Data Structures and Algorithms for Information Processing

Lecture 12: Sorting
Outline

• Correctness proof digression
• Consider various sorts, analyze
• Insertion, Selection, Merge, Radix
• Upper & Lower Bounds
• Indexing
What Does This Method Compute?

int doubleTheNumber(int m) {
    int n = m;
    while(n > 1) {
        if (n % 2 == 0) n = n / 2;
        else n = 3 * n + 1;
    }
    return 2 * m;
}

A proof of termination is required.

Please call my cell if you can show this either way.
The Jar Game

A jar contains $n \geq 1$ marbles. Each is of color red or of blue. Also we have an unlimited supply of red marbles.

Will the following algorithm terminate?

From http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html
The Jar Game

while (# of marbles in the jar > 1) {
    choose (any) two marbles from the jar;
    if (the two marbles are of the same color)
    { toss them aside;
        place a RED marble into the jar;
    }
    else {
        toss the chosen RED marble aside;
        place the chosen BLUE marble back into the jar;
    }
}

http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html
Find A Loop Invariant

• Can we find a loop invariant that will help us to prove the following theorem:

The last remaining ball will be blue if the initial number of blue balls was odd and red otherwise.

From http://www.cs.uofs.edu/~mccloske/courses/cmgs144/invariants_lec.html
Sorting Demonstration

Intuitive Introduction

Main’s slides from Chapter 12
Insertion Sort

Consider each item once, insert into growing sorted section.

```java
void insertionSort(int A[]) {
    for(int i=1; i<A.length; i++)
        for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);
}
```
Insertion Sort

void insertionSort(int A[]) {
    for(int i=1; i<A.length; i++)
        for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);
}

• runs in O(n^2), where n = A.length.
• If A is sorted already, runs in O(n).
• Use if you’re in a hurry to code it, and speed is not an issue.
Proving Insertion Sort Correct

What is the invariant?

\[
\text{void insertionSort(int A[])} \{ \\
\quad \text{for(int i=1; i < A.length; i++)} \\
\quad \quad \text{for(int j=i; j>0 && A[j]<A[j-1]; j--)} \\
\quad \quad \quad \text{swap}(A[j],A[j-1]); \\
\}\n\]

\[
(\forall 0 \leq t < u < i)[A[t] \leq A[u]]
\]

i=1, it’s trivially true. When when i=n, array is sorted.
Now consider inner loop

```java
void insertionSort(int A[]) {
    for(int i=1; i < A.length; i++)
        for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);
}
```

Trivially true when \( j = i \), and implies outer loop invariant when it exits.
What happens inside inner loop?

```java
void insertionSort(int A[]) {
    for(int i=1; i < A.length; i++)
        for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);
}
```

exit inner loop
What is the Average Time for Insertion Sort?

( Best is $O(n)$, Worst is $O(n^2)$)

- Running time is proportional to number of swaps.
- Each swap of adjacent items decreases disorder by one unit where
  \[
  \text{disorder} = \text{number of } i < j \text{ such that } A[i] > A[j]
  \]
- Therefore running time is proportional to disorder and average running time is proportional to average disorder.
Average disorder

<table>
<thead>
<tr>
<th>Sequence</th>
<th>disorder</th>
<th>Reversed Sequence</th>
<th>disorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>0</td>
<td>4321</td>
<td>6</td>
</tr>
<tr>
<td>1243</td>
<td>1</td>
<td>3421</td>
<td>5</td>
</tr>
<tr>
<td>1324</td>
<td>1</td>
<td>4231</td>
<td>5</td>
</tr>
<tr>
<td>1342</td>
<td>2</td>
<td>2431</td>
<td>4</td>
</tr>
<tr>
<td>1423</td>
<td>2</td>
<td>3241</td>
<td>4</td>
</tr>
<tr>
<td>1432</td>
<td>3</td>
<td>2341</td>
<td>3</td>
</tr>
<tr>
<td>2134</td>
<td>1</td>
<td>4312</td>
<td>5</td>
</tr>
<tr>
<td>2143</td>
<td>2</td>
<td>3412</td>
<td>4</td>
</tr>
<tr>
<td>2314</td>
<td>2</td>
<td>4132</td>
<td>4</td>
</tr>
<tr>
<td>2413</td>
<td>3</td>
<td>3142</td>
<td>3</td>
</tr>
<tr>
<td>3124</td>
<td>2</td>
<td>4213</td>
<td>4</td>
</tr>
<tr>
<td>3214</td>
<td>3</td>
<td>4123</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>

for n=4 Average disorder = 72/24 = 3
What is the Average Disorder?

**Theorem:** The average disorder for a sequence of n items is $n(n-1)/4$

**Proof:** Assume all permutations of array $A$ equally likely. If $A^R$ is the reverse of $A$, then $\text{disorder}(A) + \text{disorder}(A^R) = n(n-1)/2$ because $A[i] < A[j]$ iff $A^R[i] > A^R[j]$. Thus the average disorder over all permutations is $n(n-1)/4$.

**Corollary:** The average running time of any sorting program that swaps only adjacent elements is $\Omega(n^2)$.

**Proof:** It will have to do $n(n-1)/4$ swaps and may waste time in other ways.
To better $O(n^2)$ we must compare non-adjacent elements

Shell Sort: Swap elements $n/2$, $n/4$, ... apart
Heap Sort: Swap $A[i]$ with $A[i/2]$
QuickSort: Swap around “median”
Idea of Merge Sort

• Divide elements to be sorted into two groups of equal size
• Sort each half
• Merge the results using a simultaneous pass through each
Psuedocode for Merge Sort

```c
void mergesort(int data[], int first, int n) {
    if (n > 1) {
        int n1 = n/2;
        int n2 = n - n1;
        mergesort(data, first, n1);
        mergesort(data, first+n1, n2);
        merge(data, first, n1, n2);
    }
}
```
How fast could a sort that uses binary comparisons run?

Consider 4 numbers, a, b, c, d. Merge Sort approach:

```
  a<b?
    y   n
     y   n
      y   n
       y   n
      c<a?
    d<a?
    c<b?
  c<d?
    y   n
     y   n
      y   n
       y   n
      d<b?
  b≤a&d≤c
```
Ask only questions you don’t know answers to.

```
<table>
<thead>
<tr>
<th></th>
<th>a&lt;b?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td>c&lt;d?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
<tr>
<td>c&lt;d?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>c&lt;a?</td>
<td></td>
</tr>
<tr>
<td>d&lt;a?</td>
<td></td>
</tr>
<tr>
<td>c&lt;b?</td>
<td></td>
</tr>
<tr>
<td>d&lt;b?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>b≤a&amp;d≤c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>c&lt;a&lt;b&amp;c&lt;d</td>
<td></td>
</tr>
<tr>
<td>a&lt;b&amp;a≤c&lt;d</td>
<td></td>
</tr>
<tr>
<td>d&lt;b≤a&amp;d≤c</td>
<td></td>
</tr>
<tr>
<td>b≤a&amp;b≤d≤c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>b&lt;c?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a&lt;b&amp;asc&lt;d&amp;csb</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>asc&lt;b&lt;d</td>
<td></td>
</tr>
<tr>
<td>asc&lt;dsb</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4 compares</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>5 compares</td>
<td></td>
</tr>
</tbody>
</table>
```
A different strategy, insertion sorts, may get lucky.

\[
\begin{align*}
\text{a} & < \text{b} \quad \text{y} \quad \text{n} \\
\text{b} & < \text{c} \quad \text{y} \quad \text{n} \\
\text{c} & < \text{d} \\
\text{a} & < \text{b} \quad \text{c} < \text{d}
\end{align*}
\]

3 compares
But it may be unlucky.
Consider all possible sorting trees. How many leaves must a sorting tree have to distinguish all possible orderings of n items?

\[ a[0] < a[1] \]
How many leaves must there be for a sorting tree for $n$ items?

$n!$, the number of different permutations.
**Theorem:** A binary tree with K leaves must have depth at least \( \lceil \log_2 K \rceil \). In other words, a BT with k leaves and depth d has d \( \geq \) \( \lceil \log_2 K \rceil \) or \( K \leq 2^d \).

**Proof:** Prove by induction that a tree of depth d can have at most, \( 2^d \) leaves.

Base: for \( d=0 \), there is 1 leaf.
Suppose true for \( d \), consider tree of depth \( d+1 \).

```plaintext
\[
\begin{array}{c}
  x \\
  \downarrow \\
  y
  \end{array}
\]
```

BIH: \( x \) and \( y \) have at most \( 2^d \) leaves so whole tree has at most \( 2 \times 2^d = 2^{d+1} \) leaves.

Now the shortest trees with K leaves must be “perfect” and their depth will be \( \lceil \log_2 K \rceil \).
So a tree with $n!$ leaves has depth at least $\log n!$. Notice that depth = the maximum number of tests one might have to perform.

$$\log n! = \log n(n-1)(n-2)\ldots 1$$

$$= \log n + \log n-1 + \log n-2 + \ldots + \log 1$$

$$\geq \log n + \ldots + \log(n/2)$$

$$\geq (n/2) \log(n/2)$$

$$\geq (n/2) \log n - n/2$$

$$= \Omega(n \log n)$$

So any sort algorithm takes $\Omega(n \log n)$ comparisons.
Is there a way to sort without using binary comparisons?

Ternary comparisons, K-way comparisons.

The basic $\Omega(n \log n)$ result will still be true, because $\Omega(\log_2 x) = \Omega(\log_k x)$.

Useful speed-up heuristic: use your data as an index of an array.
Consider sorting tray of letters

```c
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts[i]=0;
for(j=0; j<tray.length; j++)
    count[tray[j]-'a']++;
j=0;
for(int i=0; i<26; i++)
    while(count[i]-- > 0) tray[j++]=i+'a';
```
Sorting tray of letters

```c
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts[i]=0;
for(j=0; j<tray.length; j++)
    count[tray[j] - 'a']++;

j=0;
for(int i=0; i<26; i++)
    while(count[i]-- > 0) tray[j++]=i+'a';
```

**Running time is O(26+tray.size()), i.e. linear!**

if tray = “abbcabbdaf”

count = {3,4,1,1,0,1,0, ..., 0}

and new tray = “aaabbbbcdf”
Why does this beat $n \log n$?

- The operation `count[tray[j]]++` is like a 26-way test; the outcome depends directly on the data.
- This is “cheating” because it won’t work if the data range grows from 26 to $2^{32}$.
- Technique can still be useful — can break up range into “buckets” and use mergesort on each bucket.
Radix Sort

A way to exploit the data-driven idea for large data spaces.

Idea: Sort the numbers by their *lowest* digit. Then sort them by the next lowest digit, being careful to break ties properly. Continue to highest digit.
Radix Sort

• Each sort must be **stable**
  The relative order of equal keys is preserved

• In this way, the work done for earlier bits is not “undone”
Radix Sort

Informal Algorithm:
To sort items A[i] with value 0...2^{32}-1 (= INT_MAX)

- Create a table of 256 buckets.
- Take all the items from the buckets 0,..., 255 in a FIFO manner, re-packing them into A.}
- Repeat using A[i]/256 mod 256
- Repeat using A[i]/256^2 mod 256
- Repeat using A[i]/256^3 mod 256
- This takes O(4*(256+A.length))
Radix Sort using Counts

The Queues can be avoided by using counts:

Let $N = \text{number of elements in array } a$

Array $a$ is indexed from 1 to $N$

Let $w = \text{the number of bits in } a[i]$

Let $m = \text{number of bits examined per pass}$

Let $M = 2^m \text{ patterns to count}$
Radix Sort using Counts

The Queues can be avoided by using counts:

```c
void RadixSort(int a[], int b[], int N) {
    int i, j, pass, count[M];
    for (pass=0; pass < (w/m); pass++) {
        for (j=0; j < M; j++) count[j] = 0;
        for (i=1; i <= N; i++)
            count[a[i].bits(pass*m, m)]++;
        for (j=1; j < M; j++)
            count[j] = count[j-1] + count[j];
        for (i=N; i >= 1; i--)
            b[count[a[i].bits(pass*m, m)]--] = a[i];
    }
    for (i=1; i <= N; i++) a[i] = b[i];
}
```
Radix Sort using Queues

const int BucketCount = 256;
void RadixSort(vector<int> &A) {
    vector<queue<int> > Table(BucketCount);
    int passes = ceil(log(INT_MAX)/log(BucketCount));
    int power = 1;
    for(int p=0; p<passes;p++) {
        int i;
        for(i=0; i<A.size(); i++) {
            int item = A[i];
            int bucket = (item/power) % BucketCount;
            Table[bucket].push(item);
        }
        i =0;
        for(int b=0; b<BucketCount; b++)
            while(!Table[b].empty()) {
                A[i++] = Table[b].front(); Table[b].pop();
            }
        power *= BucketCount;
    }
}
Radix Sort

In general it takes time

\[ O( \text{Passes}^*(\text{NBuckets}+\text{A.length})) \]

where \( \text{Passes} = \left\lceil \frac{\log(\text{INT\_MAX})}{\log(\text{NBuckets})} \right\rceil \)

It needs \( O(\text{A.length}) \) in extra space.
Next Time

• The next topic will be **Quicksort**, a very fast, practical, and widely used algorithm