Searching, Sorting, and Lower Bounds

Searching
- Sequential search
- Binary search
- Interpolation search

Sorting and Lower Bounds
- Insertion sorts
- Lower bounds on sorting
- Heapsort
- Distribution sorts
- Lower bounds on selection
Sequential Search of an Ordered List

Problem definition

**Input:** \( x \), and \( a_1 < \ldots < a_n \)

\( a_0 \) defined to be \(-\infty\) and \( a_{n+1} \) defined to be \(+\infty\)

**Output:** \( i \) such that \( x = a_i \), 0 if no such \( i \) exists

Sequential Search

\[
i \leftarrow 1 \\
\text{while } (i < n) \text{ and } (x > a_i) \text{ do } i \leftarrow i + 1 \\
\text{if } x \neq a_i \text{ then } i \leftarrow 0 \\
\text{return } i
\]

- Where are *element comparisons* performed?

- What is the fewest number of element comp’s when \( n = 3 \)? 2
  What is the worst-case number of element comp’s when \( n = 3 \)? 3
  What is the worst-case number of element comp’s for general \( n \)?
Sequential search of an Ordered list

Average-Case Analysis

i ← 1
while (i < n) and (x > a_i) do i ← i + 1
if x ≠ a_i then i ← 0
return i

- Case 1: x is in the list
  - AVG = \( \sum_{i=1}^{n} \text{Prob}(x = a_i) \times (\# \text{ of comp's to find } x) \)

  Assume \( x \) equally likely to be any of the \( a \)'s
  so \( \text{Prob}(x = a_i) = \frac{1}{n} \)

  if \( x = a_i \), then the \# of comparisons to find \( x \) is
  \( i + 1 \) (when \( i < n \)) and \( n \) (when \( i = n \))

  AVG = \( \frac{1}{n} \sum_{i=1}^{n-1} (i + 1) + \frac{1}{n} n \)
  = \( \frac{1}{n} \left( \frac{n(n+1)}{2} - 1 \right) + 1 \)
  = \( \frac{n}{2} + \frac{3}{2} - \frac{1}{n} \)
Sequential search of an Ordered list

Average-Case Analysis

\[ i \leftarrow 1 \]
\[ \text{while } (i < n) \text{ and } (x > a_i) \text{ do } i \leftarrow i + 1 \]
\[ \text{if } x \neq a_i \text{ then } i \leftarrow 0 \]
\[ \text{return } i \]

• Case 2: \( x \) is not in the list
  ► AVG = \( \sum_{i=1}^{n+1} \text{Prob}(a_{i-1} < x < a_i) \times (\# \text{ of comp's to find } x) \)

  ► Assume \( x \) equally likely to be in each of the \( n+1 \) intervals
    so \( \text{Prob}(a_{i-1} < x < a_i) = \frac{1}{n+1} \)

  ► if \( a_{i-1} < x < a_i \), then the \# of comp’s to find \( x \) is
    \( i + 1 \) (when \( i < n \)) and \( n \) (when \( i \geq n \))

  ► AVG = \( \frac{1}{n+1} \sum_{i=1}^{n-1} (i + 1) + \frac{1}{n+1} \sum_{i=n}^{n+1} n \)

    = \( \frac{1}{n+1} \left( \frac{n(n+1)}{2} - 1 \right) + \frac{2n}{n+1} \)

    = \( \frac{n}{2} + 2 - \frac{3}{n+1} \)
Sequential search of an Ordered list

Average-Case Analysis

\[ i \leftarrow 1 \]
\[ \text{while } (i < n) \text{ and } (x > a_i) \text{ do } i \leftarrow i + 1 \]
\[ \text{if } x \neq a_i \text{ then } i \leftarrow 0 \]
\[ \text{return } i \]

- Case 3: 50\% case 1 and 50\% case 2
  
  \[ \text{AVG} = \frac{1}{2} \left( \frac{n}{2} + \frac{3}{2} - \frac{1}{n} \right) + \frac{1}{2} \left( \frac{n}{2} + 2 - \frac{3}{n+1} \right) \]
  
  \[ = \frac{n}{2} + \frac{7}{4} - O\left(\frac{1}{n}\right) \]
Binary Search of an Ordered List

Problem definition

**Input:** $x$, and $a_1 < \ldots < a_n$  
$x_0$ defined to be $-\infty$ and $a_{n+1}$ defined to be $+\infty$  
**Output:** $i$ such that $x = a_i$, 0 if no such $i$ exists

Binary Search

\[
i \leftarrow 1 \\
j \leftarrow n \\
\text{while } i < j \text{ do} \\
\quad m \leftarrow \lfloor \frac{i+j}{2} \rfloor \\
\qquad \text{if } x > a_m \text{ then} \\
\qquad \quad i \leftarrow m + 1 \\
\qquad \text{else} \\
\qquad \quad j \leftarrow m \\
\text{if } x \neq a_i \text{ then} \\
\quad i \leftarrow 0 \\
\text{return } i
\]

Worst-Case Analysis

- $C(1) = 1$
- $C(2) = 2$
- $C(n) = 1 + C(\lceil \frac{n}{2} \rceil)$
  \[
  = 2 + C(\lceil \frac{n}{2^2} \rceil) \\
  \ldots \\
  = \lceil \log n \rceil + C(1) \\
  = \lceil \log n \rceil + 1
  \]
Average-Case Analysis

Let \( n = 2^k + r, \ 0 \leq r < 2^k \ (r < n/2) \)

Example: \( n = 2^3 + 2 \)

- Case 1: \( x \) is in the list (equal distribution)
  - Always do \( k \) iterations; sometimes \( \left( \frac{2r}{n} \right) \) of the time) one more and one final check for equality
  - \( \bar{C} = k + \frac{2r}{n} + 1 \), where \( k = \lfloor \lg n \rfloor \)

- Case 2: \( x \) is not in the list \((n + 1) \) intervals equally likely)
  - Each interval goes with the succeeding list member
    - (The last interval goes with the last list member)
    - The last list member never requires an extra iteration
  - \( \bar{C} = k + \frac{2r}{n+1} + 1 \), where \( k = \lfloor \lg n \rfloor \)
Interpolation Search


- Assumption: values \( \{a_i\} \) are distributed relatively uniformly

Example: To look up *cat* in a dictionary, expect to find *cat* near beginning

\[
i \leftarrow 1; \quad j \leftarrow n
\]

\[
L_0 \leftarrow a_i; \quad H_1 \leftarrow a_j
\]

if \( x < L_0 \) then return 0

if \( x \geq H_1 \) then \( i \leftarrow j \)  // loop invariant: \( x \geq L_0 \) and \( x < H_1 \)

while \( i < j \) do

\[
m \leftarrow \left\lfloor i + (j - i) \frac{x - L_0}{H_1 - L_0} \right\rfloor  \quad // \text{loop invariant: } m \geq i \text{ and } m < j
\]

\[
MID \leftarrow a_{m+1}
\]

if \( x < MID \) then

\[
j \leftarrow m; \quad H_1 \leftarrow MID
\]

else

\[
i \leftarrow m + 1; \quad L_0 \leftarrow MID
\]

if \( x \neq a_i \) then \( i \leftarrow 0 \)

return \( i \)

- Average case complexity \( O(\lg \lg n) \)
- Worst case complexity \( O(n) \)

Example: 1, 2, 3, 4, 5, \ldots, 999, 1000, 10^9

Look for 1000
Sorting Algorithms

Straight Insertion Sort

for $j \leftarrow 2$ to $n$ do
  $i \leftarrow j - 1$
  $K \leftarrow K_j$
  while $(i \geq 1)$ and $(K < K_i)$ do
    $K_{i+1} \leftarrow K_i$
    $i \leftarrow i - 1$
  
  $K_{i+1} \leftarrow K$

- $C = WC$ # of comp's $= \sum_{j=2}^{n} (j - 1) = \frac{n(n-1)}{2} = \theta(n^2)$
- $\bar{C} = \text{avg # of comp's} \approx \frac{1}{2}C$
- $M = \# \text{ of record moves} = \theta(n^2)$
- $T = WC$ total time $= \theta(n^2)$
- $S = \text{space other than I/O} = \theta(1)$

Binary Insertion Sort Use binary search to find correct place
- $O(\lg j)$ comp's for $j$-th insertion, but still $\theta(j)$ record moves
- $C, \bar{C} \in \theta(n \lg n)$, $M, T \in \theta(n^2)$
- can use linked lists to avoid $O(j)$ record moves but then cannot use binary search!
**Sorting Algorithms**

**Shellsort** [Knuth Vol 3, pp.84-95; Baase pp.197-200]

- $\delta$-sort: Divide input records into $\delta$ tables, and sort each table by straight insertion

  ```plaintext
  for table ← 1 to $\delta$ do
    $j$ ← table + $\delta$
    while $j \leq n$ do
      $i$ ← $j - \delta$
      $K$ ← $K_j$
      while ($i \geq 1$) and ($K < K_i$) do
        $K_{i+\delta}$ ← $K_i$
        $i$ ← $i - \delta$
      $K_{i+\delta}$ ← $K$
      $j$ ← $j + \delta$
  ```

- Do $\delta$-sort for a decreasing sequence of $\delta$-values, with last value = 1
  - If $\delta = \{2^i - 1\}$ then $T = \theta(n^{3/2})$
  - Example: for $n = 40$, $\{\delta\} = \{31,15,7,3,1\}$
  - If $\delta = \{2^p3^q\}$ then $T = \theta(n \lg^2 n)$
  - Example: for $n = 40$, $\{\delta\} = \{36,32,27,24,18,16,12,9,8,6,4,3,2,1\}$
Lower Bounds for Sorting – Inversions

An inversion within a sequence of values is a pair of values that are "out of order" relative to each other.

Example: for sequence (3,14,1,5,9), the pair (3,1) is an inversion.

Theorem. Any sorting algorithm, that removes at most one inversion after each step, requires at least \( \frac{n(n-1)}{2} \) steps in the worst case and at least \( \frac{n(n-1)}{4} \) steps on average.

Proof.
- Consider the \( n! \) sequences containing integers \( I = \{1, 2, \ldots, n\} \).
- Define reverse of \( A = (a_1, a_2, \ldots, a_n) \) to be \( A^R = (a_n, a_{n-1}, \ldots, a_1) \).
- Note: \( A \) is the reverse of \( A^R \) and, when \( n > 1 \), no sequence is its own reverse.
- So the \( n! \) sequences can be partitioned into \( \frac{n!}{2} \) sequence pairs, each pair consisting of a sequence and its reverse.
- There are \( \frac{n(n-1)}{2} \) value-pairs \((i,j)\) such that \( i > j \) and \( i, j \in I \).
- For each such value-pair \((i,j)\) and for each sequence-pair \( \{A, A^R\} \), \((i,j)\) will be an inversion in exactly one of \( \{A, A^R\} \).
- So, total # of inversions among both \( A \) and \( A^R \) is exactly \( \frac{n(n-1)}{2} \) and the average number of inversions is exactly \( \frac{n(n-1)}{4} \).
Theorem. During a sorting process, the sum of the distances travelled by the records of a random permutation is \((n^2 - 1)/3\).

- Therefore, any sorting algorithm which moves records only a constant number of positions in one step requires \(\Omega(n^2)\) steps.

Proof.

- Let \(A = (a_1, a_2, \ldots, a_n)\) be a random permutation of \((1, 2, \ldots, n)\).
- Then \(|a_j - j|\) is the distance travelled by record \(j\) during sorting.
- \(a_j\) can equally likely be any of \(\{1, 2, \ldots, n\}\), so the expected value, \(E\), of \(|a_j - j|\) is
  \[
  E = \frac{1}{n}(|1 - j| + |2 - j| + \ldots + |j - j| + \ldots + |n - j|)
  \]
  \[
  = \frac{1}{n} \left( \sum_{i=1}^{j-1} i + \sum_{i=1}^{n-j} i \right) = \frac{1}{n} \left( \frac{(j-1)j}{2} + \frac{(n-j)(n-j+1)}{2} \right)
  \]
- The sum, \(S\), of the distances travelled by the records during sorting is
  \[
  S = \sum_{j=1}^{n} |a_j - j| = \frac{1}{n} \sum_{j=1}^{n} \left( \frac{(j-1)j}{2} + \frac{(n-j)(n-j+1)}{2} \right)
  \]
  (Note that the terms are the same for complementary values of \(j\))
  \[
  = \frac{1}{n} \sum_{j=1}^{n} (j - 1)j = \frac{1}{n} \sum_{j=1}^{n} (j^2 - j)
  \]
  \[
  = \frac{1}{n} \left( \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} \right) = \frac{n^2 - 1}{3}
  \]
**Theorem.** Any algorithm for sorting \( n \) elements by comparisons requires \( \Omega(n \log n) \) comparisons.

**Proof.**
- Any such algorithm, \( A \), can be modeled by a decision tree, \( T \)
- \( T \) has at least \( n! \) leaves, because the result of sorting \( n \) elements can be any of the \( n! \) permutations of the elements

- An easy proof by induction (HW??) shows that a binary tree (in which each non-leaf has two sons) of height \( h \) has \( \leq 2^h \) leaves therefore \( \text{height}(T) \geq \lg(n!) \)

- Since \( n! \geq n(n-1) \ldots \left\lfloor \frac{n}{2} \right\rfloor \geq \left( \frac{n}{2} \right)^{\frac{n}{2}} \) therefore \( \lg(n!) \geq \frac{n}{2} \lg \frac{n}{2} = 0.5n \lg n - .5n \), which is \( \Omega(n \log n) \)

- Note: Stirling’s approximation gives \( \lg(n!) \approx n \lg n - 1.44n \)
Heapsort

- A file \((K_1, \ldots, K_n)\) is a maxheap if, for \(1 < j \leq n\), \(K_{\lfloor j/2 \rfloor} \geq K_j\) (the heap criteria)
- Therefore, \(K_1 = \max\{K_i\}\)
- A heap can be thought of as the level-order traversal of an almost complete b.t. where each father has value \(\geq\) that of his sons

Example of maxheap:

```
          16
         /  \
        11   9
       /    /  \
      10   5   6  8
     /    /    /    \
    1    2    4    
```

corresponds to: \((16, 11, 9, 10, 5, 6, 8, 1, 2, 4)\)

Heapsort algorithm:

1. Transform arbitrary input file to become a maxheap (biggest # now first)
2. Exchange the first and last cells (biggest # now in correct place)
3. Fix up rest of the file to become a (smaller) heap and iterate
Heapsort – Heapify and Buildheap

Transform Input File to Become a Maxheap

- Initially, nodes at the lowest height (0) satisfy heap criteria
- Iterate rearranging nodes so each node $X$ at next height, $h$, also satisfies heap criteria ($X$'s value $\geq$ that of $X$'s sons)
  - If it is, then subtree headed by $X$ satisfies the heap criteria
  - If not, exchange $X$ with larger of $X$'s sons
  - If this messes up subtree (of height $h - 1$) then fix subtree

$\text{HEAPIFY}(i, n)$ assumes $i + 1, \ldots, n$ are okay; makes $i, \ldots, n$ okay

\[
j \leftarrow i \\
\text{while } j \leq n/2 \text{ do} \\
\text{the sons of } j \text{ are } 2j \text{ and } 2j + 1 \quad / / \text{ only one son if } n = 2j \\
k \leftarrow \text{son of } j \text{ with larger value} \\
\text{if } A[k] > A[j] \text{ then} \\
A[k] \leftrightarrow A[j] \\
j \leftarrow k \\
\text{else} \\
j \leftarrow n
\]

$\text{BUILDHEAP}(n)$ transforms input into heap

for $i \leftarrow n$ downto 1 do

$\text{HEAPIFY}(i, n)$
Heapsort – Heapify and Buildheap

Time Analysis:

- For heap of height $h$:

  HEAPIFY takes $T_H(h) \leq T_H(h - 1) + c$, so $T_H(h) = O(h)$

- BUILDHEAP calls HEAPIFY once for each node

- $N_h = \# \text{ nodes of height } h \leq \left\lceil \frac{n}{2^h} \right\rceil \leq 2\frac{n}{2^h}$

- So,

  $$T_B(n) \leq \sum_{h=1}^{\lfloor \log n \rfloor} T_H(h)N_h$$

  $$\leq \sum_{h=1}^{\lfloor \log n \rfloor} (c \cdot h)2\frac{n}{2^h}$$

  $$< 2cn \sum_{h=1}^{\infty} \frac{h}{2^h}$$

  $$= O(n)$$
Heapsort Algorithm:

\[
\text{HEAPSORT}(n) \\
\quad \text{BUILDHEAP}(n) \\
\quad \text{for } \text{size} \leftarrow n \text{ downto } 2 \text{ do} \\
\quad \quad \text{exchange } A[1] \leftrightarrow A[\text{size}] \\
\quad \quad \text{HEAPIFY}(1, \text{size}-1)
\]

Time Analysis:
- Time for HEAPIFY(1,size) is \(O(\lg \text{ size}) \leq O(\lg n)\)
- Time for Heapsort is \(O(n \lg n)\)

\[
\begin{align*}
C &= \theta(n \lg n) \\
M &= \theta(n \lg n) \\
S &= \theta(1) \\
T &= \theta(n \lg n)
\end{align*}
\]
Distribution Sorts – Bucket Sort \(^{(7)}\)

Not based on comparisons between keys, but on their representation.

**Bucket Sort** assumes key values are in known range \([1 : m]\)

\[
\begin{align*}
&\text{for } j \leftarrow 1 \text{ to } m: \quad L_j \leftarrow \Lambda \\
&\text{for } i \leftarrow 1 \text{ to } n: \quad \text{append } K_i \text{ to end of } L_{K_i} \\
&L \leftarrow \Lambda \\
&\text{for } j \leftarrow 1 \text{ to } m: \quad \text{concatenate } L_j \text{ to end of } L
\end{align*}
\]

- \(T = \theta(m + n)\)
- \(S = \theta(m + n)\)

**Example:** 2\(_a\), 7, 1, 8, 2\(_b\)
1 \rightarrow 1
2 \rightarrow 2\(_a\) \rightarrow 2\(_b\) (stable, see next slide)
3 \rightarrow
4 \rightarrow
5 \rightarrow
6 \rightarrow
7 \rightarrow 7
8 \rightarrow 8
9 \rightarrow
Not based on comparisons between keys, but on their representation

**Bucket Sort** assumes key values are in known range $[1 : m]$

- $T = \theta(m + n)$
- $S = \theta(m + n)$

- A sorting algorithm is **stable** if equal-valued keys retain their order after being sorted
  - If input record $R_i$ has key value $K_i$ and $R_j$ has key $K_j$, with $i < j$ and $K_i = K_j$
  - and, after sorting, record $R_i$ is in location $i'$ and $R_j$ in location $j'$
  - then it is guaranteed that $i' < j'$
Distribution Sorts – Bucket Sort

Not based on comparisons between keys, but on their representation

Bucket Sort assumes key values are in known range \([1 : m]\)

- \(T = \theta(m + n)\)
- \(S = \theta(m + n)\)

- A sorting algorithm is **stable** if equal-valued keys retain their order after being sorted
  - If input record \(R_i\) has key value \(K_i\) and \(R_j\) has key \(K_j\), with \(i < j\) and \(K_i = K_j\)
  - and, after sorting, record \(R_i\) is in location \(i'\) and \(R_j\) in location \(j'\)
  - then it is guaranteed that \(i' < j'\)

- Bucket Sort is a stable sorting algorithm (if implemented carefully)

- Maybe not very useful – **Example**: phone numbers
  - 100 students, 10-digit phone numbers \(\Rightarrow 10^{10} + 100 \approx 10^{10}\)
Distribution Sorts – Lexicographic (Radix) Sort

Approach

- Let keys be $K_1, \ldots, K_n$ where $K_i = d_{i,1}, \ldots, d_{i,r}$ expressed in base $M$
- Do a bucket sort on the least significant digit of the keys keeping those with equal digits in the same order
- Iterate on the next significant digit, etc.

Implementation

```
Q ← Λ
for j ← 0 to $M − 1$ do: $Q_j$ ← Λ
Q ← $K_1, \ldots, K_n$
for $t ← r$ downto 1 do
    while $Q ≠ Λ$ do // distribution phase
        $K_i$ ← Q
        $Q_{d_{i,t}}$ ← $K_i$
    for $j ← 0$ to $M − 1$ do // collecting phase
        $Q ← Q \mid Q_j$
        $Q_j$ ← Λ
    $T = θ(r(M + n))$
    $S = θ(M + n)$
    Example: phone numbers
        ▶ 100 students, 10-digit phone numbers ⇒ 10(10 + 100) ≈ 10^3
```
Radix Sort Example

- Keys: 3, 14, 1, 5, 9, 2, 6
- Expressed in base 3
- Collect buckets in order

```
BUCKET 0: 001, 002, 010, 012, 020
BUCKET 1: 100, 112
BUCKET 2:
```
Upper bound for selecting MaxMin

Algorithm to determine Max and Min of \( \{x_1, \ldots, x_n\} \)

- find Max (takes \( n - 1 \) comp’s)
- then find Min of rest (\( n - 2 \) comp’s)
- so \( 2n - 3 \) is an upper bound

Lower bound for selecting MaxMin  use an adversary argument

- Assume \( n \) distinct values, then \( n(n-1) \) possible answers
- Outcome of each comparison by an algorithm
  \( \Rightarrow \) fewer answers remain possible
- Adversary gives outcomes to comparisons, consistent with previous outcomes, trying to delay resolution of the problem
- Analyze the effects
Bounds for Selection – MaxMin

Lower bound for selecting MaxMin adversary argument

- Need \( n - 1 \) keys to lose at least one comparison (the \( n \)th key is Max)
- Similarly, need \( n - 1 \) keys to win at least one comparison
- Each first Loss and each first Win is a Unit of information so we need \( 2n - 2 \) Units
- The adversary gives outcomes that minimize Units given

Example: compare \( x : y \), if \( x \) prev lost a comp then ok to let \( x \) lose

- **Type-A comp** (2 keys never in any prev comp) gives 2 Units
- **Type-B comp** (any other comp) can be limited to give \( \leq 1 \) Unit
- Making \( a \) Type-A comps and \( b \) Type-B comps \( \Rightarrow \) get \( \leq 2a + b \) Units
- Fastest way to get Units is to use Type-A comps
- Impossible to use more than \( X = \lfloor \frac{n}{2} \rfloor \) Type-A comps, (why??)
- To get \( 2n - 2 \) Units,
  - best to use \( X \) Type-A comps and \( 2n - 2 - 2X \) Type-B comps
- So, total number of comps is
  \[
  C \geq (X) + (2n - 2 - 2X) = 2n - 2 - X = 2n - 2 - \lfloor \frac{n}{2} \rfloor
  \]
- Therefore lower bound is \( \text{LB} = 2n - 2 - \lfloor \frac{n}{2} \rfloor \geq 3n - 2 \)
- \( \text{UB} = 2n - 3 \), so there is a gap between upper and lower bounds (gap will be addressed later)
Bounds for Selection – Second Largest

Upper bounds for selecting Second Largest

- **Simple algorithm:**
  - find Max (takes \(n - 1\) comp’s)
  - then find Max of rest (\(n - 2\) comp’s)
  - so \(2n - 3\) is an upper bound

- **Tournament algorithm:**
  - pair off all keys, iterate pairing winners of previous rounds
  - Max is determined using \(n - 1\) comparisons, why??
  - \(\leq \lceil \lg n \rceil\) keys lost to Max (one per level)
  - any other key, \(k\), lost to some key (not Max), so \(k\) is not Second
  - find max of \(\lceil \lg n \rceil\) keys using \(\lceil \lg n \rceil - 1\) comp’s
  - total of \(\leq n + \lceil \lg n \rceil - 2\) comp’s (a better upper bound)
Bounds for Selection – Second Largest

Lower bound for selecting Second Largest

- Assume distinct values, use adversary argument
- Define the adversary as follows:
  - Give all keys a weight (fictitious value used by adversary)
  - Initially, for all keys $k$, $wt(k) \leftarrow 1$
  - When comparing keys $x : y$
    - if $wt(x) > wt(y)$ or $wt(x) = wt(y) > 0$
      - then
        - $wt(x) \leftarrow wt(x) + wt(y)$
        - $wt(y) \leftarrow 0$
        - return outcome $x > y$
      - otherwise give answer consistent with previous replies
- To find second largest, must determine largest — why??
  - So at least $n - 1$ comparisons resulting in new losers
- Will prove $\exists \geq \lceil \lg n \rceil$ distinct keys that lose directly against Max
Bounds for Selection – Second Largest

- Proving $\exists \geq \lceil \lg n \rceil$ distinct keys that lose directly against Max
  1. $\text{wt}(x) = 0$ iff $x$ has lost a comparison
  2. $\text{wt}(x) \neq 0 \Rightarrow x$ could still be Max
  3. $\sum_{i=1}^{n} \text{wt}(i) = n$ (total weight is preserved)
  4. When finished, only one key $x^*$ has non-zero weight (else $\geq 2$ possible answers) and so $x^*$ is Max
  5. $x^*$ has directly won against $\geq \lceil \lg n \rceil$ distinct keys
     - $\text{wt}(x^*) = n$ when algorithm finished
     - Let $W_k = \text{wt}(x^*)$ just after $k$th comp of $x^*$ against a prev undefeated key
     - $W_{k-1} \geq \text{wt}(\text{that } k\text{th key})$
     - $W_k = W_{k-1} + \text{wt(that } k\text{th key)}$
       - $\leq 2W_{k-1}$
     - $n = W_k \leq 2W_{k-1} \leq 2^2W_{k-2} \leq \ldots \leq 2^kW_0 = 2^k$
     - So, $k \geq \lg n$, but $k$ is integer and so $k \geq \lceil \lg n \rceil$

- So $\exists (n-1)$ distinct losers and $\exists \lceil \lg n \rceil$ keys lost to Max
- All but one of those $\lceil \lg n \rceil$ keys must lose again
- Gives lower bound $n-1 + \lceil \lg n \rceil - 1 = n + \lceil \lg n \rceil - 2$
Bounds for Selection – Median

[ Blum, Floyd, Pratt, Rivest, Tarjan ]

Lower bound for selecting Median

- Let $X^*$ be the median ($\lceil \frac{n}{2} \rceil$-th smallest value)
- To know that $X^*$ is the median, must have $n - 1$ crucial comp’s
  that ensure, for each $x$, it is known which of $x < X^*$ or $X^* < x$ holds
- Equivalently, must have a DAG of comp’s
  (where edge $x \rightarrow y$ means that a comp determined that $x < y$)
  with a path $x \leadsto X^*$ for elements for which can conclude $x < X^*$
  and a path $X^* \leadsto x$ for elements for which can conclude $X^* < x$
- All other comp’s (not used in this crucial DAG) are non-crucial
Bounds for Selection – Median

- Definition of the adversary
  - Partition elements into categories Small, Medium, and Large
  - Initially, all elements are Medium
  - When comparing \( a : b \)
    - \( a, b \) in different categories \( \Rightarrow \) answer accordingly
      (comparison might be crucial)
    - \( a, b \) both in Small or both in Large \( \Rightarrow \) answer consistently
      (comparison might be crucial)
    - if \( a, b \) both in Medium and are the last 2 in Medium
      \( \Rightarrow \) give answer \( a < b \) and place \( a \) in Small and \( b \) in Medium
      (comparison might be crucial)
    - if \( a, b \) both in Medium and at least 3 in Medium
      \( \Rightarrow \) answer that \( a < b \) and place \( a \) in Small and \( b \) in Large
      (comparison is non-critical)
- \( n \) odd \( \Rightarrow \) there are \( \geq \frac{n-1}{2} \) non-critical comparisons
- \( n \) even \( \Rightarrow \) there are \( \geq \frac{n-2}{2} \) non-critical comparisons
Bounds for Selection – Median

- Definition of the adversary
  - Partition elements into categories Small, Medium, and Large
  - Initially, all elements are Medium
  - When comparing \( a : b \)
    - \( a, b \) in different categories ⇒ answer accordingly
      (comparison might be crucial)
    - \( a, b \) both in Small or both in Large ⇒ answer consistently
      (comparison might be crucial)
    - if \( a, b \) both in Medium and are the last 2 in Medium
      ⇒ give answer \( a < b \) and place \( a \) in Small and \( b \) in Medium
      (comparison might be crucial)
    - if \( a, b \) both in Medium and at least 3 in Medium
      ⇒ answer that \( a < b \) and place \( a \) in Small and \( b \) in Large
      (comparison is non-crucial)
  - \( n \) odd ⇒ there are \( \geq \frac{n-1}{2} \) non-crucial comp’s
  - \( n \) even ⇒ there are \( \geq \frac{n-2}{2} \) non-crucial comp’s
  - Thus, always \( \geq \lfloor \frac{n-1}{2} \rfloor \) non-crucial comp’s
  - Adding the \( n - 1 \) crucial comp’s
    ⇒ \( LB = n - 1 + \lfloor \frac{n-1}{2} \rfloor \) comp’s