# Admin

- Assign 2 due Wed
- Today's topics
  - Functional recursion
- Reading
  - Reader ch. 4-5-6 (today-W-F)

Lecture #8

# Solving problems recursively

- A recursive function calls itself — wacky!
- Idea: solve problem using coworkers (clones) who work and act like you
  - Delegate similar, smaller problem to clone
  - Combine result from clone(s) to solve total problem
  - Work toward trivial version that is directly solvable
- For problems that exhibit "self-similarity"
  - Structure repeats within at different levels of scale
  - Solving larger problem means solving smaller problem(s) within
- Feels mysterious at first
  - "Leap of faith" required
  - With practice, master the art of recursive decomposition
  - Eventually grok the underlying patterns

# Functional recursion

- Function that returns answer/result
  - Outer problem result uses result from smaller, same problem(s)
- Base case
  - Simplest version of problem
  - Can be directly solved
- Recursive case
  - Make call(s) to self to get results for smaller, simpler version(s)
  - Recursive calls must advance toward base case
  - Results of recursive calls combined to solve larger version

# Power example

- C++ has no exponentiation op
- Iterative formulation for Raise function
  - $base^{exp}$ = base * base * .... * base (exp times)

```
int Raise(int base, int exp)
{
   int result = 1;
   for (int i = 0; i < exp; i++)
      result *= base;
   return result;
}
```

# Recursive version

- Now consider recursive formulation
  - $base^{exp} = base * base^{exp-1}$

```
int Raise(int base, int exp)
{
    if (exp == 0)        } Base case
        return 1;
    else
        return base * Raise(base, exp-1);   } Recursive case
}
```

# More efficient recursion

$base^{exp} = base^{exp/2} * base^{exp/2}$ (* base if exp is odd)

```
int Raise(int base, int exp)
{
  if (exp == 0)
    return 1;
  else {
    int half = Raise(base, exp/2);
    if (exp % 2 == 0)
      return half * half;
    else
     return base * half * half;
  }
}
```

# Avoid "arm's length" recursion

- Aim for simple, clean base case
  - No need to anticipate other earlier stopping points
  - Avoid looking ahead before recursive calls, just let simple base case handle

```
int Raise(int base, int exp)
{
  if (exp == 0) return 1;
  else if (exp == 1) return base;
  else if (exp == 2) return base * base;
  else if (exp == 3) return base * base * base;
  else return base * Raise(base, exp - 1);
}
```

# Recursion and efficiency

- Recursion provides no guarantee of (in)efficiency
  - Recursion can require same resources as alternative approach
    - Or recursion may be much more or much less efficient
  - For problems with simple iterative solution, iteration is likely the best
- Why recursion then?
  - Can express with clear, direct, elegant code
  - Can intuitively model a task that is recursive in nature
  - Solution may require recursion — iteration won't do!

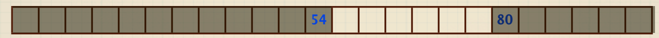
# Palindromes

- A palindrome string reads same when reversed
  - e.g. "was it a car or a cat i saw", "go hang a salami im a lasagna hog"
- Recursive insight
  - First and last letter match and interior is palindrome
- Base case?

```
bool IsPalindrome(string s)
{
   if (s.length() <= 1) return true;
   return s[0] == s[s.length()-1] &&
          IsPalindrome(s.substr(1, s.length()-2));
}
```

# Binary search

- Searching for key within vector
  - *Linear search* starts at beginning and searches to end
  - *Binary search* uses divide-and-conquer (requires **sorted** vector)
    - Much faster method!
- Recursive insight:
  - Consider middle elem of vector, if key, you're done
  - Otherwise decide which half to recursively search




- Base case?

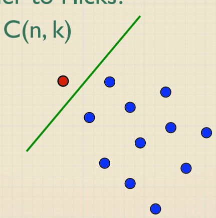
# Binary search code

```
const int NotFound = -1;

int BSearch(Vector<string> &v,
             int start, int stop, string key)
{
    if (start > stop) return NotFound;

   int mid = (start + stop)/2;
   if (key == v[mid])
       return mid;
   else if (key < v[mid])
       return BSearch(v, start, mid-1, key);
   else
       return BSearch(v, mid+1, stop, key);
}
```

- Classic "divide and conquer" algorithm
  - Operates very efficiently! Double size of vector, how much longer to search?

# Choosing a subset

- Reader ch 4, exercise 8
  - Given N things, how many different ways can you choose K of them?
    - e.g. given a dorm of 60 people, how many different groups of 4 people can go together to Flicks?
  - N-choose-K, written as C(n, k)



Number of subsets that include ● = C(n-1, k-1)

\+ Number of subsets that don't include ● = C(n-1, k)

# Choose code

- Simplest base case
  - when **no** choices remain at all

```
int C(int n, int k)
{
  if (k == 0 || k == n)
    return 1;
  else
    return C(n-1, k) + C(n-1, k-1);
}
```