Lecture 7: Function Calls and Recursion

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

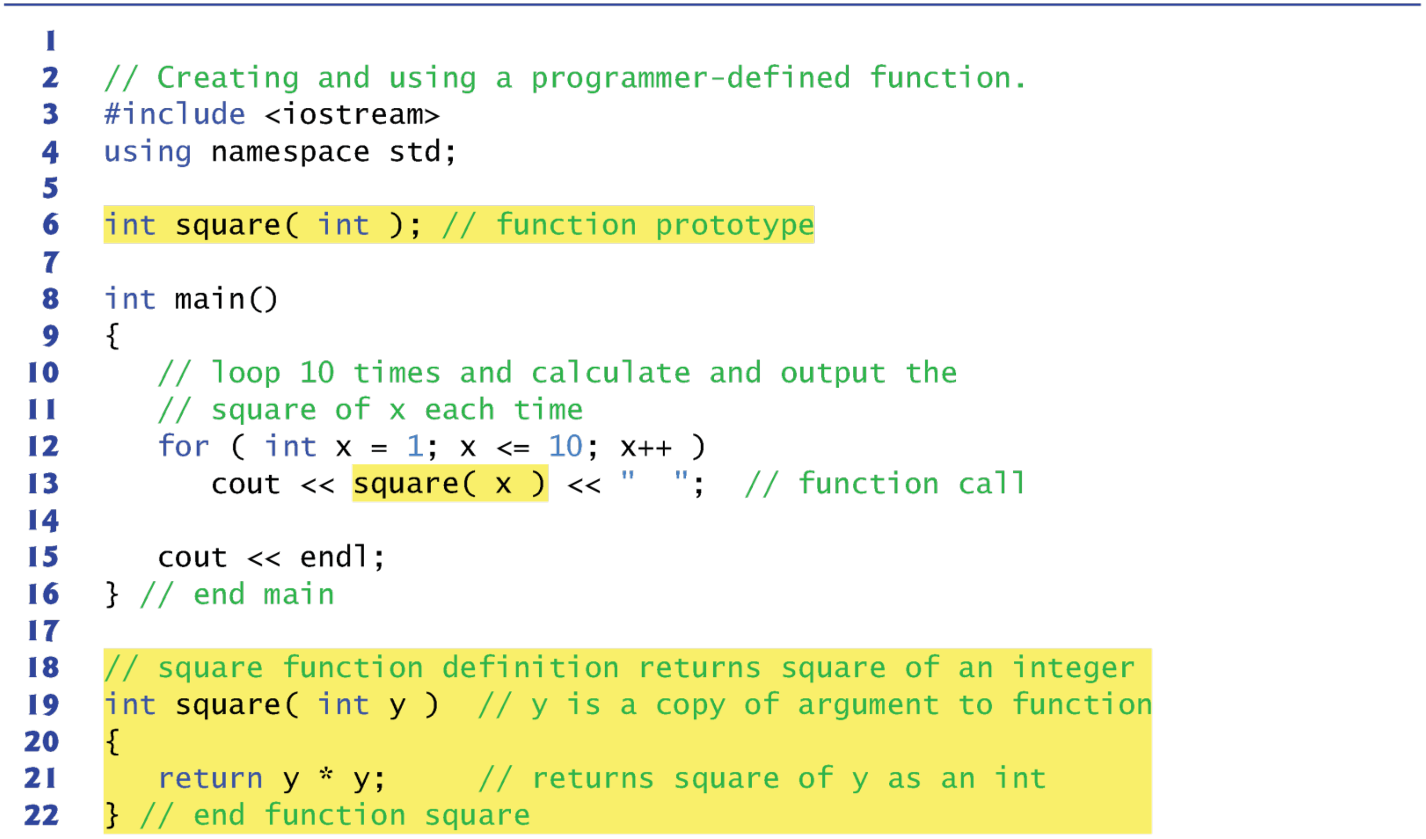
Recap: Function Definition

- ❑ **The format of a function definition is as follows:**

```
return-value-type  function-name( parameter-list )
{
    declarations and statements
}
```

- ❑ **The *function-name* is any valid identifier.**
- ❑ **The *return-value-type* is the data type of the returned result to the caller.**
  - ❑ The type `void` indicates that a function does not return a value.
- ❑ **All variables defined in a function are local variables—they're known only in the function in which they're defined.**
- ❑ **Most functions have a list of parameters that provide the means for communicating information between functions.**
  - ❑ A function's parameters are also local variables of that function.

Example

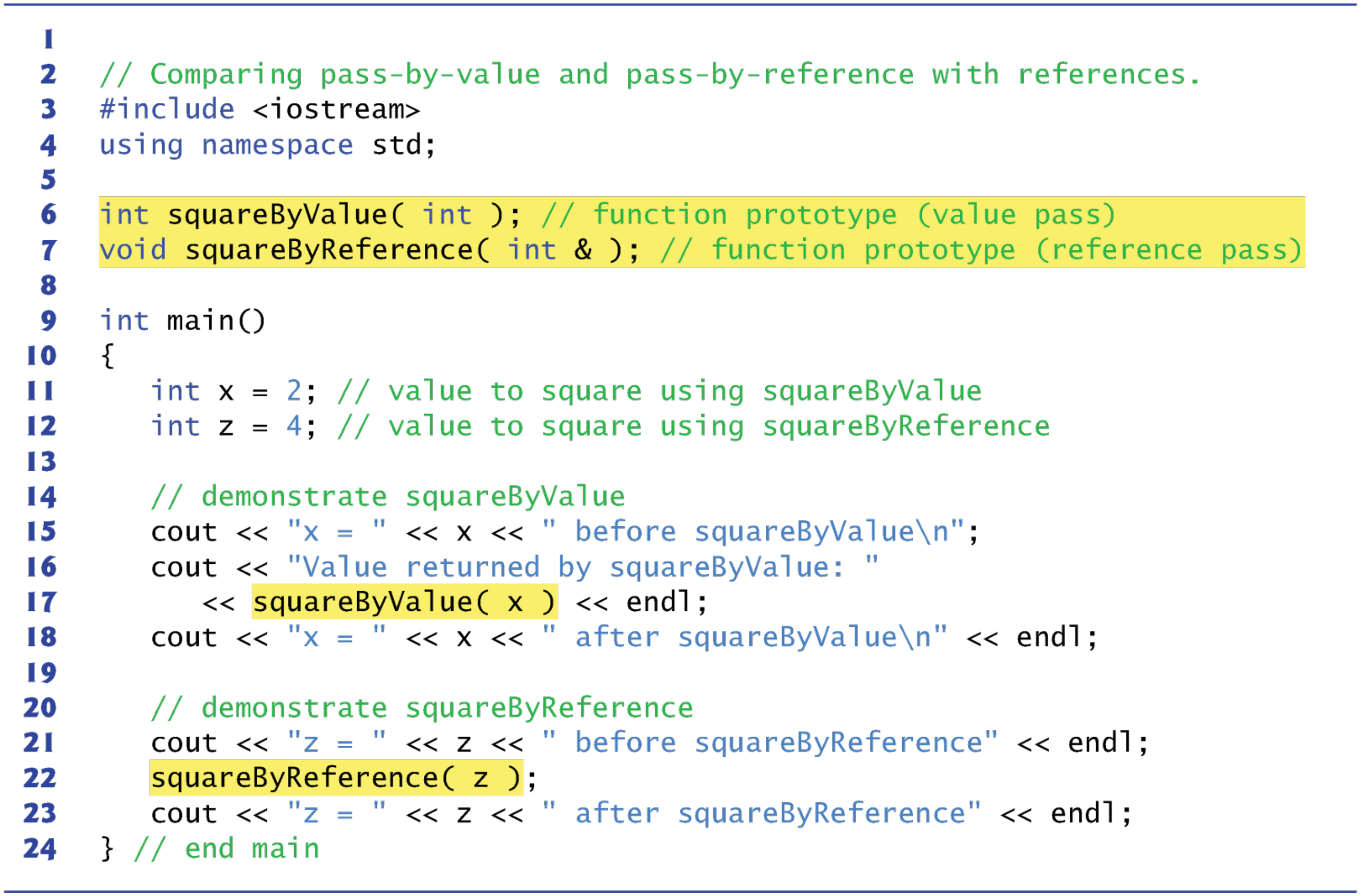


```cpp

// Creating and using a programmer-defined function.
#include <iostream>
using namespace std;

int square( int ); // function prototype

int main()
{
   // loop 10 times and calculate and output the
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;
} // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int
} // end function square
```

Call by Value vs Call by Reference

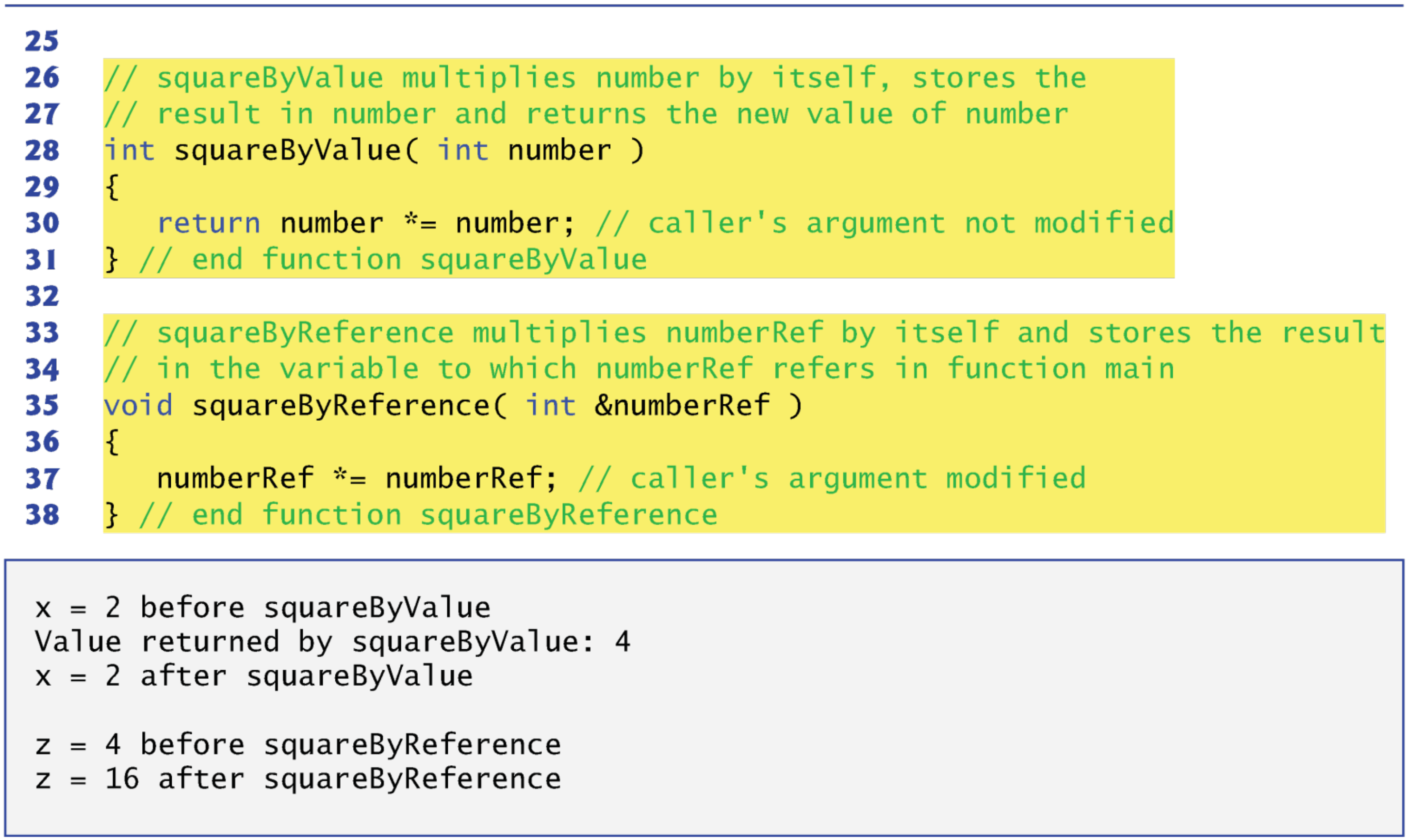
- ❑ **Call by value**
  - ❑ copies the actual **value** of an argument into the formal parameter of the function
- ❑ **Call by reference**
  - ❑ copies the address of an argument into the formal parameter

Call by Value vs Call by Reference



```cpp

// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using namespace std;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
} // end main
```

Call by Value vs Call by Reference



```cpp

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Reference is ”Alias”

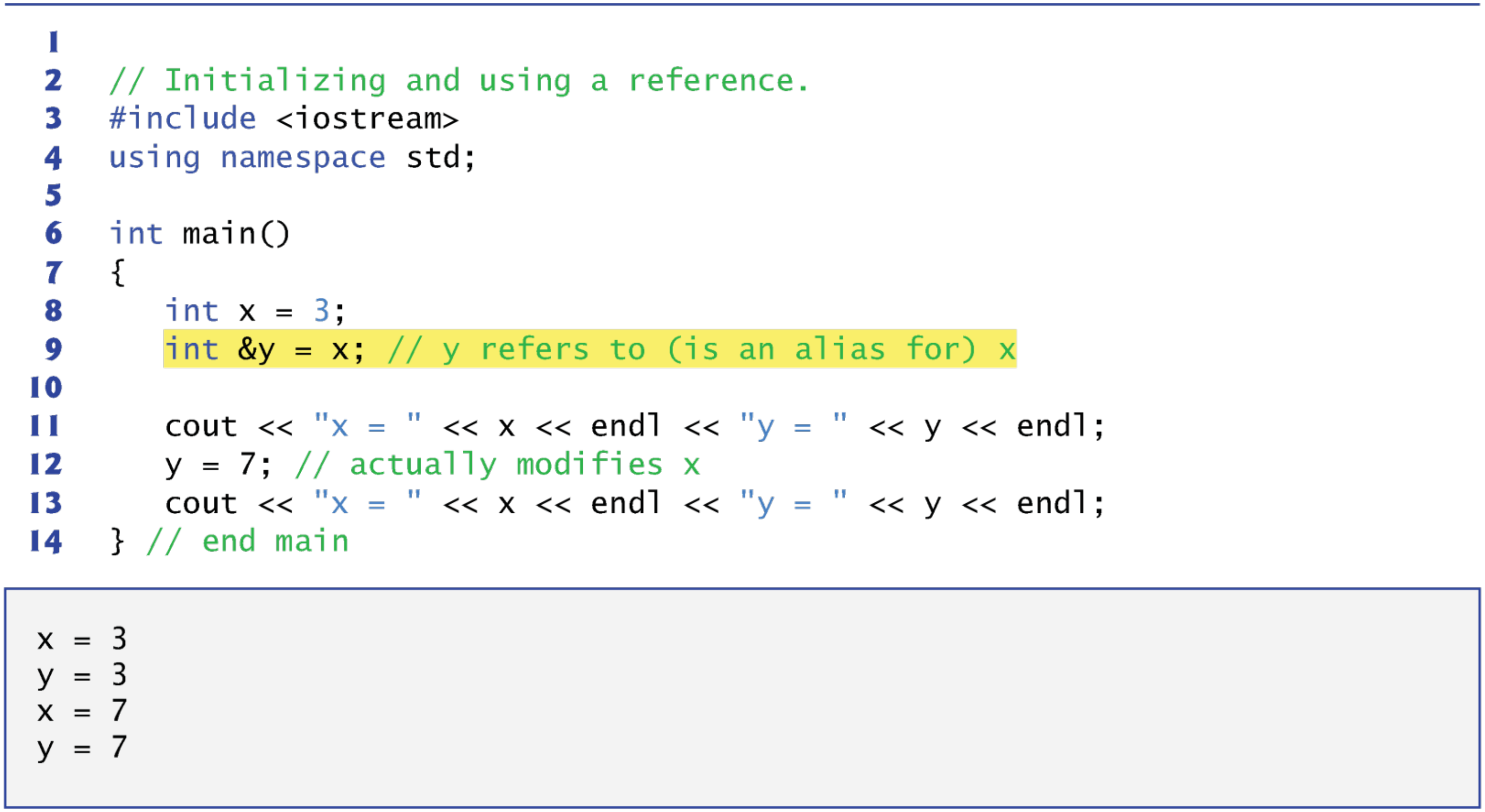
- **References can also be used as aliases for other variables within a function.**
- **For example, the code**

```
int count = 1; // declare integer variable
count
int &cRef = count; // create cRef as an alias
for count
cRef++; // increment count (using its alias
cRef)
```

increments variable `count` by using its alias `cRef`.

- **Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.**
- **Once a reference is declared as an alias for another variable, all operations performed on the alias are actually performed on the original variable.**

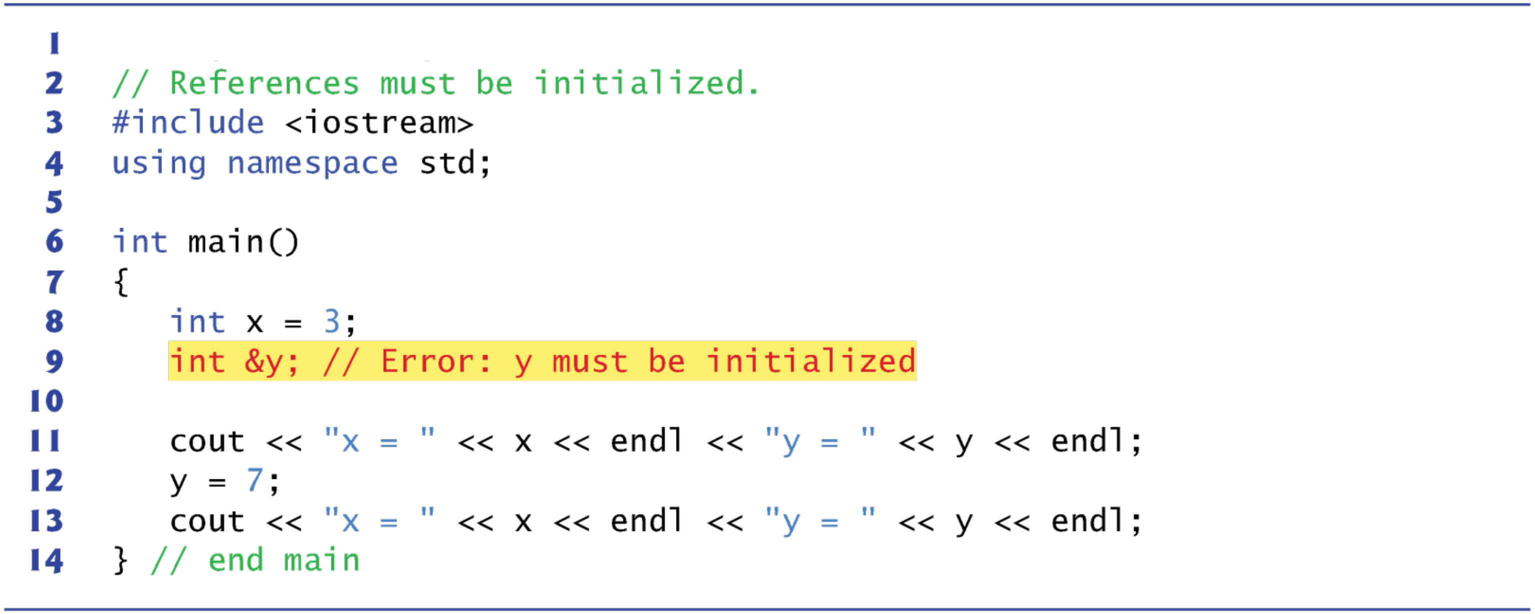
Reference is “Alias”



```

// Initializing and using a reference.
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    int &y = x; // y refers to (is an alias for) x

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7; // actually modifies x
    cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

Reference must be Initialized!



```

// References must be initialized.
#include <iostream>
using namespace std;

int main()
{
   int x = 3;
   int &y; // Error: y must be initialized

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

Recursive Function

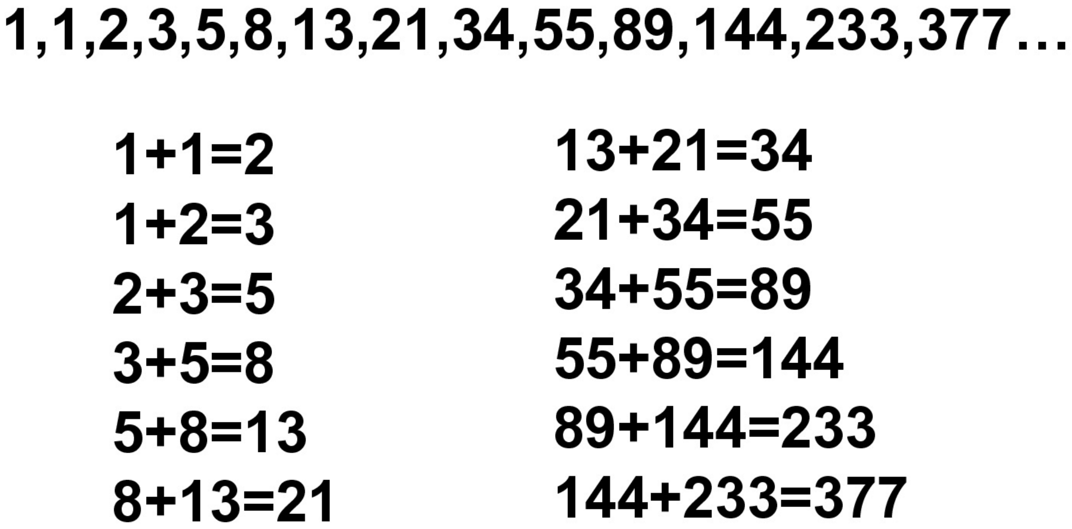
- ❑ **A recursive function is a function that calls itself, either directly, or indirectly (through another function).**
- ❑ **The function only knows how to solve the simplest case(s), or so-called base case(s).**
  - ❑ If the function is called with a base case, the function simply returns a result
  - ❑ For complex problem, the function divides a problem into
    - What it can do (base case) → return the result
    - What it cannot do → resemble the original problem, but be a slightly simpler or smaller version
    - The function calls a new copy of itself (recursion step) to solve the smaller problem
  - ❑ Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Fibonacci Number (iterative)

Write a function named ”fib" that takes a positive integer N and returns the N-th Fibonacci number

e.g. N=6, returns 8

e.g. N=9, returns 34



1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21

13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

Fibonacci Number (Recursive)

Write a function named ”fib" that takes a positive integer N and returns the N-th Fibonacci number

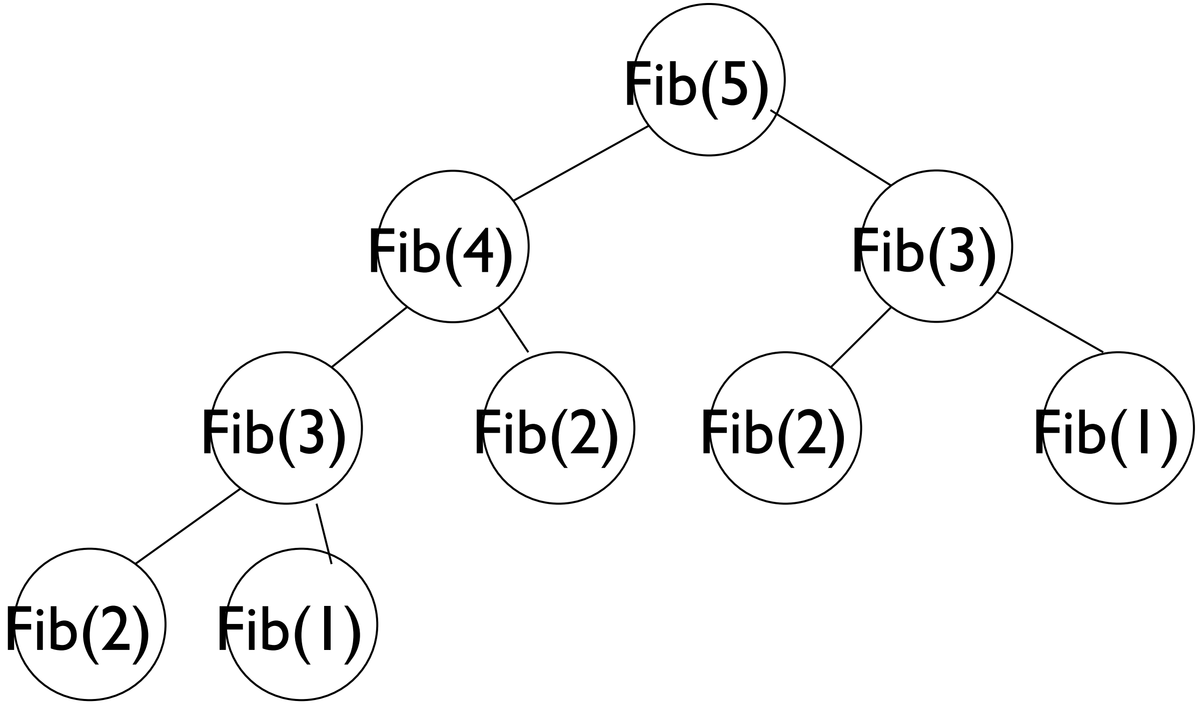
e.g. N=6, returns 8

e.g. N=9, returns 34

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

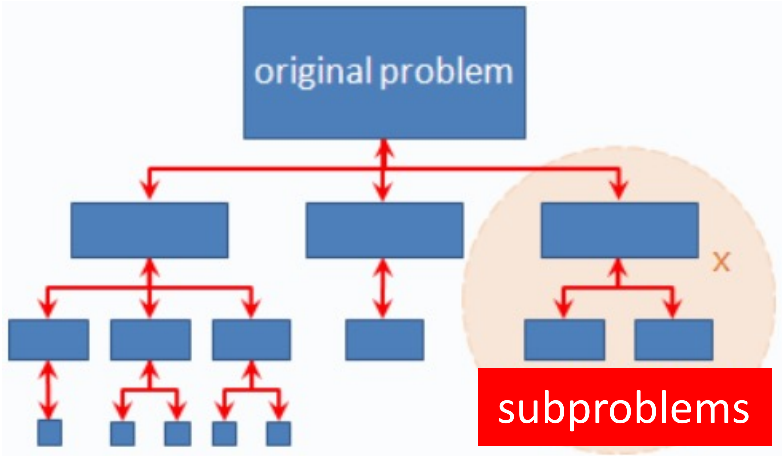
?

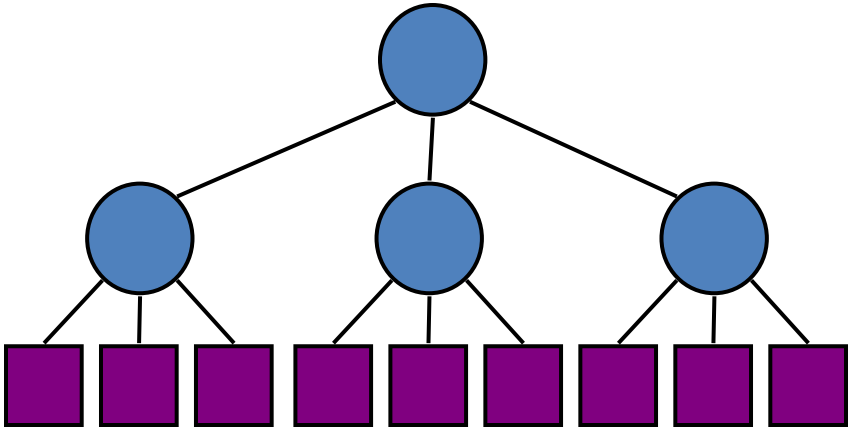
Duplicate Computations …



Divide and Conquer

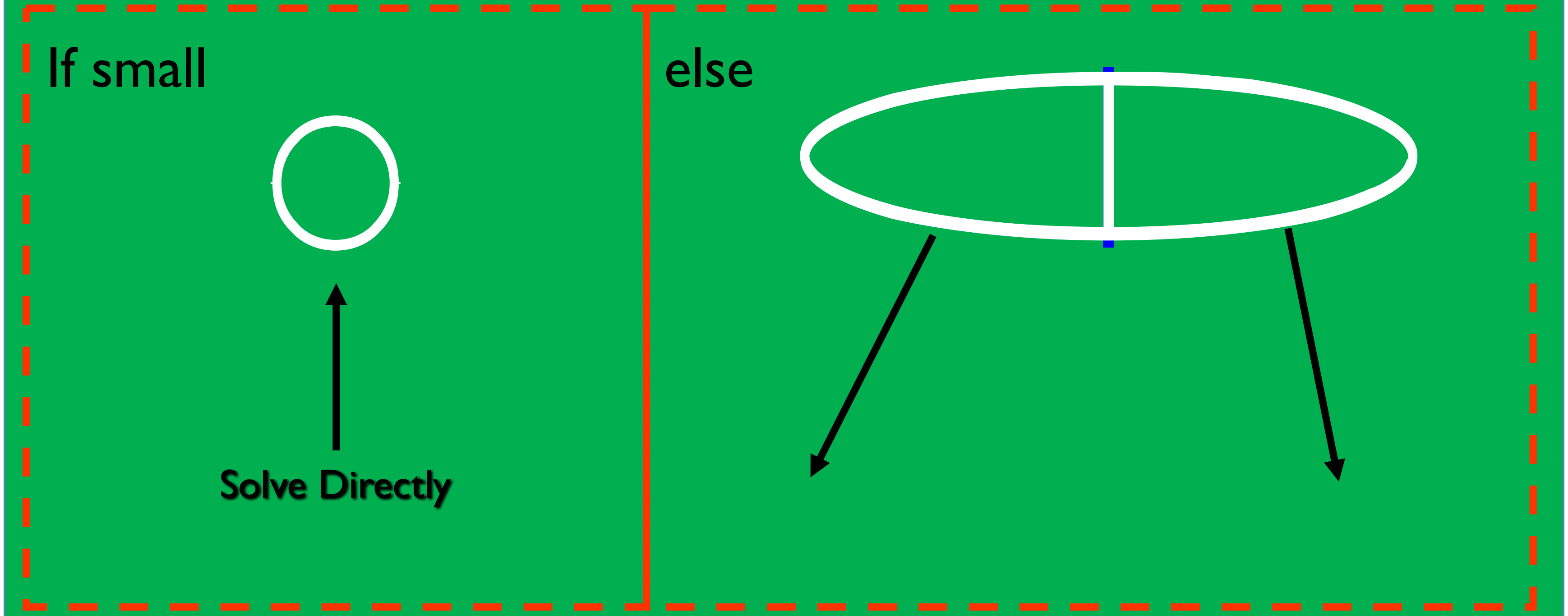
- □ **Divide & Conquer**
  - □ Divide: Divide the original problem into smaller subproblems
  - □ Recurse: Solve each small subproblem recursively
  - □ Conquer: Combine these subproblems all the way to the top
- □ **Illustration**





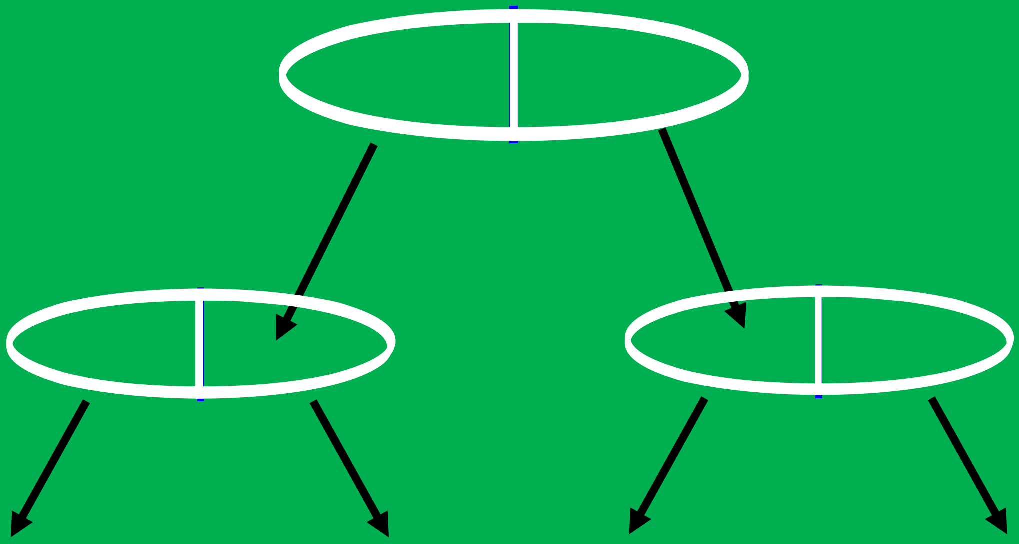
Visualization of Divide and Conquer

- Divide



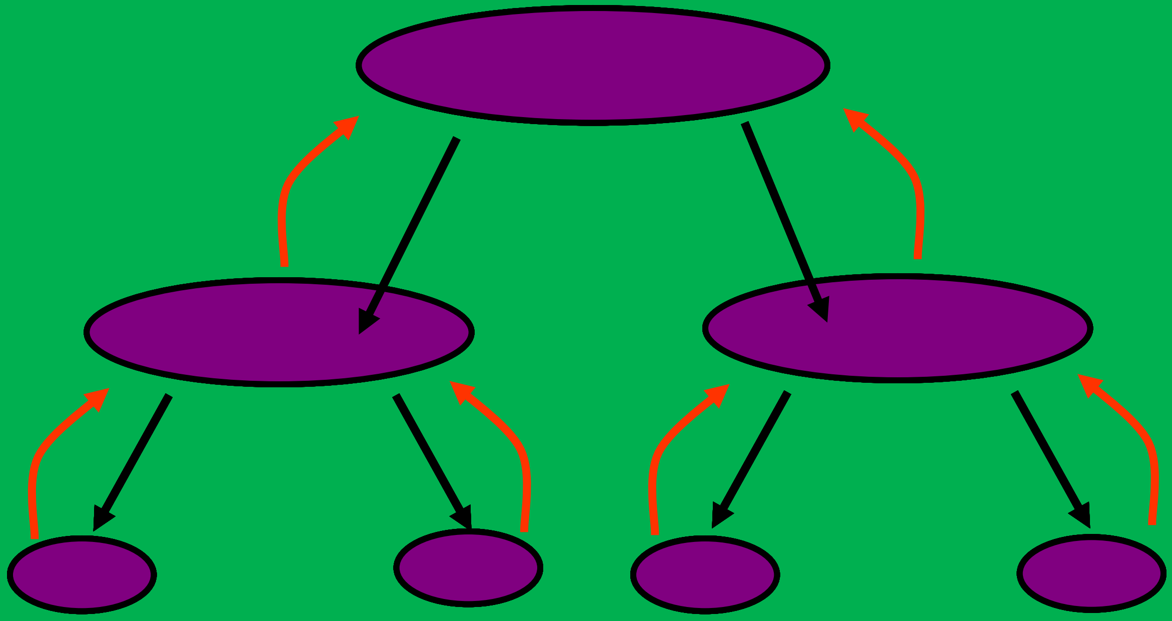
Visualization of Divide and Conquer

- ❑ Recursive



Visualization of Divide and Conquer

- ❑ Solve and Conquer



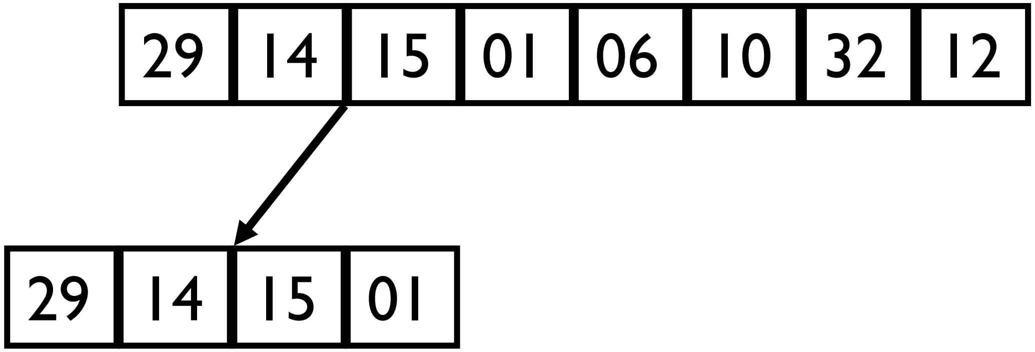
Example: Find the Maximum Value

- Divide

29	14	15	01	06	10	32	12

Example: Find the Maximum Value

❑ Divide

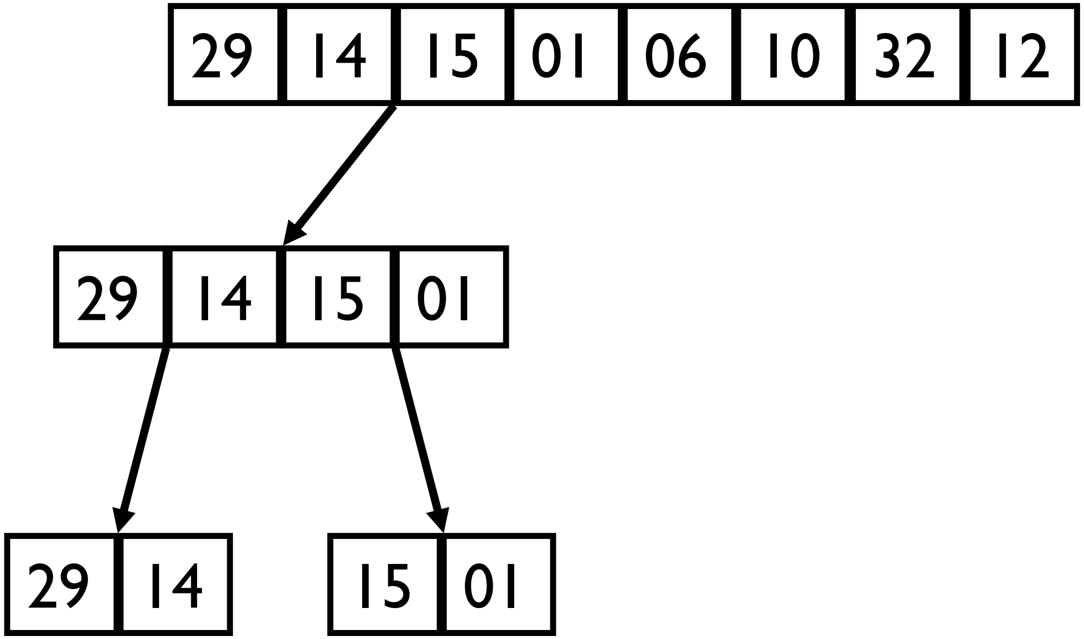


29	14	15	01	06	10	32	12

29	14	15	01

Example: Find the Maximum Value

❑ Divide



29 14 15 01 06 10 32 12

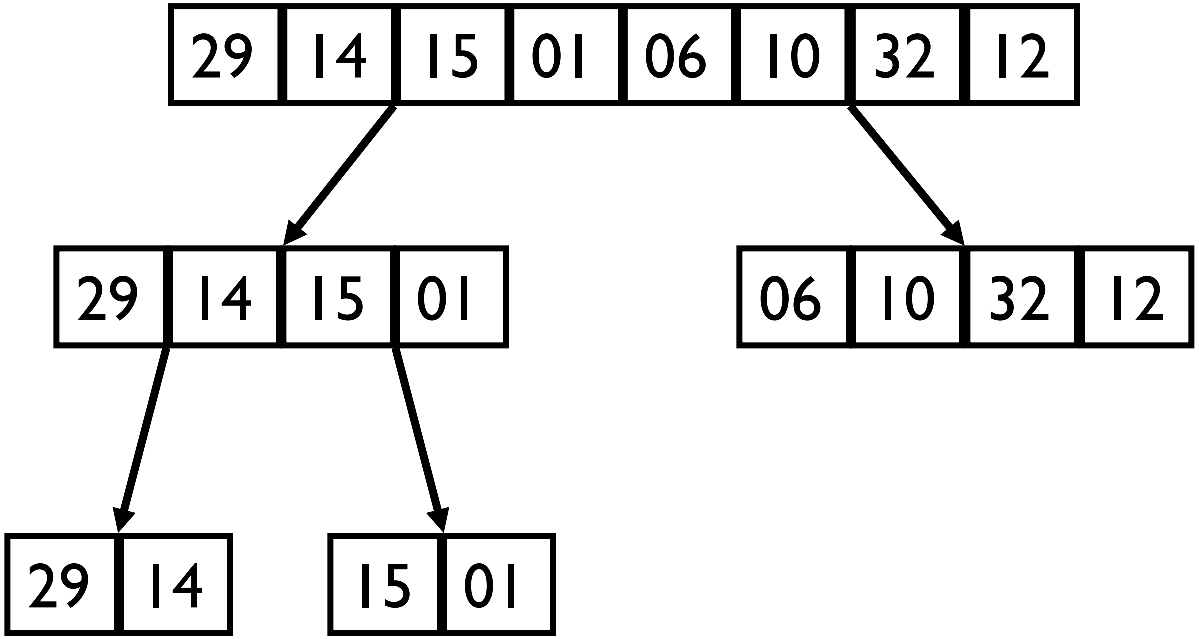
29 14 15 01

29 14

15 01

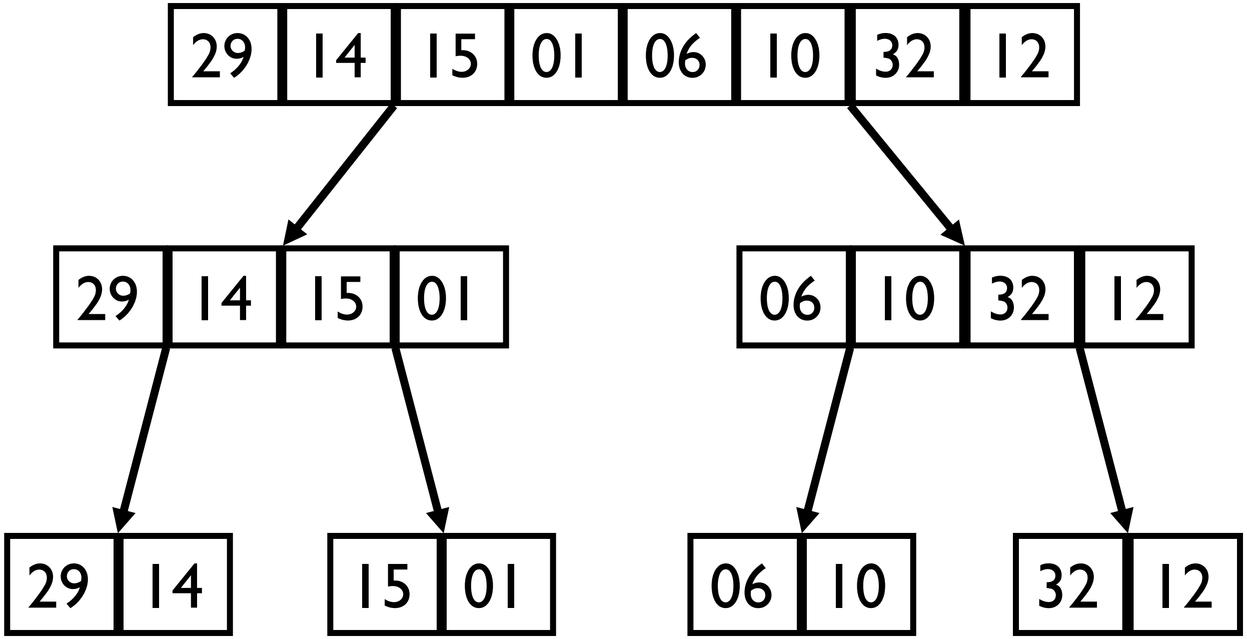
Example: Find the Maximum Value

- ❑ Divide



Example: Find the Maximum Value

❑ Divide



29 14 15 01 06 10 32 12

29 14 15 01

06 10 32 12

29 14

15 01

06 10

32 12

Example: Find the Maximum Value

- Conquer

29 14

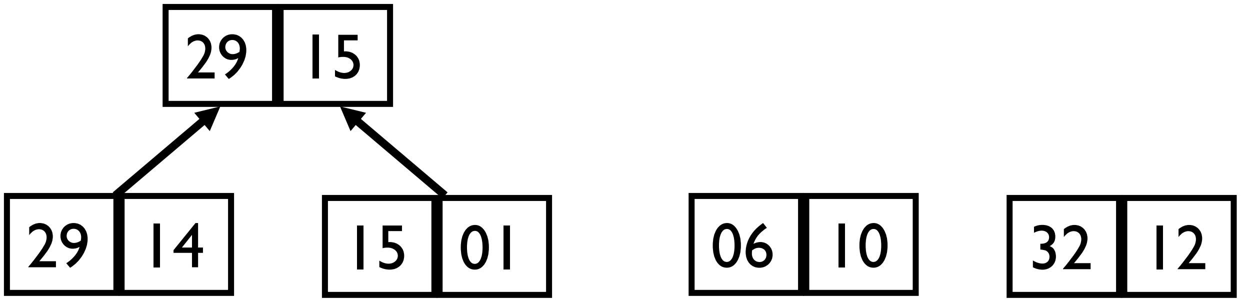
15 01

06 10

32 12

Example: Find the Maximum Value

- Conquer

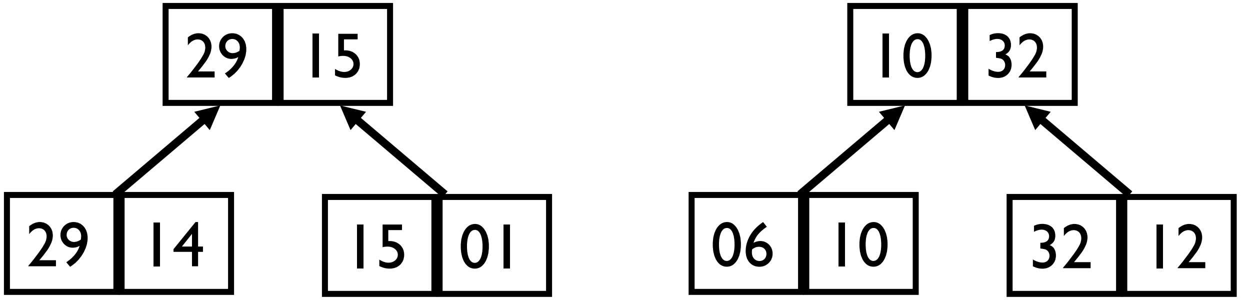


29 15

29 14   15 01   06 10   32 12

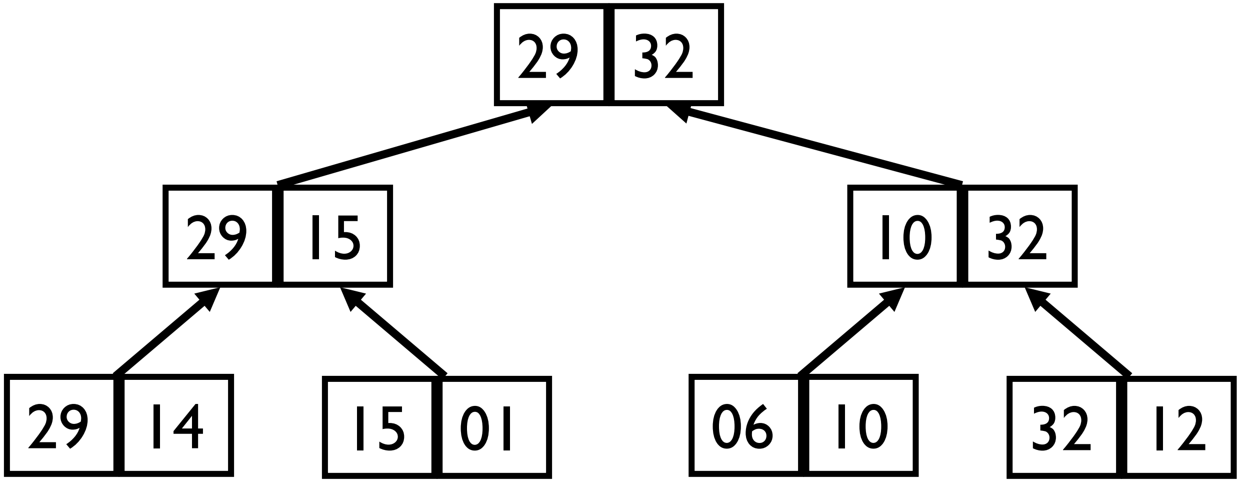
Example: Find the Maximum Value

- Conquer



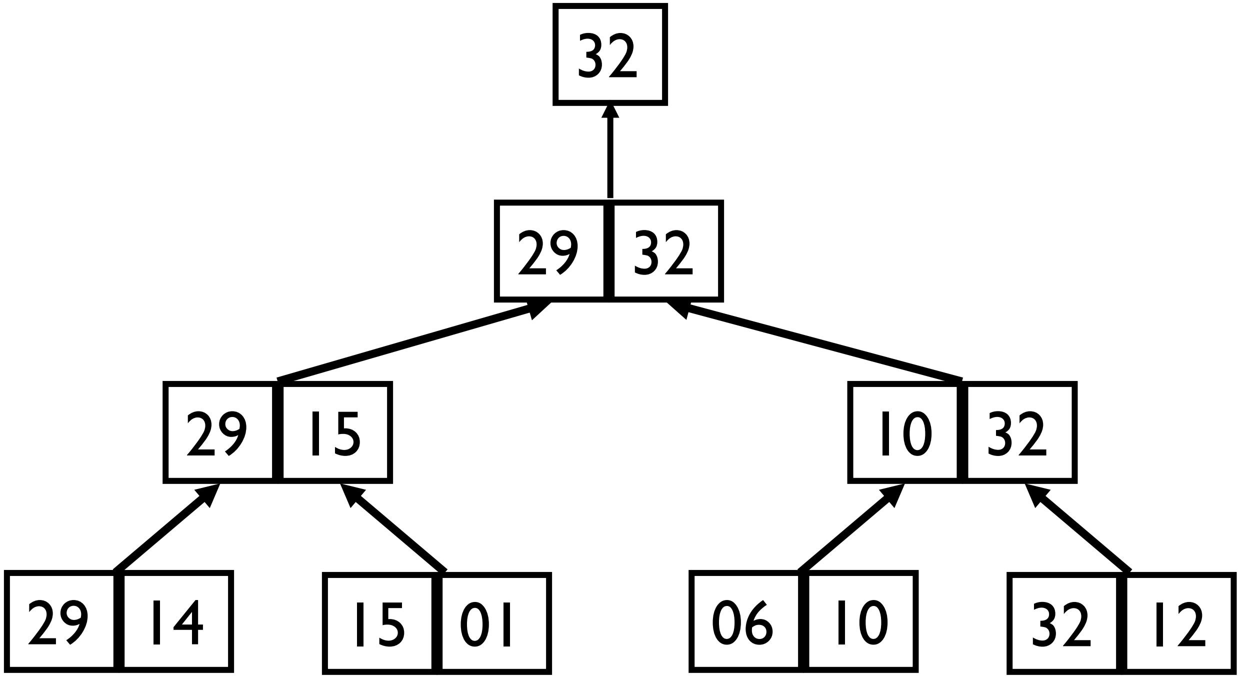
Example: Find the Maximum Value

- Conquer



Example: Find the Maximum Value

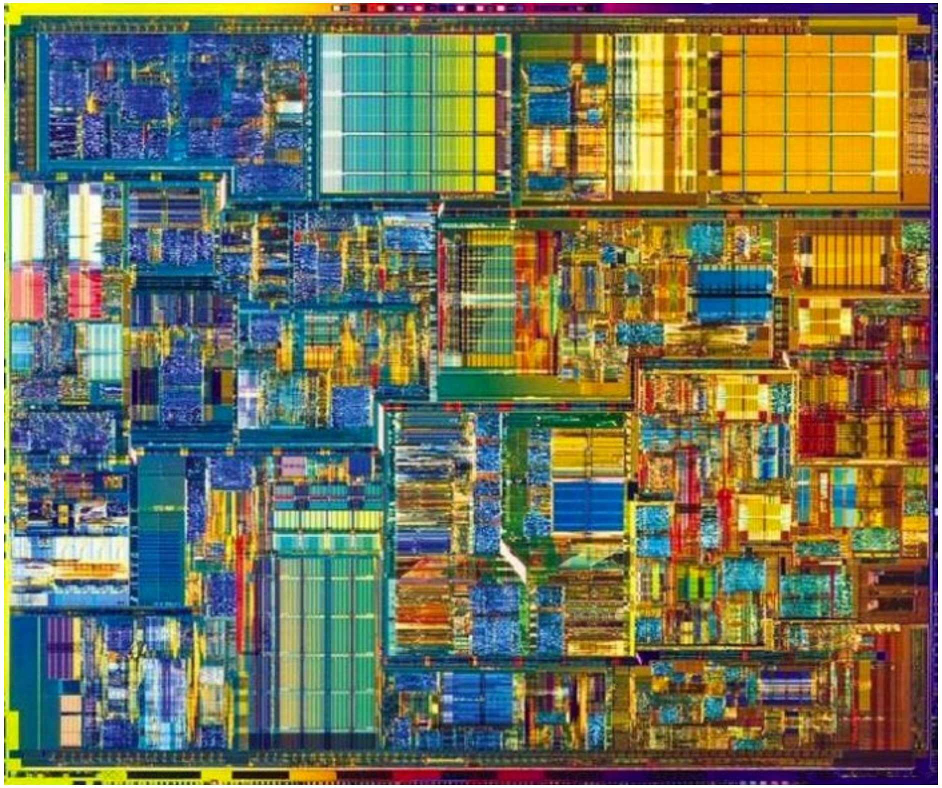
❑ Conquer



Divide and Conquer is Heavily used in CAD

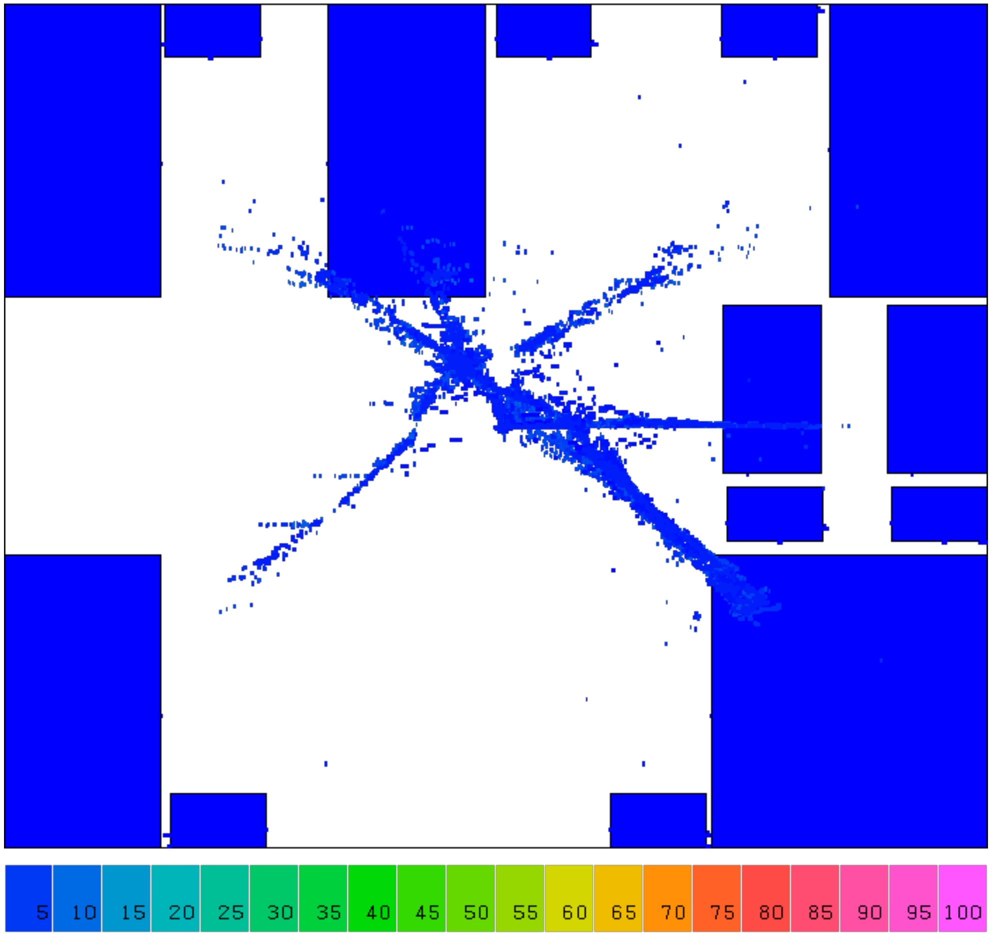
- **Modern circuits sizes are too large to handle in flat**

2000 – Intel Pentium4
42M transistors
1.5MHz; 224 mm^2

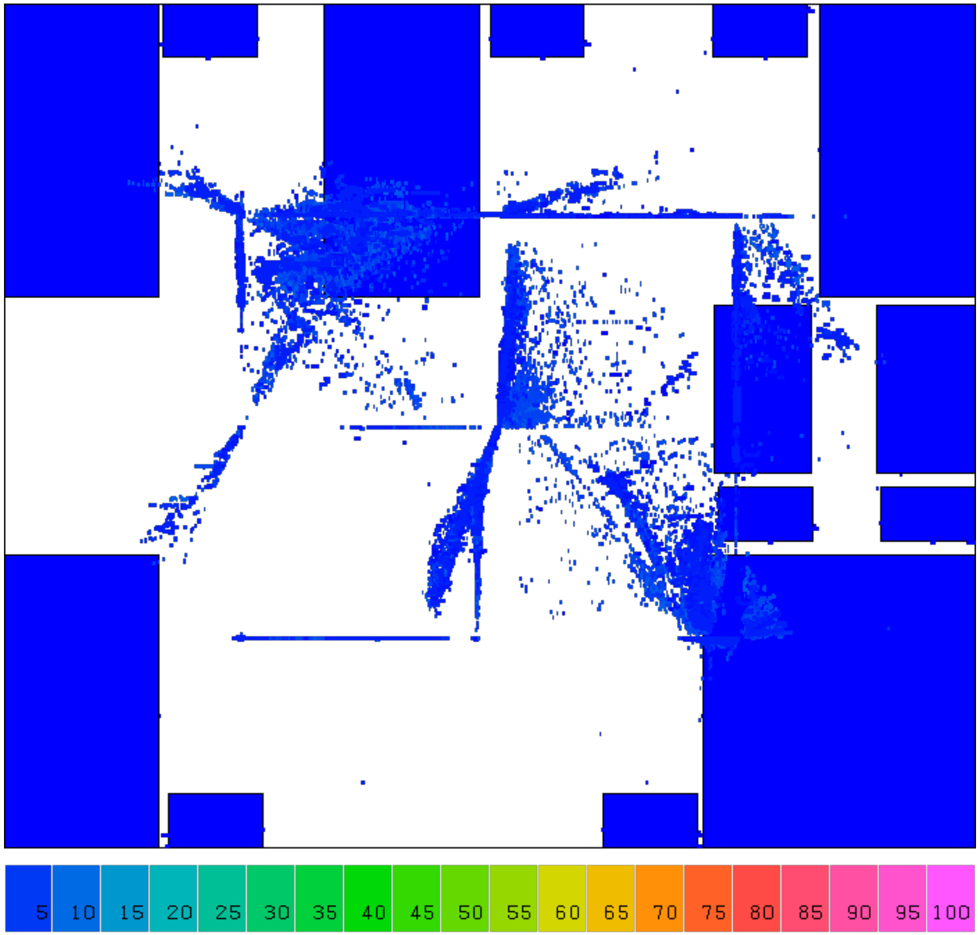


Directly solving the original problem takes forever to finish …

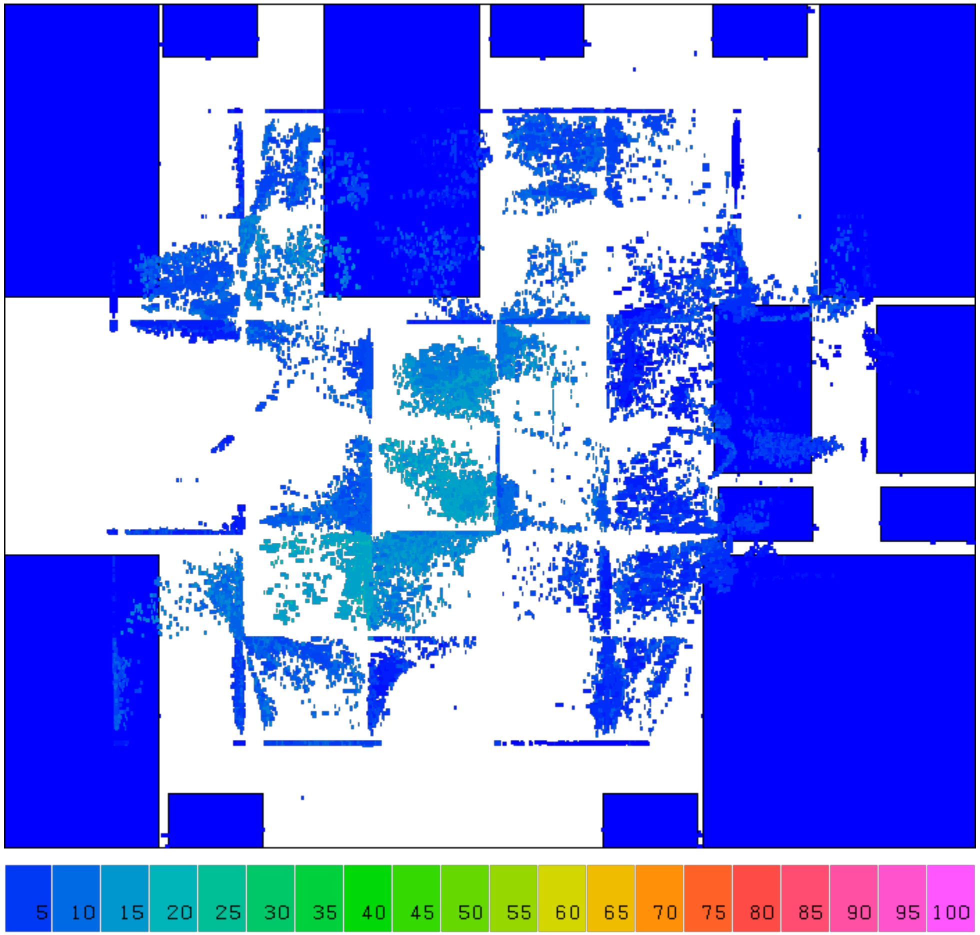
Example: Placement



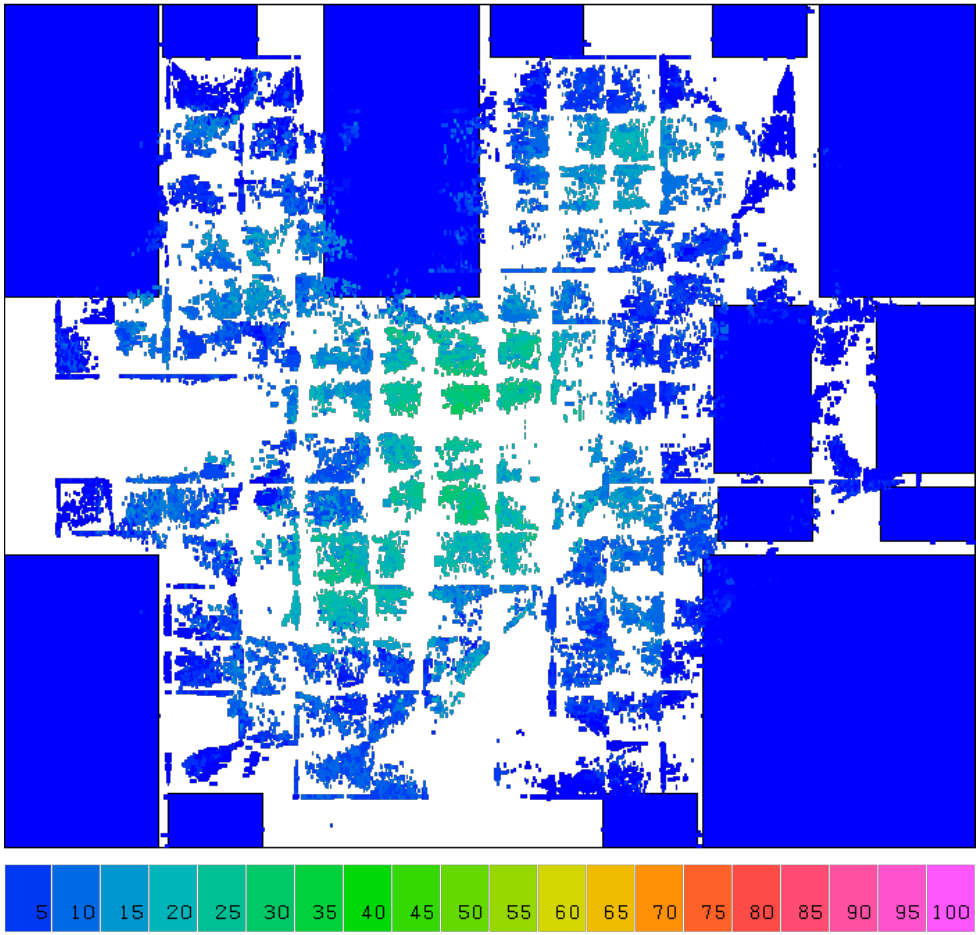
Example: Placement



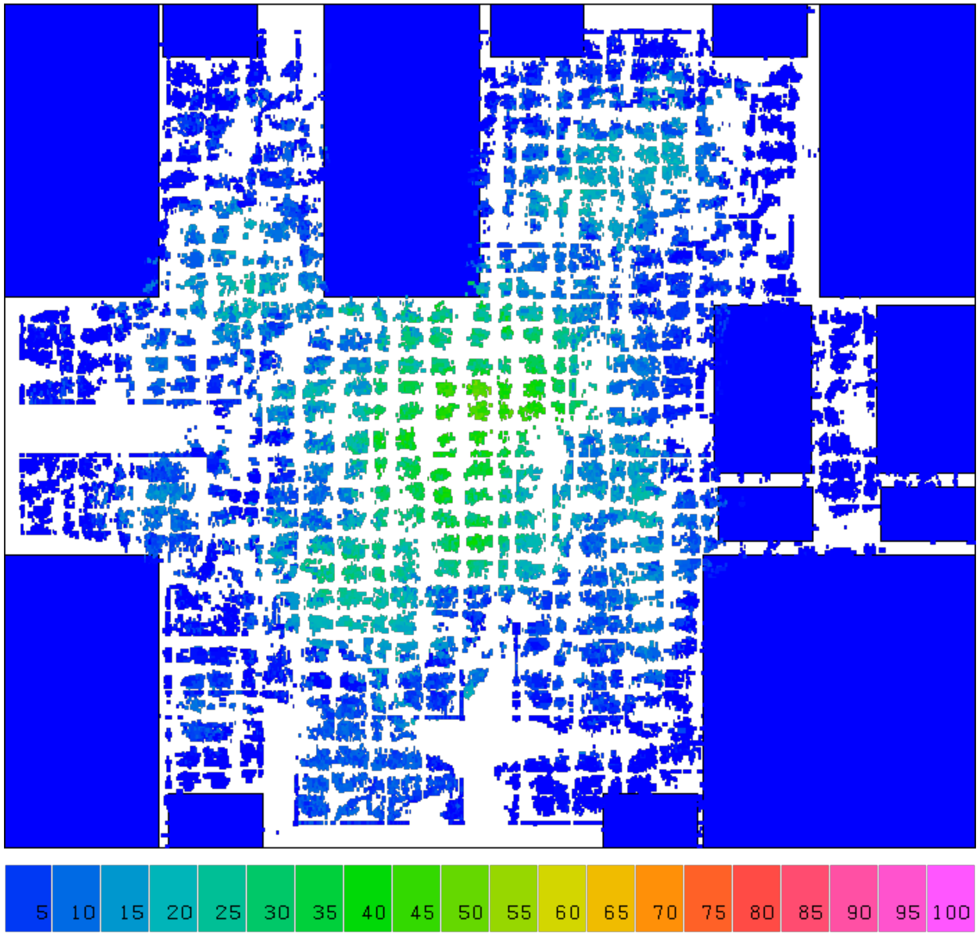
Example: Placement



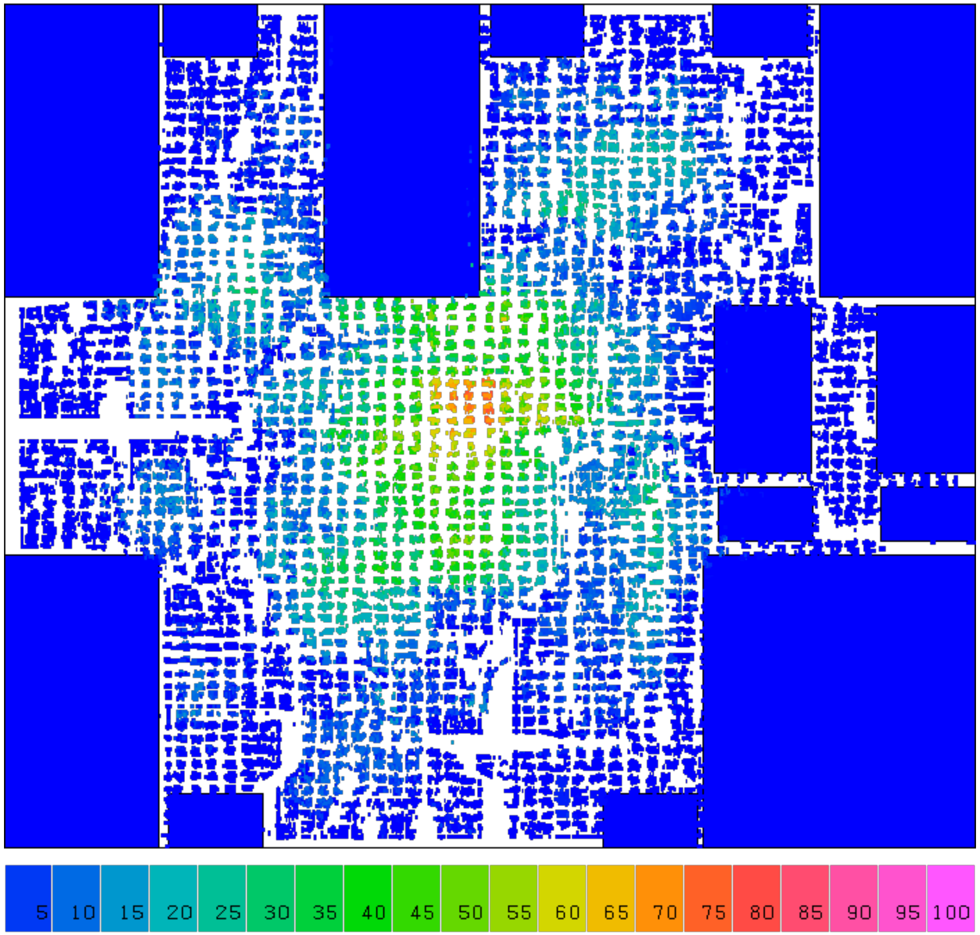
Example: Placement



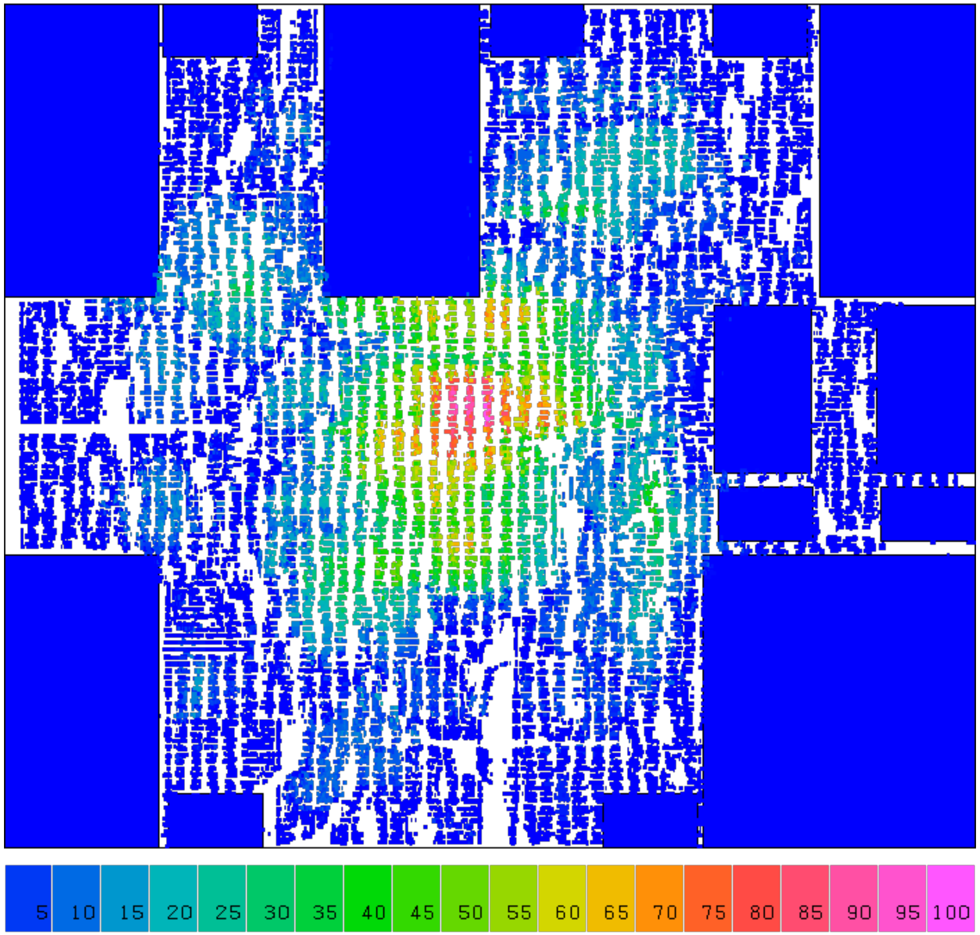
Example: Placement



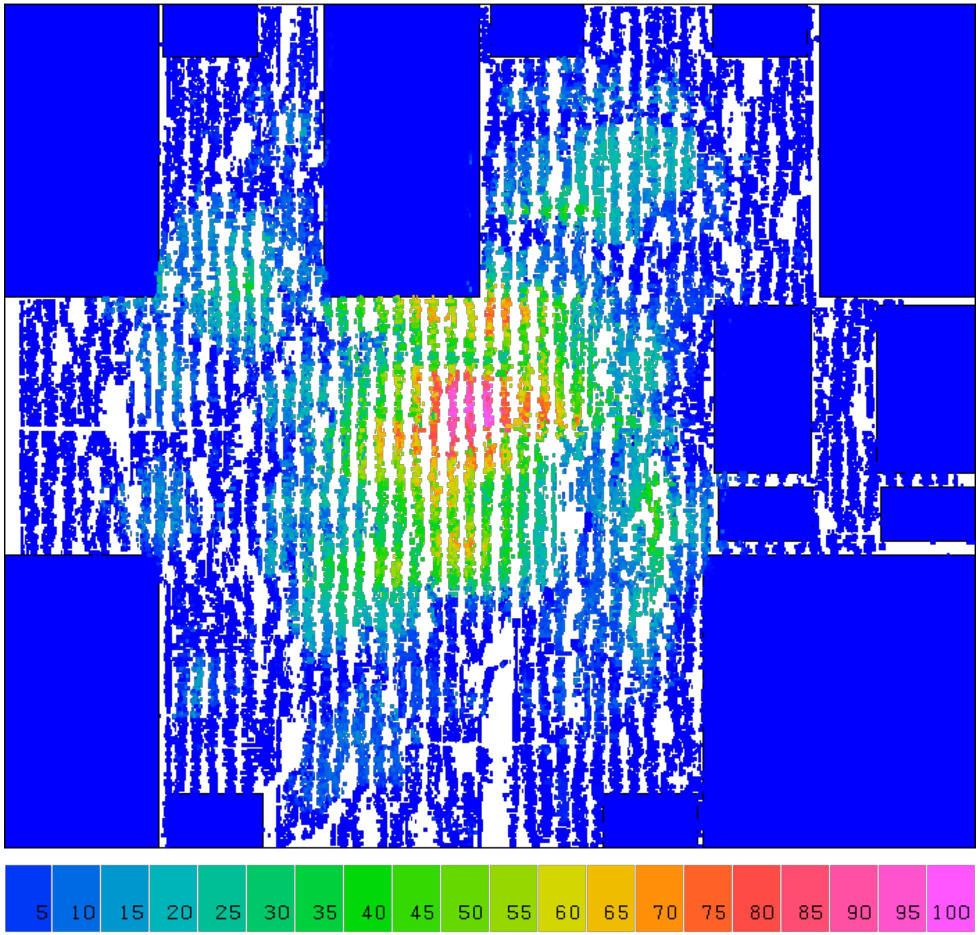
Example: Placement



Example: Placement



Example: Placement



Summary

- ❑ **Function**
- ❑ **Recursive Function**
- ❑ **Divide and Conquer**
    - ❑ Used to solve 90% of the computer science problems
    - ❑ Break a large problem into smaller pieces
    - ❑ Solve each smaller piece
    - ❑ Merge the solutions

LAB #1: Lower Bound of Sum

Write two functions named ”boundary_iterative” and “boundary_recursive” that both takes one integer argument, call it "goal" and return as its value the *smallest positive integer* n for which 1+2+3+. . . +n is at least equal or larger than “goal”

e.g. boundary_iterative(9) returns 4, as 1+2+3+4 >= 9 but 1+2+3<9

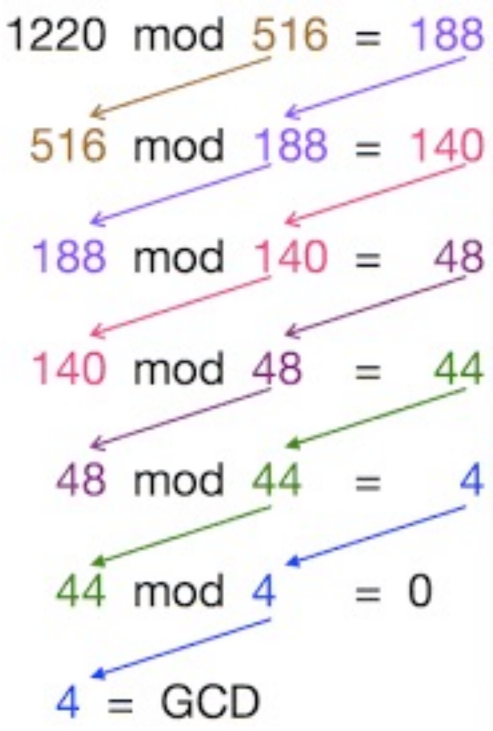
e.g., boundary_recursive(21) returns 6, as 1+2+3+4+5+6=21 but 1+2+3+4+5<21

LAB #2: Iterative and Recursive GCD

Write two functions named ”gcd_iterative” and “gcd_recursive” that both take two positive integers and return their greatest common divisor (GCD) using iterative for-loop and recursion

e.g. gcd_iterative(1220, 516) = 4

e.g. gcd_recursive(1220, 516) = 4



1220 mod 516 = 188

516 mod 188 = 140

188 mod 140 = 48

140 mod 48 = 44

48 mod 44 = 4

44 mod 4 = 0

4 = GCD