Sorting and Element Selection

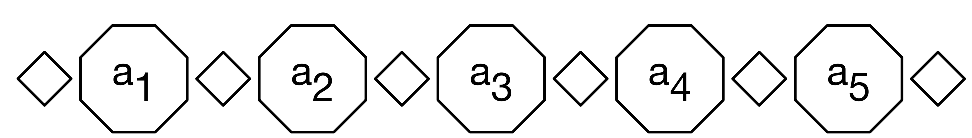
Thomas Schwarz, SJ

Permutations

- A permutation of the set $\{1,2,\ldots,n\}$ is a reordering of the numbers where each number between 1 and n appears exactly once.

Permutations

- How many permutations are there?
  - Use recurrence!
    - In a permutation of $\{1,2,\ldots,n\}$, where is the n located?
    - There are $n-1$ other numbers.
    - This gives us $n-2$ gaps and spots before and after



Permutations

- Let $n!$ be the number of permutations of n elements
  - This gives us the recurrence
    - $n! = n \cdot (n-1)!$
  - which can be unfolded very simply
    - $$n! = \prod_{i=1}^{n} i$$

Permutations

How do we determine its asymptotic growth?

$$n! = \prod_{i=1}^{n} i$$

Use Logarithms!

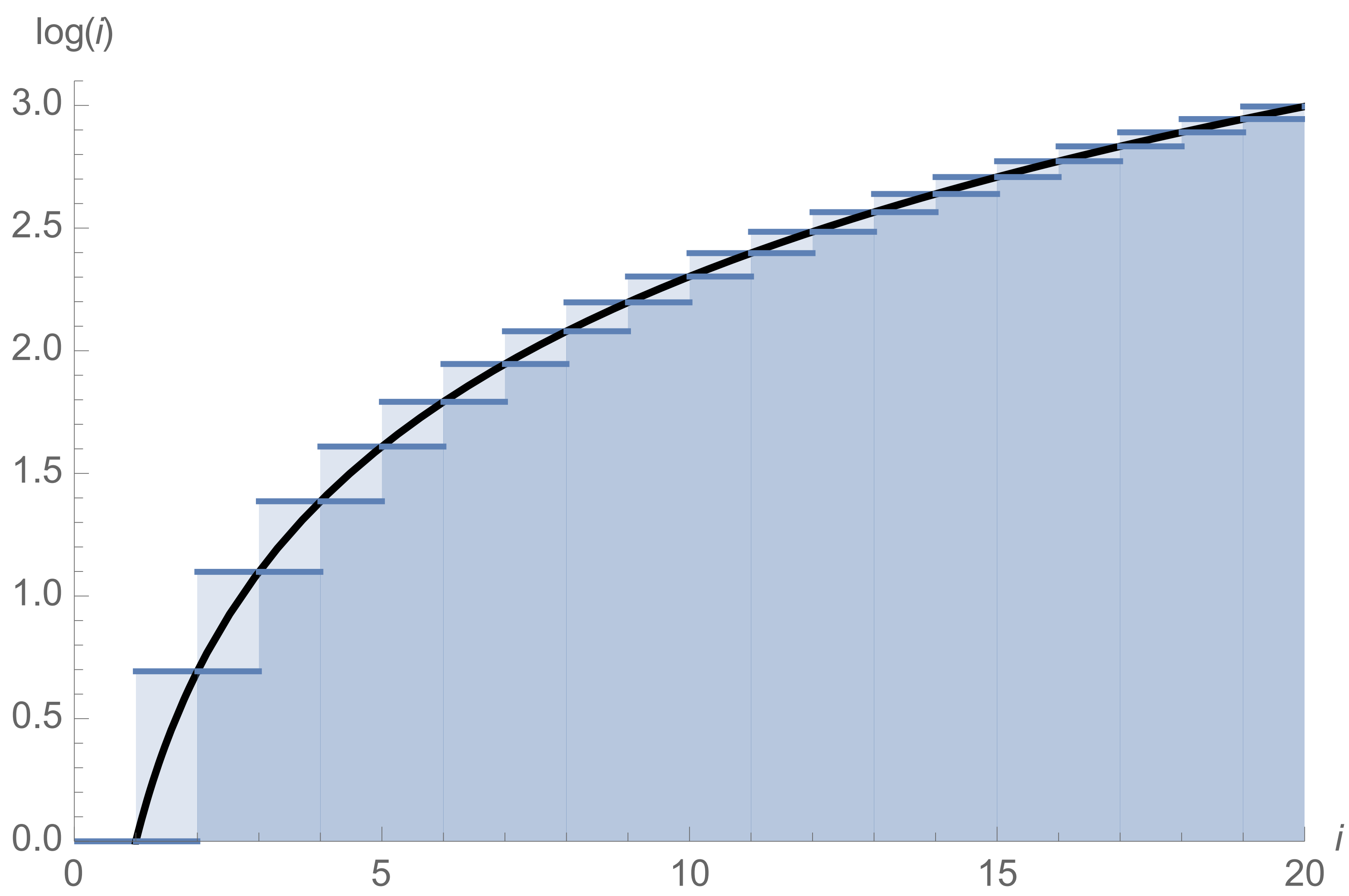
Permutations

- Approximation of the factorial

Use $$\log n! = \sum_{i=1}^{n} \log(i)$$

Use an integral!

Permutations



$$\sum_{i=1}^{n-1} \log(i) < \int_1^n \log(x)dx < \sum_{i=1}^{n} \log i$$

Permutations

$$
\begin{aligned}
\log(n!) &= \sum_{i=1}^{n} \log(i) \\
&\approx \int_{i=1}^{n} \log(x) dx \\
&= [x \log x - x]_1^n \\
&= n \log(n) - n + 1
\end{aligned}
$$

Permutations

Therefore

$$\begin{aligned} n! &\approx \exp(n\log(n) - n - 1) \\ &= \exp(\log(n^n) - n + 1) \\ &= n^n \cdot e^{-n} \cdot e \\ &= e \cdot \left(\frac{n}{e}\right)^n \end{aligned}$$

Permutations

An analysis of the error substituting the Riemann sum for an integral gives Stirling's formula (invented by de Moivre)

$$\sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n} \leq n! \leq en^{n+\frac{1}{2}}e^{-n}$$

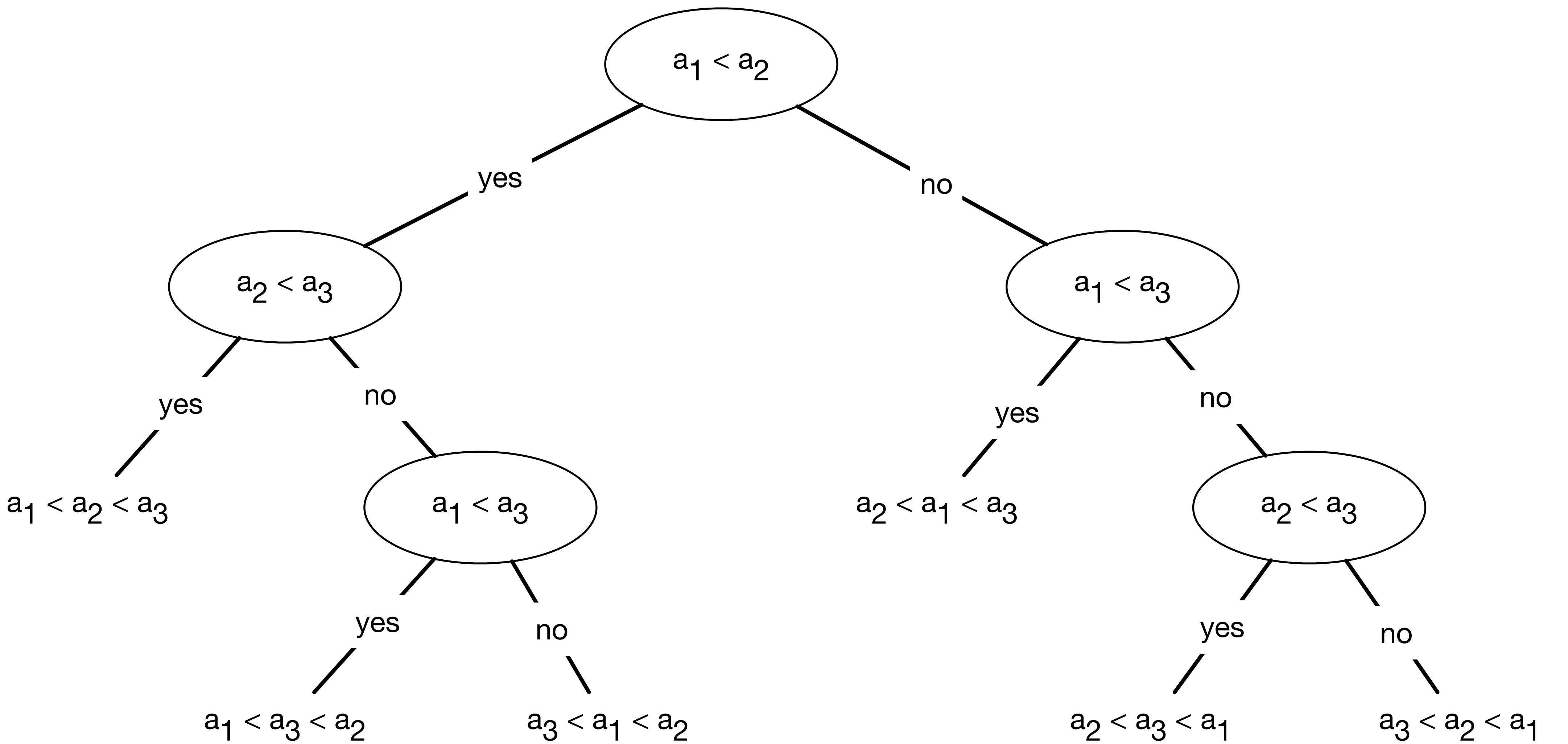
Sorting by Comparison

- Many sorting algorithms use comparisons
- An algorithm needs to be able to sort with all orders of inputs, i.e. distinguish between $n!$ arrangements of the input by order
  - assuming all elements are different

Sorting by Comparison

- Sorting algorithm makes a comparison, then decides on what to do
- Can be represented as a binary tree

Sorting by Comparison



A fictitious algorithm for sorting
as a Decision Tree

Sorting by Comparison

- Represent any comparison based algorithm by such a tree
- Any run of the algorithm represents a path from the root to a leaf node
- Leaf nodes represent an algorithm finishing,
  - So they need to have an ordering, i.e. a permutation of the input array

Sorting by Comparison

- How many leaves does a tree with N leaves have?
- A tree of height h has how many leaves?
  - Height 0: only root, one leaf
  - Height 1: only root plus one or two leaves: ≤ 2
  - Height 2: at most two nodes at height one have at most $\leq 2^2$ leaves
  - Induction: Height h has at most 2^h leaves

Sorting by Comparison

- Relationship between height of decision tree and number of elements to be sorted:
  - Need to have at least $n!$ leaves:
  - $2^h \geq n!$
    - which implies
      - $h \geq \log_2(n!) = \dfrac{1}{\log(2)} \log(n!)$
      - $\approx \dfrac{1}{\log(2)} n \log(n) - n + 1$
      - $= \Theta(n \log(n))$

Sorting by Comparison

- Since the height of the decision tree is the worst time runtime, we have

    - *The runtime of a comparison based sorting algorithm is* $\Omega(n\log(n))$

Linear Time Sorting

- Counting sort
  - Assume we want to sort numbers in $\{1,2,\ldots,k-1,k\}$
  - Create a dictionary with keys in $\{1,2,\ldots,k-1,k\}$
    - E.g. as an array `Int(1:k)`
  - Walk through the array, updating the count
  - Once the count is done, go through the dictionary in order of the keys, emitting as many keys as the count

Linear Time Sorting

- Counting sort:
  -

10	3	4	10	12	4	5	3	8	9	2	2	5	10	1	2	7

  - create a counting array:
  -

1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:	13:

  - Walk through the array and calculate counts
  -

1: 1	2: 3	3: 3	4: 2	5: 2	6: 0	7: 1	8: 1	9: 1	10: 3	11: 0	12: 1	13: 0

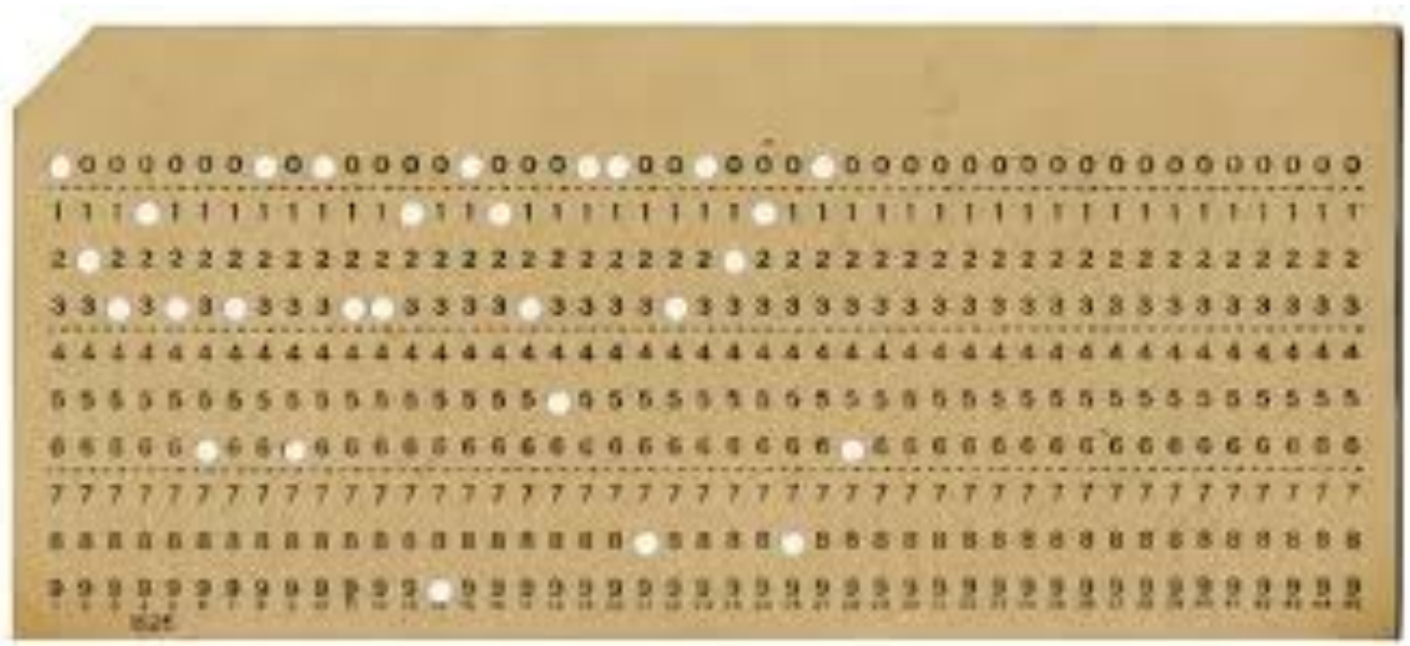
  - Emit keys according to count
    - 1 2 2 2 3 3 3 4 4 5 5 7 8 9 10 10 10 12

Linear Time Sorting

- If there are n elements in the array, then counting sort uses
    - $\sim k$ to create and evaluate the counting array
    - $\sim n$ to update the counting array
- Therefore: counting sort run-time is $\Theta(n+k)$

Linear Time Sorting

- Radix Sort
  - Imagine sorting punch cards with by ID in the first columns



Linear Time Sorting

- Simple Method:
    - Create heaps of cards based on the first digit
        - Then recursively sort the heaps

Linear Time Sorting

- Better method:
  - Sort according to the last digit
    - Then use a *stable sort* to sort after the second-last digit
    - Then use a stable sort to sort after the third-last digit

Linear Time Sorting

- Stable sort:
  - Leave order of elements with the same key during sorting
  - Insertion sort, merge sort, bubble sort, counting sort are all stable
  - Heap sort, selection sort, shell sort, and quick sort are not

Linear Time Sorting

- Radix sort:
  - ```
    for i in range(length(key), 0, -1):
            stable_sort on digit i of key
    ```
    ```

Linear Time Sorting

135	220	302	023
242	321	203	122
122	221	220	135
023	242	321	144
220	122	221	203
144	302	122	220
321	023	023	221
221	203	135	242
203	144	242	302
302	135	144	321

Linear Time Sorting

- Radix sort correctness
  - What would be a loop invariant?

Linear Time Sorting

- Assume n keys of d digits in $\{0,1,\ldots,r-1\}$
- Use counting sort to sort in time $\Theta(n+r)$
- Radix sort then takes $\Theta(d(n+r))$ time

Linear Time Sorting

- Given n numbers of b bits each
- Assume $b = O(\log(n))$
- Choose $r = \lfloor \log_2(n) \rfloor$.
    - Divide the b-bit numbers into "digits" of length r
    - Thus, each round of radix sort takes time $\Theta(n + 2^r)$
    - There are $\lceil \frac{b}{r} \rceil$ rounds
    - So, radix sort takes $\Theta(\frac{b}{r}(n + 2^r)) = \Theta(\frac{b}{r}(n + n)) = \Theta(n)$ time!

Selection

Selection Problems

- Given an unordered array:
  - Find the k-largest (-smallest) element in an unordered array
  - Naïve Solution:
    - Sort (usually in time $\Theta(n \log n)$)
    - Pick element $n - k$ or k of the sorted array

Selection Problem

- Finding the maximum
- Finding the maximum and minimum at the same time
- Finding the k^{th} largest element
- Finding the median

Maximum

- Obvious algorithm:

```
def max(array):
    result = array[0]
    for i in range(1, len(array)):
        if array[i]>result:
            result = array[i]
```

- $n - 1$ comparisons

Maximum

- Toy algorithm:
  - Partition array into $\lfloor n/2 \rfloor$ pairs.
    - (There might be an additional element).
  - Use one comparison in order to select the largest of each pair (plus the odd one out if exists)
  - These form an array of length $\lfloor n/2 \rfloor + 1$
  - Recursively call the toy algorithm

Maximum

- What is the recurrence relation?

Maximum

- $T(n) = T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
- T(2) = 1

- Now use substitution to get an idea of solving the recurrence

Maximum

- Assume n is a power of 2

Maximum

- Recurrence then becomes
  - $T(n) = T(n/2) + n/2, \quad T(2) = 1$
  - $\quad\quad = T(n/4) + n/4 + n/2$
  - $\quad\quad = T(n/8) + n/8 + n/4 + n/2$
  - $\quad\quad\quad \ldots$
  - $\quad\quad = T(2) + 2 + 4 + 8 + \ldots + n/8 + n/4 + n/2$
  - $\quad\quad = n - 1$

Maximum

- Now prove by induction for all $n \in \mathbb{N}$
- $T(n) = T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
- $T(2) = 1$

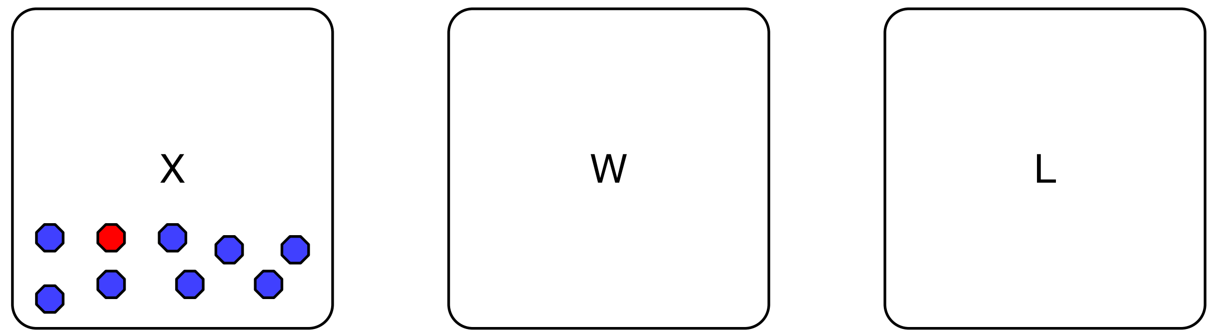
Maximum

- Induction Hypothesis: $T(m) = m - 1$ if $m < n$.
- $T(n)$
  - $= T(n - \lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$
  - $= n - \lfloor n/2 \rfloor - 1 + \lfloor n/2 \rfloor$
  - $= n - 1$

Maximum

- In fact:
  - *Theorem: Finding the maximum of an array of length n costs at least $n-1$ comparisons*
  - *Proof*: Place all elements into three buckets:
    - One for not-looked at
    - One for won all comparisons
    - One for lost all comparisons

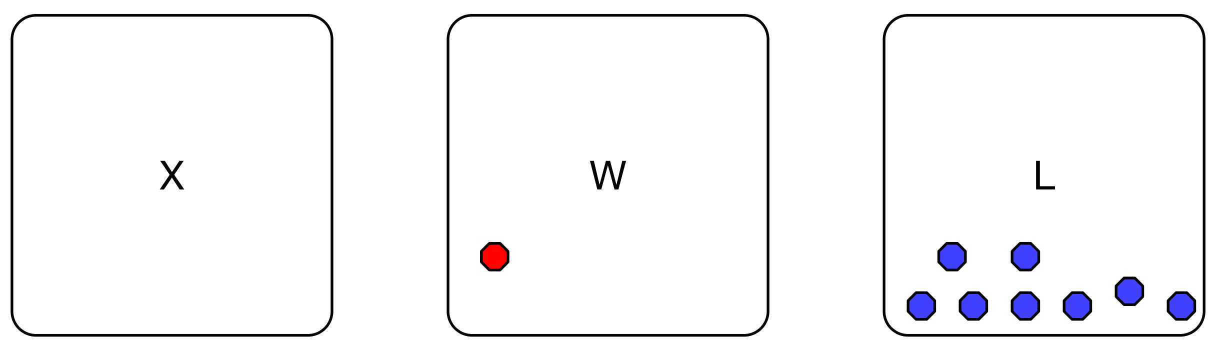
Maximum



- A single comparison can involves 6 cases
  - X-X: move two elements from X, one into W, one into L
  - X-W: move one element from X into W or move one element from X into W and one from W into L
  - X-L: move one element from X into W or one into L
  - W-W: move one element from W to L
  - W-L: nothing or move one element from W to L
  - L-L: nothing

Maximum

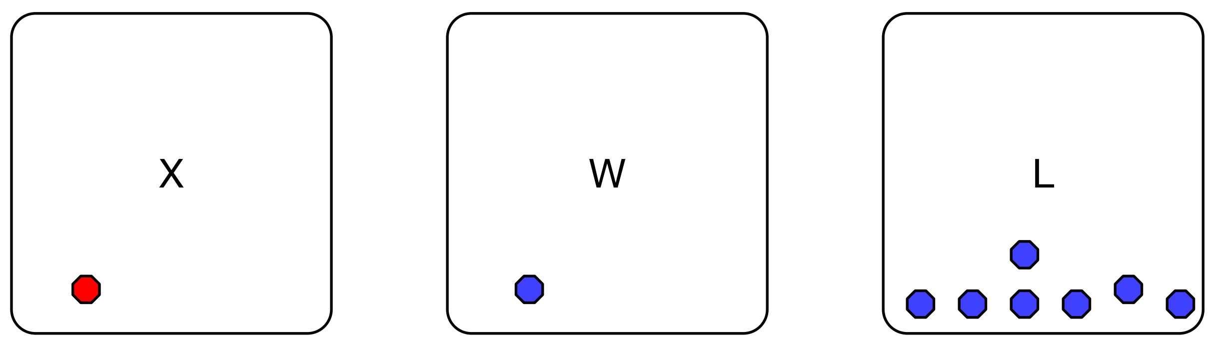
- To have finished the algorithm:
  - No elements left in X
  - Only one element left in W



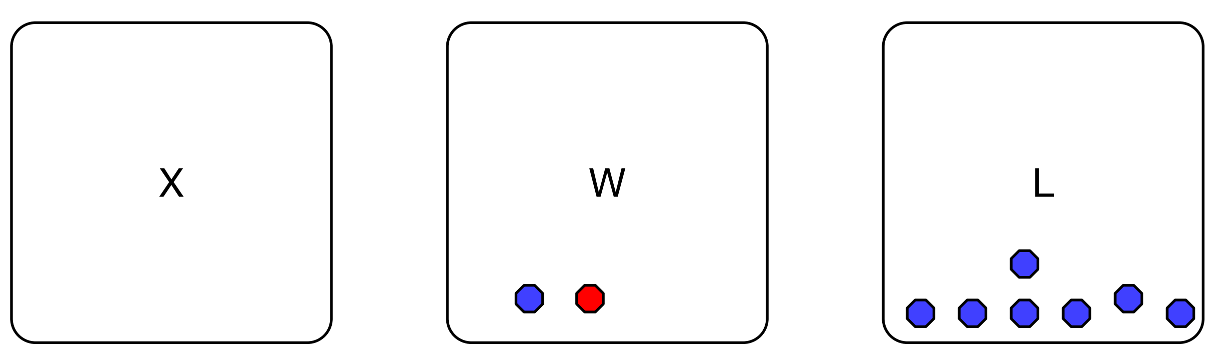
  - Otherwise, can construct counterexample

Maximum

- One left in X: could be the maximum



- Two (or more) left in W:
  - Which one is the maximum?



Maximum

- Each comparison sends at most one element to L
- At best, $n - 1$ comparisons

Combined Maximum and Minimum

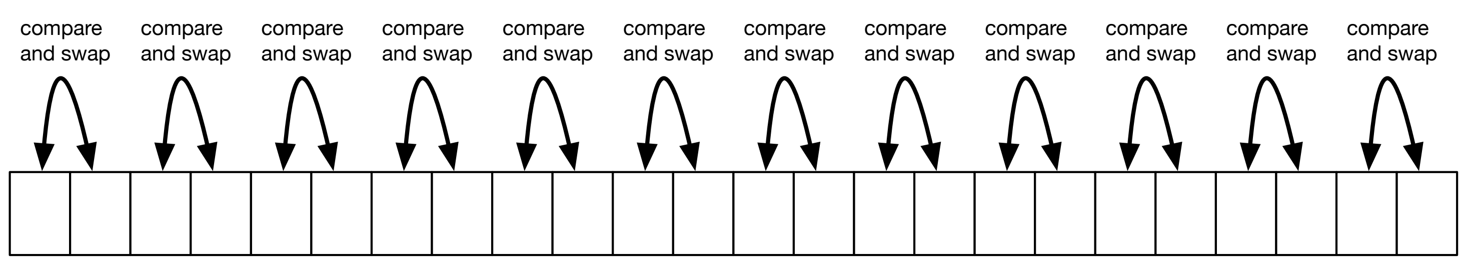
- Combined Maximum and Minimum
  - Naïve algorithm:
    - Calculate the max, then the min (can exclude the max)
      - $m - 1 + m - 2 = 2m - 3$ comparisons

Combined Maximum and Minimum

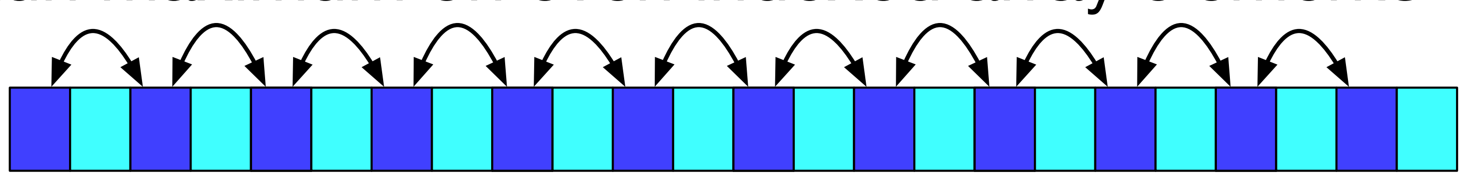
- A better algorithm
  - Divide the array into pairs
  - Compare the values of each pair
  - Place the winner of each pair in one array, the looser of each array in a second array
    - (Or use swapping so that the winners are in even position and the losers are in odd positions)
  - Now use maximum and minimum on the two sub-arrays

Combined Maximum and Minimum

- Case 1: n is even
  - There are $n/2$ pairs or $n/2$ comparisons



  - Run maximum on even indexed array elements



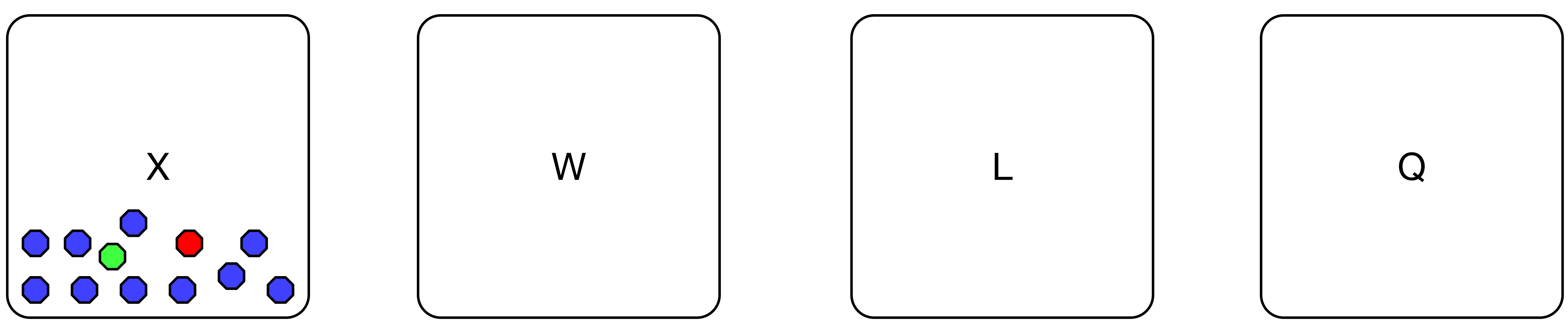
  - This gives us $n/2 - 1$ comparisons
  - Same for minimum
  - Total is $n/2 + n/2 - 1 + n/2 - 1 = \dfrac{3n}{2} - 2$ comparisons

Combined Maximum and Minimum

- Case: n is odd
  - Run algorithm on the first $n-1$ elements
    - $\dfrac{3n-3}{2} - 2$ comparisons
  - Then add two comparisons to see whether the last element is either minimum or maximum
    - Total of $\dfrac{3n-3}{2}$ comparisons

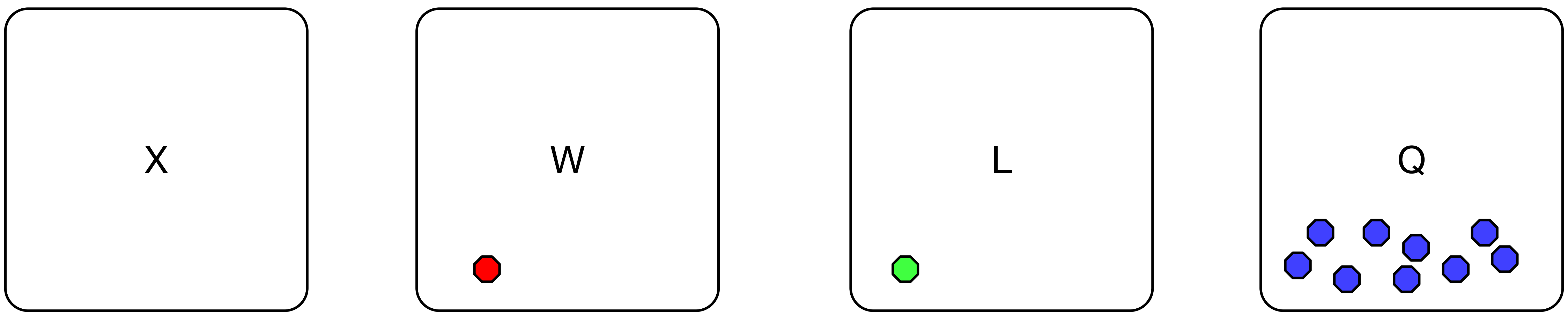
Combined Maximum and Minimum

- Can we do better?
  - Use a more sophisticated bin method
  - X - not looked at, W - won every comparison, L - lost every comparison, Q - at least one win and at least one loss



Combined Maximum and Minimum

- To be successful, need to move everything out of X and have only one element in W and L



- Otherwise can have a counter-example

Combined Maximum and Minimum

- Just counting the moves is not sufficient
  - Example:
    - We compare an element $w \in W$ with an element $l \in L$
    - Possibly: $w < l$
      - And we move both elements to the Q bucket
  - So, possible to move all n elements out of X into $W \cup L$ in $n/2$ comparisons and $n-2$ elements out of $W \cup L$ into Q in $n/2 - 1$ comparisons
  - Only gives $n-1$ moves!

Combined Maximum and Minimum

- Use an **adversary** argument
  - Algorithm can only depend on the knowledge of the previous comparisons when making a decision
- An adversary is allowed to change all values as long as the results of the comparisons stay the same
  - If $w \in W$ and $l \in L$, then the only thing the algorithm knows is that w has won all of its comparisons and l has lost all of its comparisons
  - Adversary therefore is allowed to change the value of l downward
  - Adversary guarantees that $w > l$.

Combined Maximum and Minimum

- With the help of the adversary who substitutes values when needed
- Potential: $\frac{3}{2}|X| + |W| + |L|$
    - Calculate net changes for comparisons between buckets

Combined Maximum and Minimum

- Compare X with X
    - Net change (-2, 1, 1, 0)
        - Potential change: 1

Combined Maximum and Minimum

- Compare X with W
  - Case 1: $x \in X, w \in W, x < w$ Net change (-1,0,1,0)
  - Case 2: $x \in X, \ w \in W, \ x > w$ Net change(-1,0,0,1)
  - The adversary can prevent Case 2 by decreasing x
    - Possible because this is the first time that we look at x
- Potential changes by $\frac{1}{2}$

Combined Maximum and Minimum

- Compare X with L
    - similar as before

Combined Maximum and Minimum

- Compare X with Q
  - The element in X changes to either W or L
    - Net change (-1, 1, 0, 0) or (-1, 0, 1, 0)
    - Potential change $\frac{1}{2}$

Combined Maximum and Minimum

- Compare W with W
  - One element looses
  - Net change (0, -1, 0, 1)
  - Potential change 1

Combined Maximum and Minimum

- Compare W with L
  - Adversary guarantees that the element in W wins by making <u>all</u> of them bigger
  - This works because each element in W has only seen wins and that does not change if the elements are made bigger.
  - No change

Combined Maximum and Minimum

- Compare W with Q
    - Since the elements in W have always won, the adversary can make them larger
    - No net change

Combined Maximum and Minimum

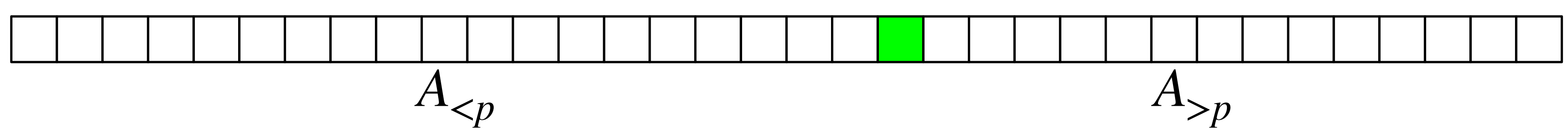
- Comparisons with L are the same as with W
- Comparisons within Q are useless, but make no changes

Combined Maximum and Minimum

- With the help of the adversary
  - Potential changes by at most 1
- Initial Potential: $\frac{3}{2}n$
- Final Potential: 2
- Need at least $\dfrac{3n-4}{2}$ comparisons

Selection

- Find the k^{th} largest element
  - Algorithm 1: Use the idea of quicksort
    - Find a random pivot and partition around it



    - Now use recursion:
      - If $k \leq$ len($A_{>p}$) find the k^{th} largest element in $A_{>p}$
      - If $k =$ len($A_{>p}$) $+ 1$, select p
      - If $k >$ len($A_{>p}$), find the $k-$len($A_{<p}$) $- 1$ largest element in $A_{<p}$

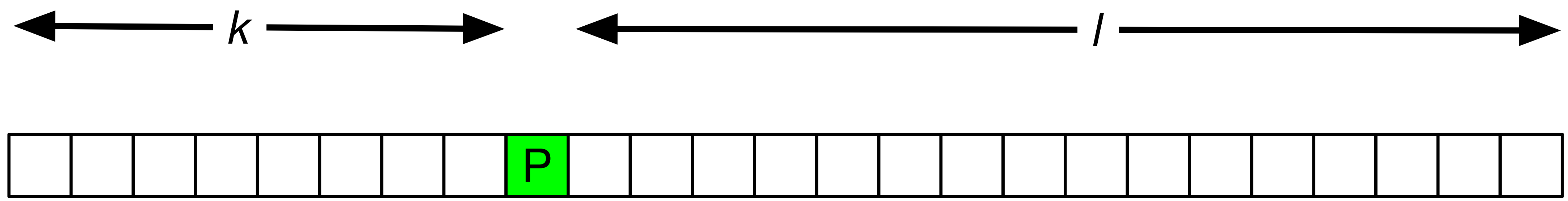
Selection

- Worst case behavior:
  - Pivot is always the maximum
  - Search in array of length one less
  - Partitioning an array of length takes $\Theta(n)$ time
    - Worst time: $\sim n + (n-1) + (n-2) + \ldots + 2 + 1$
    - $= \dfrac{n(n+1)}{2}$
    - $= \Theta(n^2)$

Selection

- Expected behavior:
  - Let $T(n)$ be the expected run-time on input array n
  - How does the pivot fall in an array?

Selection



- Call either $T(k)$ or $T(l) = T(n-k-1)$ or are done
- Bad luck assumption:
  - its always the one for the larger array
- All positions of the pivot are equally probable

Selection

- Gives a recurrence
  - $$T(n) \leq 2 \sum_{i=\lfloor n/2 \rfloor}^{n-1} \frac{1}{n} T(i) + dn$$
  - where dn is the costs of partitioning
- Now assume that $T(n) \leq cn$

Selection

Then:

$$
\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} \frac{1}{n} T(i) + dn \\
&\leq \frac{2c}{n} \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor n/2 \rfloor} i \right) + dn \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + dn \\
&\leq \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + dn
\end{aligned}
$$

Selection

$$\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + dn$$

$$= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4-3n/2+2}{2}\right) + dn$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + dn$$

$$= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + dn$$

Selection

$$= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + dn$$

$$= cn - \left(\frac{cn}{4} - \frac{c}{2} - dn\right)$$

which is $\leq$ cn if and only if

Selection

$$\frac{cn}{4} - \frac{c}{2} - dn \geq 0$$

$$\iff cn \geq 2c + 4dn$$

$$\iff c \geq 2c/n + 4d$$

If we assume $n \geq 4$, then the right side is at most $\dfrac{c}{2} + 4d$

Thus, if $c > 8d$ then the previous calculation goes through

Selection

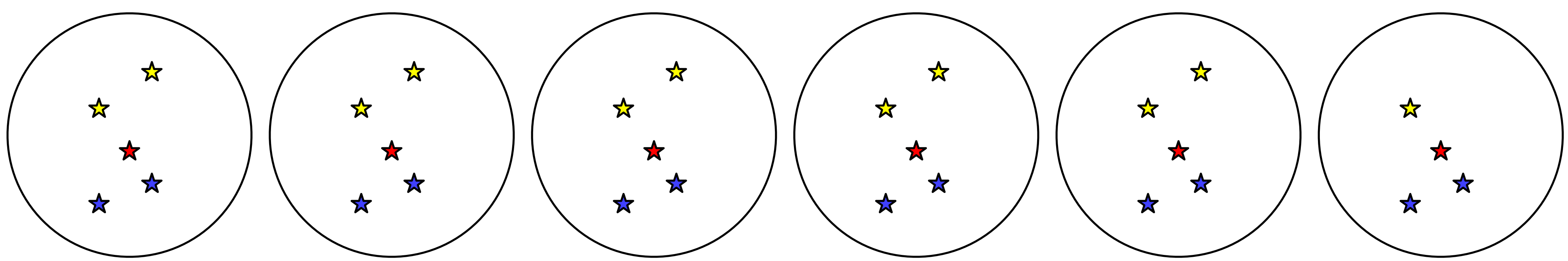
- We have shown
    - $T(n) < Cn$ if $n \geq 4$ and $C \geq 8d$
- Make C larger if necessary to obtain
    - $T(1) \leq C, T(2) \leq 2C, T(3) \leq 3C, T(4) \leq 4C$
- Then: Induction base works and Induction hypothesis works.
- So: expected runtime is linear
- But: we can do better

Selection

- Linear worst case selection
  - Idea: Improve the selection of the pivot!
  - Need to take at most linear time for the pivot selection

Selection

- Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of five elements and possibly one additional group
- In each group, choose the median (middle element)
    - In the last one, you might need to break a tie



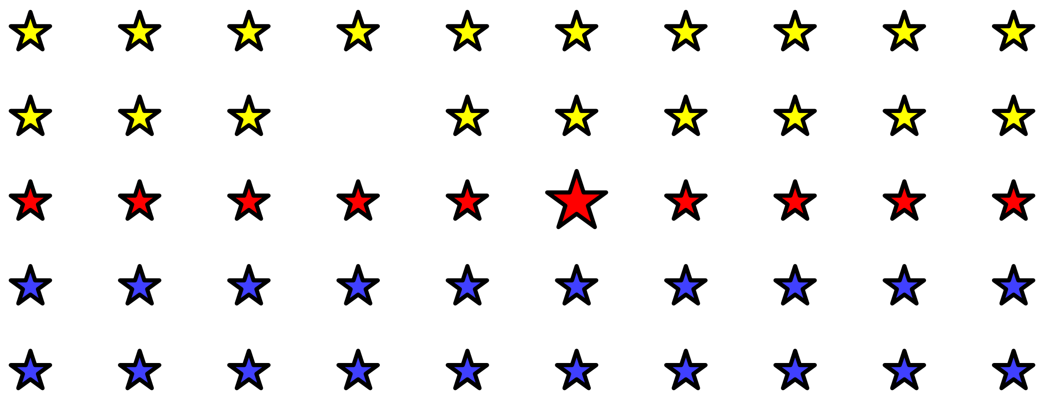
- Then select the median of the medians by **recurrence**

Selection

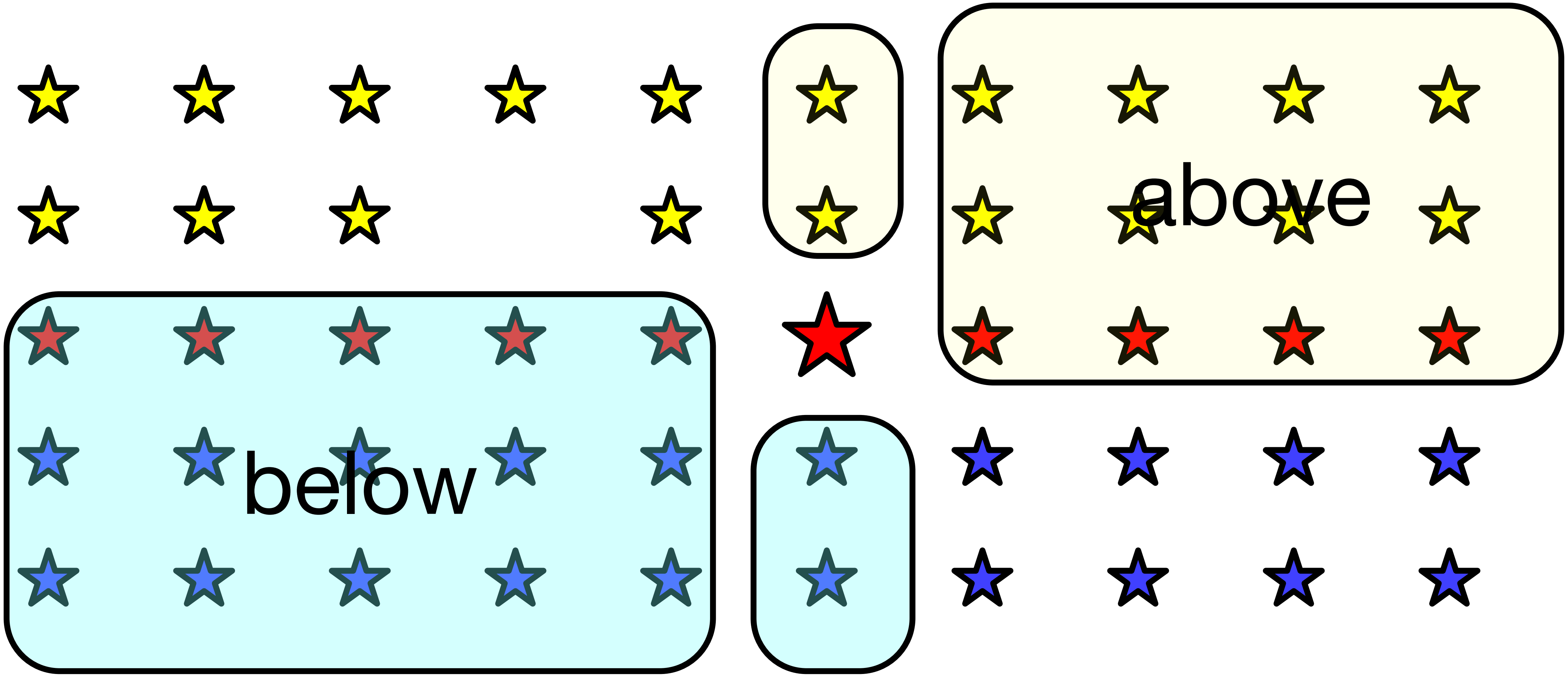
- Show that the median of medians divides the array fairly well
- Show that adding up the costs, we still are linear

Selection

- About half the medians are below the median of medians
- About half the medians are atop of the median of medians
- This allows us to guarantee that a certain number of elements is below and a certain number of elements is above the median of medians



Selection



A number of elements are below and above the median of medians <u>for sure.</u>

Selection

- At least half of the medians are greater or equal than the median of medians
- At least half of the $\lceil n/5 \rceil$ contributes at least three elements that are larger
    - Discard the group that is smaller and the group with the median of median
- The number of elements larger than the median of medians is at least
    - $3\left(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil - 2\right)$

Selection

- $3\left(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil - 2\right) \geq \frac{3n}{10} - 6$ larger than the median of medians
- $3\left(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil - 2\right) \geq \frac{3n}{10} - 6$ smaller than the median of medians

Selection

- $T(n)$ run time of the algorithm
  - Division into groups of five: $\Theta(n)$
  - Determination of the medians: $\Theta(n)$ because there are $\Theta(n)$ groups and we sort them in constant time to get the median
  - Determination of the median of median by recurrence $T(\lceil \frac{n}{5} \rceil)$
  - Partitioning around the median of medians $\Theta(n)$
  - Recursive call on at most $n - \frac{3n}{10} - 6 = \frac{7n}{10} + 6$ elements

Selection

- Total runtime:
  - $T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(0.7n + 6) + an$
- Show that this is linear using induction / substitution
- Again: induction step only needs to work for large enough n

Selection

$$T(n) \leq c(\frac{n}{5} + 1) + c(\frac{7n}{10} + 6) + an$$

$$= 0.9cn + 7c + an$$

This is at most cn if and only if $7c + an \leq 0.1cn$.

Since $7c + an \leq 0.1cn \iff \frac{70}{n}c + 10a \leq c$, we assume $n > 140$ so that c needs to be larger than $20a$.

Selection

- We also need to make c larger than $T(1)$, $T(2)/2$, …, $T(140)/140$
- Then we have an induction base on 140 values
- And an induction step that works
- So $T(n) \leq cn$

Selection

- This algorithm makes no assumptions on the input
- Unless our results on linear sorting