SORTING

Insertion sort
Selection sort
Quicksort
Mergesort
And their asymptotic time complexity

See lecture notes page, row in table for this lecture, for file searchSort/Algorithms.zip

Insertion sort
Selection sort
Quicksort
Mergesort

A3 and Prelim

- 379/607 (62%) people got 65/65 for correctness on A3
- 558/607 (92%) got at least 60/65 for correctness on A3

- Prelim: Next Tuesday evening, March 14
  Read the Exams page on course website to determine when you take the prelim (5:30 or 7:30) and what to do if you have a conflict.
- If necessary, complete CMS assignment P1Conflict by the end of Wednesday (tomorrow).
- So far, only 15 people filled it out!

InsertionSort

A loop that processes elements of an array in increasing order has this invariant

**inv:**

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>post: b</th>
<th>0</th>
<th>b.length</th>
</tr>
</thead>
</table>

**inv:**

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>post: b</th>
<th>0</th>
<th>b.length</th>
</tr>
</thead>
</table>

Pre: b
Post: b sorted

for (int i = 0; i < b.length; i = i + 1) { maintain invariant }

Each iteration, i = i + 1: How to keep inv true?

- **inv:**
  - b
  - i
  - b.length

- **e.g.:**
  - b
  - i
  - b.length

Push b[i] down to its shortest position in b[0..i]

Will take time proportional to the number of swaps needed

What to do in each iteration?

- **inv:**
  - b
  - i
  - b.length

- **e.g.:**
  - b
  - i
  - b.length

Push b[i] to its sorted position in b[0..i], then increase i

InsertionSort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i + 1) {
    Push b[i] to its sorted position in b[0..i], then increase i
}
```

Note English statement in body. **Abstraction.** Says what to do, not how.

This is the best way to present it. We expect you to present it this way when asked.

Later, can show how to implement that with an inner loop

Many people sort cards this way Works well when input is nearly sorted
Push b[i] down ...

Q: b[0..i-1] is sorted
// Push b[i] down to its sorted position in b[0..i]
int k = i;
while (k > 0 && b[k] < b[k-1]) {
    Swap b[k] and b[k-1]
    k = k-1;
}
// R: b[0..i] is sorted

invariant P: b[0..i] is sorted except that b[k] may be < b[k-1]

2 5 3 5 5 ? ?

example

How to write nested loops

// sort b[], an array of int
// inv: b[0..i-1] is sorted
int i = 0;
for (i = 0; i < b.length; i++) {
    if (k = i;)
        while (k > 0 && b[k] < b[k-1]) {
            Swap b[k] and b[k-1]
            k = k-1;
        }
    //Push b[i] down to its sorted position in b[0..i]
}
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i++) {
    if (k = i;)
        while (k > 0 && b[k] < b[k-1]) {
            Swap b[k] and b[k-1]
            k = k-1;
        }
    //Push b[i] down to its sorted position in b[0..i]
}

InsertionSort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i++) {
    Push b[i] down to its sorted position in b[0..i]
}

Pushing b[i] down can take i swaps. Worst case takes
\[ 1 + 2 + 3 + \ldots + \frac{\text{sum of all integers}}{2} \approx \frac{n^2}{2} \text{ swaps.} \]

SelectionSort

// sort b[], an array of int
// inv: b[0..i-1] is sorted AND b[0..i-1] <= b[i..]
for (int i = 0; i < b.length; i++) {
    int m = index of minimum of b[i..];
    Swap b[i] and b[m];
}

Another common way for people to sort cards

Runtime

with n = b.length

- Worst-case O(n^2)
- Best-case O(n)
- Expected-case O(n^2)

SelectionSort

// Swap b[i] and b[m]
int t = b[i];
b[i] = b[m];
b[m] = t;

Swap b[i] and b[m]
**Partition algorithm of quicksort**

pre: $x$                          ?

\[
\begin{array}{c}
\text{h} \\
\text{h+1} \\
\text{k}
\end{array}
\]

x is called the pivot

Swap array values around until $b[h..k]$ looks like this:

\[
\begin{array}{c}
\text{h} \\
\text{j} \\
\text{t} \\
\text{k}
\end{array}
\]

post: $\leq x$ $x$ $\geq x$

**Partition algorithm**

pre: $b$                          ?

\[
\begin{array}{c}
\text{h} \\
\text{h+1} \\
\text{k}
\end{array}
\]

post: $b$ $\leq x$ $x$ $\geq x$

Combine pre and post to get an invariant

\[
\begin{array}{c}
\text{h} \\
\text{j} \\
\text{t} \\
\text{k}
\end{array}
\]

\[
\begin{array}{c}
\leq x \\
\geq x
\end{array}
\]

invariant needs at least 4 sections

**QuickSort procedure**

```java
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return; // Base case
    int j = partition(b, h, k);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}

int partition(b, h, k) {
    \[
    \begin{array}{c}
    \text{h} \\
    \text{j} \\
    \text{k}
    \end{array}
    \]

    j = h; t = k;
    while (j < t) {
        if (b[j+1] <= b[j]) {
            Swap b[j+1] and b[j]; j = j+1;
        } else {
            Swap b[j+1] and b[t]; t = t-1;
        }
    }
    \}

    Takes linear time: $O(k+1-h)$
```

**QuickSort**

QuickSort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

83 years old.

Developed QuickSort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. “Ah!” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.
Worst case quicksort: pivot always smallest value

```
x0  >= x0
j
```
partitioning at depth 0

```
x0  x1  >= x1
j
```
partitioning at depth 1

```
x0  x1  x2  >= x2
j
```
partitioning at depth 2

```/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);  QS(b, j+1, k);
}
```

Depth of recursion: O(n)
Processing at depth i: O(n-i)
Total time: O(n log n).

Best case quicksort: pivot always middle value

```
<= x0  x0  >= x0
0  j  n  depth 0. 1 segment of size ~n to partition.
```

```
<=x1 x1 x0 <=x2 x2 >=x2
```
Depth 2. 2 segments of size ~n/2 to partition.
Depth 3. 4 segments of size ~n/4 to partition.

Max depth: O(log n).  Time to partition on each level: O(n)
Total time: O(n log n).

Average time for Quicksort: n log n. Difficult calculation

QuickSort complexity to sort array of length n

```/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    QS(b, h, j-1);  QS(b, j+1, k);
}
```

Worst-case space: O(n log n)

Partition. Key issue. How to choose pivot

Choosing pivot

Ideal pivot: the median, since it splits array in half
But computing is O(n), quite complicated

Popular heuristics: Use
• first array value (not so good)
• middle array value (not so good)
• Choose a random element (not so good)
• median of first, middle, last, values (often used)!
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted. */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}

 Runs in time linear in size of b[h..k].
Look at this method in file searchSortAlgorithms.zip
found in row for lecture on Lecture notes page of course website

/** Sort b[h..k] */
public static void mergesort(int[] b, int h, int k) {
    if (size b[h..k] < 2)
        return;
    int t= (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}

Mergesort

QuickSort versus MergeSort

/** Sort b[h..k] */
public static void QS(int[] b, int h, int k) {
    if (k – h < 1)
        return;
    int j= partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}

One processes the array then recurses.
One recurses then processes the array.
Analysis of Matrix Multiplication

Multiply \( n \)-by-\( n \) matrices \( A \) and \( B \):

Convention, matrix problems measured in terms of \( n \), the number of rows, columns
- Input size is really \( 2n^2 \), not \( n \)
- Worst-case time: \( O(n^3) \)
- Expected-case time: \( O(n^3) \)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] += a[i][k]*b[k][j];
    }
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        throw new Exception();
    }
```

An aside. Will not be tested.
Lower Bound for Comparison Sorting

- Goal: Determine minimum time required to sort \( n \) items
  - Note: we want worst-case, not best-case time
  - Best-case doesn’t tell us much. E.g. Insertion Sort takes \( O(n) \) time on already-sorted input
  - Want to know worst-case time for best possible algorithm

How can we prove anything about the best possible algorithm?
Want to find characteristics that are common to all sorting algorithms
Limit attention to comparison-based algorithms and try to count number of comparisons

An aside. Will not be tested.
Lower Bound for Comparison Sorting

- Comparison-based algorithms make decisions based on comparison of data elements
- Gives a comparison tree
- If algorithm fails to terminate for some input, comparison tree is infinite
- Height of comparison tree represents worst-case number of comparisons for that algorithm
- Can show: Any correct comparison-based algorithm must make at least \( n \log n \) comparisons in the worst case

How many input permutations are possible? \( n! \sim 2^{n \log n} \)

For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree
To have at least \( n! \sim 2^{n \log n} \) leaves, it must have height at least \( n \log n \) (since it is only binary branching, the number of nodes at most doubles at every depth)
Therefore its longest path must be of length at least \( n \log n \), and that is its worst-case running time

Quicksort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.
Eliminate this problem by doing some of it iteratively and some recursively
Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!

It's on the next two slides. You do not have to study this for the prelim!

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1= h, k1= k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j= partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1]) {
            QS(b, h, j-1); h1= j+1;
        } else {
            QS(b, j+1, k1); k1= j-1;
        }
    }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n