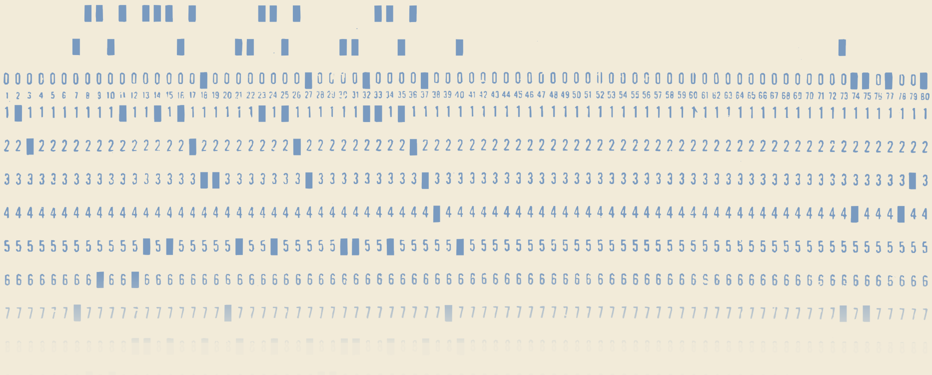


Lecture 09: Sorting III

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: October 31, 2024

Announcements

1. Fourth Quiz, due Friday
   - Similar format to before
   - Covers (Balanced) Binary Search Trees (Lectures 6–7)
   - Quiz is **closed resource**
     - No books, notes, internet, etc.
     - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment (Draft) Posted
   - Due Wednesday, 13 November
3. Attendance Code:

Meeting Goals

- Discuss non-comparison based sorting
  - RadixSort
  - CountingSort
- Beyond worst-case sorting
- More Divide & Conquer algorithms

From Last Time

Sorting by Divide and Conquer:

- MERGESORT: worst case $O(n \log n)$ running time
- QUICKSORT: worst case $O(n^2)$, expected time $O(n \log n)$

From Last Time

Sorting by Divide and Conquer:

- MergeSort: worst case $O(n \log n)$ running time
- QuickSort: worst case $O(n^2)$, expected time $O(n \log n)$

Lower Bounds:

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons to sort arrays of length n in the worst case.

From Last Time

Sorting by Divide and Conquer:

- MERGESORT: worst case $O(n \log n)$ running time
- QUICKSORT: worst case $O(n^2)$, expected time $O(n \log n)$

Lower Bounds:

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons to sort arrays of length n in the worst case.

So we're, like, done with sorting right?

Non Comparison-Based Sorting

Non Comparison-Based Sorting

Theorem

*Any **comparison-based** sorting algorithm requires* $\Omega(n \log n)$ *comparisons to sort arrays of length* n *in the worst case.*

Recall:

- A **comparison-base sorting algorithm** is any algorithm whose decisions are made only made based on the outcomes of comparison operations
- The actual numerical values are not used, only relative order
- For example, adding the same fixed value to each element of the array has *no effect* on the operations performed by the algorithm

Questions.

- What would **non**-comparison based algorithm look like?
- How efficient could such an algorithm be?

Warmup: Sorting Binary Values

Question. How efficiently can we sort a *binary array?*

$$a = [1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1]$$

Method 1. Use the SPLIT method from QUICKSORT with pivot 0.

- This will take $\Theta(n)$ time!
- **Generalization:** RADIXSORT

Method 2. Count the number of 0's and 1's in a, then write this many 0's and 1's in order.

- This will also take $\Theta(n)$ time!
- **Generalization:** COUNTINGSORT

Binary Representation of Numbers

Recall. Every number can be represented in binary notation:

- $1 = 1_2$
- $2 = 10_2$
- $3 = 11_2$
- $4 = 100_2$
- $5 = 101_2$
- $\vdots$

More formally:

$$(b_k b_{k-1} \cdots b_1 b_0)_2 = \sum_{i=0}^{k} b_i 2^i$$

where each $b_i \in \{0, 1\}$.

Pictorially: $10110_2 =$ 

Comparing binary values. To determine if $b < c$, perform *bit-wise* comparison.

```
procedure BitwiseCompare(b, c)
    i ← k
    while i > 0 do
        if b_i < c_i then
            return True
        else if b_i > c_i then
            return False
        end if
        i ← i − 1
    end while
    return False
end procedure
```

Binary Representation of Numbers

Recall. Every number can be represented in binary notation:

- $1 = 1_2$
- $2 = 10_2$
- $3 = 11_2$
- $4 = 100_2$
- $5 = 101_2$
- $\vdots$

More formally:

$$(b_k b_{k-1} \cdots b_1 b_0)_2 = \sum_{i=0}^{k} b_i 2^i$$

where each $b_i \in \{0, 1\}$.

Pictorially: $10110_2 =$ 

Comparing binary values. To determine if $b < c$, perform *bit-wise* comparison.

```
procedure BitwiseCompare(b, c)
    i ← k
    while i > 0 do
        if b_i < c_i then
            return True
        else if b_i > c_i then
            return False
        end if
        i ← i − 1
    end while
    return False
end procedure
```

Binary Representation of Numbers

PollEverywhere

Which is the largest binary value?

1. 100101001110111_2
2. 100011001110111_2
3. 100101011110111_2
4. 100101011100111_2

pollev.com/comp526

Comparing binary values. To determine if $b < c$, perform *bit-wise* comparison.

procedure BitwiseCompare(b, c)
  $i \leftarrow k$
  while $i > 0$ **do**
    if $b_i < c_i$ **then**
      return True
    else if $b_i > c_i$ **then**
      return False
    end if
    $i \leftarrow i - 1$
  end while
  return False
end procedure

Binary Representation of Numbers

Main Observation. We can compare values by incrementally reading bits.

- The first bit on which b and c differ determines whether or not $b < c$
  - Do not need to read the entire value unless $|b - c| \leq 1$.

Comparing binary values. To determine if $b < c$, perform *bit-wise* comparison.

```
procedure BitwiseCompare(b, c)
    i ← k
    while i > 0 do
        if b_i < c_i then
            return True
        else if b_i > c_i then
            return False
        end if
        i ← i − 1
    end while
    return False
end procedure
```

Binary Representation of Numbers

Main Observation. We can compare values by incrementally reading bits.

- The first bit on which b and c differ determines whether or not $b < c$
  - Do not need to read the entire value unless $|b - c| \leq 1$.

Radix Sort Idea. Sort values by incrementally reading bits.

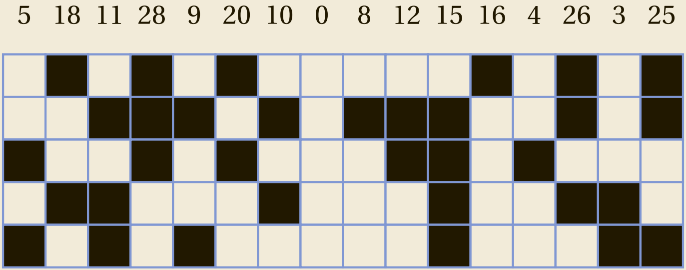
- Compare individual bits rather than entire values
- Split numbers according to bit comparisons

Comparing binary values. To determine if $b < c$, perform *bit-wise* comparison.

```
procedure BitwiseCompare(b, c)
    i ← k
    while i > 0 do
        if b_i < c_i then
            return True
        else if b_i > c_i then
            return False
        end if
        i ← i − 1
    end while
    return False
end procedure
```

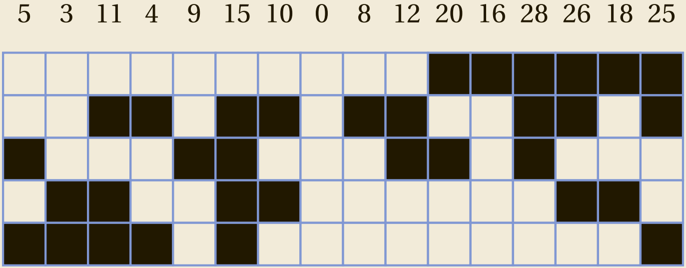
Example: Sorting with Bit Comparison

Consider the array $a = [5, 18, 11, 28, 9, 20, 10, 0, 8, 12, 15, 16, 4, 26, 3, 25]$



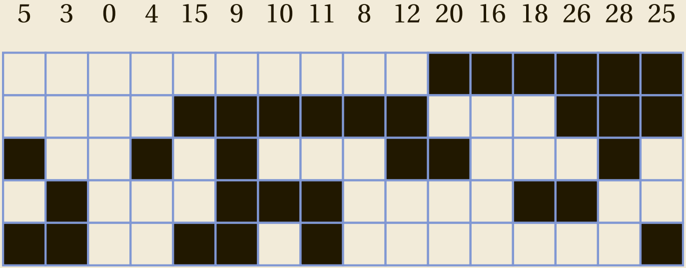
Example: Sorting with Bit Comparison

Consider the array $a = [5, 18, 11, 28, 9, 20, 10, 0, 8, 12, 15, 16, 4, 26, 3, 25]$



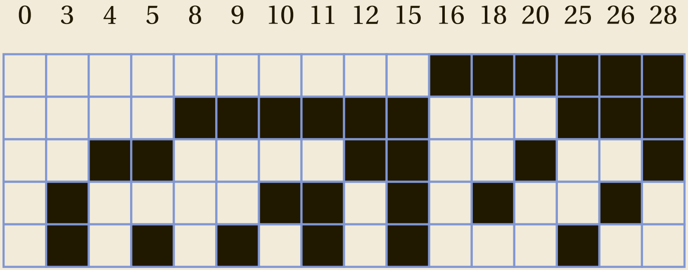
Example: Sorting with Bit Comparison

Consider the array $a = [5, 18, 11, 28, 9, 20, 10, 0, 8, 12, 15, 16, 4, 26, 3, 25]$



Example: Sorting with Bit Comparison

Consider the array $a = [5, 18, 11, 28, 9, 20, 10, 0, 8, 12, 15, 16, 4, 26, 3, 25]$



RadixSort in (Pseudo)Code

Denote the bth bit of $a[i]$ by $a[i][b]$

```
procedure BitSplit(a, min, max, b)
    i ← min, j ← max
    while i < j do
        while a[i][b] = 0 and i < max do
            i ← i + 1
        end while
        while a[j][b] = 1 and j > min do
            j ← j − 1
        end while
        if i = max or j = min then
            return i or j
        end if
        Swap(a, i, , j)
        i ← i + 1, j ← j + 1
    end while
    return i − 1
end procedure
```

RadixSort in (Pseduo)Code

Denote the bth bit of $a[i]$ by $a[i][b]$

```
procedure BitSplit(a, min, max, b)
    i ← min, j ← max
    while i < j do
        while a[i][b] = 0 and i < max do
            i ← i + 1
        end while
        while a[j][b] = 1 and j > min do
            j ← j − 1
        end while
        if i = max or j = min then
            return i or j
        end if
        Swap(a, i, , j)
        i ← i + 1, j ← j + 1
    end while
    return i − 1
end procedure
```

PollEverywhere

What is the running time of BitSplit as a function of $n = \max - \min$?

pollev.com/comp526

RadixSort in (Pseduo)Code

Denote the bth bit of $a[i]$ by $a[i][b]$

```
procedure BitSplit(a, min, max, b)
    i ← min, j ← max
    while i < j do
        while a[i][b] = 0 and i < max do
            i ← i + 1
        end while
        while a[j][b] = 1 and j > min do
            j ← j − 1
        end while
        if i = max or j = min then
            return i or j
        end if
        Swap(a, i, , j)
        i ← i + 1, j ← j + 1
    end while
    return i − 1
end procedure
```

```
procedure RadixSort(a, b, min, max)
    if b < 0 or min = max then
        return
    end if
    i ← BitSplit(a, min, max, b)
    RadixSort(a, min, i, b − 1)
    RadixSort(a, i + 1, max, b − 1)
end procedure
```

RadixSort in (Pseduo)Code

Denote the bth bit of $a[i]$ by $a[i][b]$

Analysis of RADIXSORT (informal)

- Consider each value of $b = B, B-1, \ldots, 0$
- All values $a[i][b]$ are read once in all calls at level b
  - total running time on level b is $\Theta(n)$

$\implies$ Total running time is $\Theta(Bn)$.

```
procedure RADIXSORT(a, b, min, max)
    if b < 0 or min = max then
        return
    end if
    i ← BITSPLIT(a, min, max, b)
    RADIXSORT(a, min, i, b − 1)
    RADIXSORT(a, i + 1, max, b − 1)
end procedure
```

RadixSort in (Pseudo)Code

Denote the bth bit of $a[i]$ by $a[i][b]$

Analysis of RadixSort (informal)

- Consider each value of $b = B, B-1, \ldots, 0$
- All values $a[i][b]$ are read once in all calls at level b
    - total running time on level b is $\Theta(n)$

$\Longrightarrow$ Total running time is $\Theta(Bn)$.

```
procedure RadixSort(a, b, min, max)
    if b < 0 or min = max then
        return
    end if
    i ← BitSplit(a, min, max, b)
    RadixSort(a, min, i, b − 1)
    RadixSort(a, i + 1, max, b − 1)
end procedure
```

Question. Is the better or worse than $\Theta(n \log n)$?

RadixSort Visualization

https://willrosenbaum.com/blog/2022/radix-sort/

CountingSort

A Simple Idea

Question. What if we already know the set of **all possible** values stored in a?

- Suppose the possible values are $0, 1, \ldots, m$
- Form an array c of counts
  - $c[i]$ stores the number of times i occurs in a.

Example.

- $a = [3, 0, 1, 2, 0, 1, 2, 1, 1, 1, 2, 0, 0, 3, 3, 1, 2, 0, 0, 0, 1, 0, 3]$
- $c = [8, 7, 4, 4]$

Question. Given c, how can we sort a?

- Add $c[i]$ copies of i to a!

CountingSort

```
procedure CountingSort(a, n, m)
    c ← 0-array of length m
    for i = 0, 1, ..., n − 1 do
        c[a[i]] ← c[a[i]] + 1
    end for
    i ← 0
    for j = 0, 1, ..., m do
        for k = 0, 1, ..., c[j] − 1 do
            a[i] ← j
            i ← i + 1
        end for
    end for
end procedure
```

CountingSort

```
procedure CountingSort(a, n, m)
    c ← 0-array of length m
    for i = 0, 1, ..., n − 1 do
        c[a[i]] ← c[a[i]] + 1
    end for
    i ← 0
    for j = 0, 1, ..., m do
        for k = 0, 1, ..., c[j] − 1 do
            a[i] ← j
            i ← i + 1
        end for
    end for
end procedure
```

PollEverywhere

What is the running time of CountingSort where a has size n and contains values from 0 to $m-1$?

1. $\Theta(nm)$
2. $\Theta(n \log m)$
3. $\Theta(n+m)$
4. $\Theta(n+\log m)$
5. $\Theta(\log n + m)$

pollev.com/comp526

CountingSort

```
procedure CountingSort(a, n, m)
    c ← 0-array of length m
    for i = 0, 1, ..., n − 1 do
        c[a[i]] ← c[a[i]] + 1
    end for
    i ← 0
    for j = 0, 1, ..., m do
        for k = 0, 1, ..., c[j] − 1 do
            a[i] ← j
            i ← i + 1
        end for
    end for
end procedure
```

Analysis:

Sorting in the Real World

Real-World Sorting?

So far we've analyzed the running time of sorting on **worst-case** inputs

Question. Are "typical" inputs to sorting close to the worst case?

- What are worst-case inputs?
  - in general, "worst-case" depends on the algorithm
  - ... but our $\Omega(n \log n)$ comparison lower bound can be extended to *random permutations*
  - $\implies$ for any algorithm, sorting a random array requires $\Omega(n \log n)$ comparisons in expectation
- Are typical inputs to sorting algorithms similar to (uniformly) random arrays **in the real world**?
  - if they are, there isn't much we can do (lower bound)
  - but if they aren't, can our sorting algorithm **adapt** to the input and **exploit** its structure?

Partially Sorted Inputs

Often, **real world** data to be sorted contains **runs** of increasing values

- Even random arrays will have *some* increasing sub-strings
- Only a decreasing array has all runs of size 1

Question. Can we exploit existing increasing runs in our data to sort it faster?

PollEverywhere

Which sorting algorithm exploits the idea that combining sorted arrays is easier than sorting from scratch?

1. HEAPSORT
2. MERGESORT
3. QUICKSORT
4. RADIXSORT

pollev.com/comp526

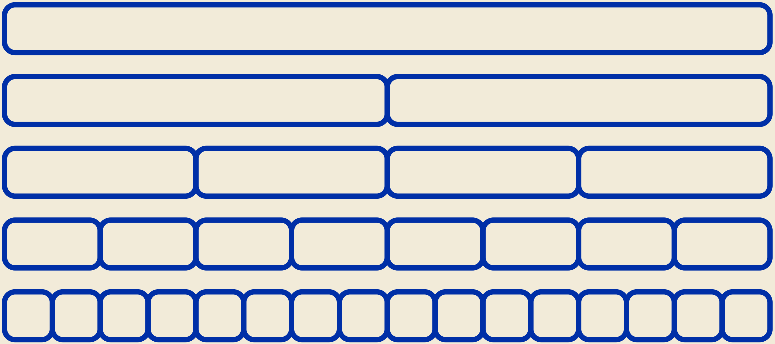
MergeSort Behaving Badly

A nice input?

4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

MergeSort Behaving Badly

MergeSort merges



4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Question. Which merges were **unnecessary**?

MergeSort with a Simple Check

A Simple Improvement

- Only MERGE if $a[i \dots k]$ is not already sorted
- Since $a[i \dots j]$ and $a[j+1 \dots k]$ are both sorted, this check can be done in $O(1)$ time.
    - How?

```
procedure MERGESORT(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MERGESORT(a, i, j)
        MERGESORT(a, j + 1, k)
        b ← COPY(a, i, j)
        c ← COPY(a, j + 1, k)
        MERGE(b, c, a, i)
    end if
end procedure
```

MergeSort with a Simple Check

A Simple Improvement

- Only MERGE if $a[i \ldots k]$ is not already sorted
- Since $a[i \ldots j]$ and $a[j+1 \ldots k]$ are both sorted, this check can be done in $O(1)$ time.
  - How?

```
procedure MERGESORT+(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MERGESORT(a, i, j)
        MERGESORT(a, j + 1, k)
        ▷ check if already sorted
        if a[j] ≤ a[j + 1] then
            return
        end if
        b ← COPY(a, i, j)
        c ← COPY(a, j + 1, k)
        MERGE(b, c, a, i)
    end if
end procedure
```

MergeSort with a Simple Check

A Simple Improvement

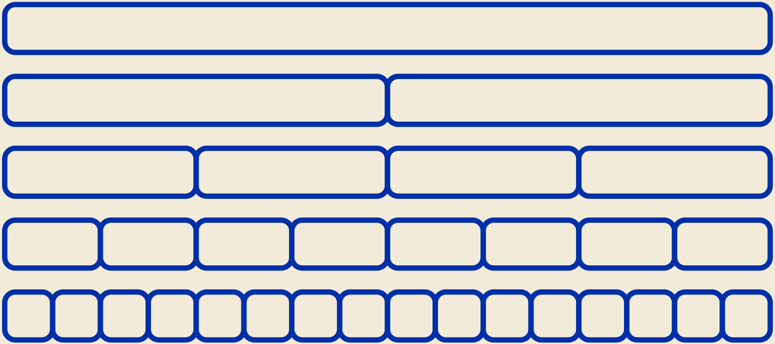
- Only MERGE if $a[i \ldots k]$ is not already sorted
- Since $a[i \ldots j]$ and $a[j+1 \ldots k]$ are both sorted, this check can be done in $O(1)$ time.
  - How?
- MERGESORT+ still has *best case* running time $\Theta(n \log n)$
  - why?

How could we improve MERGESORT so that **best case** running time is $o(n \log n)$?

procedure MERGESORT+(a, i, k)
 if $i < k$ **then**
  $j \leftarrow \lfloor (i + k)/2 \rfloor$
  MERGESORT(a, i, j)
  MERGESORT$(a, j+1, k)$
  ▷ check if already sorted
  if $a[j] \leq a[j+1]$ **then**
   return
  end if
  $b \leftarrow$ COPY(a, i, j)
  $c \leftarrow$ COPY$(a, j+1, k)$
  MERGE(b, c, a, i)
 end if
end procedure

Further MergeSort Improvements

MergeSort Merges

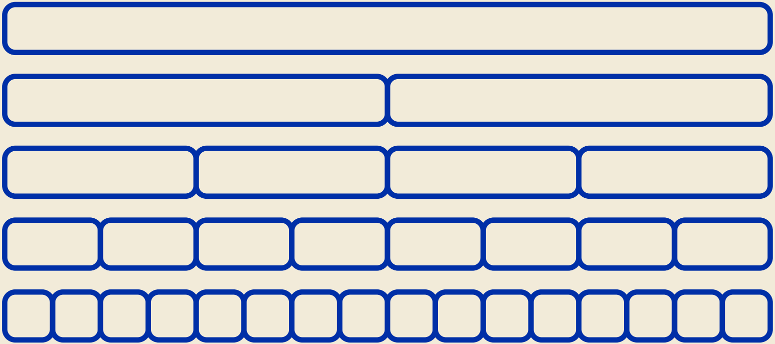


4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Question. Which **recursive calls** were unnecessary?

Further MergeSort Improvements

MergeSort Merges



4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Question. How could we have avoided unnecessary recursive calls?

Further MergeSort Improvements

Existing Runs

	Run 1								Run 2			Run 3			
4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Idea. Use existing runs in the data and only sort **runs**!

Further MergeSort Improvements

Run 1									Run 2			Run 3			
4	6	7	9	11	12	13	15	16	2	5	10	1	3	8	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

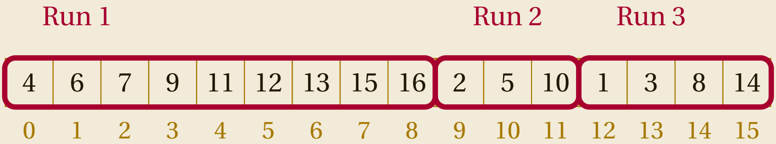
PollEverywhere

Which merge would it be more efficient to perform **first**?

1. 1 and 2 first
2. 2 and 3 first
3. no (significant) difference

pollev.com/comp526

Further MergeSort Improvements



Merge order matters!

Merge Trees and PowerSort

Overall Strategy

- MergeSort but:
  - don't sort runs that are already sorted
  - only split along run boundaries
- Remaining design choice: In what *order* should we perform the Merge operations?

Merge Trees and PowerSort

Overall Strategy

- MergeSort but:
  - don't sort runs that are already sorted
  - only split along run boundaries
- Remaining design choice: In what *order* should we perform the Merge operations?
  - optimal merge trees are possible, but too costly to find
  - use good **approximation** to optimal merge tree:

$\Longrightarrow$ **PowerSort** algorithm used by Python

- developed by Sebastian Wild (my predecessor for COMP526) and others
- **open competition for improvements!**

Divide & Conquer

So Far

We've seen how effective the Divide & Conquer strategy is for **sorting**

…what about Divide & Conquer **other problems**?

So Far

We've seen how effective the Divide & Conquer strategy is for **sorting**

…what about Divide & Conquer **other problems**?

Problem 1. k-Selection:

- Given an array a of n numbers, find the kth largest number

So Far

We've seen how effective the Divide & Conquer strategy is for **sorting**

. . . what about Divide & Conquer **other problems**?

Problem 1. k-Selection:

- Given an array a of n numbers, find the kth largest number

Problem 2. Majority:

- Given an array a of n items, is there an item that is repeated more than $> n/2$ times?

So Far

We've seen how effective the Divide & Conquer strategy is for **sorting**

...what about Divide & Conquer **other problems**?

Problem 1. k-Selection:

- Given an array a of n numbers, find the kth largest number

Problem 2. Majority:

- Given an array a of n items, is there an item that is repeated more than $> n/2$ times?

There are **WAY MORE** applications of Divide & Conquer as well!

- Versatile general problem solving strategy

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

Modify QuickSort!

- Choose pivot p
- Perform split
- *only recurse on half that contains kth smallest value*
  - this will be the half that contains index k

k-Selection

Problem. Given an array a of n numbers, find the kth smallest number.

Simple solution.

- sort a in $O(n \log n)$ time
- return $a[k]$

Can we do better?

Modify QuickSort!

- Choose pivot p
- Perform split
- *only recurse on half that contains kth smallest value*
  - this will be the half that contains index k

```
procedure QuickSelect(a, min, max, k)
    if max − min ≤ 1 then
        return a[min]
    end if
    p ← SelectPivot(a, min, max)
    j ← Split(a, min, max, p)
    if j = k then
        return a[k]
    else if j < k then
        QuickSelect(a, j + 1, max, k)
    else
        QuickSelect(a, min, j − 1, k)
    end if
end procedure
```

For Next Time

Questions to Consider

1. If we choose a pivot uniformly at random for QuickSelect, what is the procedure's expected running time?
2. Can we choose a pivot *deterministically* that gives this same running time?
3. How efficiently can we solve the majority problem?
    - Hint: if a value v is a majority, then it must be a majority on some half of the array.

Starting next week:

- Text Searching

Scratch Notes