Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort
Definitions

Abstract Problem: defines a function from any allowable input to a corresponding output

Instance of a Problem: a specific input to abstract problem

Algorithm: well-defined computational procedure that takes an instance of a problem as input and produces the correct output

An Algorithm must halt on every input with correct output.
Algorithm Analysis

• Worst-case time complexity*
• (Worst-case) space complexity
• Average-case time complexity
Worst-Case Analysis

Two Techniques:
1. **Counts and Summations:**
   - Count number of steps from pseudocode and add
2. **Recurrence Relations:**
   - Use invariant, write down recurrence relation and solve it

We will use big-Oh notation to write down time and space complexity (for both worst-case & average-case analyses).

Compute the worst possible time of all input instances of length N.
• We say that
  ▪ \( F(n) = O(G(n)) \)

  If there exists two positive constants, \( c \) and \( n_0 \), such that
  ▪ For all \( n \geq n_0 \), we have \( F(n) \leq c \, G(n) \)

• Thus, to show that \( F(n) = O(G(n)) \), you need to find two positive constants that satisfy the condition mentioned above

• Also, to show that \( F(n) \neq O(G(n)) \), you need to show that for any value of \( c \), there does not exist a positive constant \( n_0 \) that satisfies the condition mentioned above
SelectionSort – Worst-case analysis

```
SelectionSort(array A)
1   N ← length[A]
2   for p ← 1 to N
3       do ▷ Compute j
4           j ← p
5           for m ← p + 1 to N
6               do if (A[m] < A[j])
7                   ▷ Swap A[p] and A[j]
8                       then j ← m
9               temp ← A[p]
11              A[j] ← temp
```

N-p comparisons
3 data movements
SelectionSort: Worst-Case Analysis

- **Data Movements**

\[
\sum_{p=1}^{N} 3 = 3 \times N = O(N)
\]

- **Number of Comparisons**

\[
\sum_{p=1}^{N} (N - p) \\
= \sum_{p=1}^{N} N - \sum_{p=1}^{N} p \\
= (N \times N) - (N)(N + 1)/2 \\
= O(N^2)
\]

- **Time Complexity** = \(O(N^2)\)
SelectionSort – Worst-case space analysis

1. **Temp Space**
   - No extra arrays or data structures
   - $O(1)$

2. **SelectionSort** (array $A$)

   1. $N \leftarrow \text{length}[A]$
   2. for $p \leftarrow 1$ to $N$
      
         do ▷ Compute $j$
         
         3. $j \leftarrow p$
         
         4. for $m \leftarrow p + 1$ to $N$
            
               
               5. then $j \leftarrow m$
               
               ▷ Swap $A[p]$ and $A[j]$
               
               6. $\text{temp} \leftarrow A[p]$
               
               
               8. $A[j] \leftarrow \text{temp}$
MergeSort

- Divide-and-Conquer Strategy
- Divide array into two sublists of roughly equal length
- Sort each sublist "recursively"
- Merge two sorted lists to get final sorted list
  - **Assumption**: Merging is faster than sorting from fresh
- Most of the work is done in merging
- Process described using a tree
  - **Top-down process**: Divide each list into 2 sublists
  - **Bottom-up process**: Merge two sorted sublists into one sorted sublist
Figure 2.4  The operation of merge sort on the array $A = (5, 2, 4, 7, 1, 3, 2, 6)$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.
Figure 2.3 The operation of lines 10–17 in the call \texttt{MERGE(A, 9, 12, 16)}, when the subarray \(A[9..16]\) contains the sequence \((2, 4, 5, 7, 1, 2, 3, 6)\). After copying and inserting sentinels, the array \(L\) contains \((2, 4, 5, 7, \infty)\), and the array \(R\) contains \((1, 2, 3, 6, \infty)\). Lightly shaded positions in \(A\) contain their final values, and lightly shaded positions in \(L\) and \(R\) contain values that have yet to be copied back into \(A\). Taken together, the lightly shaded positions always comprise the values originally in \(A[9..16]\), along with the two sentinels. Heavily shaded positions in \(A\) contain values that will be copied over, and heavily shaded positions in \(L\) and \(R\) contain values that have already been copied back into \(A\). (a)–(h) The arrays \(A, L,\) and \(R,\) and their respective indices \(k, i,\) and \(j\) prior to each iteration of the loop of lines 12–17. (i) The arrays and indices at termination. At this point, the subarray in \(A[9..16]\) is sorted, and the two sentinels in \(L\) and \(R\) are the only two elements in these arrays that have not been copied into \(A\).

Merge uses an extra array & lots of data movements.
**Assumption**: Array A is sorted from \([p..q]\) and from \([q+1..r]\).

**Space**: Two extra arrays L and R are used.

**Sentinel Items**: Two sentinel items placed in lists L and R.

**Merge**: The smaller of the item in L and item in R is moved to next location in A

**Time**: \(O(\text{length of lists})\)
**MergeSort**

```plaintext
MERGE-SORT(A, p, r)
1  if p < r
2    then q ← \(\lfloor (p + r)/2 \rfloor\)
3    MERGE-SORT(A, p, q)
4    MERGE-SORT(A, q + 1, r)
5    MERGE(A, p, q, r)
```

Time Complexity Recurrence: \(T(N) = 2T(N/2) + O(N)\)
<table>
<thead>
<tr>
<th>Recurrence; Cond</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n); a = b$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n); a &lt; b$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n); f(n) = O(n^{\log_b a - \epsilon})$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n); f(n) = O(n^{\log_b a})$</td>
<td>$T(n) = \Theta(n^{\log_b a \log n})$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n); f(n) = \Theta(f(n)); af(n/b) \leq cf(n)$</td>
<td>$T(n) = \Omega(n^{\log_b a \log n})$</td>
</tr>
</tbody>
</table>
Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available.
- Recursion-tree method works in those cases:
  - Write down the recurrence as a tree with recursive calls as the children.
  - Expand the children.
  - Add up each level.
  - Sum up the levels.
- Useful for analyzing divide-and-conquer algorithms.
- Also useful for generating good guesses to be used by substitution method.
Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part (a) shows $T(n)$, which is progressively expanded in (b)-(d) to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of $cn$. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$. 
Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$.

Part (a) shows $T(n)$, which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).
Figure 4.2  A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$. 

Total: $O(n \log n)$
Solving Recurrences using Master Theorem

**Master Theorem:**

Let $a,b \geq 1$ be constants, let $f(n)$ be a function, and let

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a-e})$ for some constant $e>0$, then
   $$T(n) = \Theta(n^{\log_b a})$$

2. If $f(n) = \Theta(n^{\log_b a})$, then
   $$T(n) = \Theta(n^{\log_b a \log n})$$

3. If $f(n) = \Omega(n^{\log_b a+e})$ for some constant $e>0$, then
   $$T(n) = \Theta(f(n))$$
Solving Recurrences by Substitution

- Guess the form of the solution
- (Using mathematical induction) find the constants and show that the solution works

**Example**

\[ T(n) = 2T(n/2) + n \]

**Guess (#1)** \( T(n) = O(n) \)

**Need** \( T(n) \leq cn \) for some constant \( c > 0 \)

**Assume** \( T(n/2) \leq cn/2 \) Inductive hypothesis

**Thus** \( T(n) \leq 2cn/2 + n = (c+1)n \)

*Our guess was wrong!!*
Solving Recurrences by **Substitution**: 2

\[ T(n) = 2T(n/2) + n \]

**Guess (\#2)** \[ T(n) = O(n^2) \]

**Need** \[ T(n) \leq cn^2 \] for some constant \( c > 0 \)

**Assume** \[ T(n/2) \leq cn^2/4 \] **Inductive hypothesis**

**Thus** \[ T(n) \leq 2cn^2/4 + n = cn^2/2 + n \]

Works for all \( n \) as long as \( c \geq 2 \) !!

But there is a lot of “slack”
Solving Recurrences by Substitution: 3

\[ T(n) = 2T(n/2) + n \]

Guess (#3) \[ T(n) = O(n \log n) \]

Need \[ T(n) \leq cn \log n \] for some constant \( c > 0 \)

Assume \[ T(n/2) \leq c(n/2)(\log(n/2)) \] \( \text{Inductive hypothesis} \)

Thus \[ T(n) \leq 2c(n/2)(\log(n/2)) + n \]

\[ \leq cn \log n - cn + n \leq cn \log n \]

Works for all \( n \) as long as \( c \geq 1 \) !!

This is the correct guess. WHY?

Show \[ T(n) \geq c'n \log n \] for some constant \( c' > 0 \)
## Solving Recurrence Relations

<table>
<thead>
<tr>
<th>Recurrence; Cond</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a = b$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a &lt; b$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$</td>
<td>$T(n) = \Theta(n^{\log_b a \log n})$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$; $af(n/b) \leq cf(n)$</td>
<td>$T(n) = \Omega(n^{\log_b a \log n})$</td>
</tr>
</tbody>
</table>
QuickSort

MergeSort
- Divide into 2 equal sublists
- Sort each sublist “recursively”
- Merge 2 sorted sublists
  - **Assumption**: Merging is faster than sorting from fresh
- Most of work is done in merging

QuickSort
- *Partition* into 2 sublists using a pivot
- Sort each sublist “recursively”
- **Concatenate** 2 sorted sublists
  - **Assumption**: Partition is faster than sorting
- Most of work is done in *partition*
- Process described using a tree
  - **Top-down process**: Partition each list into 2 sublists
  - **Bottom-up process**: Concatenate two sorted sublists into one sorted sublist
Figure 8.10  Quicksort

Select pivot

Partition

Quicksort small items

Quicksort large items
Partition Algorithm

• Pick a pivot
• Compare each item to a pivot and create two lists:
  ▪ L = list of all items smaller than the pivot
  ▪ R = list of all items larger than the pivot
• One scan through the list is enough, but seems to need extra space
• How to design an in-place partition algorithm!

O(N) time
Figure A  If 6 is used as pivot, the end result after partitioning is as shown in the Figure B.

```
2 1 4 5 0 3 9 8 7 6
```

Figure B  Result after Partitioning

```
2 1 4 5 0 3 6 8 7 9
```
QuickSort

QuickSort(array A, int p, int r)
1   if (p < r)
2       then q ← Partition(A, p, r)
3       QuickSort(A, p, q − 1)
4       QuickSort(A, q + 1, r)

To sort array call QuickSort(A, 1, length[A]).

Partition(array A, int p, int r)
1   x ← A[r]                           ▷ Choose pivot
2   i ← p − 1
3   for j ← p to r − 1
4       do if (A[j] ≤ x)
5           then i ← i + 1
7   exchange A[i + 1] ← A[r]
8   return i + 1
Time Complexity

- \( T(N) = O(N) + T(N_1) + T(N_2) \)
- On the average, \( N_1 = N_2 = N/2 \)
- Thus, average-case complexity = \( O(N \log N) \)
- Worst-case: Either \( N_1 \) or \( N_2 = 0 \)
  - Thus, \( T(N) = O(N) + T(N - 1) \)
  - \( T(N) = O(N^2) \)