

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

```
1 // Creating and using a programmer-defined function.
2 #include <iostream>
3 using namespace std;
4
5 int square( int ); // function prototype
6
7 int main()
8 {
9     // Loop 10 times and calculate and output the
10    // square of x each time
11    for ( int x = 1; x <= 10; x++ )
12        cout << square( x ) << " ";
13    // function call
14
15    cout << endl;
16 } // end main
17
18 // square function definition returns square of an integer
19 int square( int y ) // y is a copy of argument to function
20 {
21     return y * y;      // returns square of y as an int
22 } // 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

```
1 // Comparing pass-by-value and pass-by-reference with references.
2 #include <iostream>
3 using namespace std;
4
5
6 int squareByValue( int ); // function prototype (value pass)
7 void squareByReference( int & ); // function prototype (reference pass)
8
9 int main()
10 {
11     int x = 2; // value to square using squareByValue
12     int z = 4; // value to square using squareByReference
13
14     // demonstrate squareByValue
15     cout << "x = " << x << " before squareByValue\n";
16     cout << "Value returned by squareByValue: "
17         << squareByValue( x ) << endl;
18     cout << "x = " << x << " after squareByValue\n" << endl;
19
20     // demonstrate squareByReference
21     cout << "z = " << z << " before squareByReference" << endl;
22     squareByReference( z );
23     cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

Call by Value vs Call by Reference

```
25
26 // squareByValue multiplies number by itself, stores the
27 // result in number and returns the new value of number
28 int squareByValue( int number )
29 {
30     return number *= number; // caller's argument not modified
31 } // end function squareByValue
32
33 // squareByReference multiplies numberRef by itself and stores the result
34 // in the variable to which numberRef refers in function main
35 void squareByReference( int &numberRef )
36 {
37     numberRef *= numberRef; // caller's argument modified
38 } // 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”

```
1 // Initializing and using a reference.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int x = 3;
8     int &y = x; // y refers to (is an alias for) x
9
10    cout << "x = " << x << endl << "y = " << y << endl;
11    y = 7; // actually modifies x
12    cout << "x = " << x << endl << "y = " << y << endl;
13
14 } // end main
```

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

Reference must be Initialized!

```
1 // References must be initialized.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int x = 3;  
8     int &y; // Error: y must be initialized  
9  
10    cout << "x = " << x << endl << "y = " << y << endl;  
11    y = 7;  
12    cout << "x = " << x << endl << "y = " << y << endl;  
13 } // 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

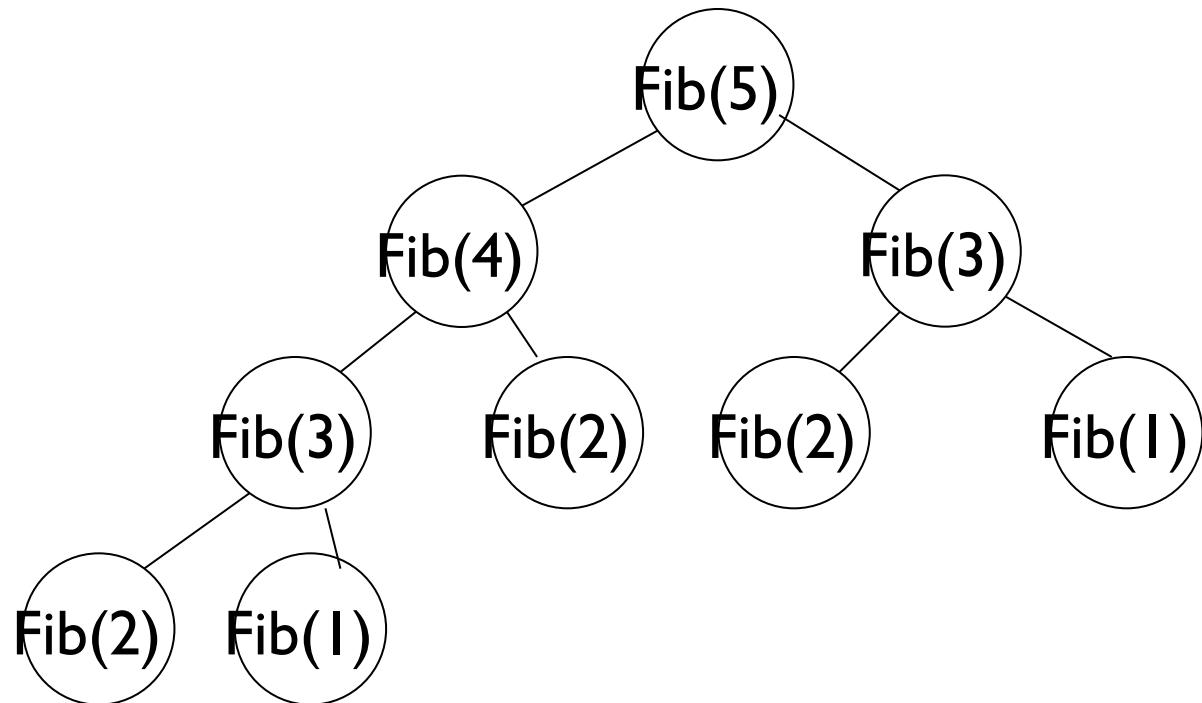
e.g. N=6, returns 8

e.g. N=9, returns 34

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

?

Duplicate Computations ...

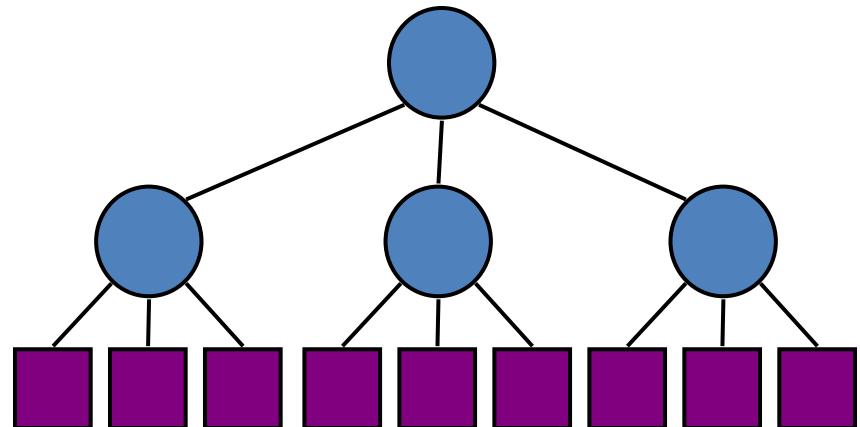
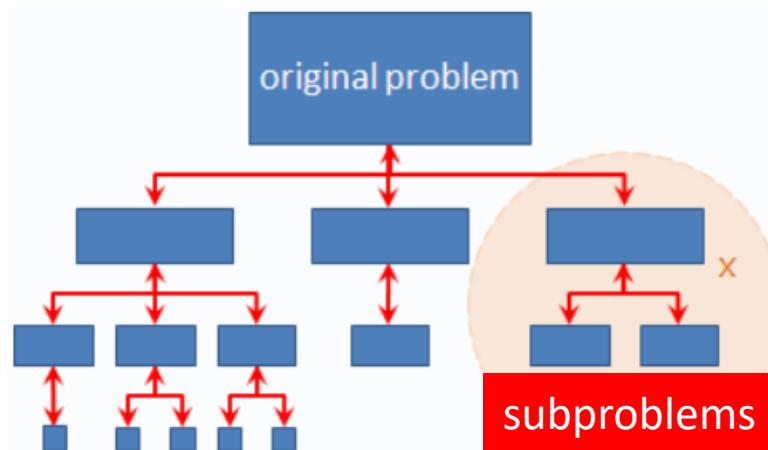


Divide and Conquer

❑ Divide & Conquer

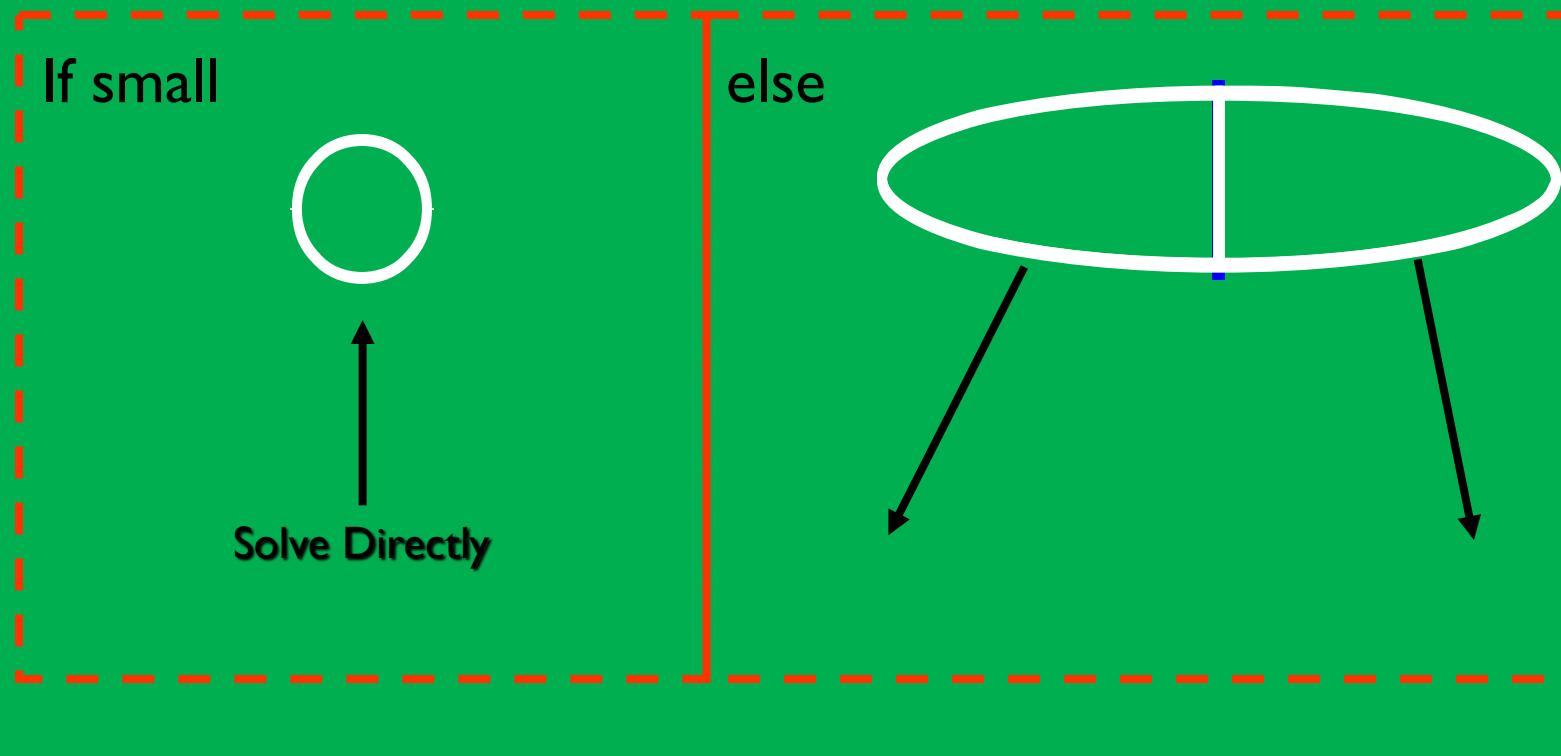
- ❑ Divide: Divide the original problem into smaller subproblems
- ❑ Recurse: Solve each small subproblem recursively
- ❑ Conquer: Combine these subproblems all the way to the top

❑ Illustration



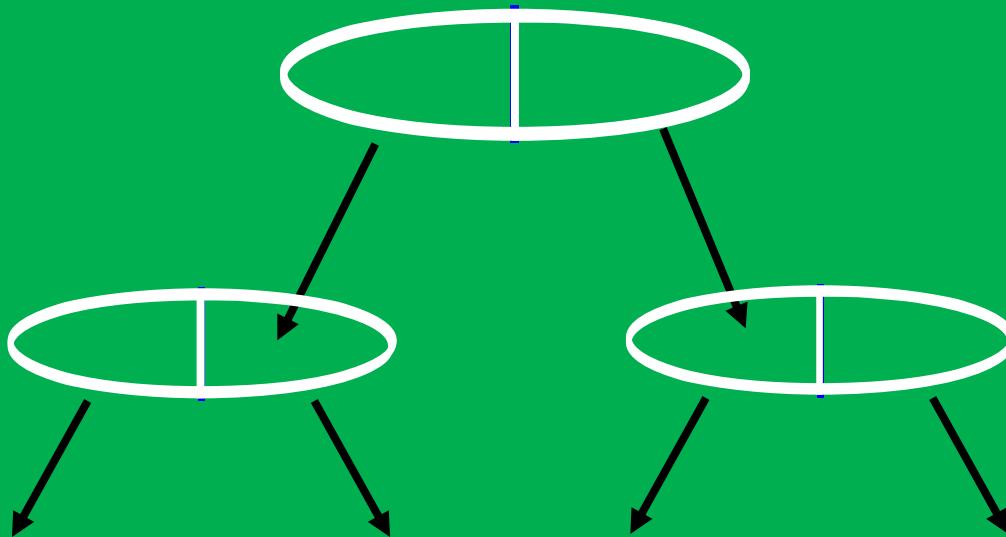
Visualization of Divide and Conquer

□ Divide



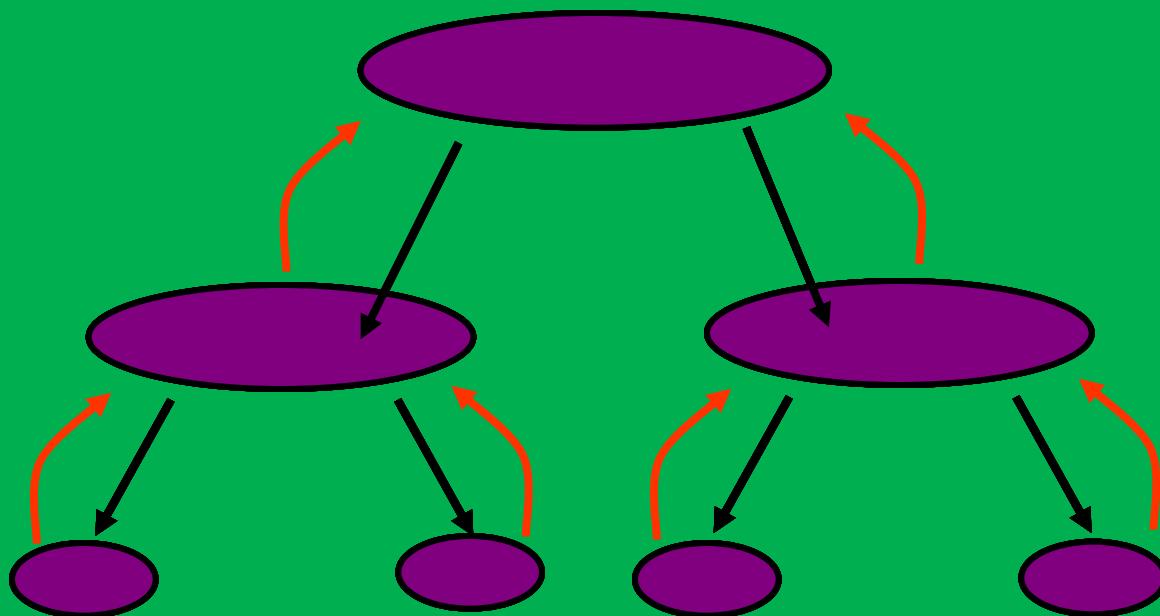
Visualization of Divide and Conquer

❑ Recursive



Visualization of Divide and Conquer

❑ Solve and Conquer



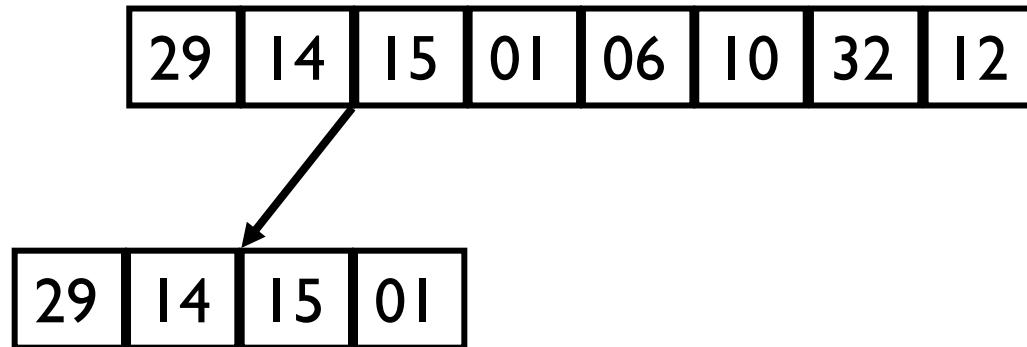
Example: Find the Maximum Value

□ Divide

29	14	15	01	06	10	32	12
----	----	----	----	----	----	----	----

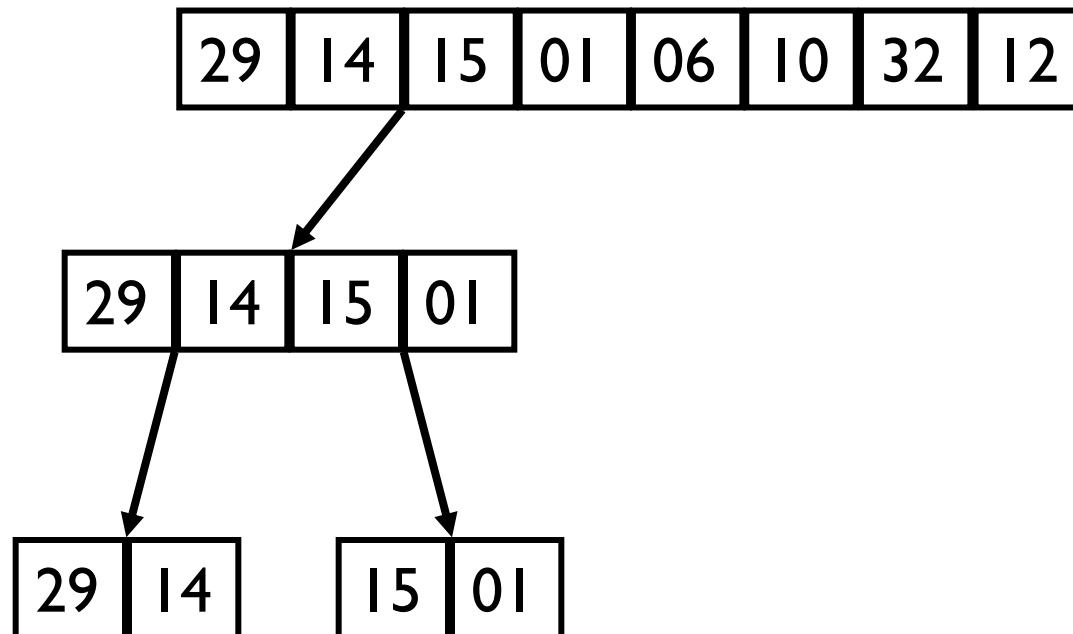
Example: Find the Maximum Value

□ Divide



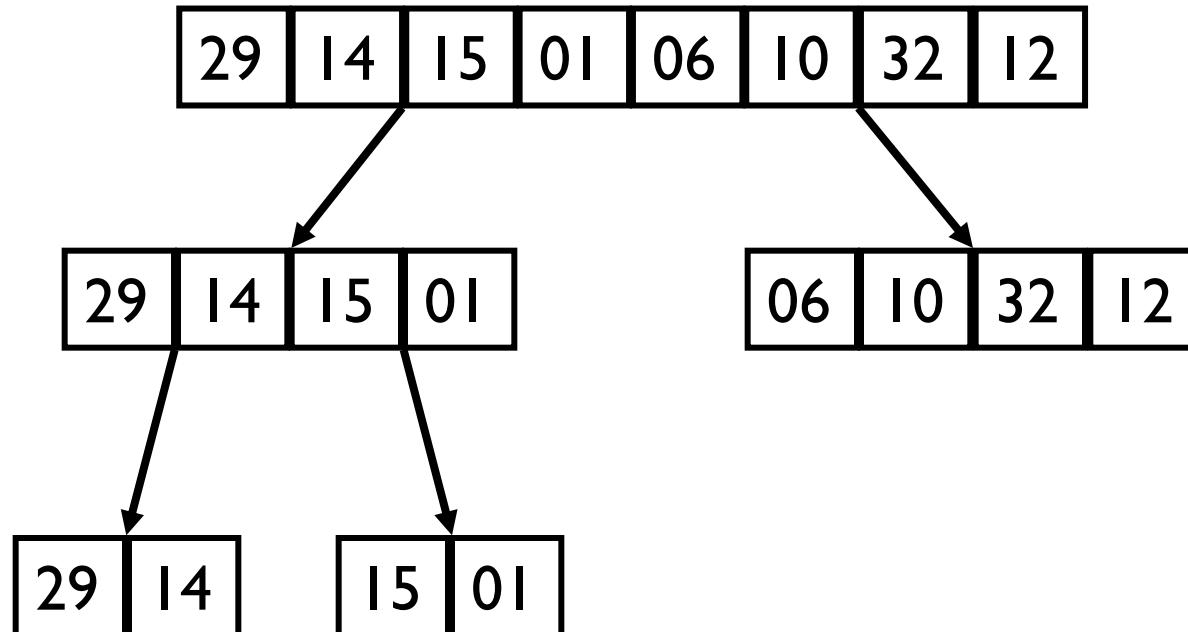
Example: Find the Maximum Value

□ Divide



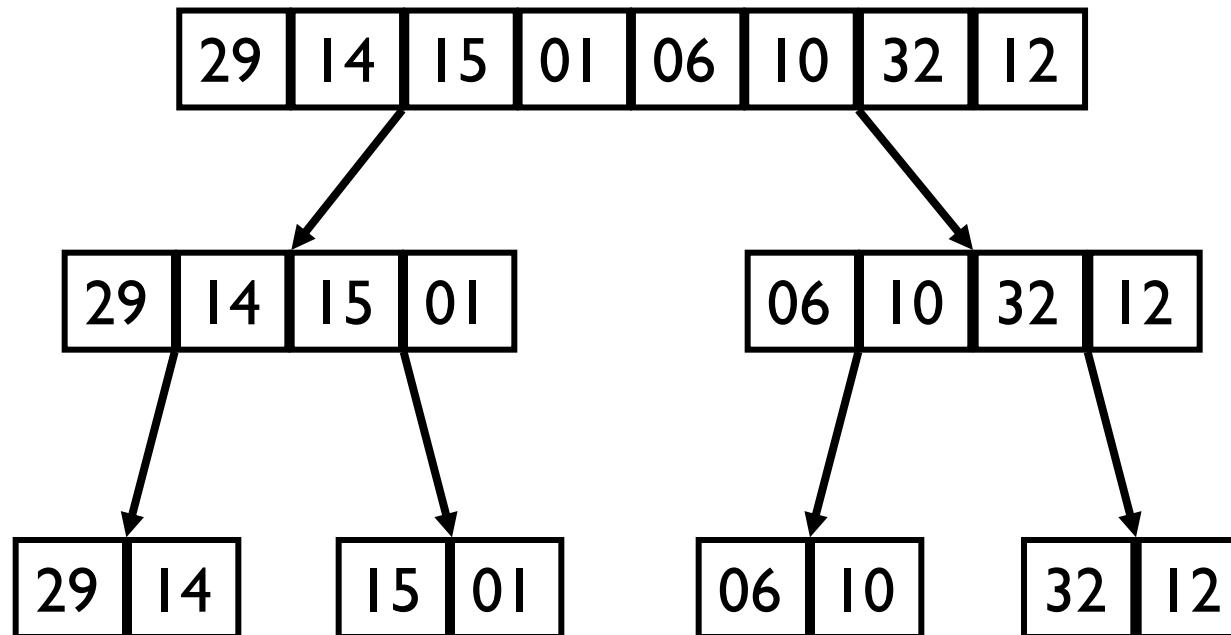
Example: Find the Maximum Value

□ Divide



Example: Find the Maximum Value

□ Divide



Example: Find the Maximum Value

□ Conquer

29	14
----	----

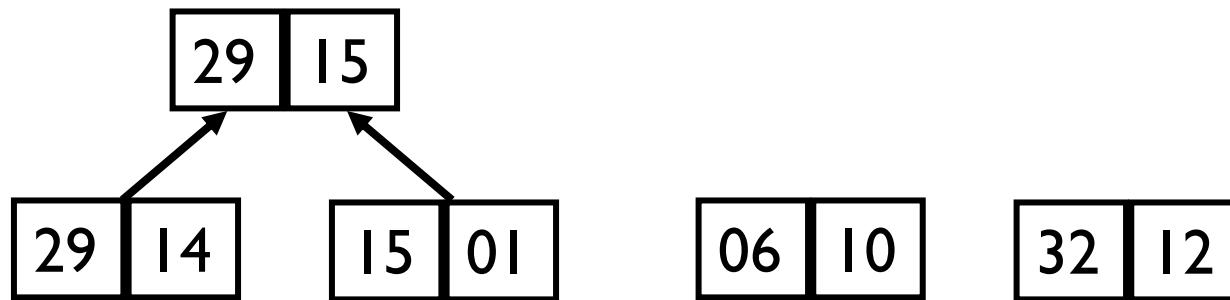
15	01
----	----

06	10
----	----

32	12
----	----

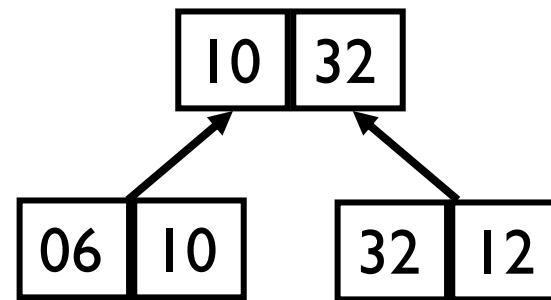
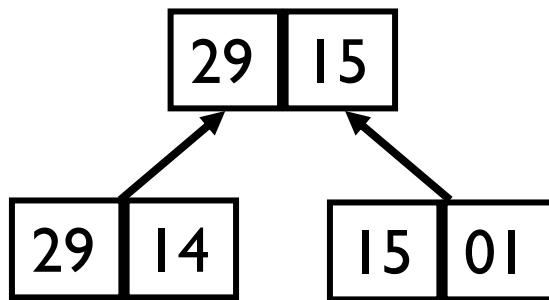
Example: Find the Maximum Value

□ Conquer



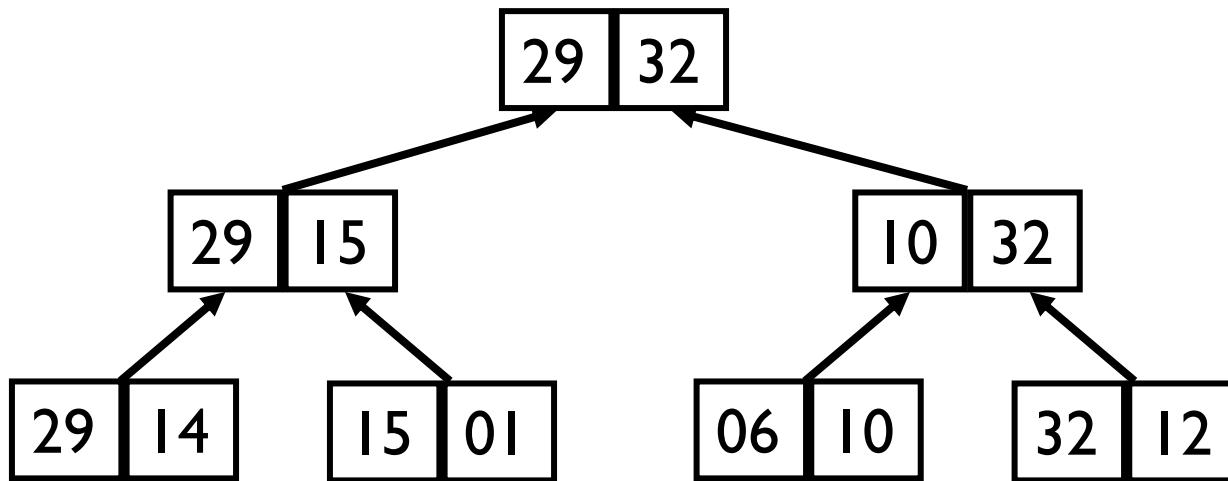
Example: Find the Maximum Value

□ Conquer



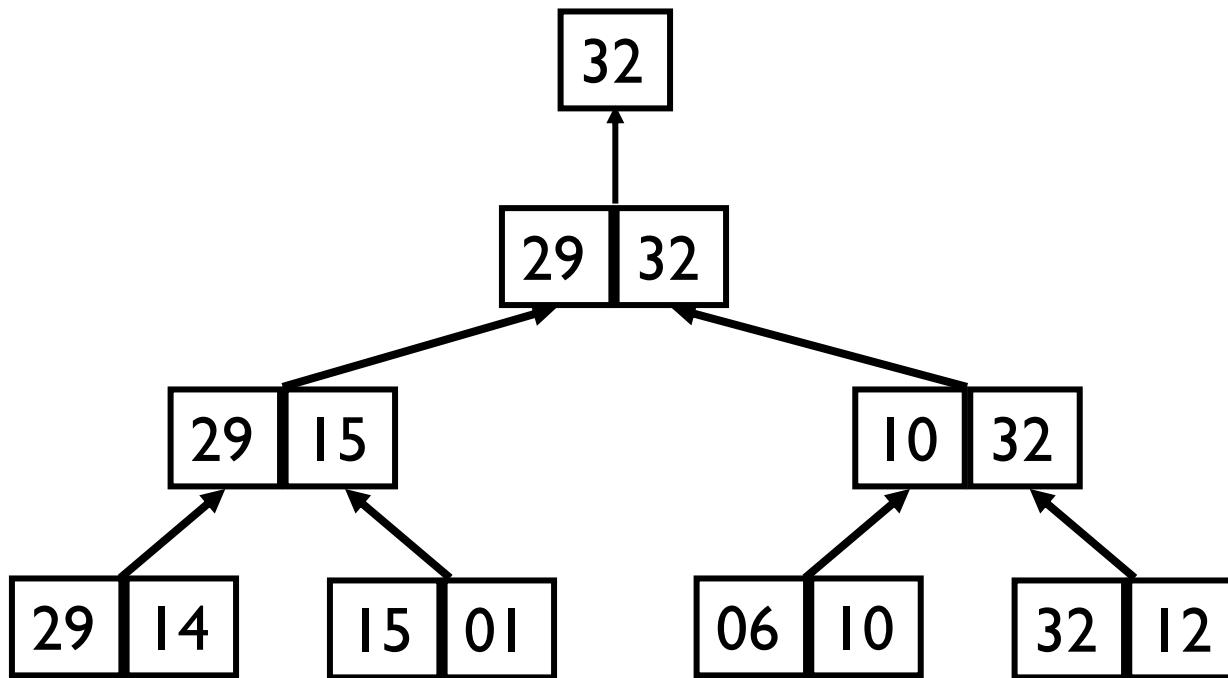
Example: Find the Maximum Value

□ Conquer



Example: Find the Maximum Value

□ Conquer

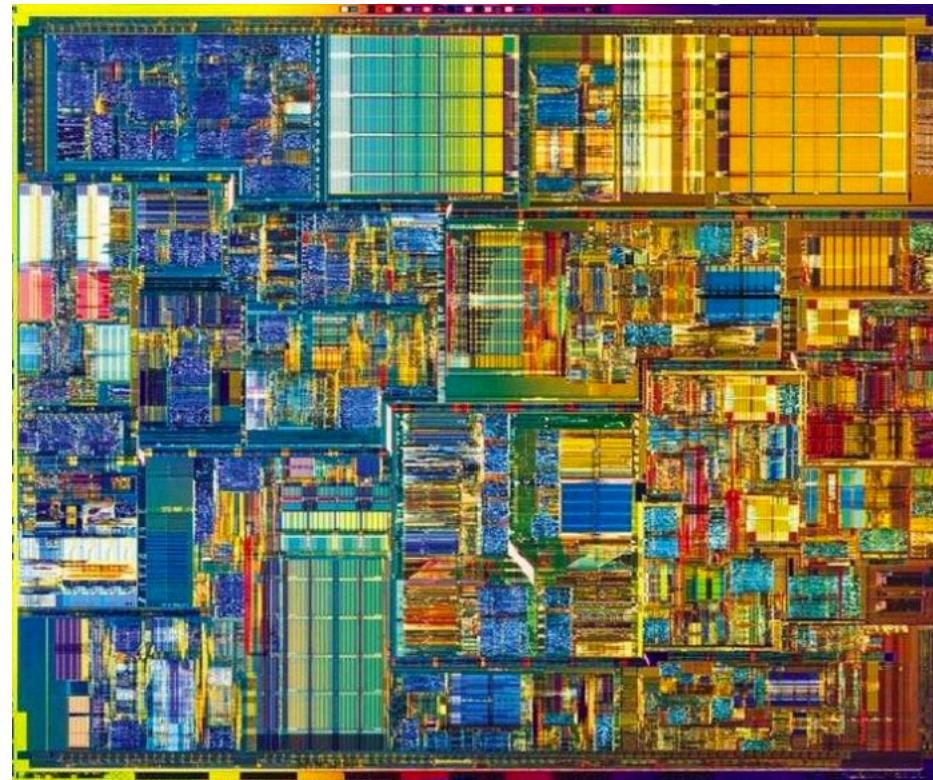


Divide and Conquer is Heavily used in CAD

- Modern circuits sizes are too large to handle in flat

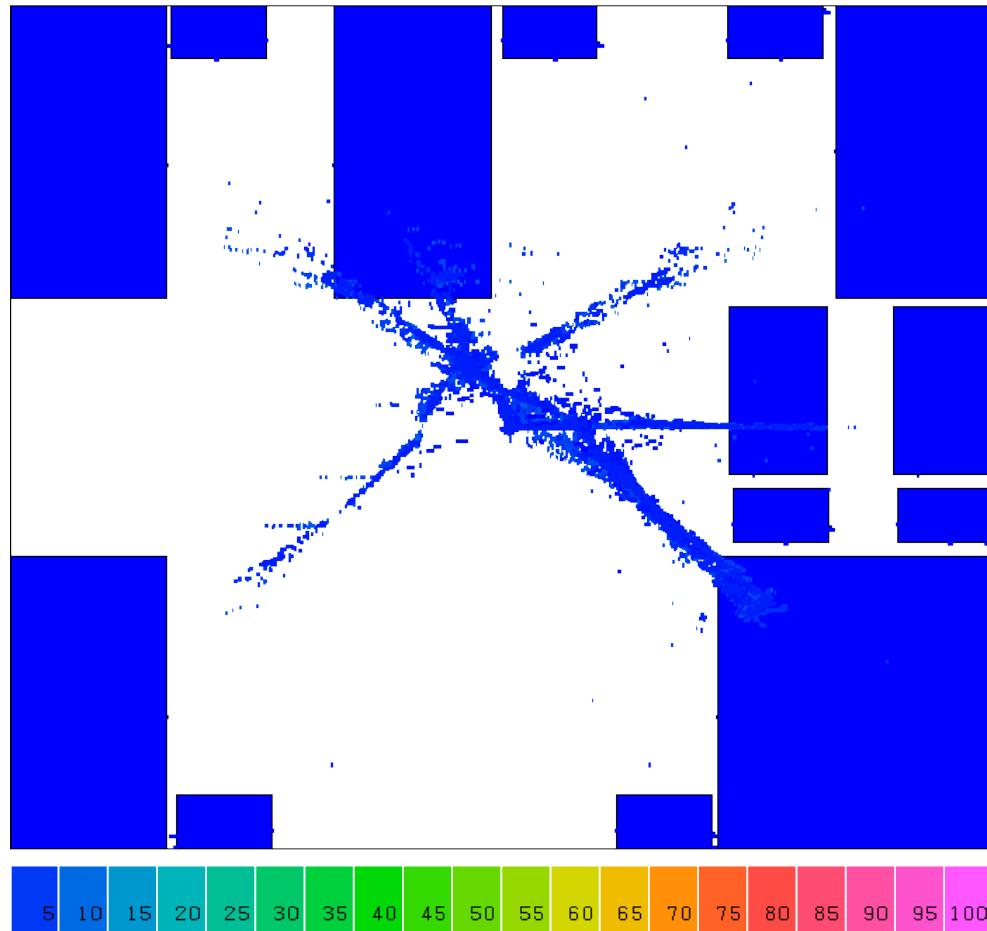


2000 – Intel Pentium4
42M transistors
1.5MHz; 224 mm²

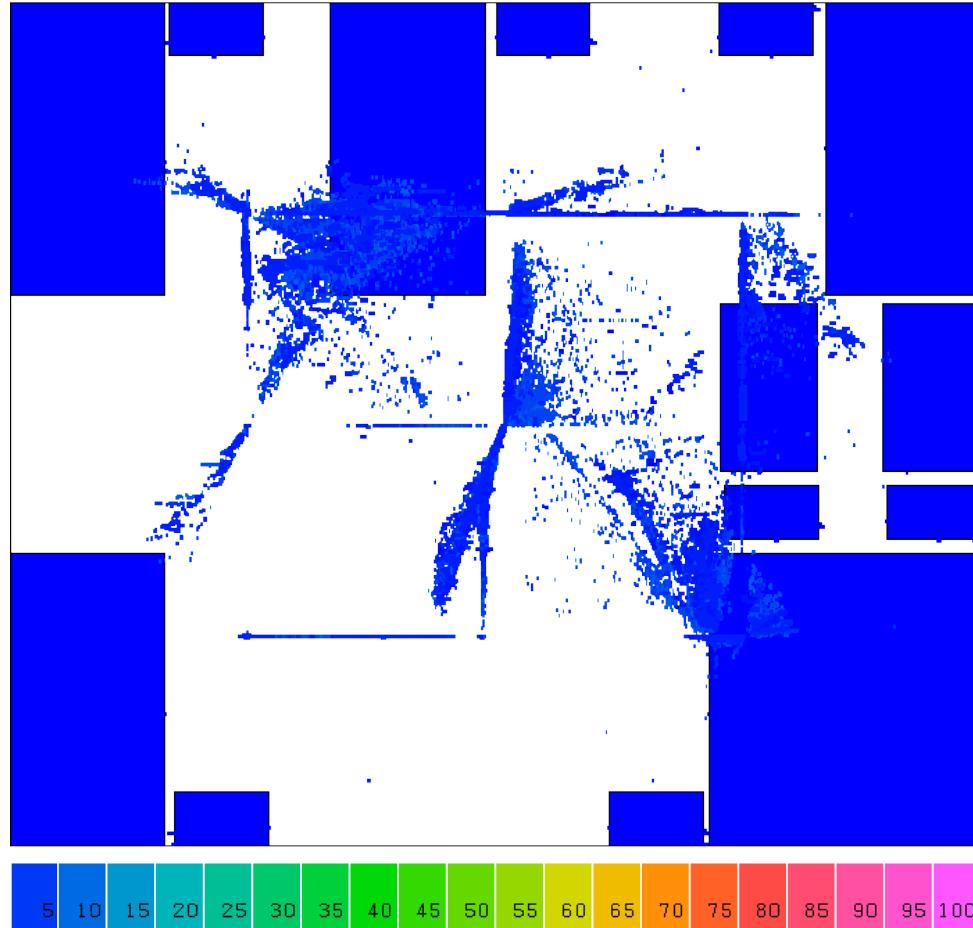


Directly solving the original problem takes forever to finish ...

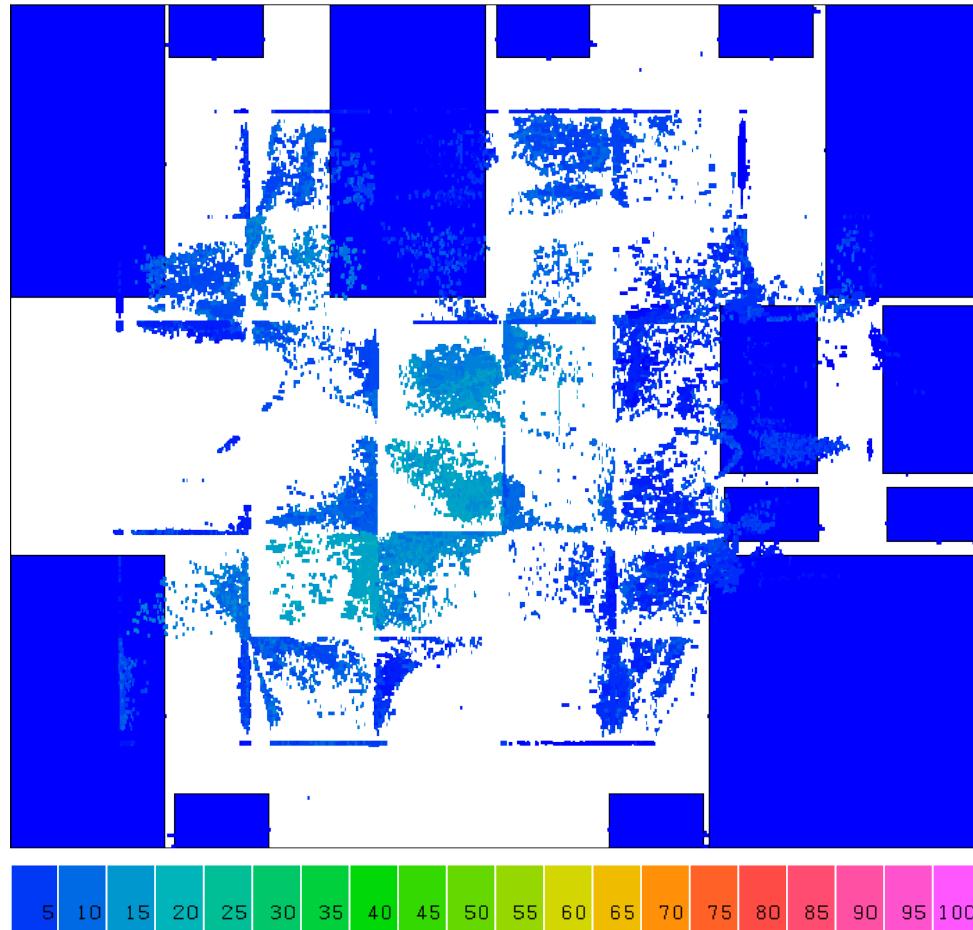
Example: Placement



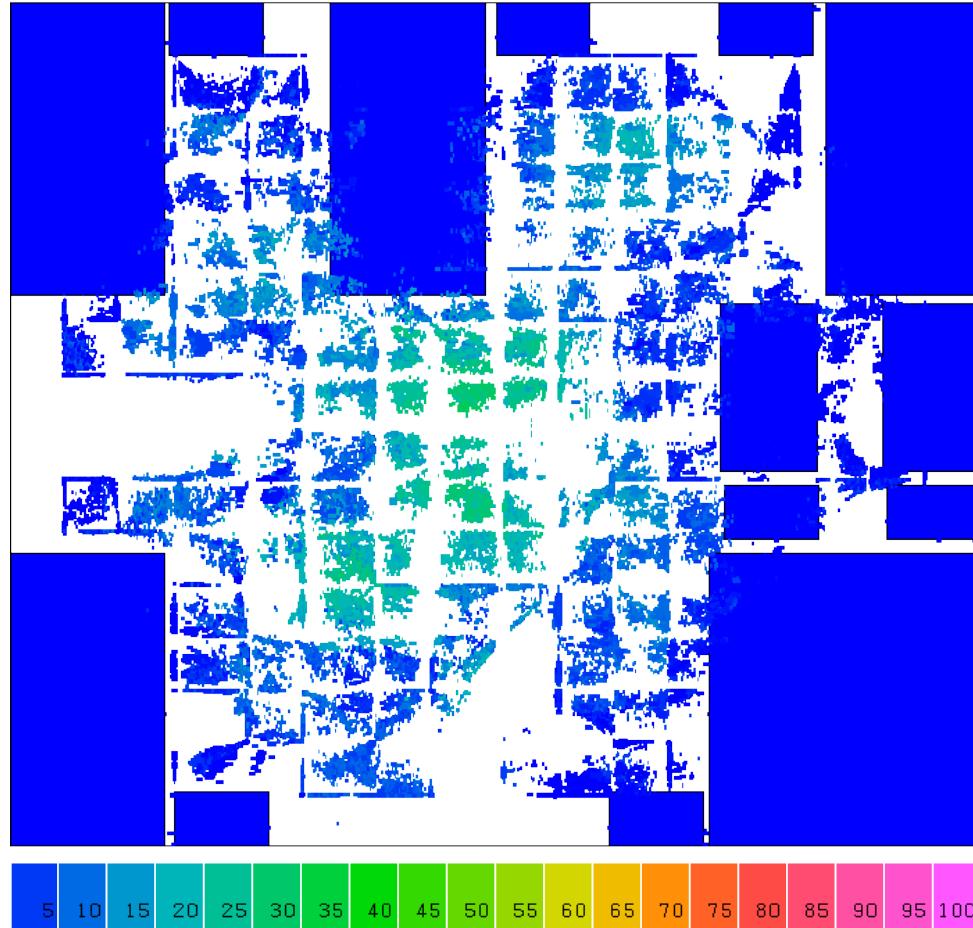
Example: Placement



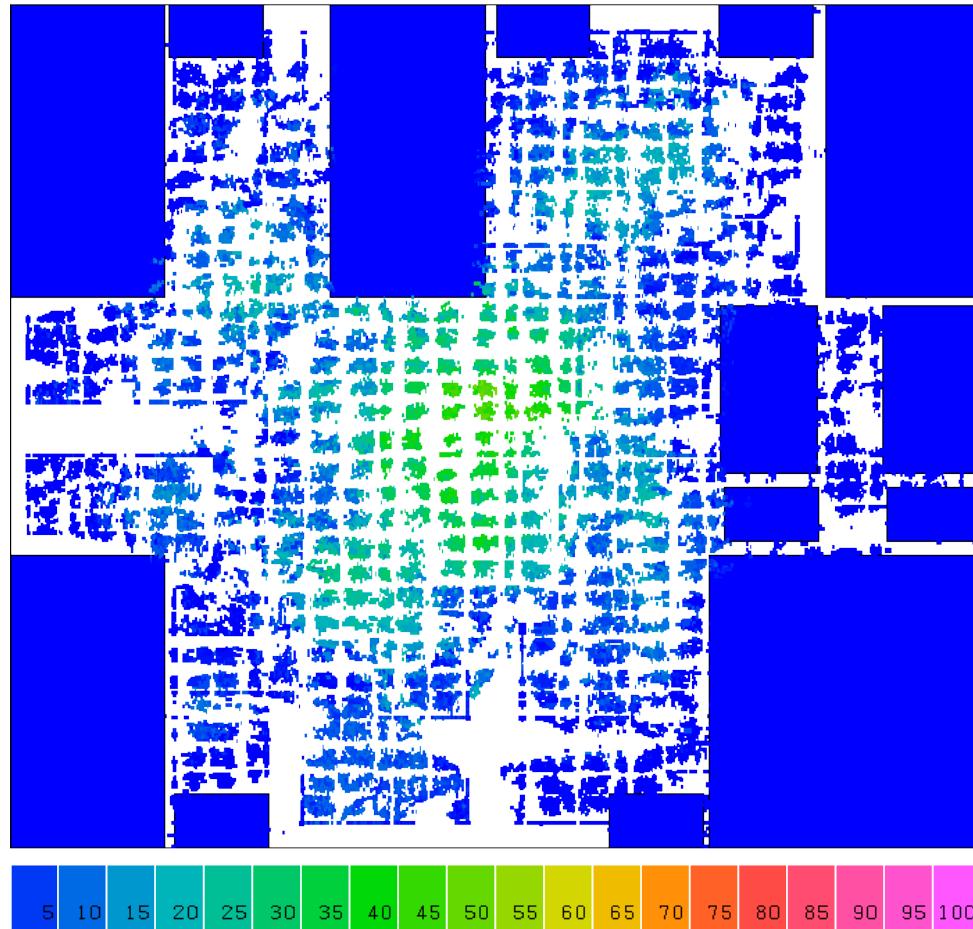
Example: Placement



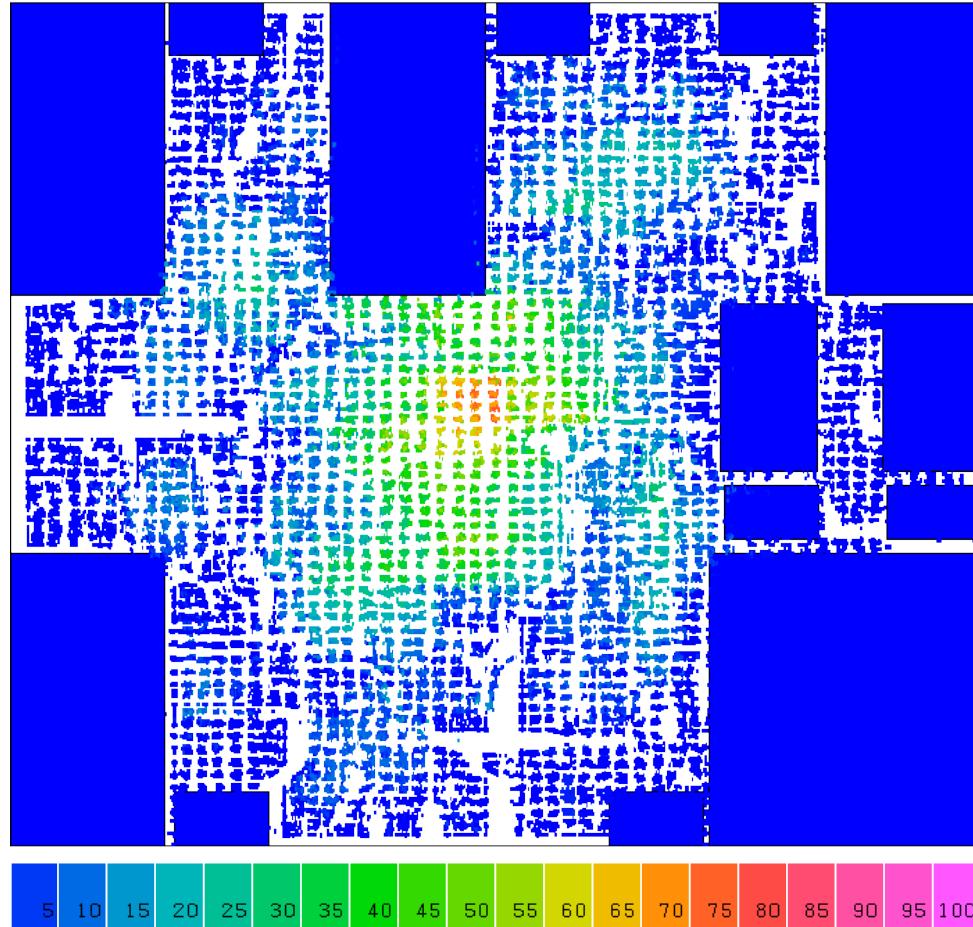
Example: Placement



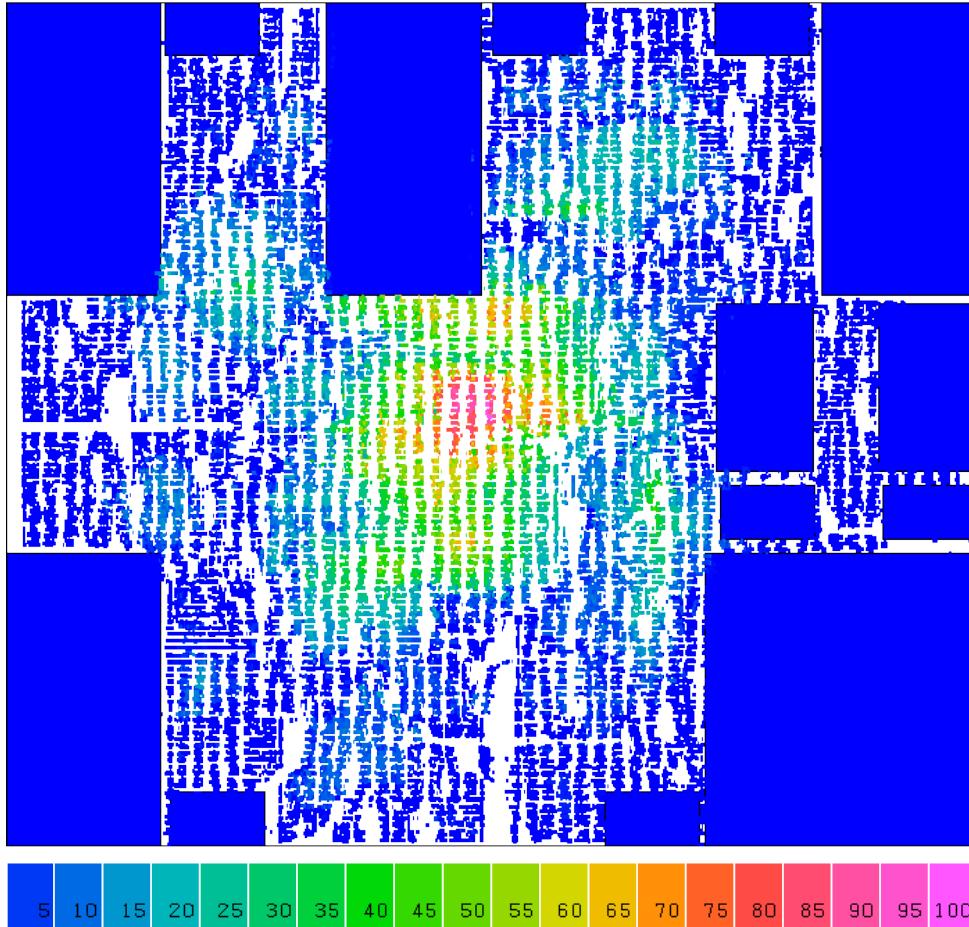
Example: Placement



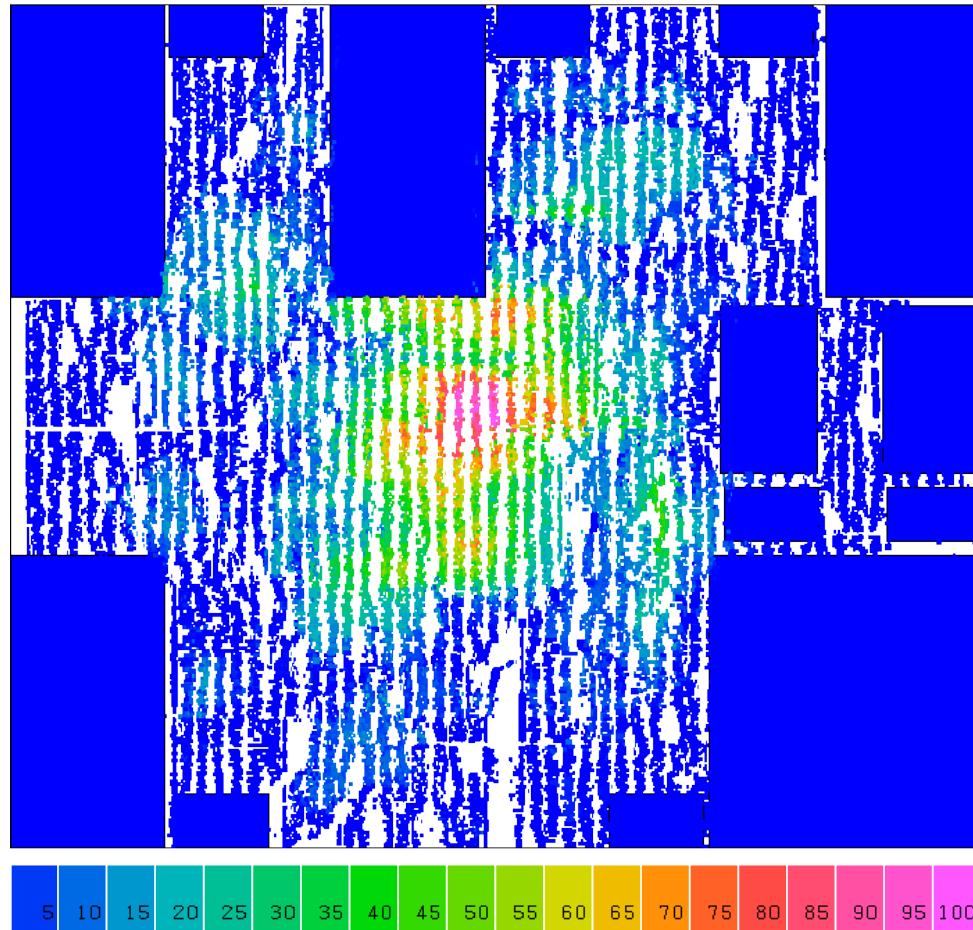
Example: Placement



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
- ❑ **PA 4 is announced (due 10/29 23:59 PM)**
- ❑ **Midterm will start next Friday class time**
 - ❑ Take-home exam with one week to finish

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 $1+2+3+\dots+n$ is at least equal or larger than "goal"

e.g. boundary_iterative(9) returns 4, as $1+2+3+4 \geq 9$ 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

