Recursion and Dynamic Programming

Recursive thinking…

- Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem

– or, in other words, a programming technique in which a method can call itself to solve a problem.

Example: Factorial

- The factorial for any positive integer *n*, written *n*!, is defined to be the product of all integers between 1 and *n* inclusive

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Non-recursive solution

```
// not recursive
public static long factorial(int n) {
    long product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

Recursive factorial

$$Factorial(N) = \begin{cases} N \cdot Factorial(N-1) & \text{if } N > 0 \\ 1 & \text{if } N = 0 \end{cases}$$

```
// recursive
public static long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Key Features of Recursions

- Simple solution for a few cases
- Recursive definition for other values
  - Computation of large N depends on smaller N
- Can be naturally expressed in a function that calls itself
  - Loops are many times an alternative

Recursive power example

- Write method pow that takes integers x and y as parameters and returns x^y.

  $x^y = x * x * x * ... * x$ (y times, in total)

  An iterative solution:

```
// not recursive
public static int pow(int x, int y) {
    int product = 1;
    for (int i = 0; i < y; i++) {
        product = product * x;
    }
    return product;
}
```

Recursive power function

- Another way to define the power function:

pow(x, 0) = 1
pow(x, y) = x * pow(x, y-1), y > 0

```
// recursive
public static int pow(int x, int y) {
    if (y == 0) {
        return 1;
    } else {
        return x * pow(x, y - 1);
    }
}
```

How recursion works

- each call sets up a new instance of all the parameters and the local variables

- as always, when the method completes, control returns to the method that invoked it (which might be another invocation of the same method)

```
pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 1
          = 64
```

Infinite recursion

- a definition with a missing or badly written base case causes **infinite recursion**, similar to an infinite loop
  - avoided by making sure that the recursive call gets closer to the solution (moving toward the base case)

```
public static int pow(int x, int y) {
    return x * pow(x, y - 1);  // Oops! Forgot base case
}
```

```
pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 4 * pow(4, -1)
          = 4 * 4 * 4 * 4 * 4 * pow(4, -2)
          = ... crashes: Stack Overflow Error!
```

Divide-and-conquer Algorithms

Divide-and-conquer algorithm: a means for solving a problem that

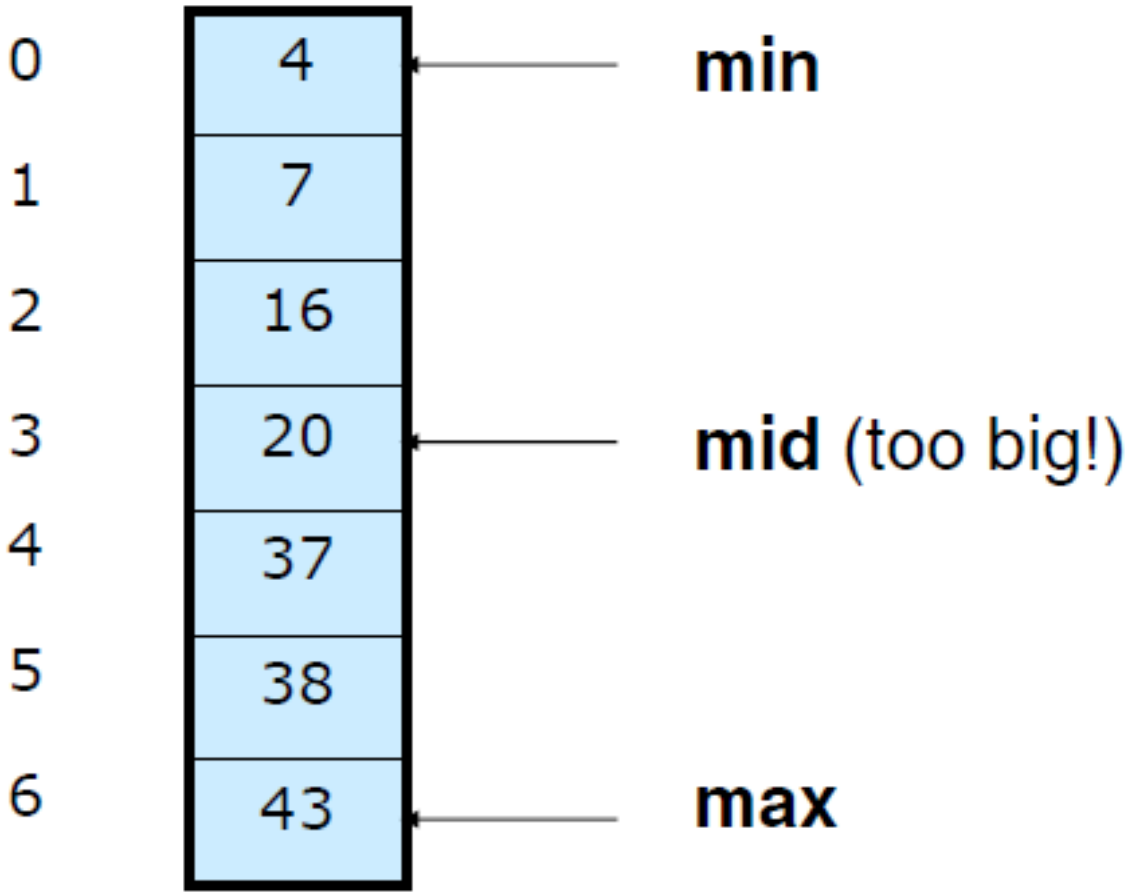
- first separates the main problem into 2 or more smaller problems,
- Then solves each of the smaller problems,
- Then uses those sub-solutions to solve the original problem

```
1: "divide" the problem up into pieces
2: "conquer" each smaller piece
3: (if necessary) combine the pieces at the
   end to produce the overall solution
```

Binary search is one such algorithm

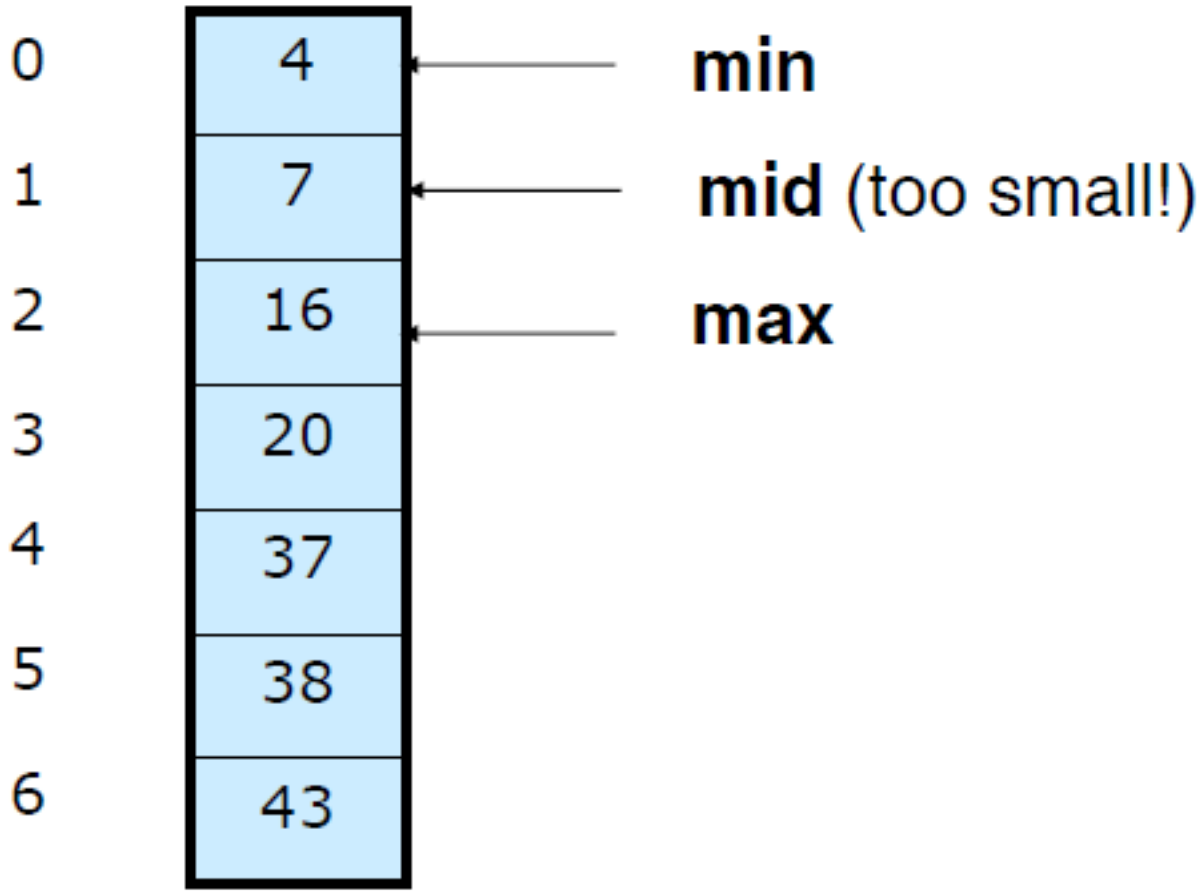
Binary Search

i = 16



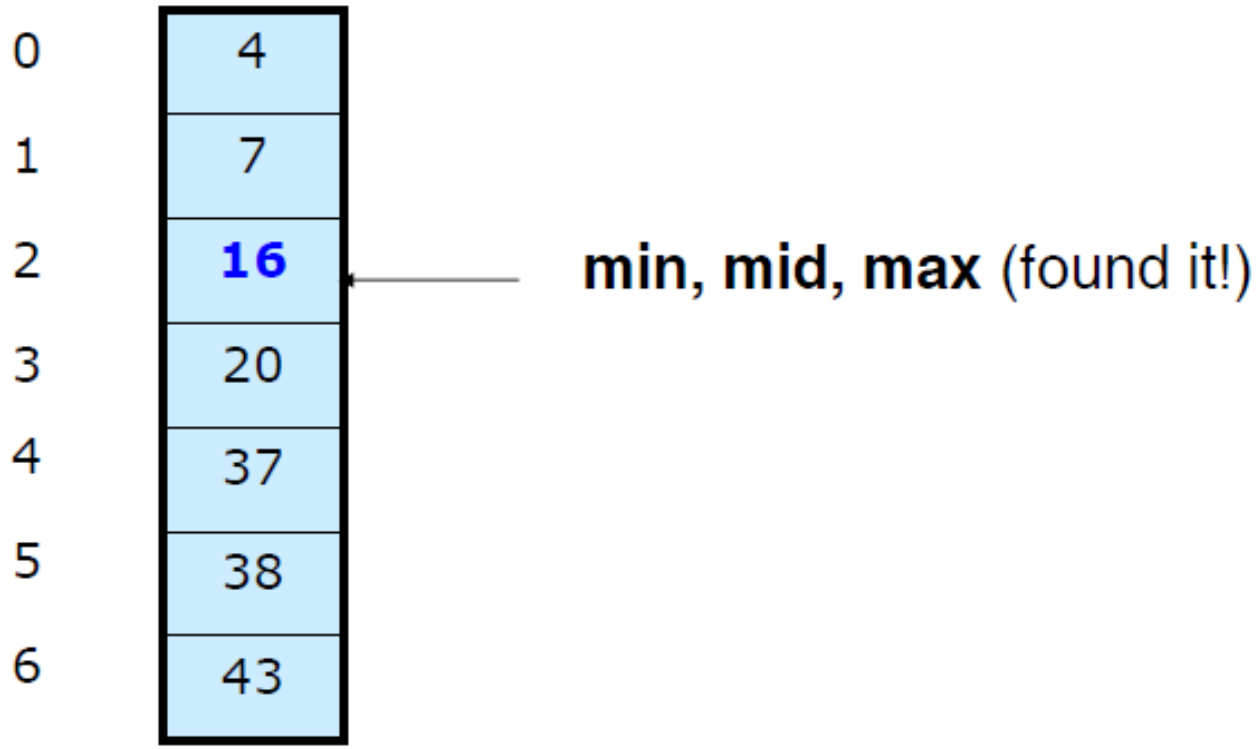
Binary Search

i = 16



Binary Search

i = 16



Binary search pseudocode

binary search array *a* for value *i*:
  if all elements have been searched,
    result is -1.
  examine middle element *a*[*mid*].
  if a[*mid*] equals *i*,
    result is *mid*.
  if *a*[*mid*] is greater than *i*,
    binary search left half of *a* for *i*.
  if *a*[*mid*] is less than *i*,
    binary search right half of *a* for *i*.

Recursive Binary Search

```
int search(int a[], int value, int start, int stop)
   {
   // Search failed
   if (start > stop)
      return -1;

   // Find midpoint
   int mid = (start + stop) / 2;

   // Compare midpoint to value
   if (value == a[mid]) return mid;

   // Reduce input in half!!!
   if (value < a[mid])
     return search(a, start, mid - 1);
   else
     return search(a, mid + 1, stop);
   }
```

Runtime of binary search

- How do we analyze the runtime of binary search?

- binary search either exits immediately, when input size <= 1 or value found (base case),
  or executes itself on 1/2 as large an input (rec. case)
  - T(1) = c
  - T(2) = T(1) + c
  - T(4) = T(2) + c
  - T(8) = T(4) + c
  - ...
  - T(n) = T(n/2) + c

- How many times does this division in half take place?

Recursive Maximum

```
int Maximum(int a[], int start, int stop)
    {
    int left, right;

    // Maximum of one element
    if (start == stop)
       return a[start];

    left = Maximum(a, start, (start + stop) / 2);
    right = Maximum(a, (start + stop) / 2 + 1, stop);

    // Reduce input in half!!!
    if (left > right)
       return left;
    else
       return right;
    }
```

Recursion vs. iteration

- every recursive solution has a corresponding iterative solution
  - For example, *N* ! can be calculated with a loop
- recursion has the overhead of multiple method invocations
- however, for some problems recursive solutions are often more simple and elegant than iterative solutions
- you must be able to determine when recursion is appropriate

Recursion can perform badly

- Fibonacci numbers

$$Fibonacci(N) = \begin{cases} 0 & \text{if } N = 0 \\ 1 & \text{if } N = 1 \\ Fibonacci(N-1) + Fibonacci(N-2) & \end{cases}$$

The higher up in the sequence, the closer two consecutive "Fibonacci numbers" of the sequence divided by each other will approach the golden ratio (approximately 1 : 1.618)

Recursion can perform badly

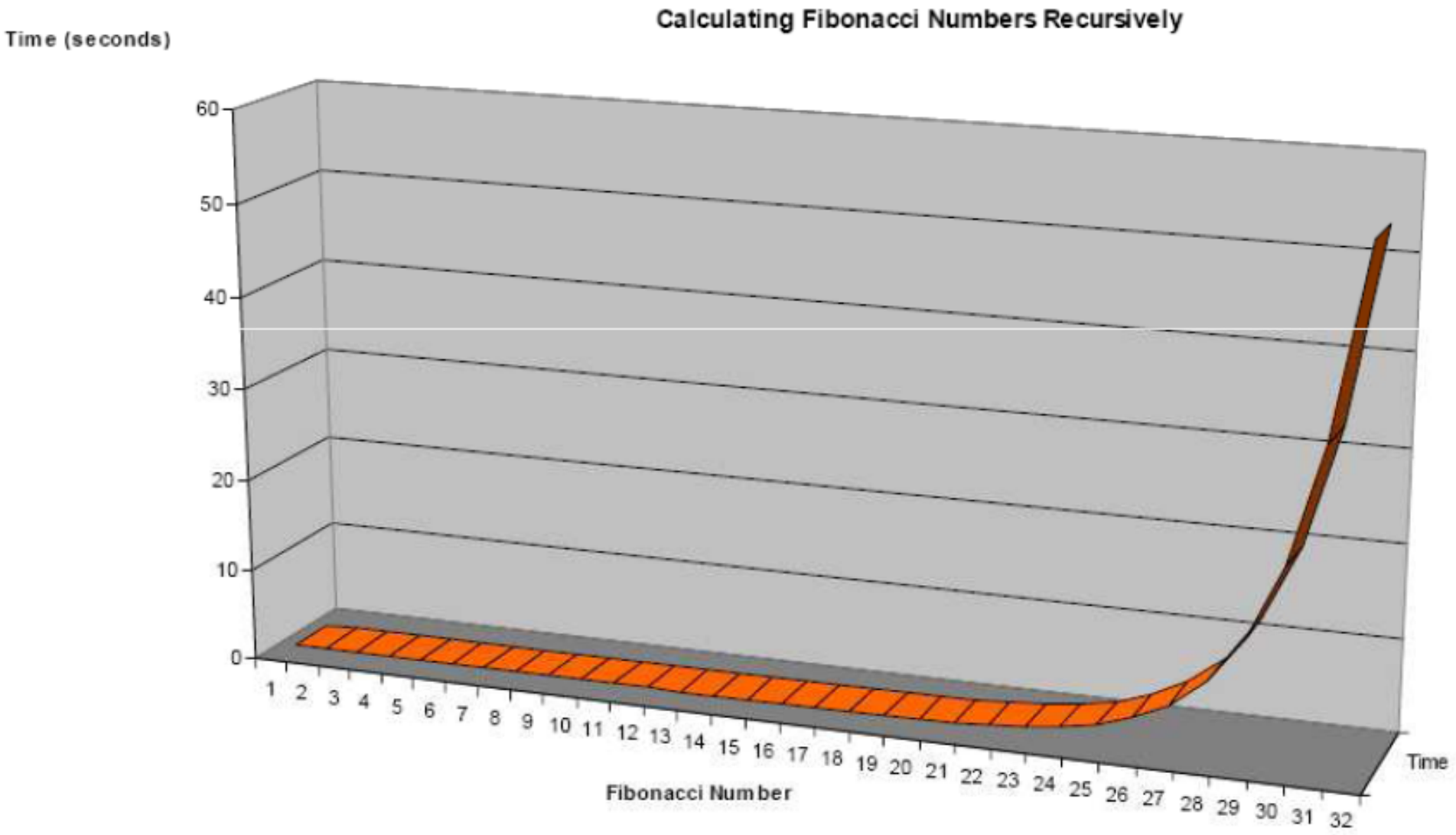
- Consider the Fibonacci numbers

$$Fibonacci(N) = \begin{cases} 0 & \text{if } N = 0 \\ 1 & \text{if } N = 1 \\ Fibonacci(N-1) + Fibonacci(N-2) & \end{cases}$$

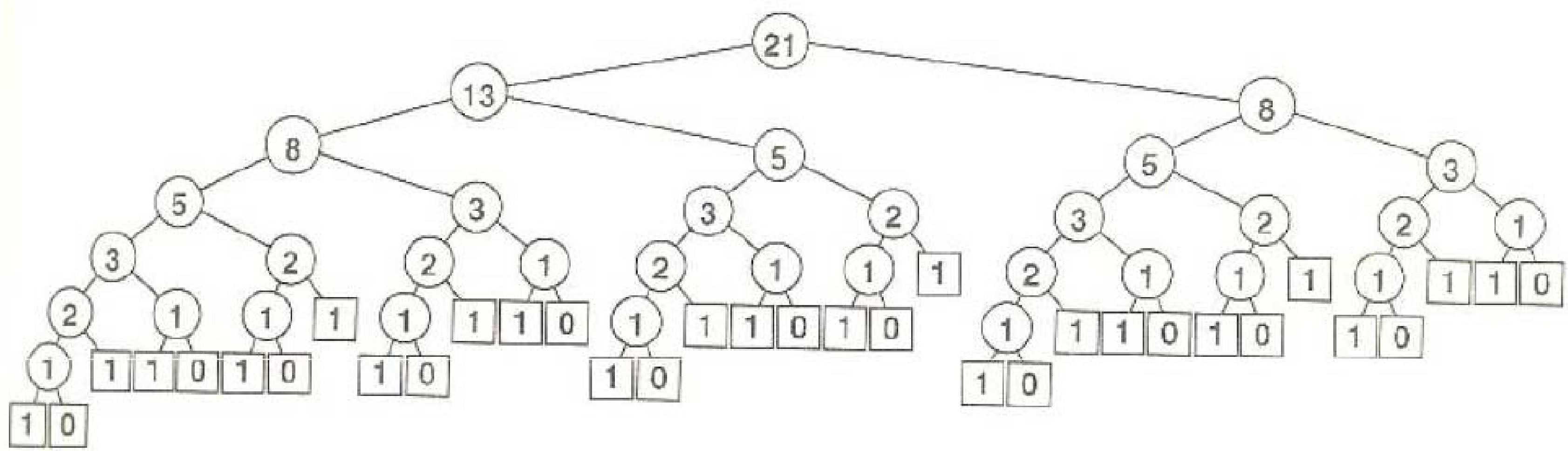
Fibonacci numbers

```
int Fibonacci(int i)
   {
   // Simple cases first
   if (i == 0)
      return 0;

   if (i == 1)
      return 1;

   return Fibonacci(i - 1) + Fibonacci(i - 2);
   }
```

Terribly slow!



What is going on?



- Certain quantities are recalculated far too many times!
- Need to avoid recalculation…
  - Ideally, calculate each unique quantity once

Dynamic Programming

- **dynamic programming**: saving results of sub-problems so that they do not need to be recomputed, and can be used in solving other sub-problems
  - example: saving results from sub-calls in a list or table
  - can dramatically speed up the number of calls for a recursive function with overlapping sub-problems

Dynamic Programming

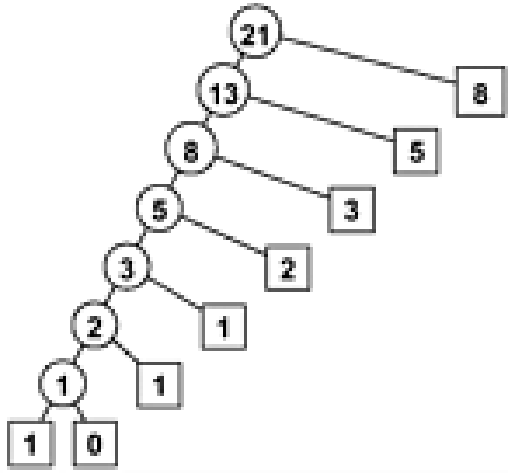
```
//non recursive
static int fibonacci(int i) {
    int [] fib = new int[i+1];

    fib[0] = 0;
    fib[1] = 1;

    for (int j=2; j<=i; j++)
        fib[j]=fib[j-1]+fib[j-2];

    return fib[i];
}
```

Dynamic Programming

```
static final int maxN = 400;
static int [] knownF = new int[maxN];

//recursive
static int fibonacci(int i) {

    if (i == 1)
        return 1;

    if (i <= 0)
        return 0;

    if (knownF[i] != 0)
        return knownF[i];

    knownF[i] = fibonacci(i-1) + fibonacci(i-2);

    return knownF[i];
}
```

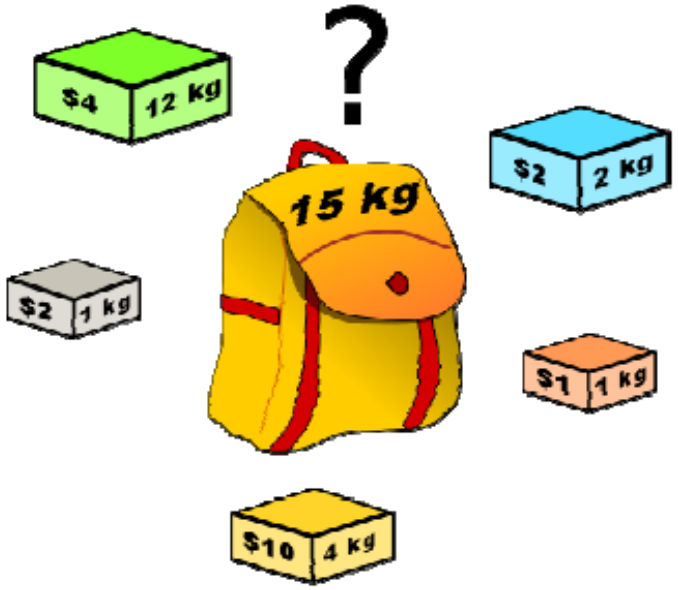
Dynamic programming

Time: linear



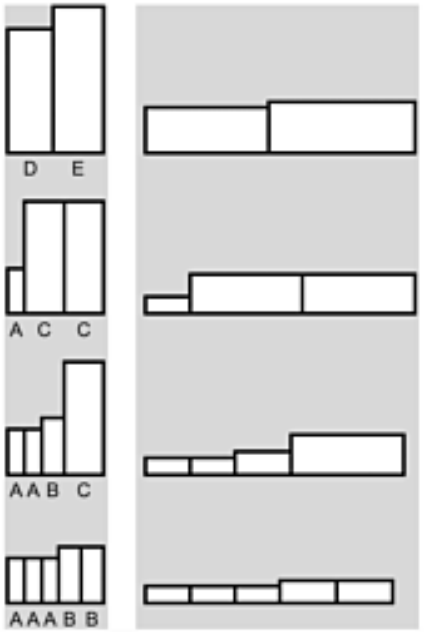
Knapsack problem

- A thief robbing a safe finds it filled with N types of items of varying size and value but has only a small knapsack of capacity M to use to carry the goods.
- ***Knapsack problem***: find a combination of items which the thief should choose for the knapsack in order to maximize the value of all the stolen items.



The Knapsack problem

	0	1	2	3	4
item	A	B	C	D	E
size	3	4	7	8	9
val	4	5	10	11	13



Size	Val
17	24
17	24
17	23
17	22

An instance of the knapsack problem consists of a knapsack capacity and a set of items of varying size (horizontal dimension) and value (vertical dimension).

This figure shows four different ways to fill a knapsack of size 17, two of which lead to the highest possible total value of 24.

Recursive solution

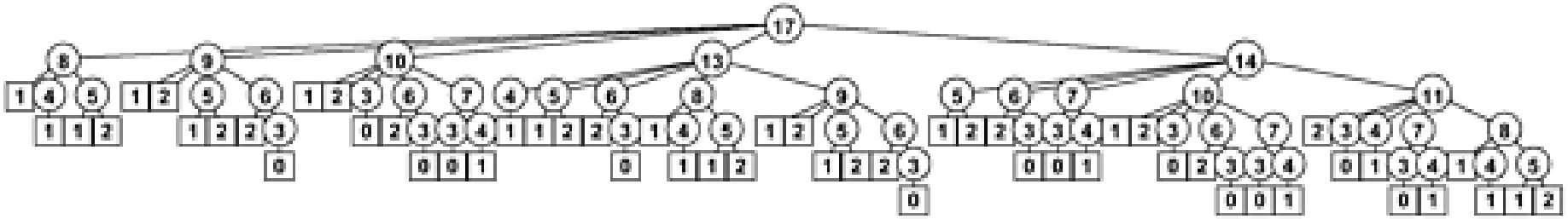
- Assume that item *i* is selected to be in the knapsack
- Find what would be the optimal value assuming optimization for all remaining items
- Requires solving a smaller knapsack problem

The Knapsack problem

We have an array of `N` items of type `Item`. For each possible item, we calculate (recursively) the maximum value that we could achieve by including that item, then take the maximum of all those values.

```
static class Item { int size; int val; }

static int knap(int cap)
  { int i, space, max, t;
    for (i = 0, max = 0; i < N; i++)
      if ((space = cap-items[i].size) >= 0)
        if ((t = knap(space) + items[i].val) > max)
          max = t;
    return max;
  }
```

The Knapsack problem



The number in each node represents the remaining capacity in the knapsack.

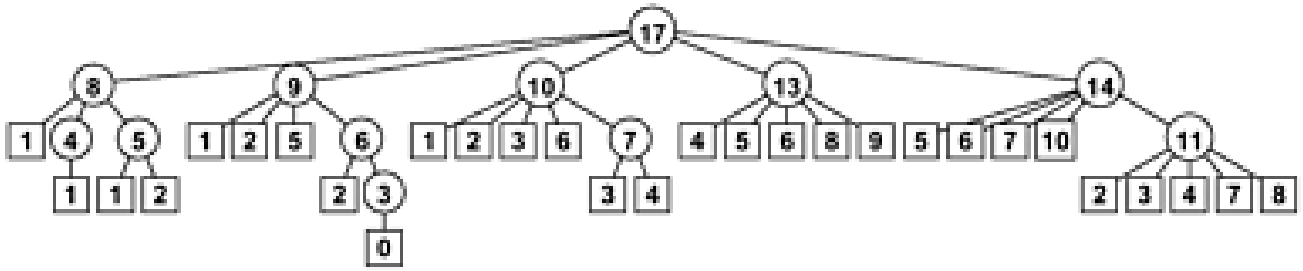
The algorithm suffers the same basic problem of exponential performance due to massive recomputation for overlapping subproblems that we considered in computing Fibonacci numbers

Exponential time !!

The Knapsack problem

```
static int knap(int cap)
  {
    int i, space, max, maxi = 0, t;
    if (maxKnown[cap] != unknown)
      return maxKnown[cap];

    for (i = 0, max = 0; i < N; i++)
      if ((space = cap-items[i].size) >= 0)
        if ((t = knap(space) + items[i].val) > max)
          { max = t; maxi = i; }

    maxKnown[cap] = max; itemKnown[cap] = items[maxi];
    return max;
 }
```

The Knapsack problem



Time consideration

- The running time is at most the running time to evaluate the function for all arguments less than or equal to the given argument

- Time ~NM (N: types of items, M: maximum total value of items)

Bottom-up and Top-down

- In bottom-up
  - We precompute all possible values
- In top-down
  - We use already computed values (on demand)
- Generally top down preferable
  - Closer to a natural solution
  - Order of computing simple
  - We may not need to solve all subproblems