Other Sorting algorithms

Goodrich et al. chapter 10

Sorting algorithms

- Last semester we looked at the bubble sort and the selection sort
  - Showed both were $O(n^2)$ algorithms
- This semester we have already studied the heap sort
  - Showed this to be an $O(n\log n)$ algorithm

- Now we will consider the following, which both use recursive algorithms
  - Merge Sort
  - Quick Sort

- And finally assess the advantages and disadvantages of each algorithm

Merge-Sort

- Based on the divide-and-conquer algorithmic design-pattern using recursion
  - If the input size makes a solution really simple, solve directly
  - Otherwise Divide input data into 2 or more disjoint subsets
  - Recur: Recursively solve the simpler problems associated with the subsets
  - Conquer: take the solutions to the smaller problems, and merge them together into a solution to the original problem

- Using divide-and-conquer for merge-sorting a sequence S
  - If S has zero or 1 elements, return immediately (already sorted)
  - Otherwise divide S into 2 sequences S1 and S2 each containing about half the elements
  - Recursively sort S1 and S2
  - Put back the elements into S by merging the 2 sorted sequences

Merge sort visual

- Diagram of merge-sort tree from Goodrich page 486

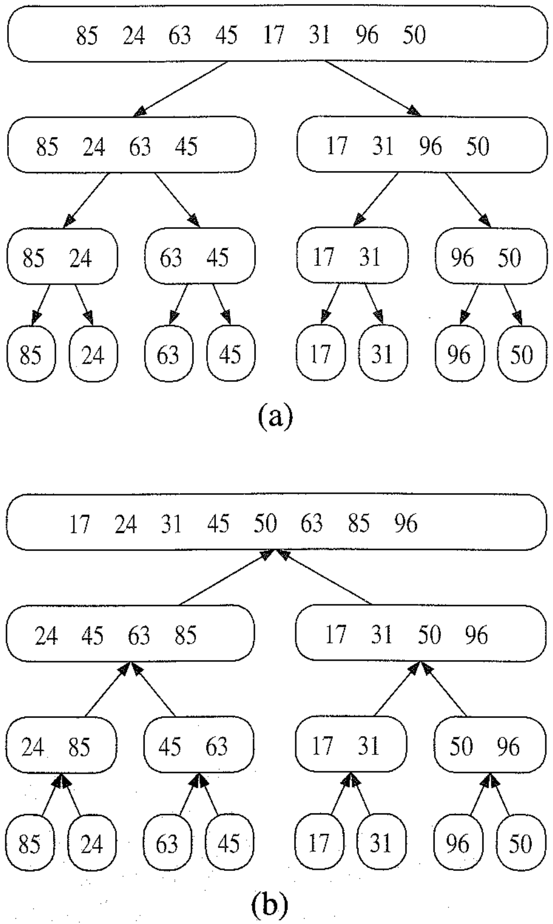


Figure 10.1: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T;

- Since the size of the input sequence roughly halves at each recursive call of merge-sort, the height of the 'merge-sort tree' is about log *n*
- *these trees are of height 3, and $log_2 8$ is 3 ($2^3 = 8$)*

The merge algorithm

- The only piece of work, aside from making recursive calls, is to merge the two sorted sub-sequences into one sorted sequence.
  - We start by getting the first item in each sub-sequence and comparing them, whichever is the smaller is removed to our sorted sequence, and the next item retrieved in its subset
  - This continues until the end of one sub-sequence is reached (1st while loop)
  - The remainder of the other sub-sequence can then just be tagged onto the end of the sorted sequence

Pseudo-code, where the 2 sorted sub-sequences to be merged are s1 and s2. s1next is smallest item left in s1, and s2next is smallest item left in s2

```
Begin
    s1next = 1st item in S1
    s2next = 1st item inS2
    while s1next and s2next both valid
        if s1next < s2next
            add s1next to S
            s1next = next element in S1
        else
            add s2next to S
            s2next = next item in S2
    end while
```

(only one of these while loops will kick in)

```
    while s1next still valid
        add s1next to S
        s1next = next element in s1
    end while

    while s2next still valid
        add s2next to S
        s2next = next element in s2
    end while
End
```

Analysis of merge-sort

- To do a merge of 2 sequences of size n1 and n2 is O(n1 + n2)
  - We can see this because after each comparison, only one item gets removed from the n1+n2 we had to compare …
  - … And if we go 'right to the wire' we will finish up comparing the final items in each sequence, so will have done n1+n2 comparisons (well, minus 1!)
- The 'merge-sort tree' associated with execution of merge-sort on a sequence of size n has height log*n*
  - So there are log*n* merges of n elements
- The total running time of the merge sort algorithm is O(nlog*n*) even in the worst case.

Code for this algorithm

- There are several ways to implement this algorithm. For simplicity, we will sort arrays of integers, and use the following functions

- The first sorts an array of integers from a given start index for a given length.
- It will be required that the given `startIndex` and `lengthToSort` will not overrun the array bounds
  - **`mergeSort(int arrayToSort[], int startIndex, int lengthToSort)`**

- The second merges the contents of an array where it has been sorted in 2 halves (from `startIndex` to `length/2`, and from `startIndex + length/2` until the full sorted length)
  - **`merge(int arraySortedInTwoHalves[], int startIndex, int length)`**

- The first function will use the second in its implementation.

- Exercise 1: write the code for the `mergeSort` function
  - *assume the merge function is supplied*

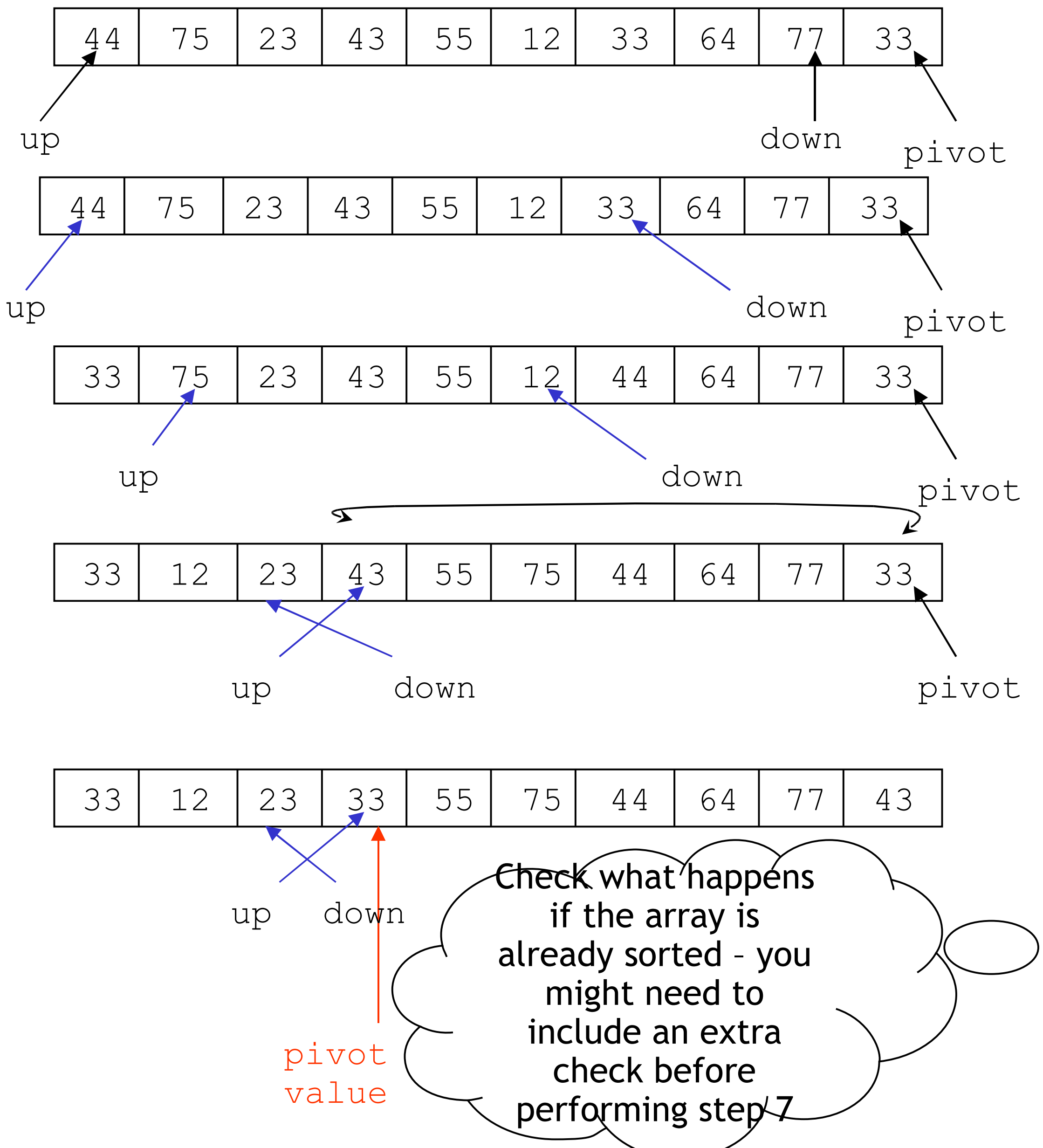
- Exercise 2: write the code for the merge function!
  - *Tip: create(using new) a temporary array to hold the merged result, merge to this array, then copy back to the array to be merged, and delete the temp array)*

Quick-Sort

- Another divide and conquer algorithm, but this time the hard work is done in the dividing step *before* the recursive calls.
- Choose one element from the unsorted sequence (we traditionally make this the last one, but it could be any element)
  - We call this element *the pivot*
- Divide the rest of the elements into 3 subsets:
  - the ones less than the pivot.
  - the ones greater than the pivot.
  - the ones the same as the pivot (if such are allowed in our sequence)
- Recur: recursively sort the sequences less than and greater than the pivot
- Conquer: put the elements into a sorted sequence by adding first the sorted ‘less than’ elements, then the ‘same as’ elements, then the sorted ‘greater than’ elements

  - Division stops if there is only 1 element in the set to sort

An in-place QuickSort partition - demo



1. Decide on the pivot value as last element in array
2. Set a pointer to either end of the rest of the array
3. Move the lower one up until it points to element larger than the pivot
4. Move the higher one down until it points to element less than or equal to the pivot
5. If the lower element (going down pointer) is to the right of the higher (going up pointer), swap them
6. Continue until the going up pointer has passed the going down one
7. Now swap the value at 'going up' pointer with value of pivot
8. And return the new index pos of the pivot

Algorithm for the quicksort partition

```
int partition(table, first, last) //see description below
BEGIN
    Define the pivot value as the contents of table[last]
    define integer up and assign value first
    define integer down and assign value last-1
    do
        while table[up] is less than pivot value, increment up
        while table[down] is greater than or equal to pivot value, decrement down
        if (up < down)
            swap table[up] and table[down]
    while up is to the left of down
    if (up < last)
        swap table[last] and table[up]
        return up
    else
        return last
    end if
END
```

see the bubble thought on last overhead – make sure you know why this check is here

Partitions an array of elements from index pos first to index pos last, so that all items to left are less than or equal to a pivot value, all items to right are greater than pivot value and returns the index position of the pivot value

There will be n-1 comparisons with the pivot, so this step is O(n)

Quick-sort visual

- Diagram from Goodrich et al, page 505

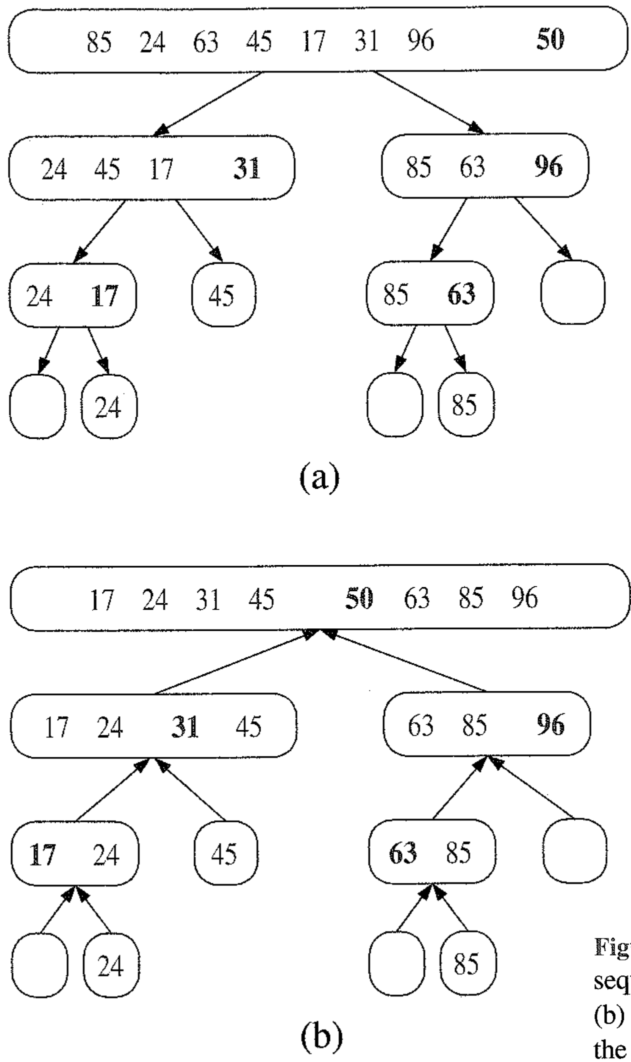
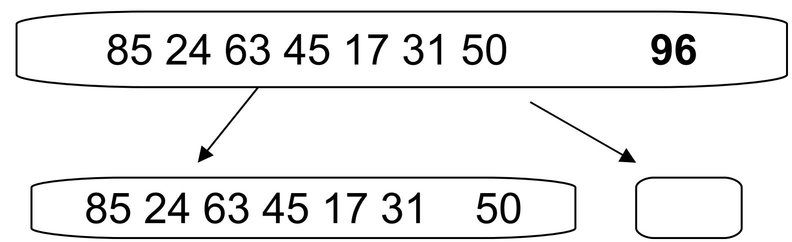


Figure 10.9: Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T. The pivot used at each level of the recursion is shown in bold.

- Showing the height of the quick sort tree is log*n* in the best case
- But if nearly all items landed on the same side of the pivot there would be a sequence of only a slightly reduced size still to sort ..



- .. and if this kept happening, the height of the tree would be close to n, not log*n*

Quick sort analysis

- In best case this will still be O(nlog*n*)
  - at each level of the tree, we make roughly n comparisons to divide around the pivot
  - And the height of the tree in best case is log*n*, so we do it log*n* times
- But in the worst case, it is still $O(n^2)$
  - Because in the worst case, the quick sort tree will be of height n
- We can improve the chances of making it O(nlog*n*) by choosing the pivot at random each time a sequence is sorted, instead of taking the last item
  - A randomised quick sort should always be O(nlog*n*) when n is sufficiently large.
- Quick-sort has an advantage over merge-sort:
  - Merge-sort always needs extra space to hold the result of the merged sequences without trampling on the already-sorted sub-sequences
  - it is possible to perform the quick-sort ‘in place’ with a careful implementation

Comparison of sorting algorithms

- This is a very big and important area
  - There are many more sorting algorithms which may be indicated in various specialised cases
- Comparing the ones we have studied, for sequences that fit entirely into a computers main memory:
- Bubble sort and selection sort are $O(n^2)$
  - Not recommended unless the number of items is guaranteed to be very small
  - Or is guaranteed to be already in a ‘nearly sorted’ format
- Merge-sort is $O(n\log n)$ in worst case
  - But difficult to make it run ‘in place’, which restricts the size of the sequence that can be sorted.
- Quick-sort is $O(n\log n)$ in most cases
  - A good ‘all-rounder’
  - Experimental studies show quick-sort and heap-sort are faster than merge-sort for sequences that fit entirely into main memory.
  - And quick-sort faster than heap-sort on average
  - But it does have that $O(n^2)$ worst case performance
- Heap-sort is $O(n\log n)$ in worst case
  - Can easily execute in place
  - Probably best choice in real-time scenarios with a fixed amount of time