CS 61B Data Structures and Programming Methodology

July 21, 2008

David Sun

Sort

- Since the dawn of computing, sorting has attracted a great deal of attention due to the complexity of solving it *efficiently* despite the simple problem statement.
- A sorting algorithm is an algorithm that puts the elements of a list in a certain order: numerical order or lexicographical order.
- Sorting supports basic searching:
  - for a number in a phone book
  - a website with the most relevant information to your search query.
- Sorting is perhaps the simplest fundamental problem that offers a large variety of algorithms, each with its own inherent advantages and disadvantages.

Bubble Sort

- Simple idea:
  - Step through the list to be sorted, compare adjacent elements, swap them if they are in the wrong order.
  - Repeat the pass through the list until no swaps are needed.
  - Invariant: after the *kth* iteration, the *k*-largest elements are at the end of the list.
- An example.

How Good Is Bubble Sort?

Let's hear what Senator Obama has to say

Bubble Sort

- Although simple to understand and implement, has worst case $O(n^2)$ running time, which means it is far too inefficient for practical usage.
- The generic **bad** algorithm:
  - "*the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems“ – Donald Knuth*

Insertion Sort

- Simple idea:
  - Starting with empty sequence of outputs *S* and the unsorted list of *n* input items *I.*
  - Add each item from input *I*, inserting into output sequence *S* at a position so the output is still in sorted order.
  - Invariant: at the *k*th iteration, the elements from *0* to *k-1* in the output sequence *S* are sorted.
- An example

Insertion Sort

- If destination is a linked list
  - *Theta(n)* worst-case time to find the right position of *S* to insert each item.
  - *Theta(1)* time to insert the item.
- If destination is an array
  - Find the right position in *S* in *O(log n)* time by binary search.
  - *Theta(n)* worst-case time to shift the larger items over to make room for the new item.

In-place Insertion Sort

- If *S* is an array, we can do an in place sort:
  - Store sorted items in the same array that initially held the input items
  - Partition the array into two pieces: the left portion (initially empty) holds *S*, and the right portion holds *I*.
  - With each iteration, the dividing line between *S* and *I* moves one step to the right.

Running time

- What's the best case?
  - Sorted array: just compare the *first* remaining element of the input against the *last* element of the sorted subsection of the array.
  - The running time is proportional to the number of *inversions.*
  - Runs in $O(n)$ where n is the number of elements.
- What's the worst-case?
  - Inversely sorted array: every iteration, you need to scan and shift the entire sorted portion of the array before inserting the next element.
  - $O(n^2)$ where n is the number of elements.

Insertion Sort Using Binary Search Tree

- You can use binary search tree to store the output:
  - Insertion into the binary search tree is *O(log n)*, so insertion sort takes *O(n log n).*
  - But there are better *O(n log n)* alternatives.

Selection Sort

- Simple idea:
  - Starting with empty output *S* and the unsorted list of *n* input items *I.*
  - Walk through *I* and find the **smallest** item, remove the item and append to the end of the output *S* .
  - Invariant: at the *k*th iteration, the *k smallest* elements of the input *I* are sorted.
- An example.

Selection Sort v.s Insertion Sort

- At the *kth* iteration
  - Selection sort must find the *smallest* item in the remaining list: selecting the lowest element requires scanning all *n* elements. Finding the next lowest element requires scanning the remaining *n* - 1 elements :

    $(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1) / 2 = \Theta(n^2)$.

    so selection sort takes *Theta(n^2)* time, even in the best case.
  - Insertion sort only examines the sorted portion of the list, so on average it examines half as many elements. For the best case, it only examines the right most element in the sorted part of the list.

In-place Selection Sort

- If *S* is an array, we can do an in-place selection sort.
- Divide the list into two parts:
  - the sublist of items already sorted, which we build up from left to right and is found at the beginning, and
  - the sublist of items remaining to be sorted, occupying the remainder of the array.
  - After finding the item in *I* having smallest key, swap it with the first item in *I*.
- An example.

Heapsort

- Heapsort is a selection sort in which I is a heap.
  1. `Start with an empty list` *`S`* `and an unsorted list` *`I`* `of n input items`
  2. `Put all the items in` *`I`* `onto an array and perform bottomUpHeap()`
  3. `At each iteration, remove the max or min element from the heap while maintaining the heap property; append the element at the end of S`
- `bottomUpHeap()` runs in linear time, and each `removeMin()` takes *O(log n)* time. Hence, heapsort is an *O(n log n)* sorting algorithm, even in the worst case.

Heapsort

- Heapsort can be implemented *in-place* using an array to achieve constant time space overhead.
  - Store the heap in reverse order.
  - As items are removed from the heap, the heap shrinks toward the end of the array, making room to add items to the end of *S*.
- An Example.
- Heapsort relies strongly on random access, so it excellent for sorting arrays, but not so for linked lists.
  - One can turn a linked list into an array. Sort the array of listnode references . When the array is sorted, link all the listnodes together into a sorted list.

Merge Two Sorted Lists

- Observation: one can merge two sorted lists into one sorted list in linear time.
- Psuedocode:

```
Let Q1 and Q2 be two sorted queues.
Let Q be an empty queue.
merge(Q, Q1, Q2) {
    while (neither Q1 nor Q2 is empty) {
          item1 = Q1.front();
          item2 = Q2.front();
          move the smaller of item1 and item2 from
          its present queue to end of Q.
    }
    concatenate the remaining non-empty queue (Q1
    or Q2) to the end of Q.
}
```

- **`merge(Q, Q1, Q2)`** takes *O(n)* time.

Recurrence (not examable)

- When an algorithm contains a recursive call to it self, the running time can be described using a recursive relation - describing the running time of a problem of size *n* in terms of running time of smaller inputs.
- General framework:
  - Let *T(n)* be the running time on a problem of size *n*
  - The problem is divided into *a* subproblems, each of which is *1/b* in size.
  - If it takes *D(n)* time to divide he problem into subproblems
  - If it takes *C(n)* time to combine the solutions to the subproblems:
  - $T(n) = a * T(n/b) + D(n) + C(n)$ *if* $n >= c$*(constant)*

Mergesort

- Mergesort is a recursive divide-and-conquer algorithm:

```
1. Start with the unsorted list I of n input items.
2. If n is 0 or 1 then it is sorted. Otherwise:
3. Break I into two halves I1 and I2 of about half the
   size.
4. Sort I1 recursively, yielding the sorted list S1.
5. Sort I2 recursively, yielding the sorted list S2.
6. Call merge() to put S1 and S2 into a sorted list S.
```

- What's the time complexity of Mergesort?
  - Each recursive call involves *O(n)* operations, and there are *O(log n)* recursive calls
  - Mergesort runs in *O(n log n).*

Mergesort

- Mergesort and heapsort are both *O(n log n),*
- Mergesort requires *Theta(n)* additional space compared to *Theta(1)* for heapsort.
- Mergesort is efficient for linked lists.
- Mergesort is not an in-place algorithm.
  - There are ways to do it, but very complicated and offer little performance gains.

Readings

- Objects, Abstraction, Data Structures and Design
  - Chapter 10 pp519 – 522 pp525 – 529 pp535 - 545