Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski
Today’s topics:

- Recursion Week Fortnight continues!
- Today:
  - More *recursive backtracking* code:
    - Gift card spending optimization
Code Example #1

GIFT CARD SPENDING TARGET
Gift card spending optimization

- You’ve been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You’ll be given:
  - `int giftCardAmt`: The amount of the gift card
  - `Set<int> itemsForSale`: A set of prices of items for sale (assume only one of each item is in stock)

**Task**: Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card??

**Return**:
- `bool`: true if you can find such a collection, otherwise false
Gift card spending optimization

- You’ve been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You’ll be given:
  - `int giftCardAmt`: The amount of the gift card
  - `Vector<Item> itemsForSale`: A set of items for sale (each has name and price)
- **Task**: Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card?
- **Return**:
  - `bool`: true if you can find such a collection, otherwise false

Your Turn:
Help me write some test cases for this function. Come up with at least one basic correctness test, and a couple tricky/edge cases. **Submit yours at pollev.com/cs106b.** One test case per submission, you may submit multiple times.

**Format example:**

4, {1, 2, 5} = false
Backtracking template

```cpp
bool backtrackingRecursiveFunction(args) {
  › Base case test for success: return true
  › Base case test for failure: return false
  › Loop over several options for “what to do next”:
    1. Tentatively “choose” one option
    2. if (“explore” with recursive call returns true) return true
    3. else That tentative idea didn’t work, so “un-choose” that option, 
       but don’t return false yet!--let the loop explore the other options before giving up!
  › None of the options we tried in the loop worked, so return false
}
```
Backtracking template: applied to Gift Card problem

```c++
bool backtrackingRecursiveFunction(args) {
    › Base case test for success: return true
    › Base case test for failure: return false
    › Loop over several options for “what to do next”:
        1. Tentatively “choose” one option
        2. if ("explore" with recursive call returns true) return true
        3. else That tentative idea didn’t work, so “un-choose” that option,
           but don’t return false yet!--let the loop explore the other options before giving up!
    › None of the options we tried in the loop worked, so return false
}
```

What is success for this problem?
Backtracking template: applied to Gift Card problem

```c
bool backtrackingRecursiveFunction(args) {  
  › Base case test for success: return true  
  › Base case test for failure: return false  
  › Loop over several options for “what to do next”:
    1. Tentatively “choose” one option
    2. if ("explore" with recursive call returns true) return true
    3. else That tentative idea didn’t work, so “un-choose” that option,
       but don’t return false yet! -- let the loop explore the other options before giving up!
  › None of the options we tried in the loop worked, so return false
}
```

Exactly $0 left on card
Backtracking template: applied to Gift Card problem

```cpp
bool backtrackingRecursiveFunction(args) {
  › Base case test for success: return true
  › Base case test for failure: return false
  › Loop over several options for “what to do next”: 
    1. Tentatively “choose” one option
    2. if (“explore” with recursive call returns true) return true
    3. else That tentative idea didn’t work, so “un-choose” that option,
       but don’t return false yet!--let the loop explore the other options before giving up!
  › None of the options we tried in the loop worked, so return false
}
```
Backtracking template: applied to Gift Card problem

```c
bool backtrackingRecursiveFunction(args) {
    › Base case test for success: return true
    › Base case test for failure: return false
    › Loop over several options for “what to do next”:
        1. Tentatively “choose” one option
        2. if (“explore” with recursive call returns true) return true
        3. else That tentative idea didn’t work, so “un-choose” that option,  
            but don’t return false yet! --let the loop explore the other options before giving up!
    › None of the options we tried in the loop worked, so return false
}
```

- Exactly $0 left on card
- Overspend/negative balance, or no items left to choose.
Backtracking template: applied to Gift Card problem

```cpp
bool backtrackingRecursiveFunction(args) {
  › Base case test for success: return true
  › Base case test for failure: return false
  › Loop over several options for “what to do next”:
    1. Tentatively “choose” one option
    2. if (“explore” with recursive call returns true) return true
    3. else That tentative idea didn’t work, so “un-choose” that option, but don’t return false yet!--let the loop explore the other options before giving up!
  › None of the options we tried in the loop worked, so return false
}
```

Exactly $0$ left on card
Overspend/negative balance, or no items left to choose.

What is “one step” for this problem?
What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item.
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount.

Items:

$1  Y/N: ___
$5  Y/N: ___
$3  Y/N: ___
$2  Y/N: ___
$10 Y/N: ___
What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item.
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount.

Items:

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$1</td>
</tr>
<tr>
<td>Blueberries</td>
<td>$5</td>
</tr>
<tr>
<td>Strawberries</td>
<td>$3</td>
</tr>
<tr>
<td>Toilet paper</td>
<td>$2</td>
</tr>
<tr>
<td>Socks</td>
<td>$10</td>
</tr>
</tbody>
</table>

Y/N:
- $1: Y
- $5: ___
- $3: ___
- $2: ___
- $10: ___

One step/decision

Delegate the rest to recursion
What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount

<table>
<thead>
<tr>
<th>Items</th>
<th>$1</th>
<th>$5</th>
<th>$3</th>
<th>$2</th>
<th>$10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y/N:</td>
<td>Y</td>
<td>___</td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
</tbody>
</table>

One step/decision

If recursion comes back with the answer that no combination works for this set and the remaining funds, reconsider our Y on the banana.
What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item.
  - The yes’es and no’s can come in any combination, we have to find a combination that sums to our gift card amount.

<table>
<thead>
<tr>
<th>Items:</th>
<th>$1</th>
<th>$5</th>
<th>$3</th>
<th>$2</th>
<th>$10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y/N: Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conclusion: one step/decision has two options to “loop” over: Y and N (for one item).
Backtracking template: applied to Gift Card problem

```cpp
bool backtrackingRecursiveFunction(args) {
    › Base case test for success: return true
    › Base case test for failure: return false
    › Loop over several options for “what to do next”:
      1. Tentatively “choose” one option
      2. if (“explore” with recursive call returns true)
         return true
      3. else That tentative idea didn’t work, so “unchoose” that option, 
         but don’t return false yet!—let the loop explore the other options before giving up!
    › None of the options we tried in the loop worked, so return false
}
```

- Exactly $0 left on card
- Overspend/negative balance, or no items left to choose.
- Taking one item, “loop” over Y and N options for that item (we won’t actually loop since Y and N are only two options, a loop is excessive)
- If both Y and N options for an item fail, we’ve exhausted all possibilities, so return false.
Comparing our solution and the design template

Try both Y and N

bool backtrackingRecursiveFunction(args) {
  // Base case test for success: return true
  // Base case test for failure: return false
  // Loop over several options for “what to do next”:
  1. Tentatively “choose” one option
  2. if (“explore” with recursive call returns true) return true
  3. else That tentative idea didn’t work, so “un-choose” that option, but don’t return false yet!--let the loop explore the other options before giving up
  None of the options we tried in the loop worked, so return false
}

// base case success: card amount is spent down to 0 exactly
if (giftCardAmt == 0) {
  return true;
}
// base case failure: we either overspent, or we need to spend more but there are no more items for to consider, so we can’t succeed
if (giftCardAmt < 0 || index == itemsForSale.size()) {
  return false;
}
// recursive case: consider 1 next item (at `in
Item item = itemsForSale[index];
// Our two choices are that we can either BUY
// other additional purchases with less money
itemsToBuy.add(item);
if (canUseFullGiftCard(giftCardAmt - item.price)) {
  return true;
}
// ...or NOT BUY THE ITEM and go on to consider
// the same amount to spend.
itemsToBuy.remove(itemsToBuy.size() - 1);
if (canUseFullGiftCard(giftCardAmt, itemsForSale, itemsToBuy, index + 1)) {
  return true;
}
return false; // if neither of the two options can work, we have exha...
Code Example #2

GIFT CARD SPENDING OPTIMIZATION
Gift card spending optimization

- You’ve been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You’ll be given:
  - `int giftCardAmt`: The amount of the gift card
  - `Set<int> itemsForSale`: A set of prices of items for sale (assume only one of each item is in stock)
- **Task:** Of all the collections of items to buy, what is one that will sum the closest to the amount on the gift card?
  - This is a slight loosening of the exact match requirement of the original problem
What do we need to change?

// base case success: card amount is spent down to 0 exactly
if (giftCardAmt == 0) {
    return true;
}

// base case failure: we either overspent, or we need to spend more
// no more items for to consider, so we can't suc
if (giftCardAmt < 0 || index == itemsForSale.size()) {
    return false;
}

// recursive case: consider 1 next item (at `index`)
Item item = itemsForSale[index];
// Our two choices are that we can either BUY THE ITEM and go on to consider
// other additional purchases with less money to spend...
itemsToBuy.add(item);
if (canUseFullGiftCard(giftCardAmt - item.price, itemsForSale, itemsToBuy, index + 1))
    return true;

// ...or NOT BUY THE ITEM and go on to consider other additional purchases with
// the same amount to spend.
itemsToBuy.remove(itemsToBuy.size() - 1);
if (canUseFullGiftCard(giftCardAmt, itemsForSale, itemsToBuy, index + 1)) {
    return true;
}
return false; // if neither of the two options can work, we have exha