Admin

- ◇ Today's topics
  - Recursive backtracking
- ◇ Reading
  - Reader ch. 6 (today)
  - Next: pointers! 2.2-2.3, linked lists 9.5(sort of)
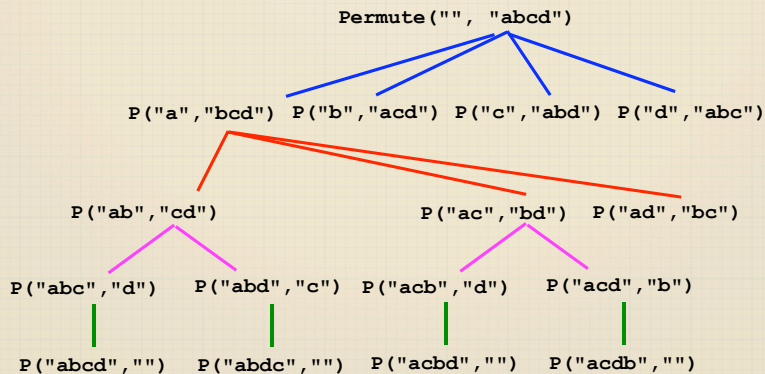- ◇ Terman cafe after class

Refresh: permute code

```
void RecPermute(string soFar, string rest)
{
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string next = soFar + rest[i];
            string remaining = rest.substr(0, i)
                              + rest.substr(i+1);
            RecPermute(next, remaining);
        }
    }
}


 // "wrapper" function
void ListPermutations(string s)
{
    RecPermute("", s);
}
```

Tree of recursive calls


```
                    Permute("", "abcd")


   P("a","bcd")  P("b","acd")  P("c","abd")  P("d","abc")


      P("ab","cd")              P("ac","bd")   P("ad","bc")


  P("abc","d")  P("abd","c")  P("acb","d")   P("acd","b")


P("abcd","")  P("abdc","")  P("acbd","")   P("acdb","")
```

Subsets

- ◇ Enumerate all subsets of input
  - "abc" has subsets "a", "b", "ab", "ac", …
  - Order doesn't matter, "ab" is same as "ba"
- ◇ Solving recursively
  - Separate one element from input
  - Can either include in current subset or not
  - Recursively form subsets including it
  - Recursively form subsets not including it
  - What is the base case?
- ◇ Remind you of any other problem you've seen?
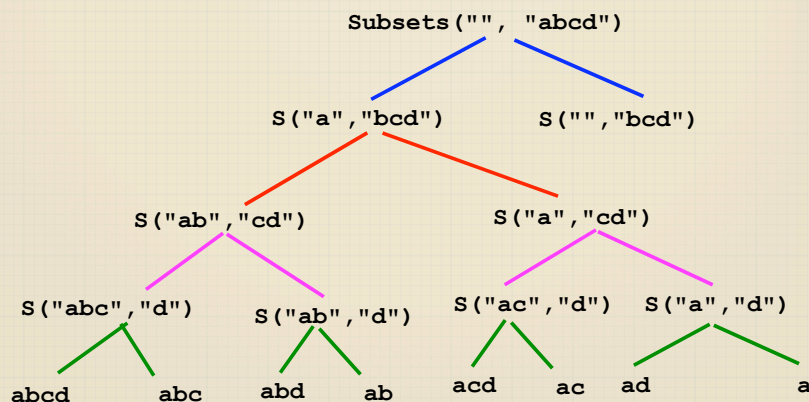 - Same patterns often resurface!

Subset strategy

◇ Result is empty, starting input is "abcd"
◇ Consider first element: "a"
◇ Add to subset, remaining input is "bcd"
◇ Recursively find all subsets from here
◇ Repeat recursion without including "a"

Subsets code

```
void RecSubsets(string soFar, string rest)
{
    if (rest == "")
        cout << soFar << endl;
    else {
            // add to subset, remove from rest, recur
      RecSubsets(soFar + rest[0], rest.substr(1));
            // don't add to subset, remove from rest,recur
      RecSubsets(soFar, rest.substr(1));
    }
}


void ListSubsets(string str)
{
    RecSubsets("", str);
}
```

Tree of recursive calls

```
              Subsets("", "abcd")

        S("a","bcd")          S("","bcd")

    S("ab","cd")         S("a","cd")

 S("abc","d")  S("ab","d")    S("ac","d")  S("a","d")

 abcd    abc    abd    ab    acd    ac   ad        a
```

Exhaustive recursion

◇ Permutations/subsets are about *choice*
  • Both have deep/wide tree of recursive calls
  • Depth represents total number of decisions made
  • Width of branching represents number of available options per decision
◇ Exhaustive recursion is, well, exhaustive
  • Explores every possible option at every decision point
  • Typically very expensive
    • N! permutations, 2^N subsets
    • (Recursion isn't the problem, there just is a huge space to explore)
◇ Consider partial exploration of exhaustive space
  • Similar exhaustive structure, but stop at first "satisfactory" outcome

Recursive backtracking

◇ Cast problem in terms of decision points
  - Identify what decisions need to be made
  - Identify what options are available for each decision
  - A recursive call makes one decision, and recurs on remaining decisions

◇ Backtracking approach
  - Design recursion function to return success/failure
  - At each call, choose one option and go with it
  - Recursively proceed and see what happens
  - If it works out, great, otherwise unmake choice and try again
  - If no option worked, return fail result which triggers backtracking (i.e. un-making earlier decisions)

◇ Heuristics may help efficiency
  - Eliminate dead ends early by pruning
  - Pursue most likely choice(s) first

Backtracking pseudocode

```
bool Solve(configuration conf)
{
    if (no more choices)      // BASE CASE
        return (conf is goal state);

    for (all available choices) {
        try one choice c;
        // solve from here, if works out, you're done
        if (Solve(conf with choice c made)) return true;
        unmake choice c;
    }

    return false;    // tried all choices, no soln found
}
```

Permute -> anagram finder

```
void RecPermute(string soFar, string rest)
{
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            RecPermute(soFar+rest[i], rest.substr(0,i)+rest.substr(i+1));
        }
    }
}


bool IsAnagram(string soFar, string rest, Lexicon &lex)
{
    if (rest == "") {
        return lex.containsWord(soFar);
    } else {
        for (int i = 0; i < rest.length(); i++) {
            if (IsAnagram(soFar+rest[i], rest.substr(0,i)+rest.substr(i+1), lex))
                return true;
        }
    }
    return false;
}
```

8 Queens

◇ Goal: place N queens on board so none threatened
  - Queen can attack in any straight line (horizontally, vertically, diagonally)

◇ Cast as in terms of decision
  - Each call will make one decision and recur on rest
  - How many decisions do you have to make?
  - What options do you have for each?
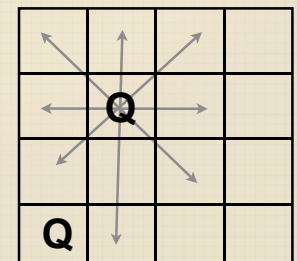
N queens code

```
bool Solve(Grid<bool> &board, int col)
{
    if (col >= board.numCols()) return true; // base case

    for (int rowToTry=0; rowToTry<board.numRows(); rowToTry++) {
        if (IsSafe(board,rowToTry,col)) {
            PlaceQueen(board, rowToTry, col);
            if (Solve(board, col + 1)) return true;
            RemoveQueen(board, rowToTry, col);
        }
    }
    return false;
}
```