CS61B Week12 Sorting

Lec33 Sorting I

Part1. The Sorting Problem

1. Defination of the sort

   A sort is a permutation (re-arrangement) of a sequence of elements that brings them into order according to some ***total order***. A total order $\leqslant$ is:

   - Total: $x \leqslant y$ or $y \leqslant x$ for all x, y
   - Reflexive: $x \leqslant x$
   - Antisymmetric: $x \leqslant y$ and $y \leqslant x$ iff x = y (x and y are equivalent).
   - Transitive: $x \leqslant y$ and $y \leqslant z$ implies $x \leqslant z$.

2. *Another version:*

   We call a pair of elements that are out of order as an inversion. The sort process is a process to reduce inversion to 0.

Part2. Selection Sort, Heap Sort

1. Selection Sort

   i. Selecting sorting N items:

   - **Find the smallest item in the unsorted portion of the array**
   - **Move it to the end of the sorted portion of the array.**
   - **Selection sort the remaining unsorted items.**

   ii. Time Complexity:
   $\Theta(N^2)$

2. Improvement of Selection sort:**Heap Sort**

   Heap is a great data structure to find the minimum element.

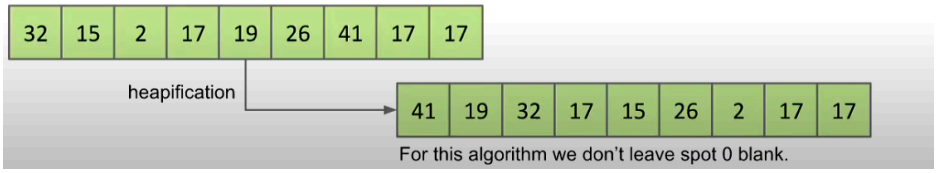
   i. Naive heapsorting N items:

   - Insert all items into a max heap, and discard input array.Create output array,
   - Repeat N times:
     - Delete largest item from the max heap
     - Put largest item from the max heap.
     - Put largest item at the end of the unused part of the output array.
   - Time Complexity of Naive Heap Sort:$O(NlogN)$:
     - Getting items into the heap $O(NlogN)$ time.
     - Selecting largest item:$O(N)$
     - Removing largest item:$O(NlogN)$
   - Space Complexity of naive Heap Sort:
     $\Theta(N)$ to build the additional copy of all our data.

   ii. In-place Heapsort

   Treat input array as an heap

- Rather than inserting into a new array of length N + 1, use a process known as "bottom-up heapification" to convert the array into a heap.
  - To bottom-up heapify, just sink nodes in reverse level order.
- Avoids need for extra copy of all data.
- Once heapified, algorithm is almost the same as naive heap sort.



Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

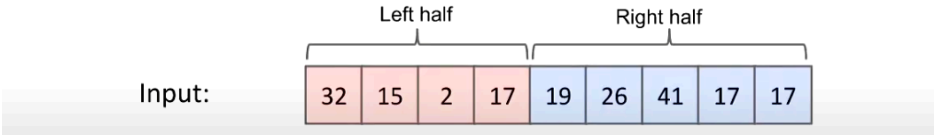
- Time Complexity of In-place Heap Sort:$O(NlogN)$:
  - Bottom-up Heapification:$O(NlogN)$
  - Selecting largest item:$O(N)$
  - Removing largest item:$O(NlogN)$
- Space Complexity:$O(1)$

  The only extra memory we need is a constant number instance variables, e.g. size.
  - Unimportant caveat: If we employ recursion to implement various heap operations, space complexity is Θ(log N) due to need to track recursive calls. The difference between Θ(log N) and Θ(1) space is effectively nothing.

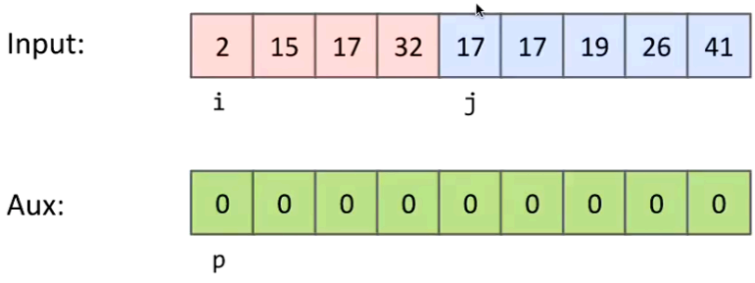
Part3.MergeSort

Top-Down merge sorting N items:

- **Split items into 2 roughly even pieces.**
- Mergesort each half.
- Merge the two sorted halves to form the final result.



  - Compare input[i] < input[j].
  - Copy smaller item and increment p and i or j.



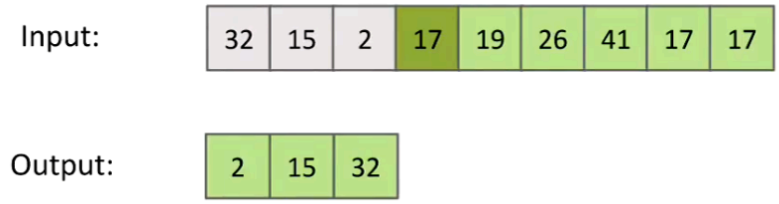
- Time Complexity:$O(NlogN)$
- Space Complexity:$O(N)$(with the aux array)

Part4.Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Time Complexity:$O(N^2)$
Also $\Theta(N+K)$, where K is the number of inversions.

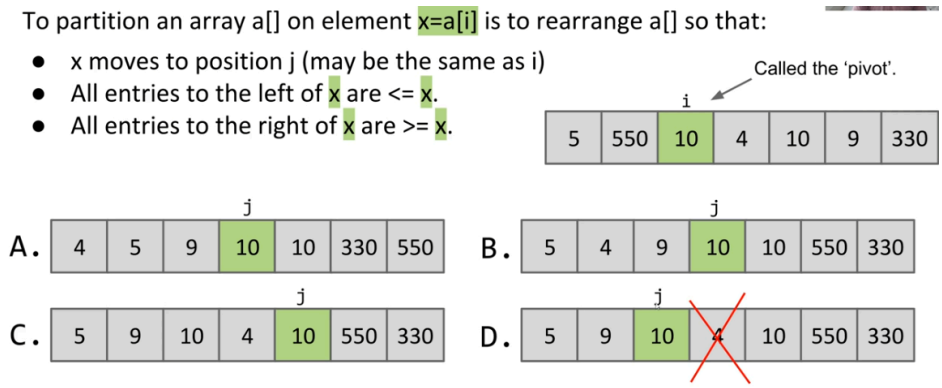
Part5. Shell's Sort(extra)

Lec34 Sorting II Quick Sort

Part 1.Partitioning(分区)

Partitioning isthe core idea of Quick Sort.

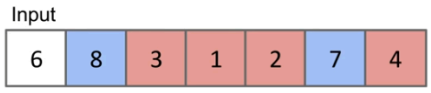
1. To partition an array a[] on element x=a[i] si to rearrange a[] so that:
   - x moves to position j(maybe the same as i)
   - **All entries to the left of x are <=x**
   - **All entries to the right of x are >=x**



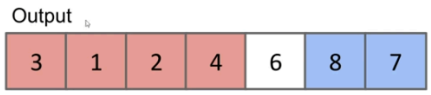
2. How to implement the partitioning?

Given an array of colors where the 0th element is white, and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).

- Relative order of red and blues does NOT need to stay the same.



Algorithm: Create another array. Scan and copy all the red items to the first R spaces. Then scan for and copy the white item. Then scan and copy the blue items to the last B spaces.

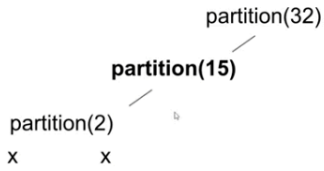


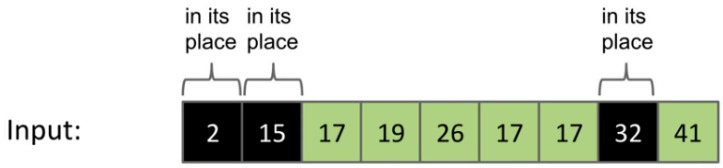
Part 2. Quick Sort

1. basic idea

Quick sorting N items:

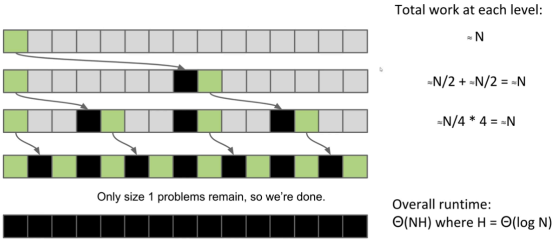
- Partition on leftmost item (2) (done).
- Quicksort left half (done).
- **Quicksort right half (up next).**





2. The Runtime

- Best case: $\Theta(NlogN)$



Total work at each level:

≈ N

≈N/2 + ≈N/2 = ≈N

≈N/4 * 4 = ≈N

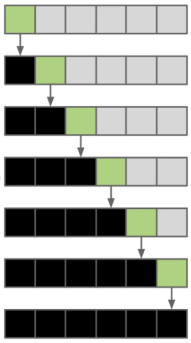
Overall runtime:
Θ(NH) where H = Θ(log N)

so: Θ(N log N)

- Worst case: $\Theta(N^2)$

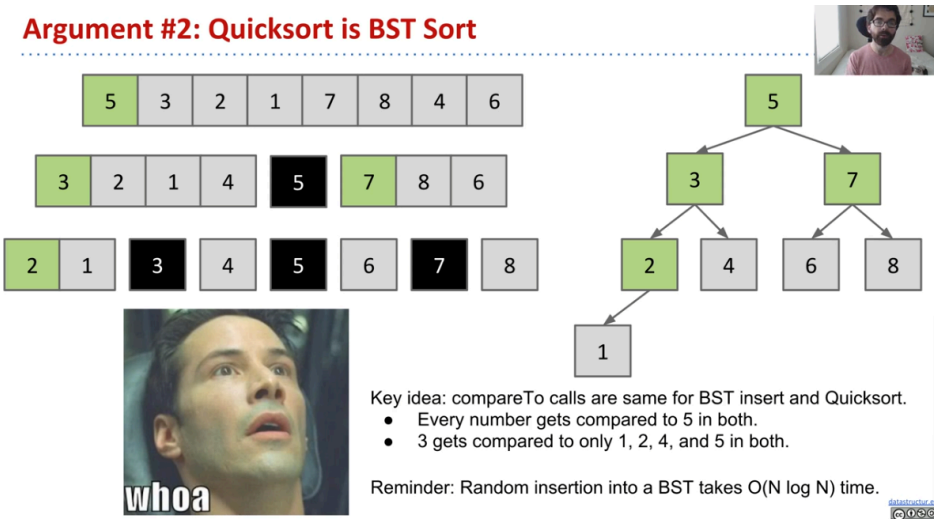
Give an example of an array that would follow the pattern to the right.

- 1 2 3 4 5 6



What is the runtime Θ(·)?

- N^2

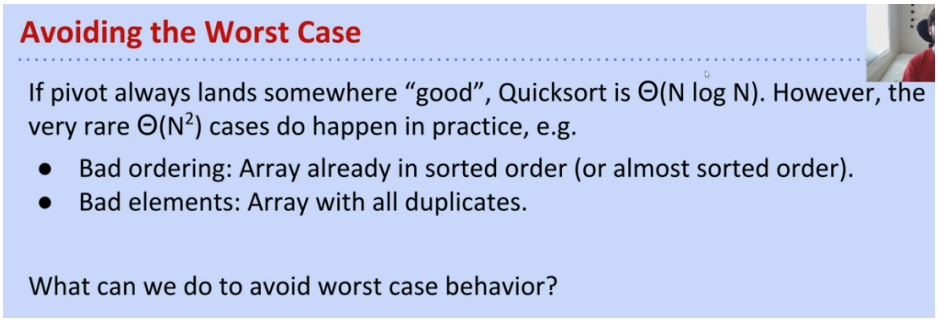


3. Avoid the worst case in the quick sort

   The performance of Quicksort(both of growth and constant factors) depend critically on:

   - How you select your pivot
   - How you partition around that pivot
   - Other optimizations you might add to speed things up

     Bad choices can be very bad indeed, resulting in $\Theta(N^2)$ runtimes.



- There are four philosophies to avoid the worst case in Quick Sort:

  i. Randomness:

  Pick a random pivot or shuffle before sorting.

  - **Strategy 1: Pick pivots randomly**
  - **Strategy 2: Shuffle(打乱) before the sort**

    This strategy doesn't work well with array of duplicate(重复), this will lead to a behavior of $\Theta(N^2)$

  ii. Smarter pivot selection:

  Calculate or approximate the median

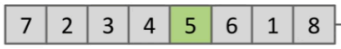
  e.g. Choose the median of the first three elements

Randomness is necessary for best Quicksort performance! For any pivot selection procedure that is:

- Deterministic
- Constant Time

The resulting Quicksort has a family of dangerous inputs that an adversary could easily generate.

- See McIlroy's "A Killer Adversary for Quicksort"



Dangerous input

iii. Introspection:

Switch to a safer sort if recursion goes to deep
Can also simply watch your recursion depth.

- If it exceeds some critical value (say 10 ln N), switch to mergesort.

Perfectly reasonable approach, though not super common in practice.

iv. Preprocess the array:Could analyze array to see if Quicksort will be slow. No obvious way to do this, though(can't just check if array is sorted, almost sorted arrays are almost slow)

Part 3.Sort Summary

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.
  - Next time: Alternate strategy for partitioning, pivot identification.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)
Insertion	Θ(1)	Θ(N^2)	Θ(N) if almost sorted
Mergesort	Θ(N)	Θ(N log N)	
Random Quicksort	Θ(log N) (call stack)	Θ(N log N) expected	Fastest sort

Lec35 Sorting III

Part 1.Quicksort Flavors vs.Mergesort

1. **QuickSortL3S**

   We call this quick sort "QuicksortL3S":

   - Pivot selection:Always use the **left** most
   - Partition algorithm:Make an array copy then do **three** scans for red, white, and blue items(while scan trivally finishes in one compare)
   - **Shuffle** before starting(to avoid worst case)

2. Comparison between QuickSort L3S and Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	2.1 seconds
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds

Quicksort didn't do so well!

3. Hoare Partitioning

Tony originally proposed a scheme where two pointers walk towards each other.

- Left pointer loves small items.
- Right pointer loves large items.
- Big idea: Walk towards each other, swapping anything they don't like.
  - End result is that things on left are "small" and things on the right are "large".

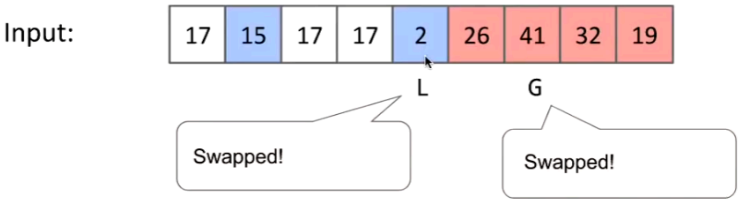
Full details here: Demo

Using this partitioning scheme yields a very fast Quicksort.

- Though faster schemes have been found since.
- Overall runtime still depends crucially on pivot selection strategy!

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - **When both pointers have stopped, swap and move pointers by one**
  - When pointers cross, you are done walking.
- Swap pivot with G.



4. Comparison between QuickSort using Hoare Partitioning and Merge Sort

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	2.1 seconds	$\Theta(N \log N)$
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds	$\Theta(N^2)$
Quicksort PickTH	Exact Median	Tony Hoare	Exact Median	10.0 seconds	$\Theta(N \log N)$

Quicksort using PICK to find the exact median (Quicksort PickTH) is terrible!

- Cost to compute medians is too high.
- Have to live with worst case $\Theta(N^2)$ if we want good practical performance.

Part 2.Selection(Quick Select)(Pick Algorithm)

Goal:Find the median

Part 3. Stability,Adaptiveness, and Optimization

1. Stability

A sort is said to be stable if order of equivalent items is preserved

A sort is said to be stable if order of equivalent items is preserved.

sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Jouni	3
Rosella	3
Bas	3
Jana	3
Fikriyya	4
Nikolaj	4

Sorting instability can be really annoying! Wanted students listed alphabetically by section.

What types of sorts are stable or not stable?

	Memory	# Compares	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)	No
Insertion	Θ(1)	$\Theta(N^2)$	Θ(N) if almost sorted	Yes
Mergesort	Θ(N)	Θ(N log N)		Yes
Quicksort LTHS	Θ(log N)	Θ(N log N) expected	Fastest sort	No

This is due to the cost of tracking recursive calls by the computer, and is also an "expected" amount. The difference between log N and constant memory is trivial.

You can create a stable Quicksort. However, unstable partitioning schemes (like Hoare partitioning) tend to be faster. All reasonable partitioning schemes yield Θ(N log N) expected runtime, but with different constants.

2. Additonal tricks we can play

- Switch to insertion sort:
  - When a subproblem reaches size 15 or lower, use insertion sort.
- Make sort ***adaptive***: Exploit existing order in array (Insertion Sort, SmoothSort, TimSort (*the* sort in Python and Java)).
- Exploit restrictions on set of keys. If number of keys is some constant, e.g. [3, 4, 1, 2, 4, 3, …, 2, 2, 2, 1, 4, 3, 2, 3], can sort faster (see 3-way quicksort -- if you're curious, see: http://goo.gl/3sYnv3).

Part 4.Shuffling(洗牌)