CS 253: Algorithms

Chapter 7

Mergesort
Quicksort

Credit: Dr. George Bebis
Sorting

• Insertion sort
  ◦ Design approach: incremental
  ◦ Sorts in place: Yes
  ◦ Best case: $\Theta(n)$
  ◦ Worst case: $\Theta(n^2)$

• Bubble Sort
  ◦ Design approach: incremental
  ◦ Sorts in place: Yes
  ◦ Running time: $\Theta(n^2)$
Sorting

- **Selection sort**
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- **Merge Sort**
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!
Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size

- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner

- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem
Merge Sort Approach

To sort an array $A[p . . r]$:

- **Divide**
  - Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each

- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do

- **Combine**
  - Merge the two sorted subsequences
Merge Sort

Alg.: MERGE-SORT(A, p, r)

if p < r

then q ← ⌊(p + r)/2⌋

MERGE-SORT(A, p, q)  ➔ Check for base case

MERGE-SORT(A, q + 1, r) ➔ Divide

MERGE-SORT(A, q + 1, r) ➔ Conquer

MERGE(A, p, q, r) ➔ Conquer

MERGE(A, p, q, r) ➔ Combine

• Initial call: MERGE-SORT(A, 1, n)
Example – n Power of 2

Divide

q = 4
Example – n Power of 2

Conquer and Merge

```
1 2 2 3 4 5 6 7
```

```
2 4 5 7
```

```
1 2 3 4
```

```
5 6 7 8
```

```
2 5
```

```
4 7
```

```
1 3
```

```
2 6
```

```
5 2 4 7
```

```
1 3 2 6
```

```
5 2 4 7 1 3 2 6
```

Example – n not a Power of 2

Divide

q = 6

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
4 & 7 & 2 & 6 & 1 & 4 & 7 & 3 & 5 & 2 & 6 \\
\end{array}
\]

q = 3

\[
\begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
4 & 7 & 2 & 6 & 1 & 4 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
7 & 8 & 9 & 10 & 11 \\
7 & 3 & 5 & 2 & 6 \\
\end{array}
\]

q = 9

\[
\begin{array}{ccccccccccc}
1 & 2 & 3 \\
4 & 7 & 2 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
4 & 5 & 6 \\
6 & 1 & 4 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
7 & 8 & 9 \\
7 & 3 & 5 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
10 & 11 \\
2 & 6 \\
\end{array}
\]

\[
\begin{array}{ccccccccccc}
1 & 2 \\
4 & 7 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
4 & 5 \\
6 & 1 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
7 & 8 \\
7 & 3 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
11 \\
3 \\
\end{array}
\]

\[
\begin{array}{ccccccccccc}
1 & 2 \\
4 & 7 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
4 & 5 \\
6 & 1 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
7 & 8 \\
7 & 3 \\
\end{array}
\quad
\begin{array}{ccccccccccc}
11 \\
3 \\
\end{array}
\]

Divide
Example – $n$ Not a Power of 2

Conquer and Merge
Merging

**Input:**

Array $A$ and indices $p$, $q$, $r$ such that $p \leq q < r$

- Subarrays $A[p\ldots q]$ and $A[q+1\ldots r]$ are sorted

**Output:** One single sorted subarray $A[p\ldots r]$
Merging

Strategy:

- Two piles of sorted cards
  - Choose the smaller of the two top cards
  - Remove it and place it in the output pile
- Repeat the process until one pile is empty
- Take the remaining input pile and place it face-down onto the output pile
Example: $\text{MERGE}(A, 9, 12, 16)$
Example (cont.)

\[
\begin{array}{cccccccccccc}
A & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\vdots & 1 & 2 & 2 & 3 & 1 & 2 & 3 & 6 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
L & 1 & 2 & 3 & 4 & 5 \\
2 & 4 & 5 & 7 & \infty \\
i
\end{array}
\quad k
\quad
\begin{array}{cccccccc}
R & 1 & 2 & 3 & 4 & 5 \\
1 & 2 & 3 & 6 & \infty \\
j
\end{array}
\]

\[
\begin{array}{cccccccccccc}
A & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\vdots & 1 & 2 & 2 & 3 & 4 & 2 & 3 & 6 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
L & 1 & 2 & 3 & 4 & 5 \\
2 & 4 & 5 & 7 & \infty \\
i
\end{array}
\quad k
\quad
\begin{array}{cccccccc}
R & 1 & 2 & 3 & 4 & 5 \\
1 & 2 & 3 & 6 & \infty \\
j
\end{array}
\]
Example (cont.)

Done!
Alg.: MERGE(A, p, q, r)

1. Compute \( n_1 \) and \( n_2 \)
2. Copy the first \( n_1 \) elements into \( L[1 \ldots n_1 + 1] \) and the next \( n_2 \) elements into \( R[1 \ldots n_2 + 1] \)
3. \( L[n_1 + 1] \leftarrow \infty; \quad R[n_2 + 1] \leftarrow \infty \)
4. \( i \leftarrow 1; \quad j \leftarrow 1 \)
5. for \( k \leftarrow p \) to \( r \)
6.   do if \( L[i] \leq R[j] \)
7.      then \( A[k] \leftarrow L[i] \)
8.      \( i \leftarrow i + 1 \)
9.   else \( A[k] \leftarrow R[j] \)
10. \( j \leftarrow j + 1 \)
Running Time of Merge

- Initialization (copying into temporary arrays):
  \( \Theta(n_1 + n_2) = \Theta(n) \)

- Adding the elements to the final array:
  - \( n \) iterations, each taking constant time \( \Rightarrow \Theta(n) \)

**Total time for Merge:** \( \Theta(n) \)
Analyzing Divide-and-Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size $n$
  - **Divide** the problem into $a$ subproblems, each of size $n/b$: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c \\
 aT(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases}$$
MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: \( D(n) = \Theta(1) \)

- **Conquer:**
  - recursively solve 2 subproblems, each of size \( n/2 \) \( \Rightarrow 2T(n/2) \)

- **Combine:**
  - MERGE on an \( n \)-element subarray takes \( \Theta(n) \) time \( \Rightarrow C(n) = \Theta(n) \)

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases}
\]
Solve the Recurrence

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n/2) + cn & \text{if } n > 1 
\end{cases}
\]

Use Master Theorem:

\[
\begin{align*}
  & a = 2, \quad b = 2, \quad \log_2 2 = 1 \\
  & \text{Compare } n^{\log_b a} = n^1 \text{ with } f(n) = cn \\
  & f(n) = \Theta(n^{\log_b a} = n^1) \Rightarrow \text{Case 2}
\end{align*}
\]

\[
\Rightarrow T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)
\]
Notes on Merge Sort

- Running time insensitive of the input

Advantage:
- Guaranteed to run in $\Theta(n\log n)$

Disadvantage:
- Requires extra space $\approx N$
Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time $\sim N \log N$
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort
Sorting Files with **Huge Records** and **Small Keys**

- **Insertion sort or bubble sort?**
  - NO, too many exchanges

- **Selection sort?**
  - YES, it takes *linear* time for exchanges

- **Merge sort or custom method?**
  - Probably not: selection sort simpler, does less swaps
Problem:
Sort a huge randomly-ordered file of small records

Example: transaction record for a phone company

Which method to use?
A. bubble sort
B. selection sort
C. merge sort, guaranteed to run in time ~NlgN
D. insertion sort
Sorting **Huge**, Randomly - Ordered Files

- **Selection sort?**
  - NO, always takes quadratic time

- **Bubble sort?**
  - NO, quadratic time for randomly-ordered keys

- **Insertion sort?**
  - NO, quadratic time for randomly-ordered keys

- **Mergesort?**
  - YES, it is designed for this problem
Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:
- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

Which sorting method to use?
A. Mergesort, guaranteed to run in time $\sim N \lg N$
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort
Sorting files that are almost in order

- Selection sort?
  - NO, always takes quadratic time

- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A

- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”

- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster
Quicksort

- Sort an array $A[p...r]$  
- **Divide**
  - Partition the array $A$ into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$  
  - Need to find index $q$ to partition the array.
Quicksort

• **Conquer**
  ◦ Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort

• **Combine**
  ◦ Trivial: the arrays are sorted in place
  ◦ No additional work is required to combine them
  ◦ When the original call returns, the entire array is sorted
QUICKSORT

Alg.: QUICKSORT(A, p, r) % Initially p=1 and r=n

if p < r

then q ← PARTITION(A, p, r)

QUICKSORT (A, p, q)
QUICKSORT (A, q+1, r)

Recurrence:

\[ T(n) = T(q) + T(n - q) + f(n) \]  \((f(n)\) depends on \PARTITION())
Partitioning the Array

- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages

- Hoare partition (see prob. 7-1)
  - Select a pivot element $x$ around which to partition
  - Grows two regions
    \[
    A[p...i] \leq x \\
    x \leq A[j...r]
    \]
Example

\[ A[p\ldots r] \]

pivot \( x = 5 \)

\[
\begin{array}{c}
5 & 3 & 2 & 6 & 4 & 1 & 3 & 7 \\
\end{array}
\]

\[
\begin{array}{c}
5 & 3 & 2 & 6 & 4 & 1 & 3 & 7 \\
\end{array}
\]

pivot \( x = 5 \)

\[
\begin{array}{c}
3 & 3 & 2 & 6 & 4 & 1 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{c}
3 & 3 & 2 & 6 & 4 & 1 & 5 & 7 \\
\end{array}
\]

pivot \( x = 5 \)

\[
\begin{array}{c}
3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{c}
3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\
\end{array}
\]

pivot \( x = 5 \)

\[
\begin{array}{c}
3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{c}
3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\
\end{array}
\]
Example
Partitioning the Array

**Alg. PARTITION** (A, p, r)

1. \( x \leftarrow A[p] \)
2. \( i \leftarrow p - 1 \)
3. \( j \leftarrow r + 1 \)
4. **while** TRUE
   5.  **do** repeat \( j \leftarrow j - 1 \) **until** \( A[j] \leq x \)
   6.  **do** repeat \( i \leftarrow i + 1 \) **until** \( A[i] \geq x \)
   7.  **if** \( i < j \) % Each element is visited once!
      8.  **then** exchange \( A[i] \leftrightarrow A[j] \)
   9. **else** return \( j \)

**Running time:** \( \Theta(n) \quad n = r - p + 1 \)
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has \( n - 1 \) elements
  - Maximally unbalanced

- Recurrence: \( q=1 \)

\[
T(n) = T(1) + T(n - 1) + n,
\]

\[
T(1) = \Theta(1)
\]

\[
T(n) = T(n - 1) + n
\]

When does the worst case happen?
Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$

- Recurrence: $q=n/2$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \lg n) \text{ (Master theorem)}$$
**here Case Between Worst and Best**

Example: 9-to-1 proportional split $\Rightarrow T(n) = T(9n/10) + T(n/10) + n$

longest path: $T(n) \leq n\log_{10/9} n + 1 = c_2 n \log n$

shortest path: $T(n) \geq n\log_{10} n = c_1 n \log n$

Thus, $\Rightarrow T(n) = \Theta(n \log n)$
How does partition affect performance?

- Any splitting of constant proportionality yields $\Theta(n\log n)$ time !!

- Consider the $(1 : n - 1)$ splitting:

  \[
  \text{ratio} = \frac{1}{n - 1} \text{ not a constant !!!}
  \]

- Consider the $(n/2 : n/2)$ splitting:

  \[
  \text{ratio} = \frac{n/2}{n/2} = 1 \text{ it is a constant !!}
  \]

- Consider the $(9n/10 : n/10)$ splitting:

  \[
  \text{ratio} = \frac{9n/10}{n/10} = 9 \text{ it is a constant !!}
  \]
How does partition affect performance?

- Any \(((a - 1)n/a : n/a)\) splitting:

\[
\text{ratio} = \frac{(a - 1)n/a}{n/a} = a - 1 \text{ it is a constant ! !}
\]
Performance of Quicksort

- **Average case**
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a mix of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree

- Therefore, \( T(n) = 2T(n/2) + \Theta(n) \) still holds!
- And **running time** of Quicksort when levels alternate between good and bad splits is \( O(n \log n) \)