“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 → known as **Mergesort**
- **Idea 2**: Partition array into small items and large items, then recursively sort the two sets → known as **Quicksort**

### Mergesort

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

### Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 | 1 3 5 6
```

```
| Auxiliary array |
```

```
2 4 8 9 | 1 3 5 6
```

```
| Auxiliary array |
```

```
1 | 2 | 3 | 4 | 5 |
```

```
| Auxiliary array |
```

```
Mergesort(A[], T[], 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);
}
```
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
    - 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
  - If the number of elements in S is 0 or 1, then return. The array is sorted.
  - Pick an element v in S. This is the pivot value.
  - Partition S-\{v\} into two disjoint subsets, \( S_1 = \{ \text{all values } x < v \} \) and \( S_2 = \{ \text{all values } x > v \} \).
  - Return QuickSort\((S_1)\), v, QuickSort\((S_2)\)

The steps of QuickSort

QuickSort Example

- Choose the pivot as the median of three.
- Place the pivot and the largest at the right and the smallest at the left

Recurring Student Activity

Merge Sort: Complexity

Merge Sort

31 16 54 4 2 17 6

Weiss
QuickSort Example

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 complexity

Quicksort: Worst case complexity

Recurring Student Activity

Quick Sort  31  16  54  4  2  17  6

QuickSort Example

QuickSort Example
QuickSort:
Average case complexity

Turns out to be $O(n \log n)$

See Section 7.7.5 for an idea of the proof.
Don’t need to know proof details for this course.

Features of Sorting Algorithms

- In-place
  - Sorted items occupy the same space as the original items. (No copying required, only $O(1)$ extra space if any.)
- Stable
  - Items in input with the same value end up in the same order as when they began.

Features of Sorting Algorithms

- In-place
  - Sorted items occupy the same space as the original items. (No copying required, only $O(1)$ extra space if any.)
- Stable
  - Items in input with the same value end up in the same order as when they began.

Sort Properties

Are the following: stable? in-place?

- Insertion Sort? No Yes Can Be No Yes
- Selection Sort? No Yes Can Be No Yes
- MergeSort? No Yes Can Be No Yes
- QuickSort? No Yes Can Be No Yes

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in $O(N \log N)$ best case running time
- Can we do any better?
- No, if the basic action is a comparison.

Sorting Model

- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)

Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = $3 \cdot 2 \cdot 1 = 3!$ (ie, “3 factorial”)
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …. (2) choices, 1 choice
  - $N(N-1)(N-2)…2(1) = N!$ possible orderings
**Decision Tree**

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
  - i.e., the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

**Decision Trees and Sorting**

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
  - i.e., by making comparisons
  - Each decision reduces the possible solution space by one half
  - Run time is $\geq$ maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

**Lower bound on Height**

- A binary tree of height $h$ has at most how many leaves?
- A binary tree with $L$ leaves has height at least:
- The decision tree has how many leaves:
- So the decision tree has height:

**log(N!) is $\Omega(N \log N)$**

$log(N!) = log(N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 1)$

$= log N + log(N-1) + log(N-2) + \ldots + log 2 + log 1$

$\geq log N + log(N-1) + log(N-2) + \ldots + \log \frac{N}{2}$

$= \frac{N}{2} log \frac{N}{2}$

$\geq \frac{N}{2} log N - \frac{N}{2} - \frac{N}{2} log N - \frac{N}{2}$

$= \Omega(N \log N)$

**$\Omega(N \log N)$**

- Run time of any comparison-based sorting algorithm is $\Omega(N \log N)$
- Can we do better if we don’t use comparisons?
BucketSort (aka BinSort)
If all values to be sorted are known to be between 1 and $K$, create an array count of size $K$, increment counts while traversing the input, and finally output the result.

Example $K=5$. Input = (5,1,3,4,3,2,1,1,5,4,5)

BucketSort Complexity: $O(n+K)$
- Case 1: $K$ is a constant
  - BinSort is linear time
- Case 2: $K$ is variable
  - Not simply linear time
- Case 3: $K$ is constant but large (e.g. $2^{32}$)
  - ???

Fixing impracticality: RadixSort
- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything
- Idea: BucketSort on each digit, least significant to most significant (lsd to msd)

Radix Sort Example (1st pass)
Input data
478
537
9
721
3
123
38
67
123
537
478
38
67
9

Bucket sort by 1’s digit

After 1st pass

Radix Sort Example (2nd pass)
After 1st pass

Bucket sort by 10’s digit

After 2nd pass

Radix Sort Example (3rd pass)
After 2nd pass

Bucket sort by 100’s digit

After 3rd pass

Invariant: after $k$ passes the low order $k$ digits are sorted.
RadixSort

- Input: 126, 328, 636, 341, 416, 131, 328

BucketSort on LSD:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

BucketSort on next-higher digit:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

BucketSort on MSD:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Student Activity:

Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?

In practice

- RadixSort only good for large number of elements with relatively small values
- Hard on the cache compared to MergeSort/QuickSort

Internal versus External Sorting

- So far assumed that accessing A[i] is fast – Array A is stored in internal memory (RAM)
  - Algorithms so far are good for internal sorting
- What if A is so large that it doesn’t fit in internal memory?
  - Data on disk or tape
  - Delay in accessing A[i] – e.g. need to spin disk and move head

Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
- External sorting – Basic Idea:
  - Load chunk of data into RAM, sort, store this “run” on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
  - Text gives some examples

Summary of Sorting

- Sorting choices:
  - O(N) – BubbleSort, Insertion Sort
  - O(N log N) average case running time:
    - Heapsort: In-place, not stable
    - Mergesort: O(N) extra space, stable.