CS171 Introduction to Computer Science II

Recursion (cont.) + MergeSort

Li Xiong
Reminders

• Hw3 due yesterday (use late credit if needed)
• Hw4 due Friday
Today

• Recursion (cont.)
  – Concept and examples
  – Analyzing cost of recursive algorithms
  – Divide and conquer
  – Dynamic programming

• MergeSort
Fibonacci Numbers

• Recursive formula:

\[ F(n) = F(n-1) + F(n-2) \]

\[ F(0) = 0, \quad F(1) = 1 \]

• 0, 1, 1, 2, 3, 5, 8, 13, .....
Fibonacci Numbers

```c
int F(int n) {
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return F(n-1)+F(n-2);
}
```

Consider $F(5)$, how is it computed?
Runtime of Recursive Fibonacci

\[
F(5) = F(4) + F(3) = F(3) + F(2) + F(2) + F(1)
\]

3/8/2012
Dynamic programming

• **Dynamically** solve a smaller problem
  – Solve each small problem only once

• Applicable when
  – Overlapping subproblems are slightly smaller (vs. divide and conquer)
  – *Optimal substructure*: the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems.
Memoization

• A technique for dynamic programming
  – A memoized function "remembers" the results corresponding to some set of specific inputs.
  – Subsequent calls with remembered inputs return the remembered result, rather than recomputing it

• General structure
  static int sol[]; //save solutions for each problem
  static ... recursiveFunc(int N) {
    if (sol[N] is available) 
      return sol[N];
    ...
    similar to regular recursion
    except: saving the solution sol[N]
  }
Fibonacci with Dynamic Programming

```java
static int sol[];
static int F(int n) {
    if (sol[n] > 0) //pre-computed already
        return sol[n];
    if (n==0) {
        sol[n] = 1;
        return 1;
    } else if (n==1) {
        sol[n] = 1;
        return 1;
    } else {
        sol[n] = F(n-1) + F(n-2);
        return sol[n];
    }
}

Example: F(5)
```
Today

• Recursion (cont.)
  – Concept and examples
  – Analyzing cost of recursive algorithms
  – Divide and conquer
  – Dynamic programming

• MergeSort
Advanced Sorting

• We’ve learned some simple sorting methods, which all have quadratic costs.
  – Easy to implement but slow.

• Much faster advanced sorting methods:
  – Merge Sort
  – Quick Sort
  – Radix Sort
MergeSort

• Basic idea
  – Divide array in half
  – Sort each half (how?)
  – Merge the two sorted halves
Merge Sort

• This is a divide and conquer approach:
  – Partition the original problem into two sub-problems;
  – Use recursion to solve each sub-problem;
  – Sub-problem eventually reduces to base case;
  – The results are then combined to solve the original problem.
Merge Two Sorted Arrays

- A **key step** in mergesort
- Assume arrays A (left half) and B (right half) are **already sorted**.
- Merge them to array C (the original array), such as C contains all elements from A and B, and remains sorted
- Use an auxiliary array aux[]

Example on board and demo

<table>
<thead>
<tr>
<th></th>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>aux[]</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Merging Two Sorted Arrays

1. Start from the first elements of A and B;
2. Compare and copy the smaller element to C;
3. Increment indices, and continue;
4. If reaching the end of either A or B, quit loop;
5. If either A (or B) contains remaining elements, append them to C.
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    assert isSorted(a, lo, mid); // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi); // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)
        aux[k] = a[k]; // copy

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
        { // merge
            if (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }

    assert isSorted(a, lo, hi); // postcondition: a[lo..hi] sorted
}
Merging Two Sorted Arrays: Analysis

• How many comparisons is required?

• How many copies?
Merging Two Sorted Arrays (Sol.)

• How many comparisons is required?
  at most \((A\cdot\text{length} + B\cdot\text{length})\)
• How many copies?
  \(A\cdot\text{length} + B\cdot\text{length}\)
 Assertions

 **Assertion.** Statement to test assumptions about your program.

 - Helps detect logic bugs.
 - Documents code.

 **Java assert statement.** Throws an exception unless boolean condition is true.

 ```java
 assert isSorted(a, lo, hi);
 ```

 **Can enable or disable at runtime.** ⇒ No cost in production code.

 ```java
 java -ea MyProgram // enable assertions
 java -da MyProgram  // disable assertions (default)
 ```

 **Best practices.** Use to check internal invariants. Assume assertions will be disabled in production code (e.g., don't use for external argument-checking).
Divide
Divide
Divide
Conquer
Conquer
Conquer
public class Merge {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        /* as before */
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
First base case encountered
Return, and continue
Mergesort: animation

50 random items

http://www.sorting-algorithms.com/merge-sort
Mergesort: empirical analysis

Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion Sort ($N^2$)</th>
<th>Mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Merge Sort Analysis

Cost Analysis

• What’s the cost of mergesort?
• Recurrence relation: $T(N) = 2 \times T(N/2) + N$

$O(N \times \log N)$
This is called log-linear cost.
Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N\lg N$.

Pf 2. [assuming $N$ is a power of 2]

\[
D(N) = 2D(N/2) + N
\]

\[
\frac{D(N)}{N} = 2\frac{D(N/2)}{N} + 1
\]

\[
= \frac{D(N/2)}{(N/2)} + 1
\]

\[
= \frac{D(N/4)}{(N/4)} + 1 + 1
\]

\[
= \frac{D(N/8)}{(N/8)} + 1 + 1 + 1
\]

\[
\vdots
\]

\[
= \frac{D(N/N)}{(N/N)} + 1 + 1 + \ldots + 1
\]

\[
= \lg N
\]
# Merge Sort

Is this a lot better than simple sorting?

<table>
<thead>
<tr>
<th># of elements</th>
<th>( N^2 )</th>
<th>( N \log N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>200</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>3,000</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>40,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
    int mid = lo + (hi - lo) / 2;
    sort (a, aux, lo, mid);
    sort (a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```
Mergesort: practical improvements

Stop if already sorted.

- Is biggest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```
Mergesort: visualization
Divide and Conquer
Bottom-up MergeSort

1. Every element itself is trivially sorted;
2. Start by merging every two adjacent elements;
3. Then merge every four;
4. Then merge every eight;
5. ... 
6. Done.
Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ....

Bottom line. No recursion needed!
public class MergeBU
{
    private static Comparable[] aux;
    
    private static void merge(Comparable[] a, int lo, int mid, int hi)
    { /* as before */ }
    
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
Summary

- Merging two sorted array is a key step in merge sort.
- Merge sort uses a divide and conquer approach.
- It repeatedly splits an input array to two sub-arrays, sort each sub-array, and merge the two.
- It requires $O(N \times \log N)$ time.

- On the downside, it requires additional memory space (the workspace array).