree such that:

- The height of the tree is at most $h$.
- The number of children of each node is at most $c$.
- The cost of processing a node is at most $p$.

The total number of nodes in the tree will be $O(c^h)$, as discussed in the math session, and hence the total cost will be in $O(c^h p)$.

One way to obtain total time in $O(g(k)n^{O(1)})$:

- Ensure $h \in O(k)$.
- Ensure $c \in O(1)$.
- Ensure $p \in O(f(k)n^{O(1)})$

Total cost is in $O(c^kf(k)n^{O(1)})$, or $O(g(k)n^{O(1)})$, for a function $g$ that is only a function of $k$, and not of $n$. 
FPT algorithm for **Vertex Cover**

Form a search tree with a partial vertex cover $P$ and a graph $H$ at each node, with the root storing the empty set and $G$.

Generating children from a node storing $(P, H)$:
- Choose an edge $\{u, v\}$ that has not yet been covered.
- Create two children as follows:
  - $P \cup \{u\}$ and the graph formed from $H$ by removing $u$ and all edges incident on $u$
  - $P \cup \{v\}$ and the graph formed from $H$ by removing $v$ and all edges incident on $v$

**Number of nodes in the tree**: $O(2^k)$ since two children of a node and only $k$ levels (add to the vertex cover at each level)

**Cost of processing a node**: Polynomial in the number of vertices in the graph

Total cost of $O(2^k n^{O(1)})$ is in FPT
Randomized algorithms

All the algorithms we have seen so far are deterministic, not randomized.

Characteristics of a randomized algorithm:

- The algorithm can flip coins to decide the course of action.
- The amount of randomness used is viewed as another resource, like time and space.
- The same algorithm on the same input may lead to different running times.
- The same algorithm on the same input may lead to different outputs.

Depending on the type of uncertainty we allow, we sacrifice either the guarantee on worst-case running time (considered in this module) or the guarantee on correctness (considered in the next module).
Compromising on speed

A Las Vegas algorithm is guaranteed to give the correct answer in expected polynomial time.

The term *average case* is used for deterministic algorithms, based on a probability distribution on the instances. A probability distribution is usually based on assumptions about distributions on instances.

The term *expected case* is used for randomized algorithms, based on the average over possible executions on a single instance. There is no dependence on assumptions about distributions on instances.
Using randomness to sidestep worst case

Key idea: Because the exact steps of the algorithm are not fixed, you can be “lucky” and avoid worst-case behaviour.

Example: In message-passing on a network of computers, if all computers are sending messages to other computers, the same link might be needed for many of the messages.

With high probability, sending messages to random locations is much faster.

Algorithm: Send messages to random locations and then from the random locations to the desired destinations.
Using randomness for sorting and selection

Use randomness to find a **pivot**.

**Sorting:**
- Compare each element to the pivot.
- Recursively sort elements smaller than the pivot.
- Recursively sort elements greater than the pivot.

**Selection:**
- Compare each element to the pivot.
- Recursively select from the correct piece.
Finding a good pivot

Various options:
- Use whatever pivot is found.
- Use only a “good enough” pivot.

Key ideas:
- The base case can be reached by reducing the problem size by at least $1/4 \Theta(\log n)$ times.
- The probability of getting a “good enough” pivot is $1/2$.

Expected running times are $\Theta(n \log n)$ for sorting and $\Theta(n)$ for selection.
Module summary

Topics covered:

- Search spaces
- Search trees
- Backtracking
- The implicit backtracking tree
- Ideas for refining backtracking
- Branch-and-bound
- Fixed-parameter algorithms
- Randomized algorithms