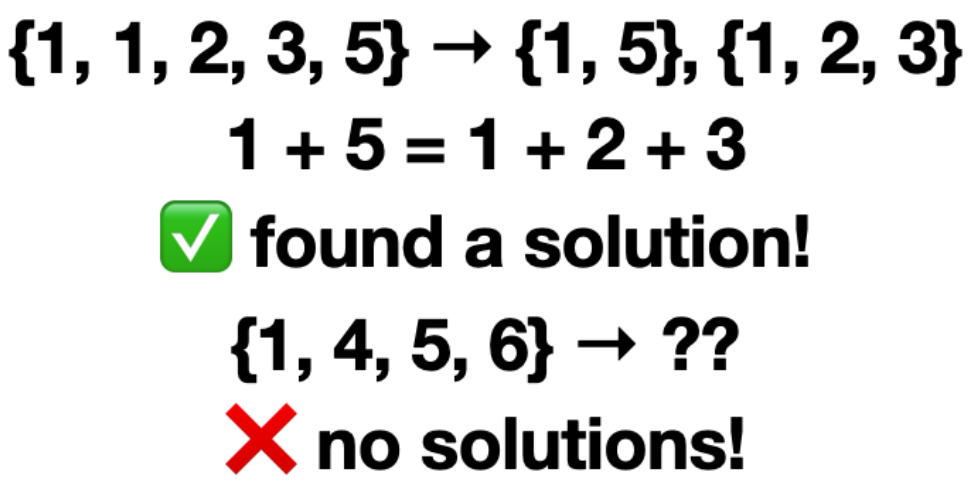
Recursive Backtracking

CS 106B: Programming Abstractions

Autumn 2020, Stanford University Computer Science Department

Lecturers: Chris Gregg and Julie Zelenski



Slide 2

Announcements

- Assignment 3 is due Friday at 11:59PM, PDT.
- There will be a recursion review session **today**, Wednesday, October 7 from 3:30-4:30pm PDT. You can join the session at this Zoom link.

Slide 3

Today's Goals:

- Introduce the idea of recursive *backtracking*
  - this is a "choose, explore, unchoose" strategy
- Talk about generating *subsets*

Slide 4

Subsets

The *power set* of set S is the set of all possible subsets of S. If the input contains the single element C, there are two possible subsets, one including C and the other empty:

```
{C}  {}
```

If the input contained element B in addition to C, we will also consider whether to include B. This power set contains four subsets:

```
{BC}  {B}  {C}  {}
```

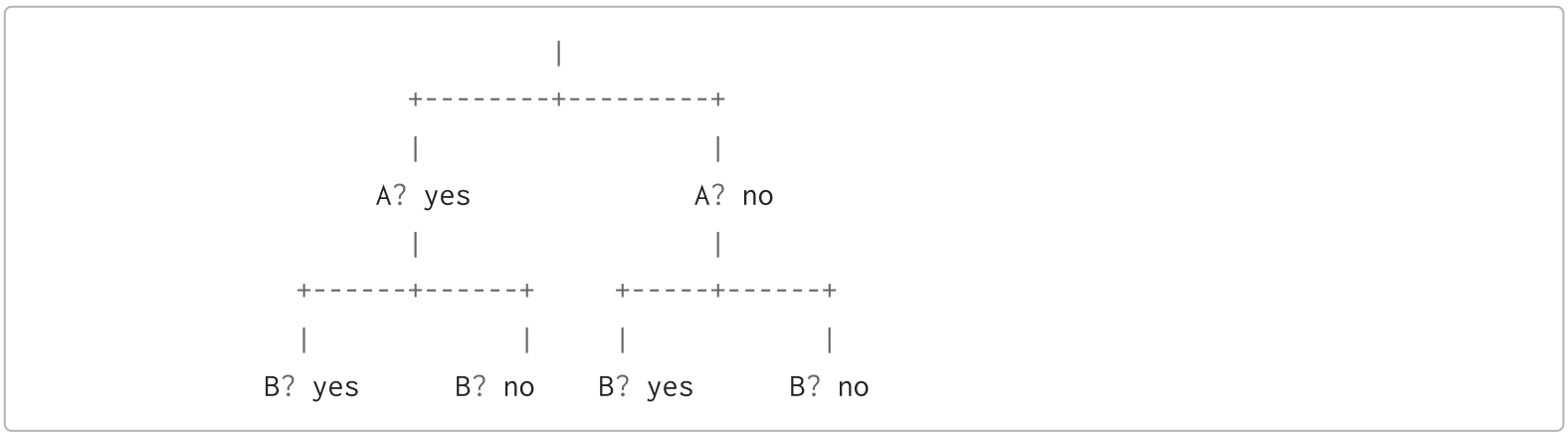
With a third element A under consideration, the power set has eight subsets:

```
{ABC}  {AB}  {AC}  {A}  {BC}  {B}  {C}  {}
```

Note how the size of power set doubles as we add an element. To construct the size N power set, we build on the N-1 power set and try adding the new element or not to each.

Let's draw a *decision tree* for generating subsets. Each level of the tree corresponds an element from the input that is being considered. The possible options for the element are to either include it in the current subset or not.

In the diagram below, each left arm in the tree indicates the option to include the current element, the right arm is without. Each path from the top to the bottom represents a sequence of recursive calls that has reached the base case. That path is one subset. You can determine which elements are contained in that subset by tracing the sequence of yes/no turns it takes.



Did you notice that the decision tree for subsets is structurally similar to the decision tree for coin-flipping? Each decision point has two options. The total number of paths to explore is 2^N.

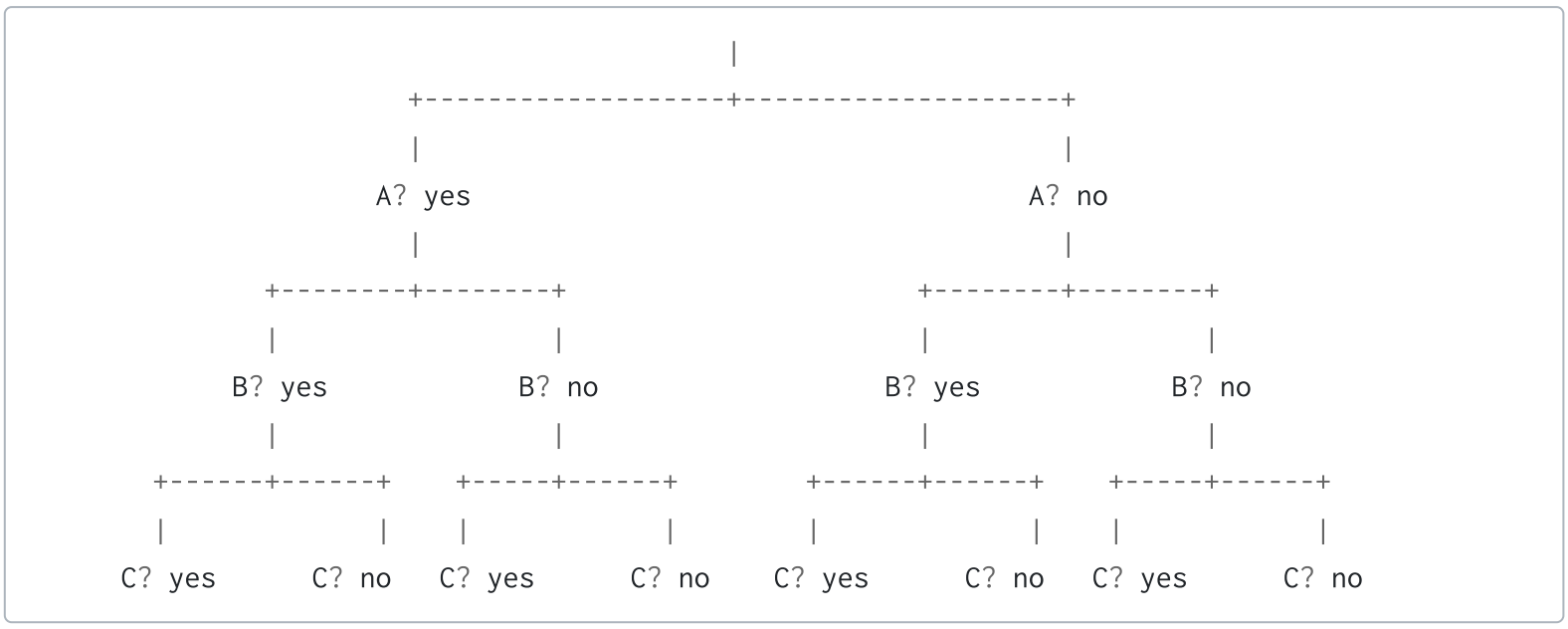
As before, the self-similarity leads to a very compact recursive solution:

```
void listSubsets(string input, string soFar)
{
    if (input.empty()) {
        cout << "{" << soFar << "}" < endl;
    } else {
        char consider = input[0];
        string rest = input.substr(1);
        listSubsets(rest, soFar + consider);  // explore with
        listSubsets(rest, soFar);             // explore without
    }
}
```

Slide 5

Tracing down and back

It is tempting to think that that recursion traverses the tree level-by-level or left to right, but this is not correct. Let's look at the path for the string **"ABC"**: it goes down to the far left, around the bottom, and back up the right side of each subtree.



The first path explored is the one that goes all the way left. At the base case, it prints that subset and control returns to the previous decision point, where the previous decision is undone and the other option is tried.

Many students can follow how the recursion moves downward in the tree, but are perplexed at how backtracking moves upwards. As we traverse downward, we move toward the base case. When we arrive at the base case, there is no further exploration possible from here. Control should now return to the previous decision point to explore other alternatives.

Let's set a breakpoint on the base case so we can stop in the debugger at the bottom of the tree. If we look at the call stack we can see the sequence of recursive calls so far. Each stack frame represents a decision point on the path from the start to here. Those stack frames waiting on the call stack are the "memory" of how we got here. When we reach the base case, we will pop its stack frame from the call stack and uncover the previous stack frame, which becomes the new topmost frame. Execution resumes in that frame and we pick up where we left off.

Understanding how/when/why the code backtracks to a previous decision point is perhaps the trickiest part of all in recursive backtracking. I highly recommend that you sketch a decision tree and walk through its traversal and/or step in the debugger to confirm your understanding of how it moves up and down the tree.

Slide 6

Choose-explore-unchoose

This *choose-explore-unchoose* structure is a classic pattern for recursive backtracking. Here it is summarized in pseudocode:

```
void explore(options, soFar)
{
    if (no more decisions to make) {
        // base case
    } else {
        // recursive case, we have a decision to make
        for (each available option) {
            choose (update options/soFar)
            explore (recur on updated options/soFar)
            unchoose (undo changes to options/soFar)
        }
    }
}
```

The details in the pseudocode are intentionally vague, e.g. what it means to "update options/soFar" or what is meant by "each available option". These details are specific to a particular search space. If you apply the general pattern to generating sequendes of coin flips, the concrete details become:

- State is length of sequence to generate, and sequence so far assembled
- The decision is what next flip to add to sequence
- Available options are H and T
- Update state by adding flip to current sequence, decrement length
- No explicit unchoose needed

Can you apply the general pattern to the letter sequences, permutations, and subsets?

Slide 7

Partitionable: determine whether a solution exists

- Write a function named **`partionable`** that takes a vector of ints and returns **`true`** if it is possible to divide the ints into two groups such that each group has the same sum. For example, the vector **`{1,1,2,3,5}`** can be split into **`{1,5}`** and **`{1,2,3}`**. However, the vector **`{1,4,5,6}`** can't be split into two.

  ```
  bool partitionable(Vector<int>& nums) { ...
  ```

- This is our first example of recursive backtracking where we make a change and must restore some data before we can move on; otherwise, the solution degrades.

- Basic idea:

  - Keep track of the two sums! Must use helper function:

    ```
    bool partitionable(Vector<int>& rest, int sum1, int sum2);
    ```

- Keep removing values from the vector until we have no more values left (base case). Replace the value we removed so we still have a valid vector for our next recursive call.
- Search each possible path
- Why do we pass in the vector by reference? Because we don't want to keep making copy after copy of the vector. If we *did* pass in the vector by value, we would not (in this case) have to replace what we remove from the vector at each stage, but that is the trade-off we are making.

Slide 8

Partitionable: let's code!

Slide 9

Partitionable: solution

```
bool partitionable(Vector<int>& nums) {
    return partitionable(nums, 0, 0); // no sums yet
}

bool partitionable(Vector<int>& rest, int sum1, int sum2) {
    if (rest.isEmpty()) {
        return sum1 == sum2;
    } else {
        int n = rest[0];
        rest.remove(0);
        bool answer = partitionable(rest, sum1 + n, sum2)
                   || partitionable(rest, sum1, sum2 + n);
        rest.insert(0, n);
        return answer;
    }
}
```

- Note that we actually don't need to go through all the solutions, necessarily, to find a correct one. Because of *short circuit evaluation*, if the first call to **`partitionable`** returns **`true`**, the second call is not ever evaluated. Neat!

Slide 10

What if we wanted to actually *find* a correct partition?

- Now we have a bigger challenge!
  - Now we have to carry along partitions until we find a correct one – that is a bit trickier.
  - We have to make sure to add/remove the values in each vector so that they remain viable for the next recursive move. That adds a bit of code:

```
bool partitionable(Vector<int> &rest, Vector<int>& v1, Vector<int>& v2) {
    if (rest.size() == 0) {
        int sum1 = 0;
        int sum2 = 0;
        for (int val : v1) {
            sum1 += val;
        }

        for (int val : v2) {
            sum2 += val;
        }

        return sum1 == sum2;
    } else {
        int n = rest[0];
        rest.remove(0);

        v1.add(n);

        bool answer1 = partitionable(rest, v1, v2);
        if (answer1) {
            rest.insert(0, n);
            return true;
        }

        v1.remove(v1.size() - 1);

        v2.add(n);

        bool answer2 = partitionable(rest, v1, v2);
        rest.insert(0, n);

        if (answer2) {
            return true;
        }

        v2.remove(v2.size() - 1);

        return false;
    }
}
```

- Notice that once we find a solution, we immediately return – we want to capture that solution, so we can't go doing more recursion.

Slide 11

What if we want to keep track of *all* solutions?

- Oh, boy! Now we have to keep track of a vector or set of solutins. *But*, we have to make sure to find all the solutions, which means no returning early. We have to be careful as before, but now we need to keep evaluating until we have exhausted the search space:

```
  void partitionable(Vector<int> &rest, Vector<int>& v1, Vector<int>& v2,
Vector<VectorPair>& allSolutions) {
      if (rest.size() == 0) {
          int sum1 = 0;
          int sum2 = 0;
          for (int val : v1) {
              sum1 += val;
          }

          for (int val : v2) {
              sum2 += val;
          }
          if (sum1 == sum2) {
              VectorPair vp;
              vp.v1 = v1;
              vp.v2 = v2;
              allSolutions.add(vp);
          }
      } else {
          int n = rest[0];
          rest.remove(0);

          v1.add(n);

          partitionable(rest, v1, v2, allSolutions);

          v1.remove(v1.size() - 1);

          v2.add(n);

          partitionable(rest, v1, v2, allSolutions);
          rest.insert(0, n);

          v2.remove(v2.size() - 1);
      }
  }
```