CSE 115
Introduction to Computer Science I
Road map

- Review (sorting)
- Empirical Demo
- Defining Custom Sorts
Sorting

Given a sequence of values that can be ordered, sorting involves rearranging these values so they go from smallest to largest (or largest to smallest).

Example:

\[17, 93, 12, 44, 82, 81, 22, 73]\]

all mixed up

Sorting rearranges items:

\[12, 17, 22, 44, 73, 81, 82, 93]\]

smallest \rightarrow \text{increasing} \rightarrow \text{largest}
Selection sort

Given a list of values, repeatedly select the smallest value and push it to the end of a list of sorted values.
Selection sort

UNSORTED LIST
[17, 93, 12, 44, 82, 81, 22, 73]

[]
SORTED LIST
Selection sort

UNSORTED LIST
[17, 93, 12, 44, 82, 81, 22, 73]

Smallest from unsorted is 12.

[]
SORTED LIST
Selection sort

UNSORTED LIST
[17, 93, 12, 44, 82, 81, 22, 73]

Remove 12 from unsorted.

[]
SORTED LIST
Selection sort

UNSORTED LIST
[17, 93, 44, 82, 81, 22, 73]

Add 12 to end of sorted.

[12]
SORTED LIST
Selection sort

UNSORTED LIST
[17, 93, 44, 82, 81, 22, 73]

Smallest from unsorted is 17.

[12]
SORTED LIST
Selection sort

UNSORTED LIST
[ 93, 44, 82, 81, 22, 73]

Remove 17 from unsorted.

[12]
SORTED LIST
Selection sort

UNSORTED LIST
[93, 44, 82, 81, 22, 73]

Add 17 to end of sorted.

[12, 17]
SORTED LIST
Selection sort

UNSORTED LIST
[93, 44, 82, 81, 22, 73]

Smallest from unsorted is 22.

[12, 17]
SORTED LIST
Selection sort

UNSORTED LIST
[93, 44, 82, 81, 73]

Remove 22 from unsorted.

[12, 17]
SORTED LIST
Selection sort

UNSORTED LIST
[93, 44, 82, 81, 73]

Add 22 to end of sorted.

[12, 17, 22]
SORTED LIST
Selection sort

UNSORTED LIST
[93, 44, 82, 81, 73]

... and so on ...

[12, 17, 22]
SORTED LIST
Selection sort

UNSORTED LIST
[93]

Smallest from unsorted is 93.

[12, 17, 22, 44, 73, 81, 82]
SORTED LIST
Selection sort

UNSORTED LIST
[ ]

Remove 93 from unsorted.

[12, 17, 22, 44, 73, 81, 82]
SORTED LIST
Selection sort

UNSORTED LIST
[]

Add 93 to end of sorted.

[12, 17, 22, 44, 73, 81, 82, 93]
SORTED LIST
Selection sort

UNSORTED LIST
[]

Since the unsorted list is empty, we're done!

[12, 17, 22, 44, 73, 81, 82, 93]
SORTED LIST
def selectionSort(unsorted):
    sorted = []
    while len(unsorted) > 0:
        sorted.append(removeSmallest(unsorted))
    return sorted

The code shown in these slides is slightly different from what was shown in class (and in earlier slide sets). It has been cleaned up and simplified.
def selectionSort(unsorerted):
    sorted = []

    As long as unsorted is not empty,
    move smallest from unsorted to sorted

    Return sorted list
def selectionSort(unsorted):
    sorted = []
    As long as unsorted is not empty,
    move smallest from unsorted to sorted
    return sorted
def selectionSort(unsorted):
    sorted = []
    while len(unsorted) > 0:
        move smallest from unsorted to sorted
        return sorted
def selectionSort(unsorted):
    sorted = []
    while len(unsorted) > 0:
        sorted.append(removeSmallest(unsorted))
    return sorted
def removeSmallest(aList):
    initialize smallest to first value in aList
    look through all values in aList
    and if a smaller value is found
    update smallest
    remove smallest from aList
    return the smallest value
removeSmallest function

def removeSmallest(aList):
    smallest = aList[0]
    look through all values in aList
    and if a smaller value is found
    update smallest
    remove smallest from aList
    return the smallest value
removeSmallest function

def removeSmallest(aList):
    smallest = aList[0]
    look through all values in aList
    and if a smaller value is found
    update smallest

    remove smallest from aList

    return smallest
def removeSmallest(aList):
    smallest = aList[0]
    for value in aList:
        if value < smallest:
            smallest = value
    aList.remove(smallest)
    return smallest
def removeSmallest(aList):
    smallest = aList[0]
    for value in aList:
        if value < smallest:
            smallest = value
    aList.remove(smallest)
    return smallest
removeSmallest function

def removeSmallest(aList):
    smallest = aList[0]
    for value in aList:
        if value < smallest:
            update smallest
    aList.remove(smallest)
    return smallest
removeSmallest function

def removeSmallest(aList):
    smallest = aList[0]
    for value in aList:
        if value < smallest:
            smallest = value
    aList.remove(smallest)
    return smallest
Merge sort

Given a list of values, split into in left and right partitions of roughly equal size. Sort each partition, then merge the two sorted partitions.
Merge sort

basic idea

[17, 93, 12, 44, 82, 81, 22, 73]
Merge sort
split into partitions

[17, 93, 12, 44, 82, 81, 22, 73]

[17, 93, 12, 44] [82, 81, 22, 73]
Merge sort

sort partitions

[17, 93, 12, 44, 82, 81, 22, 73]

[17, 93, 12, 44] [82, 81, 22, 73]

[12, 17, 44, 93] [22, 73, 81, 82]
Merge sort
merge partitions

\[
\begin{align*}
[17, 93, 12, 44, 82, 81, 22, 73] \\
[17, 93, 12, 44] & [82, 81, 22, 73] \\
& . \\
& . \\
[12, 17, 44, 93] & [22, 73, 81, 82] \\
[12, 17, 22, 44, 73, 81, 82, 93]
\end{align*}
\]
Merge sort
merge partitions

[17, 93, 12, 44, 82, 81, 22, 73]

[17, 93, 12, 44] [82, 81, 22, 73]

[17, 93] [12, 44] [82, 81] [22, 73]

[17] [93] [12] [44] [82] [81] [22] [73]

[17, 93] [12, 44] [81, 82] [22, 73]

[12, 17, 44, 93] [22, 73, 81, 82]

[12, 17, 22, 44, 73, 81, 82, 93]
mergeSort function

def mergeSort(X):
    mergeSortHelper(X, 0, len(X))
    return X

mergeSort calls mergeSortHelper with the endpoints of the initial partition
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    As long as partition can be split
    split into two roughly equal-sized partitions
    sort each partition (using merge sort)
    merge two sorted partitions into one
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    if Right - Left > 1 :
        Mid = (Left  + Right) // 2
        mergeSortHelper(X, Left, Mid)
        mergeSortHelper(X, Mid, Right)
        merge(X, Left, Mid, Right)

        split into two roughly equal-sized partitions
        sort each partition (using merge sort)
        merge two sorted partitions into one
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    if Right - Left > 1:
        Mid = (Left + Right) // 2
        sort each partition (using merge sort)
        merge two sorted partitions into one
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    if Right - Left > 1:
        Mid = (Left + Right) // 2
        sort left partition
        sort right partition
        merge two sorted partitions into one
def mergeSortHelper(X, Left, Right):
    if Right - Left > 1:
        Mid = (Left + Right) // 2
        mergeSortHelper(X, Left, Mid)
        mergeSortHelper(X, Mid, Right)
    merge(X, Left, Mid, Right)

# sort right partition
# merge two sorted partitions into one
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    if Right - Left > 1:
        Mid = (Left + Right) // 2
        mergeSortHelper(X, Left, Mid)
        mergeSortHelper(X, Mid, Right)
    merge two sorted partitions into one
mergeSortHelper function

def mergeSortHelper(X, Left, Right):
    if Right - Left > 1:
        Mid = (Left + Right) // 2
        mergeSortHelper(X, Left, Mid)
        mergeSortHelper(X, Mid, Right)
    merge(X, Left, Mid, Right)
def merge(X, L, M, R):

    create empty sorted list
    create "pointers" to left & right partitions

    compare the smallest values from left and right
    add smaller of the two to sorted list

    copy remaining data from left partition to sorted

    copy remaining data from right partition to sorted

    copy from sorted back to original list
def merge(X, L, M, R):
    temp = []
    lp = L
    rp = M
    while lp < M and rp < R:
        if X[lp] < X[rp]:
            temp.append(X[lp])
            lp = lp + 1
        else:
            temp.append(X[rp])
            rp = rp + 1
    while lp < M:
        temp.append(X[lp])
        lp = lp + 1
    while rp < R:
        temp.append(X[rp])
        rp = rp + 1
    for i in range(L,R):
        X[i] = temp[i-L]

- Compare the smallest values from left and right
- Add smaller of the two to sorted list
- Copy remaining data from left partition to sorted
- Copy remaining data from right partition to sorted
- Copy from sorted back to original list
def merge(X, L, M, R):
    temp = []
    lp = L
    rp = M
    while lp < M and rp < R:
        if X[lp] < X[rp]:
            temp.append(X[lp])
            lp = lp + 1
        else:
            temp.append(X[rp])
            rp = rp + 1
    while lp < M:
        temp.append(X[lp])
        lp = lp + 1
    while rp < R:
        temp.append(X[rp])
        rp = rp + 1
    for i in range(L, R):
        X[i] = temp[i-L]
def merge(X, L, M, R):
    temp = []
    lp = L
    rp = M
    while lp < M and rp < R:
        if X[lp] < X[rp]:
            temp.append(X[lp])
            lp = lp + 1
        else:
            temp.append(X[rp])
            rp = rp + 1
    while lp < M:
        temp.append(X[lp])
        lp = lp + 1
    for i in range(L, R):
        X[i] = temp[i - L]

    copy remaining data from right partition to sorted
    copy from sorted back to original list
merge function

def merge(X, L, M, R):
    temp = []
    lp = L
    rp = M
    while lp < M and rp < R:
        if X[lp] < X[rp]:
            temp.append(X[lp])
            lp = lp + 1
        else:
            temp.append(X[rp])
            rp = rp + 1
    while lp < M:
        temp.append(X[lp])
        lp = lp + 1
    while rp < R:
        temp.append(X[rp])
        rp = rp + 1
    for i in range(L,R):
        X[i] = temp[i-L]
    copy from sorted back to original list
merge function

def merge(X, L, M, R):
    temp = []
    lp = L
    rp = M
    while lp < M and rp < R:
        if X[lp] < X[rp]:
            temp.append(X[lp])
            lp = lp + 1
        else:
            temp.append(X[rp])
            rp = rp + 1
    while lp < M:
        temp.append(X[lp])
        lp = lp + 1
    while rp < R:
        temp.append(X[rp])
        rp = rp + 1
    for i in range(L,R):
        X[i] = temp[i-L]
Selection vs Merge sort
# comparisons are a good measure of work
assume input has N items

Selection sort requires roughly \( N^2 \) comparisons.

\( N^2 \) grows quickly:

\[(2N)^2 = 4 \ N^2\]

Double input size quadruples time needed.

Merge sort requires roughly \( N \log_2(N) \) comparisons.

\( N \log_2(N) \) grows slowly:

\[(2N) \log_2(2N) = 2N [\log_2(2)+\log_2(N)]\]
\[= 2N [1+\log_2(N)]\]

Double input size a little more than doubles time needed.
Road map

Review (sorting)

▶ Empirical Demo ◀

Defining Custom Sorts
Empirical demonstration

First demo

Generate random data set (list of random integers)

Make a copy of the original data.
Time how long it takes to sort the copy with selection sort.

Remember that selection sort needs $N^2$ time to sort $N$ items:

doubling the input size should quadruple the time.
Empirical demonstration

Second demo

Generate random data set (list of random integers)

Make a copy of the original data.
Time how long it takes to sort the copy with selection sort.

Make a copy of the original data.
Time how long it takes to sort the copy with merge sort.

Remember that merge sort needs $N \log_2 N$ time to sort $N$ items:
doubling the input size should a little more than double the time.
Empirical demonstration

Third demo

Generate random data set (list of random integers)
Make a copy of the original data.
Time how long it takes to sort the copy with selection sort.
Make a copy of the original data.
Time how long it takes to sort the copy with merge sort.
Make a copy of the original data.
Time how long it takes to sort the copy with merge sort.

How much time does Tim Sort need to sort N items?
Tim Sort

Tim Sort is a hybrid sorting algorithm.

It uses a slow sorting algorithm (insertion sort) for small partitions, but a fast algorithm (merge sort) for large partitions.

Like merge sort, in the worst case it needs $N \log_2 N$ time to sort $N$ items. Its overhead is smaller (it runs faster on small partitions).

Very fast in practice.

Is is the built-in sorting algorithm in Python (and several other languages too).
Custom sorting

Suppose we have data on students stored in a dictionary:

```
[ 
  { "fname": "Sally", "lname":"Smith", "pn":"342083", "age":"23" },
  { "fname": "Barb",  "lname":"Woods", "pn":"934850", "age":"21" },
  { "fname": "Bo",    "lname":"Meele", "pn":"393847", "age":"22" },
  { "fname": "Amy",   "lname":"Fable", "pn":"705834", "age":"21" }
]
```

We could sort these dictionaries in many ways: by any of the fields, or by combinations of fields.

How do we specify what the sorting key(s) should be?
Custom sorting

The sort function can take a 'key' argument, which specifies the value to sort the data by.

That value must itself must be sortable.

The value is determined by a function which takes a data element input and returns a value; the data elements will be sorted by the values natural sort.
Custom sorting

students = [
    { "fname": "Sally", "lname": "Smith", "pn": "342083", "age": "23" },
    { "fname": "Barb", "lname": "Woods", "pn": "934850", "age": "21" },
    { "fname": "Bo", "lname": "Meele", "pn": "393847", "age": "22" },
    { "fname": "Amy", "lname": "Fable", "pn": "705834", "age": "21" }
]

Function to return first name:

```python
def fN(V): return V["fname"]
```

How to sort by first name:

```python
students.sort(key = fN)
```
Custom sorting

students = [
    { "fname": "Sally", "lname":"Smith", "pn":"342083", "age":"23" },,
    { "fname": "Barb",  "lname":"Woods", "pn":"934850", "age":"21" },,
    { "fname": "Bo",    "lname":"Meele", "pn":"393847", "age":"22" },,
    { "fname": "Amy",   "lname":"Fable", "pn":"705834", "age":"21" }
]

Function to return length of first name:

def fNLen(V): return len(V["fname"])

How to sort by length of first name:

students.sort(key = fNLen)
Custom sorting

The sort function can take a comparator argument, which specifies how to determine the relative order of two data elements.

The relative order is determined by a function which takes two data elements x and y as input and returns:

- a negative value (typically -1) if x comes before y,
- a positive value (typically 1) if x comes after y,
- zero if x and y belong at the same place in the order.
Custom sorting

```javascript
var students = [
{ "fname": "Sally", "lname":"Smith", "pn":"342083", "age":"23" },
{ "fname": "Barb",  "lname":"Woods", "pn":"934850", "age":"21" },
{ "fname": "Bo",    "lname":"Meele", "pn":"393847", "age":"22" },
{ "fname": "Amy",   "lname":"Fable", "pn":"705834", "age":"21" }
];

function fN(X,Y) {
    if (X["fname"] < Y["fname"]) { return -1; }
    if (X["fname"] > Y["fname"]) { return 1; }
    return 0;
}

students.sort(fN);
```

**Function to determine relative order of first names:**

```javascript
function fN(X,Y) {
    if (X["fname"] < Y["fname"]) { return -1; }
    if (X["fname"] > Y["fname"]) { return 1; }
    return 0;
}
```

**How to sort by first name:**

```javascript
students.sort(fN);
```
var students = [
    { "fname": "Sally", "lname": "Smith", "pn": "342083", "age": "23" },
    { "fname": "Barb", "lname": "Woods", "pn": "934850", "age": "21" },
    { "fname": "Bo", "lname": "Meele", "pn": "393847", "age": "22" },
    { "fname": "Amy", "lname": "Fable", "pn": "705834", "age": "21" }
];

function fNLen(X,Y) {
    if (X["fname"].length < Y["fname"].length) { return -1; }
    if (X["fname"].length > Y["fname"].length) { return 1; }
    return 0;
}

students.sort(fNLen);

Function to determine relative order of lengths of first names:

function fNLen(X,Y) {
    if (X["fname"].length < Y["fname"].length) { return -1; }
    if (X["fname"].length > Y["fname"].length) { return 1; }
    return 0;
}

How to sort by length of first name:

students.sort(fNLen);