

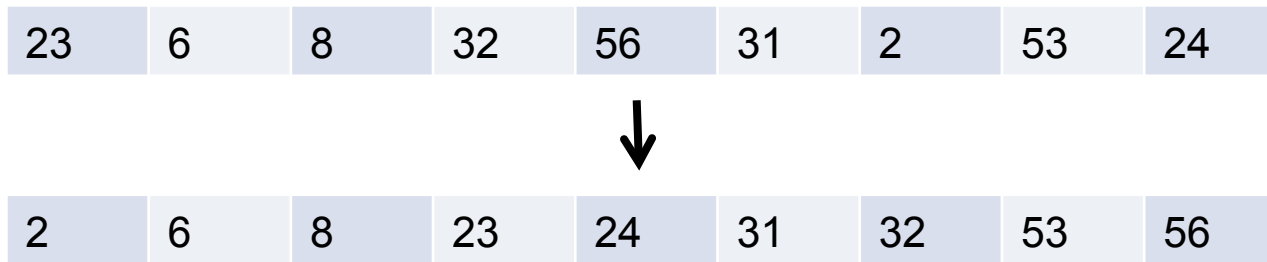
Scientific Computation

Spring 2019

Lecture 2

Sorting

- The two foundational problems in computer science are searching and sorting
- Problem statement: *Given an unsorted list, return a list with the same elements in ascending order*



- Motivation: maintaining sorted lists facilitates fast searches
- Many different algorithms, we will just consider 2

Sorting

- **Algorithm 1**

23	6	8	32	56	31	2	53	24
----	---	---	----	----	----	---	----	----

Step 1: Sort first 2 elements

Step 2: Sort first 3 elements (knowing that 1st two have been sorted)

Step N: sort first N elements (knowing that 1st N-1 have been sorted)

Sorting

- Algorithm 1: 7 of 9 iterations

23	6	8	32	56	31	2	53	24
----	---	---	----	----	----	---	----	----

6	23	8	32	56	31	2	53	24
---	----	---	----	----	----	---	----	----

6	8	23	32	56	31	2	53	24
---	---	----	----	----	----	---	----	----

6	8	23	32	56	31	2	53	24
---	---	----	----	----	----	---	----	----

6	8	23	32	56	31	2	53	24
---	---	----	----	----	----	---	----	----

6	8	23	31	32	56	2	53	24
---	---	----	----	----	----	---	----	----

2	6	8	23	31	32	56	53	24
---	---	---	----	----	----	----	----	----

Sorting

- Algorithm 1

23	6	8	32	56	31	2	53	24
----	---	---	----	----	----	---	----	----

Python implementation:

```
for j in range(1, len(L)):
    #compare L[j] with L[i]
    i = j-1
    key = L[j]

    while i >= 0:

        if key < L[i]: #shift from i to i-1
            i = i-1
            if i < 0: #unless i is already 0
                L[i+2:j+1] = L[i+1:j]
                L[i+1] = key

        else: #insert key at i+1
            L[i+2:j+1] = L[i+1:j]
            L[i+1] = key
            i = -1
```

Sorting

- Algorithm 1

23	6	8	32	56	31	2	53	24
----	---	---	----	----	----	---	----	----

Correctness: If $1^{\text{st}} j$ elements are sorted, then after the j^{th} iteration, $1^{\text{st}} j + 1$ elements are sorted

Speed:

Best case: array is already sorted, requires $N-1$ comparisons

Worst case: j^{th} iteration requires $j-1$ comparisons for each j ($j=1,2, \dots, N-1$) $\rightarrow N(N-1)/2$ comparisons

On average, expect $N(N-1)/4$

Asymptotic time, $O(N^2)$ operations

Can we do better?

Sorting

- A “divide and conquer” approach worked well for search
- Can we do something similar for sorting?
- Test the basic idea:
 - divide array into left and right halves
 - Sort each half
 - Then merge the two halves
- Cost of sorting 2 halves should be half of sorting the full array
- Merge can be done in $O(N)$ operations
- Save $N*(N-1)/8$ during sorting, need extra $O(N)$ during merging
 - Substantial savings for ‘large’ N

Merging

- Merge can be done in $O(N)$ operations?
- How do we merge L and R below into a sorted array, M?

6	8	23	32	2	24	31	53
---	---	----	----	---	----	----	----

- Fill each element of merged array sequentially
 - 1st element: compare L[0] with R[0]

2								
---	--	--	--	--	--	--	--	--

- 2nd element: compare L[0] with R[1]

2	6							
---	---	--	--	--	--	--	--	--

i^{th} element: compare leftmost unassigned element of L with leftmost unassigned element of R

Requires N comparisons and N assignments

Merging

i^{th} element: compare leftmost element of L (not in M) with leftmost element of R (not in M)

Python implementation:

- Outer loop: add one element from either L or R to M each iteration
- Keep track of leftmost indices in L and R

```
indL, indR=0,0
```

```
for i in range(n):
    if L[indL]<R[indR]: #add element from L to M
        value = L[indL]
        indL = indL+1
    else: #add element from R to M
        value = R[indR]
        indR = indR+1
    M.append(value)
    #Check if all elements in L or R have been assigned
    if indL>len(L)-1:
        M = M + R[indR:]
        break
    elif indR>len(L)-1:
        M = M + L[indL:]
        break
return M
```