Data Structures: Lecture 13

Herbert H. Chang, Ph.D.
張恆華
Eng Sci Ocean Eng
National Taiwan University

Quick Sort, Disjoint Sets, and Selection

Quicksort

- Quicksort is a recursive divide-and-conquer algorithm, like mergesort.
- Quicksort is the fastest known comparison-based sort for arrays, even though it has a $\Theta(n^2)$ worst-case time.
- If properly designed, it virtually always runs in O(n log n) in practice.
- Given an unsorted list I of items, quicksort chooses a "pivot" item v from I, then puts each item of I into one of two unsorted lists, depending on whether its key is less or greater than v's key.

Algorithm

Start with unsorted list I of n items

Choose pivot item v from I

Partition I into 2 unsorted lists I1 & I2

- I1: All keys smaller than v's key.
- I2: All keys larger than v's key.
- Items with same key as v can go into either list.
- The pivot does not go into either list.

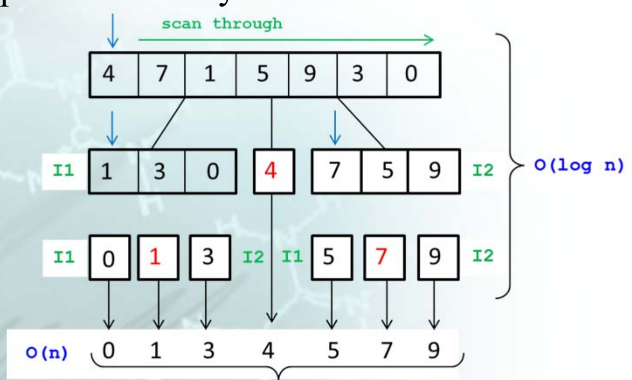
Sort I1 recursively, yielding sorted list S1

Sort I2 recursively, yielding sorted list S2

Concatenate S1, v, & S2, yielding sorted list S

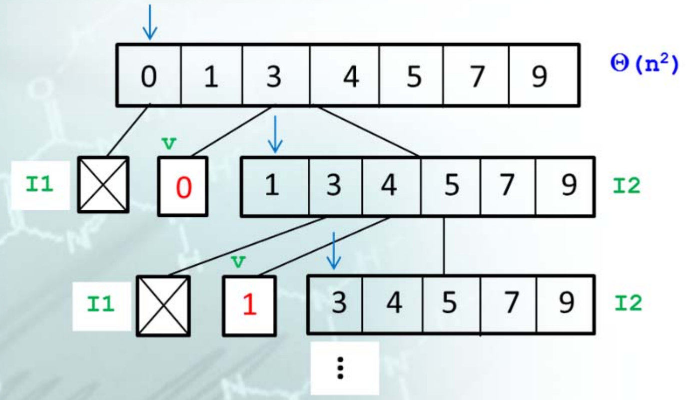
Example

- The pivot v is always chosen to be the first item.



- We get lucky: pivot always be the item having the median key.
- Running time: O(n log n).

Example: already sorted list



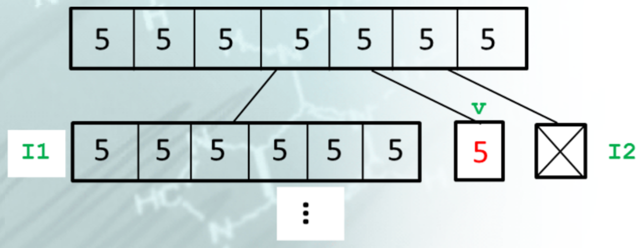
- When the input list I happens to be already sorted, choosing the pivot to be the first item of the list is a disastrous policy.

Choosing a pivot

- We need a better way to choose a pivot.
- Randomly select an item from I to serve as pivot.
- With a random pivot, we can expect "on average" to obtain a ¼ ~ ¾ split.
- The average running time with random pivots is O(n log n).
- An even better way to choose a pivot (when n is larger than 50 or so) is called the "median-of-three" strategy.
- "Median-of-three": select 3 random items from I, then choose the middle key among them.

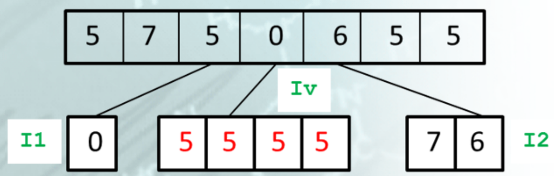
Quicksort on linked lists

- Suppose we put all items with same key as v into I1.
- If we try to sort a list in which every single item has the same key, then ”every” item will go into list I1, and quicksort will have quadratic running time!.



Far better solution

- When sorting a linked list, a far better solution is to partition I into 3 unsorted lists: I1, I2, & Iv. Iv contains pivot & all items with same key as v.
- Sort I1 & I2 recursively, yielding S1 and S2; not Iv.
- Finally, concatenate S1, Iv, S2 to yield S.



- Unfortunately, with linked lists, selecting a pivot is annoying.

Quicksort on arrays

- In-place quicksort is very fast. But a fast in-place quicksort is tricky to code.
- Suppose we have an array **a**.
- We want to sort the items starting at **a**[low] and ending at **a**[high].
- We choose a pivot v; swap it with the last item, **a**[high].

3	8	0	9	5	7	4
low				v		high

3	8	0	9	4	7	5

i j

Quicksort on arrays

- We employ two array indices, i and j. Index i is initially "low - 1", and index j is initially "high“.
- Indices i & j sandwich items to sort (not including pivot).
- Invariants:
  - All items at or left of index i have a key ≤ pivot’s key.
  - All items at or right of index j have a key ≥ pivot’s key.
- Repeatedly advance i to key ≥ pivot.
- Repeatedly decrement j to key ≤ pivot.

3	8	0	9	4	7	5
	i			j		

Quicksort on arrays

- Then, swap items at i & j.

3	4	0	9	8	7	5
	i			j		

- We repeat this sequence until the indices i and j meet in the middle, $i \geq j$.

3	4	0	9	8	7	5
		j	i			

- Then, we move the pivot back into the middle by swapping the pivot with the item at index i.

3	4	0	5	8	7	9
	I1				I2	

Quicksort on arrays

- How about items with same key as pivot?
  → Handling these is particularly tricky.
- We'd like to put them on a separate list (as we did for linked lists) → in-place is too complicated.

⇨Make sure each index i & j, stops whenever it reaches a key equal to the pivot.

- Every key equal to the pivot takes part in one swap.
- Swapping an item equal to the pivot may seem unnecessary, but it has an excellent side effect: if all the items in the array have the same key, half these items will go into I1, and half into I2, giving us a well-balanced recursion tree.

Pseudocode

```
void quicksort(Comparable[] a, int low, int high) {
// Comparable is some variable type supporting
comparable function
  if(low < high) {
    int pivotIndex = random number from low to high;
     Comparable pivot = a[pivotIndex];
     a[pivotIndex] = a[high];
     // swap pivot with last item
     a[high] = pivot;
     int i = low – 1;
     int j = high;
```


Pseudocode

```
    do {
      do{i++;} while (a[i].compareTo(pivot)<0);
      do{j--;} while ((a[j].compareTo(pivot)>0)
        && (j>low));
      if(i < j) {
        swap a[i] and a[j];
      }
    } while (i < j);
```

Pseudocode

```
    a[high] = a[i];
    a[i] = pivot;
    // put pivot in middle
    quicksort(a, low, i-1);
    // recursively sort left list
    quicksort(a, i+1, high);
    // recursively sort right list
  }
}
```

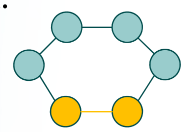

Disjoint Sets

- A disjoint sets data structure represents a collection of sets that are disjoint.
- No item is in more than one set.
- A collection of disjoint sets is called a <u>partition</u>.
- <u>Universe</u> of items: all of the items that can be a member of a set. Each item is a member of exactly one set.

Two operations

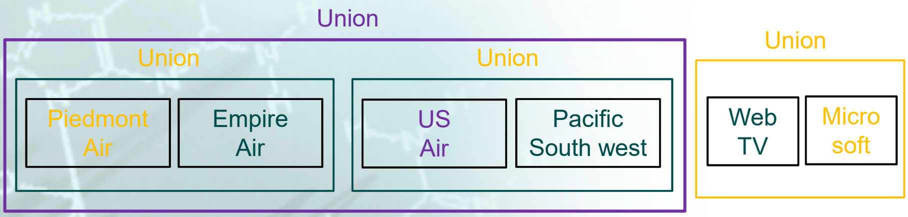
- Union: merges two sets into one.
- Find: takes an item and tells us what set it is in.

- Data structure designed to support these operations are called partition or union/find data structures.
- Applications: Kruskal's algorithm for computing the minimum spanning tree of a graph.



Example: corporations

- Union/find data structures begin with every item in a separate set.
- The query "find(Empire Air)" returns "Empire Air".



- After 3 unions: "find(Empire Air)" returns "Piedmont Air".
- After the final union: "find(Empire Air)" returns "US Air".

List-Based Disjoint Sets

- Each set references a lists of the items in that set.
- Each item references the set that contains it.

- With this data structure, Find operations take O(1) time.
- Union operations are slow.

- List-based disjoint sets use the quick-find algorithm.

Tree-Based Disjoint Sets

- Union operations take O(1) time.
- Find operations are slower.

- Quick-union is faster overall than quick-find.
- To support fast unions, each set is maintained as a tree.
- The quick-union data structure comprises a forest (a collection of trees).

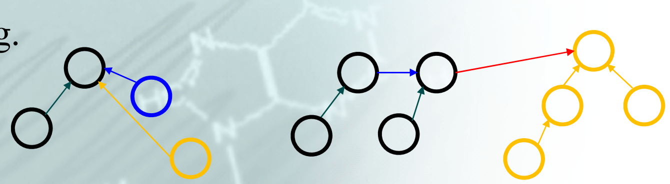
The Quick-Union Algorithm

- Each item is initially the root of its own tree.
- No child or sibling references; only parent.
- The true identity of each set is recorded at its root.
- Union: make the root of one set be a child of the root of the other set.

- Find: follow the chain of parent references from an item to the root of its tree.

Union-by-size

- The cost of Find operation is proportional to the item’s depth in the tree.
- Union-by-size: keep items from getting too deep by uniting sets intelligently.
- At each root, we record the size of its tree.
- When we unite two trees, we make the smaller one a subtree of the larger one, breaking ties arbitrarily.
- e.g.



Implementing Quick-Union with an Array

- Suppose the items are non-negative integers, numbered from zero.
- We'll use an array to record the parent of each item.
- If an item has no parent, we'll record the size of its tree, record the size *s* as the negative number -*s*.
- Initially, every item is the root of its own tree, so we set every array element to -1.

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9

Example

- The forest illustrated below is represented by the array beneath it.

1	-4	-1	8	5	8	1	3	-5	1
0	1	2	3	4	5	6	7	8	9

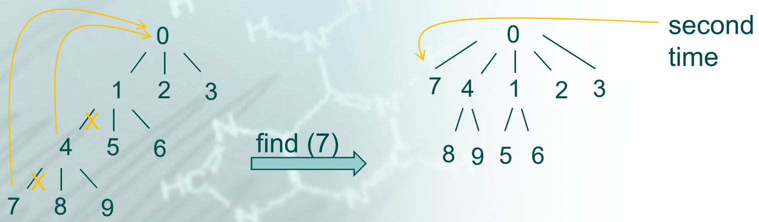
Quick-Union-by-Size

- Let root1 and root2 be two items that are roots of their respective trees.

```
void union (int root1, int root2) {
  if (array[root2]<array[root1]) {
  // root2 has larger tree
      array[root2] += array[root1];
      // update # of items in root2's tree
      array[root1] = root2;
      // make root2 parent of root1
  } else { // root1 has equal or larger tree
      array[root1] += array[root2];
      array[root2] = root1;
  }
}
```


Path compression

- Suppose a tall tree, and we perform find() repeatedly on its deepest leaf.
- find( ): we walk up the tree from leaf to root, move every node we encounter up the tree so that it becomes a child of the root.



- This technique is called "path compression"

Find with path compression

- Let x be an item whose set we wish to identify:
- Here is code for find, which returns the identity of the item at the root of the tree.

```
int find(int x) {
    if (array[x] < 0) { // x is root of tree; return it
        return x;
    } else {
// find out root; compress path by making root x's parent
        array[x] = find(array[x]);
        return array[x];      // return the root
    }
}
```

27

H. Chang © NTU

Running Time

- Union: $\Theta(1)$ time
- Find: $\Theta(\log u)$ worst-case time, where u is number of unions prior to the find.
- Average running time close to constant thanks to path compression.
- Bottom line: a sequence of f find and u union operations takes $\Theta(u + f\alpha(f + u, u))$ time worst case, where α is an extremely slow-growing "inverse Ackermann function", never > 4 for any values of f & u you'll ever use.

28

H. Chang © NTU

Selection

- Suppose that we want to find the k*th* smallest key in a list.
- In other words, we want to know which item has index j, where j = k – 1, if the list is sorted.
- But if we don't actually need to sort the list, is there a faster way?
- This problem is called "selection".
- Example: Median of a set of n keys → the item whose index is j = (n-1)/2 in the sorted list, where n is odd.

29

Quickselect

- We can modify quicksort to perform selection.
- Observe that when we choose a pivot v and use it to partition the list into three lists I1, Iv, and I2, we know which of the three lists contains index j, because we know the lengths of I1 and I2.
- Therefore, we only need to search one of the three lists.

30

Quickselect algorithm

- Here's the quickselect algorithm for finding the item at index j – i.e., having the (j+1)th smallest key.

```
Start with an unsorted list I of n items.
Choose a pivot v from I.
Partition I into three lists I1, Iv, 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.`
- `Iv contains the pivot v.`
- `Items with the same key as V go into any list. (List-based: Iv; array-based: I1&I2).`

Quickselect algorithm

```
if (j < |I1|) {
    Recursively find the item with index j in
    I1; return it.
} else if (j < |I1|+|Iv|) {
    return the pivot v.
} else { // j >= |I1| + |Iv|
    Recursively find the item with index j-
    |I1|-|Iv| in I2; return it.
}
```

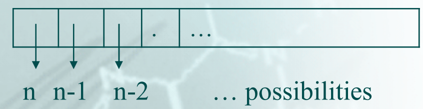
Quickselect

- The advantage of quickselect over quicksort is that we only have to make one recursive call, instead of two.
- Quickselect is much faster than quicksort, it runs in $\Theta(n)$ average time if we select pivots randomly.
- We can easily modify the code for quicksort on arrays to do selection.
- Recall that when the partition stage finishes, the pivot is stored at index "i“. In the quickselect pseudocode, just replace |I1| with i and |Iv| with 1.

Lower Bound on Comparison-Based Sorting

- Suppose we have a scrambled array of n numbers with each from 1….n occurring once.
- How many possible orders can they be in?
- The answer is n!

$$n! = 1 \times 2 \times 3 \times \ldots\ldots \times (n-2) \times (n-1) \times n$$



- Each order is a permutation of the numbers, and there are n! possible permutations.

Lower Bound Analysis

- Observe that if $n > 0$:

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n \leq \overbrace{n \times n \times \cdots \times n \times n}^{n \text{ times}} = n^n$$

and (supposing n is even)

$$n! \geq \overbrace{\frac{n}{2} \times \left(\frac{n}{2}+1\right) \times \cdots \times (n-1) \times n}^{n/2 \text{ times}}$$

$$\geq \underbrace{\frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times \frac{n}{2}}_{n/2 \text{ times}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

So $n!$ is between $\left(\frac{n}{2}\right)^{\frac{n}{2}}$ and n^n.

$\log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} log \frac{n}{2} \in \Theta(n \log n)$ and $\log(n)^n = n \log n$

Hence, $\log(n!) \in \Theta(n \log n)$.

35 H. Chang © NTU

Comparison-Based Sort

- A comparison-based sort is one in which all decisions are based on comparing keys (using "if" statements).
- A correct sorting algorithm must generate a <u>different</u> sequence of true/false answers for each different permutation of $1 \ldots n$. (Because it takes a different sequence of data moments to sort each permutation.)
- There are $n!$ different permutations → $n!$ different sequences of true/false answers.

36 H. Chang © NTU

Comparison-Based Sort

- If a sorting algorithm asks $\leq d$ true/false questions, it generates $\leq 2^d$ different sequences of true/false answers.
- If it correctly sorts every permutation, then $n! \leq 2^d$, so $\log(n!) \leq d$ and $d \epsilon \, \Omega(n \log n)$.
- Algorithm spends $\Theta(d)$ time asking d sequences.

EVERY comparison-based sorting algorithm takes $\Omega(n \log n)$ worst-case time.

- Fast algorithms make q-way decisions for large q, instead of true/false (2-way) decisions.
- Some of these algorithms run in linear time.