CS472 Module 4 Part B - The Nature of Sorting

Athens State University

February 9, 2017

Outline

Contents

1 Why study sorting?

Sorting? Again?

- You see sorting three times: in CS318, CS372, and now here in CS472
- So, why?
  - Sorting is the basic building block that many algorithms are built around
  - Most of the meta-heuristics used in algorithm design appear in the context of sorting
  - Computers have spent more time sorting stuff than doing anything else
  - Sorting is the most thoroughly studied problem in computer science

Consider...

n	$(n^2)/4$	n lg(n)
10	25	33
100	2500	664
1000	250000	9965
10000	25000000	132877
100000	2500000000	1660960

- Quadratic algorithm may work if $n = 10000$ but not beyond that point
- So, looking for $O(n * lg(n))$ algorithms

2 Pragmatics of sorting

Pragmatics of sorting

- Increasing or decreasing order?
  - *Ascending order* - $S_i \leq S_{i+1} \forall 1 \leq i < n$
  - *Decreasing order* - $S_i \geq S_{i+1} \forall 1 \leq i < n$
- Sorting just the key or an entire record?

Pragmatics of sorting

- What should we do with equal keys?
  - Just where in the list do you put Michael Jackson (the musician) vs. Michael Jackson (the world renowned expert on craft beer)?
  - *Stable sort* - A stable sort is one that leaves items in the same relative order as in original permutation
  - Some algorithms can fall back to quadratic worst-case performance if one does not take large numbers of ties (quicksort is well known for this behavior)

Pragmatics of sorting

- What about non-numerical data?
  - *Alphabetizing* - Sorting of text strings
  - *Collating sequence* - rules considering relative placement of keys
    - Example: Does *Brown-Williams* come before or after *Brown America*, and before or after *Brown, John*?

Pragmatics of sorting

- Here's where we introduce the idea of a *comparison function*
- *Comparison function* - A relation in an ADT's interface set I that for a value of $s \in S$ partitions the value set S into three subsets: s, $a : a < s$ and $b : b > s$.
- More useful is the pairwise-element version of such functions s.t. given $a, b \in S$, return "$<$" if $a < b$, "$>$" if $a > b$, or "$=$" if $a = b$.

Pragmatics of sorting

- Consider the *qsort()* library function in the C standard library

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
   int (*compare) (const void *, const void *));
```

- We see something similar with the *Comparable* interface in Java and *IComparable* interface in C#

```
public interface IComparable {
   int CompareTo( Object obj);
}
```

3 Sorting is the starting point

Sorting is the starting point

- *Searching* - binary search is $O(lg(n))$, if data is sorted
- *Closet pair* - Given a set of n numbers, how do you find the pair of numbers with the smallest difference between them?
  - Sort the list, then closest pair must lie next to each other in the list
  - Linear search the list, get an $O(nlg(n))$ algorithm!
- *Element uniqueness* - Are there any duplicates in a given set of n items?
- *Frequency distribution* - Given a set of n items, which element occurs the largest number of times in the set?
- *Selection* - What is the k-th item in an array?

Example: What is the intersection of two sets?
Problem statement

Give an efficient algorithm to determine whether two sets of size m and n are disjoint. Analyze the worst-cast time complexity in terms of m and n, considering the case where m is substantially smaller than n.

Example: What is the intersection of two sets?
Solutions

1. Sort the big set
   - Set can be sorted in $O(n * lg(n))$ time.
   - Binary search with each of the m elements in the second set
   - Total time required will be $O((n + m) * lg(n))$
2. Sort the small sets
   - Set can be sorted in $O(m * lg(m))$ time.
   - Binary search with each of the n elements in the big set
   - Total time will be $O((n + m) * lg(m))$
3. Sort both sets
   - Sort both sets with $O(n * lg(n))$ and $O(m * lg(m))$ time
   - Compare the smallest element in each set, discard that element if they are identical
   - Repeat recursively. Note this will occur in linear time
   - Total cost is $O(n * lg(n) + m * lg(m) + n + m)$

Example: What is the intersection of two sets?
Analysis

- Note that $lg(m) < lg(n)$ when $m < n$
- And note that $n + m < 2n$ when $m < n$
- Thus, we can claim that $(n + m) * lg(m)$ is asymptotically less than $n * lg(n)$
- So, sorting the small set is the best option

4 Brute-force sorting

Brute-force sorting

- Let's reconsider the question:
  - What would be the most straightforward method for solving the sorting problem?
- Many different answers to that philosophical question
  - But we'll look at selection sort and bubble sort

Brute-force sorting: Selection sort

- Scan the input to find the its smallest element and swap it with the first element.
- Then starting with the second element, find the smallest element and swap it with the second element
- In general, on pass i, $0 \le i \le n-2$, find the smallest element in $A[i \dots n-1]$ and swap it with $A[i]$
- Example: use selection sort to sort <7 3 2 5>

Brute force sorting : Selection sort

Algorithm 1: Sort a given array by selection sort

```
Input: An array A[0..n-1] of orderable elements
Output: An array A[0..n-1] sorted in ascending order
for i ← 0..(n-2) do
    min ← 1;
    for j ← (i+1) .. (n-1) do
        if A[j] < A[min] then
            min ← j;
    swap A[i] and A[min];
```

Brute force sorting: Selection sort: analysis

For sorting, the basic operation is the comparison of the array elements.
The number of times the comparison is executed depends the array size n and is given by the formula

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n} (n-1-i) = \frac{(n-1)n}{2}$$

Brute force sorting: Bubble sort

The very first sort most students is taught is the *bubble sort*

- Compare adjacent elements and swap if they are out of order
- Largest element "bubbles" to last position of the list
- Repeat with next element for up to $n-1$ passes through the list

Brute force sorting: Bubble sort: algorithm

Algorithm 2: Sort a given list by bubble sort

Input: An array A[0..(n-1)] of orderable elements
Output: Array A[0..(n-1)] sorted in ascending order
for $i \leftarrow$ *0..(n-2)* **do**
  for $j \leftarrow$ *0 .. (n-2-i)* **do**
    if *A[j+1] < A[j]* **then**
      swap A[j] with A[j+1];

Brute force sorting: Bubble sort: analysis

Note the similarity to selection sort:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \\ &= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \\ &\in \Theta(n^2) \end{aligned}$$

5 Key points

Key points

- Why study sorting?
- Pragmatics of sorting
- Brute force sorting: Selection and bubble sorts