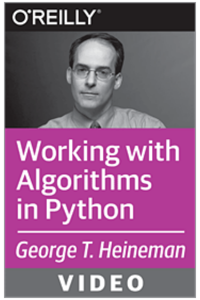
O'REILLY®

O (n log n) Behavior



Divide And Conquer Optimal Behavior

- For many problems the following approach works
  - Divide problem into two sub-problems of ½ the size
  - Solve each sub-problem
  - Combine partial results of sub-problems into solution
- Demonstrate with MergeSort
  - Algorithm structure
  - Performance analysis

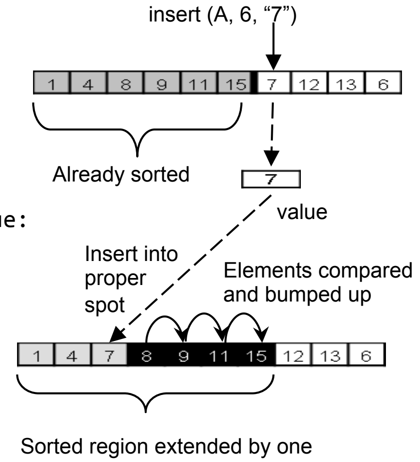
INSERTIONSORT
Non-Optimal Behavior

- Common sorting algorithm
  - Reduces problem size by one with each pass
- Far too much swapping
  - Consider when initial list is in reverse order

INSERTIONSORT

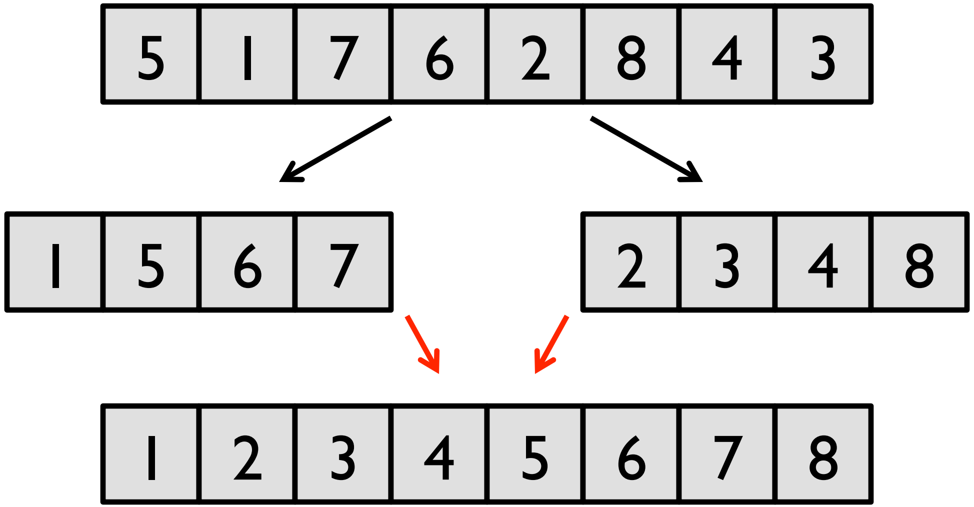
Best	Average	Worst	
O(n)	$O(n^2)$	$O(n^2)$	Array

```
def sort (A):
  for i in range(1,len(A)):
    insert (A, i, A[i])
end

def insert (A, idx, value):
  i = pos-1
  while i ≥ 0 and A[i] > value:
    A[i+1] = A[i]
    i = i-1

  A[i+1] = value

end
```



Divide And Conquer Algorithm Structure

- Problem subdivided into <u>two half-sized</u> problems



- The challenge remains in designing a recursive algorithm
- The issue is that you need additional space equivalent to size of array being sorted

COPYMERGESORT Inefficient Storage

- Merge process is O(*n*)
  - Excessive extra storage
- Creates new arrays
  - Smaller problems
  - Returned merge

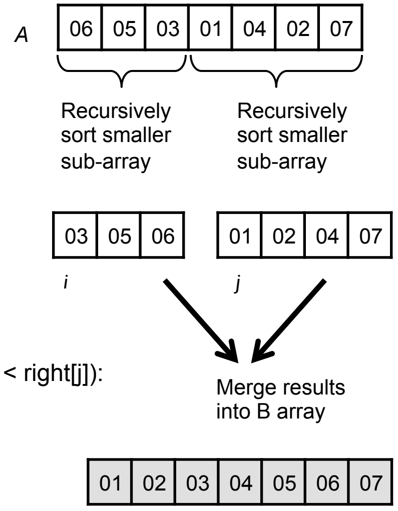
COPYMERGESORT

Best	Average	Worst
O(n log n)	**O(n log n)**	**O(n log n)**

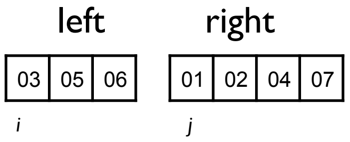
Recursion Array

Divide and Conquer

```
def copyMergeSort (A):
  if len(A) < 2: return A

  mid = len(A)/2
  left = copyMergeSort(A[:mid])
  right = copyMergeSort(A[mid:])

  i = j = 0
  B = []
  while len(B) < len(A):
    if j ≥ len(right) or (i < mid and left[i] < right[j]):
      B.append(left[i])
      i += 1
    elif j < len(right):
      B.append(right[j])
      j += 1
return B
```



Merge
Four Cases To Consider



- Work from left to right in increasing order
- When both have elements remaining
  - Take from left when left[i] < right[j]
  - Take from right otherwise
- Take from right if *i* has exhausted its range
- Take from left if *j* has exhausted its range

MERGESORT Final Version

- Merge process is O(*n*)
  - Requires single extra array
- **result** array must be a duplicate of A
  - Enables base case of recursion to succeed
- Check out code to see how

MERGESORT

Best	Average	Worst
O(n log n)	**O(n log n)**	**O(n log n)**

Recursion
Array
Divide and Conquer

```
def mergeSort (A, result, start, end):
  if end – start < 2: return
  if end – start == 2:
    swap result[start], result[start+1] if reversed and return
  mid = (end+start)/2
  mergeSort(result, A, start, mid)
  mergeSort(result, A, mid, end)
  i = start
  j = mid
  idx = start
  while idx < end:
    if j ≥ end or (i < mid and A[i] < A[j]):
      result[idx] = A[i]
      i += 1
    else:
      result[idx] = A[j]
      j += 1
    idx += 1
```

Divide And Conquer Performance Analysis

- Count number of operations of MERGESORT
  - Recall that algorithm is recursive
  - *t(n)* represents cost of problem of size *n*
  - *m(n)* represents cost of merging two sorted sub-lists

$$t(n) = 2 * t\left(\frac{n}{2}\right) + m(n)$$

Divide And Conquer
O (n log n)

$$t(n) = 2 * t\left(\frac{n}{2}\right) + m(n)$$

- When *m(n)* is O(*n*) then follow chain of logic

$$t(n) = 2 * \left[2 * t\left(\frac{n}{4}\right) + m\left(\frac{n}{2}\right)\right] + m(n)$$

$$t(n) = 2^2 * t\left(\frac{n}{4}\right) + 2 * m\left(\frac{n}{2}\right) + m(n)$$

$$t(n) = 2^2 * t\left(\frac{n}{4}\right) + 2 * c * n$$

$$t(n) = 2^{\log(n)} * t\left(\frac{n}{n}\right) + \log(n) * c * n \rightarrow O(n + n * log(n))$$

$$O(n + n * log(n)) \downarrow O(n * log(n))$$