Assignment Announcements

Assignment 2:

- 1c compiles on Ubuntu. MacOS can behave differently
- 2c has been omitted from the assignment. Revised solutions will be posted

Assignment 3:

- 1b: row 3 of the table has been corrected

Merge Sort

CMPT 125
Mo Chen
SFU Computing Science
7/2/2020

Lecture 15

Today

- Merge Sort: a Divide and Conquer Sort

Different Sorts of Sorts

So far, we have seen two implementations of sorting:

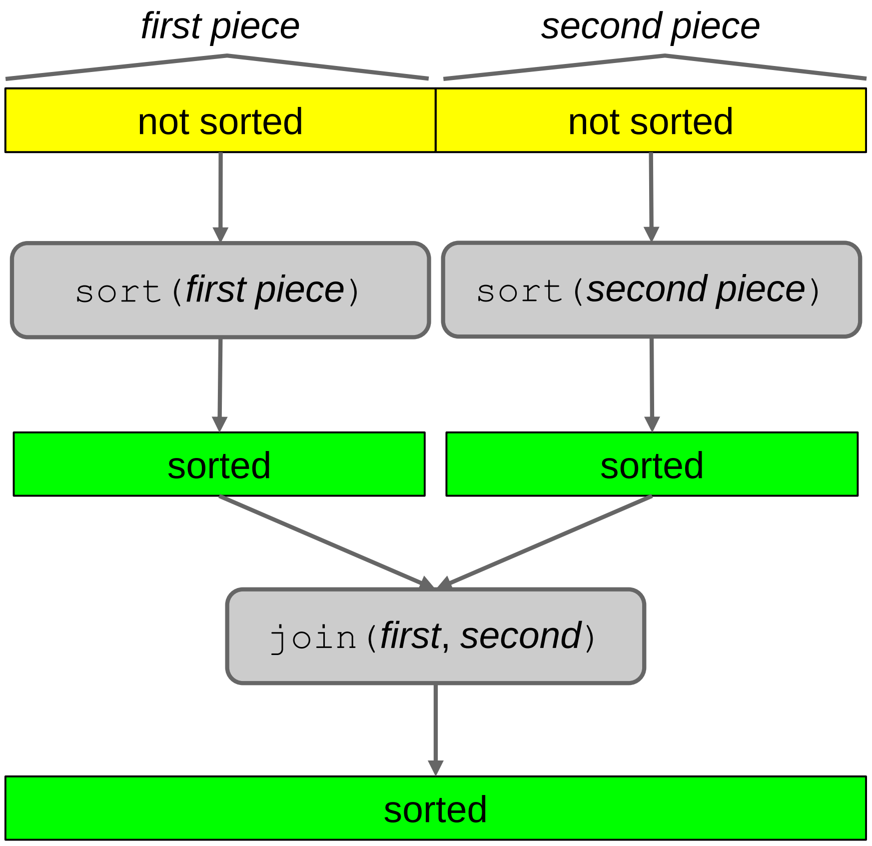
- Selection Sort - find the min, swap it with position 0; find the second min, swap it with position 1; . . . ; working incrementally - $O(N^2)$
- Insertion Sort - incrementally insert an element to a growing list of sorted elements - also $O(N^2)$

To get better performance, we need a non-incremental algorithm

Sorting by Recursion

Use Divide and Conquer to sort recursively.

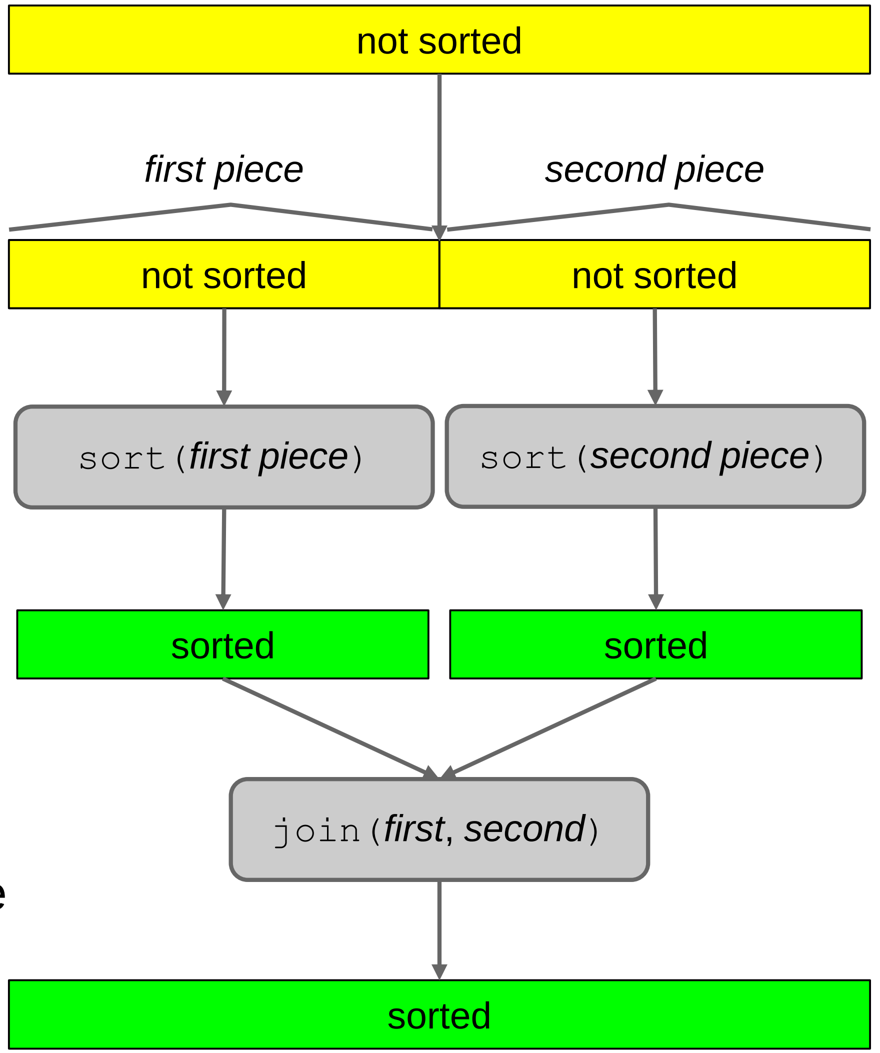
1. Split the array into two roughly equal pieces.
2. Recursively sort each half.
    - This works because each piece is *smaller*.
3. Join the two pieces together to make one sorted array.



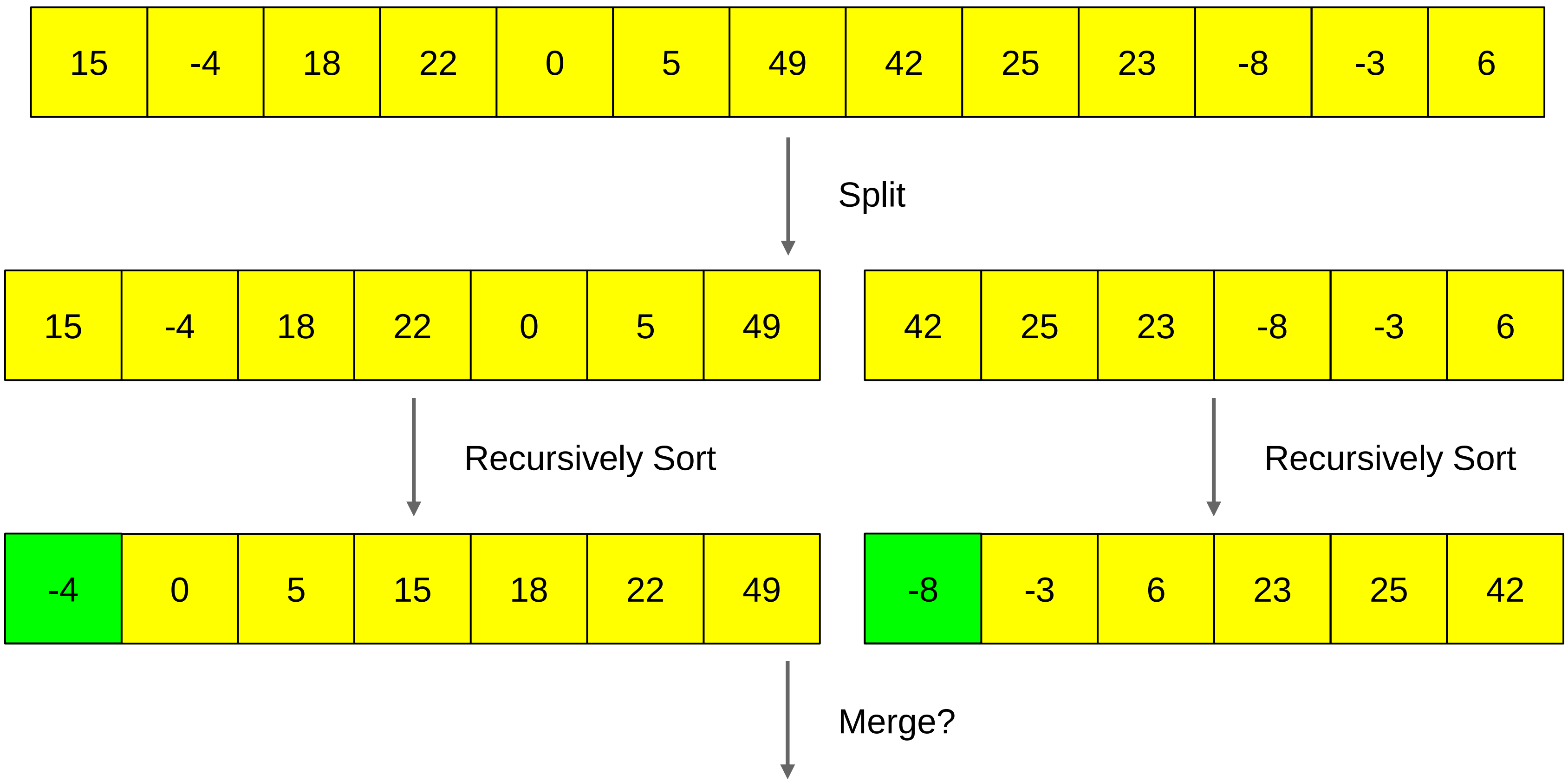
Two famous sorts behave this way: *mergesort* and *quicksort*.

Merge Sort

1. Split the array into two roughly equal pieces.
   - split by index: `[first..mid]` and `[mid+1..last]`
2. Recursively sort each half.
   - two recursive calls to `sort()`
   - assume smaller cases are sorted correctly
3. Join the two pieces together to make one sorted array.
   - Q. How can you quickly combine two sorted pieces into one?
   - *Merge* the two arrays



Example



Merge strategy is similar to Selection Sort: repeatedly find the min and place it.

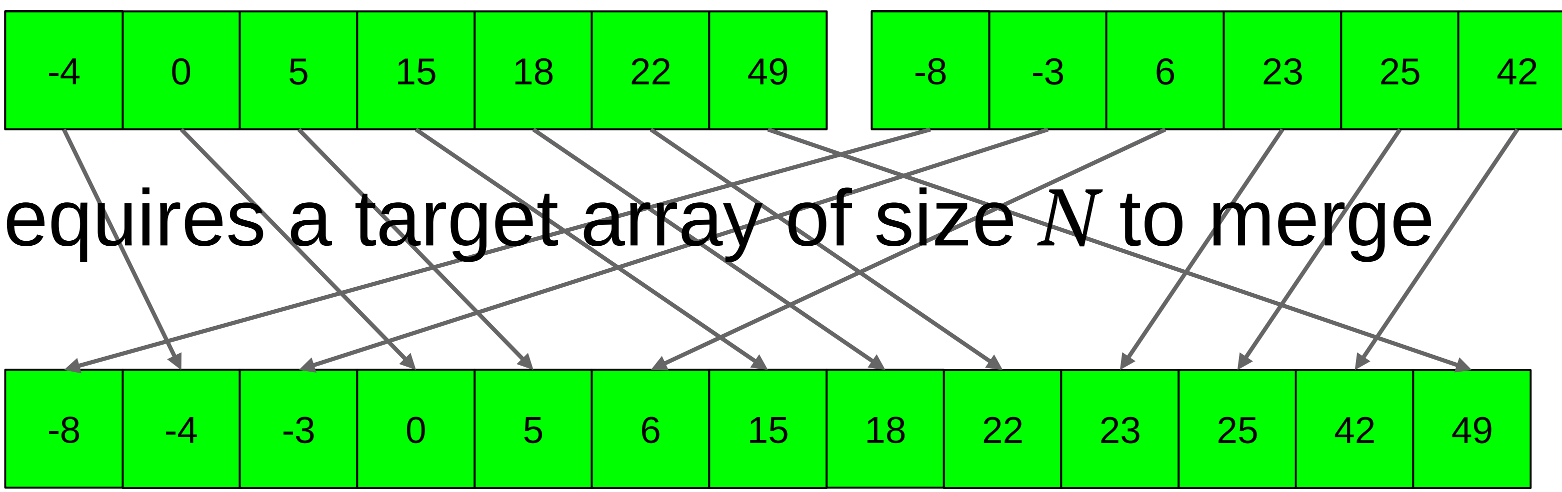
Q. How much time is required to find the min?

- it must be one of the heads of the two sorted subarrays. $\Rightarrow O(1)$

Merge Example

Strategy:

1. Find the min. Where is it?
    - It must be one of the heads of the two sorted subarrays
    - Compare and take the smaller.
2. Place the min into the next sequential position.



Requires a target array of size N to merge

MergeSort Code

```
//  Post:  arr[first..last] are sorted
void mergeSort(int arr[], int first, int last) {
```

- Base case
  - return if fewer than 2 elements
- Split array into two roughly equal pieces
  - compute `mid` element
- Recursively sort each piece
- Join the two sorted pieces together by merging
  - place the smallest min of each sorted piece

```
}
```

MergeSort Code

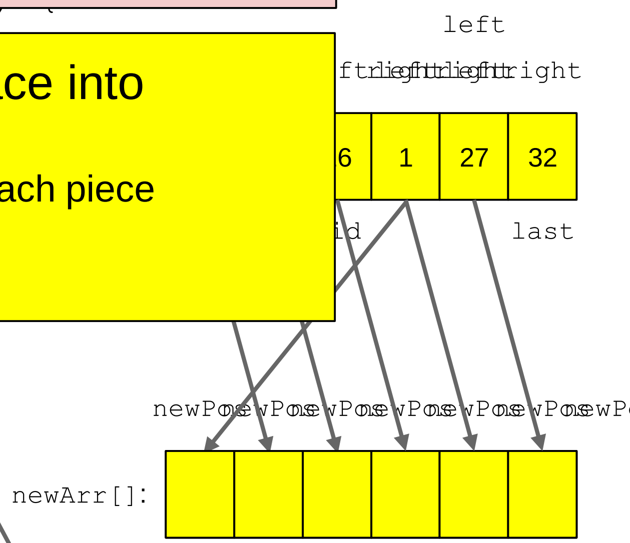
```
//  Post:  arr[first..last] are sorted
void mergeSort(int arr[], int first, int last) {
    //  Base case
    if (last <= first) return;

    //  Split array
    int mid = (first+last) / 2;

    //  Recursively sort
    mergeSort(arr, first, mid);
    mergeSort(arr, mid+1, last);

    //  Join
    merge(arr, first, mid, last);
}
```

Merge Code

```
//  Pre:  arr[first..mid] and arr[mid+1..last] are sorted
//  Post:  arr[first..last] are sorted
void merge(int arr[], int first, int mid, int last) {
```

An array bounds error occurs when you run out of elements from the left piece or on the right piece.

- Repeat for `N` elements
  - Take the smallest unplaced element and place into position
    - Maintain indices `left`, `right` for the heads of each piece
    - Compare the heads
    - Place the min in sequence into a temporary array




```
    arrCpy(newArr, arr + first, len);
 }
```

post-increment operator. Equivalent code:

```
newArr[newPos] = arr[left];
left++;
```

Merge Code

```
//  Pre:  arr[first..mid] and arr[mid+1..last] are sorted
//  Post:  arr[first..last] are sorted
void merge(int arr[], int first, int mid, int last) {
    int len = last-first+1;  int newArr[len];
    int left = first;  int right = mid+1;
    for (int newPos = 0; newPos < len; newPos++) {
        if (arr[left] < arr[right]) {
            newArr[newPos] = arr[left++];
        } else {
            newArr[newPos] = arr[right++];
        }
    }
    // arrCpy(source, destination, number of elements)
    arrCpy(newArr, arr + first, len);
}
```

post-increment operator. Equivalent code:

```
newArr[newPos] = arr[left];
left++;
```

A Bug!

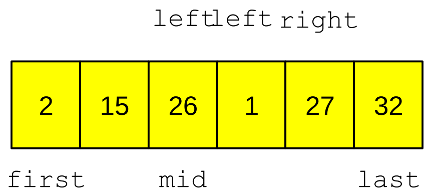
The merge strategy:

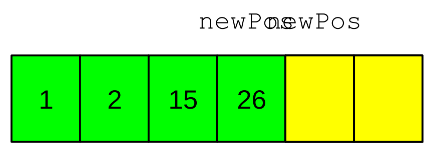
- Take the smallest [remaining] element of each sorted piece and place into position
- Fails when one piece runs out of elements

Solutions:

- Append $+\infty$ to the end of each piece
  - good in theory, but has practical issues
- Copy remaining elements from unfinished piece
  - a while loop will be required

Merge Code - Fixed






```
//  Pre:  arr[first..mid] and arr[mid+1..last] are sorted
//  Post:  arr[first..last] are sorted
void merge(int arr[], int first, int mid, int last) {
    int len = last-first+1;  int newArr[len];
    int left = first;  int right = mid+1;  int newPos = 0;
    while(left <= mid && right <= last) {
        if (arr[left] < arr[right]) {
            newArr[newPos++] = arr[left++];
        } else {
            newArr[newPos++] = arr[right++];
        }
    }
    //  Flush non empty piece
    arrCpy(arr + left, newArr + newPos, mid - left + 1);
    arrCpy(arr + right, newArr + newPos, last - right + 1);

    arrCpy(newArr, arr + first, len);
}
```

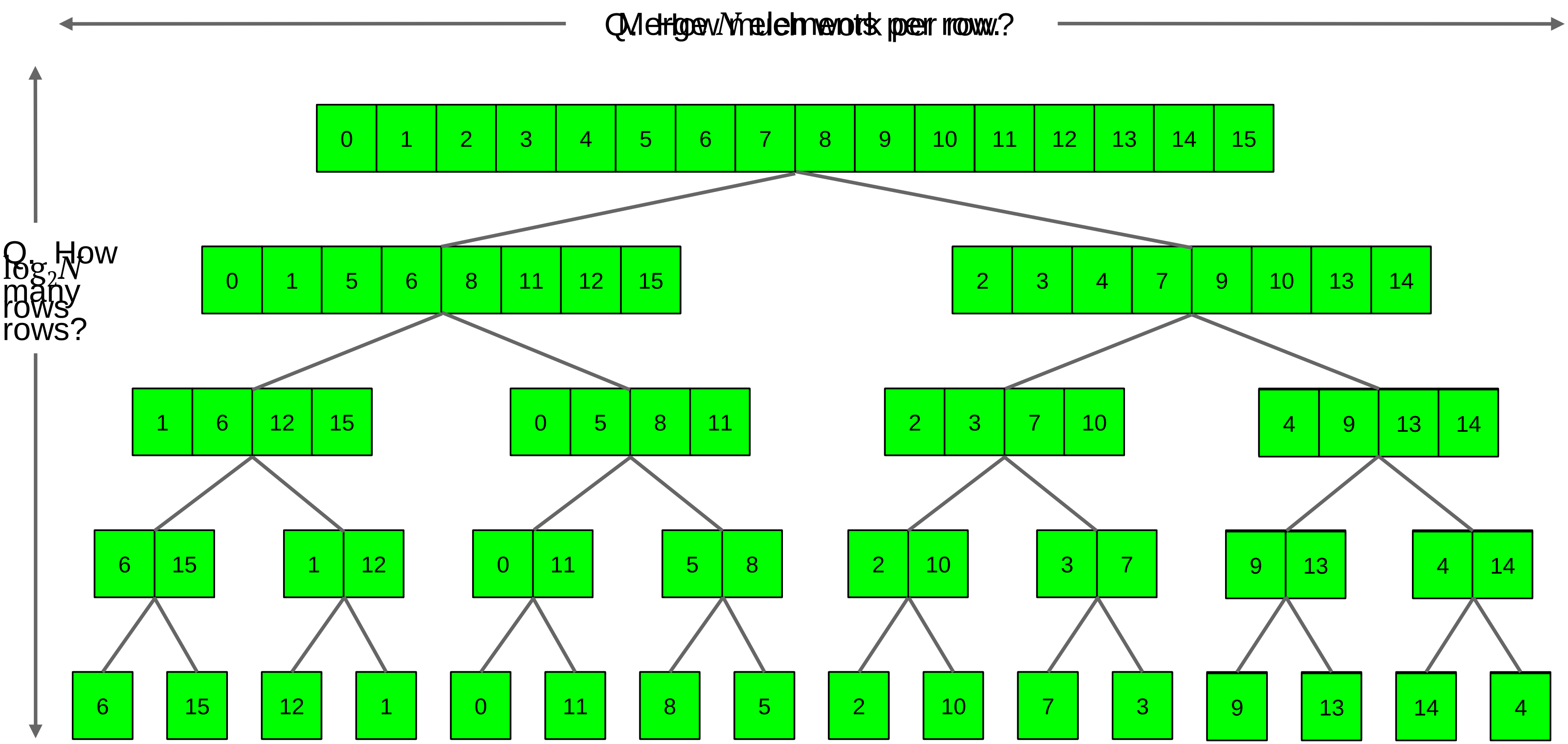
Q. What's the running time for `merge()`?

Running Time Analysis

- $O(N)$ work per row
- $O(\log N)$ rows

$\Rightarrow$ $O(N \log N)$ running time

Visualize with a *recursion tree*:



Visualization