COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 23: Exhaustive Search & Backtracking II

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW7

Due @ 11:59PM ET

HW8

Released by EoW, due 12/01 @ 11:59PM ET

Final Survey

See Tolu’s email!

+1% for >= 80% completion

Quiz!

www.comp285-fall22.ml or Blackboard

Recall where we ended last lecture...

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 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.

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:** 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.

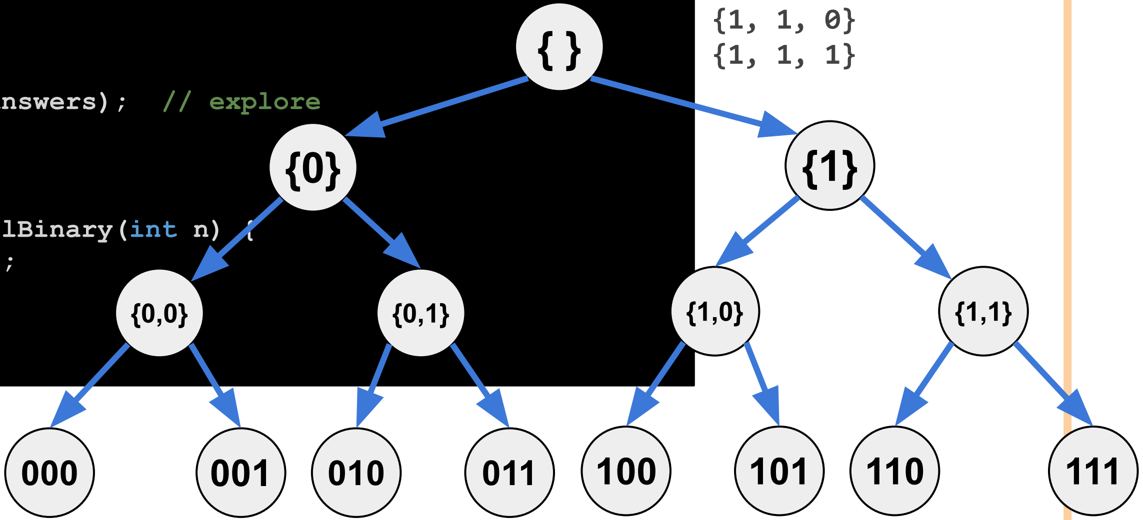
Generate All Binary Implementation

```
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 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;
}
```

Example #1: 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.

Example #1: Dice Sum Implementation

```
void diceSumHelper(int diceLeft, int desiredSum, int currentSum,
                   std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  }
  // recursive case
  for (int i = 1; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelper(diceLeft-1, desiredSum, currentSum+i, currentRolls); // explore
    currentRolls.pop_back();  // unchoose
  }
}
```

Example #1: Dice Sum Implementation

```
void diceSumHelper(int diceLeft, int desiredSum, int currentSum,
                   std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  }
  // recursive case
  for (int i = 1; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelper(diceLeft-1, desiredSum, currentSum+i, currentRolls); // explore
    currentRolls.pop_back();  // unchoose
  }
}
```

Can we do better?

Example #1: Dice Sum Implementation

```
void diceSumHelper(int diceLeft, int desiredSum, int currentSum,
                   std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  }
  // recursive case
  for (int i = 1; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelper(diceLeft-1, desiredSum, currentSum+i, currentRolls); // explore
    currentRolls.pop_back();  // unchoose
  }
}
```

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.

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Example #1: Dice Sum Implementation

```
void diceSumHelper(int diceLeft, int desiredSum, int currentSum,
                   std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  } else if ((currentSum + diceLeft * 1) > desiredSum || (currentSum + diceLeft * 6) < desiredSum){
    return;
  }
  // recursive case
  for (int i = 1; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelper(diceLeft - 1, desiredSum, currentSum + i,
                  currentRolls); // explore
    currentRolls.pop_back();     // unchoose
  }
}
```

There's no hope of finding a solution at the end of this path...

Big Questions!

- What are combinations versus permutations?
- What are other examples of exhaustive search + backtracking?

Big Questions!

- What are combinations versus permutations?
- What are other examples of exhaustive search + backtracking?

Combinations vs. Permutations

- What if the order of our selection or results do not matter (such that we are dealing with **combinations** instead of **permutations**).
  - Combination Example: all the possible teams of 2 you can form from 10 people
  - Permutation Example: all the possible 7-digit phone numbers you can from digits

- For example with diceSum, what if we now want to treat {1, 3} and {3, 1} as the same roll?

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Poll: Combinations versus Permutations

Output all the words you can spell from a set of letters.

1. Permutations

Output all the smoothies that can be made from a set of fruits

2. Combinations

Output all the types of pizzas you can make from a set of toppings

2. Combinations

Output all the possible schedules for how a set of tasks can be completed

1. Permutations

Exhaustive Search Combinations Combinations Intuition: double-nested for-loop

Example #1.5: Dice Sum Combination Implementation

- What if we now want to treat {1, 3} and {3, 1} as the same roll?

```
void diceSumHelper(int diceLeft, int desiredSum, int currentSum,
                   std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  } else if ((currentSum + diceLeft * 1) > desiredSum || (currentSum + diceLeft * 6) < desiredSum){
    return;
  }
  // recursive case
  for (int i = 1; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelper(diceLeft - 1, desiredSum, currentSum + i,
                  currentRolls); // explore
    currentRolls.pop_back();     // unchoose
  }
}
```

Let’s code it!!!

Example #1.5: Dice Sum Combination Implementation

```cpp
void diceSumHelperCombination(int diceLeft, int desiredSum, int currentSum,
                              int choiceIdx, std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  } else if (currentSum + diceLeft * 1 > desiredSum || currentSum + diceLeft * 6 < desiredSum) {
    return;
  }
  // recursive case
  for (int i = choiceIdx; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelperCombination(diceLeft - 1, desiredSum, currentSum + i, i, currentRolls); // explore
    currentRolls.pop_back();                                   // unchoose
  }
}
```

Big Questions!

- What are combinations versus permutations?
- What are other examples of exhaustive search + backtracking?

Example #2: Subsets

Given an vector<int> nums of unique elements, return all possible subsets (the power set).

Input: vector<int> nums of unique integer values

Output: all possible subsets

Example: nums = {1,2,3} should output {{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}

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?

Let's code it!!!

Example #2: Subsets Implementation

```
void findAllSubsetsHelper(vector<int> nums, int choiceIdx, vector<int> currCombo) {
  if (choiceIdx == nums.size()) {
    printAnswer(currCombo);
    return;
  }
   // not choose item
  findAllSubsetsHelper(nums, choiceIdx + 1, currCombo);

  // choose item
  currCombo.push_back(nums[choiceIdx]);
  findAllSubsetsHelper(nums, choiceIdx + 1, currCombo);
  currCombo.pop_back();
}
```

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Big Questions!

- What are combinations versus permutations?

- What are other examples of exhaustive search + backtracking?

In-class exercise: Gift Card Balance

Given a vector<int> candidates of distinct prices and an int giftCardBalance, return a list of all unique combinations of items you could buy using your gift card. Assume there are **unlimited** copies of each item.

Input: vector<int> of unique integer prices and an int giftCardBalance
Output: all subsets of items with prices that sum to giftCardBalance
Example: candidates = {2, 3, 6, 7}, target = 7 returns {{2, 2, 3}, {7}}

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?

In-class exercise: Gift Card Balance

Given a vector<int> candidates of distinct prices and an int giftCardBalance, return a list of all unique combinations of items you could buy using your gift card. Assume there are **unlimited** copies of each item.

Input: vector<int> of unique integer prices and an int giftCardBalance
Output: all subsets of items with prices that sum to giftCardBalance
Example: candidates = {2, 3, 6, 7}, target = 7 returns {{2, 2, 3}, {7}}

1. **Choose:** We'll iterate over each number and choose whether or not it should be included. We will use choiceIdx to ensure order does not matter.
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?

In-class exercise: Gift Card Balance

Given a vector<int> candidates of distinct prices and an int giftCardBalance, return a list of all unique combinations of items you could buy using your gift card. Assume there are **unlimited** copies of each item.

Input: vector<int> of unique integer prices and an int giftCardBalance
Output: all subsets of items with prices that sum to giftCardBalance
Example: candidates = {2, 3, 6, 7}, target = 7 returns {{2, 2, 3}, {7}}

1. **Choose:** We'll iterate over each number and choose whether or not it should be included. We will use choiceIdx to ensure order does not matter.
2. **Explore:** Purchase the item (or not), update giftCardBalance, and recurse.
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?

In-class exercise: Gift Card Balance

Given a vector<int> candidates of distinct prices and an int giftCardBalance, return a list of all unique combinations of items you could buy using your gift card. Assume there are **unlimited** copies of each item.

Input: vector<int> of unique integer prices and an int giftCardBalance
Output: all subsets of items with prices that sum to giftCardBalance
Example: candidates = {2, 3, 6, 7}, target = 7 returns {{2, 2, 3}, {7}}

1. **Choose**: We'll iterate over each number and choose whether or not it should be included. We will use choiceIdx to ensure order does not matter.
2. **Explore**: Purchase the item (or not), update giftCardBalance, and recurse.
3. **Unchoose**: Un-purchase the item, update the giftCardBalance
4. **Base case**: What should we do when finished? How to know when finished?

In-class exercise: Gift Card Balance

Given a vector<int> candidates of distinct prices and an int giftCardBalance, return a list of all unique combinations of items you could buy using your gift card. Assume there are **unlimited** copies of each item.

Input: vector<int> of unique integer prices and an int giftCardBalance
Output: all subsets of items with prices that sum to giftCardBalance
Example: candidates = {2, 3, 6, 7}, target = 7 returns {{2, 2, 3}, {7}}

1. Choose: We'll iterate over each number and choose whether or not it should be included. We will use choiceIdx to ensure order does not matter.
2. Explore: Purchase the item (or not), update giftCardBalance, and recurse.
3. Unchoose: Un-purchase the item, update the giftCardBalance
4. Base case: ???

Big Questions!

- What are combinations versus permutations?
- What are other examples of exhaustive search + backtracking?

Constraint Satisfaction Problems

- Problems that have requirements, and we need to search all possibilities then check whether they have the requirements.
- *Sudoku*

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- *N-Queens*: given a NxN chess board, place N queens on the board without any of queens attacking each other (attack demo, backtrack demo)

Takeaways

- We can use exhaustive search & backtracking to discover all **permutations** (order matters) and all **combinations** (order does not matter), usually with the help of a choiceIdx.
- To solve an exhaustive search / backtracking problem, remember the rough template / outline: create a helper, think about when you are finished building a potential answer, and plan how to *choose* / *explore* / *unchoose*.
- All possible subsets, sudoku solving, N-queens, etc. are classic problems for which exhaustive search / backtracking is necessary

COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 23: Exhaustive Search & Backtracking II

Lecturer: Chris Lucas (cflucas@ncat.edu)