CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 11

October 10, 2013

- Common techniques to solve various problems.
- Divide and conquer
- Backtracking
- Greedy algorithms
- Dynamic programming
- We will use several examples, some problems you have seen before (sorting), to demonstrate the use of such techniques.
- We will work with a software package that programs simple board games.

Sorting and Binary Search

- One of the most fundamental problems in CS.
- Problem definition: Given a series of elements with a well-defined order, return a series of the elements sorted according to this order.
- Simple (insertion) Sort – runs in quadratic time
- BubbleSort – runs in quadratic time
- Shellsort – runs in sub-quadratic time
- Mergesort – runs in O(NlogN) time
- Quicksort – runs in average O(NlogN) time

Mergesort

- 3 steps
  1. Return if the number of items to sort is 0 or 1
  2. Recursively Mergesort the first and second halves separately
  3. Merge the two sorted halves into a sorted group
- This approach is called “divide and conquer”.
- Divide the problem into sub-problems, “conquer” (solve) them separately and merge the results.
- Mergesort is an O(N*logN) algorithm

The Mergesort Algorithm

```
public static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [ ] a )
{
    AnyType [ ] tmpArray = (AnyType []) new Comparable[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}

/**
 * Internal method that makes recursive calls.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */
private static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
                int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```

MergeSort Performance

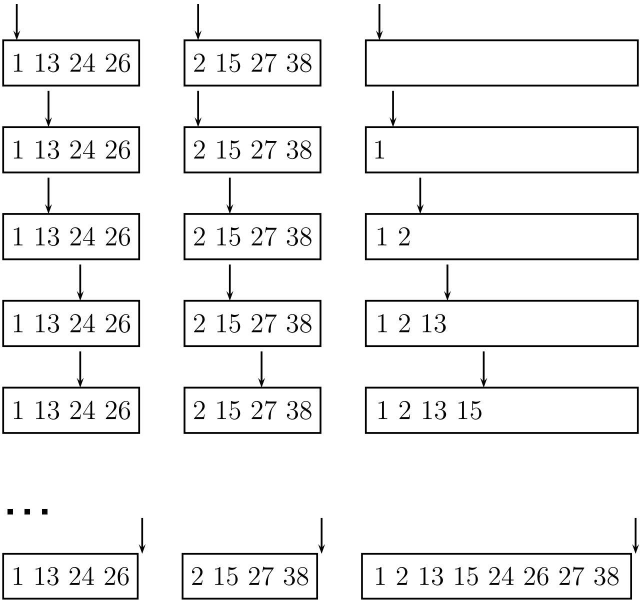
$$
\begin{aligned}
T(N) = {} & 2 * T(N/2) + O(N) \\
= {} & 2 * (2 * T(N/4) + O(N/2)) + O(N) \\
= {} & 4 * T(N/4) + O(N) + O(N) \\
= {} & 4 * (2 * T(N/8) + O(N/4)) + O(N) + O(N) \\
= {} & 8 * T(N/8) + O(N) + O(N) + O(N) \\
= {} & ..... = 2 \log N * T(1) + O(N) + O(N) + ... + O(N) \\
= {} & N * O(1) + O(N) + O(N) + ... + O(N).
\end{aligned}
$$

The terms are expanded logN times, each produces an O(N). log N terms of $O(N) = O(N \log N)$

Internal Merge Method

```
private static <AnyType extends Comparable<? super AnyType>>
void merge( AnyType [ ] a, AnyType [ ] tmpArray,
            int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

Linear-time Merging of Sorted Arrays



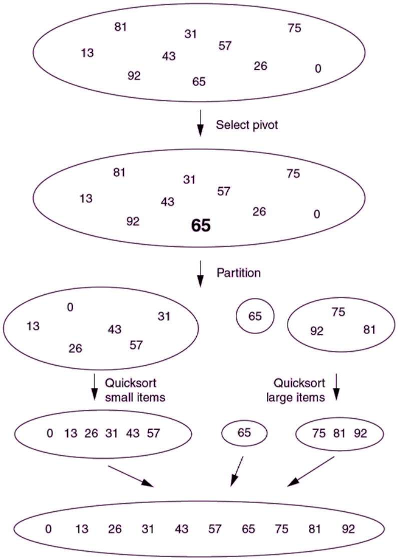
Quicksort Algorithm

4 steps:

1. Return if the number of elements in S is 0 or 1
2. Pick a “pivot” – element v in S
3. Partition $S - \{v\}$ into 2 disjoint sets:
   $L = \{x \in S - \{v\} | x < v\}$, $R = \{x \in S - \{v\} | x > v\}$
4. Return the result of Quicksort(L) followed by v followed by Quicksort(R)

Notice that after each partition the pivot is in its final sorted position.

Quicksort Algorithm



Quicksort Analysis

$T(N) = O(N) + T(|L|) + T(|R|)$

- The first term refers to the partition, which is linear in N.
- The second and third are recursive calls to subarrays of size L and R, respectively.
- Similar to mergesort analysis, so should be $O(N \log N)$... or is it?
- The result depends on the size of L and R. If roughly the same – yes. Otherwise – if one partition is O(1) and the other O(N), may be quadratic!

Picking the Pivot

- A wrong way
  - Pick the first element or the larger of the first two elements
  - If the input has been presorted or is reverse order, this is a poor choice
- A safe choice
  - Pick the middle element
- Median-of-three
  - Pivot equal to the median of the first, middle and last elements
  - Nothing guarantees asymptotic O(N*logN), but it can be shown that mostly this is the case.

Binary Search

- Definition: Search for an element in a sorted array.
- Return array index where element is found or a negative value if not found.
- Implemented in Java as part of the Collections API.
- Idea from the book start in the middle of the array.
- If the element is smaller than that, search in the smaller half. Otherwise – search in the larger half.

Binary Search Implementation

```
static <T> int binarySearch(T[] a, T key,
Comparator<?  super T> c)
static int binarySearch(Object[] a, Object key)
```

- The version without the Comparator uses "natural order" of the array elements, i.e., calls compareTo of the element type to compare elements.
- Thus the elements need to be Comparable – the element type implements Comparable<ElementType> in the generics setup.
- Or the old Comparable works here too.

Binary Search Implementation

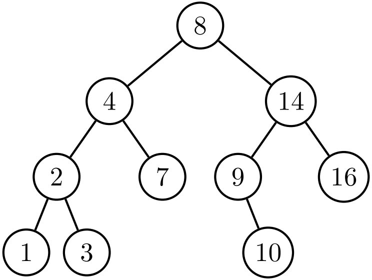
```java
/**
 * Performs the standard binary search
 * using two comparisons per level.
 * @return index where item is found, or NOT_FOUND.
 */
public static <AnyType extends Comparable<? super AnyType>>
              int binarySearch( AnyType [ ] a, AnyType x )
{
    int low = 0;
    int high = a.length - 1;
    int mid;

    while( low <= high )
    {
        mid = ( low + high ) / 2;

        if( a[ mid ].compareTo( x ) < 0 )
            low = mid + 1;
        else if( a[ mid ].compareTo( x ) > 0 )
            high = mid - 1;
        else
            return mid;
    }

    return NOT_FOUND;     // NOT_FOUND = -1
}
```

figure 5.11

Basic binary search that uses three-way comparisons

Binary Search Tree

$$T(N) = T(N/2) + O(1)$$
$$T(N) = O(logN)$$



- What is that $<?superT>$ clause?
- The $Comparable <?superT>$ specifies that T ISA $Comparable < Y >$, where Y is T or any superclass of it.
- This allows the use of a compareTo implemented at the top of an inheritance hierarchy (i.e., in the base class) to compare elements of an array of subclass elements.
- For example, we commonly use a unique id for equals, hashCode and compareTo across a hierarchy, and only want to implement it once in the base class.

Sorting Implementation

```
static void sort(Object[] a)

static <T> void sort(T[] a, Comparator<? super T> c)
```

Default – natural order of elements from small to large. Possible to define another Comparator.

Sorting – Comments

- It can be shown that in the general case (comparison based sorting) we can't do better than O(N*logN) in the worst case.
- When assumptions can be made on the input – linear sorting is possible.
- Example – N integers all between 1 and O(N).