Sorting

- Rearrange $a[0]$, $a[1]$, …, $a[n-1]$ into ascending order. When done, $a[0] \leq a[1] \leq ... \leq a[n-1]$
- $8, 6, 9, 4, 3 \Rightarrow 3, 4, 6, 8, 9$
Sort Methods

- Insertion Sort
- Bubble Sort
- Selection Sort
- Count Sort
- Shaker Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort
Insert An Element

- Given a sorted list/sequence, insert a new element
- Given 3, 6, 9, 14
- Insert 5
- Result 3, 5, 6, 9, 14
Insert an Element

- 3, 6, 9, 14 insert 5
- Compare new element (5) and last one (14)
  - Shift 14 right to get 3, 6, 9, , 14
  - Shift 9 right to get 3, 6, , 9, 14
  - Shift 6 right to get 3, , 6, 9, 14
  - Insert 5 to get 3, 5, 6, 9, 14
Insert An Element

// insert t into a[0:i-1]

int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
a[j + 1] = t;
Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements
Insertion Sort

- Sort 7, 3, 5, 6, 1
- Start with 7 and insert 3 => 3, 7
- Insert 5 => 3, 5, 7
- Insert 6 => 3, 5, 6, 7
- Insert 1 => 1, 3, 5, 6, 7
Insertion Sort

```java
for (int i = 1; i < a.length; i++) {
    // insert a[i] into a[0:i-1]
    // code to insert comes here
}
```
Insertion Sort

for (int i = 1; i < a.length; i++) {
    // insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}

Complexity

- Space/Memory
- Time
  - Count a particular operation
  - Count number of steps
  - Asymptotic complexity
Memory Usage by Recursion

1. `int factorial(int n)`
2. `// Compute n!`
3. `if (n <= 1) return 1;`
4. `else return n * factorial(n - 1);`
5. `}

Number of recursive calls made: `max(n, 1)`

Each call uses 4 bytes for return address and 4 for parameter.

Total memory used: `8 * max(n, 1)`
Permutations

```cpp
void permutations(T list[], int k, int m)
{ // Generate all permutations of list[k:m].
  int i;
  if (k == m) { // list[k:m] has one permutation, output it
    copy(list, list+m+1,
         ostream_iterator<T>(cout, ""));
    cout << endl;
  }
  else { // list[k:m] has more than one permutation
    // generate these recursively
    for (i = k; i <= m; i++)
      {
        swap(list[k], list[i]);
        permutations(list, k+1, m);
        swap(list[k], list[i]);
      }
}
```
Comparison Count

```java
for (int i = 1; i < a.length; i++) {
    // insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
```
Comparison Count

- Pick an instance characteristic ... $n, n = a.length$ for insertion sort
- Determine count as a function of this instance characteristic.
Comparison Count

for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];

How many comparisons are made?
Comparison Count

for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];

number of compares depends on a[]s and t as well as on i
Comparison Count

- Worst-case count = maximum count
- Best-case count = minimum count
- Average count
Worst-Case Comparison Count

for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];

a = [1, 2, 3, 4] and t = 0 => 4 compares
a = [1,2,3,...,i] and t = 0 => i compares
Worst-Case Comparison Count

for (int i = 1; i < n; i++)
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];

total compares = 1 + 2 + 3 + … + (n-1)
    = (n-1)n/2
Step Count

A step is an amount of computing that does not depend on the instance characteristic n

10 adds, 100 subtracts, 1000 multiplies can all be counted as a single step

n adds cannot be counted as 1 step
Step Count

for (int i = 1; i < a.length; i++) {
    // insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
Step Count

s/e isn’t always 0 or 1

\[ x = \text{MyMath.sum}(a, n); \]

where \( n \) is the instance characteristic has a s/e count of \( n \)
for (int i = 1; i < a.length; i++) {
    // insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
Step Count

for (int i = 1; i < a.length; i++)
{ 2i + 3}

step count for
    for (int i = 1; i < a.length; i++)
is n

step count for body of for loop is
2(1+2+3+…+n-1) + 3(n-1)
= (n-1)n + 3(n-1)
= (n-1)(n+3)
Asymptotic Complexity of Insertion Sort

- O\( (n^2) \)
- What does this mean?
Complexity of Insertion Sort

- Time or number of operations does not exceed $c.n^2$ on any input of size $n$ ($n$ suitably large).
- Actually, the worst-case time is $\Theta(n^2)$ and the best-case is $\Theta(n)$
- So, the worst-case time is expected to quadruple each time $n$ is doubled
Complexity of Binary Search

```
1. template<class T>
2. int binarySearch(T a[], int n, const T& x)
3. { // Search a[0] <= a[1] <= ... <= a[n-1] for x.
4.   // Return position if found; return -1 otherwise.
5.   int left = 0; // left end of segment
6.   int right = n - 1; // right end of segment
7.   while (left <= right) {
8.     int middle = (left + right)/2; // middle of segment
9.     if (x == a[middle]) return middle;
10.    if (x > a[middle]) left = middle + 1;
11.   } else right = middle - 1;
12. }
13. return -1; // x not found
14.}
```
QuickSort

1 12 5 26 7 14 3 7 2  
unsorted

1 12 5 26 7 14 3 7 2  
pivot value = 7

12 ≥ 7 ≥ 2, swap 12 and 2

1 2 5 26 7 14 3 7 12  
26 ≥ 7 ≥ 7, swap 26 and 7
Quicksort

1 2 5 7 7 14 3 26 12

7 ≥ 7 ≥ 3, swap 7 and 3

i j

1 2 5 7 3 14 7 26 12

i > j, stop partition

j i

1 2 5 7 3 14 7 26 12

run quick sort recursively

...
Quicksort

```c
void quickSort(int arr[], int left, int right) {
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    /* partition */
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    }

    if (left < j)
        quickSort(arr, left, j);
    if (i < right)
        quickSort(arr, i, right);
}
```
Complexity of Insertion Sort

- Is $O(n^2)$ too much time?
- Is the algorithm practical?
# Practical Complexities

$10^9$ instructions/second

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000$</td>
<td>1mic</td>
<td>10mic</td>
<td>1milli</td>
<td>1sec</td>
</tr>
<tr>
<td>$10000$</td>
<td>10mic</td>
<td>130mic</td>
<td>100milli</td>
<td>17min</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1milli</td>
<td>20milli</td>
<td>17min</td>
<td>32years</td>
</tr>
</tbody>
</table>
Impractical Complexities

$10^9$ instructions/second

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^4$</th>
<th>$n^{10}$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17min</td>
<td>$3.2 \times 10^{13}$</td>
<td>$3.2 \times 10^{283}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>116 days</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td>$3 \times 10^7$ years</td>
<td>??????</td>
<td>???????</td>
</tr>
</tbody>
</table>
Faster Computer Vs Better Algorithm

Algorithmic improvement more useful than hardware improvement.

E.g. $2^n$ to $n^3$