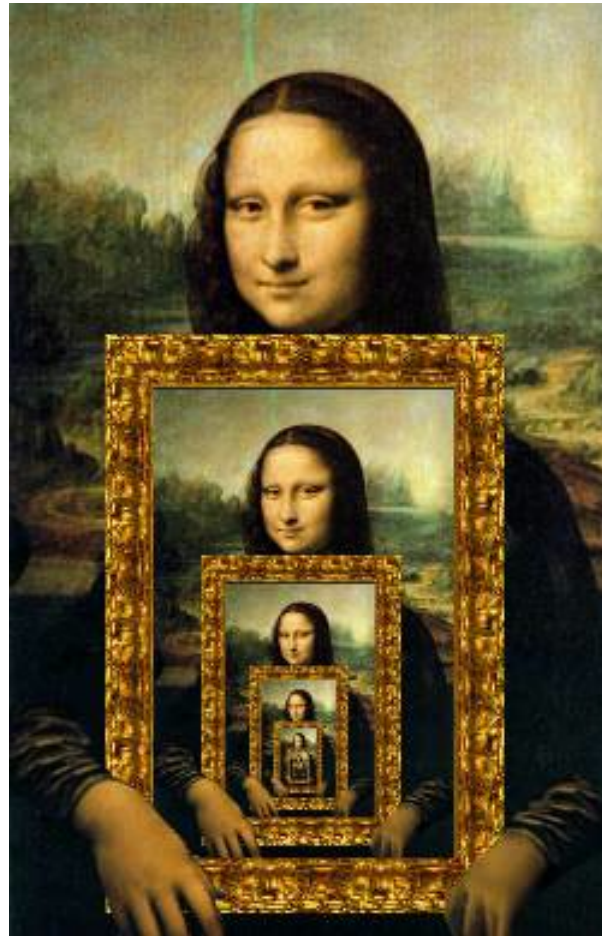


Algorithms II



Agenda

Recursion

- Mathematical induction
- Rules of recursion
- Divide and conquer
- Dynamic programming
- Backtracking

Verification of algorithms

Proof methods

(useful for algorithm verification as well as for algorithm design)

Proof by contradiction:

1. Assume the proposition is false
2. Show that this leads to a contradiction
3. Hence the proposition must be true

Theorem. There is an infinite number of primes.

Proof (Euclid):

Assume there is a *finite* number of primes: $2, 3, 5, \dots, p$.

Now define $N = (2 \cdot 3 \cdot 5 \cdots p) + 1$.

N is greater than p . But N is not exactly divisible by any of the primes (the remainder is 1). So, N must be a prime.

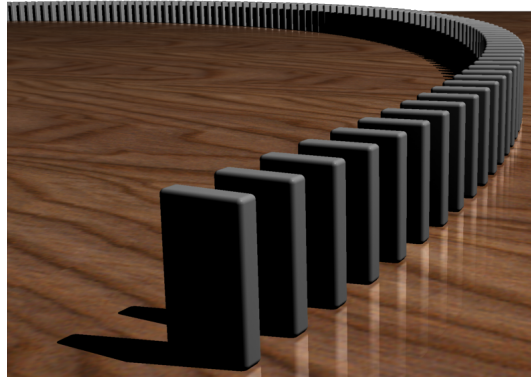
We have found a contradiction, which proves the theorem.

Mathematical induction

informal description



A long row of dominoes is standing on end:



If

(1) the first domino will fall, and

(2) whenever a domino will fall, its successor will fall

then **all** of the dominoes will fall.

Even if the row is infinitely long.

Mathematical induction

formal description



Let T be a theorem involving an integer parameter N .

T is true for all integers $N \geq c$, where c is a constant, if the following two conditions are satisfied:

1. **The base case:**

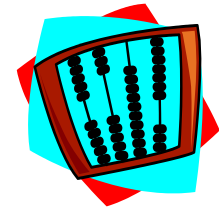
T holds for $N = c$.

2. **The induction step:**

If T holds for $N-1$, then T holds for N .

The assumption in the induction step is called the **induction hypothesis**.

Example



Theorem:

The sum $S(N)$ of the first N natural numbers is $N(N+1)/2$.

Proof:

(1) **Base case:**

For $N = 1$ we have $S(1) = 1$, which agrees with $1(1+1)/2 = 1$.

(2) **Induction step:**

Assume the theorem holds for $N-1$, that is, $S(N-1) = (N-1)N/2$.

$$S(N) = S(N-1) + N = (N-1)N/2 + N = N(N+1)/2$$

Hence the theorem also holds for N .

Money change



Theorem: Any amount of money ≥ 4 dollars may be changed using just 2 dollar and 5 dollar bills.

Proof:

(1) **Base case:**

4 dollars may be exchanged into two 2 dollar bills.

(2) **Induction step:**

Assume $N-1$ dollars may be changed. We can show that this change can be used for a change of N dollars.

Either the change contains a 5 dollar bill; or it does not.

In the first case, we replace the 5 dollar bill with three 2 dollar bills.

In the second case we replace two 2 dollar bills with a 5 dollar bill.

Strong induction



Let T be a theorem involving an integer parameter N .

T is true for all integers $N \geq c$, where c is a constant, if the following two conditions are satisfied:

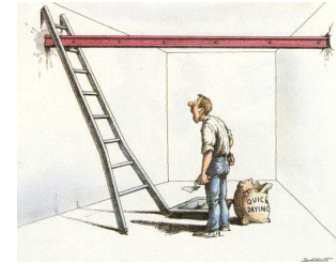
1. **The base case:**

T holds for $N = c$.

2. **The induction step:**

If T holds for any k , $c \leq k < N$, then T holds for N .

Induction can be used in algorithm design



Induction can be used constructively. Solution of small problems are used for solving larger problems.

- (1) Start with an instance of the problem.
- (2) Try to solve this problem assuming that the same problem – but of smaller size – has been solved.

Algorithm design example

Sorting N numbers in increasing order



Assume that we can sort $N-1$ numbers.

Then we can sort N numbers as follows:

Sort $N-1$ of the numbers and insert the N 'th number at its right position (*Sorting by insertion*)

or

Find the smallest of the N numbers, sort the remaining $N-1$ numbers, and append the sorted sequence to the first number (*Sorting by selection*)

The maximum contiguous subsequence sum problem

Given (possible negative) integers A_1, A_2, \dots, A_N , find the maximum value of

$$\sum_{k=i}^j A_k$$

The maximum contiguous subsequence sum is zero if all the integers are negative.

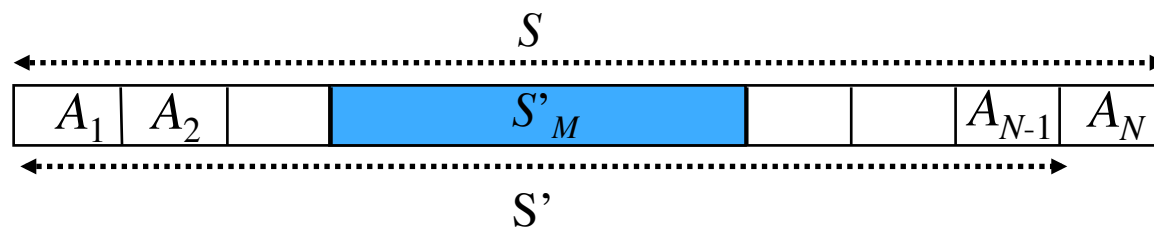
Example. If the input is $(-2, 11, -4, -1, 13, -5, 2)$, then the answer is 19, which represents the sum of the contiguous subsequence $(11, -4, -1, 13)$.

Base case: If $N = 1$, it is easy to find the maximum subsequence:
If the number is not negative, it just consists of the number.
Otherwise, it is empty.

Induction hypothesis: We know how to find the maximum contiguous subsequence sum for a sequence of length $N-1$.

Let $S = (A_1, A_2, \dots, A_N)$ and $S' = (A_1, A_2, \dots, A_{N-1})$.

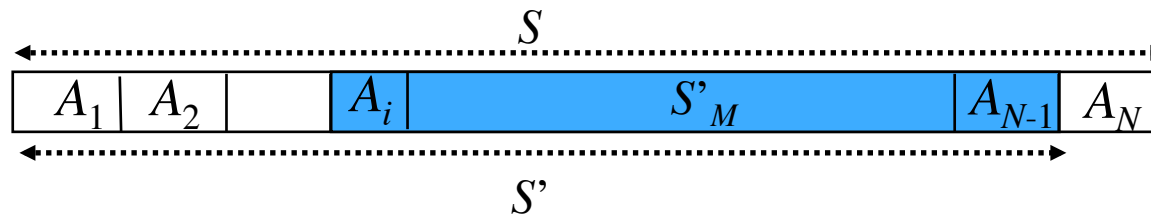
Let S'_M be the maximum subsequence for S' .

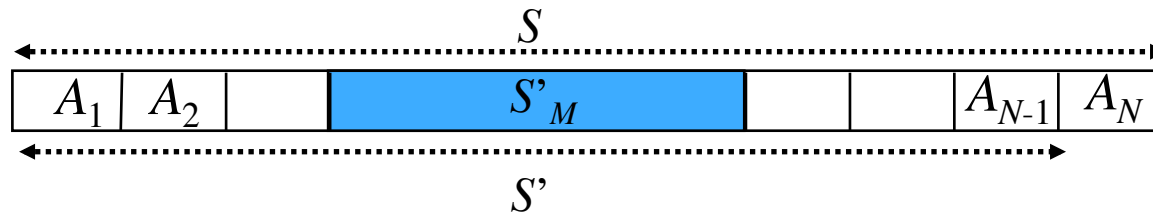


Suppose S'_M is empty. Then the maximum subsequence for S is also empty if A_N is negative; otherwise it is equal to (A_N) .

Suppose S'_M is not empty, that is, $S'_M = (A_i, A_{i+1}, \dots, A_j)$, for $1 \leq i \leq j \leq N-1$.

If $j = N-1$, then S'_M is extended with A_N if and only if A_N is positive.





If $j < N-1$, there are two cases:

- (1) Either S'_M is also maximal for S , or
- (2) there is another subsequence that is not maximal for S' , but is maximal for S when A_N is added.

Which of the two cases we have cannot be determined from the available information: S'_M .

However, since A_N may only extend a sequence that ends in A_{N-1} , the decision may be made if know the maximum **suffix** $(A_i, A_{i+1}, \dots, A_{N-1})$ for S' .

The induction hypothesis must be strengthened

Stronger induction hypothesis: We know how to find the maximum contiguous subsequence sum **and the maximum suffix sum** for a sequence of length $N-1$.

This leads to the linear algorithm

```
maxSum = maxSuffixSum = 0;
for (i = 1; i <= N; i++) {
    maxSuffixSum += A[i];
    if (maxSuffixSum > maxSum)
        maxSum = maxSuffixSum;
    else if (maxSuffixSum < 0)
        maxSuffixSum = 0;
}
```

Recursion



- **Recursive definition** of X : X is defined in terms of itself.
- Recursion is useful when a **general** version of X can be defined in terms of **simpler** versions of X .
- A problem is **solved recursively** by
 - (1) **decomposing** it into smaller problems of the *same type* as the original problem,
 - (2) repeat this process until all sub-problems are so simple that they can be solved easily, and
 - (3) **combine** the solutions of the sub-problems to obtain a solution of the original problem.

Recursive evaluation of the sum of the first N integers

$$S(1) = 1$$
$$S(N) = S(N-1) + N \text{ for } N > 1$$

figure 7.1

Recursive evaluation
of the sum of the first
 N integers

```
1 // Evaluate the sum of the first n integers
2 public static long s( int n )
3 {
4     if( n == 1 )
5         return 1;
6     else
7         return s( n - 1 ) + n;
8 }
```

Printing a number in decimal form

figure 7.2

A recursive routine for printing N in decimal form

```
1 // Print n in base 10, recursively.
2 // Precondition: n >= 0.
3 public static void printDecimal( long n )
4 {
5     if( n >= 10 )
6         printDecimal( n / 10 );
7     System.out.print( (char) ('0' + ( n % 10 ) ) );
8 }
```

$n = 36372$

print '3637' followed by '2'

Printing a number in any base

figure 7.3

A recursive routine for printing N in any base

```
1     private static final String DIGIT_TABLE = "0123456789abcdef";
2
3     // Print n in any base, recursively.
4     // Precondition: n >= 0, base is valid.
5     public static void printInt( long n, int base )
6     {
7         if( n >= base )
8             printInt( n / base, base );
9         System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
10    }
```

```

1 public final class PrintInt
2 {
3     private static final String DIGIT_TABLE = "0123456789abcdef";
4     private static final int    MAX_BASE    = DIGIT_TABLE.length( );
5
6     // Print n in any base, recursively
7     // Precondition: n >= 0, 2 <= base <= MAX_BASE
8     private static void printIntRec( long n, int base )
9     {
10        if( n >= base )
11            printIntRec( n / base, base );
12        System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
13    }
14
15    // Driver routine
16    public static void printInt( long n, int base )
17    {
18        if( base <= 1 || base > MAX_BASE )
19            System.err.println( "Cannot print in base " + base );
20        else
21        {
22            if( n < 0 )
23            {
24                n = -n;
25                System.out.print( "-" );
26            }
27            printIntRec( n, base );
28        }
29    }
30 }

```

figure 7.4

A robust number-printing program

Implementation of recursion

A recursive method calls a **clone** of itself. That clone is simply another method with different parameters.

At any instant only one clone is active; the rest are pending.

Recursion may be handled using a **stack** (since methods return in reverse order of their invocation). Java, like other languages, implements methods by using an internal stack of *activation records*.

An *activation record* contains relevant information about the method including the values of the parameters and local variables.

Stack of activation records

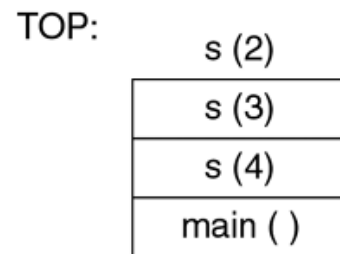
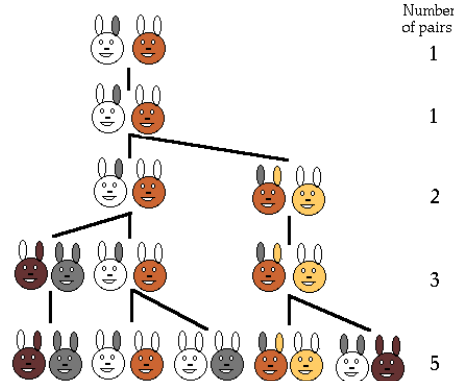


figure 7.5

A stack of activation records



Number of pairs

1

1

2

3

5

Fibonacci numbers

$$F(0) = 0$$

$$F(1) = 1$$

$$F(N) = F(N-1) + F(N-2) \text{ for } N > 1$$



Fibonacci, 1202

Beginning with a single pair of rabbits, if every month each productive pair bears a new pair, which becomes productive when they are 1 month old, how many pairs of rabbits will there be after N months?

figure 7.6

A recursive routine for Fibonacci numbers: A bad idea

```

1 // Compute the Nth Fibonacci number.
2 // Bad algorithm.
3 public static long fib( int n )
4 {
5     if( n <= 1 )
6         return n;
7     else
8         return fib( n - 1 ) + fib( n - 2 );
9 }

```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Fibonacci numbers

The recursive implementation is simple but very inefficient. By induction we can verify that the number of calls $C(N) = C(N-1) + C(N-2) + 1$ for $N \geq 3$ is equal to $F(N+2) + F(N-1) - 1$. Thus, the number of calls is larger than the number we are trying to compute. The inefficiency is due to wasted calculations.

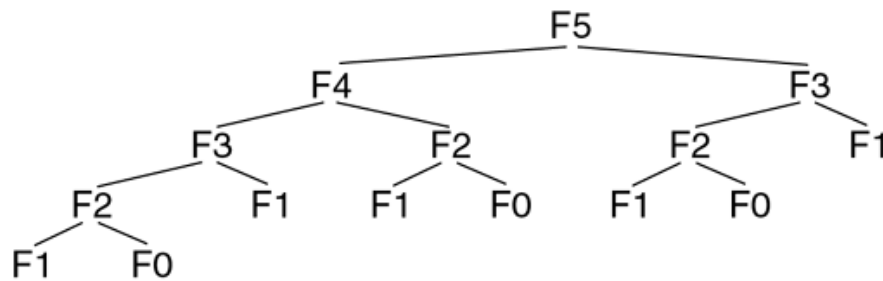


figure 7.7

A trace of the recursive calculation of the Fibonacci numbers

For $N = 40$, $F(40) = 102,334,115$, and the total number of recursive calls are more than 300,000,000.



Four basic rules of recursion

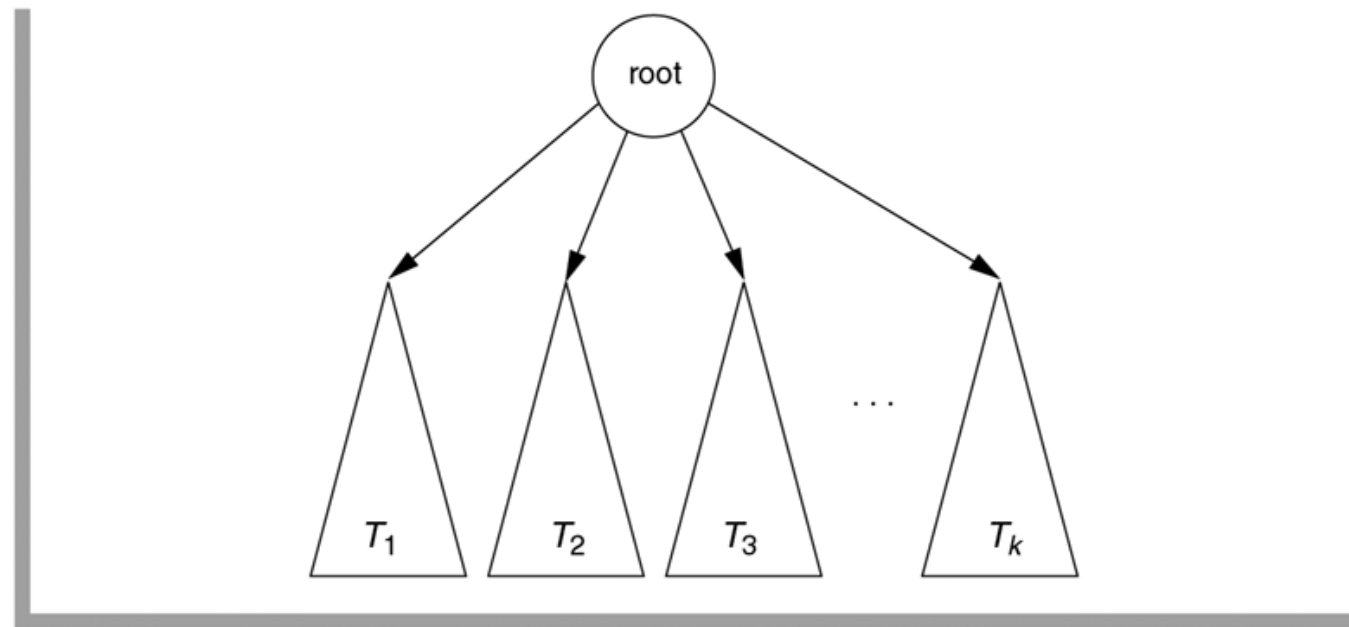
1. **Base case:** Always have at least one case that can be solved without recursion.
2. **Make progress:** Any recursive call must progress towards a base case.
3. **“You gotta believe”:** Always assume that the recursive call works.
4. **Compound interest rule:** Never duplicate work by solving the same instance of a problem in separate recursive calls.

Recursive definition of a tree

Either a tree T is empty or it consists of a root and zero or more nonempty subtrees T_1, T_2, \dots, T_k , each of whose roots are connect by an edge from the root of T .

figure 7.8

A tree viewed recursively



A tree

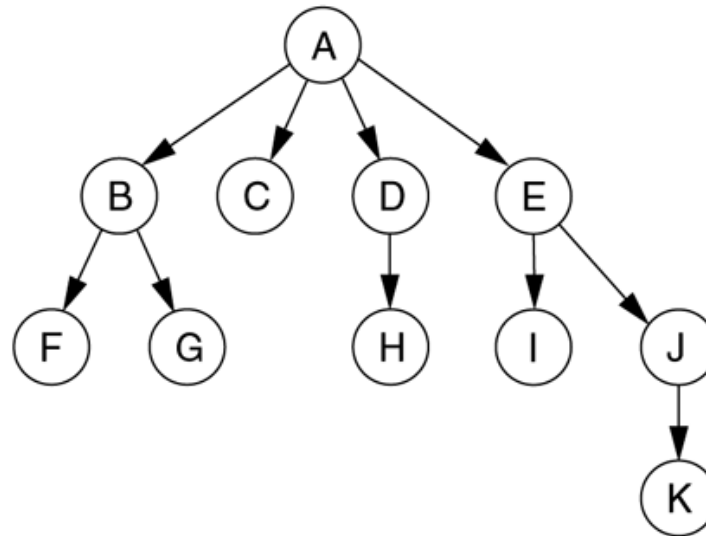
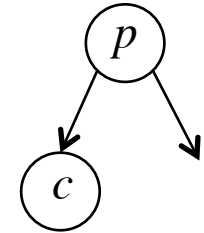


figure 7.9

A tree

Non-recursive definition of a tree



A tree consists of a set of nodes and a set of directed edges that connect pairs of nodes. A **rooted tree** has the following properties:

- One node is distinguished as the root.
- Every node c , except the root, is connected by an edge from exactly one other node p .
Node p is c 's *parent*, and c is one of p 's *children*.
- A unique path traverses from the root to each node.
The number of edges that must be followed is the *path length*.

Factorial

The product of the first N positive integers

$$1! = 1$$
$$N! = N \cdot (N-1)!$$

```
1 // Evaluate n!  
2 public static long factorial( int n )  
3 {  
4     if( n <= 1 ) // base case  
5         return 1;  
6     else  
7         return n * factorial( n - 1 );  
8 }
```

figure 7.10

Recursive
implementation of the
factorial method

```
factorial = n;  
while (--n > 1)  
    factorial *= n;
```

Iterative
implementation

Binary search

figure 7.11

A binary search routine, using recursion

```
1  /**
2  * Performs the standard binary search using two comparisons
3  * per level. This is a driver that calls the recursive method.
4  * @return index where item is found or NOT_FOUND if not found.
5  */
6  public static <AnyType extends Comparable<? super AnyType>>
7  int binarySearch( AnyType [ ] a, AnyType x )
8  {
9      return binarySearch( a, x, 0, a.length -1 );
10 }
11
12 /**
13 * Hidden recursive routine.
14 */
15 private static <AnyType extends Comparable<? super AnyType>>
16 int binarySearch( AnyType [ ] a, AnyType x, int low, int high )
17 {
18     if( low > high )
19         return NOT_FOUND;
20
21     int mid = ( low + high ) / 2;
22
23     if( a[ mid ].compareTo( x ) < 0 )
24         return binarySearch( a, x, mid + 1, high );
25     else if( a[ mid ].compareTo( x ) > 0 )
26         return binarySearch( a, x, low, mid - 1 );
27     else
28         return mid;
29 }
```

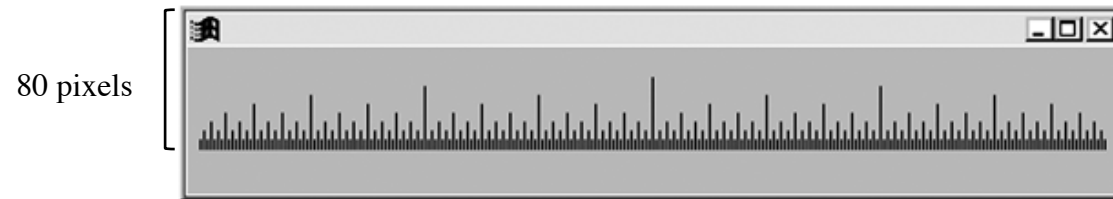


figure 7.12

A recursively drawn ruler

```
1 // Java code to draw Figure 7.12.
2 void drawRuler( Graphics g, int left, int right, int level )
3 {
4     if( level < 1 )
5         return;
6
7     int mid = ( left + right ) / 2;
8
9     g.drawLine( mid, 80, mid, 80 - level * 5 );
10
11     drawRuler( g, left, mid - 1, level - 1 );
12     drawRuler( g, mid + 1, right, level - 1 );
13 }
```

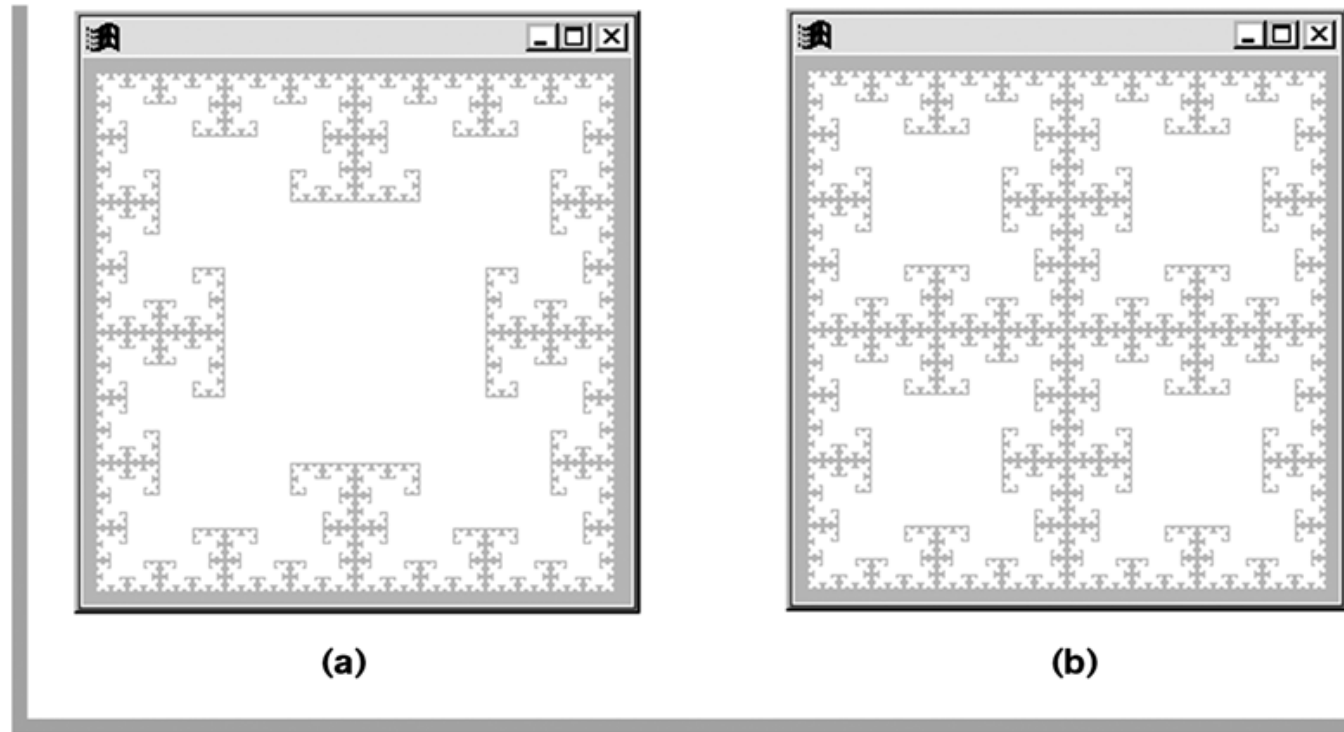
figure 7.13

A recursive method for drawing a ruler

Call: `drawRuler(g, 10, 520, 8)` using frame width = 531 and frame height = 110

figure 7.14

(a) A fractal star outline drawn by the code shown in Figure 7.15; (b) The same star immediately before the last square is added



Gray background

figure 7.15

Code for drawing the
fractal star outline
shown in Figure 7.14

```
1 // Draw picture in Figure 7.14.
2 void drawFractal( Graphics g, int xCenter,
3                 int yCenter, int boundingDim )
4 {
5     int side = boundingDim / 2;
6
7     if( side < 1 )
8         return;
9
10    // Compute corners.
11    int left = xCenter - side / 2;
12    int top = yCenter - side / 2;
13    int right = xCenter + side / 2;
14    int bottom = yCenter + side / 2;
15
16    // Recursively draw four quadrants.
17    drawFractal( g, left, top, boundingDim / 2 );
18    drawFractal( g, left, bottom, boundingDim / 2 );
19    drawFractal( g, right, top, boundingDim / 2 );
20    drawFractal( g, right, bottom, boundingDim / 2 );
21
22    // Draw central square, overlapping quadrants.
23    g.fillRect( left, top, right - left, bottom - top );
24 }
```

```
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Color;

public class FractalStar extends Frame {
    private static final int theSize = 256;

    public void paint(Graphics g) {
        setBackground(Color.gray);
        g.setColor(Color.white);
        drawSFractal(g, theSize / 2 + 10, theSize / 2 + 30, theSize);
    }

    private void drawFractal(Graphics g, int xCenter, int yCenter,
        int boundingDim) { ... }

    public static void main(String[ ] args ) {
        Frame f = new FractalStar( );
        f.setSize(theSize + 20, theSize + 40);
        f.setVisible(true);
    }
}
```

The RSA cryptosystem

(Rivest, Shamir og Adleman, 1977)



Problem: Alice wants to send a secret message to Bob in such a way that only Bob can read it.

Solution: Bob publishes two numbers, e and N , that people should use when sending him messages.

Encryption: Alice sends a message M as the number $R = M^e \pmod{N}$.

Decryption: Bob obtains the original message by computing $R^d \pmod{N}$, where d is a number that only Bob knows.

Computing the RSA constants

Determination of e , d and N :

- 1) Choose two large primes p and q .
Typically, these would be 100 digits or so each.
- 2) Compute $N = pq$.
- 3) Compute $N' = (p-1)(q-1)$.
- 4) Choose $e > 1$ such that $\gcd(e, N') = 1$.
- 5) Choose d such that $ed \pmod{N'} = 1$.
[i.e., such that d is *multiplicative inverse* to $e \pmod{N'}$].

It can be shown that $(M^e)^d = M \pmod{N}$ for any message M .

Bob must destroy p , q , and N' . He tells anybody who wants to send him a message the values of e and N , but he keeps d secret.

An example

- (1) $p = 47$ and $q = 79$ (two primes)
- (2) $N = pq = \underline{3713}$
- (3) $N' = (p-1)(q-1) = 3588$
- (4) $e = \underline{37}$ ($\gcd(e, N') = 1$)
- (5) $d = \underline{97}$ ($ed \pmod{N'} = 1$, since $ed = 3589$)

Example continued

($e = 37$, $d = 97$, $N = 3713$)

Message: **ATTACK AT DAWN**

Coding: A = 01, B = 02, C = 03, D = 04, ...

ATTACK AT DAWN
0120200103110001200004012314 (blocks of two characters)

Encryption using the public key 37:

0120³⁷ = **1404** (mod 3713) **2001**³⁷ = **2392** (mod 3713) ...
1404239235360001328422802235

Decryption using the secret key 97:

1404⁹⁷ = **0120** (mod 3713) **2392**⁹⁷ = **2001** (mod 3713) ...
0120200103110001200004012314

Security of the RSA cryptosystem

If d can be determined using the knowledge of e and N , the security of the system is compromised.

If N can be factorized, $N = pq$, then d can be reconstructed.

The caveat is that factorization is very hard to do for large numbers. Using today's technology it will take millions of years for a computer to factorize a number of 200 digits.

Algorithms

- (1) *Modular exponentiation* (M^e and R^d) ←
- (2) *Primality testing* (p and q must be primes)
- (3) *Multiplication of long integers* $((p-1)(q-1))$
- (4) *Greatest common divisor* ($\gcd(e, N')$) ←
- (5) *Multiplicative inverse* ($ed \pmod{N'} = 1$) ←

Exponentiation

Compute x^n where x is an integer, and n is a non-negative integer.

Simple algorithm:

```
power = 1;
for (int i = 1; i <= n; i++)
    power *= x;
```

Number of multiplications: n .

Exponentiation

Efficient algorithm

If n is even, then

$$x^n = (x \cdot x)^{\frac{n}{2}}$$

If n is odd, then

$$x^n = x \cdot x^{n-1} = x \cdot (x \cdot x)^{\lfloor \frac{n}{2} \rfloor}$$

```
public static long power(long x, int n) {
    if (n == 0)
        return 1;
    long tmp = power(x * x, n / 2);
    if (n % 2 != 0)
        tmp *= x;
    return tmp;
}
```

Number of multiplications $< 2 \log_2 n$.

Modular exponentiation

```
1  /**
2  * Return x^n (mod p)
3  * Assumes x, n >= 0, p > 0, x < p, 0^0 = 1
4  * Overflow may occur if p > 31 bits.
5  */
6  public static long power( long x, long n, long p )
7  {
8      if( n == 0 )
9          return 1;
10
11     long tmp = power( ( x * x ) % p, n / 2, p );
12
13     if( n % 2 != 0 )
14         tmp = ( tmp * x ) % p;
15
16     return tmp;
17 }
```

figure 7.16

Modular
exponentiation routine

Greatest common divisor

Recursive version of Euclid's algorithm

figure 7.17

Computation of
greatest common
divisor

```
1  /**
2  * Return the greatest common divisor.
3  */
4  public static long gcd( long a, long b )
5  {
6      if( b == 0 )
7          return a;
8      else
9          return gcd( b, a % b );
10 }
```

```
return b == 0 ? a : gcd(b, a % b);
```

Extended Euclid's algorithm

Given two integers a and b , the *extended Euclid's algorithm*, *fullGcd*, computes their greatest common divisor, d , as well as integers x and y such that $d = ax + by$.

Example: $a = 13, b = 17, d = 1, x = 4, y = -3$

Base case: If $b = 0$, then $d = a, x = 1$, and $y = 0$.

Induction hypothesis: We know how to compute d, x and y for $fullGcd(b, a \bmod b)$.

Let d_1, x_1 and y_1 denote the values computed by $fullGcd(b, a \bmod b)$.

$$d_1 = bx_1 + (a \bmod b)y_1 = bx_1 + (a - \left\lfloor \frac{a}{b} \right\rfloor b)y_1 = ay_1 + b(x_1 - \left\lfloor \frac{a}{b} \right\rfloor y_1)$$

Thus, for $fullGcd(a, b)$ we have

$$d = d_1, x = y_1, y = x_1 - \left\lfloor \frac{a}{b} \right\rfloor y_1$$

Implementation of Extended Euclid's algorithm

```
long[] fullGcd(long a, long b) { // returns {d, x, y}
    if (b == 0)
        return new long[] {a, 1, 0};
    long[] t = fullGcd(b, a % b);
    return new long[] {t[0], t[2], t[1] - (a / b) * t[2]};
}
```

Multiplicative inverse

The solution $1 \leq x \leq n$ to the equation

$$ax \equiv 1 \pmod{n}$$

is called the *multiplicative inverse* of a , mod n .

A call of *fullGcd*(a, n) returns d, x , and y , such that $d = ax + ny$, and d is the greatest common divisor of a and n .

If $d = 1$, then x must be the multiplicative inverse of a , mod n .

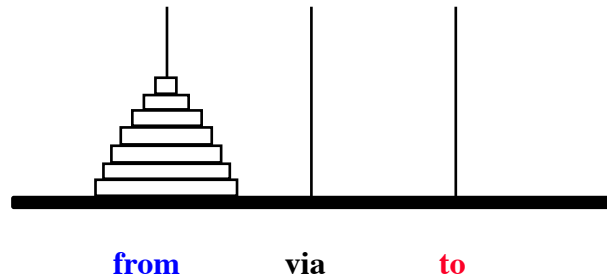
If $d \neq 1$, then a has no multiplicative inverse, mod n .

figure 7.18

A routine for
determining
multiplicative inverse

```
1 // Internal variables for fullGcd
2 private static long x;
3 private static long y;
4
5 /**
6  * Works back through Euclid's algorithm to find
7  * x and y such that if gcd(a,b) = 1,
8  * ax + by = 1.
9  */
10 private static void fullGcd( long a, long b )
11 {
12     long x1, y1;
13
14     if( b == 0 )
15     {
16         x = 1;
17         y = 0;
18     }
19     else
20     {
21         fullGcd( b, a % b );
22         x1 = x; y1 = y;
23         x = y1;
24         y = x1 - ( a / b ) * y1;
25     }
26 }
27
28 /**
29  * Solve ax == 1 (mod n), assuming gcd( a, n ) = 1.
30  * @return x.
31  */
32 public static long inverse( long a, long n )
33 {
34     fullGcd( a, n );
35     return x > 0 ? x : x + n;
36 }
```

Towers of Hanoi



Problem. Move the disks on peg **from** to peg **to**. Only one disk may be moved at a time, and no disk may be placed on top of a smaller disk.

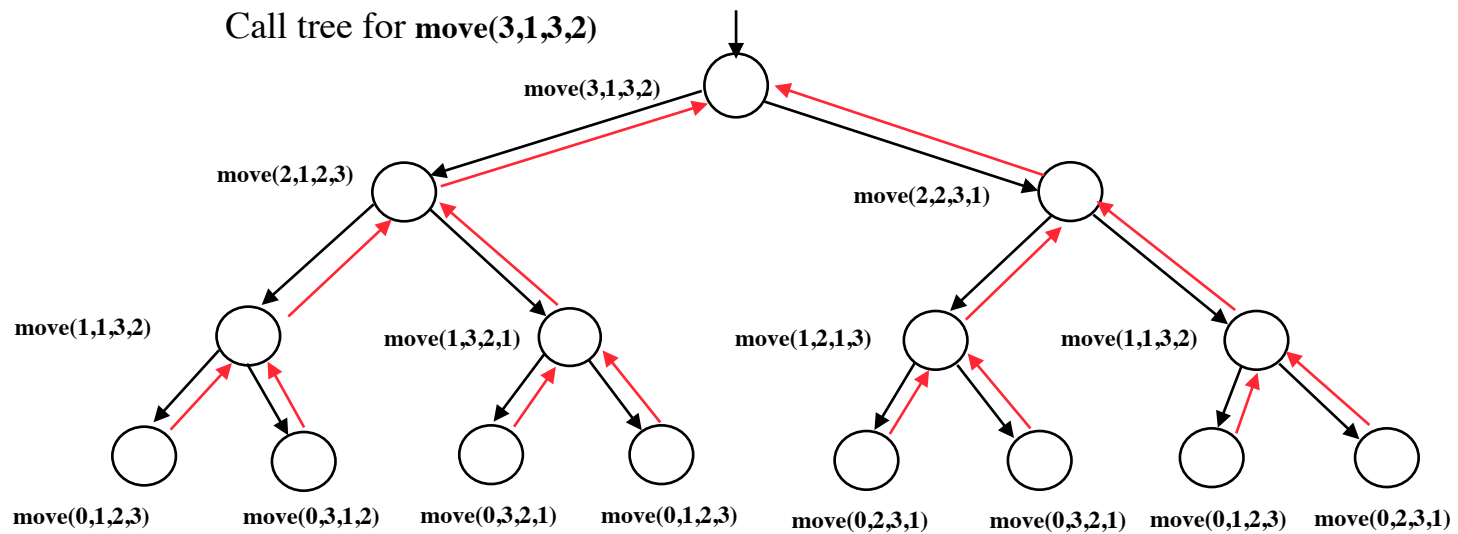
To move the n disks from peg **from** to peg **to**:

1. move the uppermost $n-1$ disks from peg **from** to peg **via**
2. move the bottom disk from peg **from** to peg **to**
3. move the $n-1$ disks from peg **via** to peg **to**.

Implementation

```
void move(int n, int from, int to, int via) {  
    if (n == 0)  
        return;  
    move(n - 1, from, via, to);  
    System.out.println("Move " + from + " to " + to);  
    move(n - 1, via, to, from);  
}
```

Call tree



Complexity

The **time complexity** is proportional to the number of moves, $M(N)$, where N is the number of disks.

$$M(N) = M(N-1) + 1 + M(N-1) = 2M(N-1) + 1, \text{ for } N > 1$$
$$M(1) = 1$$

which has the solution $M(N) = 2^N - 1$.

The **space complexity** is proportional to the maximum number of unfinished calls of move, that is $O(N)$.

The total time needed for 64 disks, given that each move takes one second, is

$$2^{64} \text{ seconds} \approx 10^{19} \text{ seconds} \approx 10^{12} \text{ years}$$

Divide-and-conquer



Divide-and-conquer is an important algorithm design technique. It is an example of the use of strong induction.

(1) Divide: If the input is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.

(2) Recur: Recursively solve the subproblems associated with the subsets.

(3) Conquer: Take the solutions to the subproblems and “merge” them into a solution of the original problem.

Template for divide-and-conquer

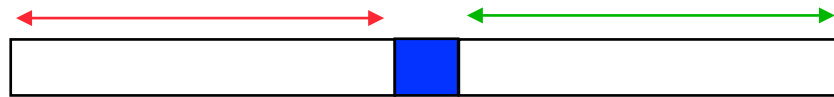
Pseudocode:

```
solve(Problem p) {  
    if (size(p) <= critical_size)  
        solve_small_problem(p);  
    else {  
        subproblem = divide(p);  
        solve(subproblem[0]);  
        solve(subproblem[1]);  
        ....  
        combine_solutions();  
    }  
}
```

Example

Computing the maximum contiguous sum

Divide the input into two halves:



Either the maximum subsequence resides

(1) entirely in the **left** part,



(2) entirely in the **right** part, or



(3) in a sequence that contains
the **center element**



Three subproblems

The first two sums are computed using recursion.

The last sum is determined as the sum of

- the maximum **suffix** for the sequence to the left of the center element (including this element), and
- the maximum **prefix** for the sequence to the right of the center element.



The maximum subsequence sum is determined as the maximum of these three values.

First Half				Second Half				
4	-3	5	-2	-1	2	6	-2	Values
4*	0	3	-2	-1	1	7*	5	Running sums

Running sum from the center (*denotes maximum for each half).

figure 7.19

Dividing the maximum contiguous subsequence problem into halves

figure 7.20

A divide-and-conquer algorithm for the maximum contiguous subsequence sum problem

```
1  /**
2  * Recursive maximum contiguous subsequence sum algorithm.
3  * Finds maximum sum in subarray spanning a[left..right].
4  * Does not attempt to maintain actual best sequence.
5  */
6  private static int maxSumRec( int [ ] a, int left, int right )
7  {
8      int maxLeftBorderSum = 0, maxRightBorderSum = 0;
9      int leftBorderSum = 0, rightBorderSum = 0;
10     int center = ( left + right ) / 2;
11
12     if( left == right ) // Base case
13         return a[ left ] > 0 ? a[ left ] : 0;
14
15     int maxLeftSum = maxSumRec( a, left, center );
16     int maxRightSum = maxSumRec( a, center + 1, right );
17
18     for( int i = center; i >= left; i-- )
19     {
20         leftBorderSum += a[ i ];
21         if( leftBorderSum > maxLeftBorderSum )
22             maxLeftBorderSum = leftBorderSum;
23     }
24
25     for( int i = center + 1; i <= right; i++ )
26     {
27         rightBorderSum += a[ i ];
28         if( rightBorderSum > maxRightBorderSum )
29             maxRightBorderSum = rightBorderSum;
30     }
31
32     return max3( maxLeftSum, maxRightSum,
33                 maxLeftBorderSum + maxRightBorderSum );
34 }
35
36 /**
37 * Driver for divide-and-conquer maximum contiguous
38 * subsequence sum algorithm.
39 */
40 public static int maxSubsequenceSum( int [ ] a )
41 {
42     return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
43 }
```

Complexity

Let $T(N)$ represent the time to solve a maximum contiguous subsequence problem of size N . Suppose N is a power of 2. Then $T(N)$ satisfies the recurrence

$$\begin{aligned}T(N) &= 2T(N/2) + O(N) \\ T(1) &= O(1)\end{aligned}$$

If $O(N)$ and $O(1)$ are replaced by N and 1, respectively, the solution is

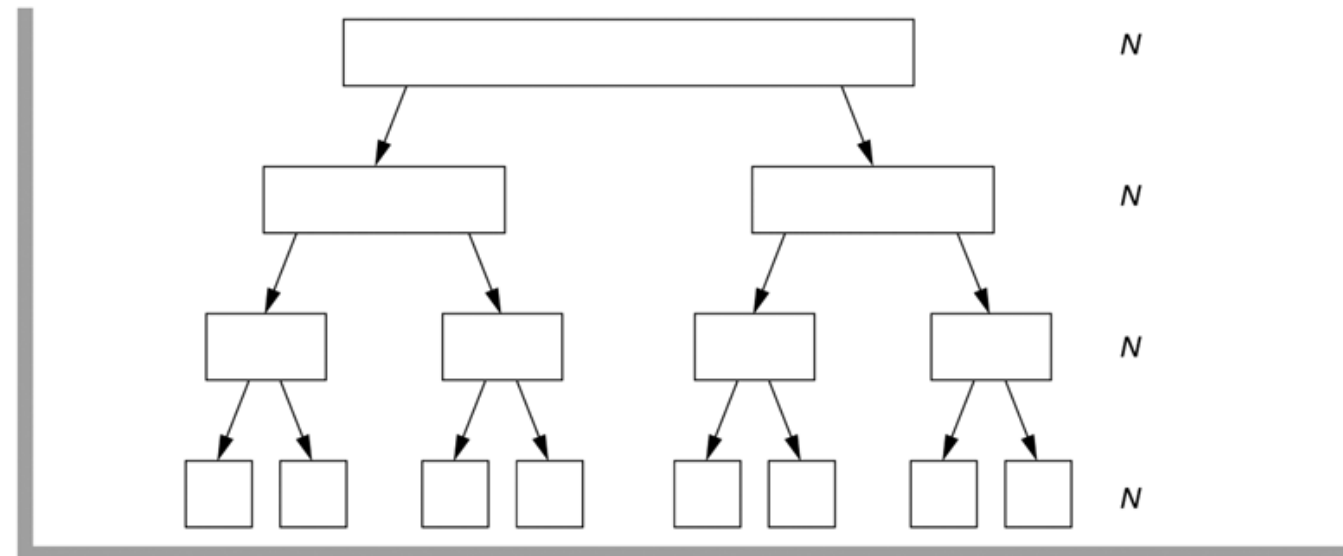
$$T(N) = N \log_2 N + N$$

Thus, the total running time is $O(N \log N)$.

Call tree

figure 7.21

Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm for $N = 8$ elements



There are about $\log_2 N$ levels

A general upper bound for divide-and-conquer algorithms

The solution to the equation $T(N) = AT(N/B) + O(N^k)$,
where $A \geq 1$ and $B > 1$, is

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{for } A > B^k \\ O(N^k \log N) & \text{for } A = B^k \\ O(N^k) & \text{for } A < B^k \end{cases}$$

Dynamic programming

Divide-and conquer (top-down):

To solve a large problem, the problem is divided into smaller problems that are solved independently.

Dynamic programming (bottom-up):

To solve a large problem, all small problems are solved, and their solutions are saved and used to solve larger problems. This process continues until the original problem has been solved.

Dynamic programming

Modern definition:

Implementation of recursive programs with overlapping subproblems. Top-down and bottom-up implementations are possible.

Top-down dynamic programming (*memoization*) means storing the results of certain calculations, which are then re-used later.

Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of calculations.

Fibonacci numbers

figure 7.6

A recursive routine for
Fibonacci numbers: A
bad idea

```
1 // Compute the Nth Fibonacci number.  
2 // Bad algorithm.  
3 public static long fib( int n )  
4 {  
5     if( n <= 1 )  
6         return n;  
7     else  
8         return fib( n - 1 ) + fib( n - 2 );  
9 }
```

Don't use this code. The same subproblems are solved many times.

Avoiding re-calculations with memoization

Maintain an array (indexed by the parameter value) containing

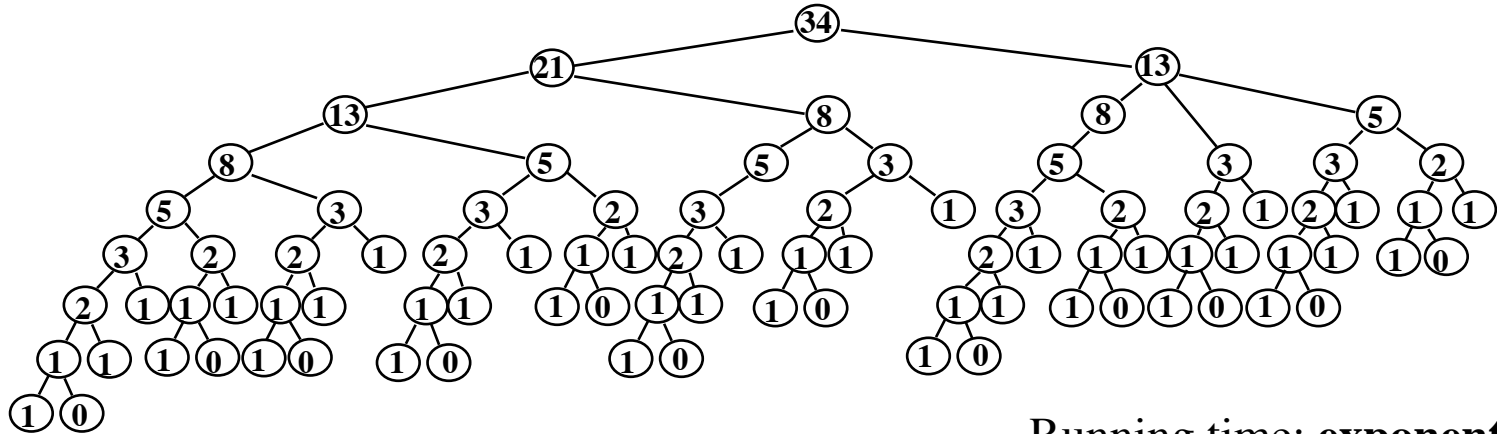
- * 0, if the recursive methods has not yet been called with this parameter value; otherwise
- * the result computed by a previous call

For the first call for a given parameter value: compute and save the result. For later calls with this parameter value: return the saved value.

```
int fib(int n) {  
    if (fibKnown[n] != 0)  
        return fibKnown[n];  
    int f = n <= 1 ? n : fib(n-1) + fib(n-2);  
    fibKnown[n] = f;  
    return f;  
}
```

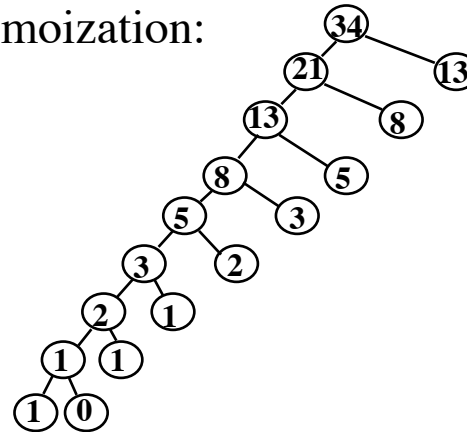
Efficiency

Simple recursive algorithm: F(9)



Running time: **exponential**

Memoization:



Running time: **linear**

Bottom-up approach

Dynamic programming (traditional):

- * Tabulate the solutions of the subproblems
- * Build the table in increasing problem size
- * Use tabulated solutions for obtaining new solutions

Example: Computation of Fibonacci numbers

```
F[0] = 0; F[1] = 1;  
for (i = 2; i <= n; i++)  
    F[i] = F[i - 1] + F[i - 2];
```

Running time is linear.

Optimal change-making



Change-making problem

For a currency with coins C_1, C_2, \dots, C_N (cents) what is the minimum number of coins needed to make K cents of change?

Example:

U.S. currency has coins in 1-, 5-, 10- and 25-cent denominations. We can make 63 cents by using two 25-cent pieces, one 10 cent piece, and three 1-cent pieces, for a total of six coins.

For U.S. coins it can be shown that a **greedy** algorithm where we repeatedly use the largest coin available always minimizes the total number of coins uses.

However, this algorithm does not work if U.S. currency included a 21-cent. The greedy algorithm would still use six coins, but the optimal solution uses three coins (all 21-cent pieces).

Top-down solution

Compute for each possible coin the minimum number of coins that can be used in a change of the remaining amount of money. Take minimum.

```
int[] coins = {1, 5, 10, 21, 25};

int makeChange(int change) {
    if (change == 0)
        return 0;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i])
            min = Math.min(min,
                1 + makeChange(change - coins[i]));
    return min;
}
```

Do not use this algorithm! Exponential time. Avoid recomputations.

Use known solutions

```
int makeChange(int change) {
    if (change == 0)
        return 0;
    if (minKnown[change] > 0)
        return minKnown[change];
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i])
            min = Math.min(min,
                1 + makeChange(change - coins[i]));
    minKnown[change] = min;
    return min;
}
```

Printing the coins used in an optimal change

Save in an array, `lastCoin`, the last coin used to make an optimal change.

```
while (change > 0) {  
    System.out.println(lastCoin[change]);  
    change -= lastCoin[change];  
}
```



```
int makeChange(int change) {
    if (change == 0)
        return 0;
    if (minKnown[change] > 0)
        return minKnown[change];
    int min = Integer.MAX_VALUE, minCoin = 0;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i]) {
            int m = 1 + makeChange(change - coins[i]);
            if (m < min)
                { min = m; minCoin = coins[i]; }
        }
    lastCoin[change] = minCoin;
    minKnown[change] = min;
    return min;
}
```

Bottom-up solution

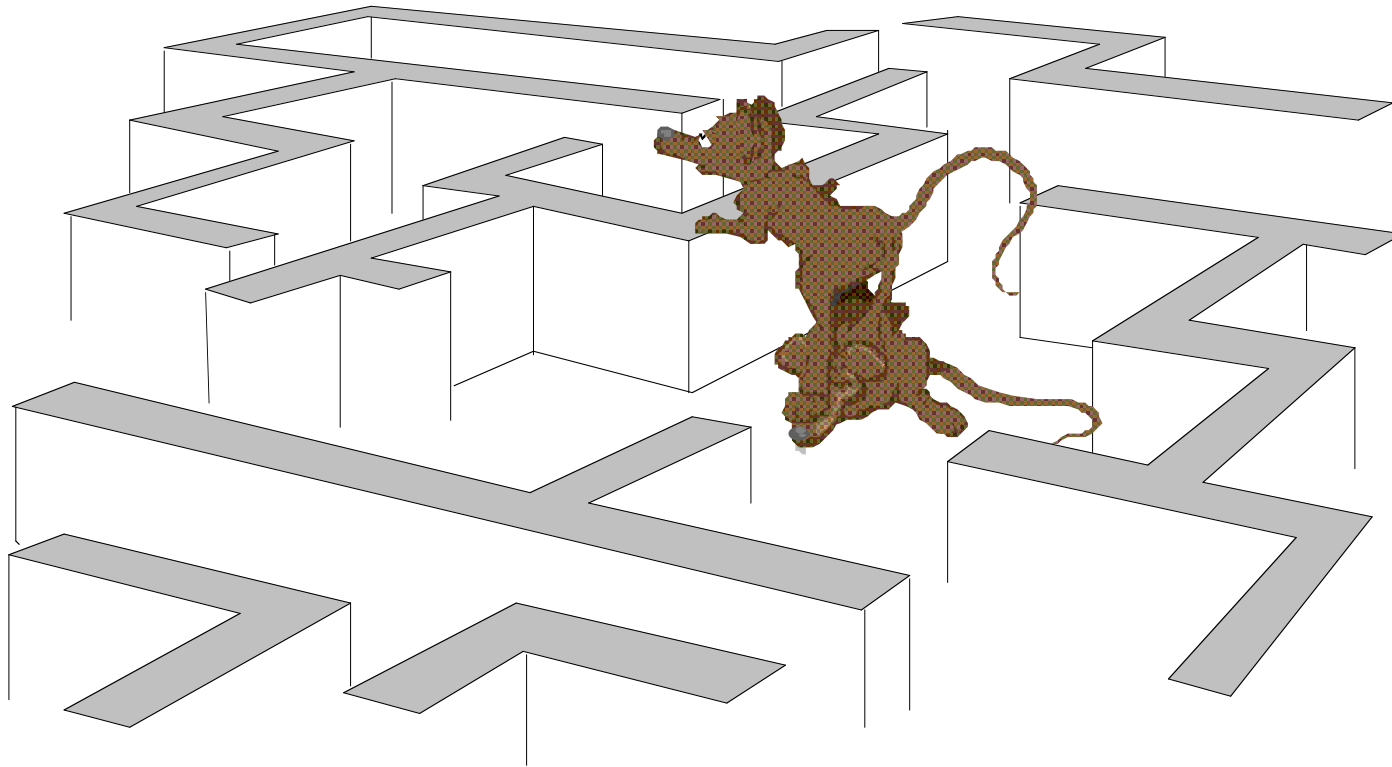
(no recursion)

Use known solutions to compute new solutions.

```
int makeChange(int change) {
    minKnown[0] = 0;
    for (int c = 1; c <= change; c++) {
        int min = Integer.MAX_VALUE;
        for (int i = 0; i < coins.length; i++)
            if (c >= coins[i])
                min = Math.min(min,
                               1 + minKnown[change - coins[i]]);
        minKnown[c] = min;
    }
    return minKnown[change];
}
```

Running time is proportional to $\text{change} * \text{coins.length}$.

Backtracking



Use recursion to try all possibilities

Search in a maze

Problem

Start →

```

WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
      W W W W W      W W      W W      W
WWWWW W      W W      W W W W      W WWWWWWWWWWW
W      WW W WWWW WWW W WWWWWW W W      W
W WWWWWWWWW W W W W      W      W WWWWWWW
W      W      W W W W W WWWWW      W      W
W WWWWWWWWWWWWW WW WW W W WWWW W WWWW W
W      W W W      WWWW W      W      W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
W      W      WWWWWWWWWWW WWWWWWWWW
W WWWWWWWWWWW WWWW WWWWW W W W WW W
W      W      W      W      W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

```

End →

Solution

Start →

```

WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
***W W W W W      W W      W W      W
WWWWW*W      W W      W W W W      W WWWWWWWWWWW
W***** WW W WWWW WWW W WWWWWW W W      W
W*WWWWWWWWW W W W W***** W*****W WWWWWWW
W*      W      W W W*W*W*WWWWW**      W
W*WWWWWWWWWWWWW WW WW*W*W*WWW** W WWWW W
W*****W*****W*W*W      *WWW W      W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
W      W      WWWWWWWWWWW*WWWWWWWWW
W WWWWWWWWWWW WWWW WWWWW W W*W***WW W
W      W      W      W      ***W*****
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

```

End →

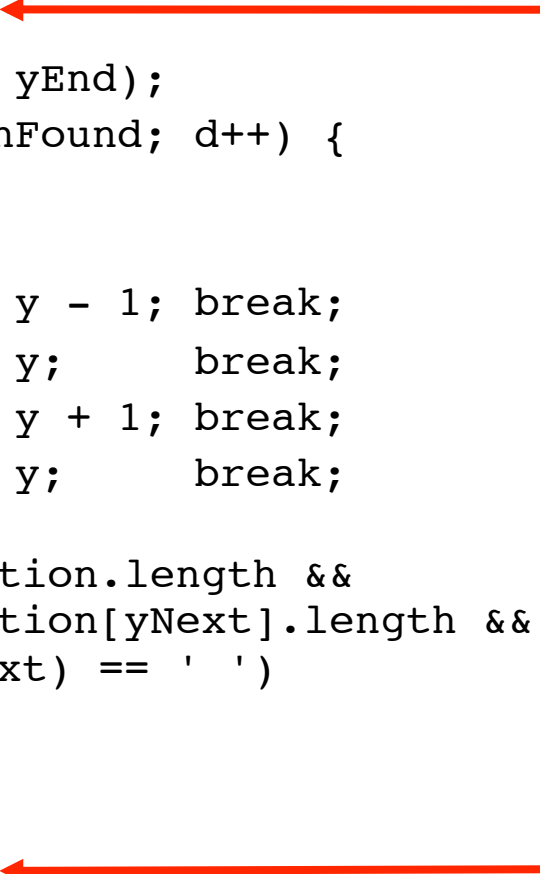
```

public class Maze {
    static String[] problem =
        {"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
         "      W W W W W      W W      W W      W",
         "XXXXXX W      W W      W W W W      W      XXXXXXXXXXXXXXXXXXXX",
         "W          WW W XXXXX WWW W XXXXXXXX W W      W",
         "W XXXXXXXXXX W W W W      W          W XXXXXXXX",
         "W          W      W W W W W W XXXXXX      W",
         "W XXXXXXXXXXXXXXXXXXXX WW WW W W XXXXX      W XXXXX W",
         "W          W W W      XXXXX W      W",
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXX W      W W      W W WW W",
         "W          W          XXXXXXXXXXXXXXX XXXXXXXXXXXXX",
         "W      XXXXXXXXXXXXXXX XXXXX XXXXXX W      W W      WW W",
         "W          W          W      W      W      ",
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"};
    static int xStart = 0, yStart = 1, xEnd = 40, yEnd = 11;
    static boolean solutionFound = false;
    static StringBuffer[] solution = new StringBuffer[problem.length];
}

```

```
public static void main(String[] args) {
    for (int x = 0; x < problem.length; x++)
        solution[x] = new StringBuffer(problem[y]);
    visit(xStart, yStart);
    if (solutionFound)
        for (int y = 0; y < solution.length; y++)
            System.out.println(solution[y]);
}
```

```
static void visit(int x, int y) {  
    solution[x].setCharAt(y, '*');  
    solutionFound = (x == xEnd && y == yEnd);  
    for (int d = 1; d <= 4 && !solutionFound; d++) {  
        int xNext = 0, yNext = 0;  
        switch(d) {  
            case 1: xNext = x;      yNext = y - 1; break;  
            case 2: xNext = x - 1; yNext = y;      break;  
            case 3: xNext = x;      yNext = y + 1; break;  
            case 4: xNext = x + 1; yNext = y;      break;  
        }  
        if (yNext >= 0 && yNext < solution.length &&  
            xNext >= 0 && xNext < solution[yNext].length &&  
            solution[yNext].charAt(xNext) == ' ' )  
            visit(xNext, yNext);  
    }  
    if (!solutionFound)  
        solution[x].setCharAt(y, ' ');  
}
```



Tic-tac-toe

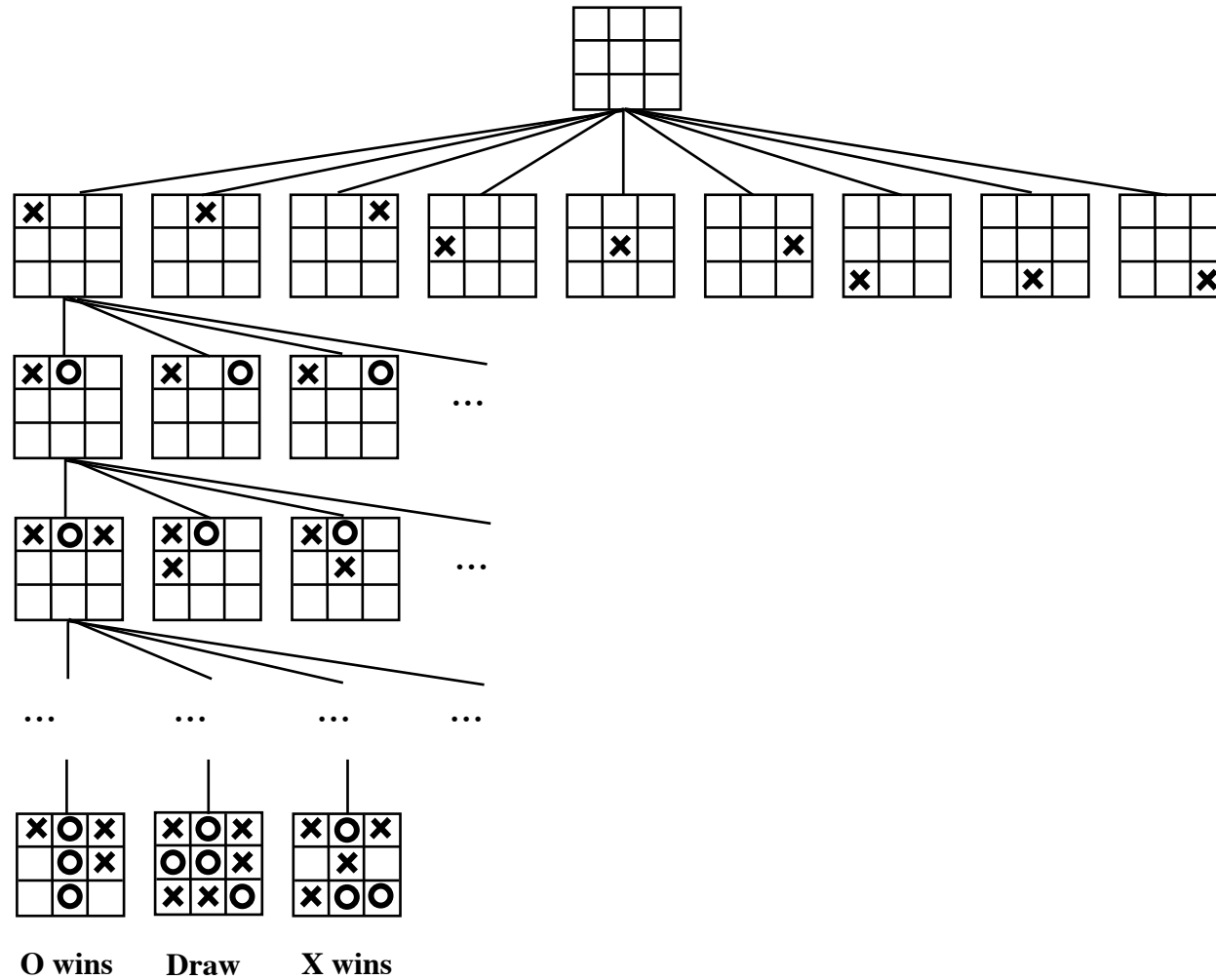


figure 7.26

Class to store an
evaluated move

```
1 final class Best
2 {
3     int row;
4     int column;
5     int val;
6
7     public Best( int v )
8         { this( v, 0, 0 ); }
9
10    public Best( int v, int r, int c )
11        { val = v; row = r; column = c; }
12 }
```

```

1 class TicTacToe
2 {
3     public static final int HUMAN      = 0;
4     public static final int COMPUTER  = 1;
5     public static final int EMPTY     = 2;
6
7     public static final int HUMAN_WIN  = 0;
8     public static final int DRAW      = 1;
9     public static final int UNCLEAR   = 2;
10    public static final int COMPUTER_WIN = 3;
11
12    // Constructor
13    public TicTacToe( )
14    { clearBoard( ); }
15
16    // Find optimal move
17    public Best chooseMove( int side )
18    { /* Implementation in Figure 7.29 */ }
19
20    // Compute static value of current position (win, draw, etc.)
21    private int positionValue( )
22    { /* Implementation in Figure 7.28 */ }
23
24    // Play move, including checking legality
25    public boolean playMove( int side, int row, int column )
26    { /* Implementation in online code */ }
27
28    // Make board empty
29    public void clearBoard( )
30    { /* Implementation in online code */ }
31
32    // Return true if board is full
33    public boolean boardIsFull( )
34    { /* Implementation in online code */ }
35
36    // Return true if board shows a win
37    public boolean isAWin( int side )
38    { /* Implementation in online code */ }
39
40    // Play a move, possibly clearing a square
41    private void place( int row, int column, int piece )
42    { board[ row ][ column ] = piece; }
43
44    // Test if a square is empty
45    private boolean squareIsEmpty( int row, int column )
46    { return board[ row ][ column ] == EMPTY; }
47
48    private int [ ] [ ] board = new int[ 3 ][ 3 ];
49 }

```

figure 7.27

Skeleton for class
TicTacToe

figure 7.28

Supporting routine for
evaluating positions

```
1 // Compute static value of current position (win, draw, etc.)
2 private int positionValue( )
3 {
4     return isAWin( COMPUTER ) ? COMPUTER_WIN :
5           isAWin( HUMAN )    ? HUMAN_WIN  :
6           boardIsFull( )     ? DRAW       : UNCLEAR;
7 }
```

```

1 // Find optimal move
2 public Best chooseMove( int side )
3 {
4     int opp;           // The other side
5     Best reply;       // Opponent's best reply
6     int dc;           // Placeholder
7     int simpleEval;   // Result of an immediate evaluation
8     int bestRow = 0;
9     int bestColumn = 0;
10    int value;
11
12    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
13        return new Best( simpleEval );
14
15    if( side == COMPUTER )
16    {
17        opp = HUMAN; value = HUMAN_WIN;
18    }
19    else
20    {
21        opp = COMPUTER; value = COMPUTER_WIN;
22    }
23
24    for( int row = 0; row < 3; row++ )
25        for( int column = 0; column < 3; column++ )
26            if( squareIsEmpty( row, column ) )
27            {
28                place( row, column, side );
29                reply = chooseMove( opp );
30                place( row, column, EMPTY );
31
32                // Update if side gets better position
33                if( side == COMPUTER && reply.val > value
34                    || side == HUMAN && reply.val < value )
35                {
36                    value = reply.val;
37                    bestRow = row; bestColumn = column;
38                }
39            }
40
41    return new Best( value, bestRow, bestColumn );
42 }

```

figure 7.29

A recursive routine for finding an optimal Tic-Tac-Toe move

The routine implements the *minimax strategy*.

For the computer, the value of a position is the *maximum* of the values of all positions that can result from making a move.

For the human, the value of a position is the *minimum* of the values of all positions that can result from making a move.



Think recursively

to obtain

- simple and precise definitions
- elegant solutions of problems that otherwise are hard to solve
- algorithms that are simple to analyze

Verification of algorithms

An algorithm A is said to be **partially correct** if the following holds:
If A terminates for a given legal input, then its output is correct.

An algorithm A is said to be **correct** (or **totally correct**) if A is partially correct and A **terminates** for any legal input.

Proof of partial correctness is made by using **assertions**.

A program **assertion** is a condition associated to a given point of the algorithm that is true each time the algorithm reaches that point.

Assertions



- **Precondition**
a condition that must always be true just prior to the execution of some section of code.
- **Postcondition**
a condition that must always be true just after the execution of some section of code.
- **Loop invariant**
a statement of the conditions that should be true on entry into a loop and that are guaranteed to remain true on every iteration of the loop.

Examples of assertions

```
{ m ≤ n } ← precondition  
sort(a, m, n);  
{ a[m] ≤ a[m+1] ≤ ... ≤ a[n] } ← postcondition
```

```
i = m;  
while  
  { a[m] ≤ a[m+1] ≤ ... ≤ a[i-1] ^  
    a[m .. i-1] ≤ a[i .. n] } ← loop invariant  
  (i < n) {  
    min = i;  
    for (j = i + 1; j ≤ n; j++)  
      if (a[j] < a[min]) min = j;  
    x = a[i]; a[i] = a[min]; a[min] = x;  
    i++;  
  }
```


Verification rules

- **Assignment:**

$$\frac{\{P_w\}}{v = w;} \quad \{w > 2\}$$
$$\{P^{w \rightarrow v}\} \quad \{v > 2\}$$

- **Selection:**

$$\frac{\{P\}}{\text{if (B)} \quad \{P \wedge B\} \\ \quad S1; \\ \text{else} \quad \{P \wedge \neg B\} \\ \quad S2;}$$

Verification example

Integer division

Given the algorithm

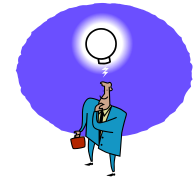
```
q = 0; r = x;
while (r >= y) {
    r = r - y;
    q = q + 1;
}
```

where x , y , q and r are integers, $x \geq 0$ and $y > 0$.

Prove that the algorithm computes the quotient q and the remainder r for the integer division of x by y , that is

$$r = x - qy \quad (0 \leq r < y)$$

The steps of the proof



- (1) The loop invariant is true at the entry of the loop
- (2) If the loop invariant is true at a given iteration it is also true at the next iteration
- (3) If the algorithm terminates, its postcondition is true
- (4) The algorithm terminates

induction

```

{ x ≥ 0, y > 0 } ← precondition of the algorithm
q = 0;
r = x;
while { r = x - qy ^ r ≥ 0 } ← loop invariant
    ( r ≥ y ) {
        { r = x - qy ^ r ≥ y } ≡ { r - y = x - (q+1)y ^ r - y ≥ 0 }
        r = r - y;
        { r = x - (q+1)y ^ r ≥ 0 }
        q = q + 1;
    }
{ r = x - qy ^ 0 ≤ r < y } ← postcondition of the algorithm

```

Termination



```
q = 0; r = x;  
while (r >= y) {  
    r = r - y;  
    q = q + 1;  
}
```

The algorithm terminates

Proof:

Since $y > 0$, each iteration will reduce r .

Thus, the loop condition will be false after a finite number of iterations.