CSCE 310J: Data Structures & Algorithms

Divide & Conquer!

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

CSCE 310J: Data Structures & Algorithms

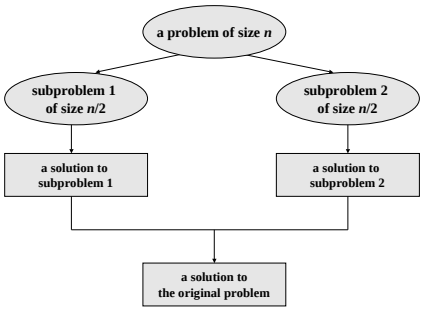
- **Giving credit where credit is due:**
  - **Most of the lecture notes are based on the slides from the Textbook's companion website**
    - **http://www.aw.com/cssuport/**
  - **Some examples and slides are based on lecture notes created by Dr. Ben Choi, Louisiana Technical University and Dr. Chuck Cusack, UNL**
  - **I have modified many of their slides and added new slides.**

Divide and Conquer

The most well known algorithm design strategy:

1. **Divide instance of problem into two or more smaller instances**
2. **Solve smaller instances recursively**
3. **Obtain solution to original (larger) instance by combining these solutions**

Divide-and-conquer technique



Divide and Conquer Examples

- **Sorting: mergesort and quicksort**
- **Tree traversals**
- **Binary search**
- **Matrix multiplication-Strassen's algorithm**
- **Convex hull-QuickHull algorithm**

General Divide and Conquer recurrence:

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^k)$

1. $a < b^k$ $T(n) \in \Theta(n^k)$
2. $a = b^k$ $T(n) \in \Theta(n^k \lg n)$
3. $a > b^k$ $T(n) \in \Theta(n^{\log_b a})$

Note: **the same results hold with O instead of Θ.**

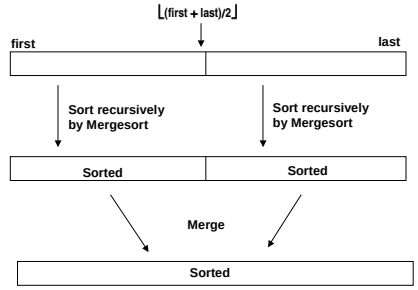
Mergesort

Algorithm:

- **Split array A[1..*n*] in two and make copies of each half in arrays B[1..⌊*n*/2⌋] and C[1.⌈*n*/2⌉]**
- **Sort arrays B and C**
- **Merge sorted arrays B and C into array A as follows:**
  - **Repeat the following until no elements remain in one of the arrays:**
    - **compare the first elements in the remaining unprocessed portions of the arrays**
    - **copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array**
  - **Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.**

Using Divide and Conquer: Mergesort

- **Mergesort Strategy**



Mergesort Examples

Animated Example:
http://www.cs.hope.edu/~alganim/animator/Animator.html
Another animated example:
http://math.hws.edu/TMCM/java/xSortLab/index.html

Algorithm: Mergesort

Input: Array E and indices first and last, such that the elements E[i] are defined for first <= i <= last.
Output: E[first], …, E[last] is a sorted rearrangement of the same elements

```
void mergeSort(Element[] E, int first, int last)
    if (first < last)
        int mid = (first+last)/2;
        mergeSort(E, first, mid);
        mergeSort(E, mid+1, last);
        merge(E, first, mid, last);
    return;
```


Merging Sorted Sequences

- **Problem:**
  - **Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C**
- **Strategy:**
  - **Determine the first item in C**
  - **It is the minimum between the first items of A and B.**
    - **Suppose it is the first items of A.**
    - **Then, rest of C consisting of merging rest of A with B.**

Algorithm: Merge

- **Merge(A, B, C)**

```
if (A is empty)
    rest of C = rest of B
else if (B is empty)
    rest of C = rest of A
else if (first of A ≤ first of B)
    first of C = first of A
    merge (rest of A, B, rest of C)
else
    first of C = first of B
    merge (A, rest of B, rest of C)
return
```

- **W(n) = n – 1**

Evaluating Sort Algorithms

- ***Run-time*: The number of basic operations performed (e.g., compare and swap)**
- ***Memory*: The amount of memory used beyond what is needed to store the data being sorted**
  - **"In place" algorithms use a constant amount of extra memory—the constant may be zero**
  - **Other algorithms are described as linear or exponential with respect to the space used.**
  - **Less is better, but there is often a space/time trade-off.**
- ***Stability*: An algorithm is stable if it preserves the relative order of equal keys**

Mergesort Complexity

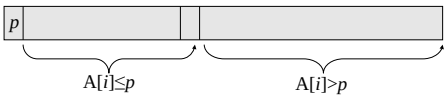
- **Mergesort always partitions the array equally.**
- **Thus, the recursive depth is always $O(\lg n)$**
- **The amount of work done at each level is $O(n)$**
- **Intuitively, the complexity should be $O(n \lg n)$**
- **Once again we have,**
  - $T(n) \leq 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \lg n)$
- **The amount of extra memory used is $O(n)$**
- **Note: Mergesort is stable**

Efficiency of mergesort

- **All cases have same efficiency: $\Theta(n \log n)$**
- **Number of comparisons is close to theoretical minimum for comparison-based sorting:**
  - $\lceil \log n ! \rceil \approx \lceil n \lg n - 1.44 n \rceil$
- **Space requirement: $\Theta(n)$ (NOT in-place)**
- **Can be implemented without recursion (bottom-up)**

Quicksort by Hoare (1962)

- **Select a *pivot* (partitioning element)**
- **Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm *Partition* in section 4.2)**
- **Exchange the pivot with the last element in the first (i.e., ≤ sublist) – the pivot is now in its final position**
- **Sort the two sublists recursively**



The partition algorithm

```
Algorithm Partition(A[l..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//       indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l;  j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

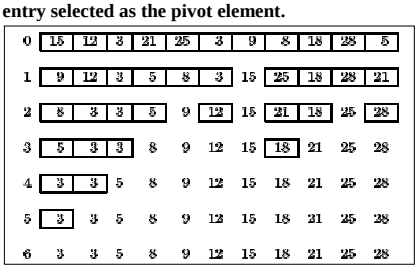

Quicksort

Animated Example:
http://www.cs.hope.edu/~alganim/animator/Animator.html

Another animated example:
http://math.hws.edu/TMCM/java/xSortLab/index.html

Quicksort Example

- Recursive implementation with the left most array entry selected as the pivot element.



Quicksort Algorithm

- Input: Array E and indices first, and last, s.t. elements E[i] are defined for first ≤ i ≤ last
- Ouput: E[first], ..., E[last] is a sorted rearrangement of the array
- Void quickSort(Element[] E, int first, int last)

```
if (first < last)
    Element pivotElement = E[first];
    Key pivot = pivotElement.key;
    int splitPoint = partition(E, pivot, first, last);
    E[splitpoint] = pivotElement;
    quickSort (E, first, splitPoint –1 );
    quickSort (E, splitPoint +1, last );
return;
```


Quicksort Analysis

- Partition can be done in O(n) time, where n is the size of the array
- Let $T(n)$ be the number of compares required by Quicksort
- If the pivot ends up at position k, then we have
  - $T(n) = T(n–k) + T(k –1) + n$
- To determine best-, worst-, and average-case complexity we need to determine the values of k that correspond to these cases.

Best-Case Complexity

- The best case is clearly when the pivot always partitions the array equally.
- Intuitively, this would lead to a recursive depth of at most lg n calls
- We can actually prove this. In this case
  - $T(n) \le T(n/2) + T(n/2) + n \Rightarrow \Theta(n \lg n)$

Worst-Case and Average-Case Complexity

- The worst-case is when the pivot always ends up in the first or last element. That is, partitions the array as unequally as possible.
- In this case
  - $T(n) = T(n–1) + T(1–1) + n = T(n–1) + n$
    $= n + (n–1) + \ldots + 1$
    $= n(n + 1)/2 \Rightarrow O(n^2)$
- Average case is rather complex, but is where the algorithm earns its name. The bottom line is:

$$A(n) \approx 1.386 n \lg n \Rightarrow \Theta(n \lg n)$$

Summary of quicksort

- *Best case*: split in the middle — $\Theta(n \log n)$
- *Worst case*: sorted array! — $\Theta(n^2)$
- *Average case*: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning avoids worst case in sorted files
  - switch to insertion sort on small subfiles
  - elimination of recursion

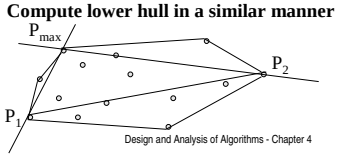
  these combine to 20-25% improvement

- Considered the method of choice for internal sorting for large files ($n \ge 10000$)

QuickHull Algorithm

Inspired by Quicksort compute Convex Hull:

- **Assume points are sorted by *x*-coordinate values**
- **Identify extreme points P_1 and P_2 (part of hull)**
- **Compute upper hull:**
  - **find point P_{max} that is farthest away from line P_1P_2**
  - **compute the hull of the points to the left of line P_1P_{max}**
  - **compute the hull of the points to the left of line $P_{max}P_2$**
- **Compute lower hull in a similar manner**



Efficiency of QuickHull algorithm

- **Finding point farthest away from line P_1P_2 can be done in linear time**
- **This gives same efficiency as quicksort:**
  - ***Worst case*: $\Theta(n^2)$**
  - ***Average case*: $\Theta(n \log n)$**
- **If points are not initially sorted by x-coordinate value, this can be accomplished in $\Theta(n \log n)$ — no increase in asymptotic efficiency class**
- **Other algorithms for convex hull:**
  - **Graham's scan**
  - **DCHull**

  also in $\Theta(n \log n)$