Plan of the lecture
- Quick-sort
- Lower bounds on comparison sorting
- Correctness of programs (loop invariants)

Quick-Sort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
- Divide: pick a random element $x$ (called pivot) and partition $S$ into
  - $L$: elements less than $x$
  - $E$: elements equal to $x$
  - $G$: elements greater than $x$
- Recur: sort $L$ and $G$
- Conquer: join $L$, $E$, and $G$

Partition of lists (or using extra workspace)
- We partition an input sequence as follows:
  - We remove, in turn, each element $y$ from $S$ and
  - We insert $y$ into $L$, $E$ or $G$, depending on the result of the comparison with the pivot $x$
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Partitioning arrays
- Perform the partition using two indices to split $S$ into $L$ and $E \cup G$ (a similar method can split $E \cup G$ into $E$ and $G$).
- Repeat until $j$ and $k$ cross:
  - Scan $j$ to the right until finding an element $\geq x$.
  - Scan $k$ to the left until finding an element $< x$.
  - Swap elements at indices $j$ and $k$

Quick-Sort Tree
- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1
Execution Example

- **Pivot selection**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 7 2 9 4 → 2 4 7 9
  - 3 8 6 1 → 1 3 8 6

Execution Example (cont.)

- **Partition, recursive call, pivot selection**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 2 4 3 1 → 2 4 7 9
  - 3 8 6 1 → 1 3 8 6

Execution Example (cont.)

- **Partition, recursive call, base case**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 2 4 3 1 → 2 4 7 9
  - 3 8 6 1 → 1 3 8 6

Execution Example (cont.)

- **Recursive call, ..., base case, join**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 2 4 3 1 → 2 4 7 9
  - 3 8 6 1 → 1 3 8 6

Execution Example (cont.)

- **Recursive call, pivot selection**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 2 4 3 1 → 1 2 3 4
  - 7 9 2 1 → 1 3 8 6

Execution Example (cont.)

- **Partition, ..., recursive call, base case**
  - 7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9
  - 2 4 3 1 → 1 2 3 4
  - 7 9 2 1 → 1 3 8 6
  - 1 → 1
  - 4 → 1
  - 9 → 9
  - 8 → 8
  - 9 → 9
  - 9 → 9
Execution Example (cont.)

Join, join

7 2 9 4 3 7 6 1 → 1 2 3 4 5 7 7 9
2 4 3 1 → 1 2 3 4
7 9 2 → 1 7 2 9
1 → 1
4 1 → 3 4
5 → 5
9 → 9
1 → 1
4 3 → 1 4
3 → 3
9 → 9
4 → 4
9 → 9

Worst-case Running Time

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
One of L and G has size \( n - 1 \) and the other has size 0
The running time is proportional to the sum
\[ a + (a - 1) + \ldots + 2 + 1 \]
Thus, the worst-case running time of quick-sort is \( O(n^2) \)

Best-case Running Time

The best case for quick-sort occurs when the pivot is the median element
The L and G parts are equal – the sequence is split in halfs, like in merge sort
Thus, the best-case running time of quick-sort is \( O(n \log n) \)

Average-case Running Time

The average case for quick-sort: half of the times, the pivot is roughly in the middle
Thus, the average-case running time of quick-sort is \( O(n \log n) \) again
Detailed proof in the textbook

Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>( O(n^2) )</td>
<td>In-place (no extra memory)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>( O(n^2) )</td>
<td>In-place (no extra memory)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>( O(n \log n) ) expected</td>
<td>In-place (+stack), randomized</td>
</tr>
<tr>
<td>heap-sort</td>
<td>( O(n \log n) )</td>
<td>In-place (if heapify the sequence to be sorted)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>( O(n \log n) )</td>
<td>In-place (if heapify the sequence to be sorted)</td>
</tr>
</tbody>
</table>

Lower bound for comparison sorting

Can model sorting which depends on comparisons between elements as a binary decision tree.
At each node, a comparison between two elements is made; there are two possible outcomes and we find out a bit more about the correct order of items in the array.
Finally arrive at full information about the correct order of the items in the array.
Comparison sorting

If a binary tree has $n!$ leaves than the minimal number of levels (assuming the tree is perfect) is $\log n! +1$.

This shows that $O(n \log n)$ sorting algorithms are essentially optimal ($\log_2 n!$ is not equal to $n \log_2 n$ but has the same growth rate modulo some hairy constants).

Correctness of algorithms

Two issues:

- Given an algorithm, prove that it is correct (always achieves the intended result, e.g. a sorted array).
- Design an algorithm with intended properties from specification (even more difficult)

Disproving correctness

- Just one counterexample is enough
- Testing may fail to discover a bug

Additional reading

- Goodrich and Tamassia, Chapter 3.5

Proving correctness

- Formulate precisely the property which has to hold
- If necessary, formulate relevant properties for smaller parts of an algorithm : program assertions
Assertions

- **Assertion**: claim about values of program variables before or after a statement or a group of statements is executed
  - Typical assertions:
    - **Precondition** (usually of a method): what we expect to hold before the method is executed.
    - **Postcondition**: what holds after the method is executed.

Hoare triples

- `{P} S {Q}`: P precondition, S statements, Q postcondition.
- **Meaning**: provided P holds before S is executed, then after S is executed, Q holds.
- For example:
  - `{x < 10} x = x + 20 {x < 30}`
  - `{x < y} while (x < y) x++ {x=y}
- For a small programming language, can provide axioms for every construct in the language and derive postconditions using axioms.

Assignment axiom

- If the only programming construct was assignment, here is an axiom to verify all programs:
  - `{Q(e substituted for x)} x = e {Q}`
- For example, if want to prove `{x < 10} x = x + 20 {x < 30}` then the assignment axiom gives
  - `{x+20 < 30} x = x + 20 {x < 30}` and from extra knowledge about math etc we derive that `{x+20 < 30}` is equivalent to `{x < 10}`.

Example

- In the programs we usually write there are lots of constructs and they also use other people's code.
- Less formal approach (but good practice): write pre- and postconditions for significant chunks of code/methods.

Proving correctness

To prove that an algorithm is correct:
- Determine preconditions and postconditions for the whole algorithm.
- Cascade statement assertions together, so that postconditions for one provide preconditions for the next.
- Prove correctness of individual statements.
- Hence show that executing algorithm with stated preconditions terminates and leads to stated post-conditions.

Loop Invariants

- **Assertions for loops** are difficult, because loops may be executed many times over, with slightly different assertions holding before and after each iteration. Focus on those assertions that remain constant between iterations.
- Known as loop *invariants*: true before and after each iteration through a loop.
Example

```java
pos_greatest = 0;
for (int j = 0; j <= i; j++) {
    if (arr[j] > arr[pos_greatest]) {
        pos_greatest = j;
    }
}
Invariant: pos_greatest is the index of the largest array element between 0 and j.
(More formally, for all k such that 0 ≤ k < j, arr[k] ≤ arr[pos_greatest].)
```

Correctness of loops

To prove correctness of a while loop (or: that assertion A holds after the loop terminates):

- Prove that the loop eventually terminates (by finding the bound function for the loop)
- Find a suitable invariant (there are infinitely many invariants for each loop, most of them useless)
- Prove that A is true after last iteration (usually by substituting the state in which the loop terminates in the invariant)

Partition algorithm

```java
small = l;  // set at the left border of the // range
large = r; // set at the right border where the // pivot sits
while(small < large) {
    if (arr[small] < pivot) small++;
    else {
        large--;
        temp = arr[small];
        arr[small] = arr[large];
        arr[large] = temp;
    }
}
temp = arr[r];
arr[r] = arr[large];
arr[large] = temp;
return large;
```

Postcondition for the partition

```java
public int partition(int[] arr, int l, int r)
// post: returns and integer k such that
//       for all indices i such that l<=i<k,
//       arr[i]<arr[k] and
//       for all indices i such that k<=i<r
//       arr[i]≥arr[k]
```

Bound function for partition

- bound function = large - small
- decreases by 1 at every step; the loop terminates when it is equal to 0 (small = large)
After the loop…

- When large = small, the invariant:
  - if \( 1 \leq i < small \), then \( arr[i] < pivot \)
  - if \( large \leq i < r \) then \( arr[i] \geq pivot \)

becomes (substitute large for small):

- if \( 1 \leq i < large \), then \( arr[i] < pivot \)
- if \( large \leq i < r \) then \( arr[i] \geq pivot \)
- After the pivot is swapped, \( pivot = arr[large] \).

Partition algorithm

```c
Assertion 1: l < r; pivot is arr[r]
while(small < large) {
    if (arr[small] < pivot) small++;
    else {
        large--;
        temp = arr[small];
        arr[small] = arr[large];
        arr[large] = temp;
    }
}

Assertion 2: pivot is arr[r];
if l \leq i < large, then arr[i] < pivot;
if large \leq i < r then arr[i] \geq pivot

temp = arr[r];
arr[r] = arr[large];
arr[large] = temp;

Assertion 3: arr[large] is the pivot
return large;
```

Informal coursework (tutorials from the 6th of December)

Given that \( l < r \) in the partition method, which of the following are loop invariants of the while loop:

1) small < r
2) small < large
3) small \leq large
4) for all \( i \) such that \( 1 \leq i < small \), \( arr[i] < pivot \)
5) for all \( j \) such that \( large \leq j \leq r \), \( arr[j] \geq pivot \)
6) for all \( j \) such that \( large < j \leq r \), \( arr[j] \geq pivot \)

Informal coursework 2

Prove correctness of selection sort:
```c
void selectionSort(int arr[], int len){
    int i,j,temp,pos_greatest;
    for( i = len - 1; i > 0; i--){
        pos_greatest = 0;
        for(j = 0; j <= i; j++){
            if(arr[j] > arr[pos_greatest]) pos_greatest = j;
        }
        temp = arr[i];
        arr[i] = arr[pos_greatest];
        arr[pos_greatest] = temp;
    }
}
```