The Sorting Problem

• **Problem.** Given a set of numbers, sort the numbers in increasing order.
  
  ◦ **Example motivation.** To be able to use binary search, we first need to sort the set.

• **Warm-up.** Can you come up with naïve inefficient approaches for sorting?
Selection Sort

- Receive set $S$ as input.
- Create empty set $T$.
- **Repeat** until $S$ is empty:
  - Find smallest element $x$ in $S$.
  - Remove $x$ from $S$.
  - Place $x$ at the end of $T$.

<table>
<thead>
<tr>
<th>$S$</th>
<th>11</th>
<th>23</th>
<th>3</th>
<th>7</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>11</th>
<th>23</th>
<th>7</th>
<th>5</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>11</th>
<th>23</th>
<th>7</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>3</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>11</th>
<th>23</th>
<th>-</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Running Time of Selection Sort

- \( n \) – initial size of the set of numbers \( S \).
- **Running time:**
  - Creating an empty \( T \): \( O(1) \).
  - Finding smallest element in set of \( m \) numbers: \( O(m) \).
  - \( j \)'th iteration of loop takes \( O(n - j + 1) \).
  - \( n \) steps each taking \( O(n) \) lead to \( O(n^2) \).
  - Total running time: \( O(n^2) \).
- Is \( O(n^2) \) tight? Can we also show a running time of \( O(n^{1.9}) \)?
A Tight Bound

- We proved that the running time of selection sort is $O(n^2)$.
- Each of the first $n/2$ iterations of the loop take at least $n/2$ steps. So selection sort has at least $n^2/4$ steps.
- The bound $O(n^2)$ is tight!

Remark about Tight Bounds

- Selection sort takes about $n^2$ steps for every set of $n$ numbers.
- Recall. The running time analysis is for the worst possible input.
  - It is enough to show that the algorithm requires about $n^2$ steps for one bad input.
Historical Anecdotes

- A study in the 1960s estimated that over a quarter of the world’s computing power is spent on sorting.
- The first code written for a “stored program” was for sorting.

John von Neumann

Faster Sorting

- Several sorting algorithms are much more efficient than selection sort.
  - We will study merge sort.
- Recursive algorithms.
  - If the input is small, solve it naively.
  - Otherwise, reduce problem to one or more smaller instances of the same problem.
Factorial Recursive Algorithm

- **Recall.** For integer \( n \geq 1 \), the factorial of \( n \) is \( n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1 \).
- **Problem.** Describe a recursive algorithm that receives \( n \) and returns \( n! \).

```python
def factorial(n):
    if n==1:
        return 1
    else:
        return n * factorial(n-1)
```

Towards Merge Sort

- **Merge sort** is a recursive algorithm.
  - We split the list into two.
  - Recursively sort each shorter list.
  - Merge both sorted lists.
Towards Merge Sort

- **Problem.**
  - Input: two sorted lists $S, T$.
  - Goal: merge both lists into one sorted list.
  - How can we do this efficiently?

```
<table>
<thead>
<tr>
<th>Sorted sequence</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 2 3 4 5 6 6</td>
<td></td>
</tr>
<tr>
<td>2 4 5 6</td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>3 6</td>
</tr>
</tbody>
</table>
```

Merging Sorted Lists

- Receive sorted Lists $S, T$.
- Create empty set $R$.
- **Repeat** until either $S$ or $T$ is empty.
  - $x$ – smallest element in $S$ and $T$.
  - **Remove $x$ from the list** it came from (if $x$ is smallest in both sets, arbitrarily choose one).
  - Add $x$ to the end of $R$.
- Add non-empty set to the end of $R$. 

```
<table>
<thead>
<tr>
<th>Sorted sequence</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 2 3 4 5 6 6</td>
<td></td>
</tr>
<tr>
<td>2 4 5 6</td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>3 6</td>
</tr>
</tbody>
</table>
```
• **Repeat** until either $S$ or $T$ is empty.
  ◦ $x$ is the **smallest element** in $S$ and $T$.
  ◦ **Remove** $x$ from the set it came from (if $x$ is the smallest in both sets, remove it from one).
  ◦ Add $x$ to the end of $R$.

<table>
<thead>
<tr>
<th>$S$</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>14</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>$R$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>$S$</th>
<th>3</th>
<th>8</th>
<th>14</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>$R$</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>8</th>
<th>14</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>3</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>$R$</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>8</th>
<th>14</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>5</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>$R$</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>8</th>
<th>14</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>$R$</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Merge Running Time

- $n$ – size of $S$.  \hspace{1em} m – size of $T$.

- **Running time:**
  - Creating empty $R$: $O(1)$.
  - Max number of steps in loop: $m + n - 1$.
  - Single step of the loop: $O(1)$. We only compare the first elements of $S$ and $T$.
  - Eventually adding the non-empty set to $R$: $O(m + n)$.
  - Total running time $O(m + n)$.

- Can we do better than $O(m + n)$?

A Tight Bound

- We merged two lists with a running time of $O(m + n)$.
- **Reading the input** requires at $m + n$ steps.
- The bound $O(m + n)$ is the best possible.
Merge Sort Algorithm

- Receive set of numbers $S$.
- If $S$ contains only one number, return $S$.
- Otherwise:
  - $S_1$ – set containing first $\lceil n/2 \rceil$ numbers of $S$.
  - $S_2$ – set containing last $\lceil n/2 \rceil$ numbers of $S$.
  - Apply merge sort recursively on $S_1$.
  - Apply merge sort recursively on $S_2$.
  - Use previous algorithm to merge the two sorted lists.
Moment of Nonsense

- Merge-sort with Transylvanian-Saxon folk dance:
  https://www.youtube.com/watch?v=XaqR3G_NVoo

Run Merge Sort

- Draw the flow of the sorting algorithm for the list \([14,5,1,2,4]\).
Merge Sort Running Time

- $T(m)$ – running time of merge sort on $m$ numbers.
  - $T(1) = O(1)$.
  - $T(n) = T([n/2]) + T([n/2]) + O(n)$.
  - For simplicity, we ignore floors and ceilings (these make no difference asymptotically).
    $$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n).$$
  - This is called a recurrence relation.

Solving the Recurrence Relation

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

$$= 2 \left( T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right) \right) + O(n)$$

$$= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot O\left(\frac{n}{2}\right) + O(n)$$

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)$$
Solving the Recurrence Relation

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \]

\[ = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot O\left(\frac{n}{2}\right) + O(n) \]

\[ = 8 \cdot T\left(\frac{n}{8}\right) + 4 \cdot O\left(\frac{n}{4}\right) + 2 \cdot O\left(\frac{n}{2}\right) + O(n) \]

\[ = O(n) + O(n) + \cdots + O(n). \]

• How many terms are in this sum?

Number of Terms

\[ T(n) = O(n) + 2 \cdot O\left(\frac{n}{2}\right) + 4 \cdot O\left(\frac{n}{4}\right) \]

\[ + 8 \cdot O\left(\frac{n}{8}\right) + \cdots + n \cdot O(1). \]

• How many terms are in this sum?
  ◦ The number of times we need to divide \( n \) by 2 until we get to 1.
  ◦ The number of terms is \( \log_2 n \).
The Running Time

\[ T(n) = 2 \cdot T \left( \frac{n}{2} \right) + O(n) \]

\[ = 4 \cdot T \left( \frac{n}{4} \right) + 2 \cdot O \left( \frac{n}{2} \right) + O(n) \]

\[ = 8 \cdot T \left( \frac{n}{8} \right) + 4 \cdot O \left( \frac{n}{4} \right) + 2 \cdot O \left( \frac{n}{2} \right) + O(n) \]

\[ = \cdots = \log_2 n \cdot O(n) = O(n \log n). \]

- The running time of **merge sort** is \( O(n \log n) \).
  - Much better than the \( O(n^2) \) of **selection sort**!

Intuition

- More intuition for the \( O(n \log n) \) running time:
  - There are \( O(\log n) \) levels in the diagram.
  - Each level requires \( O(n) \) steps.
How Does Python Sort?

- When asking Python to sort a list, it uses an algorithm called Timsort.
  - Similar to merge sort, but with a few extra tricks.
  - Worst-case running time remains $O(n \log n)$.
  - Runs faster for various real-world cases (for example, when parts of the list are already sorted).
  - Also used in Java, on the Android platform, and more.

Exercise: Recurrence Relation

- $T(m)$ – running time of algorithm $X$ on input of size $m$.
- We know that (ignoring floors/ceilings):
  - $T(1) = O(1)$.
  - $T(n) \leq T\left(\frac{n}{2}\right) + n$.
- **Problem.** Find the running time of algorithm $X$. 
Exercise: Recurrence Relation

- \( T(m) \) – running time of algorithm \( X \) on input of size \( m \).
- We know that (ignoring floors/ceilings):
  - \( T(1) = O(1) \).
  - \( T(n) \leq T \left( \frac{n}{2} \right) + n \).
  - \( T(n) = T \left( \frac{n}{2} \right) + n = T \left( \frac{n}{4} \right) + n + \frac{n}{2} = \ldots \)
  - \( = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + O(1) \)

Recall: Geometric Series

- (From Calculus) A geometric series is
  \[ \sum_{n=0}^{\infty} a \cdot r^n. \]
- Examples.
  - \( 1 + 2 + 4 + 8 + 16 + 32 + \ldots \)
  - \( \frac{4}{5} + \left( \frac{4}{5} \right)^2 + \left( \frac{4}{5} \right)^3 + \left( \frac{4}{5} \right)^4 + \ldots \)
- If \( |r| < 1 \) then
  \[ \sum_{n=0}^{\infty} a \cdot r^n = \frac{a}{1 - r}. \]
Geometric Sums

- If $|r| < 1$ then
  \[ \sum_{n=0}^{\infty} a \cdot r^n = \frac{a}{1 - r}. \]

- $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots$ is a geometric series with $a = n$ and $r = 1/2$.
  - The sum is thus smaller than $\frac{n}{1 - 1/2} = 2n$.

Exercise: Recurrence Relation

- $T(m)$ – running time of algorithm $X$ on input of size $m$.
- We know that (ignoring floors/ceilings):
  - $T(1) = O(1)$.
  - $T(n) \leq T\left(\frac{n}{2}\right) + n$.

\[
T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + n + \frac{n}{2} = \cdots \\
= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + O(1) < 2n + O(1) \\
= O(n).
\]
Group Exercise

• Problem. Design an efficient algorithm that receives a set of numbers and sorts them as follows:
  ◦ First the **odd** numbers appear in increasing order, and then the **even** numbers appear in decreasing order.
  ◦ What is the running time of your algorithm?

\[ T = 8 \ 1 \ 5 \ 11 \ 12 \ 6 \]

\[ T' = 1 \ 5 \ 11 \ 12 \ 8 \ 6 \]

Group Exercise

• Problem.
  • Design an efficient algorithm that receives a **set** of numbers \( S \).
    ◦ The algorithm checks if there are **two** numbers in \( S \) whose sum is 5,000.
    ◦ The algorithm returns “yes” or “no” accordingly.
    ◦ What is the running time?
    ◦ (as a **warmup** you can start with a simple algorithm that is not so efficient)
Divide and Conquer

- **Divide and Conquer.** Splitting a problem into several smaller problem and handling each sub-problem recursively.
  - Merge sort relies on the divide and conquer approach.

Buying and Selling Stock

- **Problem.** We are given the price of a stock at the beginning of every day, for \( n \) days.
  - Find the maximum profit possible when buying one unit at the beginning of one day and then selling it at the end of a day.
Stock Problem: Naïve Algorithm

- **Naïve algorithm.**
  - Go over every possible starting day.
  - For each starting day, check every ending day.
  - While doing the above process, keep the largest profit found so far.
  - After going over every option, return the maximum profit.

How many pairs of days are we checking?
- < \( n^2 \) pairs. Since there are \( n \) choices for the buying day and \( n \) choices for the selling day.
- (If you know basic combinatorics, the exact number of pairs is \( \binom{n+1}{2} = \frac{n^2+n}{2} \).)
Stock Problem: Naïve Algorithm

- **Naïve algorithm.**
  - Go over **every possible starting day**.
  - For each starting day, check **every ending day**.
  - While doing the above process, **keep the maximum profit** found so far.

- **Running time.**
  - Pairs of days to check: $O(n^2)$.
  - Each check: $O(1)$.
  - Total: $O(n^2)$.

Group Exercise

- **Problem.** Discuss in groups.
  - How could you use the **divide and conquer approach** for the stock problem?
  - **Hint.** Split list of price changes into two.
Stock Problem: Divide and Conquer

- **Divide and conquer** approach:
  - If list contains one number, solve naïvely.
  - Otherwise, cut list of changes into two.
    - $L_1$ – list of *first* $\lfloor n/2 \rfloor$ changes.
    - $L_2$ – list of *last* $\lfloor n/2 \rfloor$ changes.
  - **Three options** for interval of maximum profit:
    - fully in $L_1$, fully in $L_2$, or starting in $L_1$ and ending in $L_2$.

Handling Three Options

- $L_1$ – list of *first* $\lfloor n/2 \rfloor$ changes.
- $L_2$ – list of *last* $\lfloor n/2 \rfloor$ changes.
- **Three options** for interval of maximum profit:
  - fully in $L_1$, fully in $L_2$, or starting in $L_1$ and ending in $L_2$.
    - **Recursively run** algorithm on $L_1$.
    - **Recursively run** algorithm on $L_2$.
    - Find optimal interval starting in $L_1$ and ending in $L_2$. **How?**
    - Return best out of three intervals.
Handling the Third Case

- **Problem.** Find max profit when buying in \( L_1 \) and selling in \( L_2 \).
  - Find the **minimum** value at the **beginning of a day** in \( L_1 \).
  - Find the **maximum** value at the **end of a day** in \( L_2 \).
  - Running time: \( \mathcal{O}(n) \).

Stock Problem Running Time

- **\( T(m) \)** – running time of algorithm with \( m \) days.
  - \( T(1) = \mathcal{O}(1) \).
  - \( T(n) = 2 \cdot T \left( \frac{n}{2} \right) + \mathcal{O}(n) \).
  - This is the same recurrence relation as for merge sort. So \( \mathcal{O}(n \log n) \).
Doing Better

- There is a simple algorithm that solves the stock problem in time $O(n)$.
  - If you want a challenge, try coming up with such an algorithm (not during class).
- Is there also a sorting algorithm with a running time faster than $O(n \log n)$?

![Graph](image)

Lower Bounds

- Functions $f(n)$ and $g(n)$ represent running times.
  - $f(n) = \Omega(g(n))$ means $g(n) = O(f(n))$.
  - Intuitively, $f(n)$ grows at least as fast as $g(n)$.
  - That is, exist $c, d > 0$ such that $f(n) \geq c \cdot g(n)$ for every $n \geq d$.
- **True** or **False**?
  - $30n + 4 = \Omega(n)$.
  - **True**. For example, by setting $c = 1$ and $d = 1$. 

47

48
Lower Bounds

- Functions $f(n)$ and $g(n)$ represent running times.
  - $f(n) = \Omega(g(n))$ means $g(n) = O(f(n))$.
  - Intuitively, $f(n)$ grows at least as fast as $g(n)$.
  - That is, exist $c, d > 0$ such that $f(n) \geq c \cdot g(n)$ for every $n \geq d$.

True or False?

- $30n + 4 = \Omega(n \log n)$.
- $n = \Omega(2^{\log_2 (\log_2 n)})$.
- $500 = \Omega(1)$.

Lower Bounds for Sorting

- We know how to sort $n$ numbers with running time $O(n \log n)$.
  - Can we do better?
- What is a simple lower bound for running time of any sorting algorithm?
  - Reading the entire input takes at least $n$ steps, so running time is $\Omega(n)$.
A Tight Bound for Sorting

- The running time of every sorting algorithm is $\Omega(n \log n)$.
  - Merge sort is best possible!
  - We now prove this claim.
  - We first show how to describe a sorting algorithm using a decision tree.

Decision Trees

- We rewrite a sorting alg as a decision tree:
  - Sorting the numbers $a_1, a_2, \ldots, a_n$.
  - We start from the top, and travel down.
  - At each node two numbers are compared, and we decide where to go by the result.
  - Eventually we get to a “leaf” containing an ordering.
Decision Tree Example

- Consider the numbers
  \[ a_1 = 5, \quad a_2 = 8, \quad a_3 = 3. \]

\[ a_1 \leq a_2 \]
\[ a_2 > a_3 \]
\[ a_1 > a_3 \]
\[ a_3 \leq a_1 \leq a_2 \]

Using a Decision Tree

- Apply the decision tree on
  \[ a_1 = 5, \quad a_2 = -3, \quad a_3 = 4. \]
Decision Tree Height

- The number of ways to order $n$ elements is $n!$.
- Every order may happen, so the decision tree contains at least $n!$ leaves.
- **Height of tree** = number of levels in tree.
  - **Example.** Decision tree in figure is of height 4.
- The **running time of a sorting algorithm** is at least the height of its decision tree.

Leaves VS Levels

- A tree with **one level** has at most **one leaf**.
- A tree with **two levels** has at most **two leaves**.
- A tree with **three levels** has at most **four leaves**.
- A tree with **four levels** has at most **eight leaves**.
- A tree with $h$ levels has at most $2^{h-1}$ leaves.
Decision Tree Height (cont.)

- Every ordering may happen, so the decision tree contains at least \( n! \) leaves.
- Height of tree = number of levels.
- A decision tree of height \( h \) has at most \( 2^{h-1} \) leaves.
- What is the min height of a tree containing \( n! \) leaves?
- We need \( 2^{h-1} \geq n! \)
  or \( h - 1 \geq \log_2(n!) \)
  or \( h \geq \log_2(n!) + 1 \)

Minimum Height

- We need \( 2^{h-1} \geq n! \)
  or \( h \geq \log_2(n!) + 1 \)
- \( n! > \left( \frac{n}{2} \right)^{n/2} \), so
  \[
  \log_2(n!) > \log_2 \left( \frac{n}{2} \right)^{n/2} = \frac{n}{2} \log_2 \left( \frac{n}{2} \right)
  \]
- The tree height is \( \Omega(n \log n) \).
- Running time of every sorting algorithm is \( \Omega(n \log n) \).
Group Exercise

• **Problem.** Discuss in groups.
  ◦ Explain why the decision tree lower bound proof fails for the stock trading problem.

The End

“No, we’re just learning how to divide. When you get to business school, you’ll learn how to divide and conquer.”