Sorting

8/04/2009

Many thanks to David Sun for some of the included slides!

Important Dates

- There IS class next week!
- Project Work
  - Wednesday's Lab 8/5/2009
- Project 3
  - You may have 1 or 2 partners.
    - NO EXCEPTIONS!
  - Due Tuesday 8/11/2009 – 10pm
- Final Review
  - Sunday 8/09/2009 – 1-4pm in 306 Soda
- Final
  - Thursday 8/13/2009 – 5-**8**pm in 10 Evans

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.

15	3	1	9	4	7	5	4	0	12
3	15	1	9	4	7	5	4	0	12
3	1	15	9	4	7	5	4	0	12
3	1	9	15	4	7	5	4	0	12
3	1	9	4	15	7	5	4	0	12
3	1	9	4	7	15	5	4	0	12
3	1	9	4	7	5	15	4	0	12
3	1	9	4	7	5	4	15	0	12
3	1	9	4	7	5	4	0	15	12
3	1	9	4	7	5	4	0	12	15

Bubble Sort

- Although not bad 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

- The 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

15	3	1	9	4	7	5	4	0	12
3	15	1	9	4	7	5	4	0	12
1	3	15	9	4	7	5	4	0	12
1	3	9	15	4	7	5	4	0	12
1	3	4	9	15	7	5	4	0	12
1	3	4	7	9	15	5	4	0	12
1	3	4	5	7	9	15	4	0	12
1	3	4	4	5	7	9	15	0	12
0	1	3	4	4	5	7	9	15	12
0	1	3	4	4	5	7	9	12	15

1	3	4	4	5	7	9	15	0	12
1	3	4	4	5	7	9	0	15	
1	3	4	4	5	7	0	9	15	
1	3	4	4	5	0	7	9	15	
1	3	4	4	0	5	7	9	15	
1	3	4	0	4	5	7	9	15	
1	3	0	4	4	5	7	9	15	
1	0	3	4	4	5	7	9	15	
0	1	3	4	4	5	7	9	15	12

What is the run-time of Insertion Sort?

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$$

$$+ \quad \sum_{i=0}^{n} i = n + (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

$$2\sum_{i=0}^{n} i = \begin{matrix} 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n + \\ n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 \end{matrix}$$

$$2\sum_{i=0}^{n} i = \begin{matrix} \boxed{1 + n} + \boxed{2 + (n-1)} + \boxed{3 + (n-2)} + \cdots + \boxed{(n-2) + 3} + \boxed{(n-1) + 2} + \boxed{n + 1} \end{matrix}$$

$$2\sum_{i=0}^{n} i = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1)$$

$$2\sum_{i=0}^{n} i = n(n+1) \quad \text{OR} \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Insertion Sort

- Linked list
  - *O(n)* worst-case time to find the right position of *S* to insert each item.
  - *O(1)* time to insert the item.
- Array
  - Find the right position in *S* in *O(log n)* time by binary search.
  - *O(n)* worst-case time to shift the larger items over to make room for the new item.

Selection Sort

- The 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

15	3	1	9	4	7	5	4	0	12
0	15	3	1	9	4	7	5	4	12
0	1	15	3	9	4	7	5	4	12
0	1	3	15	9	4	7	5	4	12
0	1	3	4	15	9	7	5	4	12
0	1	3	4	4	15	9	7	5	12
0	1	3	4	4	5	15	9	7	12
0	1	3	4	4	5	7	15	9	12
0	1	3	4	4	5	7	9	15	12
0	1	3	4	4	5	7	9	12	15
0	1	3	4	4	5	7	9	12	15

Selection Sort vs 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 :

    $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 = O(n^2)$.

    so selection sort takes $O(n^2)$ 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.

Heapsort

- Heapsort is a type of 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 – Remember the demo?

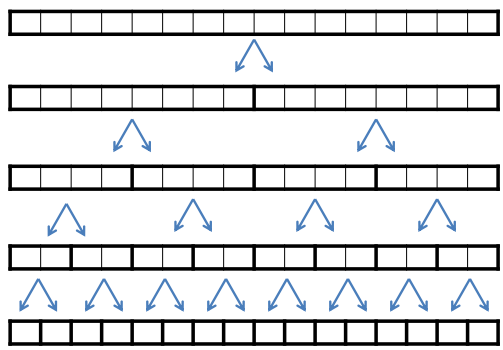
- Heapsort can be implemented *in-place* using an array to achieve constant time space overhead.
  - Store the heap in reverse order.
  - As items are removed from the heap, the heap shrinks toward the end of the array, making room to add items to the end of *S*.
- Heapsort relies strongly on random 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.



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 *O(n)* additional space compared to *O(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.

Quick Sort

- Quicksort is a recursive divide-and-conquer algorithm, like mergesort.
- Quicksort is in practice the fastest known comparison-based sort for arrays, even though it has $O(n^2)$ worst-case running time.
- If properly designed, however, it virtually always runs in *O(n log n)* time.

The Algorithm

```
1. Given an unsorted list I of n
   items
2. Choose a pivot item v from I.
3. Partition I into two unsorted
   lists I1 and I2  I1 contains all
   items whose keys are smaller than
   v's key. I2 contains all items
   whose keys are larger than v's.
   Equal values can go either way.
```

- (continued)

```
4. After the partition v is in it's
   final position.
5. Sort I1 recursively, yielding the
   sorted list S1.
6. Sort I2 recursively, yielding the
   sorted list S2.
7. Concatenate S1, v, and S2
   together, yielding a sorted list
   S.
```

Running Time of Quicksort

- The running time depends on whether the partitioning is *balanced* or *unbalanced*, which depends on the choice of the pivot
  - If choice of pivot is good, quicksort runs asymptotically as fast as mergesort: *O(n log n)*.
  - If choice of pivot is bad, then it runs asymptotically as slowly as insertion sort: $O(n^2)$.