COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 22: Exhaustive Search & Backtracking I

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW7

Due Tuesday @ 11:59PM ET

HW7

Walkthrough videos Pt. 1 and Pt. 2

Recall where we ended last lecture…

What is Dynamic Programming?

- It is an algorithm design paradigm
  - like divide-and-conquer, greediness, etc. are algorithm design paradigms.
- Usually, it is for solving **optimization problems**
  - E.g., ***shortest***, ***best***, ***maximum/minimum*** option
  - (Fibonacci numbers aren't an optimization problem, but they are a good example of dynamic programming anyway…)
- Similar to greedy, there are two properties to look for…

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

1. Relate subproblems
   - How do subproblems build upon or use other subproblems?

1. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

1. Solve original problem

Example #1: Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

Step 1: Define recursive subproblem

Prefixes:

X	A	C	G	G	T		
Y	A	C	G	C	T	T	A

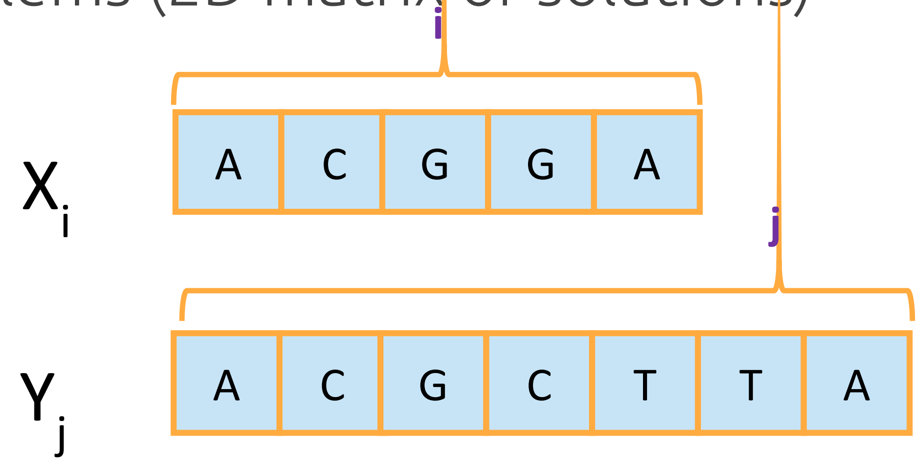
Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let $C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Examples:
$C[2,3] = 2$
$C[4,4] = 3$

Step 2: Relate subproblems

- Write C[i,j] in terms of the solutions to smaller sub-problems (2D matrix of solutions)



$C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Quick Overview of Approach

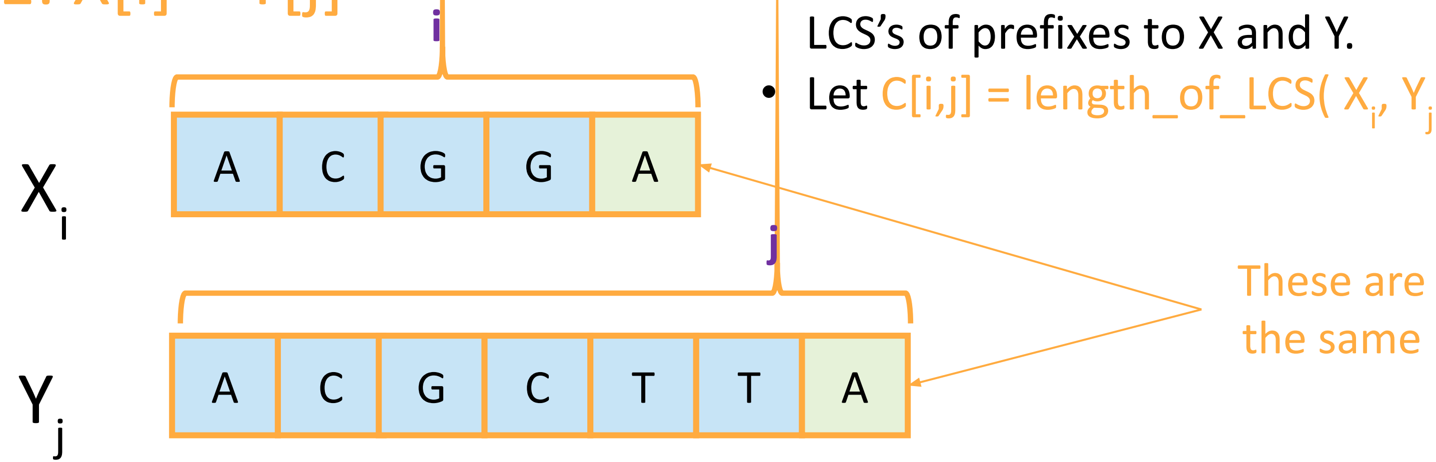
LCS(X = “ace”, Y = “abcde”) 1+LCS(X = “ac”, Y = “abcd”)

LCS(X = “aca”, Y = “abcde”) max(LCS(X=“ac”, Y=“abcde”), LCS(X=“aca”, Y = “abcd”))

Two cases

Case 1: X[i] = Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

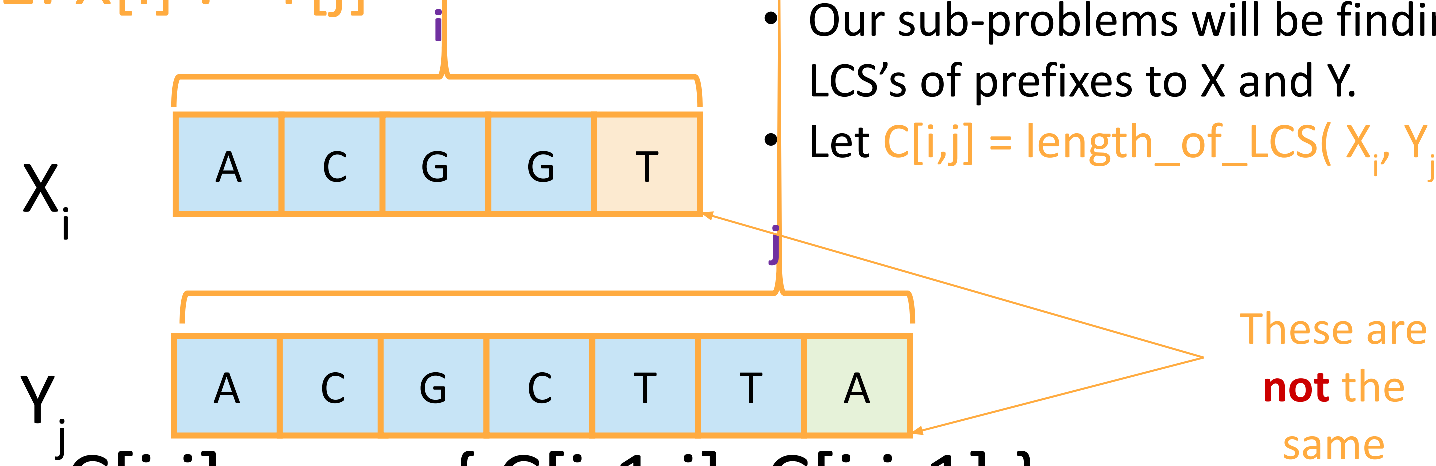


- Then C[i,j] = 1 + C[i-1,j-1].
  - because $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1})$ followed by $\boxed{A}$

Two cases

Case 2: X[i] != Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)



- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
  - either LCS(X_i,Y_j) = LCS(X_{i-1},Y_j) and T is not involved,
  - or LCS(X_i,Y_j) = LCS(X_i,Y_{j-1}) and A is not involved,

Recursive Formulation of the Optimal Solution

X_0							
Y_j	A	C	G	C	T	T	A

Case 0

- $C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$

Case 1

X_i	A	C	G	G	A		
Y_j	A	C	G	C	T	T	A

Case 2

X_i	A	C	G	G	T		
Y_j	A	C	G	C	T	T	A

Step #3/#4: Longest Common Subsequence

- LCS(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…n.
  - **For** i = 1,…,m
    - For j = 1,…,n:
      - **If** X[i] = Y[j]:
        - C[i,j] = C[i-1,j-1] + 1
      - **Else:**
        - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

Running time: O(nm)

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X	A	C	G	G	A

Y	A	C	T	G

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

So the LCM of X and Y has length 3.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Capacity: 10

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all items.
  - What's the most valuable way to fill the knapsack?

Total weight: 10
Total value: 42

- 0/1 Knapsack:
  - Suppose I have only one copy of each item.
  - What's the most valuable way to fill the knapsack?

Total weight: 9
Total value: 35

Step #1: Define recursive subproblem

- Sub-problems:
  - Unbounded Knapsack with a smaller knapsack.
  - K[x] = value you can fit in a knapsack of capacity x

First solve the problem for small backpack

Then larger …

Then larger …

Step #2: Relate subproblems

item i

- Suppose this is an optimal solution for capacity x:

Say that the optimal solution contains at least one copy of item i.

Weight w_i

Value v_i

Capacity x

Value V

- Then this is the optimal solution for capacity x - w_i:

Do we agree?

Capacity x – w_i

Value V - v_i

Step #2: Relate subproblems

- Let K[x] be the optimal value for capacity x.

$K[x] = \max_i \{$  $+$  $\}$

Optimal way to fill the smaller knapsack

The value of item i.

The maximum is over all i so that $w_i \leq x$.

$$K[x] = \max_i \{ K[x - w_i] + v_i \}$$

- (And K[x] = 0 if the maximum is empty).
  - That is, if there are no i so that $w_i \leq x$

Step #3/#4: Bottom Up Approach

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

$K[x] = \max_i \{$  +  $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Example

	0	1	2	3	4
K	0				
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[1] = ITEMS[0] + 

	0	1	2	3	4
K	0	1			
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[2] = ITEMS[1] + 

	0	1	2	3	4
K	0	1	2		
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[2] = ITEMS[0] + 💡

	0	1	2	3	4
K	0	1	4		
ITEMS		🐢	💡		

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{\, K[x], K[x - w_i] + v_i \,\}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[3] = ITEMS[2] + 🐢

	0	1	2	3	4
K	0	1	4	5	
ITEMS		🐢	💡	💡 🐢	

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[3] = ITEMS[0] + 

	0	1	2	3	4
K	0	1	4	6	
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[4] = ITEMS[3] + 🐢

	0	1	2	3	4
K	0	1	4	6	7
ITEMS		🐢	💡	🍉	🍉 🐢

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{\, K[x], K[x - w_i] + v_i \,\}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[4] = ITEMS[2] + 

	0	1	2	3	4
K	0	1	4	6	8
ITEMS					 

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[4] = ITEMS[2] + 💡

	0	1	2	3	4
K	0	1	4	6	8
ITEMS		🐢	💡	🍉	💡 💡

- UnboundedKnapsack(W, n, weights, values):
    - K[0] = 0
    - ITEMS[0] = Ø
    - **for** x = 1, …, W:
        - K[x] = 0
        - **for** i = 1, …, n:
            - **if** $w_i \leq x$:
                - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
            - If K[x] was updated:
                - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
    - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Final solution is K[4]
Max value of 8 using two 💡

Big Questions!

- What’s the overview of ES&B and when to use this approach?
- What is exhaustive search?
- What is backtracking?

Big Questions!

- What’s the overview of ES&B and when to use this approach?
- What is exhaustive search?
- What is backtracking?

Exhaustive Search & Backtracking

- Sometimes, the only way to solve a certain problem is through brute force, i.e. trying out every possible combination of values in order to get the correct answer. This process is called **exhaustive search**.
  - We have been dealing with $O(n^k)$ for the most part (polynomial time), but this approach gets into $O(2^n)$ which we've seen are very slow, but sometimes, it's the best we can do.
- We can reduce the cost in practice sometimes with **backtracking**, i.e. stopping early when we see we've hit a dead end while building our answer.
- The word "backtracking" is often colloquially used to refer to exhaustive search as well, even when there are no search constraints.

When to Use?

- When first working on an optimization problem, see if a Greedy approach might make sense.
- If not Greedy, see if a Dynamic Programming approach might make sense.
- If not Dynamic Programming, fall back to exhaustive search.
- Other clues to use backtracking include any time a problem asks for “all possibilities”, “all combinations”, “all options”, etc.
- Backtracking is often written recursively, and there are guidelines to use when designing your approach.

Big Questions!

- What's the overview of ES&B and when to use this approach?
- What is exhaustive search?
- What is backtracking?

Exhaustive Search General Approach

Pseudocode

- **Base case:** if there are no more decisions to be made, stop
- Otherwise, let's handle one decision now, and the rest with recursion.
  - "**Choose**" a choice from all possible choices C by modifying the possibility you are exploring
  - "**Explore**" future choices that could follow with recursion
  - "**Unchoose**" (if necessary), reverting our state to what it was before the "choose" step.

Questions to ask:

1. **Choose**: What are we choosing at each step? What are we stepping over?
2. **Explore:** How will we modify the arguments before recursing?
3. **Unchoose:** How do we un-modify the arguments (if needed)?
4. **Base case:** What should we do when finished? How to know when finished?

Example: Generate All Binary

Write a function that returns a vector of vector<bool> representing all binary values that have n digits.

Input: n

Output: a vector of all binary strings with exactly n digits.

Example: If n = 2, we want output {{0,0}, {0,1}, {1,0}, {1,1}}

Note: We could do this with bit arithmetic, but to practice exhaustive search, we will do it with recursion and string building.

1. **Choose**: What are we choosing at each step? What are we stepping over?
2. **Explore**: How will we modify the arguments before recursing?
3. **Unchoose**: How do we un-modify the arguments (if needed)?
4. **Base case**: What should we do when finished? How to know when finished?

Example: Generate All Binary

Write a function that returns a vector of vector<bool> representing all binary values that have n digits.

Input: n

Output: a vector of all binary strings with exactly n digits.

Example: If n = 2, we want output {{0,0}, {0,1}, {1,0}, {1,1}}

Note: We could do this with bit arithmetic, but to practice exhaustive search, we will do it with recursion and string building.

1. **Choose:** We'll iterate over each digit and choose whether it should be 1 or 0
2. **Explore:** Add 1 or 0 and recurse.
3. **Unchoose:** After exploring with 1 or 0 by pushing back, we want to remove it.
4. **Base Case:** When the length of vector<bool> we're building is equal to n, we add it to our final answer.

Let’s code it!!!

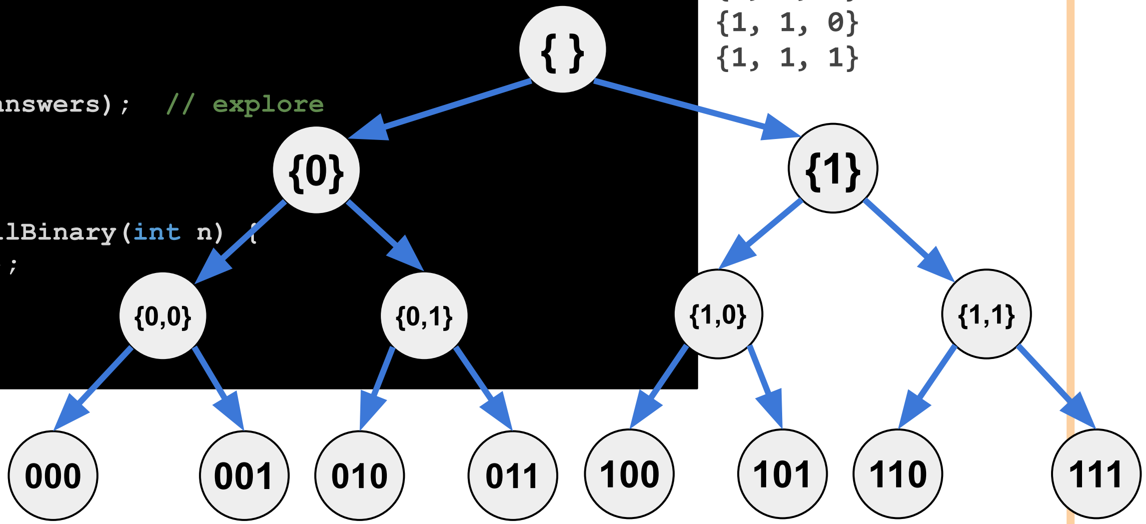
Generate All Binary Implementation

```
void generateAllBinaryHelper(
   int n, std::vector<bool> currBinary&, std::vector<std::vector<bool>>& answers) {
 if (n == currBinary.size()) {  // base case
   answers.push_back(currBinary);
   return;
 }
 currBinary.push_back(false);  // choose
 generateAllBinaryHelper(n, currBinary, answers);  // explore
 currBinary.pop_back();  // unchoose

 currBinary.push_back(true);  // choose
 generateAllBinaryHelper(n, currBinary, answers);  // explore
 currBinary.pop_back();  // unchoose
}

std::vector<std::vector<bool>> generateAllBinary(int n) {
 std::vector<std::vector<bool>> answers{};
 generateAllBinaryHelper(n, {}, answers);
 return answers;
}
```

```
generateAllBinary(3)
output:
{0, 0, 0}
{0, 0, 1}
{0, 1, 0}
{0, 1, 1}
{1, 0, 0}
{1, 0, 1}
{1, 1, 0}
{1, 1, 1}
```



- Helper gives us a way to keep track of variables between calls that we don't need to expose to a caller.
- Initializing answers to an empty vector and having the reference across function calls allows us to conveniently push_back answers.

Generate All Binary Implementation

```
void generateAllBinaryHelper(
   int n, std::vector<bool> currBinary&, std::vector<std::vector<bool>>& answers) {
 if (n == currBinary.size()) {  // base case
   answers.push_back(currBinary);
   return;
 }
 currBinary.push_back(false);  // choose
 generateAllBinaryHelper(n, currBinary, answers);  // explore
 currBinary.pop_back();  // unchoose

 currBinary.push_back(true);  // choose
 generateAllBinaryHelper(n, currBinary, answers);  // explore
 currBinary.pop_back();  // unchoose
}

std::vector<std::vector<bool>> generateAllBinary(int n) {
 std::vector<std::vector<bool>> answers{};
 generateAllBinaryHelper(n, {}, answers);
 return answers;
}
```

Suppose now we want to return all n-digit numbers in base-10. What do we change?

```
generateAllDecimal(2) output:
{0,0}
{0,1}
{0,2}
{0,3}
...
{9,8}
{9,9}
```

Let's code it!!!

Generate All Decimal Implementation

```
void generateAllDecimalHelper(
   int n, std::vector<int> currDecimal&, std::vector<std::vector<int>>& answers) {
 if (n == currDecimal.size()) {   // base case
   answers.push_back(currDecimal);
   return;
 }

 for (int i = 0; i < 10; i++) {
   currDecimal.push_back(i); // choose
   generateAllDecimalHelper(n, currDecimal, answers); // explore
   currDecimal.pop_back(); // unchoose
 }

}

std::vector<std::vector<int>> generateAllDecimal(int n) {
 std::vector<std::vector<int>> answers{};
 generateAllDecimalHelper(n, {}, answers);
 return answers;
}
```

Suppose now we want to return all n-digit numbers in base-10. What do we change?

```
generateAllDecimal(2)
output:
{0,0}
{0,1}
{0,2}
{0,3}
...
{9,8}
{9,9}
```

Big Questions!

- What's the overview of ES&B and when to use this approach?
- What is exhaustive search?
- What is backtracking?

Recursive Backtracking General Approach

- Backtracking is essentially exhaustive search with conditions. We make sure we're only exploring valid choices, and our base case may include checks to see if we can stop exploring the current path.
- We ask questions in a similar way as before, so writing the exhaustive search version then adding constraints later is generally a good approach.

1. **Choose**: What are we choosing at each step? What are we stepping over?
2. **Explore:** How will we modify the arguments before recursing?
3. **Unchoose:** How do we un-modify the arguments (if needed)?
4. **Base case:** What should we do when finished? How to know when finished?

Example: Dice Sum

Write a function that takes # of dice to roll and a desired sum of all values then outputs all possible rolls that will give exactly that sum.

Input: number of dice to roll d, and a desired sum to roll n
Output: all possibilities that add to that sum
Example: diceSum(2, 4) = {{1, 3}, {2, 2}, {3, 1}}

1. Choose: What are we choosing at each step? What are we stepping over?
2. Explore: How will we modify the arguments before recursing?
3. Unchoose: How do we un-modify the arguments (if needed)?
4. Base case: What should we do when finished? How to know when finished?

Example: Dice Sum

Write a function that takes # of dice to roll and a desired sum of all values then outputs all possible rolls that will give exactly that sum.

Input: number of dice to roll d, and a desired sum to roll n
Output: all possibilities that add to that sum
Example: diceSum(2, 4) = {{1, 3}, {2, 2}, {3, 1}}

1. **Choose:** We'll iterate over each dice and choose whether it should be 1, 2, … 6
2. **Explore:** Add one of them and recurse
3. **Unchoose:** After exploring a value for a dice, remove before exploring the next.
4. **Base Case:** When the length of diceRolls we're building is equal to d, we are finished and check to see if we should add this to our vector of final answers.

Let’s code it!!!

Dice Sum Implementation with No Constraints

```
void diceSumHelper(
 int diceLeft, int desiredSum, int currentSum,
 std::vector<int>& currentRolls, std::vector<std::vector<int>>& answers
) {
 if (currentSum == desiredSum && diceLeft == 0) {
   answers.push_back(currentRolls);
   return;
 } else if (diceLeft == 0) {
   return;
 // } else if (CHANGE_HERE) {
 //   return;
 } else {
   for (int i = 1; i < 7; i++) {
     currentRolls.push_back(i);
     diceSumHelper(diceLeft - 1, desiredSum, currentSum + i, currentRolls, answers);
     currentRolls.pop_back();
   }
 }
}

std::vector<std::vector<int>> diceSum(int numDice, int desiredSum) {
 std::vector<std::vector<int>> answers{};
 std::vector<int> currentRolls;
 diceSumHelper(numDice, desiredSum, 0, currentRolls, answers);
 return answers;
}
```

Suppose we have to roll a sum of 20 with four dice, but our first 2 dice are 1s

Suppose we have to roll a sum of 7 with four dice, but our first two dice sum up to 6.

COMP 285
Analysis of Algorithms

Welcome to COMP 285

Lecture 22: Exhaustive Search & Backtracking I

Lecturer: Chris Lucas (cflucas@ncat.edu)