Beyond $\Theta(n^2)$: Mergesort

C. Seshadhri

University of California, Santa Cruz
sesh@ucsc.edu

April 4, 2016
We saw numerous algorithms that sort in $\Theta(n^2)$. We also saw that the total number of possible comparisons is $\binom{n}{2} = \Theta(n^2)$. Can we overcome this bound? Are there sorting algorithms that can significantly beat $n^2$?
Beyond $\Theta(n^2)$

- We saw numerous algorithms that sort in $\Theta(n^2)$.
- We also saw that the total number of possible comparisons is $\binom{n}{2} = \Theta(n^2)$.
- Can we overcome this bound? Are there sorting algorithms that can significantly beat $n^2$?
Advice from history
Advice from history

Divide et impera: Divide and conquer
Divide and Conquer: Basic Anatomy

- **Divide** the problem into smaller subproblems
- **Conquer** subproblems by solving them recursively. (If subproblem is small, solve directly.)
- **Combine** the solutions to the subproblems into full solution
Divide and Conquer: Sorting

Input is array $A$ of size $n$

- **Divide:** Input $A$ into two subarrays of size $n/2$
- **Conquer:** Recursively sort the subarrays. When array is of size 1, sorting is trivial.
- **Combine:** Merge two sorted arrays into sorted version of $A$. 
Mergesort

**MERGE-SORT**(*A*)
1. \( n = A.\text{length} \)
2. **if** \( n == 1 \), **return** \( A \)
3. \( L = A[1..n/2] \)  // Technically, use \( \lfloor n/2 \rfloor \)
4. \( R = A[n/2 + 1..n] \)
5. \( L_{\text{sort}} = \text{MERGE-SORT}(L) \)
6. \( R_{\text{sort}} = \text{MERGE-SORT}(R) \)
7. **return** \( \text{MERGE}(L_{\text{sort}}, R_{\text{sort}}) \)  // Process to merge two sorted arrays
The power of Divide and Conquer

- **Sorting**: You know what this is
- **Merging**: Given two sorted arrays $B$ and $C$, merge into a fully sorted version
The power of Divide and Conquer

- **Sorting:** You know what this is
- **Merging:** Given two sorted arrays $B$ and $C$, merge into a fully sorted version
- Merging looks easier than Sorting.
- Amazingly, a solution of merging yields a solution for sorting!
Proof that Merge-Sort works

Induction on $n$, size of input

Base case

- $n = 1$: Merge-Sort return $A$, which is trivially sorted.

Thus, $L_{sort}$ and $R_{sort}$ are sorted versions of $L$ and $R$ respectively. Since Merge works correctly, output is sorted version of $A$!
Proof that Merge-Sort works

Induction on $n$, size of input

**Base case**
- $n = 1$: Merge-Sort return $A$, which is trivially sorted.

**Induction**
Assume $n > 1$. By (strong) induction hypothesis, Merge-Sort correctly sorts any input of size $\leq n - 1$. Specifically, Merge-Sort works on $L$ and $R$. 

Quod Erat Demonstrandum
Proof that Merge-Sort works

Induction on $n$, size of input

**Base case**

- $n = 1$: Merge-Sort return $A$, which is trivially sorted.

**Induction**

Assume $n > 1$. By (strong) induction hypothesis, Merge-Sort correctly sorts any input of size $\leq n - 1$. Specifically, Merge-Sort works on $L$ and $R$.

Thus, $L_{\text{sort}}$ and $R_{\text{sort}}$ are sorted versions of $L$ and $R$ respectively. Since Merge works correctly, output is sorted version of $A$!
Proof that Merge-Sort works

Induction on $n$, size of input

**Base case**
- $n = 1$: Merge-Sort return $A$, which is trivially sorted.

**Induction**
Assume $n > 1$. By (strong) induction hypothesis, Merge-Sort correctly sorts any input of size $\leq n - 1$. Specifically, Merge-Sort works on $L$ and $R$.

Thus, $L_{\text{sort}}$ and $R_{\text{sort}}$ are sorted versions of $L$ and $R$ respectively. Since Merge works correctly, output is sorted version of $A$!

Quod Erat Demonstrandum
How to merge?

Given two sorted arrays $B$ and $C$, each of size $m$, what is the time complexity of finding the minimum over all elements in $B$ and $C$? Show me your best answer.

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$
How to merge?

Given two sorted arrays $B$ and $C$, each of size $m$, what is the time complexity of finding the minimum over all elements in $B$ and $C$? Show me your best answer.

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$

The minimum of $B$ is the first element in $B$, the minimum of $C$ is the first element in $C$. Just take the minimum of those two!
How to merge?

Given two sorted arrays $B$ and $C$, each of size $m$, what is the time complexity of finding the second minimum over all elements in $B$ and $C$? Show me your best answer.

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$
Mergesort

How to merge?

Given two sorted arrays $B$ and $C$, each of size $m$, what is the time complexity of finding the second minimum over all elements in $B$ and $C$? Show me your best answer.

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$

If the minimum of $B$ and $C$ is $B[1]$, then find the minimum of $B[2..m]$, $C$. Otherwise, find the minimum of $B$, $C[2..m]$. 
Mergesort

Merging

Assume both inputs have length \( m \).

\text{MERGE}(B, C)

1. \( B[m+1] = C[m+1] = \infty \)
2. Initialize \( D \) as empty array
3. \( i = j = k = 1 \)
4. \textbf{while} \( B[i] < \infty \) or \( C[j] < \infty \)
5. \textbf{if} \( B[i] < C[j] \)
6. \( D[k] = B[i] \)
7. \( i = i + 1 \)
8. \textbf{else}
9. \( D[k] = C[j] \)
10. \( j = j + 1 \)
11. \( k = k + 1 \)
12. \textbf{return} \( D \)
Run it!

Run merge on $B = 2 4 5 7$ and $C = 1 2 3 6$
- (R) Please run it.
- (B) We get it, let’s proceed.
Merging two sorted arrays

(a) 

(b) 

(c) 

(d)
Merging two sorted arrays

Mergesort
Merge: Loop Invariant

At the beginning on each while loop, \( D \) contains all elements of \( B[1..i-1] \) and \( C[1..j-1] \) in sorted order.
Merge: Loop Invariant

At the beginning on each \textbf{while} loop, $D$ contains all elements of $B[1..i-1]$ and $C[1..j-1]$ in sorted order.

(When \textbf{while} loop terminates, $i = j = m + 1$. So by loop invariant, $D$ is the desired output.)
Merging

Assume both inputs have length $m$.

\[
\text{MERGE}(B, C) \\
1 \quad B[m + 1] = C[m + 1] = \infty \\
2 \quad \text{Initialize } D \text{ as empty array} \\
3 \quad i = j = k = 1 \\
4 \quad \textbf{while } B[i] < \infty \text{ or } C[j] < \infty \\
5 \quad \quad \textbf{if } B[i] < C[j] \\
6 \quad \quad \quad D[k] = B[i] \\
7 \quad \quad \quad i = i + 1 \\
8 \quad \quad \textbf{else} \\
9 \quad \quad \quad D[k] = C[j] \\
10 \quad \quad \quad j = j + 1 \\
11 \quad k = k + 1
\]
Time complexity of Merge

What is the time complexity of Merge? (Assume both $B$ and $C$ have size $m$.)

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$
- (O) Neither of these
What is the time complexity of Merge? (Assume both $B$ and $C$ have size $m$.)

- (R) $O(1)$
- (B) $O(m)$
- (G) $O(m^2)$
- (O) Neither of these
Mergesort

Merging

Assume both inputs have length $m$.

**MERGE**(B, C)

1. $B[m + 1] = C[m + 1] = \infty$
2. Initialize $D$ as empty array
3. $i = j = k = 1$
4. while $B[i] < \infty$ or $C[j] < \infty$
   5. if $B[i] < C[j]$
      6. $D[k] = B[i]$
      7. $i = i + 1$
   8. else
      9. $D[k] = C[j]$
      10. $j = j + 1$
11. $k = k + 1$
Mergesort

Time complexity of Merge

In every iteration of while loop, exactly one of $i$ or $j$ increments. Thus, $i + j$ increases by exactly one in each iteration. The initial value of $i + j$ is 2. The final value is $2m + 2$. So, the while loop runs for $2m$ iterations.

Alternately, in each iteration, we find one more element in sorted order. We do this $2m$ times to get the full answer.

The time complexity is $\Theta(m)$. 
Back to Mergesort

 Merge-Sort($A$)

1. $n = A.length$
2. **if** $n == 1$, **return** $A$
3. $L = A[1..n/2]$ // Technically, use $\lfloor n/2 \rfloor$
4. $R = A[n/2 + 1..n]$
5. $L_{\text{sort}} = \text{Merge-Sort}(L)$
6. $R_{\text{sort}} = \text{Merge-Sort}(R)$
7. **return** $\text{MERGE}(L_{\text{sort}}, R_{\text{sort}})$ // This works in $\Theta(n/2) = \Theta(n)$ time
Let’s run Merge-Sort on 8 4 3 7 1 2 6 5.
Recursion tree

- You just saw the **recursion tree** of the algorithm running on input
- Each **node** of the tree corresponds to a (recursive) run of Merge-Sort
- The **root** corresponds to the initial call, to 8 4 3 7 1 2 6 5.
- **Children** of a node are the calls made from that node. (Children on the root are the calls to 8 4 3 7 and 1 2 6 5.)
- **Leaves** are nodes that do not make recursive calls. They correspond to calls on singletons.
- **Depth** of node is the distance from root.
- All nodes at a fixed depth form a **level**.
Understanding the recursion tree

Start with Merge-Sort on array A with \( n \) elements. Assume \( n \) is a power of 2.
Understanding the recursion tree

Consider the recursion tree and look at a node at depth $i$. What is the input size for that call of Merge-Sort?

- (R) No reason why all nodes at depth $i$ have same input size
- (B) $i$
- (G) $n/i$
- (O) $n/2^i$
Understanding the recursion tree

Consider a node at depth $i$. What is the input size for that call of Merge-Sort?

- (R) No reason why all nodes at depth $i$ have same input size
- (B) $i$
- (G) $n/i$
- (O) $n/2^i$

Each recursive call decreases the input size by a factor of 2. At depth $i$, it reduces to $n/2^i$. 
Understanding the recursion tree

How many nodes at depth $i$ in the recursion tree?

- (R) $i + 1$
- (B) $2i$
- (G) $2^i$
- (O) $n/2i$
Understanding the recursion tree

How many nodes at depth \( i \) in the recursion tree?

- (R) \( i + 1 \)
- (B) \( 2i \)
- (G) \( 2^i \)
- (O) \( n/2^i \)

Each call to Merge-Sort (that is not a leaf) makes two recursive calls. The number of nodes doubles at each level. Thus, level \( i \) has \( 2^i \) nodes, where each has an input size of \( n/2^i \).
Understanding the recursion tree

What is the maximum depth of recursion tree?

- (R) $\log_2 n$
- (B) $\sqrt{2n}$
- (G) $n/2$
- (O) $n - 4$
Understanding the recursion tree

What is the maximum depth of recursion tree?

- (R) $\log_2 n$
- (B) $\sqrt{2n}$
- (G) $n/2$
- (O) $n - 4$

A node at depth $i$ has an input of size $n/2^i$. At depth $\log_2 n$, the size is 1. Such a node is a leaf.
Understanding the recursion tree

How many nodes in the recursion tree?

- (R) $3n/2 + 3$
- (B) $n$
- (G) $2n - 1$
- (O) $n^2/4 - 1$
Understanding the recursion tree

How many nodes in the recursion tree?

- (R) $3n/2 + 3$
- (B) $n$
- (G) $2n - 1$
- (O) $n^2/4 - 1$

Total number of nodes is

$$\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n+1} - 1 = 2n - 1$$
We understand the recursion tree

- It has $2n - 1$ nodes and maximum depth $\log_2 n$.
- There are $2^i$ nodes at depth $i$, and each such node corresponds to a call of Merge-Sort with input size $n/2^i$. 
Time-complexity analysis of Merge-Sort

- For each node, consider the run time of Merge within that node/call. Sum all of these for the total run time.
```plaintext
Mergesort

Back to Mergesort

**Merge-Sort(A)**

1. \( n = A.length \)
2. \( \text{if } n == 1, \text{ return } A \)
3. \( L = A[1..n/2] \)  // Technically, use \([n/2]\)
4. \( R = A[n/2 + 1, .. n] \)
5. \( L_{sort} = \text{Merge-Sort}(L) \)
6. \( R_{sort} = \text{Merge-Sort}(R) \)
7. \( \text{return } \text{Merge}(L_{sort}, R_{sort}) \)
```
Understanding the time complexity

What is the run time of Merge in a node at depth $i$?

- (R) $\Theta(i)$
- (B) $\Theta(2^i)$
- (G) $\Theta(n/2^i)$
- (O) $\Theta(n)$
Understanding the time complexity

What is the run time of Merge in a node at depth $i$?

- (R) $\Theta(i)$
- (B) $\Theta(2^i)$
- (G) $\Theta(n/2^i)$
- (O) $\Theta(n)$

The input size for a node at depth $i$ is $n/2^i$. The run time of Merge on two inputs of size $n/2^{i+1}$ is $\Theta(n/2^{i+1}) = \Theta(n/2^i)$. 
Understanding the time complexity

What is the total run time of Merges of nodes at depth $i$?

- (R) $\Theta(2^i)$
- (B) $\Theta(n/2^i)$
- (G) $\Theta(n)$
- (O) $\Theta(n \cdot i)$
Understanding the time complexity

What is the total run time of Merges of nodes at depth $i$?

- (R) $\Theta(2^i)$
- (B) $\Theta(n/2^i)$
- (G) $\Theta(n)$
- (O) $\Theta(n \cdot i)$

There are $2^i$ nodes at depth $i$, and the Merge in each of them takes $\Theta(n/2^i)$ time. Total time is $\Theta(2^i \times n/2^i) = \Theta(n)$. 
What is the total run time of Merge-Sort on an input of size $n$?

- (R) $\Theta(n)$
- (B) $\Theta(n \log_2 n)$
- (G) $\Theta(n^{1.5})$
Ergo

What is the total run time of Merge-Sort on an input of size $n$?

- (R) $\Theta(n)$
- (B) $\Theta(n \log_2 n)$
- (G) $\Theta(n^{1.5})$

The total run time of Merges at depth $i$ is $\Theta(n)$. The maximum depth is $\log_2 n$, each “level” contributes $\Theta(n)$. So the total is $\Theta(n \log_2 n)$. 
Time-complexity analysis of Merge-Sort

- For each node, consider the run time of Merge within that node/call. Sum all of these for the total run time.
- Let us break the nodes into groups at the same depth (also called levels).
- The Merge in a node at depth $i$ takes $\Theta(n/2^i)$ time. The total run time of all Merges at depth $i$ is $\Theta(n)$
- Thus, each level has a run time of $\Theta(n)$.
- There are $\log_2 n$ levels, so total run time is $\Theta(n \log_2 n)$.
Mergesort

Merge-Sort vs Insertion Sort

- Merge-Sort is $\Theta(n \log_2 n)$, Insertion-Sort is $\Theta(n^2)$.
- Consider input of size $n = 10^6$, so sorting 1 million numbers.
- Ignoring constant in $\Theta$, that’s $10^6 \log_2(10^6)$ vs $10^{12}$ operations.
What is it?

Give your estimate for $\log_2(10^6)$.

- (R) Between 1-5
- (B) Between 10-50
- (G) Between 100-500
- (O) Between 1000-5000
What is it?

Give your estimate for $\log_2(10^6)$.

- (R) Between 1-5
- (B) Between 10-50
- (G) Between 100-500
- (O) Between 1000-5000

\[
\log_2(10^6) = 6 \cdot \log_2(10)
\]

\[
3 < \log_2(10) < 4
\]

\[
\log_2(10^6) \approx 19.9
\]

(A useful trick: $2^{10}$ is \(1024\) \approx \(1000\).

Thus, $\log_2(1000) \approx 10$. So $\log_2(10^2) \approx 20$.)
What is it?

Give your estimate for \( \log_2(10^6) \).

- (R) Between 1-5
- (B) Between 10-50
- (G) Between 100-500
- (O) Between 1000-5000

\[
\log_2(10^6) = 6 \cdot \log_2(10)
\]

\[
3 < \log_2(10) < 4
\]

\[
\log_2(10^6) \approx 19.9
\]

(A useful trick: \( 2^{10} \) is 1024 \( \approx \) 1000. Thus, \( \log_2(1000) \approx 10 \). So \( \log_2(1000^2) \approx 20 \).)
Merge-Sort vs Insertion Sort

- Merge-Sort is $\Theta(n \log_2 n)$, Insertion-Sort is $\Theta(n^2)$.
- Consider input of size $n = 10^6$, so sorting 1 million numbers.
- Ignoring constant in $\Theta$, Merge-Sort takes 20 million operations. Insertion Sort takes 1 trillion operations.
Merge-Sort vs Insertion Sort

- Merge-Sort is $\Theta(n \log_2 n)$, Insertion-Sort is $\Theta(n^2)$.
- Consider input of size $n = 10^6$, so sorting 1 million numbers.
- Ignoring constant in $\Theta$, Merge-Sort takes 20 million operations. Insertion Sort takes 1 trillion operations.

Merge-Sort is awesome!