Data Structure

Lecture#17: Internal Sorting 2 (Chapter 7)

U Kang
Seoul National University

In This Lecture

- Main ideas and analysis of Merge sort
- Main ideas and analysis of Quicksort

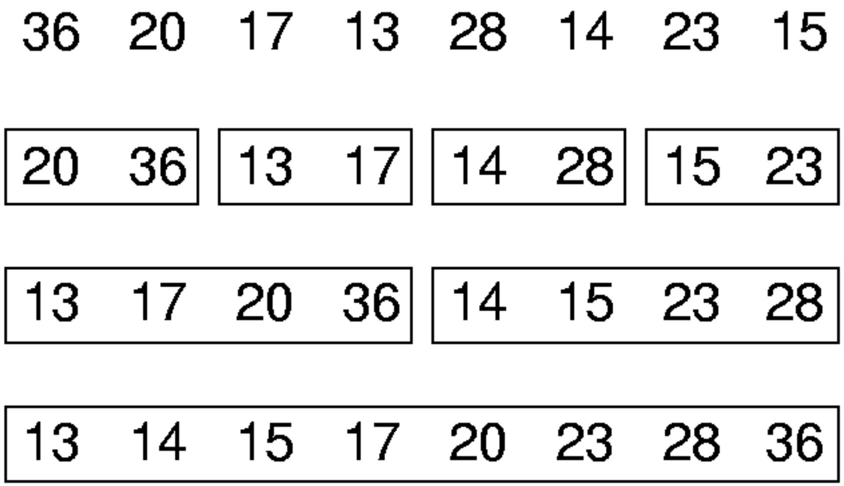
Merge Sort (1)

- Split the input array in half, sort the halves, and merge the sorted values

- An example of “Divide and Conquer” approach

Merge Sort – Pseudo Code

```
List mergesort(List inlist) {
  if (inlist.length() <= 1)return inlist;
  List l1 = half of the items from inlist;
  List l2 = other half of items from inlist;
  return merge(mergesort(l1),
               mergesort(l2));
}
```



36 20 17 13 28 14 23 15

20 36 | 13 17 | 14 28 | 15 23

13 17 20 36 | 14 15 23 28

13 14 15 17 20 23 28 36

Merge sort Implementation

```
static <E extends Comparable<? super E>>
void mergesort(E[] A, E[] temp, int l, int r) {
  int mid = (l+r)/2;
  if (l == r) return;
  mergesort(A, temp, l, mid);
  mergesort(A, temp, mid+1, r);
  for (int i=l; i<=r; i++)  // Copy subarray
    temp[i] = A[i];
  // Do the merge operation back to A
  int i1 = l; int i2 = mid + 1;
  for (int curr=l; curr<=r; curr++) {
    if (i1 == mid+1) // Left sublist exhausted
      A[curr] = temp[i2++];
    else if (i2 > r) // Right sublist exhausted
      A[curr] = temp[i1++];
    else if (temp[i1].compareTo(temp[i2])<0)
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
  }
}
```

Optimized Merge sort

- Two optimizations in the implementation of previous slide
  - Use insertion sort when the input array size is small
    - Insertion sort: no need for recursion and copying the input array to temp
  - Skip checking the end of list

Optimized Mergesort

```
void mergesort(E[] A, E[] temp, int l, int r) {
  int i, j, k, mid = (l+r)/2;
  if (l == r) return;  // List has one element
  if ((mid-l) >= TH)
    mergesort(A, temp, l, mid);
  else inssort(A, l, mid-l+1);
  if ((r-mid) > TH)
    mergesort(A, temp, mid+1, r);
  else inssort(A, mid+1, r-mid);
  // Do merge.  First, copy 2 halves to temp.
  for (i=l; i<=mid; i++) temp[i] = A[i];
  for (j=1; j<=r-mid; j++)
    temp[r-j+1] = A[j+mid];
  // Merge sublists back to array
  for (i=l,j=r,k=l; k<=r; k++)
    if (temp[i].compareTo(temp[j])<0)
      A[k] = temp[i++];
    else A[k] = temp[j--];
}
```

Mergesort Cost

- Mergesort cost:
  - $\Theta(n \log n)$ in the best, average, and worst cases
- Mergesort is also good for sorting linked lists.
  - Because merging requires only sequential access
- Mergesort requires twice the space.

Quicksort

- Another “Divide and Conquer” approach

- Given an input array
  - Select a value (called “pivot”) in the array
  - Rearrange the array so that values smaller than the pivot is located in the left-side of the pivot, values larger than the pivot is in the right-side of the pivot
    - This is called the “partition” operation
    - Values equal to pivot can be in either side

Quicksort

```
static <E extends Comparable<? super E>>
void qsort(E[] A, int i, int j) {
  int pivotindex = findpivot(A, i, j);
  DSutil.swap(A, pivotindex, j);
  // k will be first position in right subarray
  int k = partition(A, i-1, j, A[j]);
  DSutil.swap(A, k, j);
  if ((k-i) > 1) qsort(A, i, k-1);
  if ((j-k) > 1) qsort(A, k+1, j);
}

static <E extends Comparable<? super E>>
int findpivot(E[] A, int i, int j)
  { return (i+j)/2; }
```

Quicksort Partition

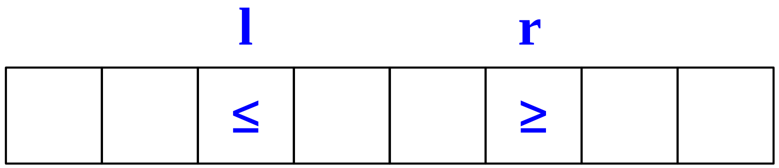
```
static <E extends Comparable<? super E>>
int partition(E[] A, int l, int r, E pivot) {
  do { (1) // Move bounds inward until they meet
    while (A[++l].compareTo(pivot)<0); (2)
    while ((r!=0) &&
           (A[--r].compareTo(pivot)>0)); (3)
    DSutil.swap(A, l, r);
  } while (l < r);
  DSutil.swap(A, l, r);
  return l;
}
```

- Return value is the index of the first element of the second half
  - A[< l]: smaller or equal to pivot,   A [≥ l]: greater than or equal to pivot
- The cost for partition is $\Theta(n)$.

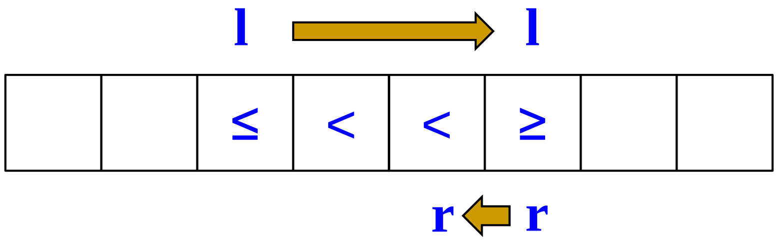
Quicksort Partition

≤: smaller than or equal to pivot
≥: greater than or equal to pivot
<: smaller than pivot
>: greater than pivot

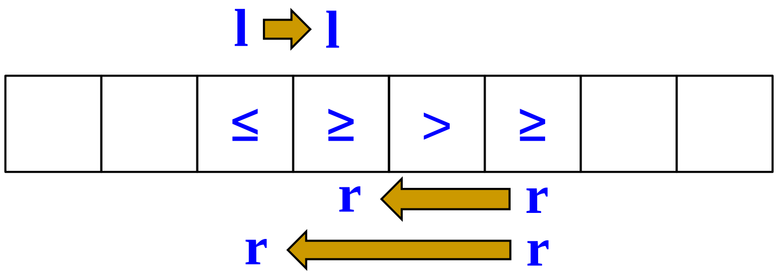
- At (1)



- Assume "while (l < r)" is violated. Then either of the followings is true
  - Violation at (2):



  - No violation at (2); violation at (3)

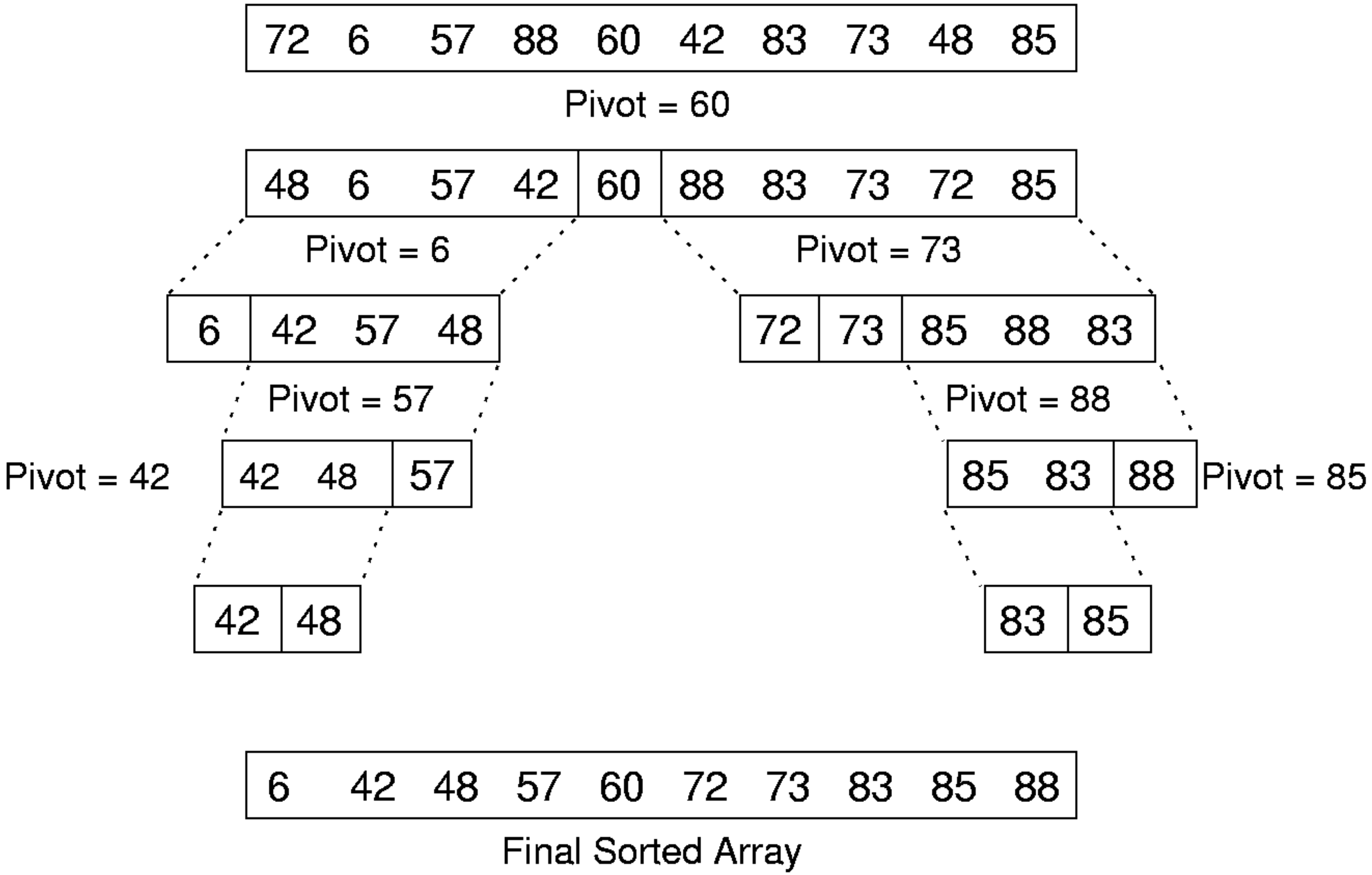


Partition Example

Pivot=60

Initial		72	6	57	88	85	42	83	73	48	60
	l										r
Pass 1		72	6	57	88	85	42	83	73	48	60
		l								r	
Swap 1		48	6	57	88	85	42	83	73	72	60
		l								r	
Pass 2		48	6	57	88	85	42	83	73	72	60
					l		r				
Swap 2		48	6	57	42	85	88	83	73	72	60
					l		r				
Pass 3		48	6	57	42	85	88	83	73	72	60
					r	l					
Swap 3		48	6	57	85	42	88	83	73	72	60
					r	l					
Reverse Swap		48	6	57	42 │	85	88	83	73	72	60
					r	l					

Quicksort Example



72 6 57 88 60 42 83 73 48 85

Pivot = 60

48 6 57 42 | 60 | 88 83 73 72 85

Pivot = 6

Pivot = 73

6 | 42 57 48

72 | 73 | 85 88 83

Pivot = 57

Pivot = 88

Pivot = 42

42 48 | 57

85 83 | 88

Pivot = 85

42 | 48

83 | 85

6 42 48 57 60 72 73 83 85 88

Final Sorted Array

Cost of Quicksort

- Best case: always partition in half.
- Worst case: bad partition.
- Average case:
  - $\boldsymbol{T(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-1-k)]}$
  - **T(0) = T(1) = c**

  - Solving the above recurrence relation leads to T(n) = Θ(*n log n*)

Cost of Quicksort

Details

- Proof of T(n) = Θ(*n log n*)
  - Rewrite the equation as $\boldsymbol{T(n) = cn + \frac{2}{n}\sum_{k=0}^{n-1} T(k)}$
  - Multiply n to both sides:

$$\begin{aligned} n\mathbf{T}(n) &= cn^2 + 2\sum_{k=1}^{n-1}\mathbf{T}(k) \\ (n+1)\mathbf{T}(n+1) &= c(n+1)^2 + 2\sum_{k=1}^{n}\mathbf{T}(k). \end{aligned}$$

  - Subtracting nT(n) from both sides:

$$\begin{aligned} (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(n+1)^2 - cn^2 + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(2n+1) + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) &= c(2n+1) + (n+2)\mathbf{T}(n) \\ \mathbf{T}(n+1) &= \frac{c(2n+1)}{n+1} + \frac{n+2}{n+1}\mathbf{T}(n). \end{aligned}$$

Details

Cost of Quicksort

- Proof of T(n) = Θ(*n log n*)
  - (cont.)
  - Using the fact that $\frac{c(2n+1)}{(n+1)} < 2c$,

$$\begin{aligned}
\mathbf{T}(n+1) &\le 2c + \frac{n+2}{n+1}\mathbf{T}(n) \\
&= 2c + \frac{n+2}{n+1}\left(2c + \frac{n+1}{n}\mathbf{T}(n-1)\right) \\
&= 2c + \frac{n+2}{n+1}\left(2c + \frac{n+1}{n}\left(2c + \frac{n}{n-1}\mathbf{T}(n-2)\right)\right) \\
&= 2c + \frac{n+2}{n+1}\left(2c + \cdots + \frac{4}{3}(2c + \frac{3}{2}\mathbf{T}(1))\right) \\
&= 2c\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \\
&= 2c\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2c + 2c(n+2)\left(\mathcal{H}_{n+1} - 1\right)
\end{aligned}$$

Optimizations for Quicksort

- Optimizations for Quicksort:
  - Better Pivot
  - Better algorithm for small sublists
    - If n is small, Quicksort is relatively slow
    - Use insertion sort or selection sort for small sublists
  - Eliminate recursion: e.g., use stack

Summary

- Merge sort
  - Main idea: ‘divide and conquer’
  - Cost analysis
  - Advantage of optimized merge sort

- Quicksort
  - Main idea: ‘divide and conquer’
  - Cost analysis

Questions?