Sorting Algorithms
Chapter 9

CS 3358
Spring 2015

Jill Seaman

Sections 9.1, 9.2, 9.3, 9.5, 9.6

What is sorting?

- Sort: rearrange the items in a list into ascending or descending order
  - numerical order
  - alphabetical order
  - etc.

55 112 78 14 20 179 42 67 190 7 101 1 122 170 8

1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

Why is sorting important?

- Searching in a sorted list is much easier than searching in an unsorted list.
- Especially for people:
  - dictionary entries (in a dictionary book)
  - phone book (remember these?)
  - card catalog in library (it used to be drawers of index cards)
  - bank statement: transactions in date order
- Most of the data displayed by computers is sorted.

Sorting

- Sorting is one of the most intensively studied operations in computer science
- There are many different sorting algorithms
- The run-time analyses of each algorithm are well-known.

Sorting algorithms covered in this class

- Selection sort
- Insertion sort
- Bubble sort

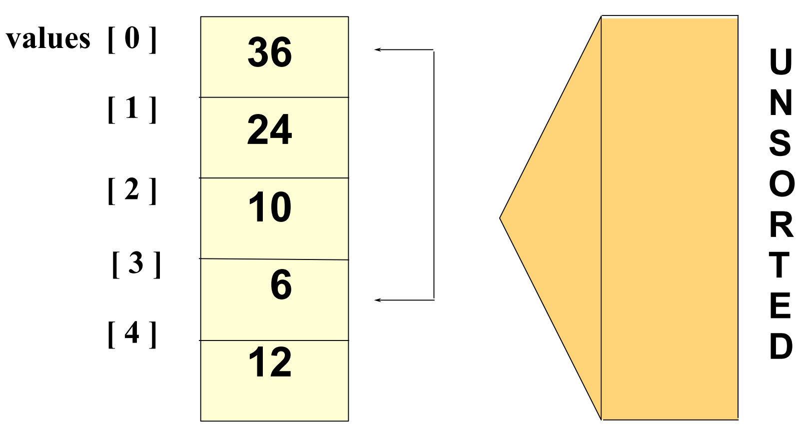
- Merge sort
- Quicksort

- Heap sort (later, when we talk about heaps)

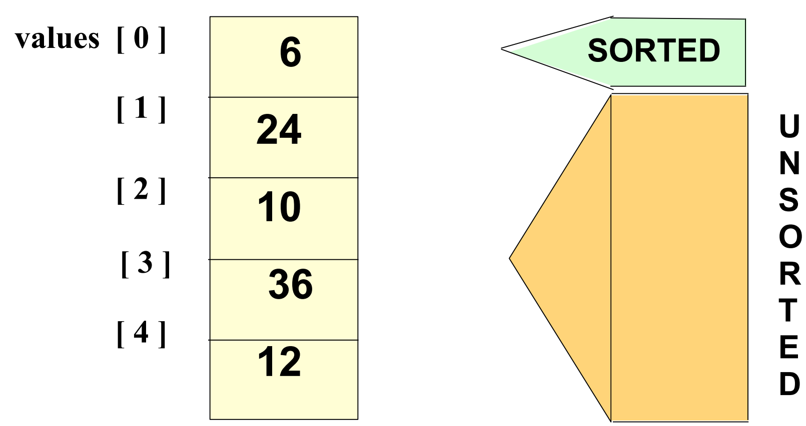
Selection sort

- There is a pass for each position (0..size-1)
- On each pass, the smallest (minimum) element in the rest of the list is exchanged (swapped) with element at the current position.
- The first part of the list (already processed) is always sorted
- Each pass increases the size of the sorted portion.

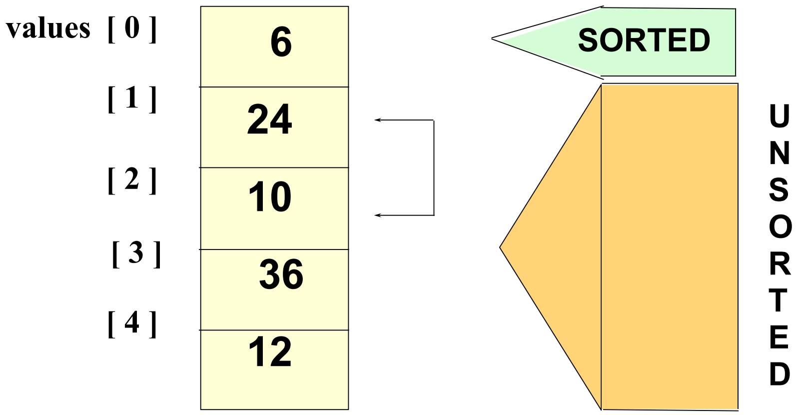
Selection Sort: Pass One



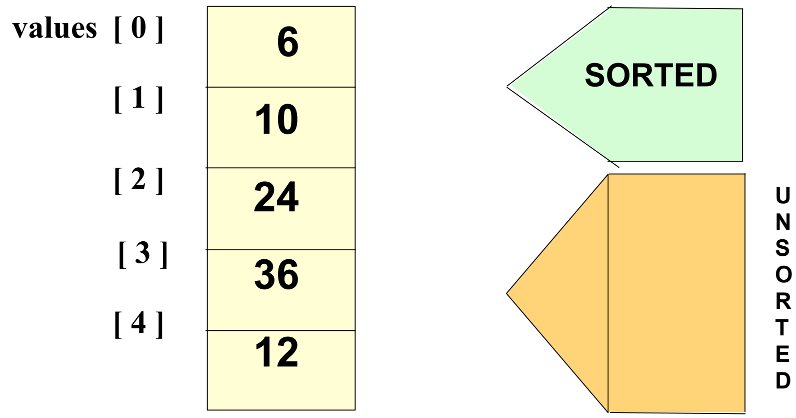
Selection Sort: End Pass One



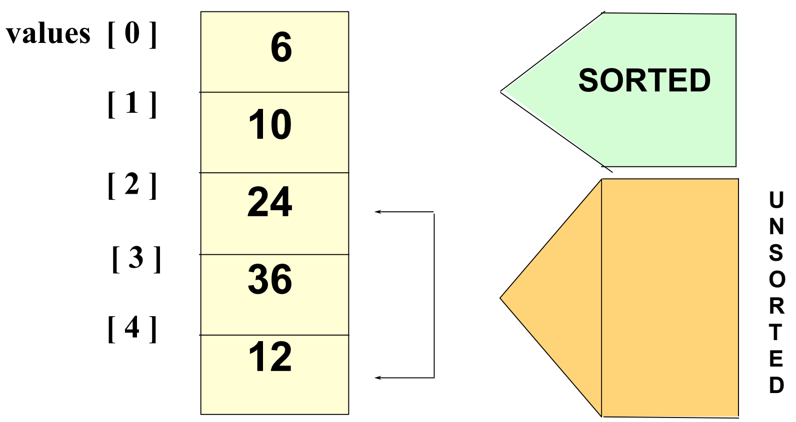
Selection Sort: Pass Two



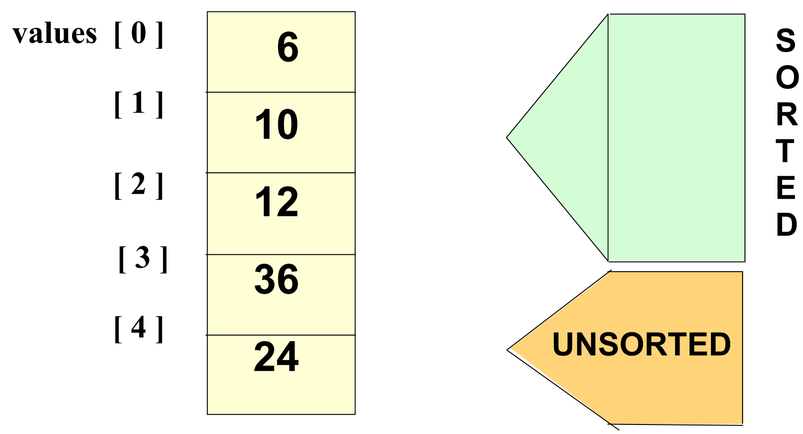
Selection Sort: End Pass Two



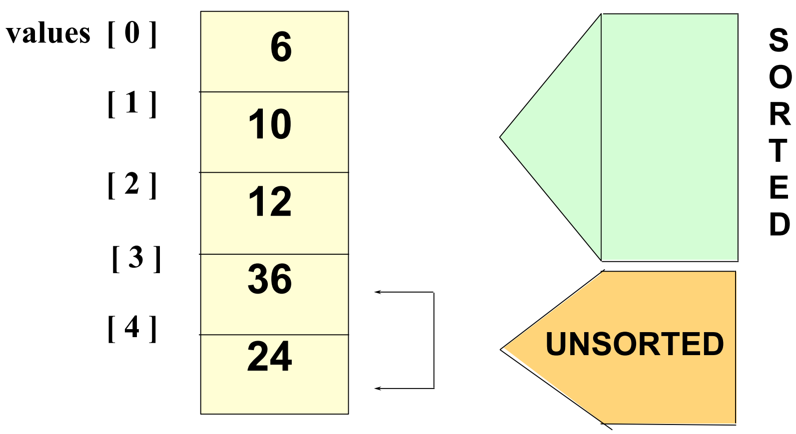
Selection Sort: Pass Three



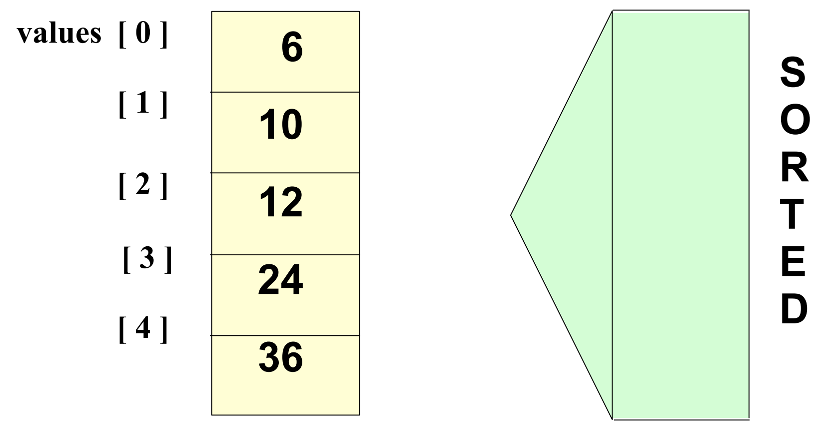
Selection Sort: End Pass Three



Selection Sort: Pass Four



Selection Sort: End Pass Four



Selection sort: code

```
template<class ItemType>
int minIndex(ItemType values[], int size, int start) {
    int minIndex = start;
    for (int i = start+1; i < size; i++)
        if (values[i] < values[minIndex])
            minIndex = i;
    return minIndex;
}

template<class ItemType>
void selectionSort (ItemType values[], int size) {
    int min;
    for (int index = 0; index < (size -1); index++) {
        min = minIndex(values, SIZE, index);
        swap(values[min],values[index]);
    }
}
```

template <class T> void swap (T& a, T& b); is in the <algorithm> library

Efficiency of Selection Sort

- N is the number of elements in the list
- Outer loop (in selectionSort) executes N-1 times
- Inner loop (in minIndex) executes N-1, then N-2, then N-3, ... then once.
- Total number of comparisons (in inner loop):

(N-1) + (N-2) + . . . + 2 + 1 = the sum of 1 to N-1

From math class: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

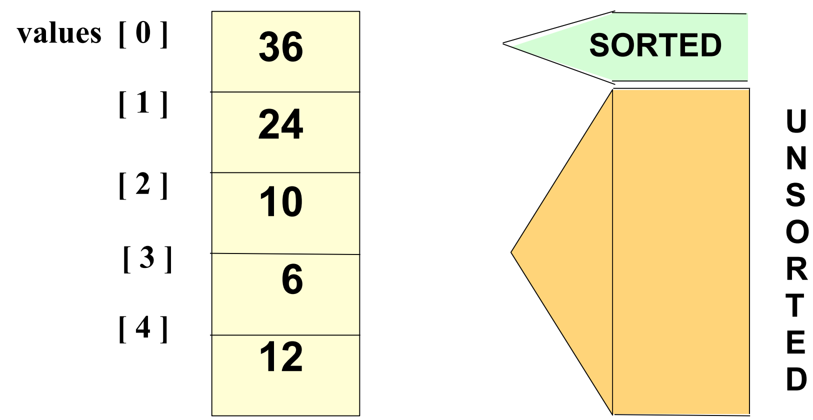
(N-1) + (N-2) + . . . + 2 + 1 = (N-1)(N-1+1)/2
= (N-1)N/2
= $(N^2$-N)/2
= $N^2/2$ - N/2

$O(N^2)$

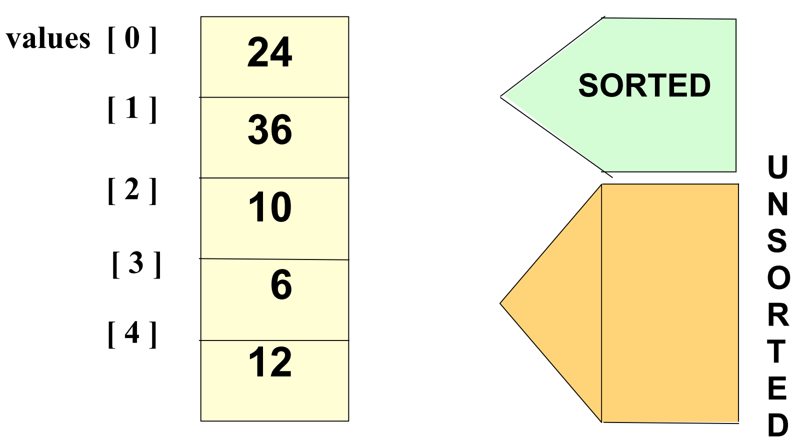
Insertion sort

- There is a pass for each position (0..size-1)
- The front of the list remains sorted.
- On each pass, the next element is placed in its proper place among the already sorted elements.
- Like playing a card game, if you keep your hand sorted, when you draw a new card, you put it in the proper place in your hand.
- Each pass increases the size of the sorted portion.

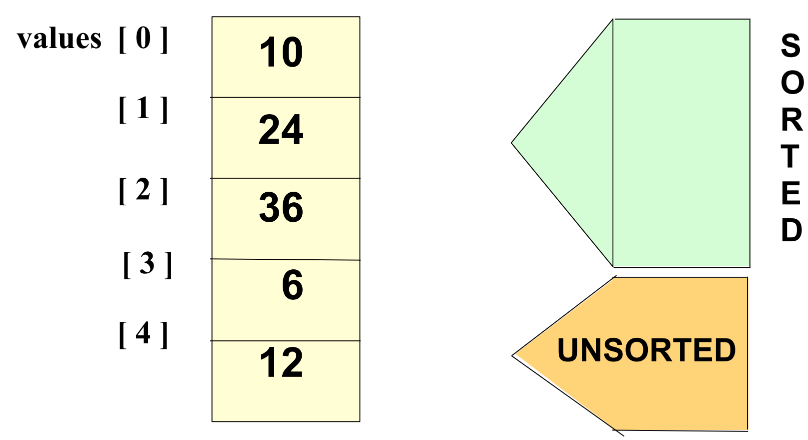
Insertion Sort: Pass One



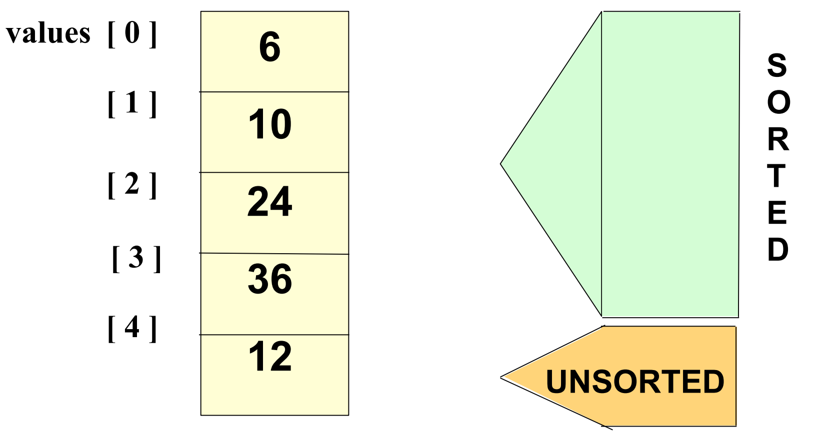
Insertion Sort: Pass Two



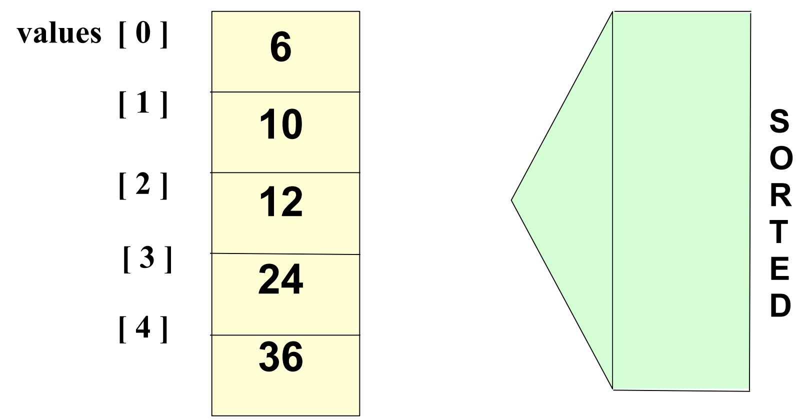
Insertion Sort: Pass Three



Insertion Sort: Pass Four



Insertion Sort: Pass Five



Insertion sort: code

```
template<class ItemType>
void insertionSort (ItemType a[], int size) {

    for (int index = 1; index < size; index++) {
        ItemType tmp = a[index];   // next element

        int j = index;     // start from the end of sorted part

        // find tmp's place, AND shift bigger elements up
        while (j > 0 && tmp < a[j-1]) {
            a[j] = a[j-1];   // shift bigger element up
            j--;
        }
        a[j] = tmp;          // put tmp in its place
    }
}
```


Insertion sort: runtime analysis

- Very similar to Selection sort
- Total number of comparisons (in inner loop):
  - At most 1, then 2, then 3 ... up to N-1 for the last element.
- So it's

$$(N-1) + (N-2) + \ldots + 2 + 1 == N^2/2 - N/2$$

$O(N^2)$

Bubble sort

- On each pass:
  - Compare first two elements. If the first is bigger, they exchange places (swap).
  - Compare second and third elements. If second is bigger, exchange them.
  - Repeat until last two elements of the list are compared.
- Repeat this process until a pass completes with no exchanges

Bubble sort

Example: first pass

- 7 2 3 8 9 1 7 > 2, swap
- 2 7 3 8 9 1 7 > 3, swap
- 2 3 7 8 9 1 !(7 > 8), no swap
- 2 3 7 8 9 1 !(8 > 9), no swap
- 2 3 7 8 9 1 9 > 1, swap
- 2 3 7 8 1 9 finished pass 1, did 3 swaps

Note: largest element is in last position

Bubble sort

Example: second and third pass

- 2 3 7 8 1 9 2<3<7<8, no swap, !(8<1), swap
- 2 3 7 1 8 9 (8<9) no swap
- finished pass 2, did one swap

2 largest elements in last 2 positions

- 2 3 7 1 8 9 2<3<7, no swap, !(7<1), swap
- 2 3 1 7 8 9 7<8<9, no swap
- finished pass 3, did one swap

3 largest elements in last 3 positions

Bubble sort

Example: passes 4, 5, and 6

- 2 3 1 7 8 9 2<3, !(3<1) swap, 3<7<8<9
- 2 1 3 7 8 9
- finished pass 4, did one swap
- 2 1 3 7 8 9 !(2<1) swap, 2<3<7<8<9
- 1 2 3 7 8 9
- finished pass 5, did one swap
- 1 2 3 7 8 9 1<2<3<7<8<9, no swaps
- finished pass 6, no swaps, list is sorted!

Bubble sort
how does it work?

- At the end of the first pass, the largest element is moved to the end (it's bigger than all its neighbors)
- At the end of the second pass, the second largest element is moved to just before the last element.
- The back end (tail) of the list remains sorted.
- Each pass increases the size of the sorted portion.
- No exchanges implies each element is smaller than its next neighbor (so the list is sorted).

Bubble sort: code

```
template<class ItemType>
void bubbleSort (ItemType a[], int size) {

    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < (size-1); i++) {
            if (a[i] > a[i+1]) {
                swap(a[i],a[i+1]);
                swapped = true;
            }
        }
    } while (swapped);
}
```


Bubble sort: runtime analysis

- Each pass makes N-1 comparisons
- There will be at most N passes
  - one to move the right element into each position
- So worst case it's: $(N-1)*N$ **$O(N^2)$**
- If you change the algorithm to look at only the **unsorted** part of the array in each pass, it's exactly like the selection sort:

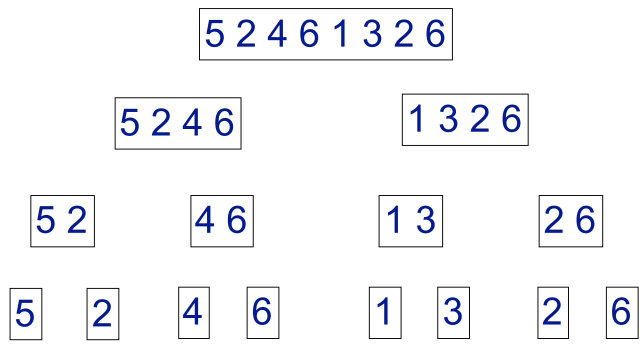
  $(N-1) + (N-2) + . . . + 2 + 1 = N^2/2 - N/2$ **still $O(N^2)$**
- What is the best case for Bubble sort?
- Are there any sorting algorithms better than $O(N^2)$?

Merge sort

- Divide and conquer!
- 2 half-sized lists sorted recursively
- the algorithm:
  - if list size is 0 or 1, return (base case) otherwise:
  - recursively sort first half and then second half of list.
  - merge the two sorted halves into one sorted list.
    - choose the smaller of the two first elements, move it to the end of the new sorted list.
    - repeat until one list is empty.
    - move the remaining list's elements to the end of the new sorted list.

Merge sort
Example

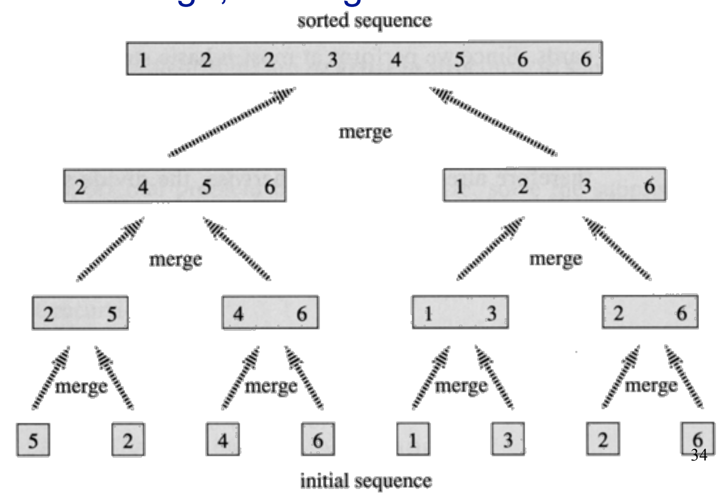
- **Recursively** divide list in half:
  - call mergeSort recursively on each one.



Each of these are sorted (base case length = 1)

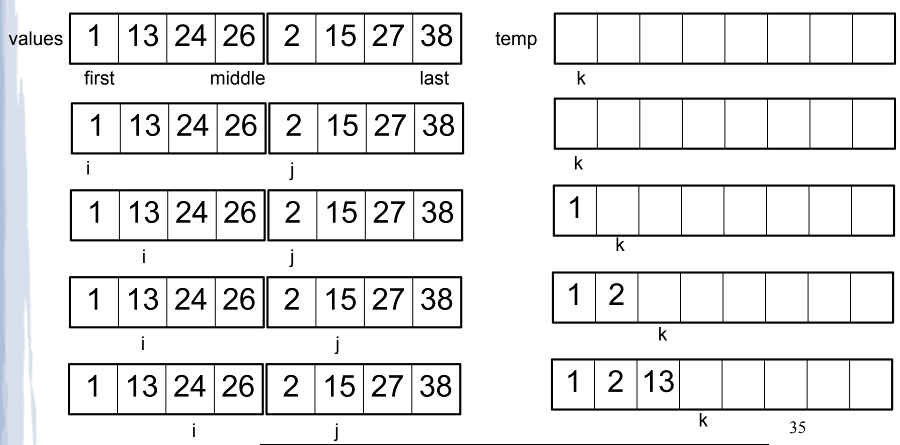
Merge sort
Example

- Calls to merge, starting from the bottom:



Merge sort
Merging

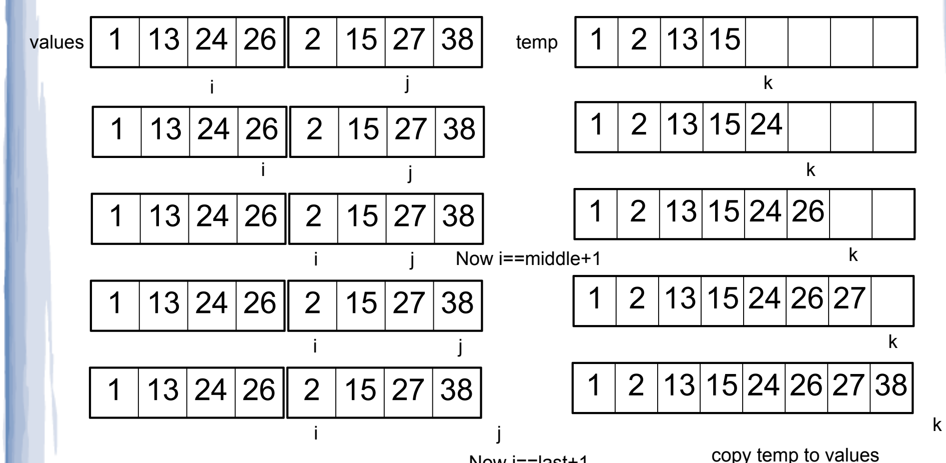
- How to merge 2 (adjacent) lists:



compare values[i] to values[j], copy smaller to temp[k]

Merge sort
Merging

- Continued:



Merge sort: code

```
template<class ItemType>
void mergeSortRec (ItemType values[], int first, int last) {
    if (first < last) {
        int middle = (first + last) / 2;

        mergeSortRec(values, first, middle);
        mergeSortRec(values, middle+1, last);

        merge(values, first, middle, last);
    }
}

template<class ItemType>
void mergeSort (ItemType values[], int size) {
    mergeSortRec(values, 0, size-1);
}
```


Merge sort: code: merge

```
template<class ItemType>
void merge(ItemType values[], int first, int middle, int last) {

    ItemType tmp[last-first+1];  //temporary array, could use dynamic alloc.
    int i=first;         //index for left
    int j=middle+1;      //index for right
    int k=0;             //index for tmp

    while (i<=middle && j<=last)   //merge, compare next elem from each array
        if (values[i] < values[j])
            tmp[k++] = values[i++];
        else
            tmp[k++] = values[j++];

    while (i<=middle)            //merge remaining elements from left, if any
        tmp[k++] = values[i++];

    while (j<=last)              //merge remaining elements from right, if any
        tmp[k++] = values[j++];

    for (int i = first; i <=last; i++) //copy from tmp array back to values
        values[i] = tmp[i-first];
}
```


Merge sort: runtime analysis

- Let's start with a run-time analysis of **merge** (of 2 sorted sublists into one sorted list)
- Let's use M as the size of the final list
  - The merging requires M (or fewer) comparisons +copies
  - Copying from the temp array is M copies
  - So merge is O(M), worst case

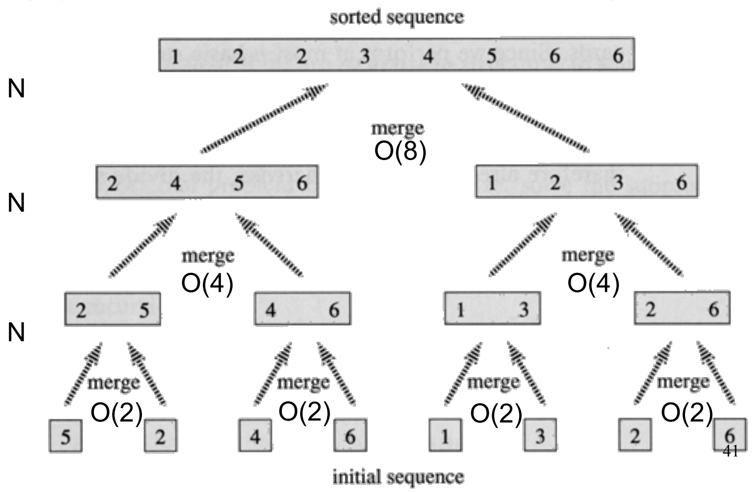
Merge sort: runtime analysis

- **At each level** in the graph (except the top)
  - merge is called on each sub-list
  - The **total** size of each sub-list (M) added up is N
  - the total execution time is O(N) for the level.
- There are $\log_2 N$ levels in the graph (== how many times can I divide N by 2 (until it's <=1))?
- So $\log_2 N$ levels times O(N) at each level:

O(N Log N)

Merge sort
Runtime analysis

- O(N) work done between each level (for merging):



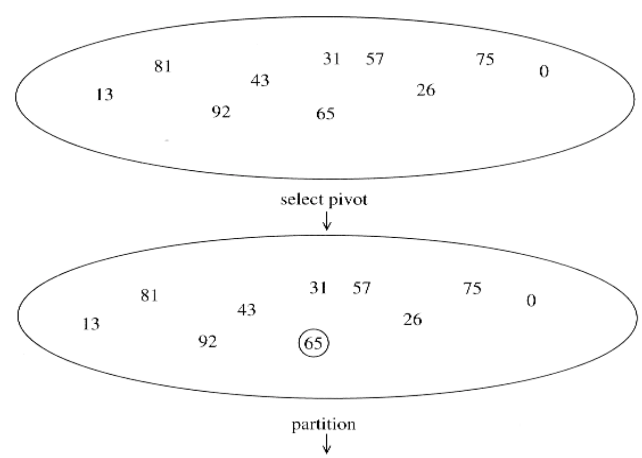
Merge sort: runtime analysis

- mergeSort has 2 recursive calls to itself.
- Why does it not have the exponential cost that the Fibonacci algorithm had?

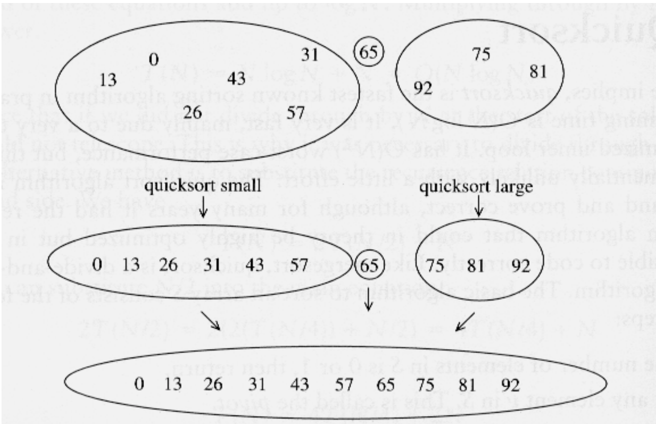
Quicksort

- Another divide and conquer!
- 2 (hopefully) half-sized lists sorted recursively
- the algorithm:
  - If list size is 0 or 1, return. otherwise:
  - **partition** into two lists:
    - pick one element as the “pivot” element
    - put all elements less than pivot in first half
    - put all elements greater than pivot in second half
  - recursively sort first half and then second half of list.

Quicksort
Example

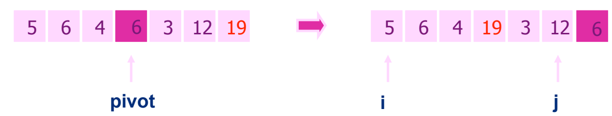


Quicksort
Example cont.



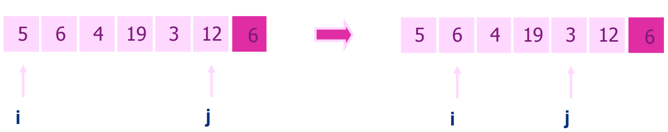
Quicksort: partitioning

- Goal: partition a sub-array A [start ... last] by rearranging the elements and returning the index of the pivot point p so that:
  - A[x]<=A[p] for x<p and A[x]>=A[p] for x>p
- the algorithm:
  - pick a pivot elem and swap with last elem
  - let i = first and j = last -1



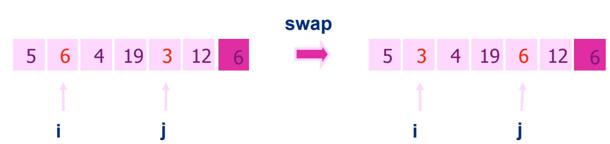
Quicksort: partitioning

- the algorithm (continued):
  - increment i while A[i] < A[last] (the pivot)
  - decrement j while A[j] > A[last] (the pivot)



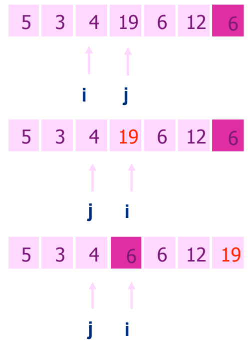
Quicksort: partitioning

- the algorithm (continued):
  - When i and j have stopped,
  - if i < j: swap A[i] and A[j]; i++; j--;
  - maintains: A[x] <= pivot for x<=i and A[x] >= pivot for x >= j



Quicksort: partitioning

- the algorithm (continued):
  - repeat until i and j have met (i==j)* or crossed (i > j):
  - swap A[i] and pivot (A[last])
    - A[i] >= pivot (i stopped there, so A[i] >= pivot)
    - puts pivot back in place
  - return i (the pivot index)



*Note: if i==j, A[j] must be the pivot value, otherwise either A[i]<pivot so i would not have stopped, or A[j]>pivot, so j would not have stopped.

Quicksort: code
version 1

```
template<class ItemType>
void quickSort (ItemType values[], int first, int last) {

    if (first < last) {    //at least two elems
        int pivotPoint;

        // partition and get the pivot point (the index)
        pivotPoint = partition(values, first, last);

        quickSort(values, first, pivotPoint - 1);
        quickSort(values, pivotPoint + 1, last);
    }
}

template<class ItemType>
void quickSort (ItemType values[], int size) {
    quickSort(values, 0, size-1);
}
```


Quicksort: code
version 1

```
template<class ItemType>
int partition(ItemType values[], int first, int last) {

    int mid = (first + last) / 2;  //use middle value as pivot

    ItemType pivotValue = values[mid];
    swap(values[last], values[mid]);  //move pivot to end

    int i=first,j=last-1;
    while (i<j) {
        while (values[i] < pivotValue) {i++;}
        while (values[j] > pivotValue) {j--;}
        if (i < j) {
            swap(values[i++], values[j--]);
        }
    }
    swap(values[i], values[last]);   //replace pivot
    return i;
}
```


Quicksort: runtime analysis

- Choice of pivot point dramatically affects running time.
- Best Case
  - Pivot partitions the set into 2 **equally sized** subsets at each stage of recursion: O(log N) levels (== how many times can I divide N by 2 (until it's <=1))
  - Partitioning at each level is O(N)
    - each element is compared to the pivot and maybe moved one time
  - O(N log N)

Quicksort: runtime analysis

- Worst Case
  - Pivot is always the smallest element, partitioning the set into **one empty subset**, and **one of size N-1**.
  - Partitioning at each level is N
    - T(N) = T(N-1) + N (time to sort N-1 plus N for partitioning)
    - T(N) = N + N-1 + . . . + 2 + 1 (from unwinding the above)
    - T(N) = N(N+1)/2
  - $O(N^2)$

Moral of the story: it pays to pick a good pivot point

Quicksort: runtime analysis

- Average Case
  - Assume left side is equally likely to have a size of 0 or 1 or 2 or ... or N elements (a random distribution)
  - O(N log N)
    - Not a trivial proof . . . most of it is in the Weiss book.

Quicksort: Picking the pivot

- Goal: ensure the worst case doesn't happen.
- Picking a pivot **randomly** is safe
  - but random number generation can be expensive
- Using the first element:
  - if the input is randomly ordered, this is ok.
  - if the input is sorted, all elements are in right half, this is the worst case = $O(N^2)$
- Use the median value (the middle value in order):
  - perfectly divides into two even sides
  - but you have to sort the list to find the median.

Quicksort: Picking the pivot

Median of Three method

- For the pivot value:
  - Take the three values at the first, last, and middle positions in the list.
  - Throw out the max and min values.
  - Use the remaining value as the pivot value.
- This is an "estimate" of the real median
  - taking median of more than 3 is not worth the time

Quicksort: Picking the pivot

Median of Three method

- Median-of-Three partitioning:
  - arrange the values (by swapping) at the first, last and middle positions so that:

    A[first] <= A[middle] <= A[last]

  - (this puts the median of the 3 in the middle).
  - swap pivot (A[middle]) with A[last-1].
  - start with i = first+1 and j=last-2 (A[first] and A[last] are already in place).
  - use same algorithm as original partitioning.

Quicksort: Picking the pivot

Median of Three method

2	5	6	4	13	3	12	19	6	A[left] = 2, A[center] = 13, A[right] = 6
2	5	6	4	6	3	12	19	13	Swap A[center] and A[right], so the three values are in order
2	5	6	4	6 (pivot)	3	12	19	13	Choose A[center] as pivot
2	5	6	4	19	3	12	6 (pivot)	13	Swap pivot and A[right – 1]

Now we only need to partition A[left + 1, …, right – 2].

Quicksort: Small Arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort
  - a cutoff between 5 and 20 is good.
  - Note: median of three partitioning requires at least 3 elements anyway

Quicksort: code

version 2

```
const in CUTOFF = 10;

template<class ItemType>
void quickSort (ItemType values[], int first, int last) {
    int pivotPoint;
    if (first + CUTOFF <= last) {  // more than CUTOFF elems
        pivotPoint = partition(values, first, last);
        quickSort(values, first, pivotPoint - 1);
        quickSort(values, pivotPoint + 1, last);
    } else {
        insertionSort(values, first,last);  //base case
    }
}

template<class ItemType>
void quickSort (ItemType values[], int size) {
    quickSort(values, 0, size-1);
}
```

Note: rewrite insertion sort for this signature:

```
insertionSort(ItemType values[],
              int first, int last);
```

Quicksort: code

version 2

Median of three partitioning

```
template<class ItemType>
int partition(ItemType values[], int first, int last) {
    //sort first, mid, last
    int mid = (first + last) / 2;
    if (values[mid] < values[first])  swap(values[mid], values[first]);
    if (values[last] < values[first]) swap(values[last], values[first]);
    if (values[last] < values[mid])   swap(values[last], values[mid]);

    ItemType pivotValue = values[mid];  // move pivot to last-1
    swap(values[last-1], values[mid]);

    int i=first+1,j=last-2;             // do the partitioning
    while (i<j) {
        while (values[i] < pivotValue) {i++;}
        while (pivotValue < values[j]) {j--;}
        if (i < j)
            swap(values[i++], values[j--]);
    }
    swap(values[i], values[last-1]);    // put pivot back in place
    return i;
}
```


Quicksort vs MergeSort

- Both run in O(N log N)
- Compared with Quicksort, Merge sort has fewer comparisons but more swapping (copying)
  - (not yet able to verify the following):
  - In Java, an element comparison is expensive but moving elements is cheap. Therefore, Merge sort is used in the standard Java library for generic sorting
  - In C++, copying objects can be expensive while comparing objects often is relatively cheap. Therefore, quicksort is the sorting routine commonly used in C++ libraries