Lecture 8: Chapter 2 part 3

Divide-and-Conquer
CS3310

Quicksort

Quicksort is another divide-and-conquer sorting algorithm. Top-Level Description:

- A **pivot item** is chosen in the array.
  - For simplicity we will choose the first item in the array for now.
- Each item in the array less than the *pivot item* is placed before it and every item greater than the pivot item is placed after it.
- Recursively call Quicksort on the subarray to the left of the pivot and the subarray to its right.

Given this array: [15, 22, 13, 27, 12, 10, 20, 25]

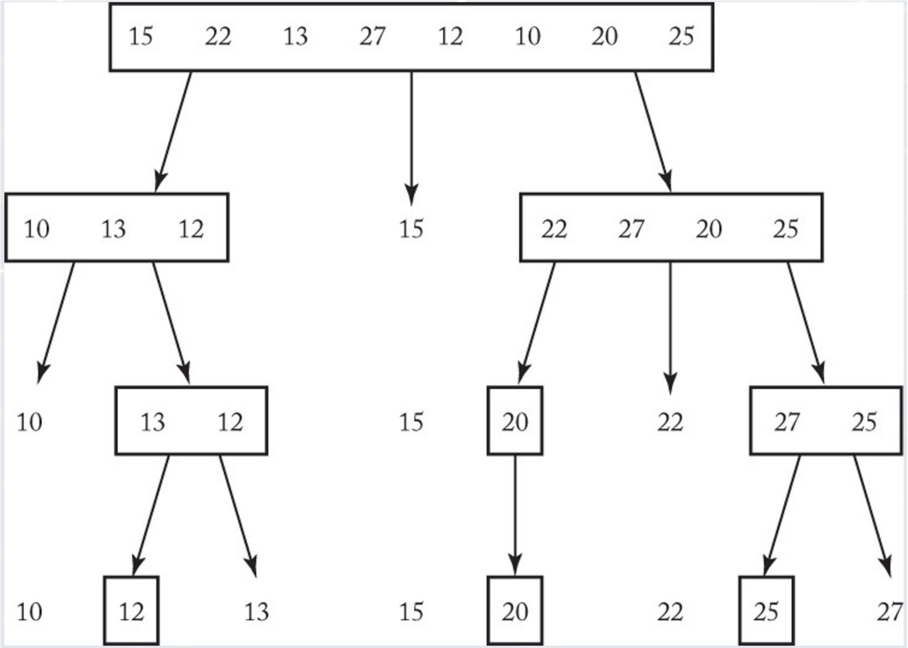
- 15 is our pivot item. Place smaller items before it, larger items after:

  [10, 13, 12, 15, 22, 27, 20, 25]
- Recursively sort the two 'subarrays' with Quicksort:

  [**10, 12, 13**, 15, **20, 22, 25, 27**]

Quicksort

The following diagram shows the recursive tree of `Quicksort`



Quicksort

Problem: Sort *n* keys in nondecreasing order

Inputs: Positive integer *n*, array of keys *S* indexed from 1 to *n*, `low` and `high` which indicate the indices of the subarray to sort.

Outputs: The array *S* containing the keys in nondecreasing order.

```
void quicksort(index low, index high)
{
      index pivotpoint;          // passed by reference and set within partition
      if (high > low)
      {
            partition(low, high, pivotpoint);
            quicksort(low, pivotpoint - 1);
            quicksort(pivotpoint + 1, high);
      }
}
```

Quicksort

- Following the book's convention, *n* and *S* are not parameters to `QuickSort`.
- A top-level call to `QuickSort`: `quickSort(1, n);`
- Just as `merge` performs most of the work in `MergeSort`, `partition` performs most of the work in `QuickSort`.

- `partition` accepts `low` and `high` as parameters, indicating which section of the array is currently being sorted.
  - It also accepts `pivotpoint` as a reference parameter.
  - The item at index `low` is chosen as the `pivotitem`.
  - Every item less than `pivotitem` is placed before it in the array.
  - `pivotpoint` is set to `pivotitem`'s new index.

Partition

Problem: Partition the array S for Quicksort

Inputs: two indices indicating the subarray of *S* to be partitioned

Outputs: `pivotpoint`, the index of `pivotitem` in the subarray indexed from `low` to `high`

```
void partition(index low, index high, index &pivotpoint)
{
      index i, j;
      keytype pivotitem;

      pivotitem = S[low];                  // choose first item as pivotitem
      j = low;                             // j tracks where to place pivotitem
      for (i = low + 1; i <= high; i++)
           if (S[i] < pivotitem)
                j++;
                exchange S[i] and S[j];
      pivotpoint = j;
      exchange S[low] and S[pivotpoint]; // put pivotitem at pivotpoint
}
```

Partition

A top-level call to `partition`.

`low = 1, high = 8`

i	*j*	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
-	-	15	22	13	27	12	10	20	25
2	1	**15**	**22**	13	27	12	10	20	25

Step 1:

```
pivotitem = ??
j = ??
i = ??
```

Partition

A top-level call to `partition`. Items compared each step are in bold.

`low = 1, high = 8`

i	*j*	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
-	-	15	22	13	27	12	10	20	25
2	1	**15**	**22**	13	27	12	10	20	25

Step 1:

```
pivotitem = S[1]
j = low = 1
i = low + 1 = 2
```

- We select the first element as `pivotitem` and compare it with `S[i]`
- `S[i]` is bigger, so we make no change.

Partition

A top-level call to `partition`. Items compared each step are in bold.

`low = 1, high = 8`

i	*j*	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
-	-	15	22	13	27	12	10	20	25
2	1	**15**	**22**	13	27	12	10	20	25
3	1 → 2	**15**	22	**13**	27	12	10	20	25

Step 2:

```
pivotitem = S[1]
j = 1 at beginning of step, 2 at end of step
i = 3
```

Compare `S[1]` with `S[i]`. `S[i]` is smaller: increment *j* and <u>then</u> swap `S[i]` and `S[j]` before step 3.

Partition

A top-level call to `partition`. Items compared each step are in bold. Red items were swapped at the end of the previous step. At the end, `pivotpoint` is set to j

	i	*j*	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
	-	-	15	22	13	27	12	10	20	25
	2	1	**15**	**22**	13	27	12	10	20	25
15 < 13. Increment j. Swap S[i] and S[j].	3	1 → 2	**15**	22	**13**	27	12	10	20	25
	4	2	**15**	13	22	**27**	12	10	20	25
15 < 12. Increment j. Swap S[i] and S[j].	5	2 → 3	**15**	13	22	27	**12**	10	20	25
15 < 10. Increment j. Swap S[i] and S[j].	6	3 → 4	**15**	13	12	27	22	**10**	20	25
	7	4	**15**	13	12	10	22	27	**20**	25
	8	4	**15**	13	12	10	22	27	20	**25**
End of list reached. Swap S[1] and S[j]	-	4	10	13	12	15	22	27	20	25

Partition Analysis

```
for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem)
        j++;
        exchange S[i] and S[j];
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];  // put pivot item at pivot point
```

What is the basic operation of `partition`?

Partition Analysis

```
for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem)
        j++;
        exchange S[i] and S[j];
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];  // put pivot item at pivot point
```

What is the basic operation of `partition`? The comparison of `S[i]` with `pivotitem`

Does `partition` have an every-case time complexity?

Partition Analysis

```
for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem)
        j++;
        exchange S[i] and S[j];
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];  // put pivot item at pivot point
```

What is the basic operation of `partition`? The comparison of `S[i]` with `pivotitem`

Does `partition` have an every-case time complexity? **Yes!** No way to break from loop early.

What is T(n)?

Partition Analysis

```
for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem)
        j++;
        exchange S[i] and S[j];
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];  // put pivot item at pivot point
```

What is the basic operation of `partition`? The comparison of `S[i]` with `pivotitem`

Does `partition` have an every-case time complexity? **Yes!** No way to break from loop early.

What is T(n)? $\boldsymbol{n - 1}$, since the first item is not compared with itself.

QuickSort Analysis

Now that we have analyzed `partition`, we can analyze `Quicksort` overall.

```
if (high > low)
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
```

What is the basic operation?

QuickSort Analysis

Now that we have analyzed `partition`, we can analyze `Quicksort` overall.

```
if (high > low)
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
```

What is the basic operation? The comparison of `S[i]` with `pivotitem` in `partition`.

Does `Quicksort` have an every-case complexity?

QuickSort Analysis

Now that we have analyzed `partition`, we can analyze `Quicksort` overall.

```
if (high > low)
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
```

What is the basic operation? The comparison of `S[i]` with `pivotitem` in `partition`.

Does `Quicksort` have an every-case complexity? **No!**

Worst case scenario?

QuickSort Analysis

Now that we have analyzed `partition`, we can analyze `Quicksort` overall.

```
if (high > low)
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
```

What is the basic operation? The comparison of `S[i]` with `pivotitem` in `partition`.

Does `Quicksort` have an every-case complexity? **No!**

Worst case scenario?

➢ The case in which the list is already sorted.

QuickSort Analysis

```
if (high > low)
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
```

- If the list is already sorted:
    - `partition` sets `pivotpoint` to 1 (since nothing is placed before the pivot item). The first recursive call to `Quicksort` passes 0 for `high` so it sorts 0 items and performs 0 operations.
    - The second recursive call to `Quicksort` sorts n - 1 items.

What is the recurrence relation?

QuickSort Analysis

```
if (high > low)
    partition(low, high, pivotpoint);
    quicksort(low, pivotpoint - 1);
    quicksort(pivotpoint + 1, high);
```

- If the list is already sorted:
  - `partition` sets `pivotpoint` to 1 (since nothing is placed before the pivot item). The first recursive call to `Quicksort` passes 0 for `high` so it sorts 0 items and performs 0 operations.
  - The second recursive call to `Quicksort` sorts n - 1 items.

What is the recurrence relation?

$$w_n = w_0 + w_{n-1} + n - 1$$

time to sort left subarray, time to sort right subarray, time to partition

QuickSort Analysis

Recurrence Relation: $w_n = w_0 + w_{n-1} + n - 1$

When the input to `QuickSort` is 0, 0 operations are performed. Therefore:

$$w_0 = 0$$
$$w_n = w_{n-1} + n - 1$$

Solving this recurrence relation gives us:

$$w_n = n(n - 1) / 2 \in \Theta(n^2)$$

QuickSort Analysis

- Thankfully, `QuickSort` is usually much more efficient than its worst case.
  - In the average case, it is $\Theta(n \lg n)$
- `QuickSort` can also be improved by choosing the pivot item in a different way if we suspect the array is likely already sorted.
  - We will discuss this strategy in detail in chapter 7.

- Another way to improve `QuickSort` uses the concept of **thresholds**, which we will cover more in the next lecture.
- There is overhead involved each time we divide an array and push new recursive calls to the stack. In recursive calls in which n is small enough, it is quicker to call an iterative algorithm on the remaining subarray rather than continue with `QuickSort`.

In-Class Exercise

Consider the following array:

{ 123, 34, 189, 56, 150, 12, 9, 240 }

1. Trace the steps taken in the top-level call to `partition`.
2. Draw the recursive tree that `QuickSort` builds when sorting this array into ascending order. Assume that the first item is chosen as the pivot.
3. Sort 65, 60_1, 60_2, 60_3 in nondecreasing order using `Quicksort`. A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. Is `Quicksort` stable?