What is sorting?

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

Why is sorting important?

- Searching in a sorted list is much easier than searching in an unsorted list.
- Especially for people
  - dictionary entries
  - phone book
  - card catalog in library
  - 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

- 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: code

```cpp
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]);
    }
}
```

Selection sort: runtime analysis

- N is the number of elements in the list
- Outer loop (in selectionSort) executes N times
- Inner loop (in minIndex) executes N-1, then N-2, then N-3, ... then once.
- Total number of comparisons (in inner loop):
  \[(N-1) + (N-2) + \ldots + 2 + 1\]
  
  Note: \[N + (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2\]
  
  \[=N^2/2 + N/2\]

  Subtract N from both sides:
  \[(N-1) + (N-2) + \ldots + 2 + 1 = N^2/2 + N/2 - N\]
  
  \[=N^2/2 - N/2\]

  \[O(N^2)\]
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 Four**

- **values**
  - 0: 6
  - 1: 10
  - 2: 24
  - 3: 36
  - 4: 12

**SORTED**

**UNSORTED**

---

**Insertion Sort: Pass Five**

- **values**
  - 0: 6
  - 1: 10
  - 2: 12
  - 3: 24
  - 4: 36

**SORTED**

---

**Insertion sort: code**

```cpp
template<class ItemType>
void insertionSort (ItemType a[], int size) {
    for (int index = 0; index < size; index++) {
        ItemType tmp = a[index];  // next element
        int j = index;            // start from the end
        // find tmp's place, AND shift bigger elements up
        while (j > 0 && tmp < a[j-1]) {
            a[j] = a[j-1];        // shift
            j--;                   // move back
        }
        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 j+1, which is 1, then 2, then 3 ... up to N
- So it's $O(N^2)$

\[ N + (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N+1)}{2} \]
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

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

Example

- 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

- 2 3 7 1 8 9  2<3<7, 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

- 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: code

```cpp
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: \((N-1)^N\) \(O(N^2)\)

- What is the best case?

- 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.
Merge sort

Example

• Recursively divide list in half:

```
5 2 4 6 1 3 2 6
5 2
4 6
1 3
2 6
5 2 4 6 1 3 2 6
```

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

• Calls to merge, starting from the bottom:

```
sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
merge
1 2 3 6
merge
```

Merge sort: code

```
template<class ItemType>
void mergeSort (ItemType values[], int first, int last) {
    if (first < last) {
        int middle = (first + last) / 2;
        mergeSort(values, first, middle);
        mergeSort(values, middle+1, last);
        merge(values, first, middle, last);
    }
}

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

```
template<class ItemType>
void merge(ItemType values[], int first, int middle, int last) {
    ItemType tmp[last-first+1];  //temporary array
    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
- Let’s use $M$ as the size of the final list
  - The merging requires $M$ (or fewer) comparisons
  - Copying from the temp array is $M$ copies
  - So merge is $O(M)$

The array can be subdivided into halves $\log_2 N$ times (there are $\log_2 N$ levels in the graph)

- At each level in the graph,
  - merge is called on each sub-list
  - The total size of each sub-list added up is $N$
  - So at each level in the graph, the total execution time is $O(N)$.

So $\log_2 N$ levels times $O(N)$ at each level:

$O(N \log N)$

Merge sort

- $O(N)$ work done at each level:

\[
\begin{array}{cccccccc}
N & 1 & 2 & 2 & 3 & 4 & 5 & 6 & 6 \\
\downarrow & & & & & & & & \\
2 & 4 & 5 & 6 & 1 & 2 & 3 & 6 \\
\downarrow & & & & & & & & \\
2 & 5 & 4 & 6 & 1 & 3 & 2 & 6 \\
\downarrow & & & & & & & & \\
5 & 2 & 4 & 6 & 1 & 3 & 2 & 6 \\
\end{array}
\]

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
    - 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 visualization

Quicksort Example

Quicksort Example cont.
QuickSort: partitioning

- Many algorithms, but must be efficient
- Goal: partition array A [start ... last] (start+last are indexes)
- Uses two indexes, i and j, starting from the front and the back of the list
- the algorithm:
  - swap pivot with last element (safekeeping)
    5 6 4 3 12 19
  - move small elements left and larger elements right.
    - let i start at first element, and j start at last-1
    - increment i while A[i] < pivot
    - decrement j while A[j] > pivot
  - When i and j have stopped, and i < j:
    - swap A[i] and A[j]
    - larger element goes to right side, smaller to left
  - the algorithm (continued):
    - When i and j have met or crossed (i >= j):
      - swap A[i] and pivot
      - puts pivot in place
      - A[i] >= pivot (i stopped there) so it stays on right side
      - return i (the pivot index)
Quicksort: partitioning

- What if all the elements are bigger than pivot?
  - i never moves, j doesn’t stop until it reaches i
  - pivot swapped with A[i], at front of list
  - all elements are to right of pivot
- What if all the elements are smaller than pivot?
  - i never stops until it is at the pivot, j doesn’t move
  - pivot swapped with itself, stays at end of list
  - all elements are to left of pivot

What if A[i] or A[j] is equal to the pivot?
- should they stop?
  - if all elements are identical:
    - i and j will always stop and swap at every position
    - lots of unnecessary swapping, but pivot ends up in the middle (good).
- if they don’t stop:
  - if all elements are identical:
    - i never stops until it is at the pivot
  - No swapping, but pivot ends up at end (bad)

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: 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
  - Partitioning at each level is $O(N)$
    - each element is compared to the pivot and maybe moved one time
    - $O(N \log N)$

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)
  - $T(N) = N + N-1 + \ldots + 2 + 1$ (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
  - Partitioning at each level is still N
    - $T(N) = \text{average cost of one recursive call, over all subproblem sizes}$
    - $T(N) = (T(0) + T(1) + \ldots + T(N-1)) / N$ (divide by N to get avg)
    - Cost for 2 recursive calls and one partitioning:
      - $T(N) = N + 2\cdot (T(0) + T(1) + \ldots + T(N-1)) / N$
  - Not a trivial proof . . . most of it is in the book.
  - $O(N \log N)$

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 random, this is ok.
  - if the input is sorted, all elements are in right half 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**

- Pivot is the median of the first, last, and middle value in the list.

- This is an “estimate” of the real median  
  - taking median of more than 3 is not worth the time

---

**Median-of-Three partitioning:**
- swap the values at first, last and middle so that:
  
  \[
  A[\text{first}] = \text{smallest}, \quad A[\text{middle}] = \text{median}, \quad A[\text{last}] = \text{biggest}
  \]

- swap pivot (median) with \(A[\text{last}-1]\)
- start with \(i = \text{first} + 1\) and \(j = \text{last} - 2\)
- increment \(i\) until it encounters an element smaller than pivot
- decrement \(j\) until it encounters an element bigger than pivot
- if \((i < j)\) swap \((A[i], A[j])\)

---

**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

```cpp
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);
    }
}

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

```cpp
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,j;                            // do the partitioning
    for (i=first+1, j=last-2; ; ) {
        while (values[i] < pivotValue) {i++;}
        while (pivotValue < values[j]) {j--;}
        if (i < j)
        swap(values[i++], values[j--]);
        else
        break;
    }
    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, Mergesort 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, Mergesort 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