CMPT 280

Topic 23: Efficient Sorting Algorithms

Mark G. Eramian

University of Saskatchewan

References

- Textbook, Chapter 23

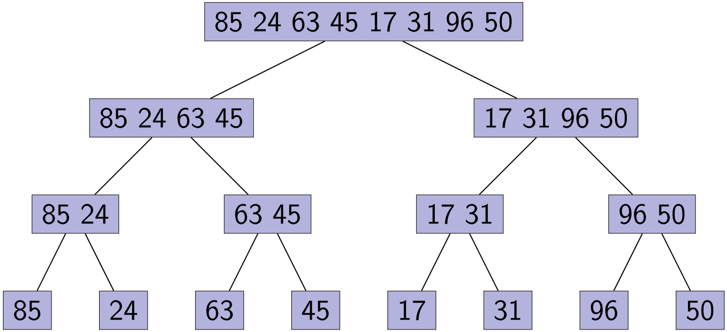
Divide And Conquer Sorts

- Merge Sort and Quick Sort solve the sorting problem using the *divide-and-conquer* approach.
- Basic divide-and-conquer idea:
  - **Divide:** Divide the problem into smaller sub-problems.
  - **Recurse:** Recursively try to solve the sub-problems. When we reach a level where the sub-problem is small enough, solve it quickly.
  - **Conquer:** Take the solutions of the subproblems and combine them into a solution of the original problem.

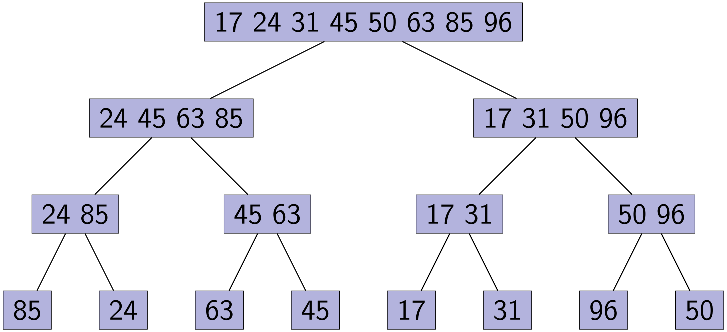
Merge Sort

- We assume that we will be sorting a sequence of elements S with n elements.
- The merge sort uses the following divide and conquer approach:
    - **Divide:** If S has zero or one elements, return S immediately, it is already sorted. Otherwise, divide S into S_1 and S_2 such that S_1 contains the first $\frac{n}{2}$ elements of S and S_2 contains the remaining $\frac{n}{2}$ (rounding up and down respectively).
    - **Recurse:** Recursively sort the sequences S_1 and S_2.
    - **Conquer:** Put elements back into S by merging the sorted sequences S_1 and S_2.
- We can visualize what is happening with a binary tree T called a *merge-sort-tree*.

A Mergesort Tree (Divide Step) – Input Sequences



A Mergesort Tree (Conquer Step) – Output Sequences



Note that the height of the merge sort tree is $\lceil \log_2 n \rceil$.

Exercise 1

This is the array of the previous example, just before the conquer step of the root of the merge sort tree. Each half contains a sorted sequence.

a) How is `merge` called to merge the two sequences?

b) Trace through `merge` step-by-step to show how the merging happens.

0	1	2	3	4	5	6	7
24	45	63	85	17	31	50	96

Exercise 1

```java
/**  Merge start1st through start2nd-1 with start2nd through end2nd in
     array inVec using temp as a temporary array */
private void merge(int[] inVec, int [] temp, int start1st,
                   int start2nd, int end2nd)
{
    int cur1 = start1st;  // index of the current item in first half
    int cur2 = start2nd;  // index of the current item in first half

    int cur3 = 0;   // index of the next location of temp
    // While neither subarray is empty...
    while ((cur1 < start2nd) && (cur2 <= end2nd))
        // Copy the smaller item from the two sequences to temp
        if (inVec[cur1] <= inVec[cur2])
            // Note: increments happen after assignment
            temp[cur3++] = inVec[cur1++];
        else
            temp[cur3++] = inVec[cur2++];

    while (cur1 < start2nd) // copy remainder (if any) of first subarray to temp
        temp[cur3++] = inVec[cur1++];

    while (cur2 <= end2nd) // copy remainder (if any) of second subarray to temp
        temp[cur3++] = inVec[cur2++];

    // Copy the temp array back to inVec[start1st...end2nd]
    cur1 = start1st;
    cur3 = 0;
    while (cur1 <= end2nd) // copy items from temp back into inVec
        inVec[cur1++] = temp[cur3++];
}
```

Analysis of Merge Operation

- Let n_1 and n_2 be the number of elements in S_1 and S_2.
- Assume insertions and deletions from first and last positions of S_1 and S_2 are $O(1)$.
- In each iteration of all three while loops, an element is removed from either S_1 or S_2.
- No elements are ever inserted into S_1 or S_2.
- Number of loop iterations in all three loops must therefore be $n_1 + n_2$.
- Running time is thus $O(n_1 + n_2)$ – linear time!.

Exercise 2

For the following array of elements, draw the merge sort tree showing the input sequences at each recursive call (after the divide step) and the output sequences at each recursive call (after the conquer step)

0	1	2	3	4	5	6	7	8	9	10	11
83	42	5	33	91	62	29	0	73	48	11	12

Analysis of Merge Sort

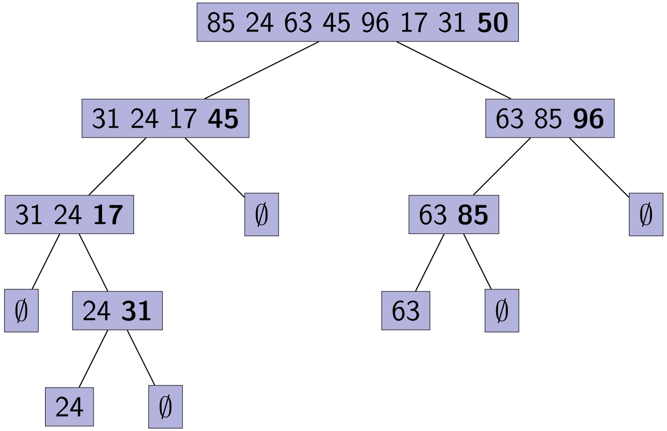
- Analysis refers to a merge-sort-tree T.
- Define *time spent at node* v to be the time taken for the divide step, the conquer step, but not the recurse step.
- Both divide and conquer steps run in at most linear time. If i is the depth of v, then the size of the sequence at v contains at most $\frac{n}{2^i}$ elements, thus time spent at node v is $O(\frac{n}{2^i})$.
- There are at most 2^i nodes at depth i, so time spent at all nodes of depth i is $O(2^i \cdot \frac{n}{2^i}) = O(n)$.
- Since the height of T is $\lceil \log n \rceil$, time spent at all $\lceil \log n \rceil$ levels is $O(n \log n)$.

Quick Sort

- Hard work done **before** recursive calls.
- Idea of quick sort of a sequence S:
  - **Divide**: If S has at least two elements, select some element x from S, called the *pivot*. Remove each element from S and place them in one of three sequences:
    - L (for elements less than x)
    - E (for elements equal to x)
    - G (for elements greater than x)
  - **Recurse:** Recursively sort L and G.
  - **Conquer:**Put S back together by concatenating the (already sorted) elements of L, E, and G.

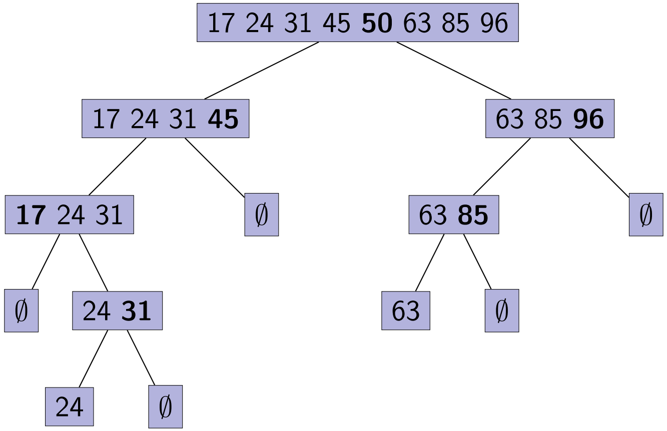
A Quick Sort Tree (Divide Step) – Input Sequences

This quick sort tree shows the input sequences processed by each recursive call. The last element of a sequence is used as the pivot; these are shown in bold. Note: order of child input sequences are affected by parent's partitioning!



A Quick Sort Tree (Conquer Step) – Output Sequences

This quick sort tree shows the output sequences of each recursive call (after the conquer step). Again, pivots are in bold.



Exercise 3

The work of quick sort is the `partition` algorithm that performs the divide step. The following array is the input to the first recursive call to quick sort at the root of the quick sort tree.

a) How is the `partition` algorithm called to partition the array?

b) Trace through the `partition` algorithm to show how the array is partitioned in-place.

0	1	2	3	4	5	6	7
19	42	77	71	41	9	83	68

Exercise 3

```
private int partition(int[ ] inVec, int start, int finish)  {
    int partitionItem = inVec[finish]; // item that will be the pivot
    int l = start-1;  // initial lower offset
    int r = finish; // initial upper offset
    while (l < r)
    {
        l++;
        // move l right, passing "small" items
        while ((l < r) && (inVec[l] <= partitionItem))
            l++;
        r--;
        // move r left, passing "large" items
        while ((l <= r) && (inVec[r] >= partitionItem))
            r--;

        // At this point, r is the offset of a "small" item
        // and l is the offset of a "large" item.
        if (l < r)  // If l and r have *not* yet crossed...
        {
            // swap the large item at offset l with the
            // small item at offset r
            int temp = inVec[l];
            inVec[l] = inVec[r];
            inVec[r] = temp;
        }
    }
    // put item l in the last offset of the subarray, and
    // partitionItem (the pivot) in position l
    inVec[finish] = inVec[l];
    inVec[l] = partitionItem;
    return l;
}
```

Timing Analysis of `partition`

- Worst case: pivot is the median, and the rest of the array is in reverse order.
- Each time through the outer while loop, l moves exactly one position to the right, and r moves exactly one position to the left.
- Swaps are $O(1)$ so each iteration of the while loop is $O(1)$ in the worst case.
- The outer while loop will execute $n/2$ times before $l >= r$.
- Total cost of while loop: $O(n)$
- Initialization and final swap are $O(1)$, so total algorithm cost is $O(n) + O(1)$ or just $O(n)$.

Exercise 4

For the following array of elements, draw the quick sort tree showing the input sequences at each recursive call (after the divide step) and the output sequences at each recursive call (after the conquer step)

0	1	2	3	4	5	6	7	8	9	10	11
83	42	5	33	91	62	29	0	73	48	11	12

Analysis of Quick-Sort

- Let $s(v)$ be the size of the sequence to be sorted at node v.
- The partitioning step is linear time: $O(s(v))$.
- The conquer step (concatenating sequences) is also at most linear: $O(s(v))$
- Thus, the time spent at each node in the quick-sort-tree is at most $O(2s(v))$, or, more simply $O(s(v))$.
- Worst case is when the sequence is **already sorted**!
- What happens to our quick-sort-tree visualization if the array is already sorted?

Analysis of Quick Sort

- Let s_i denote the sum of the input sizes of nodes at depth i. Clearly $s_0 = n$.
- In the worst case, $s_i = n - i$.
- Thus the worst case running time is

$$O\left(\sum_{i=0}^{n-1} s_i\right) \text{ which is } O\left(\sum_{i=0}^{n-1}(n-i)\right) \text{ which is } O\left(\sum_{i=1}^{n} i\right)$$

- We know that $\sum_{i=1}^{n} i$ is $O(n^2)$, thus quick sort runs in $O(n^2)$ in the worst case.
- However, in the best case quick sort runs in $O(n \log n)$ time.
- What is the best case??

Sorting Algorithms

Quick Sort - Remarks

- Quick Sort is not stable.
- In practice, quick sort is often a little faster than merge sort.
- Merge sort, however, gives you a guaranteed worst case time complexity of $O(n \log n)$. Quick sort does not. The worst case is still $O(n^2)$.
- Can we modify quick sort to get a worst case bound of $O(n \log n)$?

Sorting Algorithms

Quick Sort - $O(n \log n)$ guarantee?

- If we always used as a pivot the median of the sequence being partitioned, we would get a balanced quick sort tree and a guarantee of $O(n \log n)$.
- But... the straightforward way to find the median is to sort the items and select the middle one – this does not help!.
- It is possible to find the median of a sequence in as quickly as $O(n)$ time. But this algorithm itself is a modification of quick sort. It would give the asymptotic $O(n \log n)$ guarantee, but it would probably increase the constant so that merge sort was always faster in practice.

Sorting Algorithms

Quick Sort - $O(n \log n)$ guarantee?

- Approximate the median: use the median of the first 3 items as the pivot. Reduces the likelyhood of a bad pivot, but still not a total guarantee.
- Choose the pivot at random: Usually does well, still no perfect guarantee. Also the cost of generating lots of random numbers can be higher than the overall savings obtained by reducing the likelihood of bad partitioning.

Heap Sort

The basics of Heap Sort:

- Heapify the input array.
- Sequentially delete items from the heap, obtaining items in reverse order, moving each one into position as it is obtained. Re-heapify after each deletion as normal.

Exercise 5

Trace through the heapify algorithm showing, step-by-step, how the following array to be sorted is converted into an arrayed heap.

0	1	2	3	4	5	6	7	8	9	10	11
83	42	5	33	91	62	29	0	73	48	11	12

Exercise 5

```
Algoirthm heapify(data)
data - array of elements to convert to a heap

n = data.length
// We can start at i = n/2-1 because larger offsets are guaranteed
// to be leaves by the properties of arrayed heaps.
for i = n/2-1 to 0
     // Make data[i] the root of a valid heap
     // by swapping data[i] with children repeatedly as needed.
     moveDown(data, i, n-1)

Algoirthm moveDown( data, first, last )
data - array of elements to be converted to heap
first - offset of element to process
last - offset of last element in the 'data' array

// select the left child of offset 'first'
largest = 2 * first + 1;
while( largest <= last )
    // if a right child exists and its item is bigger than data[largest]...
    if( largest < last  and  data [largest] < data[largest+1] )
        largest++         // select the right child instead

    // If item at offset 'first' is smaller than its larger child
    // it does not have the heap property, so swap it with its larger child.
    if( data[first] < data[largest] )
        swap( data[first], data[largest] )  // swap it
        first = largest           // follow the item down
        largest = 2 * first + 1   // select its new left child
    else
        // cause loop to end -- data[first] is the root of a valid heap
        largest = last + 1
```

Exercise 6

Beginning with the solution to Exercise 5, show step-by-step how the heap sort is completed by tracing through the heap sort algorithm.

```
Algorithm heapSort(data)
data - Array of items to be sorted

heapify(data)
i = data.length-1
while( i > 0 )
    // Move the next largest item into sorted position
    swap(data[0], data[i])
    i = i - 1;
    // Re-heapify by moving the new root down.
    moveDown(data, 0, i)
```

Efficiency of Sorts

	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$ (array in reverse order)	$O(n)$ (array already sorted)
Quick Sort*	$O(n^2)$ (array already sorted)	$O(n \log n)$ (by luck, pivot is always the median)
Merge Sort	$O(n \log n)$	$O(n \log n)$
Tree Sort**	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$

* Quick sort is $O(n \log n)$ in the average case.

Next Class

- Next class reading: Chapter 24: Linear-time Sorting Algorithms