A: Use of arrays (space) to improve efficiency of algorithms

B: Dictionary

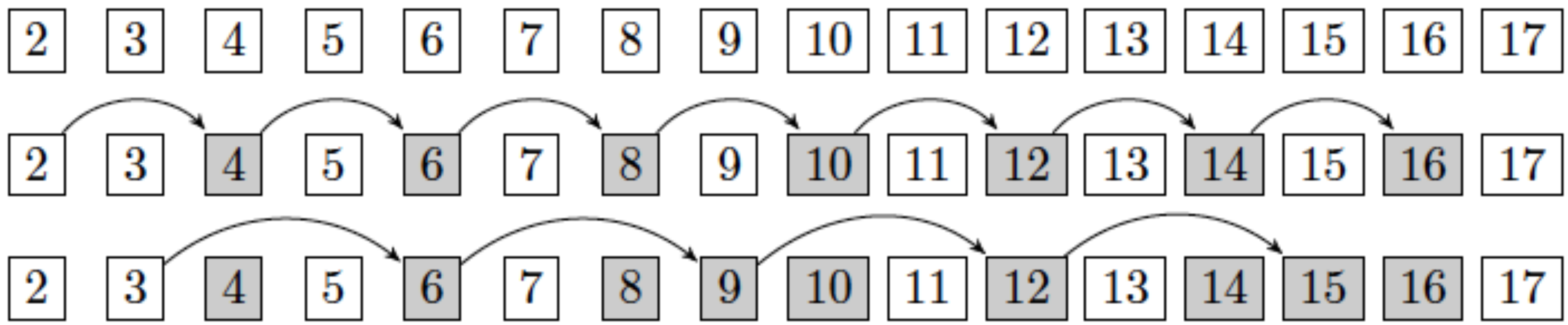
Prime Numbers:

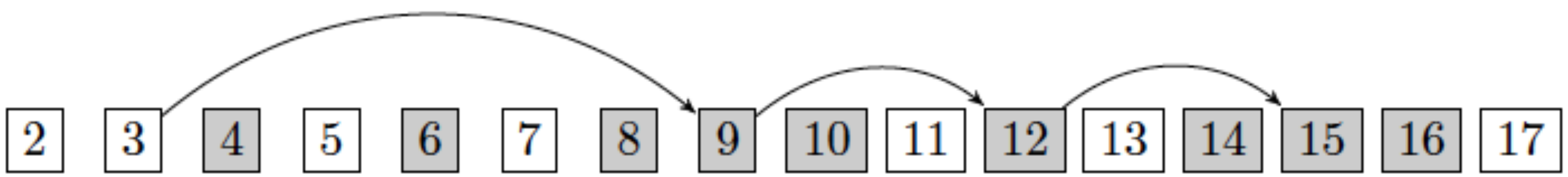
- Given n find all prime numbers from 2 to n.
- Generalises earlier strategy
- One approach is to do test_prime(n) for all numbers upto n.
  - Not efficient

Prime Numbers: Seive of Eratosthenes

- Avoid multiples of ALL smaller primes – cross (mark) them.
- Algorithm:
  - For 2 <= x <= n
    - ...

Algorithm
Seive of Eratosthenes





Algorithm
Seive of Eratosthenes

- Array of [2..n]
- Initialise:
  - All numbers UNCROSSED
  - x = 2
- WHILE (x <= n)
  - Proceed to next uncrossed number x. This is a PRIME
  - CROSS all multiples of x

Algorithm
Seive of Eratosthenes

```
def sieve(n):
   save = [True] * (n+1)
   save[0]=save[1]=False
   i = 2
   while (i*i <= n):
     if (save[i]):
       k = i*i
       while (k<=n):
         save[k] = False
         k += i
     i += 1
   return save
```

function sieve

```
n = int(input('Give n:'))
primes=sieve(n)
for i in range(n+1):
  if primes[i]:
    print(i)
```

driver program

Algorithm
Seive of Eratosthenes

- Time complexity

$$\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\ldots=\sum_{p_j\leqslant\sqrt{n}}\frac{n}{p_j}=n\cdot\sum_{p_j\leqslant\sqrt{n}}\frac{1}{p_j}$$

  – O(nloglogn) – proof is not in the scope here
- Extra space
  – Need array of size ~ n
- Can reduce extra space – segmented sieve

Fibonacci Numbers (Revisit)

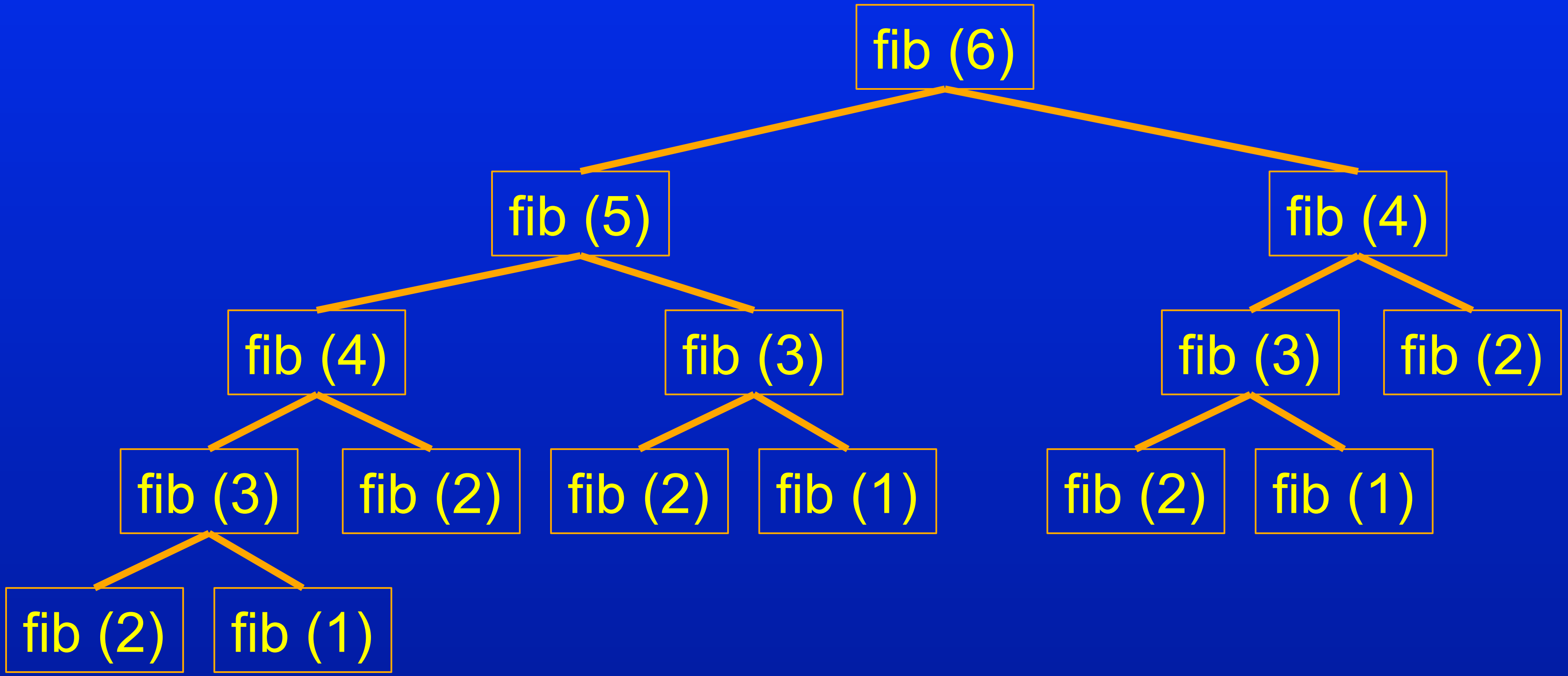
$$
\text{fib}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & n > 2 \end{cases}
$$

Recursive Algorithm

```
def fib(n):
  if (n==1):
    return 0
  if (n==2):
    return 1
  else:
    return fib(n-1)+fib(n-2)
```

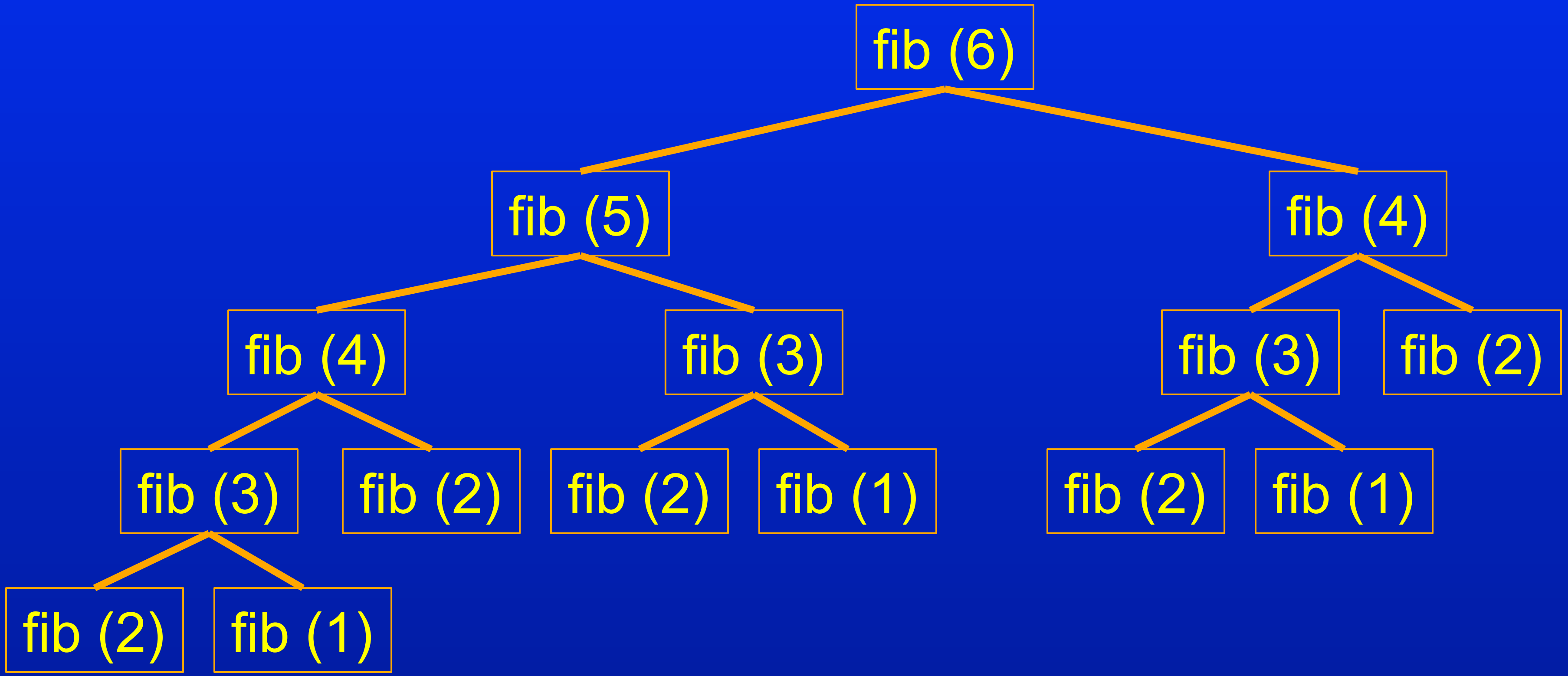
Courtesy Prof PR Panda CSE Department IIT Dehi

Fibonacci Numbers (Revisit)



Courtesy Prof PR Panda CSE Department IIT Dehi

Fibonacci Numbers (Revisit)



Complexity $O(2^n)$

Courtesy Prof PR Panda CSE Department IIT Dehi

Fibonacci Numbers (Modified)

- Based on memorization
- Save result when first computed
- Implement using an array (list)

Fibonacci Numbers (Modified)

```
def Fibonacci(n,save):
   if (save[n]>-1):
      return save[n]
   else:
      result = Fibonacci(n-1,save) + Fibonacci(n-2,save)
      save[n]=result
      return result

n=int(input('Input n:'))
save = [-1 for i in range(n+1)]
save[1]=0
save[2]=1
f = Fibonacci(n,save)
print(f)
```

Fibonacci Numbers (Modified)

```
def Fibonacci(n,save):
   if (save[n]>-1):
      return save[n]
   else:
      result = Fibonacci(n-1,save) + Fibonacci(n-2,save)
      save[n]=result
      return result

n=int(input('Input n:'))
save = [-1 for i in range(n+1)]
save[1]=0
save[2]=1
f = Fibonacci(n,save)
print(f)
```

Complexity O(n)

Fibonacci Numbers (Modified)

- fib(34): 3524578
  - For the naïve recursive program the number of calls is 11405773
  - For the modified program the number of calls is 65

Dictionary

Motivation

- Consider that one wants to associate name (id) with grades of students.
- Can obtain through two separate lists
  - names: [‘Mukesh’, ‘Sham’, ‘Arpita’, ‘Neha’]
  - grades:[‘A-’,’B’,’A’,’C’]
- Separate list of same length for each item
- Associated information stored across lists at same index
- Retrieval and manipulation is not easy

Motivation

- Consider that one wants to associate name (id) with grades of students.
- Can obtain through two separate lists
  – names: [‘Mukesh’, ‘Sham’, ‘Arpita’, ‘Neha’]
  – grades:[‘A-’,’B’,’A’,’C’]
- Separate list of same length for each item
- Associated information stored across lists at same index
- Retrieval and manipulation is not easy

Dictionary

- Natural data structure to store pairs of data.
  - key (custom index by label)
  - value

```
grades={'Mukesh':'A-','Sham':'B','Arpita':'A','Neha':'C'}
```

Dictionary

- Lookup:
  - similar to indexing into list
  - looks up the key and returns the value associated with the key
  - if key is not found returns error

```
grades={'Mukesh':'A-','Sham':'B','Arpita':'A','Neha':'C'}
```

  - print(grades['Sham']) → B
  - print(grades['Amit']) → Error

Dictionary

- Other operations:
  - add an entry:
    - grades[‘Ankit’]=‘B-’

{'Mukesh': 'A-', 'Sham': 'B', 'Arpita': 'A', 'Neha': 'C', 'Ankit': 'B-'}

  - test if key is in dictionary
    - Mukesh in grades → returns True
    - Suresh un grades → returns False
  - delete an entry
    - del(grades[‘Neha’])

Dictionary

- Other operations:
  - update an entry:
    - grades.update({‘Ankit’:‘B-’})

{'Mukesh': 'A-', 'Sham': 'B', 'Arpita': 'A', 'Neha': 'C', 'Ankit': 'B-'}

    - grades.update({‘Neha’:‘B-’})

{'Mukesh': 'A-', 'Sham': 'B', 'Arpita': 'A', 'Neha': ‘B-', 'Ankit': 'B-'}

  - get for getting the value for a key
    - grades.get(‘Mukesh’) → returns A
  - pop for removing a specific item
    - grades.pop(‘Neha’)

Dictionary

- Other operations:
  - grades.keys() gives the keys, the order may not be guaranteed

    dict_keys(['Mukesh', 'Sham', 'Arpita', 'Neha'])
  - grades.values() gives the values, the order may not be guaranteed

    dict_values(['A-', 'B', 'A', 'C'])
  - grades.items() gives the contents

    dict_items([('Mukesh', 'A-'), ('Sham', 'B'), ('Arpita', 'A'), ('Neha', 'C')])

List vs Dictionary

List	Dictionary
Ordered sequence of elements	Matches keys to values
Indices have an order	No order is guaranteed
Index is an integer	Key can be any immutable type

Dictionary is also known as associate array or hashmap in other programming languages

Fibonacci Numbers (Modified)

```
def fib_e(n,d):
  if n in d:
    return d[n]
  else:
    save = fib_e(n-1,d)+fib_e(n-2,d)
    d[n] = save
    return save

n=int(input('Please give n:'))
d={1:0, 2:1}
print(fib_e(n,d))
```

Use of Dictionary

Complexity O(n)