Sorting

Sorting Problem

**Input:** An array of *comparable* elements

**Output:** The same elements, sorted in ascending order

- One of the most well-studied algorithmic problems
- Has lots of practical applications
- You should already know a few algorithms...

### SelectionSort

```python
def selectionSort(A):
    for i in range(0, len(A) - 1):
        m = i
        for j in range(i+1, len(A)):
            if A[j] < A[m]:
                m = j
        swap(A, i, m)
```

### InsertionSort

```python
def insertionSort(A):
    for i in range(1, len(A)):
        j = i - 1
        while j >= 0 and A[j] > A[j+1]:
            swap(A, j, j+1)
            j = j - 1
```
Common Features

It’s useful to look for larger patterns in **algorithm design**.

Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:

- **Pick**: Pick an element in the unsorted part of the array
- **Place**: Insert that element into the sorted part of the array

For both algorithms, one of these is “easy” (constant time) and the other is “hard” \(O(n)\) time. Which ones?

**Analysis of SelectionSort**

Each loop has \(O(n)\) iterations, so the total cost is \(O(n^2)\).

What about a big-\(\Theta\) bound?

**Arithmetic Series**

An arithmetic series is one where consecutive terms differ by a constant.

General formula:

\[
\sum_{i=0}^{m} (a + bi) = \frac{(m + 1)(2a + bm)}{2}
\]

So the worst-case of SelectionSort is

This is \(\Theta(n^2)\), or **quadratic time**.
Worst-Case Family

Why can’t we analyze InsertionSort in the same way?

We need a family of examples, of arbitrarily large size, that demonstrate the worst case.

Worst-case for InsertionSort:

Worst-case cost:

SelectionSort (Recursive Version)

def selectionSortRec(A, start=0):
    if (start < len(A) - 1):
        m = minIndex(A, start)
        swap(A, start, m)
        selectionSortRec(A, start + 1)

minIndex

def minIndex(A, start=0):
    if start >= len(A) - 1:
        return start
    else:
        m = minIndex(A, start+1)
        if A[start] < A[m]:
            return start
        else:
            return m

Analysis of minIndex

Let $T(n)$ be the worst-case number of operations for a size-$n$ input array.

We need a recurrence relation to define $T(n)$:

$$ T(n) = \begin{cases} 
1, & n \leq 1 \\
4 + T(n-1), & n \geq 2 
\end{cases} $$

Solving the recurrence:
Analysis of recursive SelectionSort

Let $S(n)$ be the worst-case for SelectionSort

What is the recurrence?
**Merge**

```python
def merge(B, C):
    A = []
i, j = 0, 0
    while i < len(B) and j < len(C):
        if B[i] <= C[j]:
            A.append(B[i])
            i = i + 1
        else:
            A.append(C[j])
            j = j + 1
    while i < len(B):
        A.append(B[i])
        i = i + 1
    while j < len(C):
        A.append(C[j])
        j = j + 1
    return A
```

**Analysis of Merge**

Each while loop has constant cost.
So we just need the total number of iterations through every loop.

<table>
<thead>
<tr>
<th></th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop 1</td>
<td>min(a, b)</td>
<td>a + b</td>
<td></td>
</tr>
<tr>
<td>Loop 2</td>
<td>0</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>Loop 3</td>
<td>0</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>min(a, b)</td>
<td>2(a + b)</td>
<td></td>
</tr>
</tbody>
</table>

\(a\) is the size of A and \(b\) is the size of B.

**Analysis of MergeSort**
Lower Bound for Sorting

Complexity of Sorting

Algorithms we have seen so far:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst-case cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$\Theta(n \log n)$</td>
</tr>
</tbody>
</table>

Million dollar question: Can we do better than $\Theta(n \log n)$?

---

Comparison Model

Elements in the input array can only be accessed in two ways:
- Moving them (swap, copy, etc.)
- Comparing two of them ($<$, $>$, $=$, etc.)

Every sorting algorithm we have seen uses this model. It is a very general model for sorting strings or integers or floats or anything else.

What operations are not allowed in this model?

---

Permutations

How many orderings (aka permutations) are there of $n$ elements?

$n$ factorial, written $n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$.

Observation: A comparison-based sort is only sensitive to the ordering of $A$, not the actual contents.

For example, MergeSort will do the same things on $[1, 2, 4, 3], [34, 36, 37, 36]$, or $[10, 20, 200, 99]$. 

---
Logarithms
Recall some useful facts about logarithms:
- \( \log_b b = 1 \)
- \( \log_b ac = \log_b a + \log_b c \)
- \( \log_b a^c = c \log_b a \)
- \( \log_b a = \frac{(\log_c a)}{(\log_c b)} \)

Now how about a lower bound on \( \lg n \)?

Lower Bound on Sorting

- A correct algorithm must take different actions for each of the possible input permutations.
- The choice of actions is determined only by comparisons.
- Each comparison has two outcomes.
- An algorithm that performs \( c \) comparisons can only take \( 2^c \) different actions.
- The algorithm must perform at least \( \lg n! \) comparisons.

Therefore... **ANY comparison-based sort is** \( \Omega(n \log n) \)

Conclusions

Any sorting algorithm that only uses comparisons must take at least \( \Omega(n \log n) \) steps in the worst case.

- This means that sorts like MergeSort and HeapSort couldn’t be much better — they are **asymptotically optimal**.
- What if I claimed to have a \( O(n) \) sorting algorithm? What would that tell you about my algorithm (or about me)?
- Remember what we learned about **summations**, **recursive algorithm analysis**, and **logarithms**.