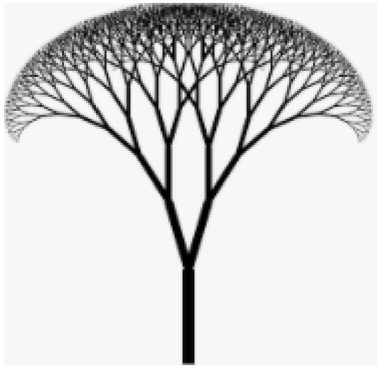
04
Recursion
Chapter 4

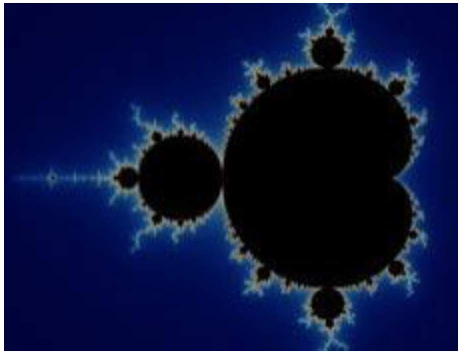
What is recursion?

A problem solving technique by which a function makes calls to itself.

Many examples in art and nature (e.g., fractals)

A powerful alternative for iterative tasks





What is recursion?

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

base condition

induction

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Recursion Callstack Example

CALL
STACK
				call: **fact(0)**
				if n==0:
				> return **1**
				else:
				return (0*fact(-1))			

			call: **fact(1)**
			if n==0:
			return 1
			else:
			> return (1*fact(0))
				call: **fact(1)**
				if n==0:
				return 1
				else:
				> return (1*fact(0))
					call: **fact(1)**
					if n==0:
					return 1
					else:
					> return (1***1**)		

		call: **fact(2)**
		if n==0:
		return 1
		else:
		> return (2*fact(1))
			call: **fact(2)**
			if n==0:
			return 1
			else:
			> return (2*fact(1))
				call: **fact(2)**
				if n==0:
				return 1
				else:
				> return (2*fact(1))
					call: **fact(2)**
					if n==0:
					return 1
					else:
					> return (2*fact(1))
						call: **fact(2)**
						if n==0:
						return 1
						else:
						> return (2***1**)	

	call: **fact(3)**
	if n==0:
	return 1
	else:
	> return (3*fact(2))
		call: **fact(3)**
		if n==0:
		return 1
		else:
		> return (3*fact(2))
			call: **fact(3)**
			if n==0:
			return 1
			else:
			> return (3*fact(2))
				call: **fact(3)**
				if n==0:
				return 1
				else:
				> return (3*fact(2))
					call: **fact(3)**
					if n==0:
					return 1
					else:
					> return (3*fact(2))
						call: **fact(3)**
						if n==0:
						return 1
						else:
						> return (3*fact(2))
							call: **fact(3)**
							if n==0:
							return 1
							else:
							> return (3***2**)

	First call	**First recursive call**	**Second recursive call**	**Third recursive call**	**Third recursive call's execution is over.**	**Second recursive call's execution is**	**First recursive call's execution is over.**

Recursion Callstack Example

C
A
L
L
 	
S
T
A
C
K
				call: **fact(0)**
				if n==0:
				> return 1
				else:
				return (0*fact(-1))

			call: **fact(1)**
			if n==0:
			return 1
			else:
			> return (1*fact(0))
				call: **fact(1)**
				if n==0:
				return 1
				else:
				> return (1*fact(0))

		call: **fact(2)**
		if n==0:
		return 1
		else:
		> return (2*fact(1))
			call: **fact(2)**
			if n==0:
			return 1
			else:
			> return (2*fact(1))
				call: **fact(2)**
				if n==0:
				return 1
				else:
				> return (2*fact(1))

	call: **fact(3)**
	if n==0:
	return 1
	else:
	> return (3*fact(2))
		call: **fact(3)**
		if n==0:
		return 1
		else:
		> return (3*fact(2))
			call: **fact(3)**
			if n==0:
			return 1
			else:
			> return (3*fact(2))
				call: **fact(3)**
				if n==0:
				return 1
				else:
				> return (3*fact(2))

	First call	**First recursive call**	**Second recursive call**	**Third recursive call**

Recursion Callstack Example

call: `fact(1)`
`if n==0:`
`    return 1`
`else:`
` > return (1*1)`		

call: `fact(2)`
`if n==0:`
`    return 1`
`else:`
` > return (2*fact(1))`
	call: `fact(2)`
	`if n==0:`
	`    return 1`
	`else:`
	` > return (2*1)`	

call: `fact(3)`
`if n==0:`
`    return 1`
`else:`
` > return (3*fact(2))`
	call: `fact(3)`
	`if n==0:`
	`    return 1`
	`else:`
	` > return (3*fact(2))`
		call: `fact(3)`
		`if n==0:`
		`    return 1`
		`else:`
		` > return (3*2)`

Third recursive call's execution is over.	**Second recursive call's execution is**	**First recursive call's execution is over.**

CALL STACK

Recursion Examples

```
def fact(n):            c1
    if n == 0:          c2
        return 1
    else:               c3
        return n * fact(n - 1)
```

$\underbrace{\texttt{fact(n - 1)}}_{T(n-1)}$

$T(n) = T(n-1) + c_1 + c_2 + c_3 = T(n-1) + c$

$= (T(n-2) + c) + c$

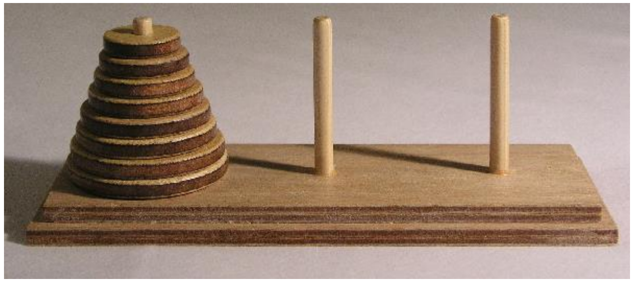
$= (T(n-3) + c) + c + c$

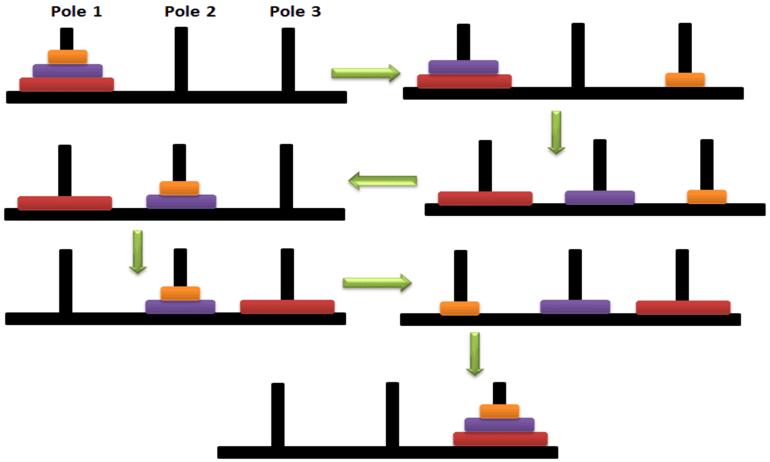
$= (T(n-i) + i*c)$

when $i=n \rightarrow T(0) + n*c \rightarrow T(n) = O(n)$

Recursion Examples

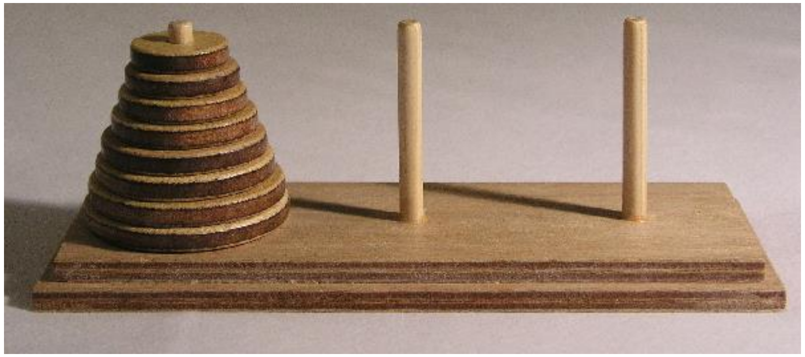
The Tower of Hanoi





Recursion Examples

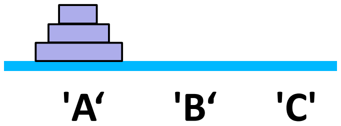
The Tower of Hanoi

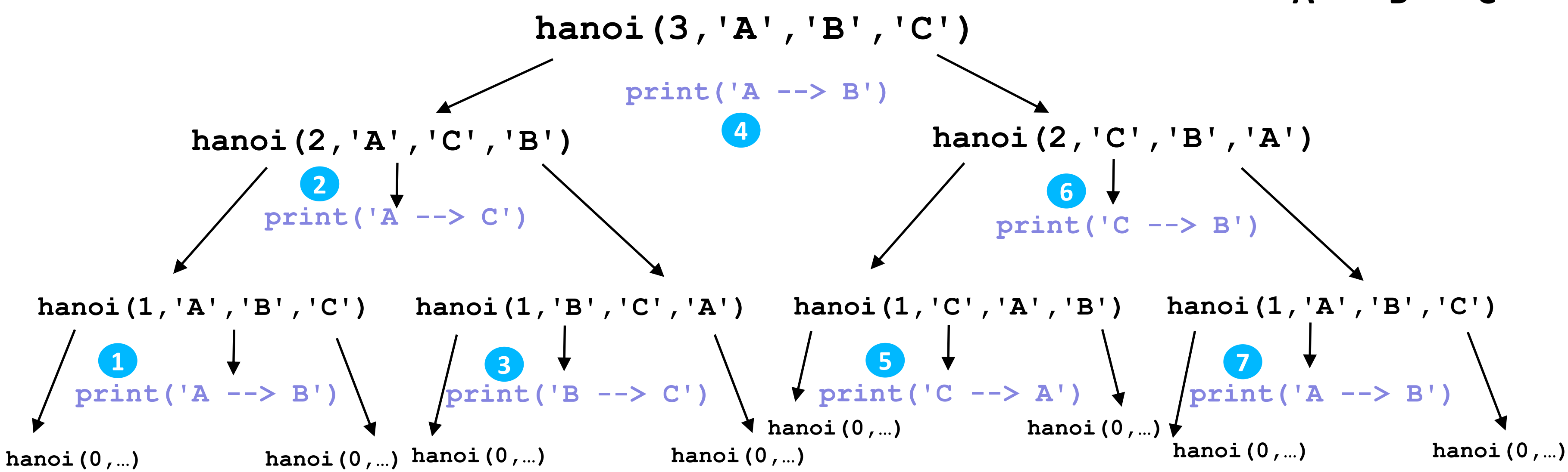


```
def hanoi(n, source, dest, spare):  # Cost
  if (n > 0):                       # c1
    hanoi(n-1, source, spare, dest) # T(n-1)
    print(f' Move top disk from pole {source} to pole {dest}')#c2
    hanoi(n-1, spare, dest, source) # T(n-1)
```

Recursion Examples

The Tower of Hanoi (call tree)



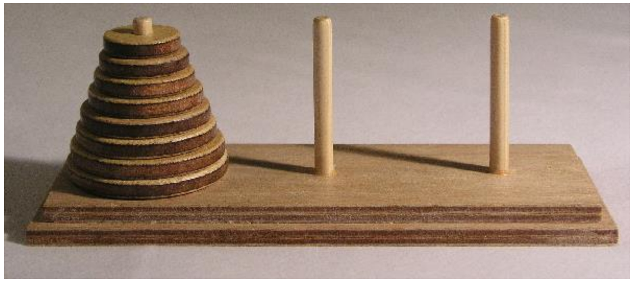



```python
def hanoi(n, source, dest, spare):   # Cost
  if (n > 0):                        # c1
    hanoi(n-1, source, spare, dest) # T(n-1)
    print(f' Move top disk from pole {source} to pole {dest}')#c2
    hanoi(n-1, spare, dest, source) # T(n-1)
```

Recursion Examples

The Tower of Hanoi

What is the cost of `hanoi(n,'A','B','C')`?



when n=0

T(0) = c1

when n>0

T(n) = c1 + T(n-1) + c2 + T(n-1)

= 2*T(n-1) + (c1+c2)

= **2*T(n-1) + c**

Recurrence equation for the growth-rate function of the Tower of Hanoi algorithm.

Recursion Examples

Methodology: *repeated substitutions*

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1th times

$= 2^i * T(n-i) + (2^{i-1}+ \dots +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ \dots +2^1+2^0)*c$

$= 2^n * c1 + \left(\sum_{i=0}^{n-1} 2^i \right)*c$

Some mathematical equalities are:

$$\sum_{i=1}^{n} i = 1+2+\dots+n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1+4+\dots+n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0+1+2+\dots+2^{n-1} = 2^n - 1$$

$= 2^n * c1 + (2^n-1)*c = 2^n*(c1+c) – c$ ➔ So, the growth rate function is **$O(2^n)$**

Recursion Examples

Palindrome

Sequence of symbols that reads the same backward as forward

e.g., Al lets Della call Ed “Stella.”, Borrow or rob?

```
def isPalindrome (s):
    if (len(s) <= 1):
        return True
    return (s[0]==s[len(s)-1]) and isPalindrome(s[1 : len(s)-1])
```

$T(n) = T(n-2) + c$

$= (T(n-4) + c) + c$

$= (T(n-6) + c) + c + c$

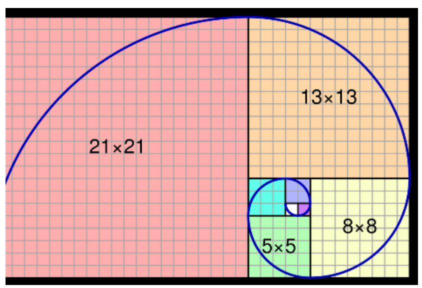
$= (T(n-i) + i/2*c)$

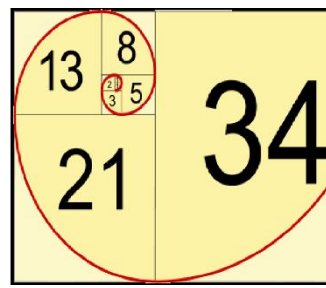
when $i=n \rightarrow T(0) + n/2*c \Rightarrow T(n) = O(n)$

Recursion Examples

Fibonacci Number

Fibonacci numbers form a sequence (i.e., Fibonacci sequence) such that each number is the sum of the two preceding ones.





Recursion Examples

```
def fib(n): # 1 1 2 3 5 8 13 21 34 ..
  if (n == 1):
        return 1
  if (n == 2):
        return 1
  else:
        return fib(n-1) + fib(n-2)
```

$T(n) = ?$

Recursion Examples

```
def fib(n): # 1 1 2 3 5 8 13 21 34 ..
  if (n == 1):
        return 1
  if (n == 2):
        return 1
  else:
        return fib(n-1) + fib(n-2)
```

$T(n) = T(n-1) + T(n-2) + c$

$= (T(n-2) + T(n-3) + c) + (T(n-3) + T(n-4) + c) + c$

= gets nasty if we continue substituting; let's find a better way.

Recursion Examples

```
def fib(n): # 1 1 2 3 5 8 13 21 34 ..
  if (n == 1):
        return 1
  if (n == 2):
        return 1
  else:
        return fib(n-1) + fib(n-2)
```

$T(n) = T(n-1) + T(n-2) + c$

$T(n-1) + T(n-2) + c < T(n-1) + T(n-1) + c = O(2^n)$ (recall the solution in Hanoi towers)
This shows that it cannot exceed $O(2^n)$, hence an upper bound.

Recursion Examples

If we can find a lower bound for T(n) and show that it is also $\Omega(2^n)$, then we can say that T(n) is $\Theta(2^n)$.

T(n-1) + T(n-2) + c > T(n-2) + T(n-2) + c (because T(n-1) > T(n-2) slightly.)

T(n) = 2T(n-2) + c

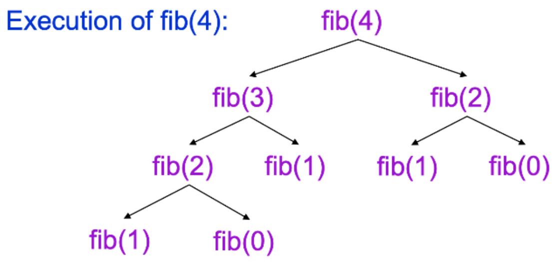
= 2(2T(n-4) + c) + c = 4T(n-4) + 3c

= 2(2(2T(n-6) + c) + c) + c = 8T(n-6) + 7c

= … = 16T(n-8) + 15c

= 2^iT(n-2i) + (2^i-1)

when i = n/2 ➔ $2^{n/2}$T(0) + $2^{n/2}$-1 = $\Omega(2^n)$ is the lower bound for T(n).



Recursion Examples

Fibonacci in O(n)

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n−1)
    return (a+b, a)
```

Recursion Examples

Fibonacci in O(n)

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n−1)
    return (a+b, a)
```

$T(n) = T(n-1) + c_1$

$T(1) = c_0$

fib(4)=(3,2)

fib(3)=(2,1)

fib(2)=(1,1)

fib(1)=(1,0)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Analysis Approaches

An algorithm can require different times to solve different problems of the same size.

e.g., searching an item in a list of n elements using sequential search. The cost could be anywhere between 1 and n value check operations.

Worst-case Analysis –The maximum amount of time that an algorithm require to solve a problem of size n.

This gives an upper bound for the time complexity of an algorithm.

Normally, we try to find worst-case behavior of an algorithm.

Analysis Approaches

Best-case Analysis –The minimum amount of time that an algorithm require to solve a problem of size n.

The best case behavior of an algorithm is NOT so useful.

Average-case Analysis –The average amount of time that an algorithm require to solve a problem of size n.

Sometimes, it is difficult to find the average-case behavior of an algorithm.

We might need to investigate all possible data organizations of a given size n, and their distribution probabilities of these organizations.

Worst-case analysis is more common than average-case analysis.

Sequential Search

```
def sequential_search(item_list, item):
    for i in range(len(item_list)):
        if item == item_list[i]:
            return item
    return -1
```

Unsuccessful Search: O(n)

Successful Search:

Best-Case: *item* is in the first location of the array O(1)

Worst-Case: *item* is in the last location of the array O(n)

Average-Case: The number of key comparisons 1, 2, ..., n   O(n)

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$

Binary Search

Binary search on a sorted array

```
def binary_search(arr, low, high, x):
    if high >= low:  # Check base case
        mid = (high + low) // 2
        if arr[mid] == x: # In middle? Return itself
            return mid
        # Smaller than mid? has to be in left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else: # Else? Has to be in right subarray
            return binary_search(arr, mid + 1, high, x)
    else: # Element is not present in the array
        return -1
```

Borrowed from GeeksForGeeks: https://www.geeksforgeeks.org/python-program-for-binary-search/

Binary Search

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

```
Low   :
Medium:
High  :

Find:
23
91
92
```

Binary Search Analysis

- For an unsuccessful search:
  - The number of invocations in the recursion is $\lfloor \log_2 n \rfloor + 1 \rightarrow O(\log_2 n)$

- For a successful search:
  - ***Best-Case:*** The number of invocations is 1. $\rightarrow O(1)$
  - ***Worst-Case:*** The number of invocations is $\lfloor \log_2 n \rfloor + 1 \rightarrow O(\log_2 n)$
  - ***Average-Case:*** The avg. # of invocations < $\log_2 n$ $\rightarrow O(\log_2 n)$

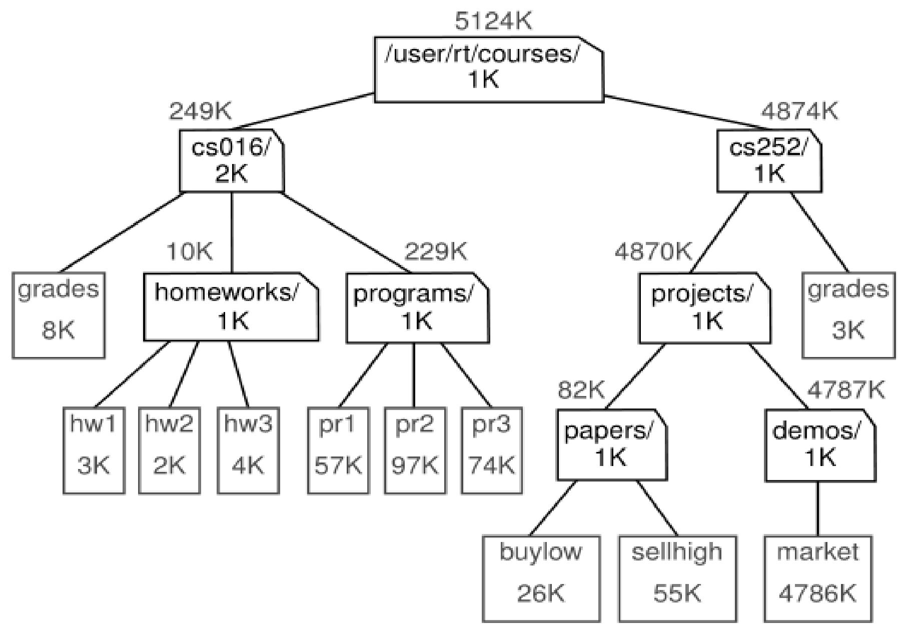
0 1 2 3 4 5 6 → an array with size 7

3 2 3 1 3 2 3 → # of invocations

The average # of invocations = 17/7 = 2.4285 < $\log_2 7$

File Systems

```
os.path.getsize(path)
os.path.isdir(path)
os.listdir(path)
os.path.join(path, filename)z
```



Algorithm DiskUsage(path):

Input: A string designating a path to a file-system entry

Output: The cumulative disk space used by that entry and any nested entries

total = size(path) {immediate disk space used by the entry}

if path represents a directory **then**

  for each child entry stored within directory path **do**

    total = total + DiskUsage(child) {recursive call}

return *total*

File Systems

disk_usage('/user/rt/course')

disk_usage('/user/rt/course/cs016')

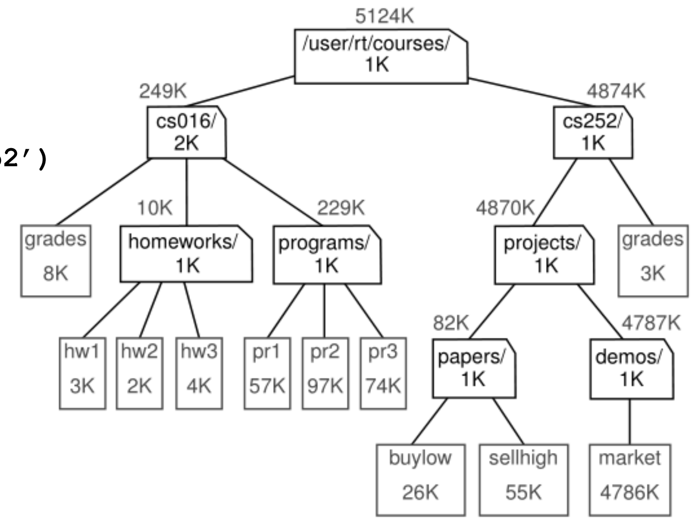
disk_usage('/user/rt/course/cs252')

disk_usage('/user/rt/course/cs016/grades')

. . .

. . .

. . .




```
import os

def disk_usage(path):
  """Return the number of bytes used by a file/folder and any descendents."""
  total = os.path.getsize(path)                # account for direct usage
  if os.path.isdir(path):                      # if this is a directory,
    for filename in os.listdir(path):          # then for each child:
      childpath = os.path.join(path, filename) # compose full path to child
      total += disk_usage(childpath)           # add child's usage to total

  print ('{0:<7}'.format(total), path)         # descriptive output (optional)
  return total                                 # return the grand total
```

Linear Recursion

The maximum number of recursive calls that may be started from within the body of a single activation: 1

Fibonacci

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n−1)
    return (a+b, a)
```

Factorial

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Recursion 1

Binary Recursion

The maximum number of recursive calls that may be started from within the body of a single activation: 2

Fibonacci

```
def fib(n): # 1 1 2 3 5 8 13 21 34 ..
    if (n == 1):
        return 1
    if (n == 2):
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Recursion 1 | Recursion 2

Binary Search

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        return -1
```

Recursion 1

Recursion 2

Multiple Recursion

The maximum number of recursive calls that may be started from within the body of a single activation: >2

Disk space usage of file system

```
import os

def disk_usage(path):
  """Return the number of bytes used by a file/folder and any descendents."""
  total = os.path.getsize(path)                     # account for direct usage
  if os.path.isdir(path):                           # if this is a directory,
    for filename in os.listdir(path):               # then for each child:
      childpath = os.path.join(path, filename)      # compose full path to child
      total += disk_usage(childpath)                  usage to total

  print ('{0:<7}'.format(total), path)              # descriptive output (optional)
  return total                                      # return the grand total
```

Recursion 1..n