Divide and Conquer Algorithms: Advanced Sorting

Prichard Ch. 10.2: Advanced Sorting Algorithms

(revisit) Properties of Growth-rate functions (1/3)
1. You can ignore low-order terms in an algorithm's growth-rate function.
   - $O(n^3+4n^2+3n)$ it is also $O(n^3)$

(revisit) Properties of Growth-rate functions (2/3)
2. You can ignore a multiplicative constant in the high-order term of an algorithm's growth-rate function.
   - $O(5n^3)$, it is also $O(n^3)$

(revisit) Properties of Growth-rate functions (3/3)
3. You can combine growth-rate functions
   - $O(n^2) + O(n)$, it is also $O(n^2+n)$
   - Which you write as $O(n^2)$
Examples

Find a growth function that has the best estimation of $O(x^2)$.

A. $f(x) = 17x + 11$
B. $f(x) = x^2 + 1000$
C. $f(x) = x \log x$
D. $f(x) = x^4/2$
E. $f(x) = 2^x$

Demonstrating Efficiency

- Computational complexity of the algorithm
  - Time complexity
  - Space complexity
    - Analysis of the computer memory required
    - Data structures used to implement the algorithm

Best, Average, and Worst Cases

- **Worst case**
  - Just how bad can it get:
    - The maximal number of steps

- **Average case**
  - Amount of time expected “usually”

- **Best case**
  - The smallest number of steps

Sequential Search

- Array of $n$ items
  - From the first one until either you find the item or reach the end of the array.
  - Best case: $O(1)$
  - Worst case: $O(n)$ ($n$ times of comparison)
  - Average case: $O(n)$ ($n/2$ comparison)
**Binary Search (1/4)**

- Searches a *sorted array* for a particular item by repeatedly dividing the array in half.
- Determines which half the item must be in and discards other half.
- Suppose that \( n = 2^k \) for some \( k \). (\( n=1,2,4,8,16,... \))
  1. Inspect the middle item of size \( n \)
  2. Inspect the middle item of size \( n/2 \)
  3. Inspect the middle item of size \( n/2^2 \)
  4. ...
  5. ...
  6. ...

**Binary Search (2/4)**

If we have \( n = 2^k \), in worst case, it will repeat this \( k \) times

**Clicker Q**

- What is the worst case for binary search?
  A. Item is at the end of the array
  B. Item is at the beginning of the array
  C. Item is not in the array
  D. The array is not sorted

**Binary Search (3/4)**

- Dividing array in *half* \( k \) times.
- Worst case
  - Algorithm performs \( k \) divisions and \( k \) comparisons.
  - Since \( n = 2^k \), \( k = \log_2 n \)
  - \( O(\log_2 n) \)
**Binary Search (4/4)**

- What if \( n \) is not a power of \( 2 \)?
- We can find the smallest \( k \) such that,

\[
2^{k-1} < n < 2^k \\
k - 1 < \log_2 n < k \\
k < 1 + \log_2 n < k + 1 \\
k = 1 + \log_2 n \text{ rounded down}
\]

Therefore, the algorithm is still \( O(\log_2 n) \).

---

**Is Binary Search is more Efficient than Linear Search? (1/2)**

- For large number, \( O(\log_2 n) \) requires significantly less time than \( O(n) \)
- For small numbers such as \( n < 25 \), does not show big difference.

---

**Sorting Algorithm**

- Organize a collection of data into either ascending or descending order.
- **Internal sort**
  - Collection of data fits entirely in the computer's main memory
- **External sort**
  - Collection of data will not fit in the computer's main memory all at once.
- We will only discuss *internal sort*. 
Aside: Sorting Redux from 161

- Simple Sorts: Bubble, Insertion, Selection
- Doubly nested loop
- Outer loop puts one element in its place
- It takes i steps to put element i in place
  - $n-1 + n-2 + n-3 + \ldots + 3 + 2 + 1$
  - $O(n^2)$ complexity
  - In place

Mergesort

- **Recursive sorting algorithm**
- Gives the same performance
- **Divide-and-conquer**
  - Step 1. Divide the array into halves
  - Step 2. Sort each half
  - Step 3. Merge the sorted halves into one sorted array

MergeSort code

```java
public static void mergesort(Comparable[] theArray, int first, int last){
    // Sorts the items in an array into ascending order.
    // Precondition: theArray[first..last] is an array.
    // Postcondition: theArray[first..last] is sorted.
    if (first >= last) {
        // if first <= last, there is nothing to do
    } else {
        int mid = (first + last) / 2; // midpoint of the array
        mergesort(theArray, first, mid);
        mergesort(theArray, mid + 1, last);
        merge(theArray, first, mid, last);
    }
}
```

why does it work?
Clicker Q

- How many times was MergeSort called?
  A. 1
  B. 6
  C. 10
  D. 20

Merge code I

```java
private static void merge (Comparable[] theArray, Comparable[] tempArray, int first, int mid, int last){
    int first1 = first;
    int last1 = mid;
    int first2 = mid+1;
    int last2 = last;
    int index = first1;   // incrementally creates sorted array
    while ((first1 <= last1) && (first2 <= last2)){
        if (theArray[first1].compareTo(theArray[first2])<0) {
            tempArray[index] = theArray[first1];
            first1++;
        } else{
            tempArray[index] = theArray[first2];
            first2++;
        } index++;
    }
```

Merge code II

```java
// finish off the two subarrays, if necessary
while (first1 <= last1){
    tempArray[index] = theArray[first1];
    first1++;
    index++;
}
while(first2 <= last2)
    tempArray[index] = theArray[first2];
    first2++;
    index++;
for (index = first; index <= last; ++index){
    tempArray[index ] = tempArray[index ];
}
```
Mergesort Complexity

- Analysis
  - Merging:
    - for total of $n$ items in the two array segments, at most $n-1$ comparisons are required.
    - $n$ moves from original array to the temporary array.
    - $n$ moves from temporary array to the original array.
    - Each merge step requires $3n - 1$ major operations

Mergesort: More complexity

- Each call to `mergesort` recursively calls itself twice.
- Each call to `mergesort` divides the array into two.
  - First time: divide the array into 2 pieces
  - Second time: divide the array into 4 pieces
  - Third time: divide the array into 8 pieces

Mergesort Levels

- If $n$ is a power of 2 (i.e. $n = 2^k$), then the recursion goes $k = \log_2 n$ levels deep.
- If $n$ is not a power of 2, there are $1 + \log_2 n$ (rounded down) levels of recursive calls to `mergesort`.

Mergesort Operations

- At level 0, the original call to `mergesort` calls merge once. (requires $3n - 1$ operations)
- At level 1, two calls to `mergesort` and each of them will call merge.
  - Total $2 \times (3 \times (n/2) - 1)$ operations required
- At level $m$, $2^m$ calls to `merge` occur.
  - Each of them will call merge with $n/2^m$ items and each of them requires $3(n/2^m) - 1$ operations. Together, $3n-2^m$ operations are required.
- Because there are $\log_2 n$ or $1+\log_2 n$ levels, total $O(n \log_2 n)$
Mergesort Computational Cost

- Since there are either \( \log_2 n \) or \( 1 + \log_2 n \) levels, mergesort is \( O(n \log_2 n) \) in both the worst and average cases.
- **Significantly faster** than \( O(n^2) \)

Clicker Q

- Is MergeSort \( O(n \log n) \) in the best case?
  A. Yes
  B. No

Stable Sorting Algorithms

- Suppose we are sorting a database of users according to their name. Users can have identical names.
- A **stable** sorting algorithm maintains the relative order of records with equal keys (i.e., sort key values). Stability: whenever there are two records \( R \) and \( S \) with the same key and \( R \) appears before \( S \) in the original list, \( R \) will appear before \( S \) in the sorted list.
- Is mergeSort stable?

Quicksort

1. Select a **pivot** item.
2. Subdivide array into 3 parts
   - Pivot in its “sorted” position
   - Subarray with **elements < pivot**
   - Subarray with **elements >= pivot**
3. **Recursively** apply to each sub-array
Invariant for partition

Pivot <- S1 -< S2 -< Unknown

first < P >= P ?

lastS1 firstUnknown last

Initial state of the array

Pivot <- Unknown

first firstUnknown last

lastS1

Lecture 12: 10/2/14

- Grammars (Prichard Ch. 6.2, Rosen Ch. 13.1)
- Stacks (Prichard Ch. 7)
- Recursion (Prichard Ch. 6.1 & 6.3)
- Queues (Prichard Ch. 8)
- Complexity (Rosen Ch. 3.2, 3.3, Prichard 10.1)
- Advanced Sorting (Prichard 10.2)

Today's Trivial Participation Quiz:
Which password was *not* on 2014 25 Most Common Passwords list?
- a. letmein
- b. trustno1
- c. monkey
- d. guessit

Quicksort Key Idea: Pivot

< p >= p

< p1 >= p1

< p2 >= p2
An invariant for the QuickSort code is:

A. After the first pass, the P< partition is fully sorted.
B. After the first pass, the P>= partition is fully sorted.
C. After each pass, the pivot is in the correct position.
D. It has no invariant.

```
public static void quickSort(Comparable[] theArray, int first, int last) {
    int pivotIndex;
    if (first < last) {
        // create the partition: S1, Pivot, S2
        pivotIndex = partition(theArray, first, last);
        // sort regions S1 and S2
        quickSort(theArray, first, pivotIndex-1);
        quickSort(theArray, pivotIndex+1, last);
    } // end if
} // end quickSort
```

1. Choose and position pivot
2. Take a pass over the current part of the array
   1. If item < pivot, move to S1 by incrementing S1 last position and swapping item into beginning of S2
   2. If item >= pivot, leave where it is
3. Place pivot in between S1 and S2

```
private static int partition(Comparable[] theArray, int first, int last) {
    Comparable tempItem;
    // place pivot in theArray[first]
    // by default, it is what is in first position
    choosePivot(theArray, first, last);
    Comparable pivot = theArray[first]; // reference pivot
    // initially, everything but pivot is in unknown
    int lastS1 = first; // index of last item in S1
```
// move one item at a time until unknown region is empty
for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown)
{ // move item from unknown to proper region
    if (theArray[firstUnknown].compareTo(pivot) < 0) {
        // item from unknown belongs in S1
        ++lastS1; // figure out where it goes
        tempItem = theArray[firstUnknown]; // swap it with first unknown
        theArray[firstUnknown] = theArray[lastS1];
        theArray[lastS1] = tempItem;
    } // end if
    // else item from unknown belongs in S2 - which is where it is!
} // end for

// place pivot in proper position and mark its location
tempItem = theArray[first];
theArray[first] = theArray[lastS1];
theArray[lastS1] = tempItem;
return lastS1;
} // end partition
QuickSort Visualizations

- http://www.sorting-algorithms.com
- Hungarian Dancers via YouTube

Average Case
- Each level involves,
  - Maximum \((n - 1)\) comparisons.
  - Maximum \((n - 1)\) swaps. \((3(n - 1)\) data movements)
  - \(\log_2 n\) levels are required.
- Average complexity \(O(n \log_2 n)\)

Clicker Q
- Is QuickSort like MergeSort in that it is always \(O(n \log n)\) complexity?
  A. Yes
  B. No

Worst Case!

```plaintext
before the partition
20 30 40 50 60 70

After the partition
20 30 40 50 60 70

quicksort(a, low, pivot-1); will not do anything!
```
Worst case analysis

- This case involves \((n-1)+(n-2)+(n-3)+...+1+0 = n(n-1)/2\) comparisons
- Quicksort is \(O(n^2)\) for the worst-case.

Selecting pivot

- Strategies for Selecting pivot
  - First value: worst case if the array is sorted.
  - Middle value or Median value: Better for the sorted data

quickSort – Algorithm Complexity

- Depth of call tree?
  - \(O(\log n)\) split roughly in half, best case
  - \(O(n)\) worst case
- Work done at each depth
  - \(O(n)\)
- Total Work
  - \(O(n \log n)\) best case
  - \(O(n^2)\) worst case

Clicker Q

- Why would someone pick QuickSort over MergeSort?
  A. Less space
  B. Better worst case complexity
  C. Better average complexity
  D. Easier to code
How fast can we sort?

- Observation: all the sorting algorithms so far are comparison sorts
  - A comparison sort must do at least $O(n)$ comparisons (why?)
  - We have an algorithm that works in $O(n \log n)$
  - What about the gap between $O(n)$ and $O(n \log n)$
- Theorem: all comparison sorts are $\Omega(n \log n)$
- MergeSort is therefore an “optimal” algorithm

Radix Sort (by MSD)

1. Take the most significant digit (MSD) of each number.
2. Sort the numbers based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate the buckets together in order.

Radix Sort

- To avoid using extra space: Radix sort by Least Significant Digit

RadixSort(A, d)

1. Take the most significant digit (MSD) of each number.
2. Sort the numbers based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate the buckets together in order.

Radix sort

- Analysis
  - $n$ moves each time it forms groups
  - $n$ moves to combine them again into one group.
  - Total $2n^d$ (for the strings of $d$ characters)
  - Radix sort is $O(n)$ for $d << n$
Radix Sort

- Radix sort is
  - Fast
  - Asymptotically fast (i.e., $O(n)$)
  - Simple to code
  - A good choice
- Can we use it for strings?
- So why not use it for every application?