Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

MCS 360 Lecture 22
Introduction to Data Structures
Jan Verschelde, 4 March 2020

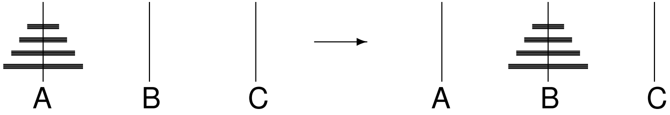
Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

The Towers of Hanoi

an ancient mathematical puzzle

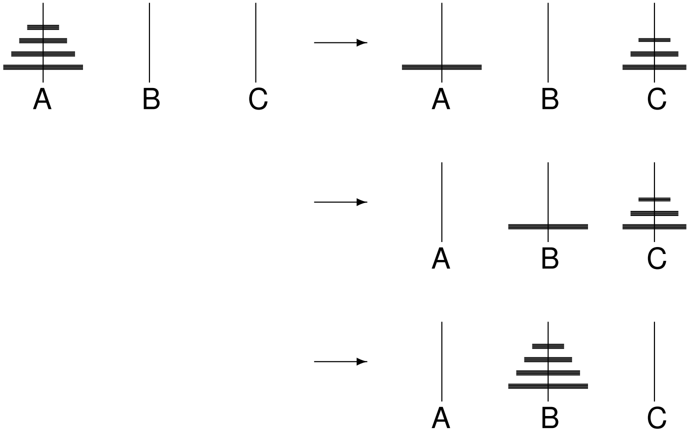
Input: disks on a pile, all of varying size,
no larger disk sits above a smaller disk,
and two other empty piles.



Task: move the disks from the first pile to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one,
you may use the third pile as buffer.

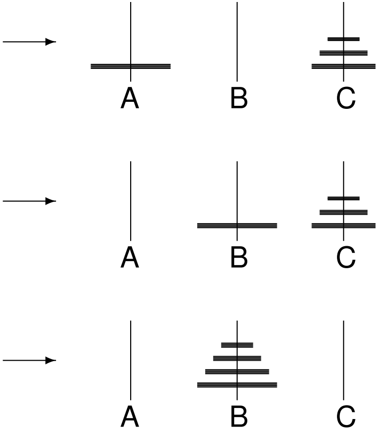
a recursive solution

Assume we know how to move a stack with one disk less.



a recursive algorithm

Base case: move one disk from A to B.
To move n disks from A to B:



Move $n - 1$ disks from A to C
using B as auxiliary pile

Move n-th disk from A to B

Move $n - 1$ disks from C to B
using A as auxiliary pile

Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

the main program

```
int main()
{
   cout << "give number of disks : ";
   int n; cin >> n;

   stack<int> A;
   for(int i=0; i<n; i++) A.push(n-i);
   cout << "stack A : "; write(A);
   cout << endl;

   stack<int> B,C;
   hanoi(n,A,B,C);
   write(A,B,C);

   return 0;
}
```

running the program

```
give number of disks : 4
stack A :  1 2 3 4
  A :  2 3 4  B :  1  C :
  A :  3 4  B :  2  C :  1
  A :   B :  1 2  C :  3 4
  A :  4  B :  3  C :  1 2
  A :  2  B :  1 4  C :  3
  A :   B :  2 3  C :  1 4
  A :  4  B :  1 2 3  C :
  A :   B :  4  C :  1 2 3
  A :  2 3  B :  1 4  C :
  A :  3  B :  2  C :  1 4
  A :  4  B :  1 2  C :  3
  A :   B :  3 4  C :  1 2
  A :  2  B :  1  C :  3 4
  A :   B :  2 3 4  C :  1
  A :   B :  1 2 3 4  C :
```

writing stacks

```
void write ( stack<int> s )
{
   stack<int> t;

   for(t=s; !t.empty(); t.pop())
      cout << " " << t.top();
}

void write ( stack<int> A, stack<int> B,
             stack<int> C )
{
   cout << " A : "; write(A);
   cout << " B : "; write(B);
   cout << " C : "; write(C); cout << endl;
}
```

a recursive solution in C++

```
void hanoi ( int n, stack<int> &A, stack<int> &B,
                    stack<int> &C )
{
   if(n==1) // move disk from A to B
   {
      B.push(A.top()); A.pop(); write(A,B,C);
   }
   else
   {
// move n-1 disks from A to C, B is auxiliary
      hanoi(n-1,A,C,B);
// move nth disk from A to B
      B.push(A.top()); A.pop(); write(A,B,C);
// move n-1 disks from C to B, A is auxiliary
      hanoi(n-1,C,B,A);
   }
}
```

analysis of the solution

algorithm Do we have an algorithm?

1. termination: n decreases in each call;
2. correctness: proof by induction.

C++ Use of STL stack, call by reference.

cost What is the number of moves?

Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code

2. Binary Search
   - sorted vectors of numbers
   - fast power function

3. the Fibonacci numbers
   - when not to use recursion
   - memoization

an example of binary search

Looking for 70 in *a sorted sequence*

```
2 3 15 39 42 48 51 58 59 64 69 73 76 81 96 97

                      59 64 69 73 76 81 96 97

                      59 64 69

                            69
```

70 does not occur

cost = # levels in tree $= \log_2(n) \Rightarrow O(\log_2(n))$

the main program

```
int search ( vector<int> v, int a, int b, int e );
// returns k, with a <= k <= b if v[k] == e,
// otherwise returns -1

int main()
{
   cout << "give n : ";
   int n; cin >> n;

   vector<int> v = generate(n);
   cout << "v : "; write(v); cout << endl;
   sort(v.begin(),v.end());
   cout << "v : "; write(v); cout << endl;

   cout << "give a number : ";
   int e; cin >> e;
   int k = search(v,0,v.size()-1,e);
```

using STL algorithms

The binary search is available for STL containers:

```
bool answer = binary_search(v.begin(),v.end(),e);
```

The `binary_search` returns true if `e` occurs.

```
    vector<int>::iterator i;
    i = lower_bound(v.begin(),v.end(),e);
    int d = i - v.begin();
    if(v[d] == e)
       cout << e << " occurs at v[" << d << "]\n";
```

The index returned by `lower_bound` is the spot where `e` can be inserted while preserving the order.

recursive binary search

```
int search ( vector<int> v, int a, int b, int e )
{
   if(a == b)
      return (v[a] == e) ? a : -1;
   else
   {
      int m = (a+b)/2;

      if(v[m] == e)
         return m;
      else if(v[m] < e)
         return search(v,m+1,b,e);
      else
         return search(v,a,m-1,e);
   }
}
```

Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

a fast power function

Repeated squaring, for example:

$$a^4 = a \times a \times a \times a = (a^2) \times (a^2) = (a^2)^2.$$

No gain with 4, but for 8: $a^8 = (a^4)^2 = ((a^2)^2)^2$.

Cost?

$$\begin{array}{rcl} ((a^2)^2)^2 & = & b^2 = b \times b, b = (a^2)^2 \\ (a^2)^2 & = & c^2 = c \times c, c = a^2 \\ a^2 & = & a \times a. \end{array}$$

We count three $\times$, observe $3 = \log_2(8)$.

In general: compute a^n with $O(log_2(n))$ multiplications.

a recursive function

```
double recursive_power ( double a, int n)
{
   if(n==0)
      return 1.0;
   else if(n==1)
      return a;
   else {
      int d = n / 2;
      int r = n % 2;
      double y = recursive_power(a,d);
      y = y*y;
      if(r==0)
         return y;
      else {
         double z = recursive_power(a,r);
         return y*z;
      }
   }
}
```

Binary Search

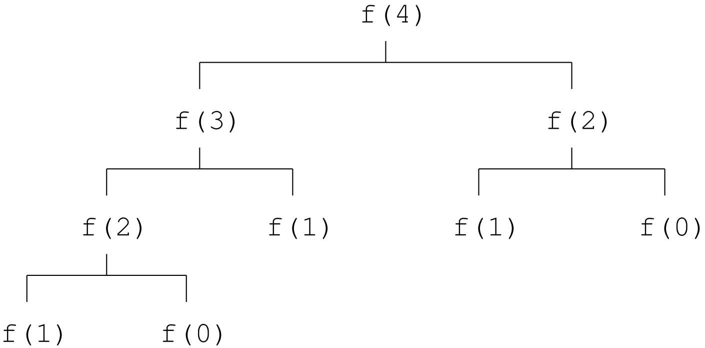
1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

the Fibonacci numbers

$f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$, $n > 1$

```
long long int f ( int n )
{
   if(n == 0)
      return 0;
   else if(n == 1)
      return 1;
   else
      return f(n-1) + f(n-2);
}
```

exponentional #calls



Let c_n = #calls to compute $f(n)$.
$c_2 = 2$, $c_3 = 4 = 2^2$, $c_4 = 8 = 2^3$
recursion: $c_n = c_{n-1} + c_{n-2} + 2 = \cdots$ is $O(2^n)$

Binary Search

1. Recursive Problem Solving
   - the towers of Hanoi
   - recursive C++ code
2. Binary Search
   - sorted vectors of numbers
   - fast power function
3. the Fibonacci numbers
   - when not to use recursion
   - memoization

memoization

We can store the results of the function calls.
Before execution of definition, lookup table.

```
long long int mf ( int n )
{
   static vector<long long int> values;

   while(values.size() <= n) values.push_back(-1);

   // code to be continued ...
```

Declaring `v` as `static` implies: `values` persists after `mf` returns.

using memoization

The code continues below

```
    if(values[n] != -1)
       return values[n];    // return value in vector
    else
    {                       // we store in values[n]
       if(n==0)
          values[n] = 0;
       else if(n==1)
          values[n] = 1;
       else
          values[n] = mf(n-1) + mf(n-2);
       return values[n];
    }
}
```

Benefit of memoization: same control logic as in recursion.

Summary + Exercises

Covered more of Chapter 7 on recursion.

Exercises:

1. Modify the program to solve the towers of Hanoi so we see the number of each move when writing stacks.
2. Write a function `trace_recursive_power` to trace the recursive function calls in the recursive algorithm to compute a^n. Refer to the code to justify why the number of multiplications is $O(\log_2(n))$.
3. Adjust the original recursive definition of `f` to compute the Fibonacci numbers to count the number of function calls. Make a table for growing values of `n`.