The Selection

Algorithm : Design & Analysis

[7]

In the last class…

- Heap Structure and Patial Order Tree Property
- The Strategy of Heapsort
- Keep the Partial Order Tree Property after the maximal element is removed
- Constructing the Heap
- Complexity of Heapsort
- Accelerated Heapsort

The Selection

- Finding *max* and *min*
- Finding the second largest key
- Adversary argument and lower bound
- Selection Problem – Median
- A Linear Time Selection Algorithm
- Analysis of Selection Algorithm
- A Lower Bound for Finding the Median

The Selection Problem

- Problem:
  - Suppose E is an array containing n elements with keys from some linearly order set, and let k be an integer such that $1 \le k \le n$. The selection problem is to find an element with the kth smallest key in E.
- A Special Case
  - Find the max/min – $k=n$ or $k=1$

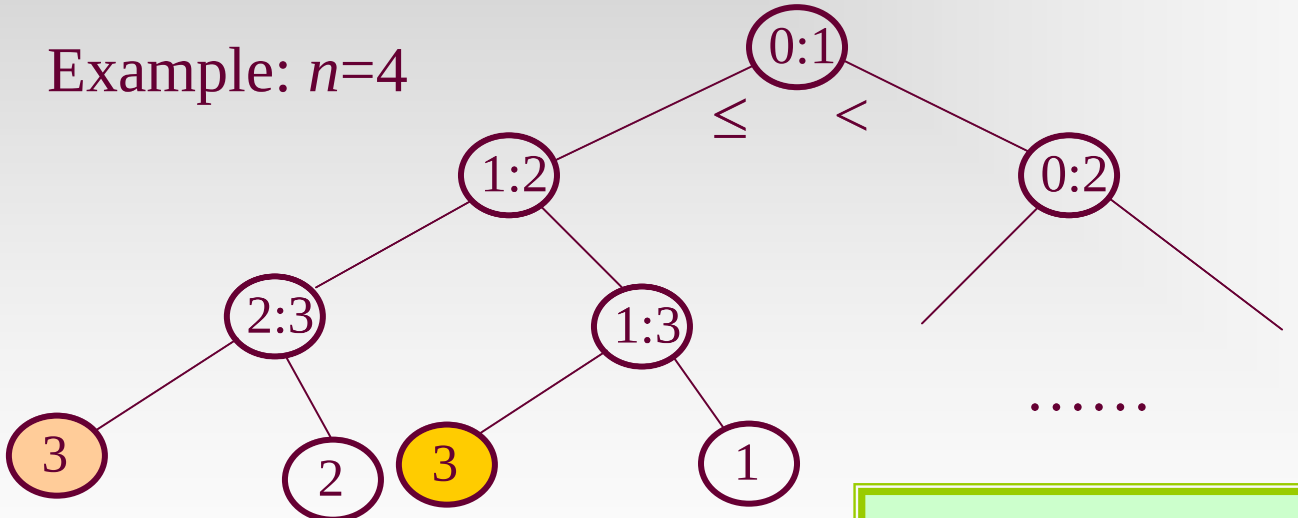
Lower Bound of Finding the Max

- For any algorithm $\mathcal{A}$ that can compare and copy numbers exclusively, in the worst case, $\mathcal{A}$ can't do fewer than ***n-1*** comparisons to find the largest entry in an array with *n* entries.
  - Proof: an array with *n* distinct entries is assumed. We can exclude a specific entry from being the largest entry only after it is determined to be "loser" to at least one entry. So, *n*-1 entries must be "losers" in comparisons done by the algorithm. However, each comparison has only one loser, so at least *n*-1 comparisons must be done.

Decision Tree and Lower Bound

Since the decision tree for the selection problem must have at least n leaves, the height of the tree is at least $\lceil \lg n \rceil$. **It's not a good lower bound.**

Example: n=4



There are more than n leaves!
In fact, 2^{n-1} leaves at least.

Finding *max* and *min*

- The strategy
  - Pair up the keys, and do $n/2$ comparisons(if n odd, having E[n] uncompared);
  - Doing findMax for larger key set and findMin for small key set respectively (if n odd, E[n] included in both sets)
- Number of comparisons
  - For even n: $n/2+2(n/2-1)=3n/2-2$
  - For odd n: $(n-1)/2+2((n-1)/2+1-1)=\lceil 3n/2 \rceil-2$

Unit of Information

- That x is *max* can only be known when it is sure that every key other than x has lost some comparison.
- That y is *min* can only be known when it is sure that every key other than y has win some comparison.
- Each win or loss is counted as one unit of information, then *any* algorithm must have at least $2n$-2 units of information to be sure of specifying the *max* and *min*.

Adversary Strategy

Status of keys *x* and *y* Compared by an algorithm	Adversary response	New status	Units of new information
N,N	$x>y$	W,L	2
W,N or WL,N	$x>y$	W,L or WL,L	1
L,N	$x<y$	L,W	1
W,W	$x>y$	W,WL	1
L,L	$x>y$	WL,L	1
W,L or WL,L or W,WL	$x>y$	No change	0
WL,WL	Consistent with Assigned values	No change	0

The principle: let the key win if it never lose, or,
let the key lose if it never win, and
change one value if necessary

Lower Bound by Adversary Strategy

- Construct a input to force *the* algorithm to do more comparisons as possible, that is, to give away as few as possible units of new information with each comparison.
- It can be achieved that 2 units of new information are given away only when the status is N,N.
- It is ***always*** possible to give adversary response for other status so that at most one new unit of information is given away, ***without any inconsistencies***.
- So, the ***Lower Bound*** is $n/2+n-2$(for even n)

An Example Using Adversary

Comparison	x_1		x_2		x_3		x_4		x_5		x_6	
	S	V	S	V	S	V	S	V	S	V	S	V
x_1,x_2	W	20	L	10	N	*	N	*	N	*	N	*
x_1,x_5	W	20							L	5		
x_3,x_4					W	15	L	8				
x_3,x_6					W	15					L	12
x_3,x_1	WL	20			W	25						
x_2,x_4			WL	10			L	8				
x_5,x_6									WL	5	L	3
x_6,x_4							L	2			WL	3

Now, x_3 is the only one which never loses, so, Max is x_3

Now, x_4 is the only one which never wins, so, x_4 is Min

8 Comparisons !

The lower bound is 7.

Raising/lowering the value according to strategy

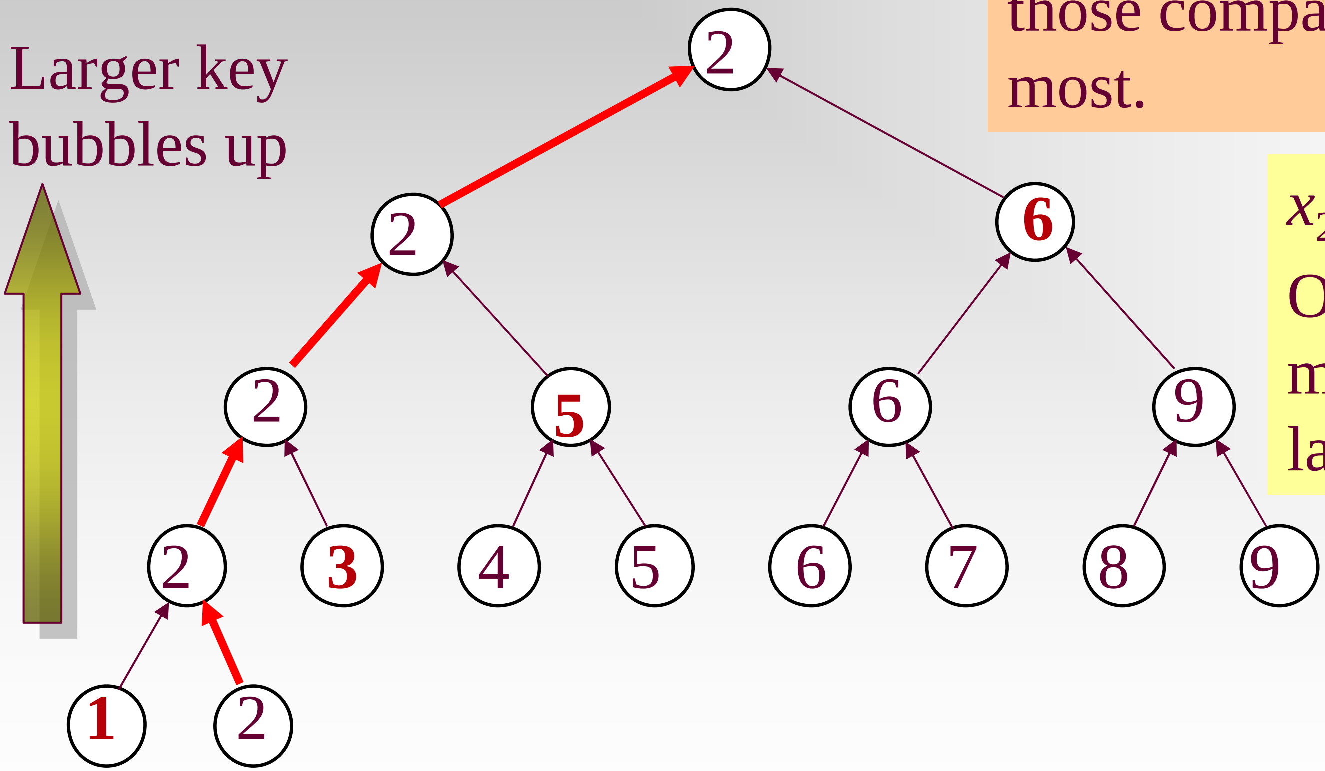
Finding the Second-Largest Key

- Using FindMax twice is a solution with 2*n*-3 comparisons.
- For a better algorithm, the idea is to collect some useful information from the first FindMax to decrease the number of comparisons in the second FindMax.
- Useful information: the key which lost to a key other than *max* cannot be the second-Largest key.
- The worst case for twice ***FindMax*** is "No information".(x_1 is Max)

Second Largest Key by Tournament

The length of the longest path is $\lceil \lg n \rceil$, as many as those compared to *max* at most.

Larger key bubbles up



x_2 is *max*

Only x_1, x_3, x_5, x_6 may be the second largest key.

Analysis of Finding the Second

- Any algorithm that finds *secondLargest* must also find *max* before. (n-1)
- The *secondLargest* can only be in those which lose directly to *max*.
- On its path along which bubbling up to the root of tournament tree, *max* beat $\lceil \lg n \rceil$ keys at most.
- Pick up *secondLargest*. ($\lceil \lg n \rceil$-1)
- $\boldsymbol{n+\lceil \lg n \rceil-2}$

Lower Bound by Adversary

- Theorem
  - Any algorithm (that works by comparing keys) to find the second largest in a set of *n* keys must do at least $n+\lceil \lg n \rceil-2$ comparisons in the worst case.
  - Proof

    There is an adversary strategy that can force any algorithm that finds *secondLargest* to compare *max* to $\lceil \lg n \rceil$ distinct keys.

Weighted Key

Note: for one comparison, the weight increasing is no more than doubled.

- Assigning a weight $w(x)$ to each key. The initial values are all 1.
- Adversary rules:

Case	Adversary reply	Updating of weights
$w(x)>w(y)$	x>y	$w(x):=w(x)+w(y);\ w(y):=0$
$w(x)=w(y)>0$	$x>y$	$w(x):=w(x)+w(y);\ w(y):=0$
$w(y)>w(x)$	$y>x$	$w(y):=w(x)+w(y);\ w(x):=0$
$w(x)=w(y)=0$	Consistent with previous replies	No change

Zero=Loss

Lower Bound by Adversary: Details

- Note: the sum of weights is always *n*.
- Let *x* is *max*, then *x* is the only nonzero weighted key, that is $w(x)=n$.
- By the adversary rules:

$$w_k(x) \leq 2w_{k-1}(x)$$

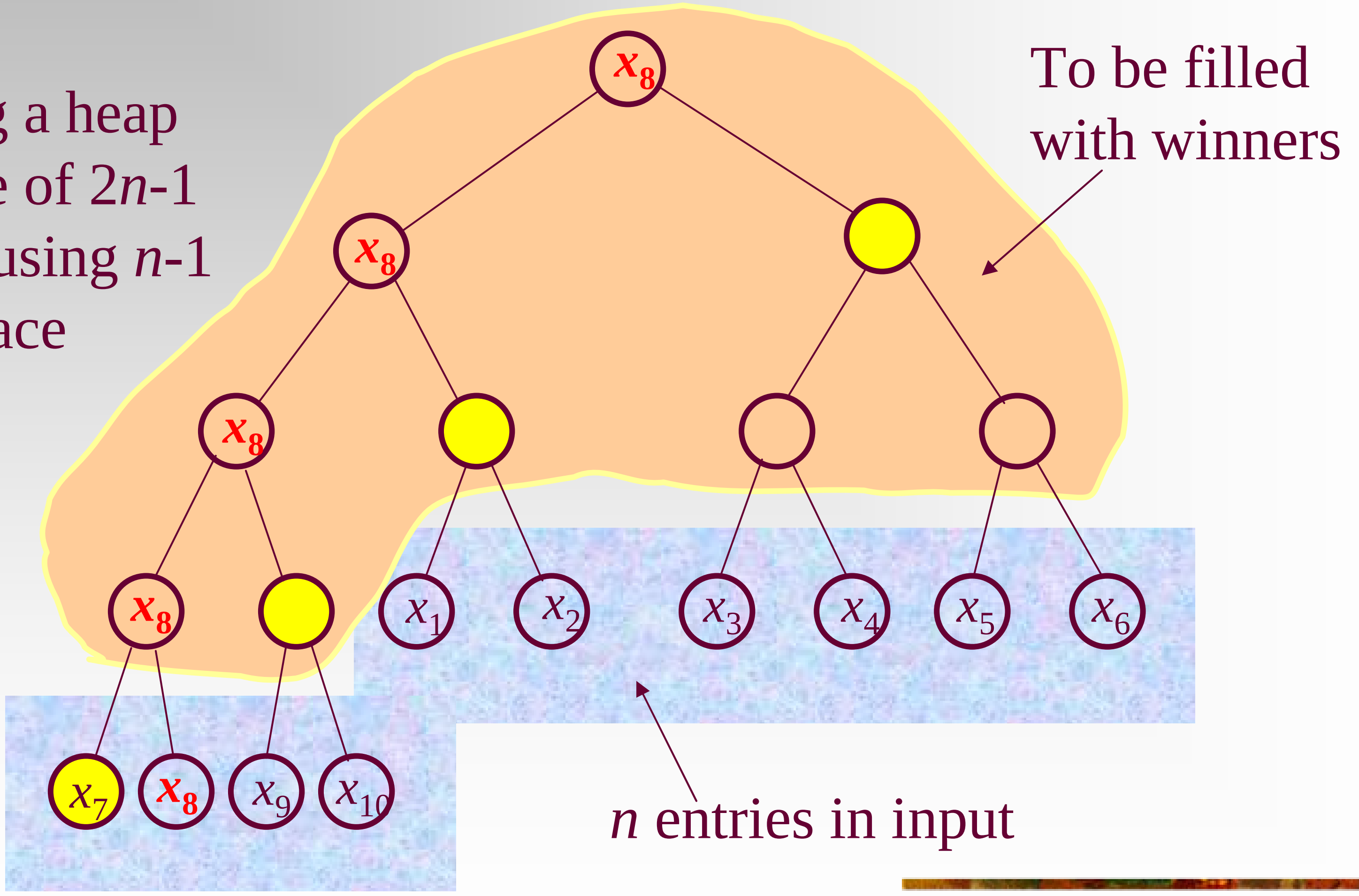
- Let *K* be the number of comparisons *x* wins against previously undefeated keys:

$$n=w_K(x) \leq 2^K w_0(x)=2^K$$

- So, $K \geq \lceil \lg n \rceil$

Tracking the Losers to *MAX*

Building a heap structure of 2*n*-1 entries, using *n*-1 extra space



Finding the Median: the Strategy

- Obervation: If we can partition the problem set of keys into 2 subsets: S1, S2, such that any key in S1 is smaller that that of S2, then the median must located in the set with more elements.
- Divide-and-Conquer: only one subset is needed to be processed recursively.

Adjusting the Rank

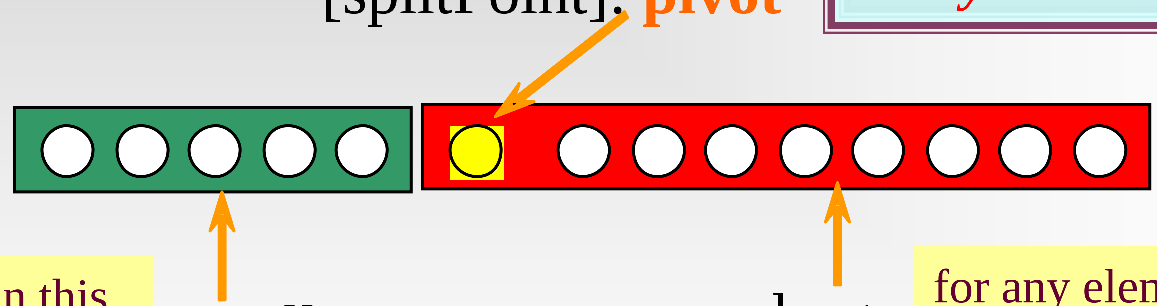
- The rank of the median (of the original set) in the subset considered can be evaluated easily.
- An example
  - Let n=255
  - The rank of median we want is 128
  - Assuming $|S_1|$=96, $|S_2|$=159
  - Then, the original median is in S_2, and the new rank is 128-96=32

Partitioning: Larger and Smaller

- Dividing the array to be considered into two subsets: "small" and "large", the one with more elements will be processed recursively.

A "bad" pivot will give a very uneven partition!

[splitPoint]: **pivot**



for any element in this segment, the key is ***less than pivot***.

small

large

for any element in this segment, the key is ***not less than pivot***.

To be processed recursively

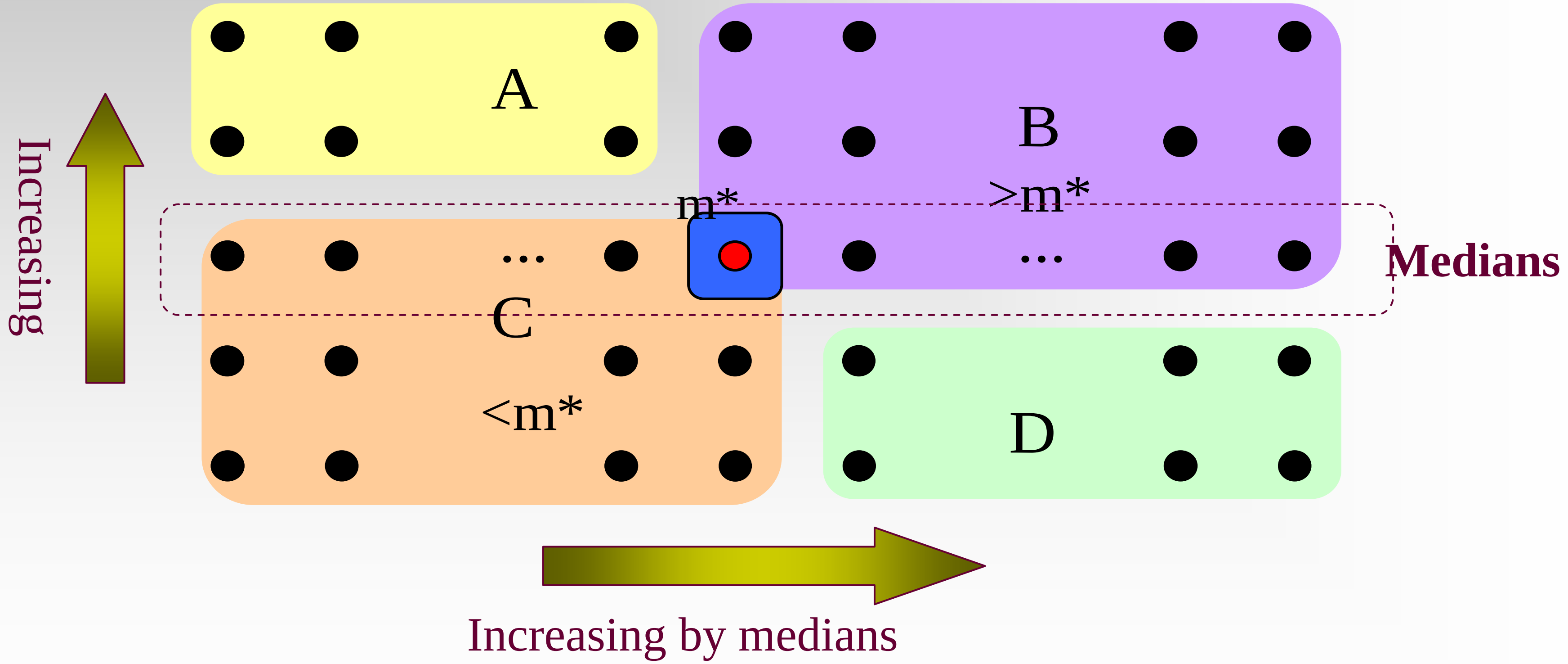
Selection: the Algorithm

- Input: S, a set of n keys; and k, an integer such that $1 \le k \le n$.
- Output: The kth smallest key in S.
- Note: Median selection is only a special case of the algorithm, with $k = \lceil n/2 \rceil$.
- Procedure
- Element select(SetOfElements S, **int** k)
  - **if** ($|S| \le 5$) **return** direct solution; **else**
  - Constructing the subsets S_1 and S_2;
  - Processing one of $\mathbf{S_1, S_2}$ with more elements, recursively.

There is the same question with quicksort-imbalanced partition

Partition Improved: the Strategy

All the elements are put in groups of 5



Constructing the Partition

- Find the m^*, the median of medians of all the groups of 5, as illustrated previously.
- Compare each key in sections A and D to m^*, and
  - Let $S_1 = C \cup \{x | x \in A \cup D \text{ and } x < m^*\}$
  - Let $S_2 = B \cup \{x | x \in A \cup D \text{ and } x > m^*\}$

(m^* is to be used as the pivot for the partition)

Divide and Conquer

if ($k=|S_1|+1$)

    return m^*;

else if ($k \le |S_1|$)

    return select(S_1,k); //recursion

else

    return select(S_2,$k-|S_1|-1$); //recursion

Counting the Number of Comparisons

- For simplicity:
  - Assuming n=5(2r+1) for all calls of *select*.
- $$W(n) \le 6\left(\frac{n}{5}\right) + W\left(\frac{n}{5}\right) + 4r + W(7r+2)$$

The extreme case: all the elements in A∪D in one subset.

Finding the median in every group of 5

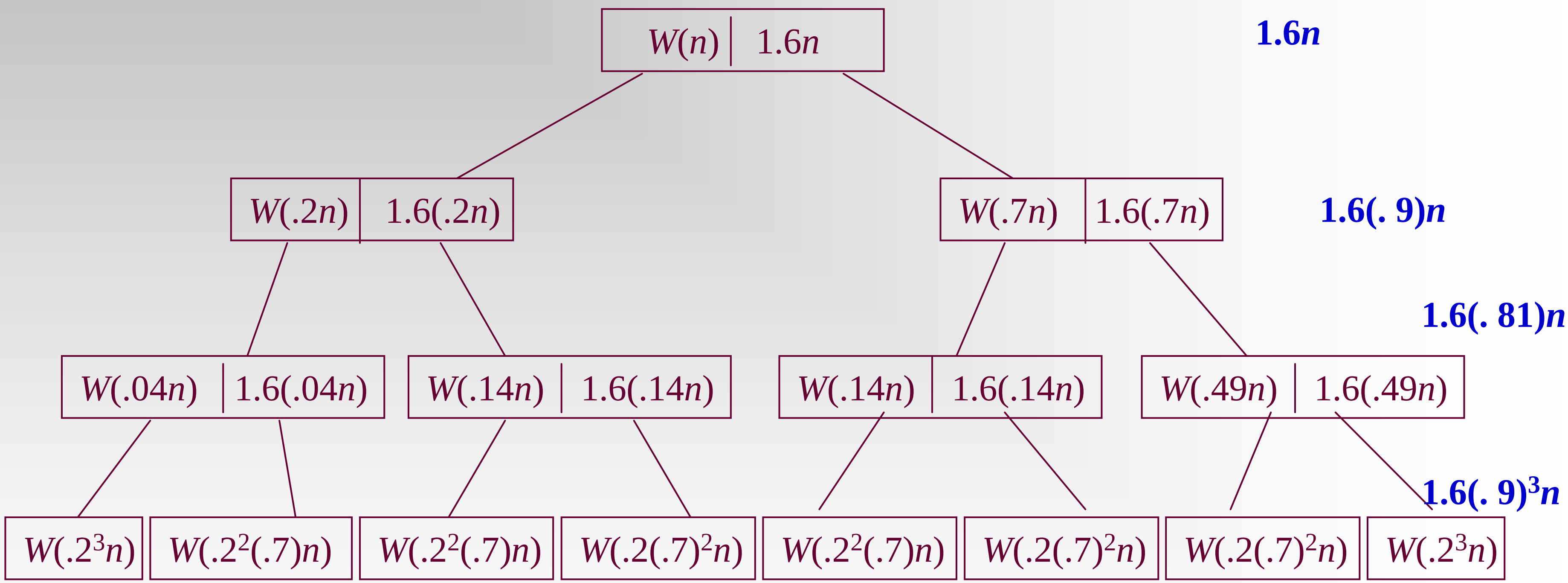
Finding the median of the medians

Comparing all the elements in A∪D with m*

- *Note: r is about n/10, and 0.7n+2 is about 0.7n*, so

$$W(n) \le 1.6n + W(0.2n) + W(0.7n)$$

Worst Case Complexity of *Select*

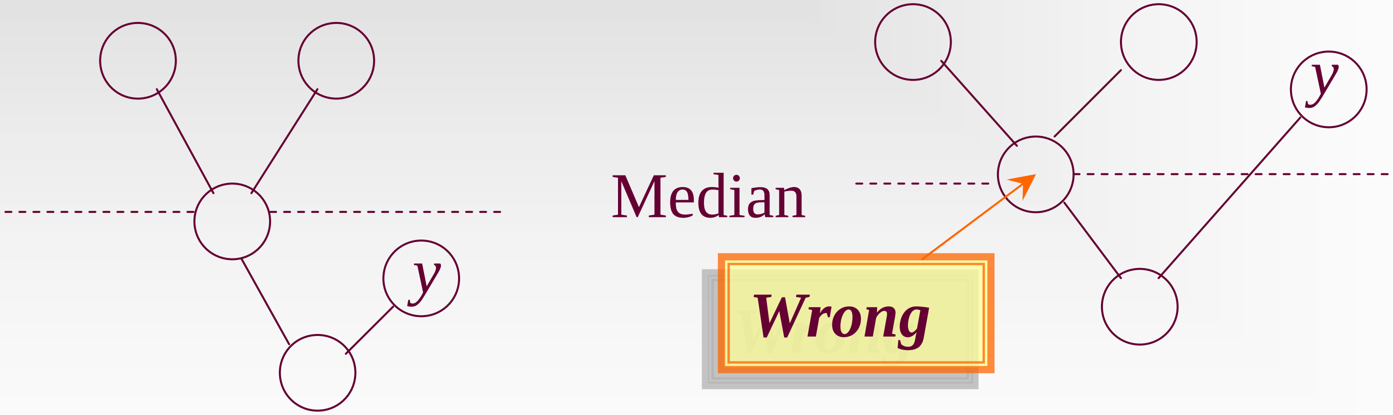


Note: Row sums is a decreasing geometric series, so

$$\boldsymbol{W(n) \in \Theta(n)}$$

Relation to Median

- Observation: Any algorithm of selection must know the relation of every element to the *median*.



Crucial Comparison

- A crucial comparison establishes the relation of some x to the median.
- Definition (for a comparison involving a key x)
  - Crucial comparison for x: the first comparison where $x>y$, for some $y \geq$ median, or $x<y$ for some $y \leq$ median
  - Non-crucial comparison: the comparison between x and y where $x>$ median and $y<$ median

Adversary for Lower Bound

- Status of the key during the running of the Algorithm:
  - *L*: Has been assigned a value ***larger*** than median
  - *S*: Has been assigned a value ***smaller*** than median
  - *N*: Has not yet been in a comparison
- Adversary rule:

Comparands	Adversary's action
N,N	one L, the another S
L,N or N,L	change N to S
S,N or N,S	change N to L

(In all other cases, just keep consistency)

Notes on the Adversary Arguments

- All actions explicitly specified above make the comparisons un-crucial.
- At least, $(n\text{-}1)/2$ L or S can be assigned freely.
- If there are already $(n\text{-}1)/2$ S, a value larger than median must be assigned to the new key, and if there are already $(n\text{-}1)/2$ L, a value smaller than median must be assigned to the new key. The last assigned value is the median.
- So, an adversary can force the algorithm to do (n-1)/2 un-crucial comparisons at least(In the case that the algorithm start out by doing $(n\text{-}1)/2$ comparisons involving two N.

Lower Bound for Selection Problem

- Theorem: Any algorithm to find the median of n keys(for odd n) by comparison of keys must do at least 3n/2-3/2 comparisons in the worst case.
- Argument:
  - There must be done n-1 crucial comparisons at least.
  - An adversary can force the algorithm to perform as many as (n-1)/2 uncrucial comparisons. (Note: the algorithm can always start out by doing (n-1)/2 comparisons involving 2 *N*-keys, so, only (n-1)/2 *L* or *S* left for the adversary to assign freely as the adversary rule.

Home Assignment

- 5.2
- 5.4
- 5.6
- 5.8
- 5.12-14
- 5.17