Sorting

CSE 373
Data Structures

Reading

- Reading
  - Goodrich and Tamassia, Chapter 10
Sorting

- Input
  - an array A of data records (Note: we have seen how to sort when elements are in linked lists: Mergesort)
  - a key value in each data record
  - a comparison function which imposes a consistent ordering on the keys (e.g., integers)

- Output
  - reorganize the elements of A such that
    - For any i and j, if i < j then A[i] ≤ A[j]

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed? O(n) additional space
  - In-place sorting – no copying – O(1) additional space
  - Somewhere in between for “temporary”, e.g. O(log n) space
  - External memory sorting – data so large that does not fit in memory
Time

• How fast is the algorithm?
  › The definition of a sorted array A says that for any i<j, A[i] < A[j]
  › This means that you need to at least check on each element at the very minimum, i.e., at least O(N)
  › And you could end up checking each element against every other element, which is O(N²)
  › The big question is: How close to O(N) can you get?

Stability

• Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  › E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  › Extremely important property for databases
  › A stable sorting algorithm is one which does not rearrange the order of duplicate keys
Bubble Sort

- “Bubble” elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
  - Bubble every element towards its correct position
    - last position has the largest element
    - then bubble every element except the last one towards its correct position
    - then repeat until done or until the end of the quarter, whichever comes first ...
Bubblesort

bubble(A[1..n]: integer array, n : integer): {
  i, j : integer;
  for i = 1 to n-1 do
    for j = 2 to n–i+1 do
}

SWAP(a,b) : {
  t : integer;
  t:=a; a:=b; b:=t;
}

Put the largest element in its place

larger value?

4/22/2004 CSE 373 SP 04 -- Sorting
Put 2\textsuperscript{nd} largest element in its place

```
1 2 3 7 8 9 10 12 18 18
larger value? 2
1 2 3 7 8 9 10 15 16 17 14
1 2 3 7 8 9 12 15 16 18 17 14
1 2 3 7 8 9 10 12 15 16 18 17 14
1 2 3 7 8 9 10 12 15 16 17 18 14
1 2 3 7 8 9 10 12 15 16 17 14 18
```

Two elements done, only n-2 more to go ...

---

Bubble Sort: \textbf{Just Say No}

- “Bubble” elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
- We bubblize for i=1 to n (i.e, n times)
- Each bubblization is a loop that makes n-i comparisons
- This is O(n\(^2\))
Insertion Sort

- What if first $k$ elements of array are already sorted?
  - $4, 7, 12, 5, 19, 16$
- We can shift the tail of the sorted elements list down and then insert next element into proper position and we get $k+1$ sorted elements
  - $4, 5, 7, 12, 19, 16$

```latex
InsertionSort(A[1..N]: integer array, N: integer) {
  i, j, temp: integer;
  for i = 2 to N {
    temp := A[i];
    j := i;
    while j > 1 and A[j-1] > temp {
    }
    A[j] = temp;
  }
}
```

- Is Insertion sort in place?
- Running time = ?
Example
Insertion Sort Characteristics

- In place and Stable
- Running time
  - Worst case is $O(N^2)$
    - reverse order input
    - must copy every element every time
- Good sorting algorithm for almost sorted data
  - Each item is close to where it belongs in sorted order.

Heap Sort

- We use a Max-Heap
- Root node = $A[1]$
- Keep track of current size $N$ (number of nodes)
Using Binary Heaps for Sorting

- Build a **max-heap**
- Do N **DeleteMax** operations and store each Max element as it comes out of the heap
- Data comes out in largest to smallest order
- Where can we put the elements as they are removed from the heap?

1 Removal = 1 Addition

- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store
  - Store the data at the end of the heap array
  - Not "in the heap" but it is in the heap array
Repeated DeleteMax

\[
\begin{array}{cccccccc}
5 & 2 & 4 & 6 & 7 & & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\(N = 3\)

\[
\begin{array}{cccccccc}
4 & 2 & 5 & 6 & 7 & & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\(N = 2\)

Heap Sort is In-place

- After all the DeleteMaxs, the heap is gone but the array is full and is in sorted order

\[
\begin{array}{cccccccc}
2 & 4 & 5 & 6 & 7 & & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\(N = 0\)
Heapsort: Analysis

- **Running time**
  - time to build max-heap is $O(N)$
  - time for $N$ DeleteMax operations is $N \cdot O(\log N)$
  - total time is $O(N \log N)$
- Can also show that running time is $\Omega(N \log N)$ for some inputs,
  - so **worst case** is $\Theta(N \log N)$
  - **Average case** running time is also $O(N \log N)$
- Heapsort is in-place but not stable (why?)

“Divide and Conquer”

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves $\rightarrow$ Mergesort
- **Idea 2**: Partition array into items that are “small” and items that are “large”, then recursively sort the two sets $\rightarrow$ Quicksort
Mergesort

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example
Auxiliary Array

• The merging requires an auxiliary array.

2 4 8 9 1 3 5 6

Auxiliary array
Auxiliary Array

- The merging requires an auxiliary array.

Auxiliary array

Merging

- normal
- target
- Left completed first

4/22/2004 CSE 373 SP 04 -- Sorting
Merging

Merging Algorithm

Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i < mid and j < right do
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
    if i > mid then //left completed//
        for k := left to target-1 do A[k] := T[k];
    if j > right then //right completed//
        k := mid; l := right;
        while k >= i do A[l] := A[k]; k := k-1; l := l-1;
        for k := left to target-1 do A[k] := T[k];
}
Recursive Mergesort

Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A, T, left, mid);
    Mergesort(A, T, mid+1, right);
    Merge(A, T, left, right);
  }

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort(A, T, 1, n);
}

Iterative Mergesort

uses 2 arrays; alternates between them
Iterative Mergesort

IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2//
i, m, parity : integer;
T[1..n]: integer array;
m := 2; parity := 0;
while m < n do
  for i = 1 to n - m + 1 by m do
    if parity = 0 then Merge(A,T,i,i+m -1);
    else Merge(T,A,i,i+m -1);
    parity := 1 – parity;
m := 2*m;
if parity = 1 then
  for i = 1 to n do A[i] := T[i];
}

How do you handle non-powers of 2?
How can the final copy be avoided?
Mergesort Analysis

- Let $T(N)$ be the running time for an array of $N$ elements.
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array.
- Each recursive call takes $T(N/2)$ and merging takes $O(N)$.

Mergesort Recurrence Relation

- The recurrence relation for $T(N)$ is:
  - $T(1) \leq a$
    - base case: 1 element array $\to$ constant time
  - $T(N) \leq 2T(N/2) + bN$
    - Sorting $N$ elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an $O(N)$ time to merge the two halves
- $T(N) = O(n \log n)$
Properties of Mergesort

• Not in-place
  › Requires an auxiliary array \(O(n)\) extra space

• Stable
  › Make sure that left is sent to target on equal values.

• Iterative Mergesort reduces copying.

Quicksort

• Quicksort uses a divide and conquer strategy, but does not require the \(O(N)\) extra space that MergeSort does
  › Partition array into left and right sub-arrays
    • Choose an element of the array, called pivot
    • the elements in left sub-array are all less than pivot
    • elements in right sub-array are all greater than pivot
  › Recursively sort left and right sub-arrays
  › Concatenate left and right sub-arrays in \(O(1)\) time
“Four easy steps”

• To sort an array $S$
  1. If the number of elements in $S$ is 0 or 1, then return. The array is sorted.
  2. Pick an element $v$ in $S$. This is the pivot value.
  3. Partition $S - \{v\}$ into two disjoint subsets, $S_1 = \{\text{all values } x \leq v\}$, and $S_2 = \{\text{all values } x \geq v\}$.
  4. Return QuickSort($S_1$), $v$, QuickSort($S_2$)

The steps of QuickSort

1. Select pivot value.
2. Partition $S$.
3. Return QuickSort($S_1$) and QuickSort($S_2$).
4. Voila! $S$ is sorted.

[Weiss]
Details, details

• Implementing the actual partitioning
• Picking the pivot
  › want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
• Dealing with cases where the element equals the pivot

Quicksort Partitioning

• Need to partition the array into left and right sub-arrays
  › the elements in left sub-array are $\leq$ pivot
  › elements in right sub-array are $\geq$ pivot
• How do the elements get to the correct partition?
  › Choose an element from the array as the pivot
  › Make one pass through the rest of the array and swap as needed to put elements in partitions
Partitioning: Choosing the pivot

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; "expensive")
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - Swap pivot with next to last element

Partitioning in-place

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit elmt A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]
Example

Choose the pivot as the median of three

\[ 0 \quad 1 \quad 4 \quad 9 \quad 0 \quad 3 \quad 5 \quad 2 \quad 7 \quad 6 \]

Median of 0, 6, 8 is 6. Pivot is 6

\[ 0 \quad 1 \quad 4 \quad 9 \quad 7 \quad 3 \quad 5 \quad 2 \quad 6 \quad 8 \]

Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example

Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap
Recursive Quicksort

Quicksort(A[]): integer array, left, right : integer): {
  pivotindex : integer;
  if left + CUTOFF ≤ right then
    pivot := median3(A,left,right);
    pivotindex := Partition(A,left,right-1,pivot);
    Quicksort(A, left, pivotindex - 1);
    Quicksort(A, pivotindex + 1, right);
  else
    Insertionsort(A,left,right);
}

Don't use quicksort for small arrays.
CUTOFF = 10 is reasonable.
Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - \( T(0) = T(1) = O(1) \)
    - constant time if 0 or 1 element
  - For \( N > 1 \), 2 recursive calls plus linear time for partitioning
  - \( T(N) = 2T(N/2) + O(N) \)
    - Same recurrence relation as Mergesort
  - \( T(N) = O(N \log N) \)

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - \( T(N) \leq a \) for \( N \leq C \)
  - \( T(N) \leq T(N-1) + bN \)
  - \( \leq T(N-2) + b(N-1) + bN \)
  - \( \leq T(C) + b(C+1) + \ldots + bN \)
  - \( \leq a + b(C + (C+1) + (C+2) + \ldots + N) \)
  - \( T(N) = O(N^2) \)
- Fortunately, average case performance is \( O(N \log N) \) (see text for proof)
Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call (O(log n) space).
- O(n log n) average case performance, but O(n^2) worst case performance.

Folklore

- “Quicksort is the best in-memory sorting algorithm.”
- Truth
  - Quicksort uses very few comparisons on average.
  - Quicksort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality