Part IV
Other Systems: I
Java Threads

Unfortunately, Java replaced the secure monitor concept of Concurrent Pascal with insecure shortcuts.

Per Brinch Hansen

Java Threads: 1/6

- **Java has two ways to create threads:**
  - **Create a new class derived from the `Thread` class and overrides its `run()` method. This is similar to that of ThreadMentor.**
  - **Define a class that implements the `Runnable` interface.**

Java Threads: 2/6

- **Method #1**: Use the `Thread` class

```
public class HelloThread extends Thread
{
      public void run()
      {
            System.out.println("Hello World");
      }
      public static void main(String[] args)
      {
            HelloThread t = new HelloThread();
            t.start();
      }
}
```

Java Threads: 3/6

- **Method #2**: Use the `Runnable` interface defined as follows:

```
public interface Runnable
{
      public abstract void run();
}
```

Java Threads: 4/6

```
class A {
   String name;
   public A(String s) { name = s; }
   public void setName(String s) { name = s; }
   public String getName() { return name; }
}
class AA extends A implements Runnable {
   public AA(String s) { super(s); }
   public void run() {
      for (int i = 0; i < 10; i++)
         System.out.println(getName()+": Hello World");
   }
   public static void main(String[] args) {
      AA f1 = new AA("Romeo");
      Thread t1 = new Thread(f1);   t1.start();
      AA f2 = new AA("Juliet");
      Thread t2 = new Thread(f2);   t2.start();
   }
}
```

Java Threads: 5/6

```
public class Fibonacci extends Thread {
   int n, result;
   public Fibonacci(int n) { this.n = n; }
   public void run()
   {
      if ((n == 0)||(n == 1))
         result = 1;
      else {
         Fibonacci f1 = new Fibonacci(n-1);
         Fibonacci f2 = new Fibonacci(n-2);
         f1.start(); f2.start();
         try {
            f1.join(); f2.join();
         } catch (InterruptedException e) {};
         result = f1.getResult()+f2.getResult();
      }
   }
   public int getResult() { return result; }
```

Part 1/2

Java Threads: 6/6

```
    public static void main(String [] args) {
        Fibonacci f1 =
            new Fibonacci(Integer.parseInt(args[0]));
        f1.start();
        try {
            f1.join();
        } catch (InterruptedException e) {};
        System.out.println("Ans = "+f1.getResult());
    }
}
```

Part 2/2

Note that this is a very inefficient way to compute Fibonacci numbers. More on this at the end of this lecture.

The `synchronized` Keyword

- The `synchronized` keyword of a block implements mutual exclusion.

```
public class Counter{
    private int count = 0;
    public int inc()
    {
        synchronized(this)
        { return ++count; }
    }
}
```

this is a critical section

Java `ReentrantLock`: 1/2

- **A lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first.**
- **A `ReentrantLock` is like the `synchronized` keyword, but the former has an *acquisition count*.**
- **The lock holder can acquire the same lock again and the *acquisition count* is increased by one, and the lock holder must release the lock twice.**
- **Use `lock()` to acquire a lock and `unlock()` to release a lock.**

Java ReentrantLock: 2/2

- The following is a typical use of locks in Java.

```
Lock myLock = new ReentrantLock();

myLock.lock();  // acquire a lock
try {
   // in critical section now
   // catch exceptions and
   //    restore invariants if needed
} finally {
   myLock.unlock();
}
```

Java `wait()` and `notify()`: 1/8

- Java has something like the monitor concept.
- Threads that access a common object share a lock, which is usually implemented by `synchronized`.
- A thread that wants to use a common object must acquire the lock associated with that object. This is the mutual exclusion of a monitor.
- After using a common object, a thread must release (i.e., unlock) the object.
- The common object, which is usually a class, can be considered as a monitor.

Java `wait()` and `notify()`: 2/8

- Java uses `wait()` to wait on an event.
- Java uses `notify()` or `notifyAll()` to indicate that an event has happened.
- Basically, Java **does not** have the `condition` type, and a programmer must "simulate" an event.
- Java uses the Mesa type monitor.

Java `wait()` and `notify()`: 3/8

- `wait()` causes the caller to release the lock.
- `wait()` should always be wrapped in a `try` block because it throws `InterruptedException`.
- `wait()` can only be invoked by the thread that owns the lock on the object (i.e., monitor).
- The thread that calls `wait()` becomes **waiting** until it is notified.
- A released thread from `wait()` is **inactive** until it can reacquire the needed lock.
- When this thread can get the lock back (i.e., **active**) is determined by the lock mechanism.

Java `wait()` and `notify()`: 4/8

- A thread calls `notify()` to release a waiting thread or `notifyAll()` to release all waiting threads and continues its execution (the Mesa type).
- The released threads become **inactive** and wait until the object's lock will become available.
- Whether an **inactive** thread can reacquire the lock again immediately is determined by the lock mechanism.
- Using `notify()` and `notifyAll()` as the last statement can avoid many potential problems.

Java wait() and notify(): 5/8

```
public class Counter implements BoundedCounter {
   protected long count = MIN;
   public synchronized long value() { return count; }
   public synchronized long inc()
      { awaitINC(); setCount(count+1); }
   public synchronized long dec()
      { awaitDEC(); setCount(count-1); }
   protected synchronized void setCount(long newVal)
      { count = newVal; notifyAll(); }
   protected synchronized void awaitINC() {
      while (count >= MAX)
         try { wait();} catch(InterruptedException e){};
   }
   protected synchronized void awaitDEC() {
      while (count <= MIN)
         try { wait();} catch(InterruptedException e){};
   }
}
```

Java wait() and notify(): 6/8

```
public class Buffer implements BoundedBuffer {
   protected Object[] buffer;
   protected int in;
   protected int out;
   protected int count;
   public Buffer(int size)
      throws IllegalArgumentException {
         if (size <= 0)
            throw new IllegalArgumentException();
         buffer = new Object[size];
   }
   public int GetCount() { return count; }
   public int capacity() { return Buffer.length; }
   // methods put() and get()
}
```

Part 1/3

Java wait() and notify(): 7/8

```
public synchronized void put(Object x)
{
   while (count == Buffer.length)
      try { wait(); }
         catch(InterruptedException e){};
   Buffer[in] = x;
   in = (in + 1) % Buffer.length;
   if (count++ == 0)
      notifyAll();
}
```

Part 2/3

Java wait() and notify(): 8/8

```
public synchronized void  get(Object x)
{
   while (count == 0)
      try { wait(); }
         catch(InterruptedException e){};
   Object x = Buffer[out];
   Buffer[out] = null;
   out = (out + 1) % Buffer.length;
   if (count-- == Buffer.length)
      notifyAll();
   return x;
}
```

Part 3/3

Java Semaphores: 1/7

- **The** `java.util.concurrent.Semaphore` **class is a counting semaphore.**
- **To use this semaphore type, you need to do this:**

```
import  java.util.concurrent.Semaphore;
```

- **There are more on concurrent programming in** `java.util.concurrent.*`**.**

Java Semaphores: 2/7

- **The semaphore counter in a Java semaphore is referred to as `permits`.**
- **There are two constructors:**
  - `Semaphore(int permits)` **: Creates a** `Semaphore` **with the given number of** `permits` **and nonfair fairness.**
  - `Semaphore(int permits, Boolean fair)` **: Creates a** `Semaphore` **with the given number of** `permits` **and the given fairness setting (i.e.,** `TRUE` **= first-in-first-out).**

Java Semaphores: 3/7

- There are several methods in the `Semaphore` class.
- However, we only mention a few.
- The `acquire()` methods (i.e., `wait()`):
  - `acquire()` : Acquires a permit from this semaphore, blocking until one is available, or the caller is interrupted.
  - `acquire(int permits)` : Acquires the given number of `permits` from this semaphore, blocking until all permits are available, or the caller is interrupted.
- `acquire()` should always be wrapped in a `try-catch (InterruptedException)` block.

Java Semaphores: 4/7

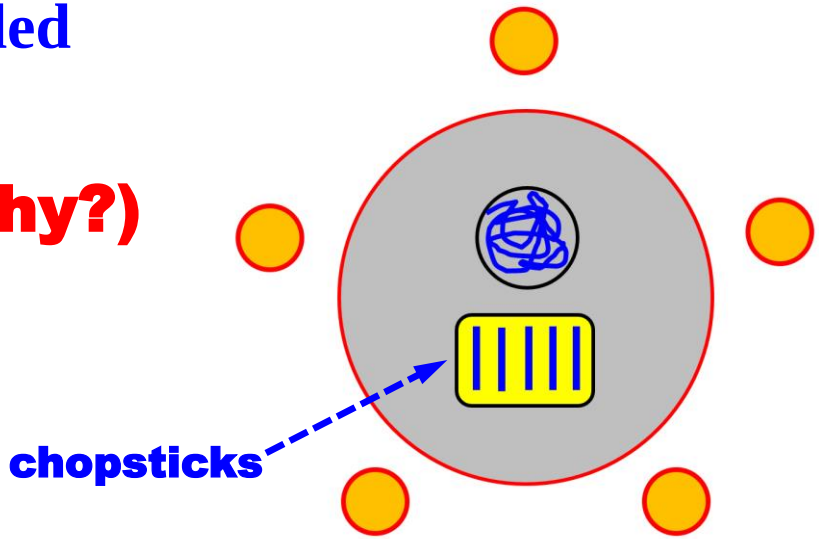
- **The `Uninterruptibly` forms of `acquire()` are available. They are similar to the conventional `wait()`.**
- **The `acquireUninterruptibly()` methods (i.e., `wait()`):**
  - **`acquireUninterruptibly()` : Acquires a permit from this semaphore, blocking until one is available~~, or the caller is interrupted~~.**
  - **`acquireUninterruptibly(int permits)` : Acquires the given number of `permits` from this semaphore, blocking until all permits are available~~, or the caller is interrupted~~.**

Java Semaphores: 5/7

- **The `release()` methods (i.e., `signal()`) are as follows:**
  - `release()` **: Releases a permit, returning it to the semaphore.**
  - `release(int permits)` **: Releases the given number of permits, returning them to the semaphore.**

Java Semaphores: 6/7

- Let us revisit the Dining Philosophers problem again with a minor twist:
  - All five chopsticks are placed at the center of the table.
  - Each philosopher must get two chopsticks to eat.
  - If a philosopher get the needed chopsticks one by one, deadlock can happen. **(Why?)**
  - Thus, each philosopher must get **BOTH** chopsticks at the same time.



Java Semaphores: 7/7

```
import  java.util.concurrent.Semaphore;

Semaphore  Chops(5);  // five chopsticks

for each philosopher

while (1) {
   // thinking
   try { Chops.acquire(2); } // getting 2 chops
      catch (InterruptedException e) { };
   // eating
   Chops.release(2);
}
```

Because hold-and-wait fails, this solution is deadlock-free

Efficient Computation of Fibonacci Numbers

Fibonacci Numbers: 1/15

- The method used on Slide 6 is very inefficient.
- How inefficient is it?
- Let f_n be the n-th Fibonacci number with $f_1 = f_2 = 1$, and t_n be the number of additions to compute f_n.
- The program on Slide 6 computes f_{n+1} with two threads, one computing f_n with t_n additions and the other computing f_{n-1} with t_{n-1} additions.
- Therefore, we have

$$t_{n+1} = \underbrace{t_n}_{\text{for } f_n} + \underbrace{t_{n-1}}_{\text{for } f_{n-1}} + \underbrace{1}_{\text{for } f_n + f_{n-1}}$$

Fibonacci Numbers: 2/15

- In fact, we have $t_n = f_n - 1$.

f_n	t_n
$f_1 = 1$	$t_1 = 0$
$f_2 = 1$	$t_2 = 0$
$f_3 = f_2 + f_1 = 2$	$t_3 = t_2 + t_1 + 1 = 0 + 0 + 1$ $= 1 = f_3 - 1$
$f_4 = f_3 + f_2 = 3$	$t_4 = t_3 + t_2 + 1 = 1 + 0 + 1$ $= 2 = f_4 - 1$
$f_5 = f_4 + f_3 = 5$	$t_5 = t_4 + t_3 + 1 = 2 + 1 + 1$ $= 4 = f_5 - 1$

Fibonacci Numbers: 3/15

- **This identity $t_n = f_n - 1$ can be proven easily with the mathematical induction method.**
- **BASE CASE: It is obvious for n = 1 or 2.**
- **INDUCTION: Now assume that $t_n = f_n - 1$ holds for 1,…, n. Because we know $t_{n+1} = t_n + t_{n-1} + 1$, $t_n = f_n - 1$ and $t_{n-1} = f_{n-1} - 1$, we have the following:**

$$
\begin{aligned}
t_{n+1} &= t_n + t_{n-1} + 1 \\
&= (f_n - 1) - (f_{n-1} - 1) + 1 \\
&= (f_n + f_{n-1}) - 1 \\
&= f_{n+1} - 1
\end{aligned}
$$

Can we obtain f_n and hence t_n explicitly?

Fibonacci Numbers: 4/15

- **Define φ and ψ as follows:**

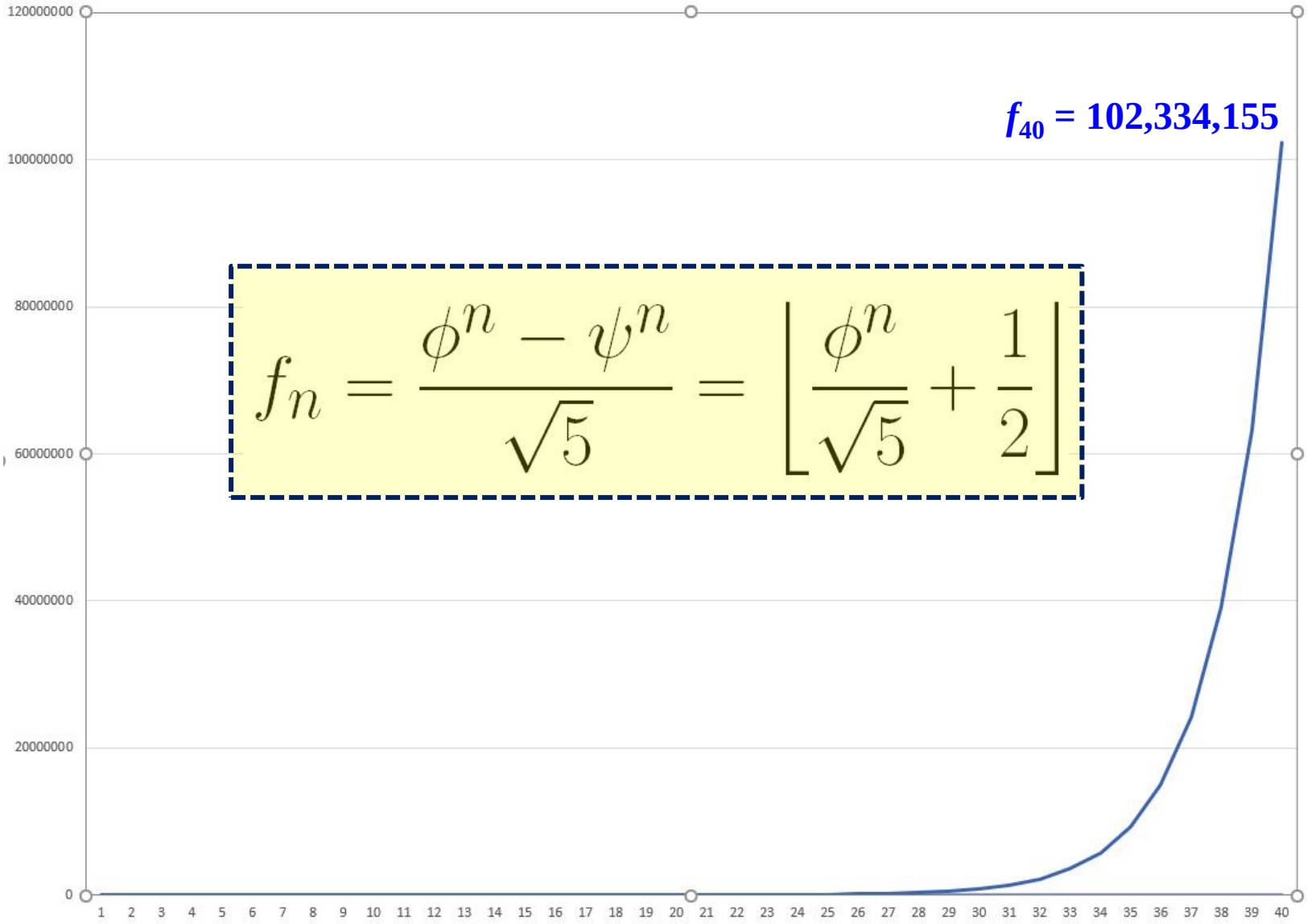
$$\phi = \frac{1+\sqrt{5}}{2} \qquad \psi = \frac{1-\sqrt{5}}{2}$$

- **Then, f_n can be computed directly as follows:**

$$f_n = \frac{\phi^n - \psi^n}{\sqrt{5}} = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

- **Thus, f_n increases exponentially. In other word, t_n is of order $O(\varphi^n)$! Thus, using the way on Slide 6 to compute Fibonacci numbers is very slow!**

Fibonacci Numbers: 5/15



$$f_n = \frac{\phi^n - \psi^n}{\sqrt{5}} = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

Fibonacci Numbers: 6/15

- You perhaps know this $O(n)$ method.

```
unsigned long Fibonacci(int n)
{
   unsigned long  previous, this, result;
   int  i;

   if (n == 0 || n == 1)
      return 1;
   else {
      previous = 0; this = 1;
      for (i = 2; i <= n; i++) {
         result   = previous + this;
         previous = this;
         this     = result;
      }
   }
   return result;
}
```

Fibonacci Numbers: 7/15

- **Can we do better (i.e., faster than $O(n)$)?**
- **YES, we need to do a bit more.**
- **Let us learn how to compute x^n fast where x and n are positive integers.**

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^k)^2 & \text{if } n = 2k \text{ (even)} \\ x \cdot x^{2k} & \text{if } n = 2k+1 \text{ (odd)} \end{cases}$$

if the exponent is even, compute $x^{n/2}$ and square the result

if the exponent is odd, compute x^{n-1} (even) and multiply x to the result

Fibonacci Numbers: 8/15

- The following is a possible implementation:

```
unsigned long Recursive_Power(unsigned long x, unsigned long n)
{
    unsigned long temp;

    if (n == 0)                          // x^0 = 1
        return 1;
    else if (n & 0x01UL == 0) {  // if n is even
        temp = Recursive_Power(x, n >> 1);
        return temp*temp;
    }
    else                                 // if n is odd
        return x * Recursive_Power(x, n-1);
}
```

is the last bit of n 0?

dividing n by 2 using right shit (1 bit)

n-1 is even

This implementation is $O(\log_2(n))$ because of this !

Fibonacci Numbers: 9/15

- **The following expression is crucial:**

$$\begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

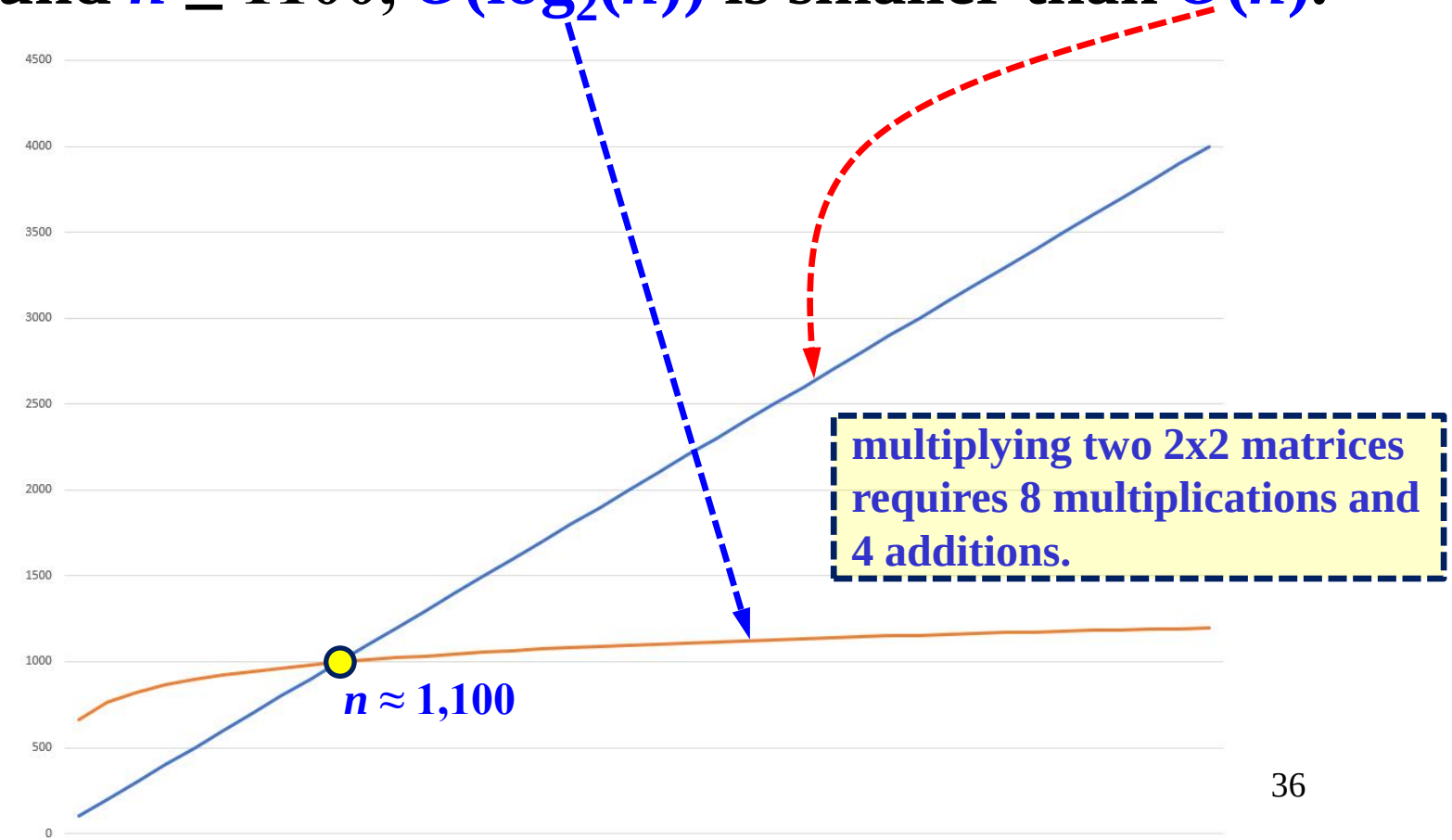
- **Therefore, we have the following:**

$$\begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- **Hence, computing f_{n+1} is equivalent to raising a 2x2 matrix to its n-th power.**
- **Because x^n can be done in the order of $O(\log_2(n))$, raising a 2x2 matrix to its n-th power is also $O(\log_2(n))$.**

Fibonacci Numbers: 10/15

- **Assume that each recursion requires 100-fold of the time of executing a single addition.**
- **Around $n \geq 1100$, $O(\log_2(n))$ is smaller than $O(n)$.**



Fibonacci Numbers: 11/15

- Can we remove recursion?
- **YES**, it is a little tricky but not difficult!
- We need to convert the n in x^n to binary.
- Consider $45_{10} = 101101_2$ and x^{45}.

$$
\begin{aligned}
x^{45} &= x^{101101_2} \\
&= x^{1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0} \\
&= \left(x^{1\cdot 2^5}\right)\cdot\left(x^{0\cdot 2^4}\right)\cdot\left(x^{1\cdot 2^3}\right)\cdot\left(x^{1\cdot 2^2}\right)\cdot\left(x^{0\cdot 2^1}\right)\cdot\left(x^{1\cdot 2^0}\right) \\
&= \left(x^{2^5}\right)\cdot\boxed{(1)}\cdot\left(x^{2^3}\right)\cdot\left(x^{2^2}\right)\cdot\boxed{(1)}\cdot\left(x^{2^0}\right) \\
&= \left(x^{2^5}\right)\cdot\left(x^{2^3}\right)\cdot\left(x^{2^2}\right)\cdot\left(x^{2^0}\right)
\end{aligned}
$$

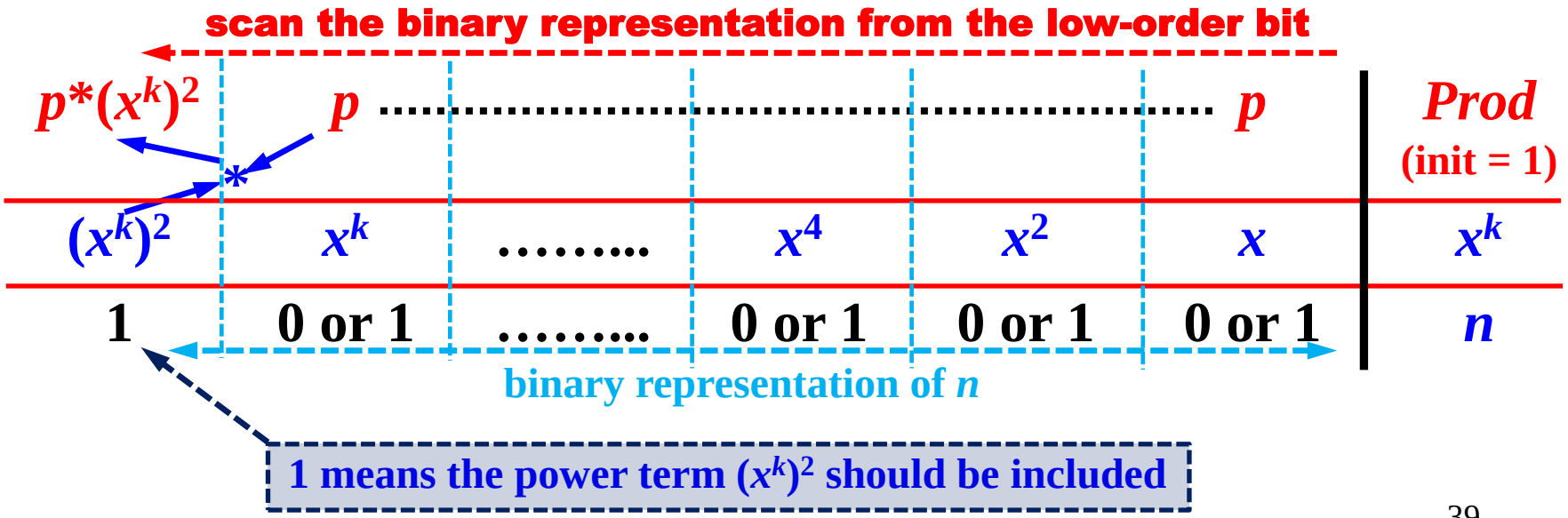
Fibonacci Numbers: 12/15

- **The binary representation of the *n* in x^n provides selectors indicating which exponents should be included: 1 – YES and 0 – NO.**
- **In $45_{10} = 101101_2$, 2^5, 2^3, 2^2 and 2^0 are included.**

$$
\begin{aligned}
x^{45} &= x^{101101_2} \\
&= x^{1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0} \\
&= \left(x^{1\cdot 2^5}\right)\cdot\left(x^{0\cdot 2^4}\right)\cdot\left(x^{1\cdot 2^3}\right)\cdot\left(x^{1\cdot 2^2}\right)\cdot\left(x^{0\cdot 2^1}\right)\cdot\left(x^{1\cdot 2^0}\right) \\
&= \left(x^{2^5}\right)\cdot\boxed{(1)}\cdot\left(x^{2^3}\right)\cdot\left(x^{2^2}\right)\cdot\boxed{(1)}\cdot\left(x^{2^0}\right) \\
&= \left(x^{2^5}\right)\cdot\left(x^{2^3}\right)\cdot\left(x^{2^2}\right)\cdot\left(x^{2^0}\right)
\end{aligned}
$$

Fibonacci Numbers: 13/15

- We need to compute x^0, x^1, x^2, x^4, x^8, etc.
- The included x^k's are multiplied together.
- In each step, the next $(x^k)^2$ is computed.
- If this power is included, multiply it to the product.



Fibonacci Numbers: 14/15

- Here is a possible implementation. It is simple!
- If your system can detect integer overflow, this solution won't work. How would you fix it?

```
unsigned long  Power(unsigned long x, unsigned long n)
{
   unsigned long  temp, power;

   temp  = 1;
   power = x;
   while (n > 0) {
      if (n & 0x01UL == 1)     // last bit = 1?
         temp = temp * power; // include this power
      power = power * power;  // compute x^{2k}
      n >> 1;                 // shift to the right
   }
   return temp;
}
```

Fibonacci Numbers: 15/15

- **The worst case is n = $\texttt{11111}\ldots\ldots\texttt{111}_2$ (all 1s).**
- **In this case, all power terms x^k must be computed and multiplied to the result.**
- **Each iteration requires two multiplications.**
- **Because there are k bits, $2\times k$ multiplications are needed.**
- **Because k is $O(\log_2(n))$, the complexity of this algorithm is $O(\log_2(n))$ and is faster than the recursive implementation.**

The End