Recursion: Computations

- Recursion (illustrated with examples)
  - Basic idea and basic principle (correctness, complexity, and running)
  - Parameterization
  - More on computational complexity
  - Tail recursion

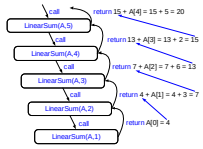
Tail Recursion

- Recursion succinctly exploits a repetitive structure in a problem
- However, it can be sometimes unnecessarily expensive

Sum of items in an array

Algorithm LinearSum(*A*, *n*):
Input:
A integer array *A* and an integer *n* such that *A* has at least *n* elements
Output:
The sum of the first *n* integers in *A*
if *n* = 1 **then**
return *A*[0]
else
return LinearSum(*A*, *n* - 1) + *A*[*n* - 1]

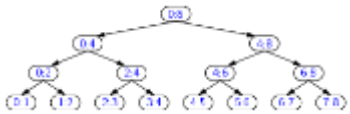
Example: LinearSum(S,5)
S = [4,3,6,2,8]



- Observe that we have to save the state and it grows proportional to n

Achieving Efficiency

- One way to achieve efficiency was the recursive binary sum method
- But this is also unnecessary: an iterative method does not take any extra space

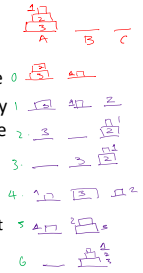


The Tower of Hanoi: Recursion

- The Tower of Hanoi puzzle
- For some problems, it's hard to figure out an iterative solution
  - Bulk move applying S from A to C (using B)
  - Move the largest remaining disk from A to B
  - Bulk move applying S from C to B (using A)

Iterative Solution

- Conceptually arrange the rods in a circle
- Perform the following 2 steps repeatedly (prior to step 2 check if you already have the solution)
  - Move the smallest disk from its current location clockwise
  - Make the only possible move that does not involve the smallest disk



Tail Recursion

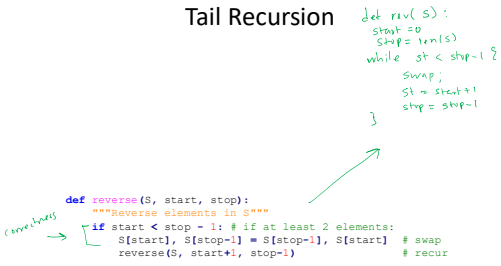
- We want to write a recursive program because it is conceptually easy
- But we want the benefits of iteration
- This is possible with tail recursion
  - Tail recursion occurs when a recursive method makes its recursive call as its last step
  - Any recursive call that is made from one context is the very last operation in that context with the return value immediately returned to the enclosing recursion

Tail Recursion

- The array reversal method is an example
- Such methods can be converted to non-recursive methods automatically
  - saves resources
- We replace the body of the recursion in a loop, and the recursive call with new parameters by a reassignment of the existing parameters

```
def reverse(S, start, stop):
    """Reverse elements in S"""
    if start < stop - 1: # if at least 2 elements:
        S[start], S[stop-1] = S[stop-1], S[start]  # swap
        reverse(S, start+1, stop-1)                # recur
```

Tail Recursion

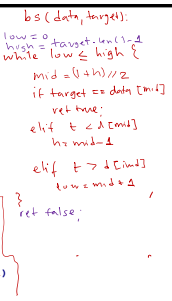


Tail Recursion

```
def reverse_iterative(S):
    """Reverse elements in sequence S."""
    start, stop = 0, len(S)
    while start < stop - 1:
        S[start], S[stop-1] = S[stop-1], S[start]  # swap
        start, stop = start + 1, stop - 1          # narrow
```

```
def reverse(S, start, stop):
    """Reverse elements in S"""
    if start < stop - 1: # if at least 2 elements:
        S[start], S[stop-1] = S[stop-1], S[start]  # swap
        reverse(S, start+1, stop-1)                # recur
```

Binary Search



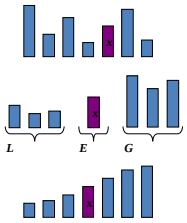

```
def binary_search(data, target, low, high):
    """Return True if target is found
    Considers the portion from [low] to [high] inclusive
    """
    if low > high:
        return False   # interval is empty; no match
    else:
        mid = (low + high) // 2
        if target == data[mid]:          # found a match
            return True
        elif target < data[mid]:
            # recur on the portion left of the middle
            return binary_search(data, target, low, mid - 1)
        else:
            # recur on the portion right of the middle
            return binary_search(data, target, mid + 1, high)
```

Iterative Binary Search

```
def binary_search_iterative(data, target):
    """Return True if target is found """
    low = 0
    high = len(data)-1
    while low <= high:
        mid = (low + high) // 2
        if target == data[mid]:          # a match
            return True
        elif target < data[mid]:
            high = mid - 1  # consider values left of mid
        else:
            low = mid + 1   # consider values right of mid
    return False             # loop ended without success
```

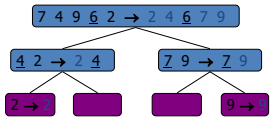
Quick-Sort

- Quick-sort is a classic in place sorting algorithm
  - Divide: pick an element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



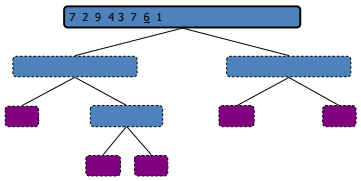
Quick-Sort Tree

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)
```



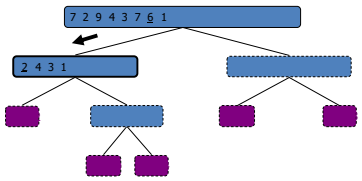
Execution Example

- Pivot selection



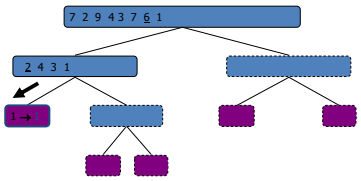
Execution Example (cont.)

- Partition, recursive call, pivot selection



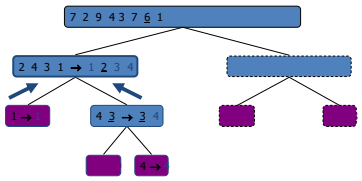
Execution Example (cont.)

- Partition, recursive call, base case



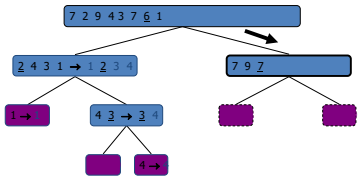
Execution Example (cont.)

- Recursive call, …, base case, join



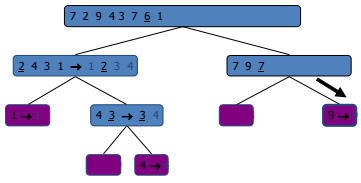
Execution Example (cont.)

- Recursive call, pivot selection



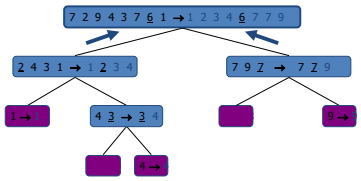
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



quicksort

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Observation: after partition, pivot is in the correct sorted position
We now just need to sort left and right partition!

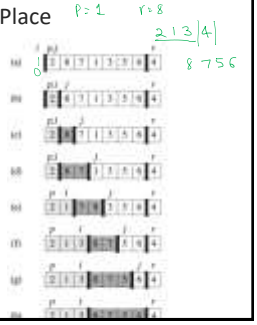
That's quicksort!

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p − 1)
        quicksort(A, p + 1, hi)
```

Partitioning In Place

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

p… i: less than or equal to x
i+1…j-1: greater than x
j…r: not yet processed



Tail Recursion

- There are two calls to quicksort
- The second call can be modified as follows

Algorithm pseudo-tail-recursive (*A, p, r*)
while *p* < r
  q = Partition (A, p, r)
  pseudo-tail-recursive (A, p, q-1)
  p = q+1

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p − 1)
        quicksort(A, p + 1, hi)
```