CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 13: Sorting I

Announcements

Finishing up hashing

- Rehashing without recomputing hash function
- Good hash functions
  - Efficient
  - Handle multiple word input
- Bad case for hashing
- Cryptographic Hash Functions
- Expected performance

Java implementation of Hashing

- Chained hash table
- Initial size is 64
- Double hash table size when λ = ¾
- Hash buckets implemented at Lists – but are converted to red-black trees at size 8
  - Guard against bad data (so O(log n))

https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java

Messing with a hash table

- Find a large number of keys that hash to same value
- For a hash function H, find x, such that H(x) = z
- H(x) = (ax + b) mod p

  $z \equiv ax + b \pmod{p} \Rightarrow a^{-1}z - b \equiv x \pmod{p}$

- If we are hashing with to H(x) mod 2^k, we find values where

  $H(x) = 0, 2^k, 2*2^k, 3*2^k, \ldots$

Cryptographic Hash Functions

- Hash functions that are hard to invert, e.g., given z, it is hard to find an x, such that h(x) = z
  - Examples, MD5, SHA-1, SHA-2, SHA-3, . . .
- Cryptographic Hash Functions are expensive to compute, so NOT appropriate for data structures
- Standard use case, store a file of passwords

Expected performance

- Worst case, everything goes in one bucket
- Load factor λ, expected number of items per bucket is λ
- Analysis, hashing N items into a table of size N, assume the hashing is random and independent
- Prob(H(X) = Y) = 1/N
- What is the probability that a particular bucket has j items?

The math: Balls in Bins

- Probability that a bin is empty is $(1 – 1/n)^n$
- Probability that a bin has one element is almost $(1– 1/n)^n$
- Approximated by a poisson process
- Expected length of the longest chain is O(log n / loglog n)

Sorting

Sorting

- Input
  - an array A of data records
  - a key value in each data record
  - a comparison function which imposes a consistent ordering on the keys
- Output
  - "sorted" array A such that
    - For any i and j, if i < j then $A[i] \le A[j]$

Space

- How much space does the sorting algorithm require?
  - In-place: no more than the array or at most O(1) addition space
  - Out-of-place: use separate data structures, copy back
  - External memory sorting: data so large that does not fit in memory

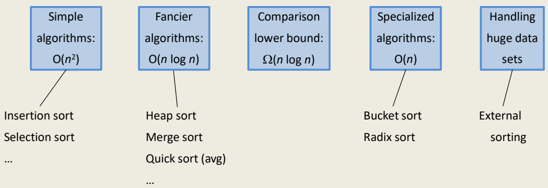
Time

How fast is the algorithm?

- requirement: for any i<j, $A[i] \le A[j]$
- This means that you need to at least check on each element at the very minimum
  - Complexity is at least:
- And you could end up checking each element against every other element
  - Complexity could be as bad as:

The big question: How close to $O(n)$ can you get?

Sorting: *The Big Picture*



Insertion Sort

1. Sort first 2 elements.
2. Insert 3rd element in order.
   (First 3 elements are now sorted.)
3. Insert 4th element in order
   (First 4 elements are now sorted.)
4. And so on…

How to do the insertion?

Suppose my sequence is:

16, 31, 54, 78, 32, 17, 6

And I've already sorted up to 78. How to insert 32?

Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

Insertion Sort code

```
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
                Swap(a[j],a[j-1])
            else
                break
     }

}
```

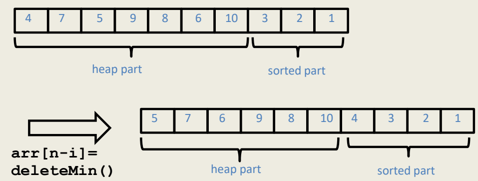

Insertion Sort

- Worst case O(n^2)
- The runtime is related to how sorted the data is
  - Run time proportional to number of pairs of out of order items
- Insertion sort is useful when there is a small number of items to sort

Heap Sort: Sort with a Binary Heap

In-place heap sort

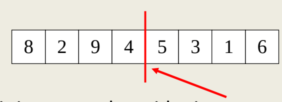
- Treat the initial array as a heap (via buildHeap)
- When you delete the i^{th} element, put it at arr[n-i]
  - It's not part of the heap anymore!



"Divide and Conquer"

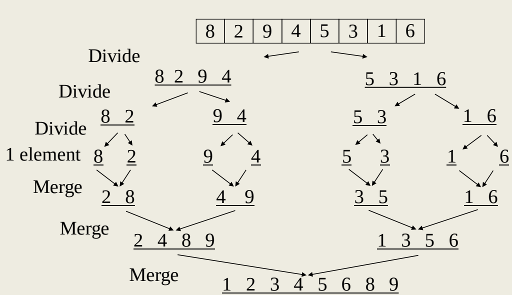
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array in half, *recursively* sort left and right halves, then *merge* two halves
  → known as Mergesort
- **Idea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

Mergesort



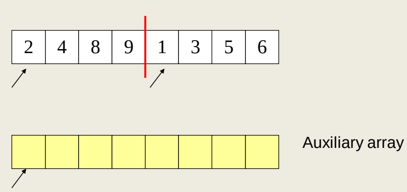
- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together

Mergesort Example



Merging: Two Pointer Method

Merge using an auxiliary array



Merging

```
Merge(A[], Temp[], left, mid, right)  {
  int i, j, k, l, target
  i = left
  j = mid + 1
  target = left
  while (i ≤ mid && j ≤ right) {
    if (A[i] ≤ A[j])
      Temp[target] = A[i++]
    else
      Temp[target] = A[j++]
    target++
  }
  if (i > mid) //left completed
    for (k = left to target-1)
      A[k] = Temp[k];
  if (j > right) //right completed
    k = mid
    l = right
    while (k ≥ i)
      A[l--] = A[k--]
    for (k = left to target-1)
      A[k] = Temp[k]
}
```


Recursive Mergesort

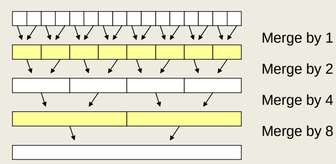
```
MainMergesort(A[1..n], n) {
  Array Temp[1..n]
  Mergesort[A, Temp, 1, n]
}

Mergesort(A[], Temp[], left, right) {
  if (left < right) {
    mid = (left + right)/2
    Mergesort(A, Temp, left, mid)
    Mergesort(A, Temp, mid+1, right)
    Merge(A, Temp, left, mid, right)
  }
}
```

What is the recurrence relation?

Mergesort: Complexity

Iterative Mergesort



Properties of Mergesort

- In-place?
- Sorted list complexity?
- Nicely extends to handle linked lists.
- Multi-way merge is basis of big data sorting.
- Java uses Mergesort on Collections and on Arrays of Objects.

Quicksort

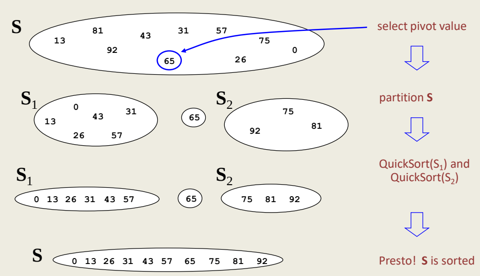
Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does.

Here's the idea for sorting array **S**:

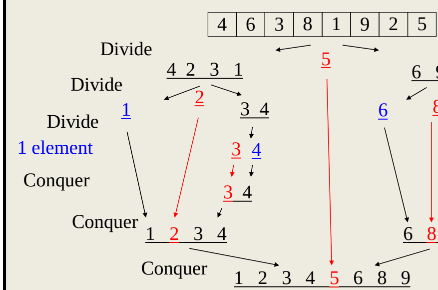
1. Pick an element *v* in **S**. This is the ***pivot*** value.
2. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ and $\mathbf{S}_2$ such that:
   - elements in $\mathbf{S}_1$ are all $\leq v$
   - elements in $\mathbf{S}_2$ are all $\geq v$
3. Return concatenation of QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Recursion ends if Quicksort( ) receives an array of length 0 or 1.

The steps of Quicksort



Quicksort Example



Pivot Picking and Partitioning

The tricky parts are:

- **Picking the pivot**
  - Goal: pick a pivot value so that $|S_1|$ and $|S_2|$ are roughly equal in size.
- **Partitioning**
  - Preferably in-place
  - Dealing with duplicates

Picking the pivot

- Choose the first element in the subarray
- Choose a value that might be close to the middle
  - Median of three
- Choose a random element

Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
  - elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- Can be done in-place with another "two pointer method"
  - Sounds like mergesort, but here we are *partitioning*, not sorting…
  - …and we can do it in-place.
- Lots of work has been invested in engineering quicksort

Quicksort Pseudocode

Putting the pieces together:

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```

Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```
Quicksort(A[], left, right) {
  if (right – left ≥ CUTOFF) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);

  } else {
    InsertionSort(A, left, right);
  }
}
```

CUTOFF = 16 is reasonable.

Quicksort run time

- What is the best case behavior?

Worst case run time

- What is the bad case for partitioning?
- Design a bad case input (assume first element is chosen as pivot)

Average case performance

- Assume all permutations of the data are equally likely
  - Or equivalently, a random pivot is chosen
- The math gets messy, but doable

Properties of Quicksort

- O(N^2) worst case performance, but O(N log N) average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- “In-place,” but uses auxiliary storage because of recursive calls.
- Used by Java for sorting arrays of primitive types.