Divide and Conquer

Lecture 13

by Marina Barsky

https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting
Warmup: Recursive array max

• Design a recursive algorithm called `findArrayMax` that returns the maximum value in an array

• Formally:
  
  **Input:** array A of length n >= 1  
  **Output:** max value of A

• Examples:  

  **Input:** A = \{4, 13, 21, 5, 2\}  
  **Output:** 21

  **Input:** A =\{-1, -3, -8, -5, -12\}  
  **Output:** -1

  **Input:** A = \{5\}  
  **Output:** 5
Recursive array max: stop condition

Algorithm `findArrayMax(A)`:

- **input:** a NONEMPTY array, A
- **output:** A's maximum element

Stop condition?

A

- if A.length == 1
  - return 0

B

- if A.length == 1
  - return 1

C

- if A.length == 1
  - return A[0]

- A
- B
- C
- None of the above
- More than one is correct
Recursive array max: recursive step

Algorithm `findArrayMax(A)`:

```python
if A.length == 1:
    return A[0]
```

  ```python
  A = A - A[0]
  return findArrayMax(A)
  ```
- else:
  ```python
  A = A - A[1]
  return findArrayMax(A)
  ```

- A
- B
- None of the above
Recursive array max: solution

**Algorithm** `findArrayMax(A)`:

- **input**: a NONEMPTY array, A
- **output**: A's maximum element

  if `A.length == 1`:
  
  return `A[0]`

  
  `A = A - A[0]`
  
  return `findArrayMax(A)`

  else:
  
  `A = A - A[1]`
  
  return `findArrayMax(A)`
# Sorting Problem

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sequence $A$ of $n$ elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Permutation $A'$ of elements in $A$ such that all elements of $A'$ are in non-decreasing order.</td>
</tr>
</tbody>
</table>
Sorting Problem

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms
Why Sorting?

• Sorting data is an important step of many efficient algorithms

• Sorted data allows for more efficient queries (binary search)
We will use Divide-and-conquer technique

1. Break into *non-overlapping* subproblems *of the same type*
2. Solve subproblems
3. Combine results
Divide: break apart
Conquer: solve subproblems
Idea: merge sort

split the array into two halves

```
7 2 5 3
```

```
7 13 1 6
```
Idea: merge sort

split the array into two halves

sort the halves recursively

7 2 5 3 7 13 1 6

7 2 5 3

7 13 1 6

2 3 5 7

1 6 7 13
Idea: merge sort

split the array into two halves

merge the sorted halves into one array
Algorithm MergeSort (array $A[1...n]$)

if $n = 1$: return $A$  # already sorted

$m \leftarrow \lfloor n/2 \rfloor$
$B \leftarrow \text{MergeSort}(A[1 ... m])$
$C \leftarrow \text{MergeSort}(A[m + 1 ... n])$
$A' \leftarrow \text{merge}(B, C)$
return $A'$
Merging Two Sorted Arrays

Algorithm Merge($B[1...p]$, $C[1...q]$)

# $B$ and $C$ are sorted
$D \leftarrow$ empty array of size $p + q$
while $B$ and $C$ are both non-empty:
  $b \leftarrow$ the first element of $B$
  $c \leftarrow$ the first element of $C$
  if $b \leq c$:
    move $b$ from $B$ to the end of $D$
  else:
    move $c$ from $C$ to the end of $D$
move what remains of $B$ or $C$ to the end of $D$
return $D$
Merge sort: example

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>13</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>
Merge sort: example

7 2 5 3 7

7 2 5 3

7 2 5 3

7 13 1 6

7 13 1 6

7 13 1 6
Merge sort: example

\[
\begin{array}{cccccc}
7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \\
7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \\
7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \\
7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \\
7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \\
\end{array}
\]
Merge sort: example
Merge sort: example

7  2  5  3  7  13  1  6

7  2  5  3      7  13  1  6
7  2      5  3  7  13      1  6
7  2  5  3      7  13  1  6
2  7  3  5  7      1  6  7  13
### Merge sort: example

```
<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>13</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>
```
Merge: example

\[
\begin{array}{c}
\text{B} \\
\begin{array}{cccc}
2 & 3 & 5 & 7 \\
i
\end{array} \\
\end{array}
\quad \begin{array}{c}
\text{C} \\
\begin{array}{cccc}
1 & 6 & 7 & 13 \\
j
\end{array} \\
\end{array}
\]

Compare \( B[i] \) and \( C[j] \)

\[
\begin{array}{c}
\text{D} \\
\end{array}
\quad \begin{array}{cccc}
\text{ } & \text{ } & \text{ } & \text{ } \\
k
\end{array}
\]
**Merge: example**

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 5 7</td>
<td>1 6 7 13</td>
</tr>
</tbody>
</table>

$i$  

$\text{Compare } B[i] \text{ and } C[j]$  

<table>
<thead>
<tr>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

$k$
Merge: example

\[
\begin{align*}
\text{B} & : \begin{array}{cccc}
2 & 3 & 5 & 7 \\
\end{array} \\
\text{C} & : \begin{array}{cccc}
1 & 6 & 7 & 13 \\
\end{array} \\
\text{D} & : \begin{array}{ccc}
1 & 2 & \\
\end{array}
\end{align*}
\]

Compare \( B[i] \) and \( C[j] \)

\( k \)
Merge: example

Compare $B[i]$ and $C[j]$

k
Merge: example

\[ B \]

\[ \begin{array}{cccc}
2 & 3 & 5 & 7 \\
\end{array} \]

\[ C \]

\[ \begin{array}{cccc}
1 & 6 & 7 & 13 \\
\end{array} \]

Compare \( B[i] \) and \( C[j] \)

\[ D \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 5 \\
\end{array} \]
Merge: example

\[ \begin{array}{c}
\text{B} \\
2 & 3 & 5 & 7 \\
\end{array} \quad \begin{array}{c}
\text{C} \\
1 & 6 & 7 & 13 \\
\end{array} \]

\[ \text{D} \]
\[ 1 & 2 & 3 & 5 & 6 \]

Compare \( B[i] \) and \( C[j] \)
Merge: example

Copy what remains in C

D

k
Merge: example

B
2 3 5 7

C
1 6 7 13

D
1 2 3 5 6 7 7 13
Merge sort: running time

Subproblem size at each level

```
1 1 1 1
```

```
 n/8 n/8 n/8 n/8 n/8 n/8 n/8 n/8
```

```
 n/4 n/4 n/4 n/4
```

```
 n/2 n/2
```

```
 n
```
Merge sort: recursion tree

The height of the tree is...
Merge sort: recursion tree

The height of the tree is $\log n$
Merge sort: recursion tree

Work at each level: all the work during *merge*

![Recursion tree diagram](image-url)
Merge sort: recursion tree

Work at each level: $O(n)$

Total: $O(n) \times \log n = O(n \log n)$
Merge Sort: running time

The running time of $\text{MergeSort}(A[1 \ldots n])$ is $O(n \log n)$.

We can prove that this running time is optimal if we consider sorting based on comparing pairs of numbers.

We cannot do (asymptotically) faster. Can we do better in practice?
Idea: Quicksort

- Divide array $A$ into 2 subarrays

- Recursively **fully sort** each subarray

- Combine the sorted subarrays by a **simple concatenation**
Quicksort

1. Split using pivot \( x \).

   - \( E \) (\( = x \))
   - \( L \) (\( \leq x \))
   - \( G \) (\( > x \))

2. Recur.

3. Concatenate.

   \( L \rightarrow E \rightarrow R \)

Select an element called **pivot**

1. Divide elements into 2 groups \( L \) (less or equal), and \( G \) (greater than pivot)
2. Conquer: recursively sort \( L \) and \( G \)
3. Combine: concatenate \( L \rightarrow E \rightarrow R \)
Example: quick sort

6  4  8  2  9  3  9  4  7  6  1
Example: quick sort

Rearrange elements with respect to 

\[ x = A[0] \]

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

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

\[
\begin{array}{cccccccccccc}
\leq 6 & > 6 \\
\end{array}
\]
Example: quick sort

6 4 8 2 9 3 9 4 7 6 1

6 is in its final position

1 4 2 3 4 6 6 9 7 8 9

sort the two parts recursively

1 2 3 4 4 6 6 7 8 9 9
QuickSort($A, \ell, r$)

if $\ell \geq r:
    \text{return}
    m \leftarrow \text{Partition}(A, \ell, r)
# A[m] is in the final position
QuickSort(A, \ell, m - 1)
QuickSort(A, m + 1, r)
Partitioning: example

- the **pivot** is \( x = A[\ell] \)
- loop \( i \) from \( \ell + 1 \) to \( r \) maintaining the following invariant:
  - \( A[k] \leq x \) for all \( \ell + 1 \leq k \leq j \)
  - \( A[k] > x \) for all \( j + 1 \leq k \leq i \)

- Partitioning: example
  6 4 2 3 9 8 9 4 7 6 1
- the pivot is $x = A[\ell]$
- move $i$ from $\ell + 1$ to $r$ maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
Partitioning: example

- the pivot is \( x = A[\ell] \)
- move \( i \) from \( \ell + 1 \) to \( r \) maintaining the following invariant:
  - \( A[k] \leq x \) for all \( \ell + 1 \leq k \leq j \)
  - \( A[k] > x \) for all \( j + 1 \leq k \leq i \)
- if encounter an out-of-order element:
  swap \( A[i] \) with \( A[j+1] \)
the pivot is $x = A[\ell]$
move $i$ from $\ell+1$ to $r$ maintaining the following invariant:

- $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
- $A[k] > x$ for all $j + 1 \leq k \leq i$

if encounter an out-of-order element:
swap $A[i]$ with $A[j+1]$
Partitioning: example

- the pivot is $x = A[\ell]$
- move $i$ from $\ell+1$ to $r$ maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$

![Partitioning diagram]
the pivot is $x = A[\ell]$
move $i$ from $\ell + 1$ to $r$ maintaining the following invariant:

- $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
- $A[k] > x$ for all $j + 1 \leq k \leq i$

if encounter an out-of-order element:
swap $A[i]$ with $A[j+1]$

Partitioning: example

\[ \begin{array}{cccccccccc}
6 & 4 & 2 & 3 & 4 & 6 & 9 & 9 & 7 & 8 & 1 \\
\end{array} \]
Partitioning: example

- the pivot is $x = A[\ell]$
- move $i$ from $\ell+1$ to $r$ maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element:
  - swap $A[i]$ with $A[j+1]$

![Partitioning example diagram](chart.png)
the pivot is $x = A[\ell]$

move $i$ from $\ell+1$ to $r$ maintaining the following invariant:

- $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
- $A[k] > x$ for all $j + 1 \leq k \leq i$

in the end, move $A[\ell]$ to its final place $j$
- the pivot is $x = A[\ell]$
- move $i$ from $\ell + 1$ to $r$ maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- in the end, move $A[\ell]$ to its final place $j$
Algorithm Partition($A$, $\ell$, $r$)

\[
x \leftarrow A[\ell] \quad \# \text{pivot}
\]

\[
j \leftarrow \ell
\]

\[
\text{for } i \text{ from } \ell + 1 \text{ to } r : \\
\quad \text{if } A[i] \leq x :
\]

\[
j \leftarrow j + 1
\]

\[
\quad \text{swap } A[j] \text{ and } A[i]
\]

\[
\text{swap } A[\ell] \text{ and } A[j]
\]

\[
\text{return } j
\]

\[
\# A[\ell + 1 \ldots j] \leq x, A[j + 1 \ldots i] > x
\]
Running time of Quick Sort

If we happen to choose the pivot $x$ in such a way that after the partitioning the array $A$ is split into even halves:

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

This is the same as in Merge sort, only here $n$ comes from partitioning, and in merge sort $n$ comes from combine (merge).

The running time of Quicksort is $O(n \log n)$
Quick Sort: summary

- Simple
- Comparison-based
- Very fast in practice
Which choice of pivot would yield an **optimal** partitioning of \( A \)?

\[
\begin{array}{cccccccc}
A & 7 & 2 & 5 & 3 & 7 & 13 & 1 & 6 & 3 \\
\end{array}
\]

A. 7

B. 6

C. 5

D. 1

E. None of the above
Which choice of pivot would yield the **worst** partitioning of $A$?

A. 7  
B. 6  
C. 5  
D. 1  
E. None of the above
Unlucky choice of pivot

If we choose a pivot in such a way that all values are greater than it, then in each recursive step we decrement a size of the problem only by 1:

\[ T(n) = O(n) + T(n-1) \]
Quick Sort: worst case complexity

Input size at each level

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>$n-1$</td>
</tr>
<tr>
<td>3</td>
<td>$n-2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k$</td>
<td>2</td>
</tr>
<tr>
<td>$k+1$</td>
<td>1</td>
</tr>
</tbody>
</table>

Work at each level

<table>
<thead>
<tr>
<th>Level</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>$n$</td>
</tr>
<tr>
<td>3</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k$</td>
<td>$n$</td>
</tr>
<tr>
<td>$k+1$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Total: $n \times n = O(n^2)$
Quick Sort: worst case complexity

Input size at each level

\[
\begin{align*}
n & \quad n \\
n - 1 & \quad n \\
n - 2 & \quad n \\
\vdots & \\
2 & \quad n \\
1 & \quad n \\
\end{align*}
\]

Work at each level

\[
\begin{align*}
n & \quad n \\
n & \quad n \\
n & \quad n \\
\vdots & \\
n & \quad n \\
\end{align*}
\]

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

Total: \( n^2 = O(n^2) \)
Pathological case

\[ T(n) = O(n^2) \]

It requires \( O(n^2) \) time to process the already sorted array which seems very inefficient since the array is already sorted!
Choosing random pivot

- We can show that if we choose \( x \) randomly there is at least 50% chance that a good pivot will be chosen!

  We can prove this using the expectation and the probabilities of random events

- If we choose all pivots at random, then half the times we do decrease the input sizes by a factor

- This implies that the height of the recursive tree will be \( (2 \log n) \) and the running time becomes \( O(n \log n) \)
RandomizedQuickSort(\(A, \ell, r\))

if \(\ell \geq r\):
    return

\(k \leftarrow \) random number between \(\ell\) and \(r\)

swap \(A[\ell]\) and \(A[k]\)

\(m \leftarrow\) Partition(\(A, \ell, r\))

# \(A[m]\) is in the final position

RandomizedQuickSort(\(A, \ell, m - 1\))

RandomizedQuickSort(\(A, m + 1, r\))
Randomized Quick sort: Summary

- Randomized Quick sort is a comparison-based algorithm based on random partitioning.
- Expected running time: $O(n \log n)$
- Still $O(n^2)$ in the worst case
- Very fast in practice