EECS 3101

Prof. Andy Mirzaian

Sorting & Selection

STUDY MATERIAL:

- [CLRS] chapters 6, 7, 8, 9
- Lecture Notes 5, 6

TOPICS

- **The Sorting Problem**
  - ➢ **Some general facts**
  - ➢ **QuickSort**
  - ➢ **HeapS … ueues**
  - ➢ **Sorting Lower Bou**
  - ➢ **Special Purpose Sorting Algorithms**
- **The Selection Problem**
- **Lower Bound Techniques**
- **Prune-&-Search**

The Sorting Problem

INPUT: A sequence A= $\langle a_1, a_2, \cdots, a_n \rangle$ of n arbitrary numbers.
OUTPUT: A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \cdots, a_{\pi(n)} \rangle$ of the input sequence, such that $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$.

Two elementary operations:

- Comparison between a pair of items a_i and a_j with =, <, >, or any logical combination thereof.

- Exchange: swap f items a_i and a_j .

Definition: An inversion is a pair of n j⟩ in the input, such that i < j but $a_i > a_j$ is out of order).

I(A) = the total # of inversions in sequence A.

In general: $0 \le I(A) \le n(n-1)/2$.

Example: A = ⟨4, 9, 4, 3, 6, 8, 2, 5⟩. I(A) = 14.
$\langle a_2, a_7 \rangle = \langle 9, 2 \rangle$ is one of the inversions in A.

Some Basic Facts

- Swapping an adjacent pair of items will change the inversion count by +1, 0, or –1.

- Any sorting algorithm that (effectively) exchanges only adjacent pairs of items is doomed to take at least $\Omega(n^2)$ steps in the worst case.

  BubbleSort, Inserti this category.

  MergeSort, QuickSort, HeapSort are

- InsertionSort takes $\Theta(n + I)$ time on every input, where
  - n = # input items, and
  - I = # inversions in the input. Why?

  This makes InsertionSort a suitable choice when the input is almost sorted (low I).

QUICKSORT

[C.A.R. Hoare, 1962]

History:

Hoare lived in Moscow for a period of time; first as pa[illegible]al Navy studying modern Russian military; then as a visiting student at Moscow [illegible] and later on, worked for the National Physical Lab stationed in Moscow. He collaborated with the Automatic Machine Translation of Russian to English Project Group. Dictionaries were on a long magnetic tape in alphabetical order. So they would first sort the words in a sentence, then in one pass would compare it with the magnetic tape. …
For the sorting part, he first thought of BubbleSort, but soon realized it was too slow. QuickSort was the 2nd algorithm he says he thought of.

Sorting Student Homework Papers

The way I sort student homework papers by name:

- I first partition them into a small number of piles by initials, e.g.,
  Pile 1: A – F
  Pile 2: G – L
  Pile 3: M – S
  Pile 4: T – Z
- Then I sort each pi tioning them further into more refined grou with the same initial.
- Then I reassemble the sorted piles in

Randomized QuickSort

Algorithm QuickSort(S)

Pre-Cond: input S is a finite sequence of arbitrary numbers

Post-Cond: output is a permutation of S in sorted order

if $|S| < 2$ **then return** S

$p \leftarrow$ a random element in S   § pivot item, why random?

3-Partition S into: § we already discussed this

$S_< \leftarrow$

$S_= \leftarrow$ time

$S_> \leftarrow$

$S'_< \leftarrow$ QuickSort($S_<$)   estion:

$S'_> \leftarrow$ QuickSort($S_>$)   ive call first?!

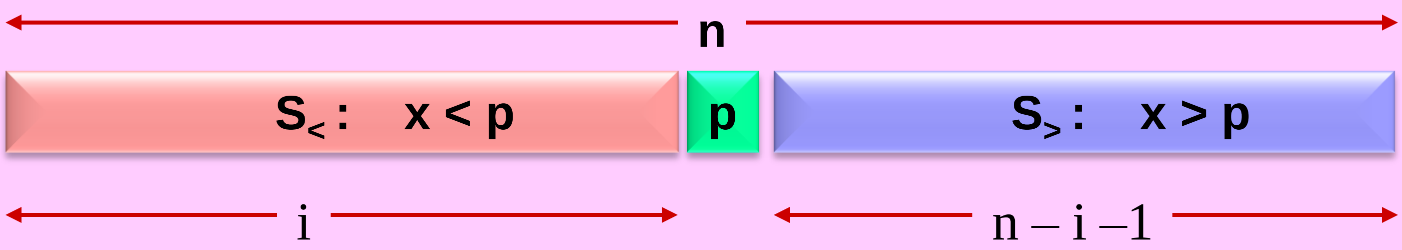
return $\langle S'_<, S_=, S'_> \rangle$

end

$S_<$: $x < p$	$S_=$: $x = p$	$S_>$: $x > p$

$T(|S|) = T(|S_<|) + T(|S_>|) + \Theta(|S|)$, $T(n) = \Theta(1)$, for $n=0,1$.

QuickSort Running Time



WLOG Assume: | $S_=$ | = 1. If it's larger, it can only help!

T(n) = T(i) +), for n=0,1.

Worst-Case:

$$T(n) = \max_i \{ T(i) + T(n-i-1) \quad = 0 .. n-1 \}$$

$$= T(n-1) + T(0) + \Theta(n)$$

$$= T(n-1) + \Theta(n)$$

$$= \Theta(n^2)$$

This occurs if at all recursion levels the selected pivot is (near) the extreme;
the largest or the smallest!

QuickSort Running Time

n

$S_<$: x < p | p | $S_>$: x > p

i | n – i –1

WLOG Assume: | $S_=$ | = 1. If it's larger, it can only help!

T(n) = T(i) +), for n=0,1.

Best-Case:

$T(n) = \min_i \{ T(i) + T(n-i-1)$ $= 0 .. n-1 \}$

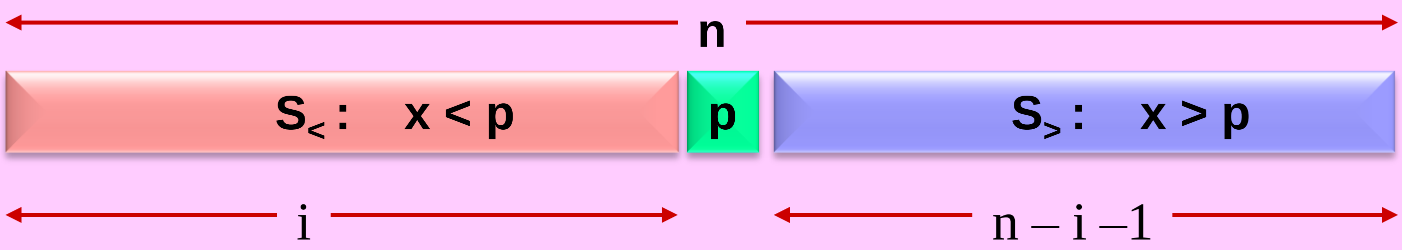
$= T(n/2) + T(n/2) + \Theta(n)$

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

$= \Theta(n \log n)$

This occurs if at all recursion levels the selected pivot is (near) the median element!

QuickSort Running Time



WLOG Assume: $|S_=| = 1$. If it's larger, it can only help!

$T(n) = T(i) +$), for n=0,1.

Expected-Case:

$$T(n) = \text{ave}_i \{ T(i) + T(n\text{-}i\text{-}1) \quad = 0 \,..\, n-1 \}$$

$$= \frac{1}{n}\sum_{i=0}^{n-1}[T(i) + T($$

$$= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + \Theta(n)$$ (full history recurrence already discussed)

$$= \Theta(n \log n)$$

Full History Recurrence: QuickSort

Example 2: $T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n, \quad \forall n \geq 0 \qquad [\Rightarrow \ T(0)=0]$

1. Multiply across by n (so that we can subsequently cancel out the summation):

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2, \quad \forall n \geq 0$$

2. Substitute n-1 for n: $(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2, \quad \forall n \geq 1$

3. Subtract (2) from (1): $nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n-1, \quad \forall n \geq 1$

4. Rearrange: $nT(n) = (n+1)T(n-1) + 2n-1, \quad \forall n \geq 1$

5. Divide by n(n+1) to make LHS & RHS look alike: $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)}, \quad \forall n \geq 1$

6. Rename: $Q(n) = \frac{T(n)}{n+1}, \quad Q(n-1) = \frac{T(n-1)}{n}, \quad \frac{2n\text{-}1}{n(n+1)} = \frac{3}{n+1} - \frac{1}{n}$

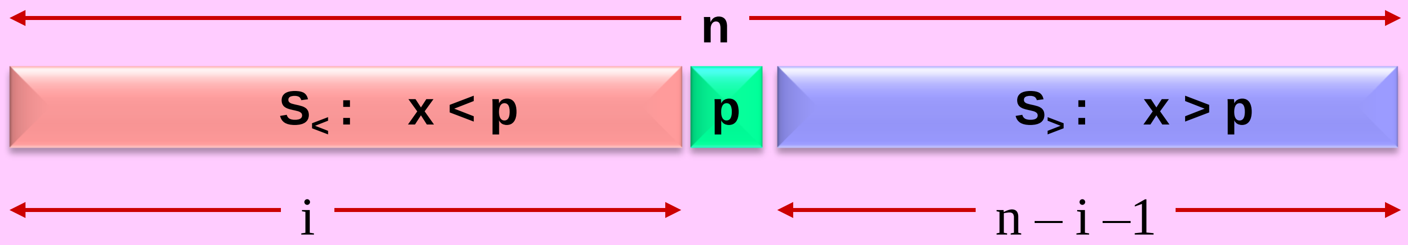
7. Simplified recurrence: $Q(n) = \begin{cases} Q(n-1) + \left[\frac{3}{n+1} - \frac{1}{n}\right] & \forall n \geq 1 \\ 0 & \text{for } n = \end{cases}$

n^th^ Harmonic number

8. Iteration: $Q(n) = \left(\frac{3}{n+1} - \frac{1}{n}\right) + \left(\frac{3}{n} - \frac{1}{n-1}\right) + \left(\frac{3}{n-1} - \frac{1}{n-2}\right) + \cdots + \left(\frac{3}{2} - -\right.$ $(n) - \frac{3n}{n+1}$

9. Finally: $T(n) = (n+1)Q(n) = 2(n+1)H(n) - 3n = \Theta(n \log n).$

Why Randomize?



Worst-Case: $T(n) = \Theta(n^2)$

Expe g n)

Best- g n)

- Randomization nullifies adverse effect ged input permutation.
- Algorithm sees the input as a random permutation.
- Probability that almost all random choices are the worst is nearly 1/n! (extremely low).
- **FACT:** Randomized QuickSort will take O(n log n) time with probability **very close** to 1 on **every** input.

HEAPSORT,

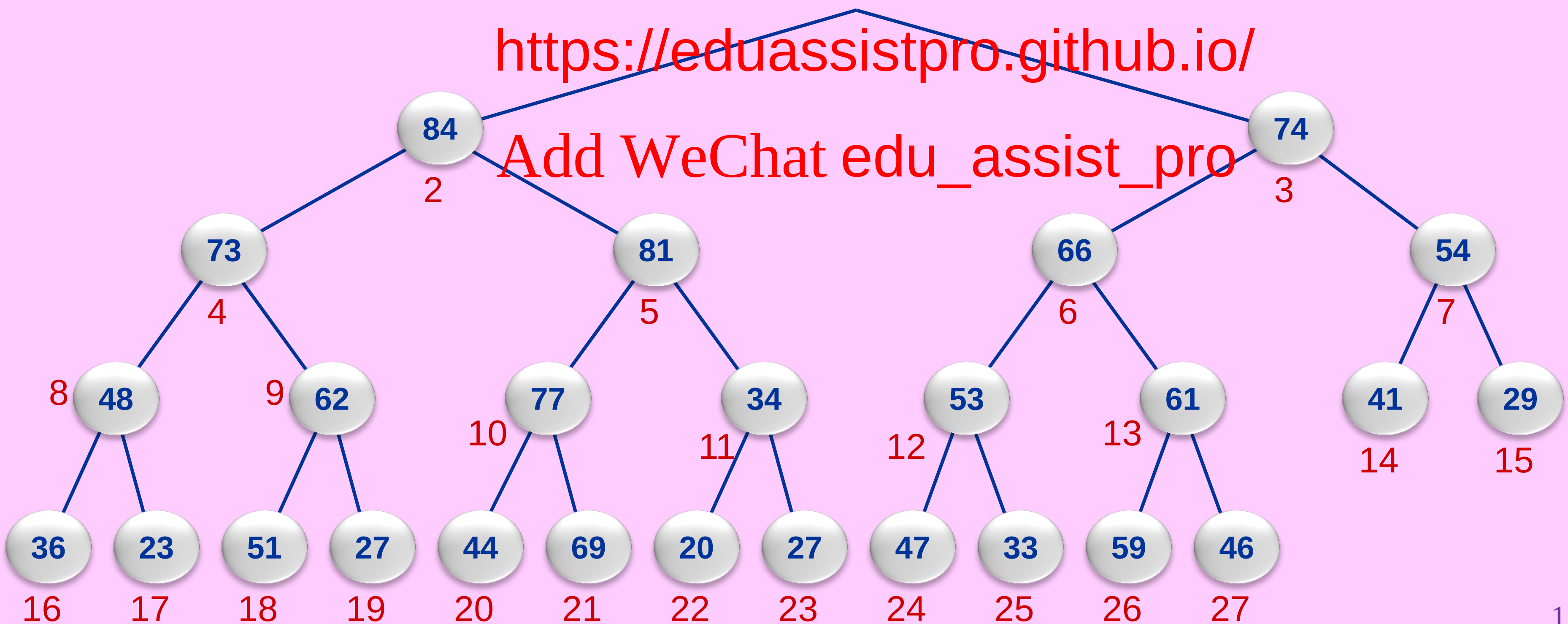
Pri ues

[J.W.J. Williams, 1964]

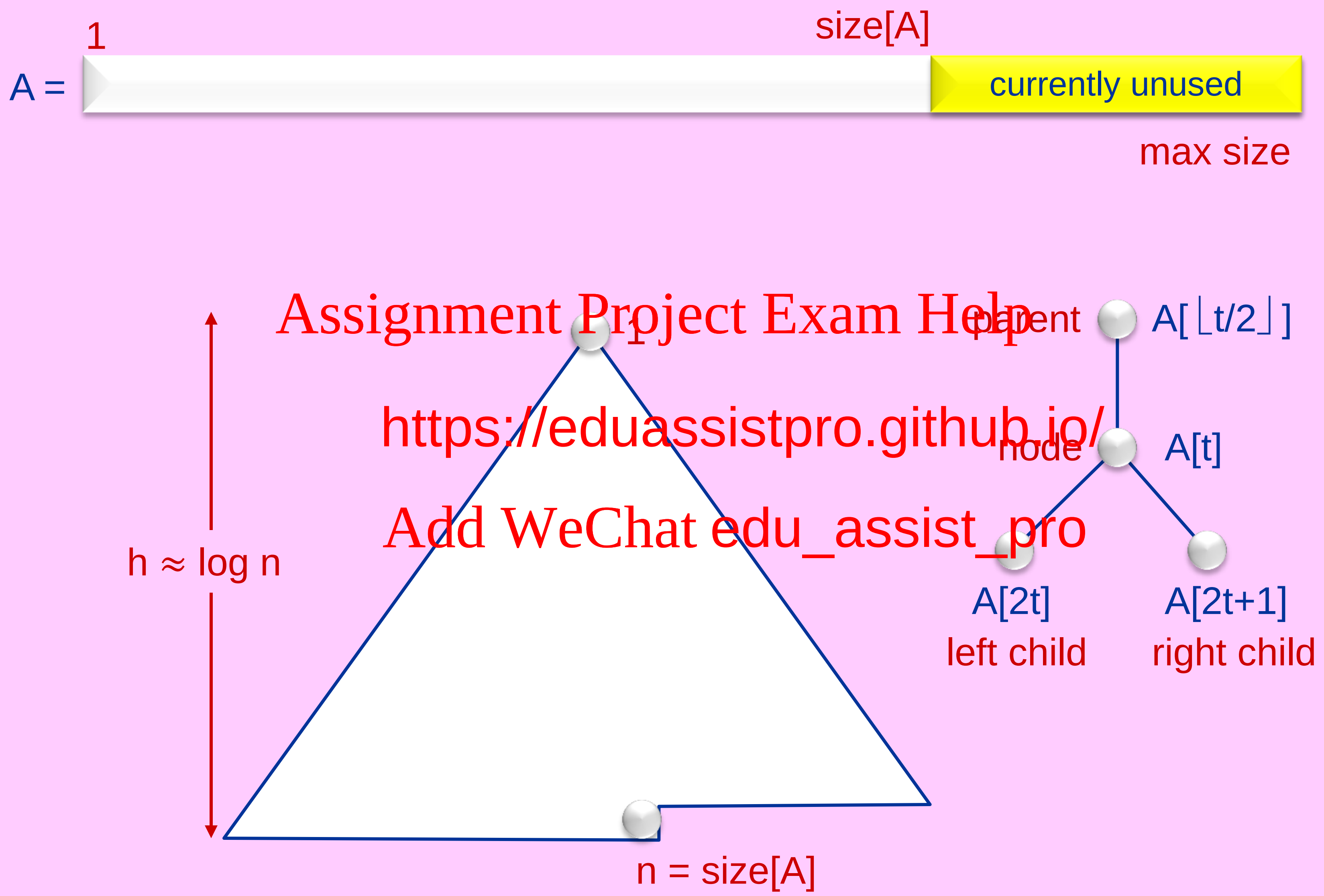
Binary Heap

- A = a binary tree with one key per node (duplicate keys allowed).
- Max Heap Order: A satisfies the following partial order:

  for every node $x \neq root[A]$: $key[x] \leq key[parent(x)]$.
- Full tree node allocation scheme: nodes of A are allocated in increasing order of level, and left-to-right within the same level.
- This allows array implementation, where array indices simulate tree pointers.

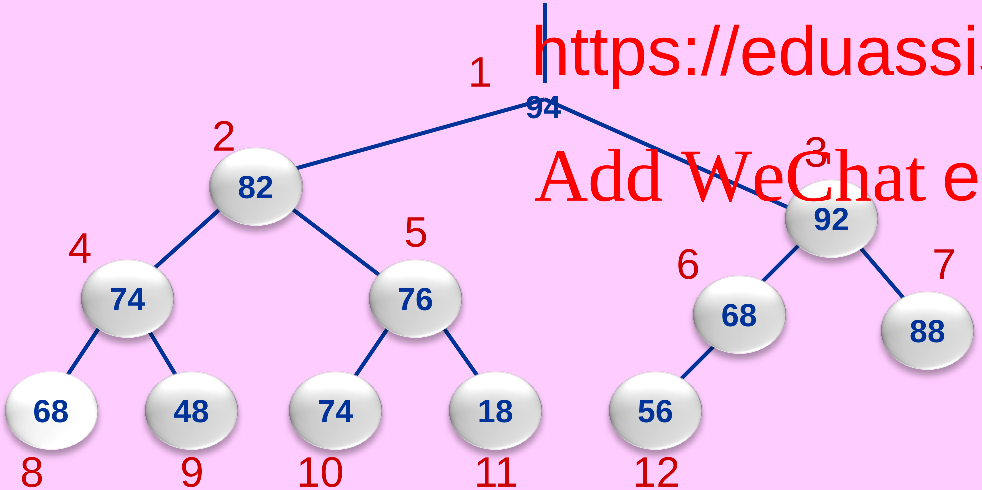


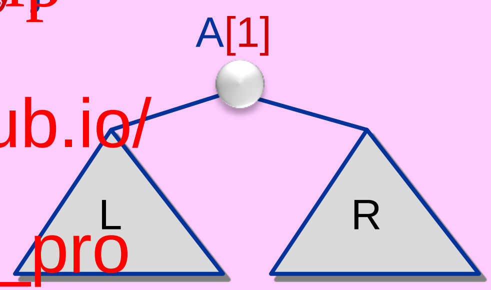
Array as Binary Heap

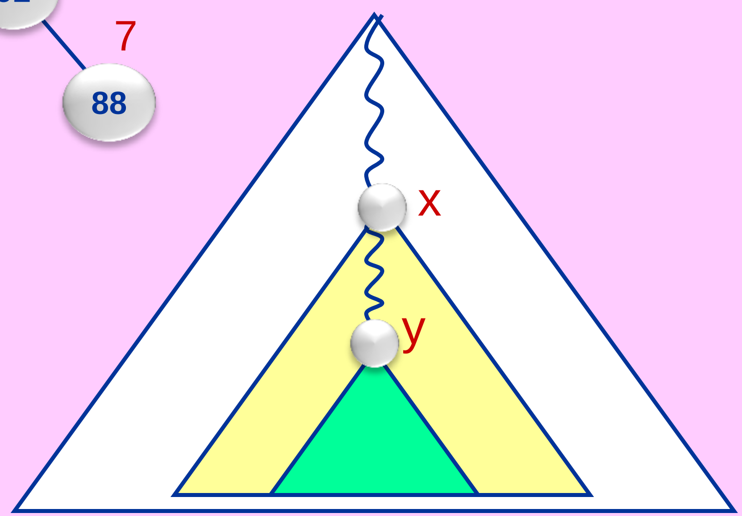


Some MAX Heap Properties

- Root A[1] contains the maximum item.
- Every root-to-leaf path appears in non-increasing order.
- Every subtree is also max heap ordered. [Recursive structure]
- The key at any node is the largest among all its descendants (inclusive).
- $\forall(x,y) \in$ AncestorOf : A[x] $\geq$ A[y], where AncestorOf = { (x,y) : node x is ancestor of node y }.







UpHeap

- A[1..n] = a max-heap.
- Suddenly, item A[t] increases in value.
- Now A is a "t upward corrupted heap":

  $\forall (x,y) \in \text{AncestorOf} : \quad y \neq t \Rightarrow A[x] \geq A[y].$

- Question: how would you rearrange A to make it a max-heap again?
- Answer: percolate A[t] up its ancestral path.

procedure UpHeap

Pre-Cond: A is a t upward corrupted heap

Post-Cond: A is rearranged into a max-heap

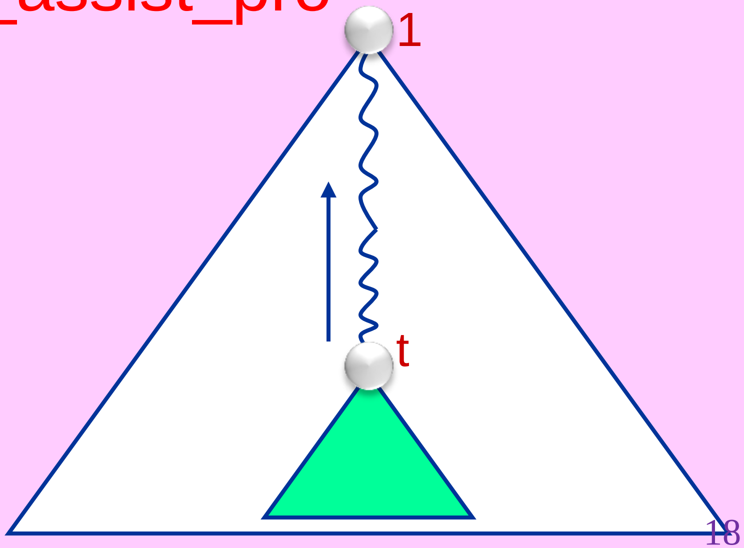
$p \leftarrow \lfloor t/2 \rfloor$ § parent of t

if $p = 0$ or $A[p] \geq A[t]$ **then return**

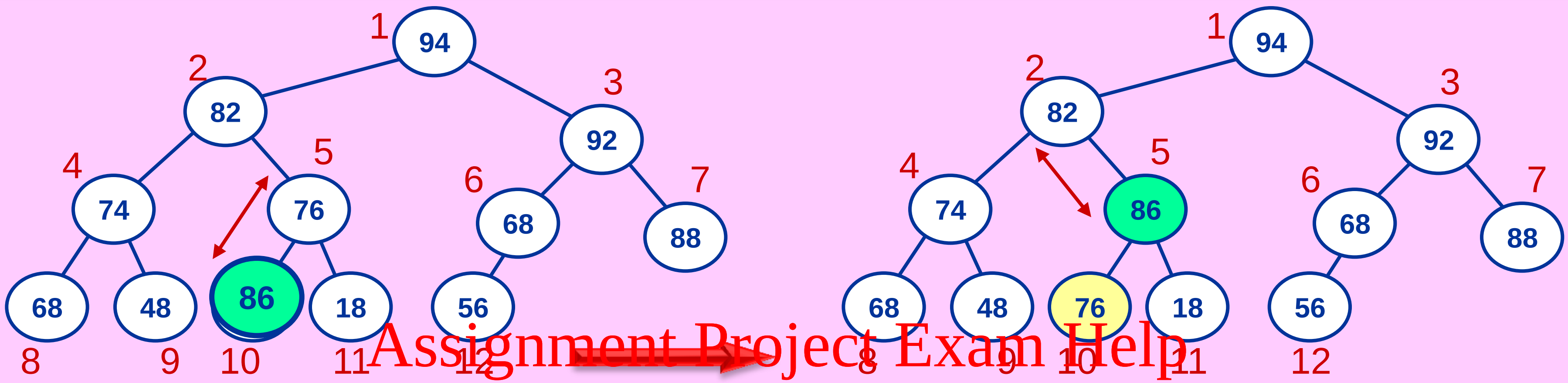
$A[t] \leftrightarrow A[p]$

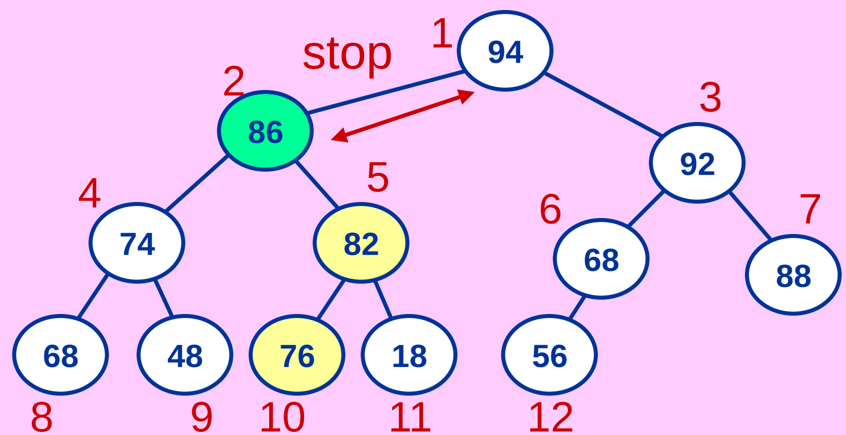
UpHeap(A,p)

end



UpHeap Example





DownHeap

- A[1..n] = a max-heap.
- Suddenly, item A[t] decreases in value.
- Now A is a "t downward corrupted heap":

$$\forall (x,y) \in \text{AncestorOf} : \quad x \neq t \Rightarrow A[x] \geq A[y].$$

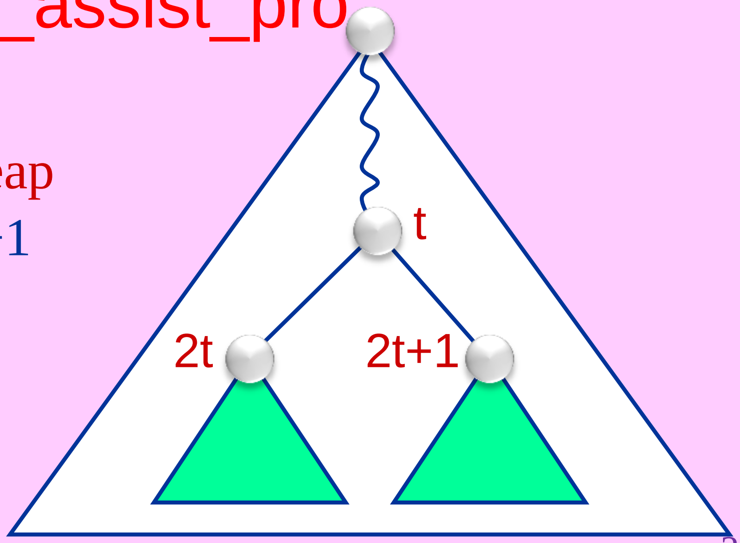
- Question: how would you rearrange A to make it a max-heap again?
- Answer: demote A[t] down along largest-child path.

procedure DownH

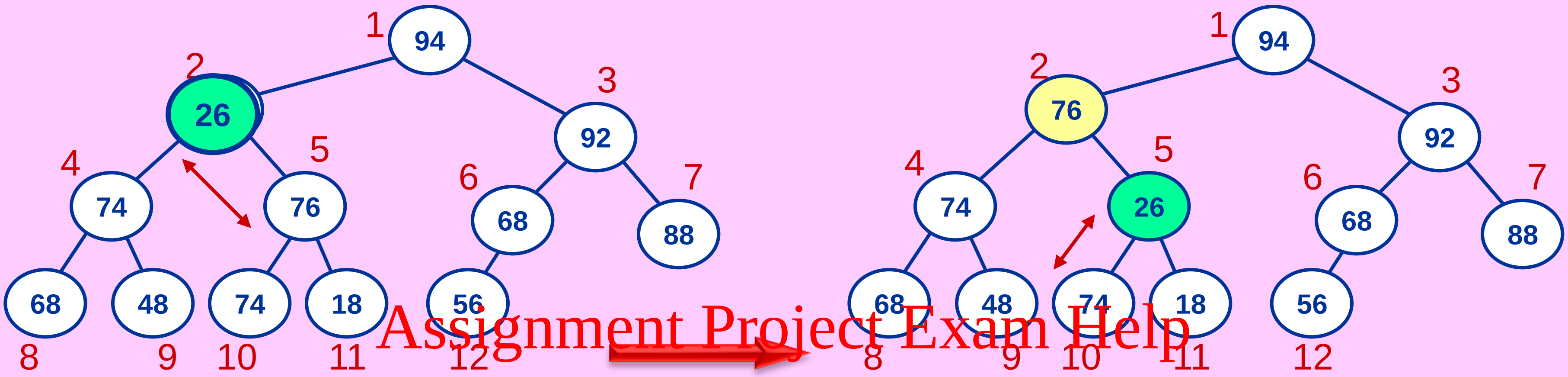
Pre-Cond: A is a t downward corrupted heap

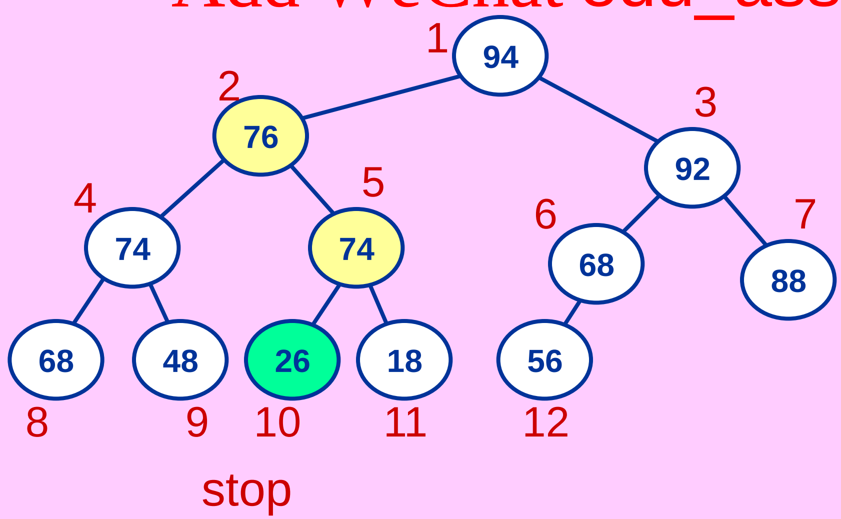
Post-Cond: A is rearranged into a max-heap

```
    c ← 2t          § left child of t
    if c > size[A] then return      § c not part of heap
    if c < size[A] and A[c] < A[c+1] then c ← c+1
    § now c is the largest child of t
    if A[t] < A[c] then
        A[t] ↔ A[c]
        DownHeap(A, c)
end
```



DownHeap Example





Construct Heap (or Heapify)

- One application of heaps is sorting. But how do we start a heap first?
- **Problem:** Given array A[1..n], rearrange its items to form a heap.
- **Solution 1: Sort** A[1..n] in descending order. Yes, but that's circular!
- **Solution 2: Build Incrementally:**

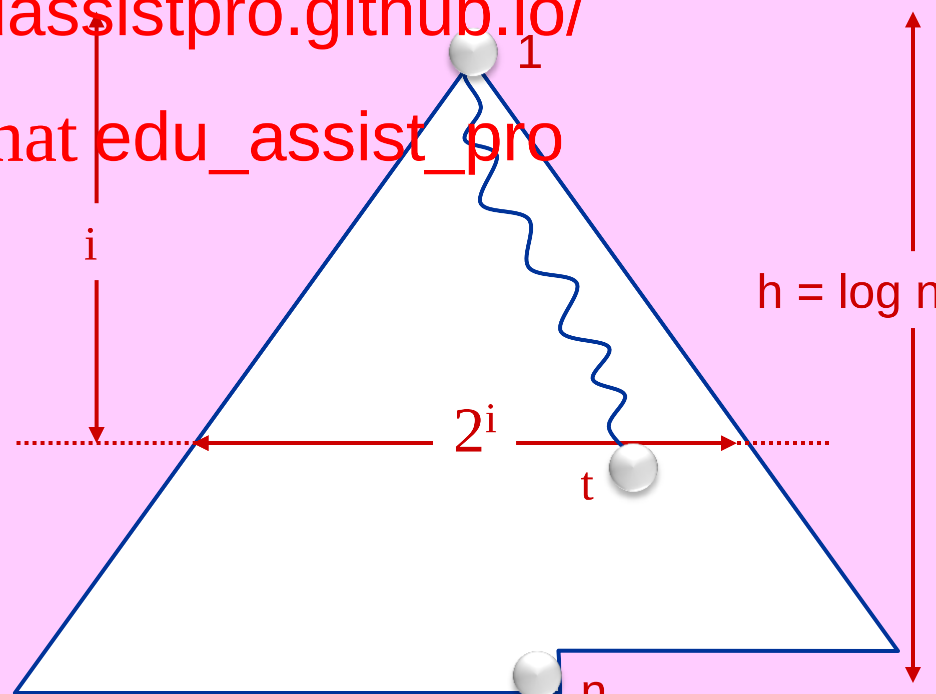
  That is, make A[1..t] a max heap while incrementing $t \leftarrow 1 .. n$.

  That is, **for** $t \leftarrow 1 .. n$ **do** UpHeap(A, t) **end**

$$\text{Time} = \Theta\left(\sum_{i=0}^{h} i\,2^i\right)$$

$$= \Theta(h\,2^h)$$

$$= \Theta(n \log n)$$



Most nodes are concentrated near the bottom with larger depths but smaller heights. **Idea: DownHeap is better!**

Heap Construction Algorithm

Solution 3: Build Backwards on t by DownHeap(A,t).

Assume items in nodes 1..t-1 are tentatively $+\infty$ so that DownHeap's Pre-Cond is met.

procedure ConstructHeap(A[1..n]) § O(n) time

Pre-Cond: input is array A[1..n] of arbitrary numbers

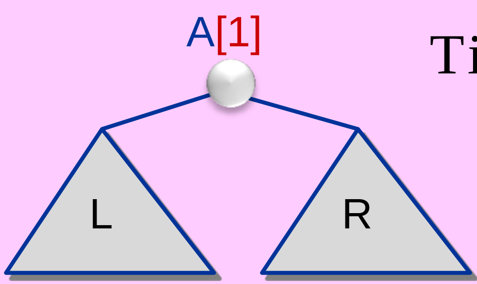
Post-Cond: A is rearranged into a max-heap

size[A] ← n § establish last node barrier

LastNonleafN

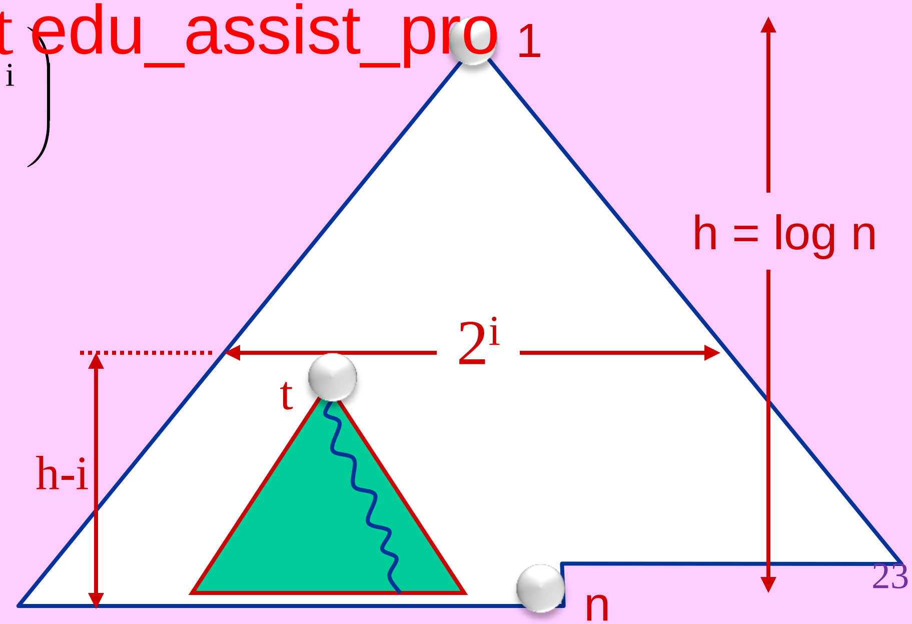
for t ← LastN wnHeap(A, t)

end

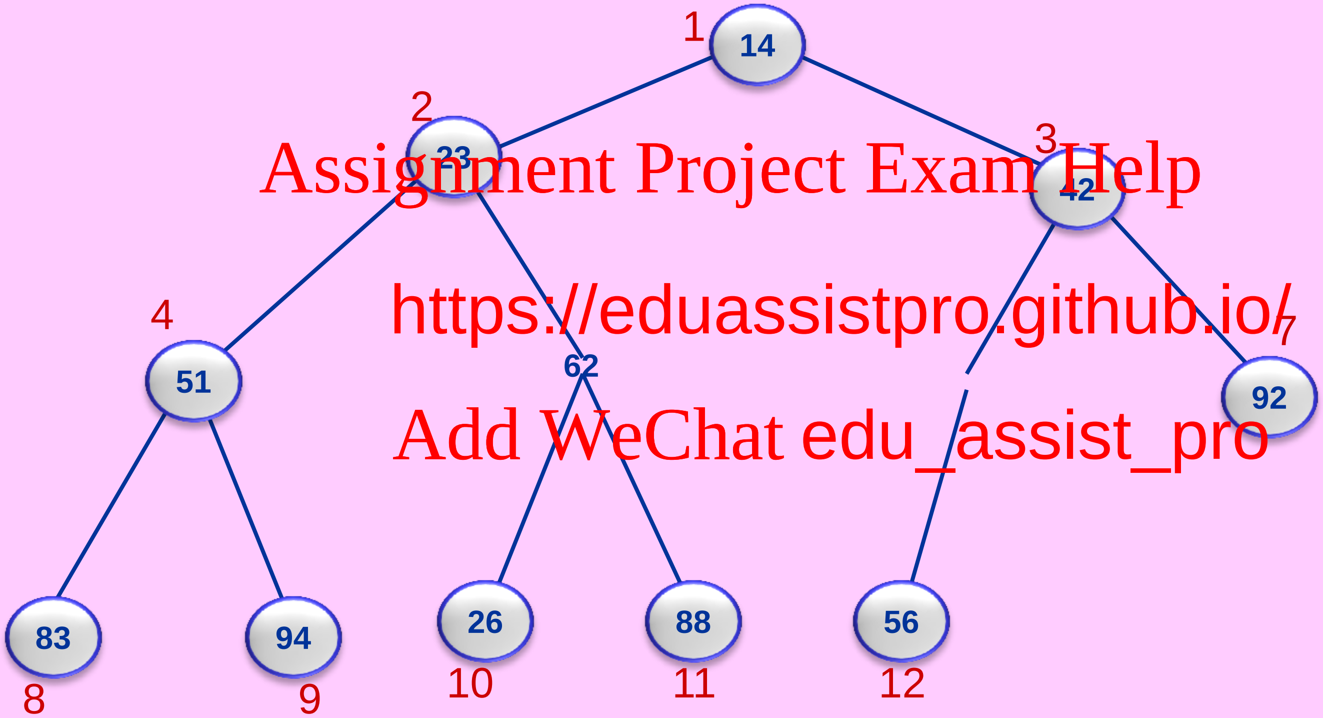


T(n) =
T(|L|) + T(|R|) + O(log n)
$\Rightarrow$ T(n) = O(n)

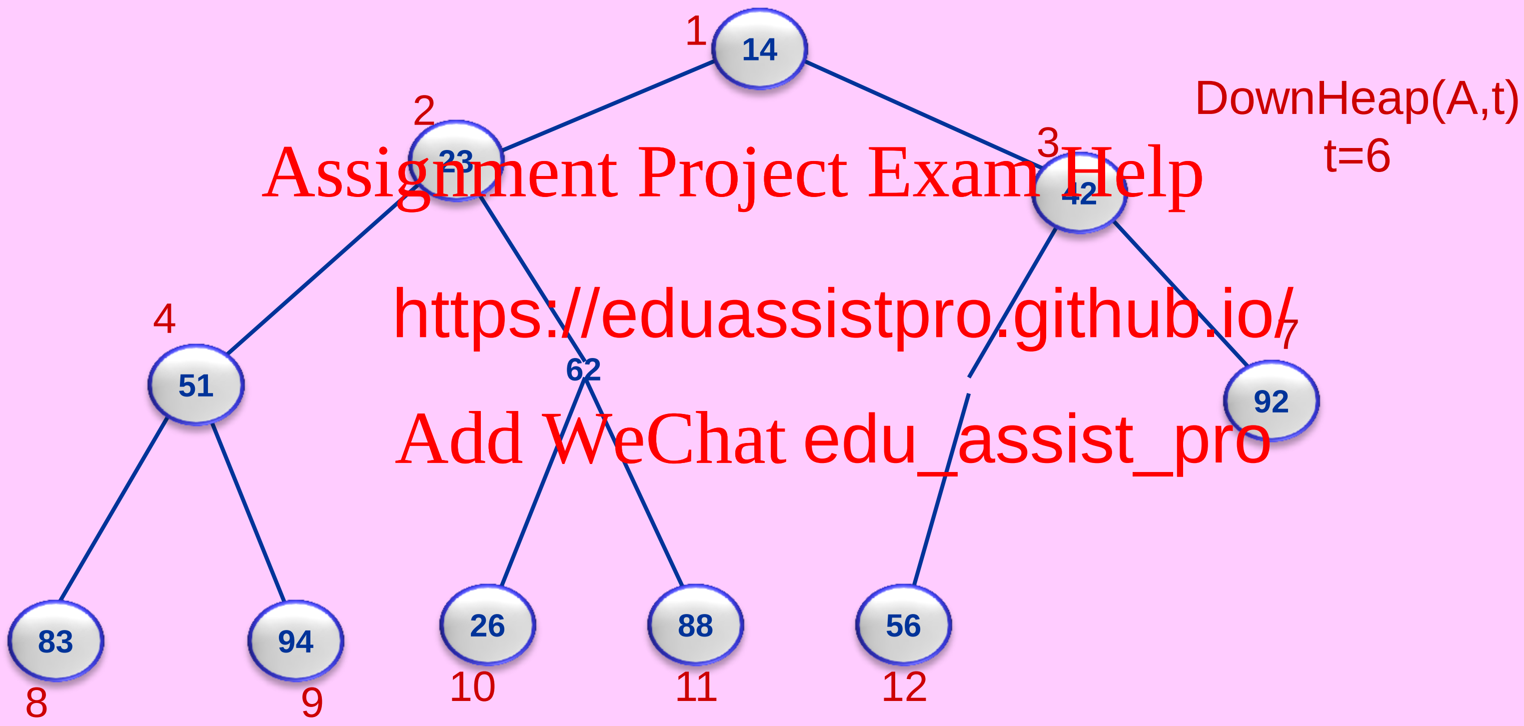
$$\text{Time} = \Theta\left(\sum_{i=0}^{h}(h-i)2^{i}\right)$$
$$= \Theta\left(\sum_{j=0}^{h} j2^{h-j}\right)$$
$$= 2^{h}\,\Theta\left(\sum_{j=0}^{h} j2^{-j}\right)$$
$$= 2^{h}\,\Theta(1)$$
$$= \Theta(n)$$



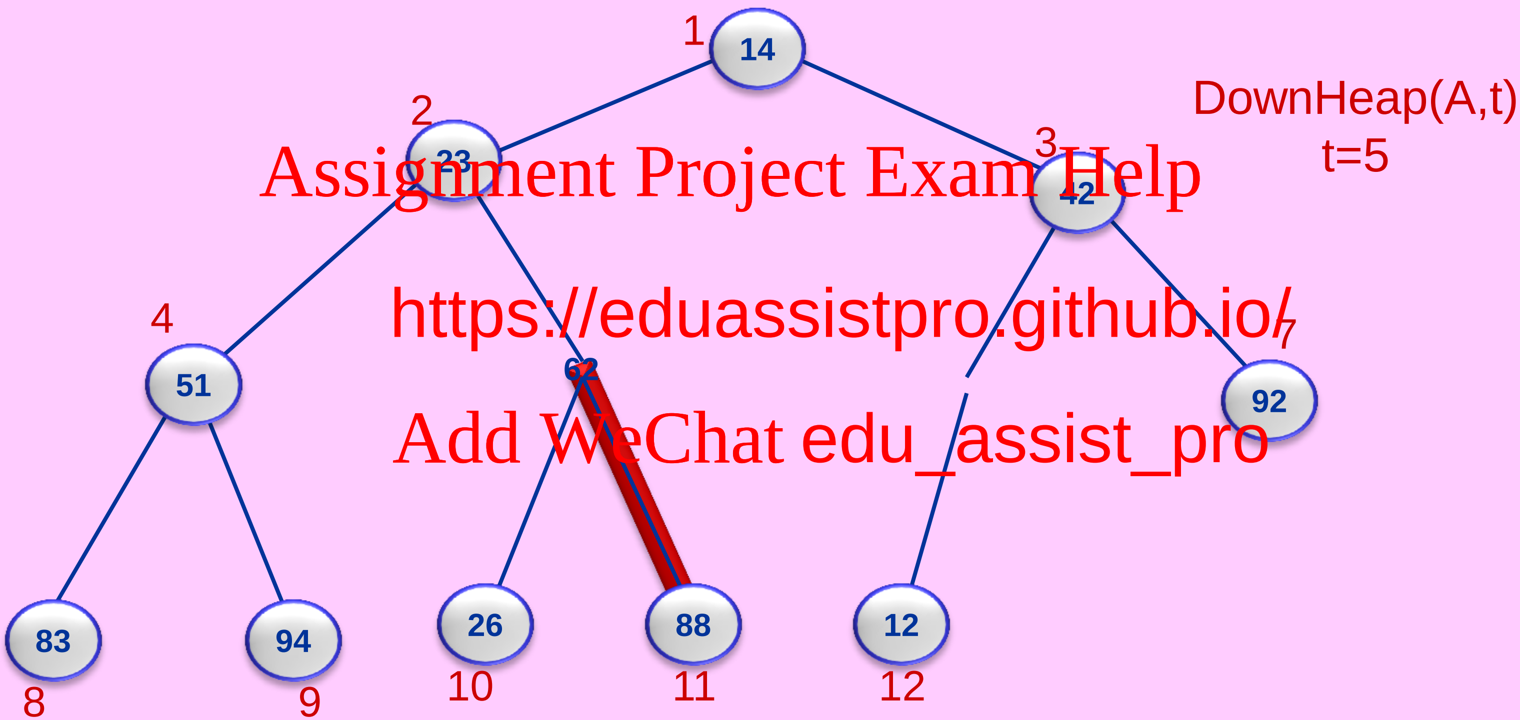
Construct Heap Example



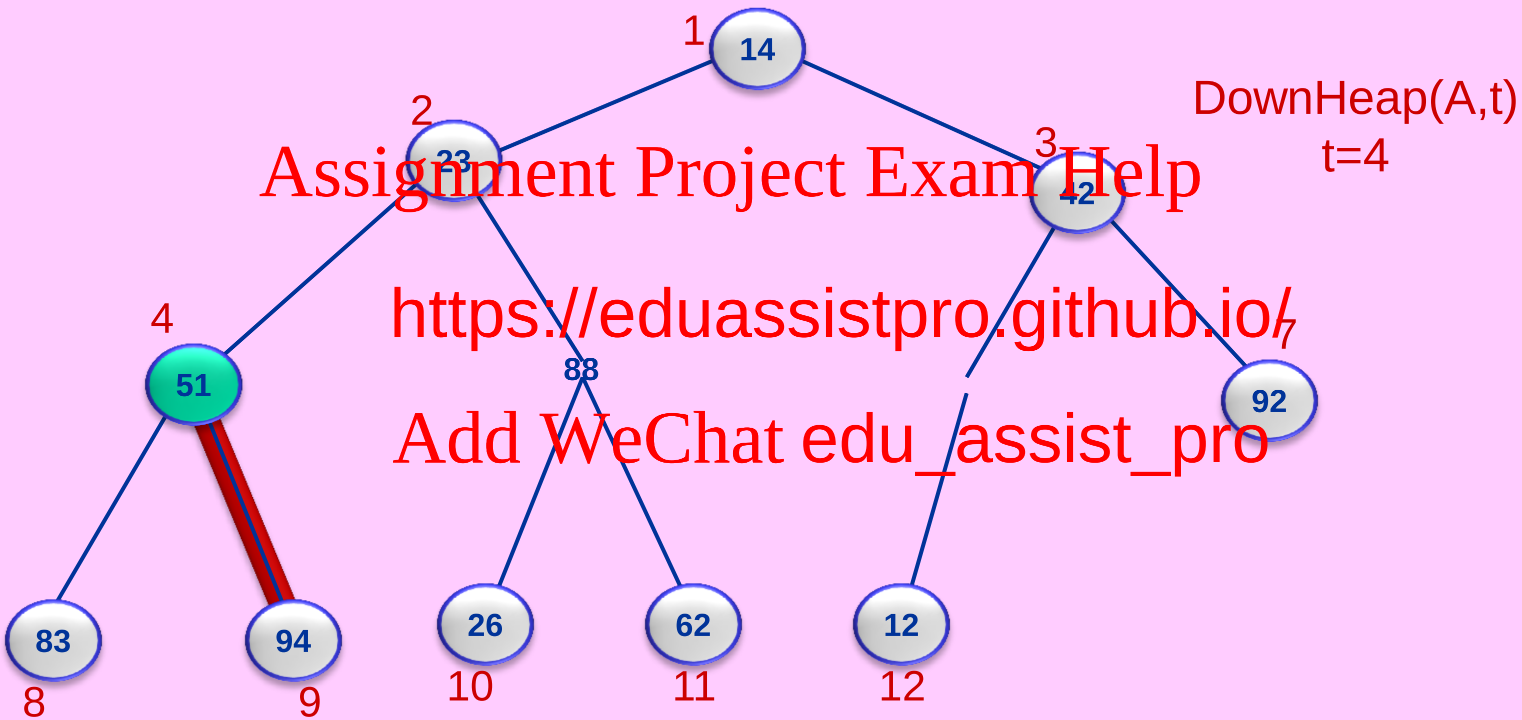
Construct Heap Example



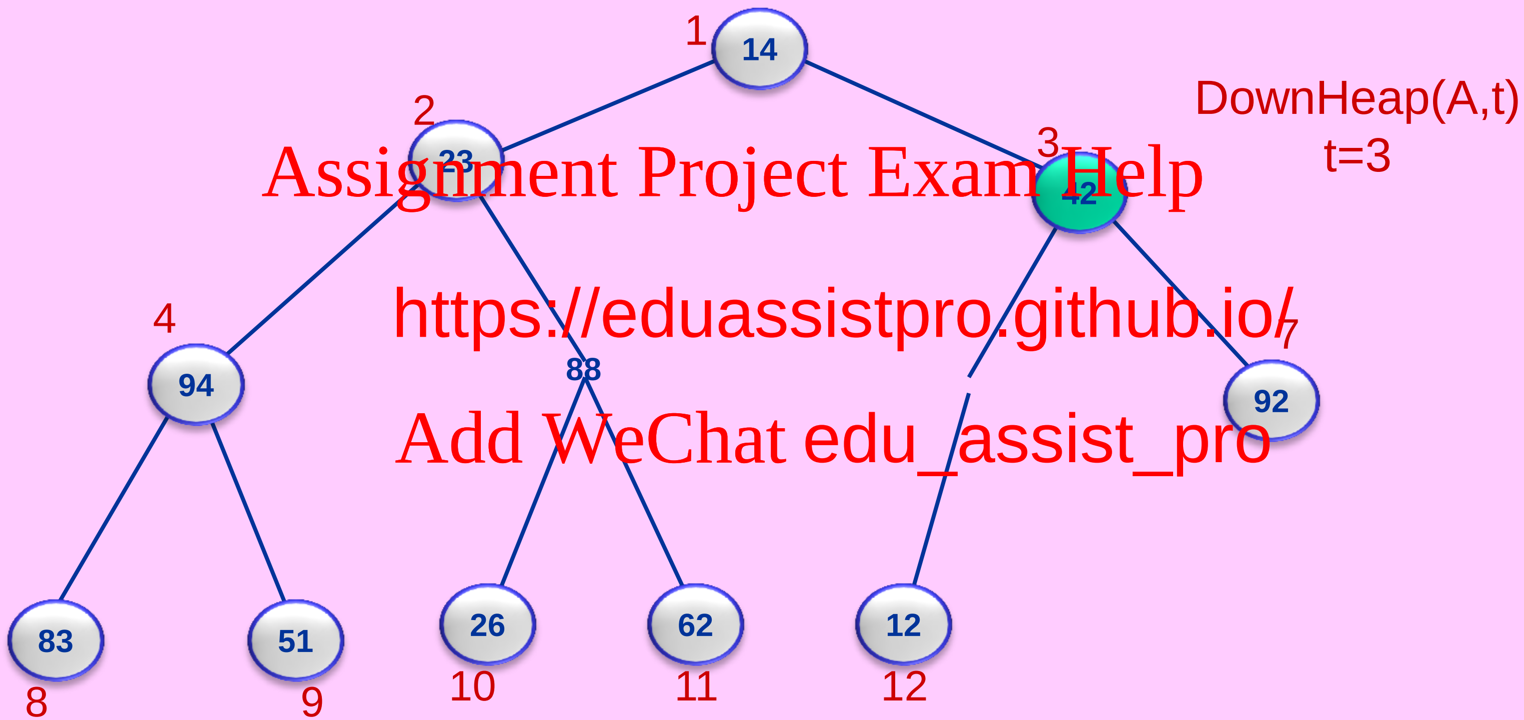
Construct Heap Example



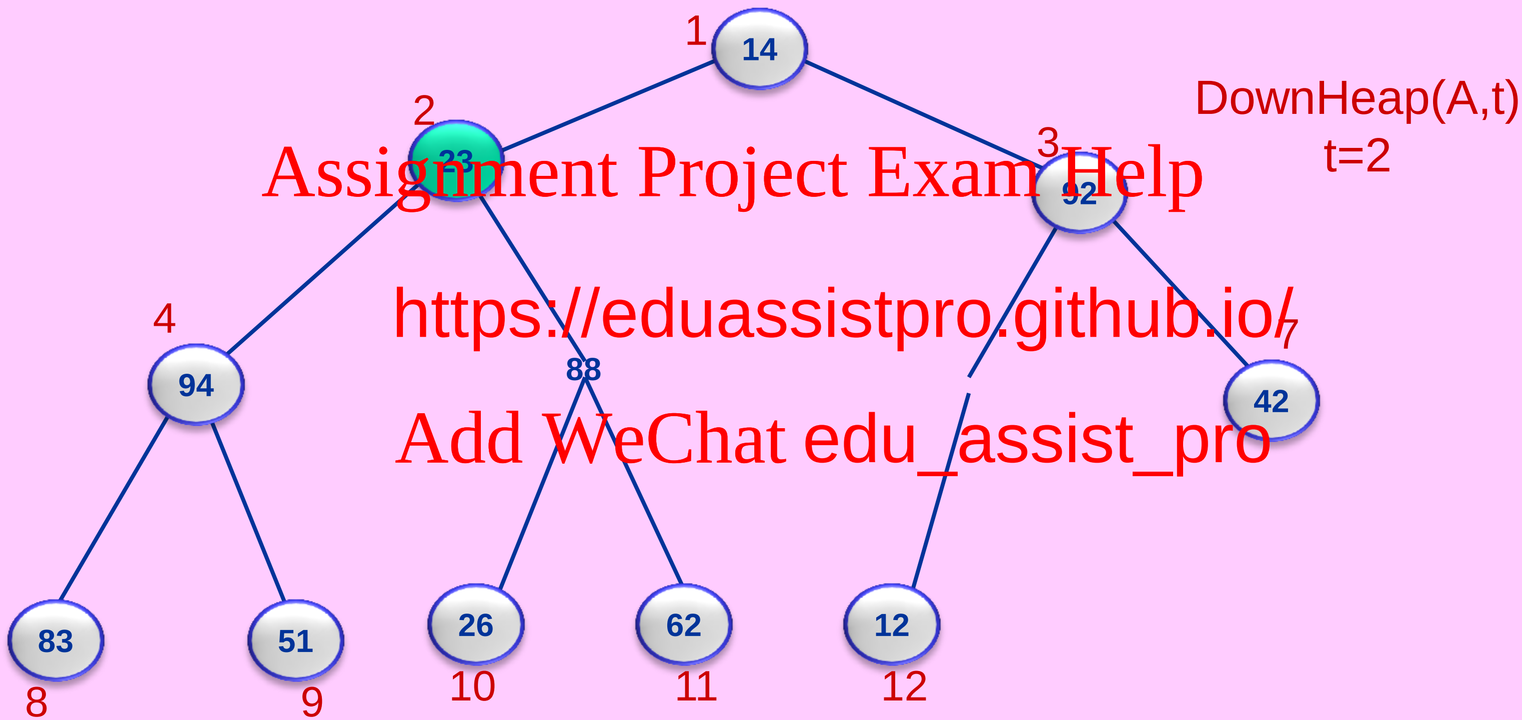
Construct Heap Example



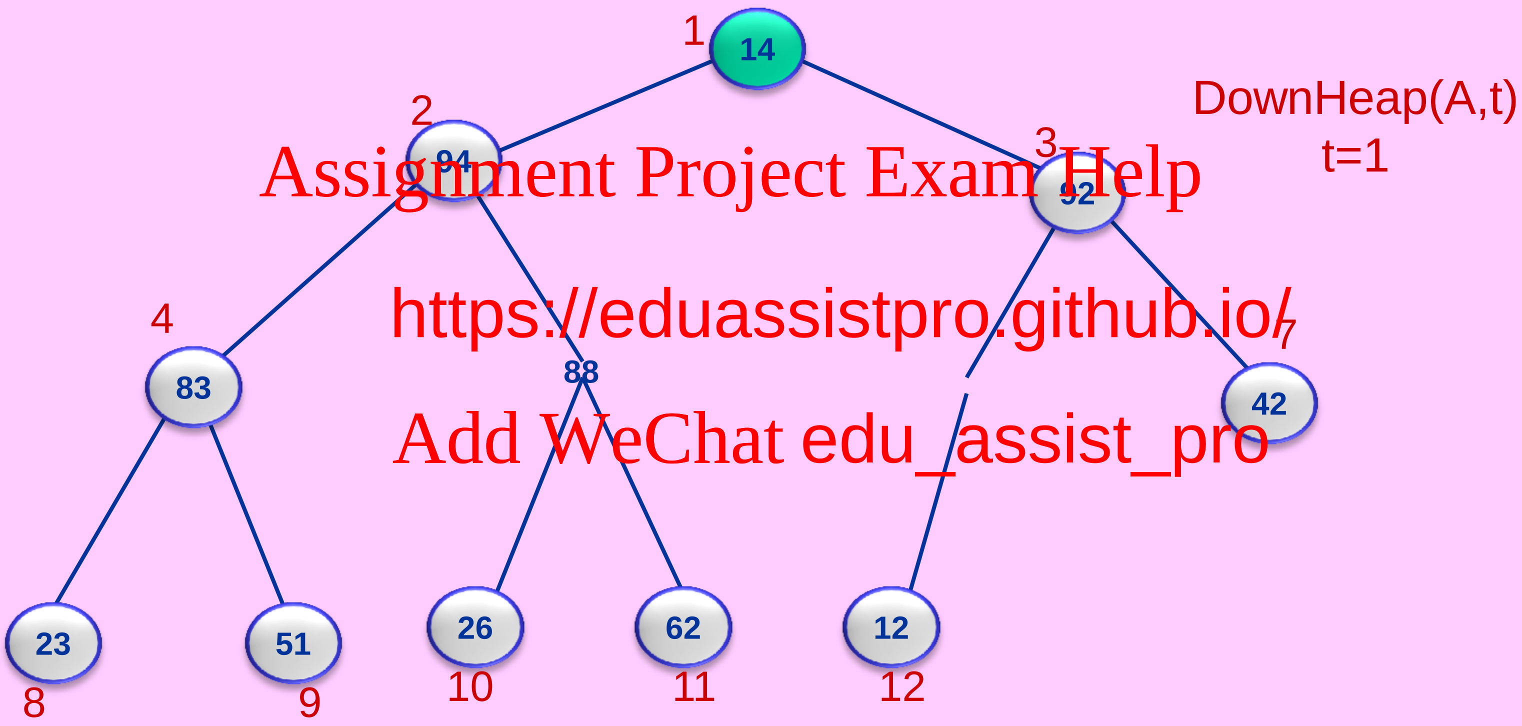
Construct Heap Example



Construct Heap Example

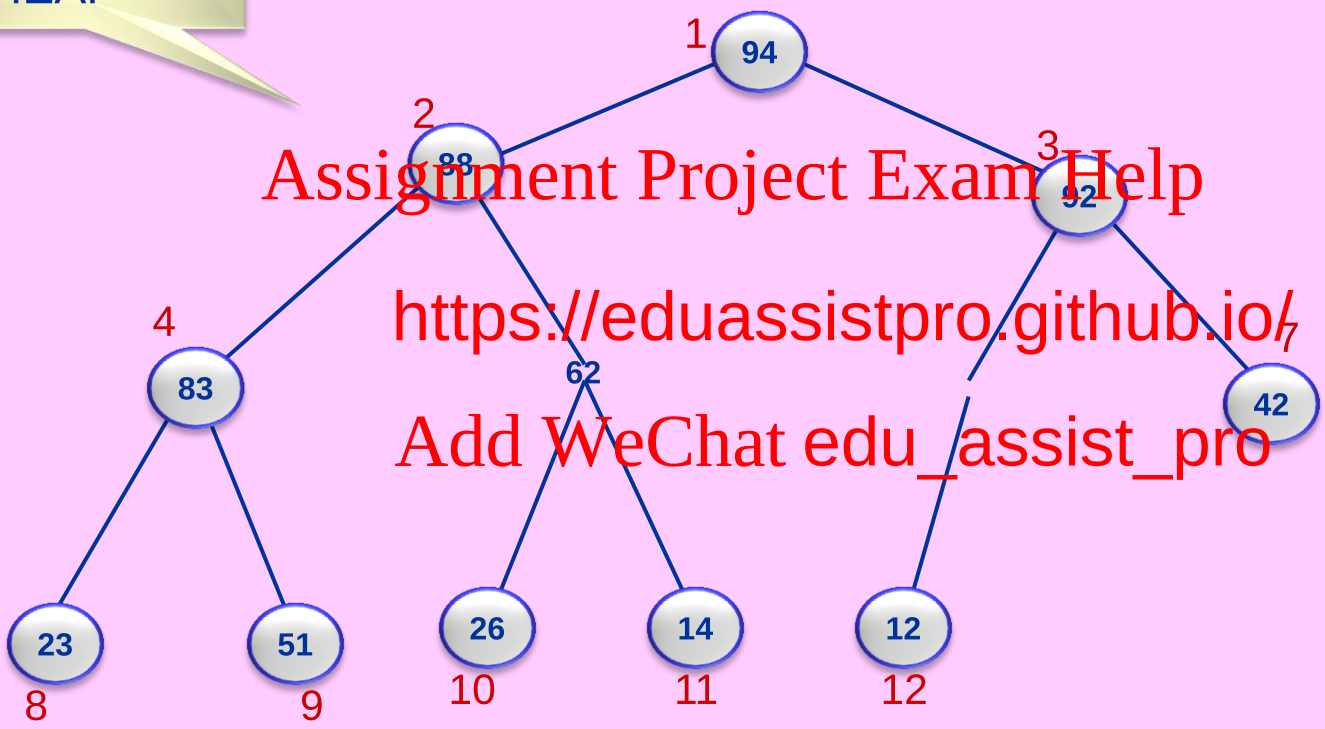


Construct Heap Example



Construct Heap Example

MAX HEAP



Heap as a Priority Queue

A **Priority Queue** (usually implemented with some “heap” structure) is an abstract Data Structure that maintains a set S of items and supports the following operations on it:

MakeEmptyHeap(S): Make an empty priory queue and call it S.

ConstructHeap(S): Construct a priority queue containing the set S of items.

Insert(x, S): Ins ... values allowed)

DeleteMax(S): Remove and return th ... m from S.

Note: Min-Heap is used if we intend to do DeleteMin instead of DeleteMax.

Priority Queue Operations

Array A as a binary heap is a suitable implementation.

For a heap of size n, it has the following time complexities:

O(1)	MakeEmptyHeap(A)	size[A] ← 0
O(n)	ConstructHeap(A[1..n])	discussed already
O(log n)	Insert(x,A) and DeleteMax(A)	see below

procedure Insert(x, A)

size[A] ← size[A] + 1

A[siz

UpHe

end

procedure DeleteMax(A

if size[A] = 0 **then return** error

MaxItem ← A[1]

A[1] ← A[size[A]]

size[A] ← size[A] – 1

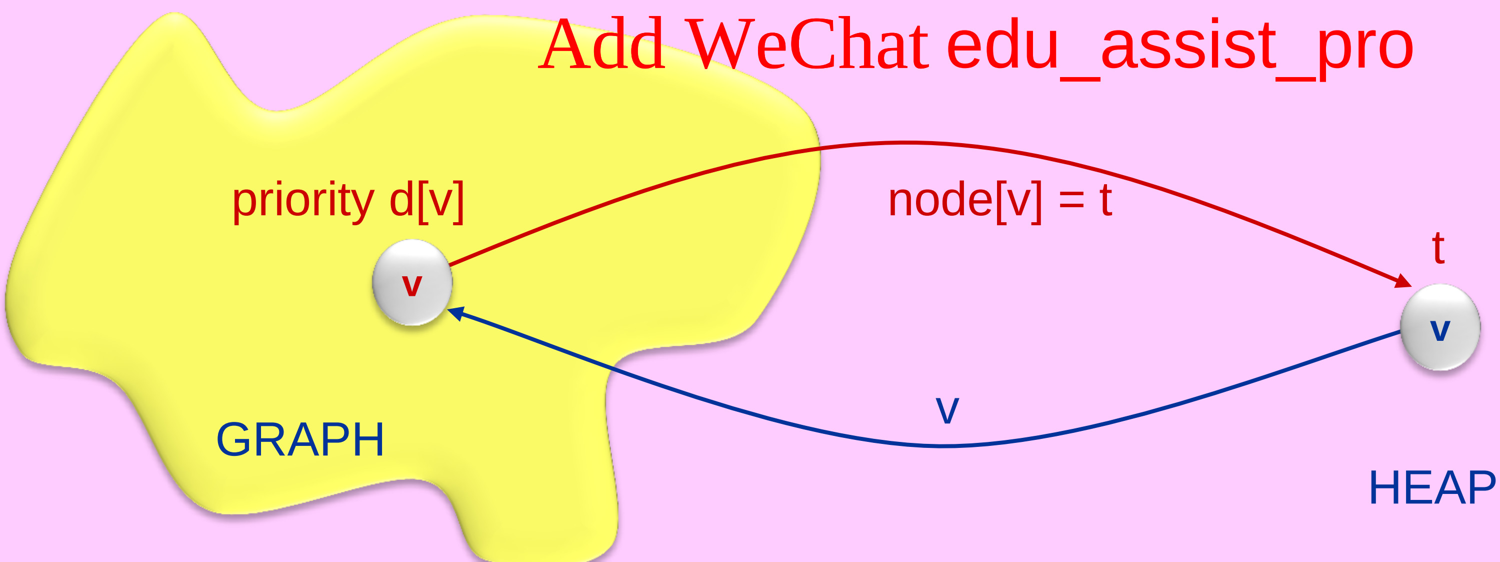
DownHeap(A, 1)

return MaxItem

end

A word on **Priority Queues**

- In many priority queue applications, an item and the priority of the item are two distinct object types.
- For instance: items are vertices in a graph, while priority of an item is the shortest distance from source to the corresponding vertex in the graph.
- We store (pointers to) items in the nodes of the heap, while item priorities are stored separately, external to the heap.
- We also maintain two-way “pointers” (node[v] and v) between items and their heap node location for direct access both ways. (node[v]=0 means v is not in the heap.)
- Heap ordering property ... eir priorities.
- **The result is a priorit ... riority queue.**



HeapSort

Algorithm **HeapSort**(A[1..n]) § O(n log n) time

Pre-Cond: input is array A[1..n] of arbitrary numbers

Post-Cond: A is rearranged into sorted order

```
    ConstructMaxHeap(A[1..n])
    for t ← n downto 2 do
        A[t] ← D
end
```

HeapSort Example (pages 36 – 68)

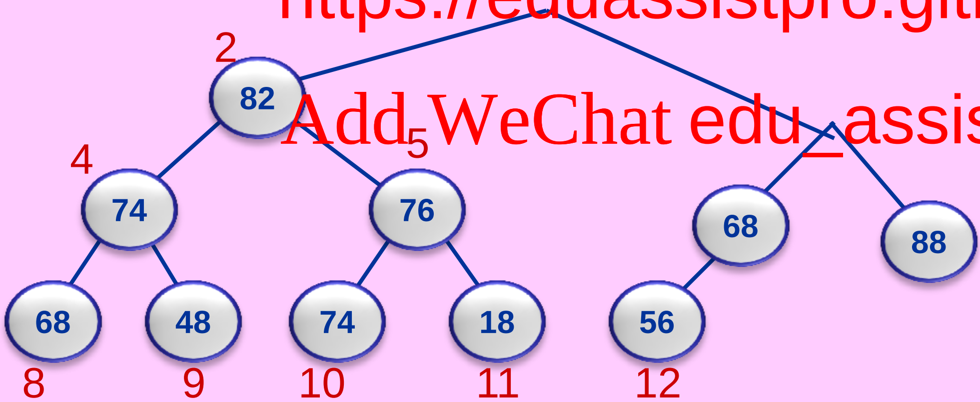
1	2	3	4	5	6	7	8	9	10	11	12
94	82	92	74	76	68	88	68	48	74	18	56

already constructed heap

Max Item

swap

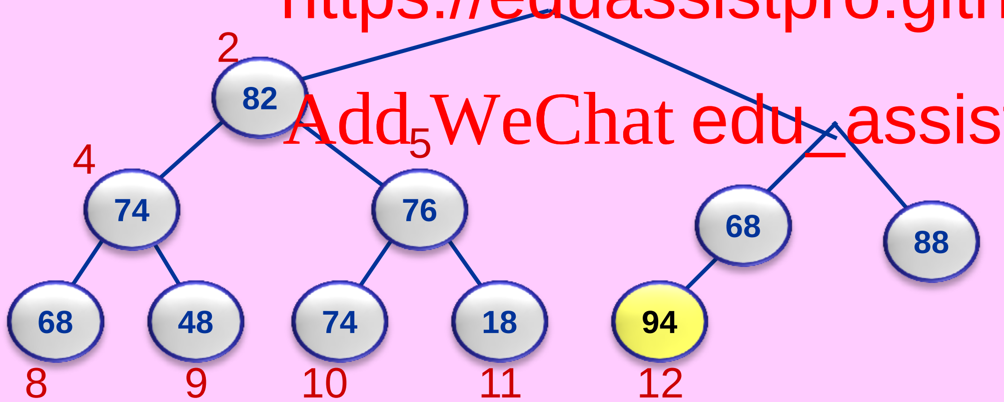
size[A] = 12



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	82	92	74	76	68	88	68	48	74	18	94

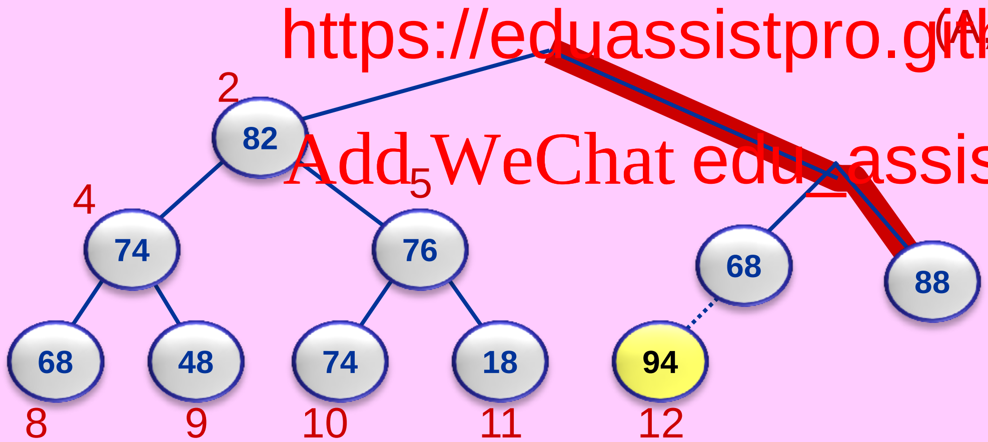
size[A] = 12



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	82	92	74	76	68	88	68	48	74	18	94

size[A] = 11



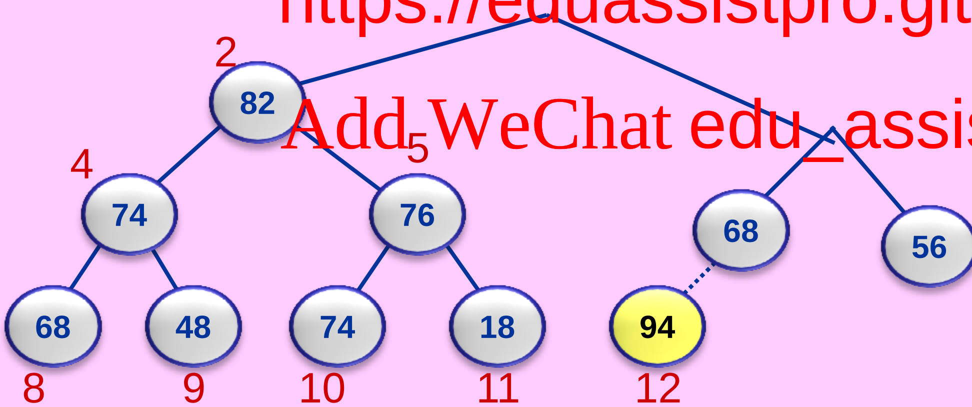
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
92	82	88	74	76	68	56	68	48	74	18	94

Max Item

swap

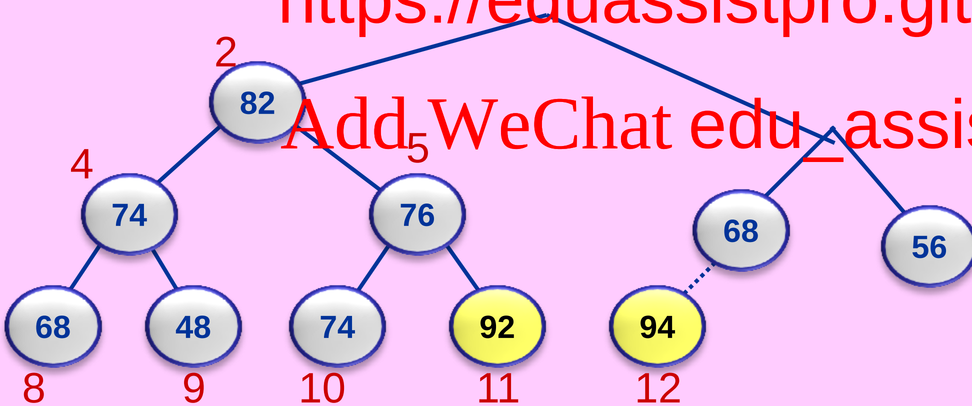
size[A] = 11



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	82	88	74	76	68	56	68	48	74	92	94

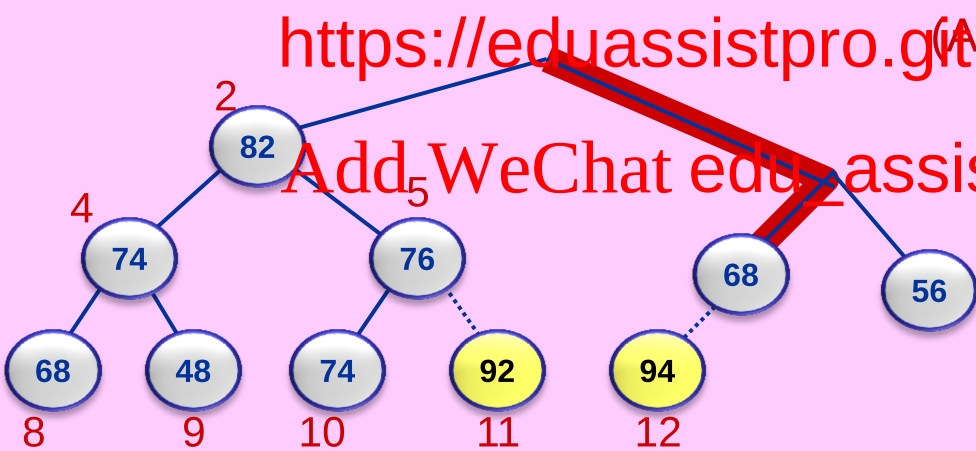
size[A] = 11



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	82	88	74	76	68	56	68	48	74	92	94

size[A] = 10



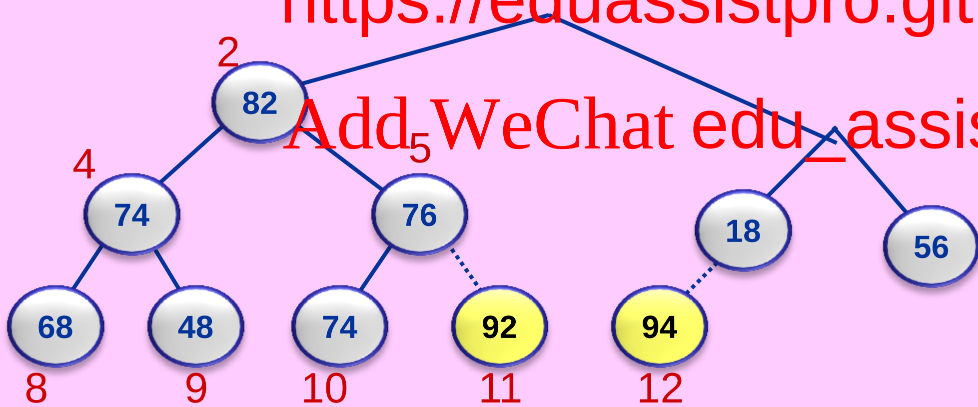
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
88	82	68	74	76	18	56	68	48	74	92	94

Max Item

swap

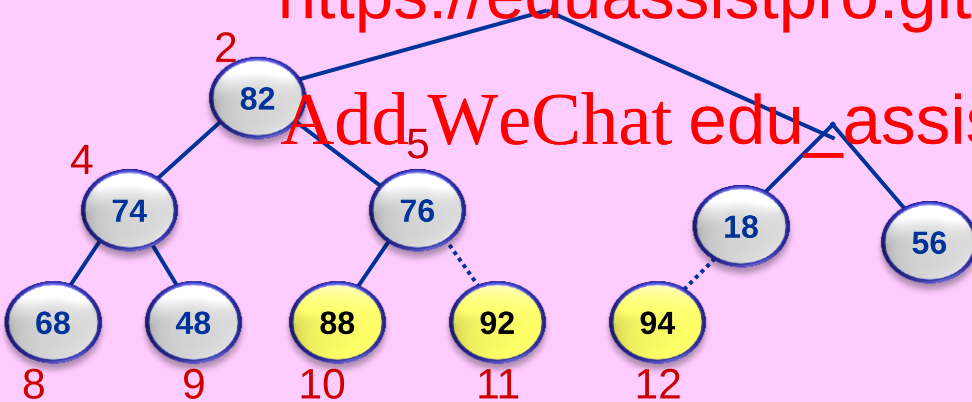
size[A] = 10



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	82	68	74	76	18	56	68	48	88	92	94

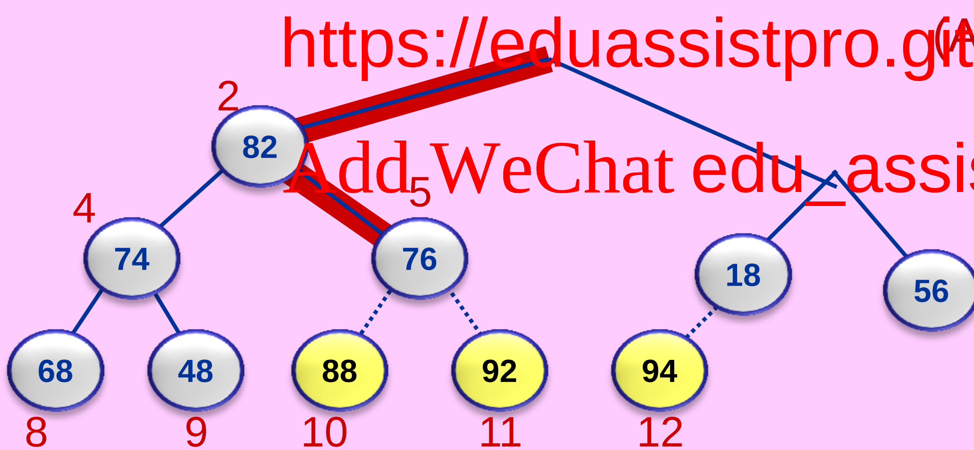
size[A] = 10



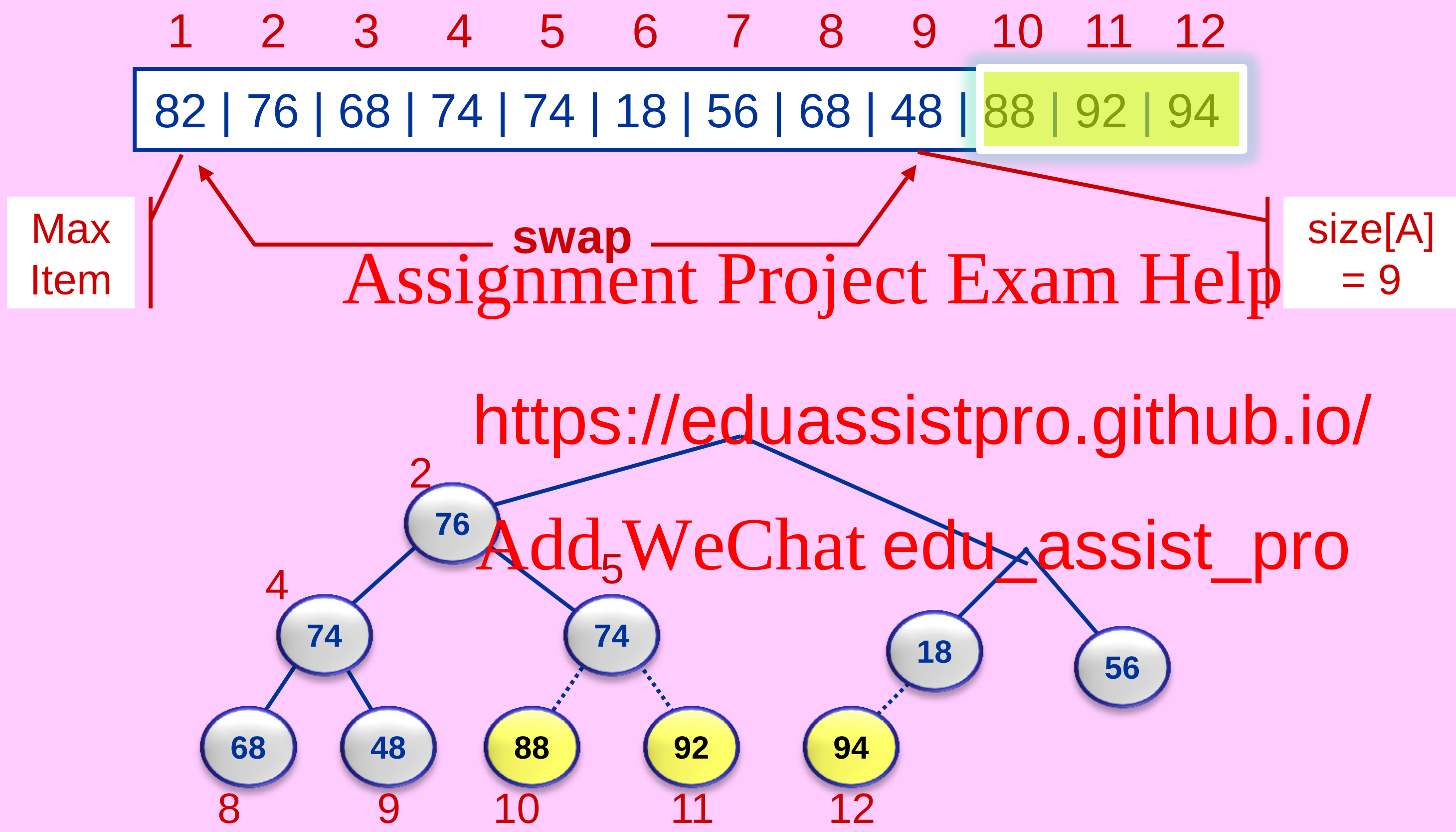
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	82	68	74	76	18	56	68	48	88	92	94

size[A] = 9



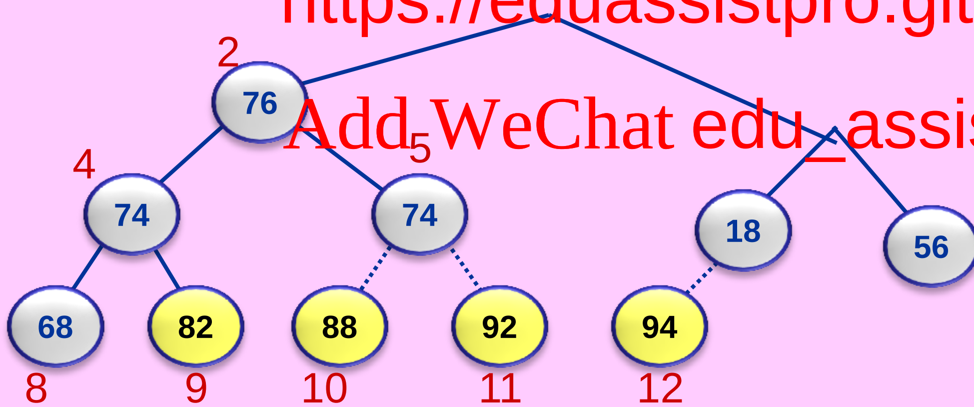
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	76	68	74	74	18	56	68	82	88	92	94

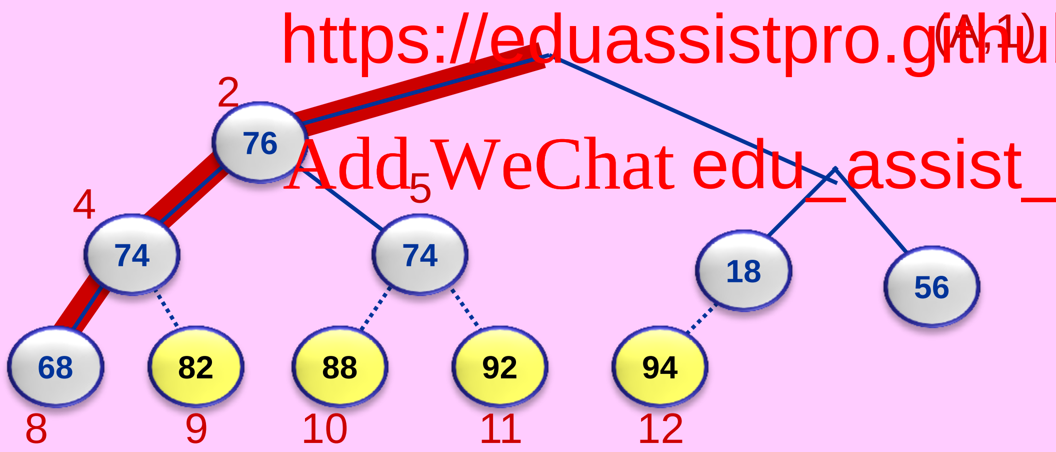
size[A]
= 9



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	76	68	74	74	18	56	68	82	88	92	94

size[A] = 8



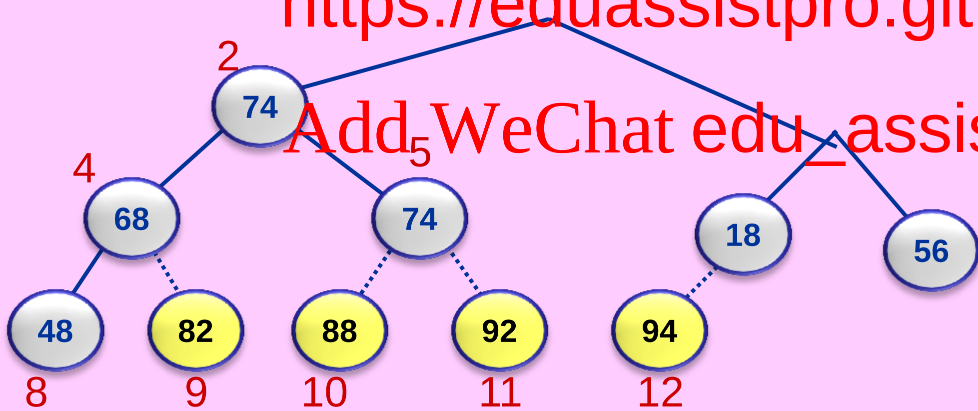
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
76	74	68	68	74	18	56	48	82	88	92	94

Max Item

swap

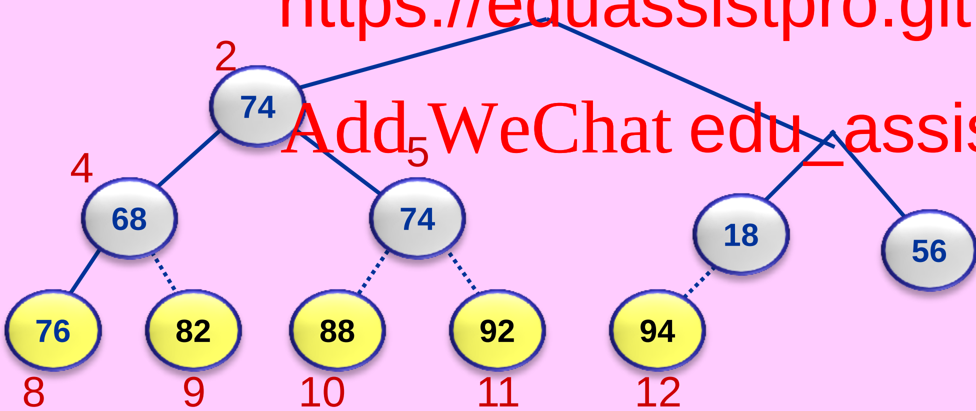
size[A] = 8



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	74	68	68	74	18	56	76	82	88	92	94

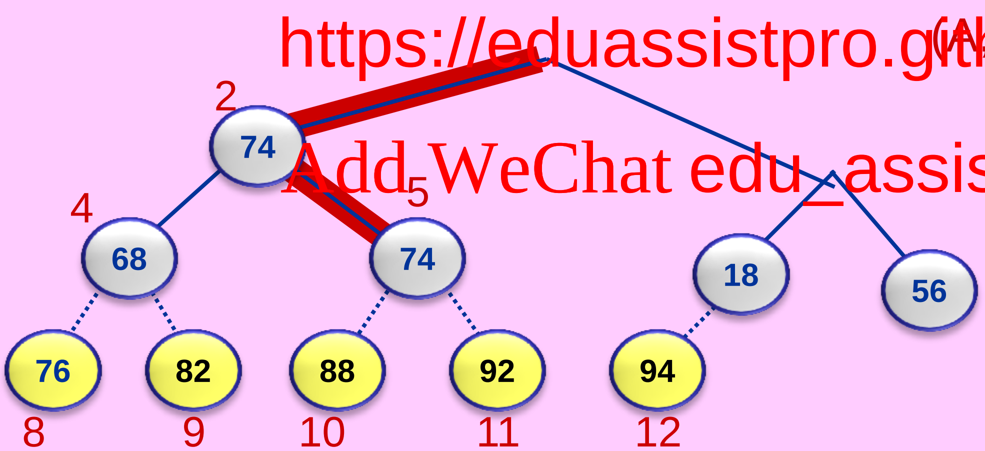
size[A] = 8



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	74	68	68	74	18	56	76	82	88	92	94

size[A]
= 7



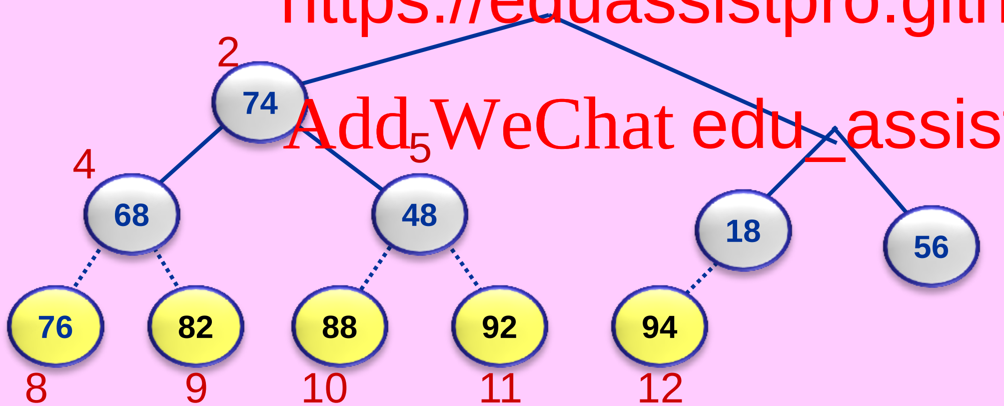
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	74	68	68	48	18	56	76	82	88	92	94

Max Item

swap

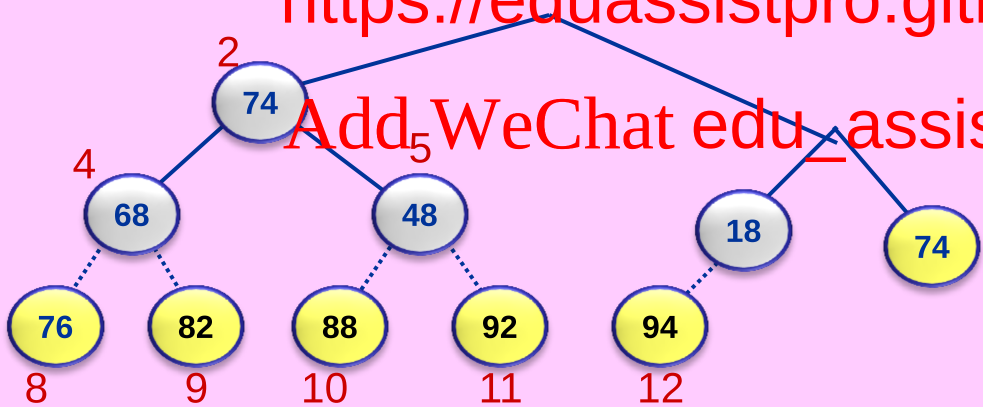
size[A] = 7



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	74	68	68	48	18	74	76	82	88	92	94

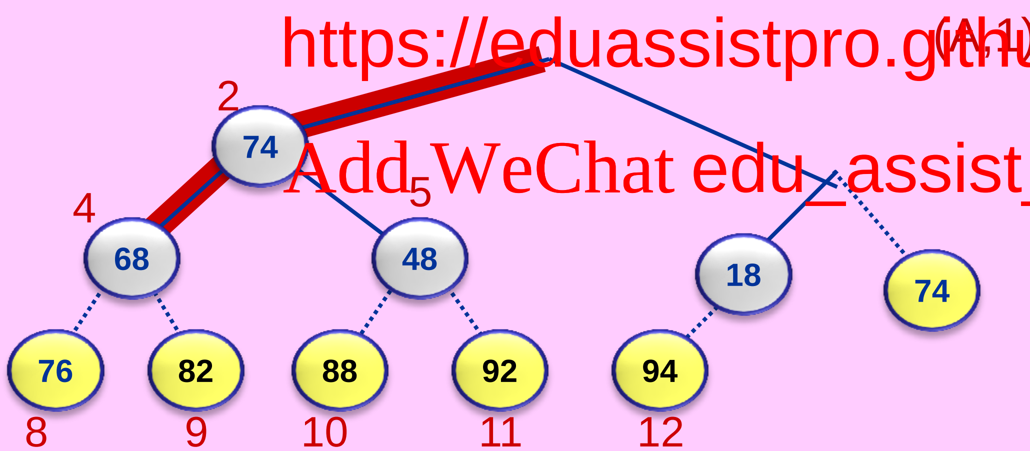
size[A] = 7



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	74	68	68	48	18	74	76	82	88	92	94

size[A] = 6



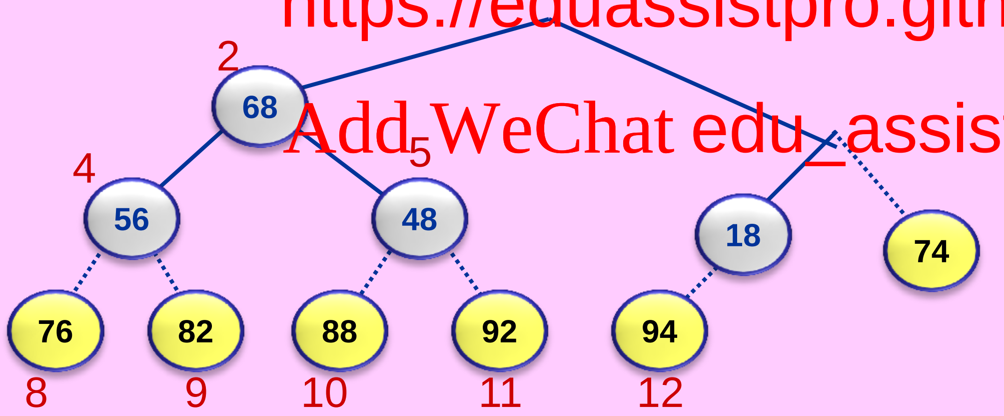
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	68	68	56	48	18	74	76	82	88	92	94

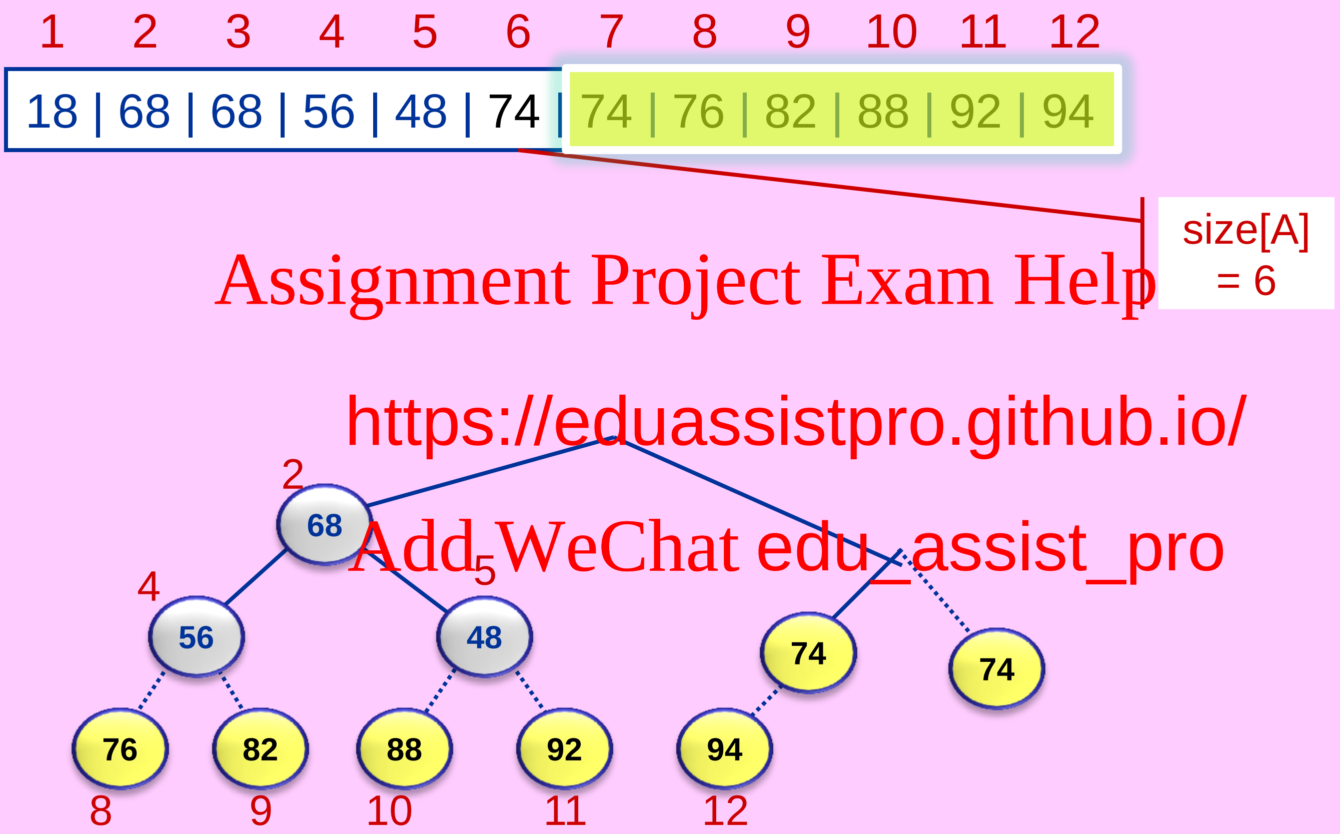
Max Item

swap

size[A] = 6



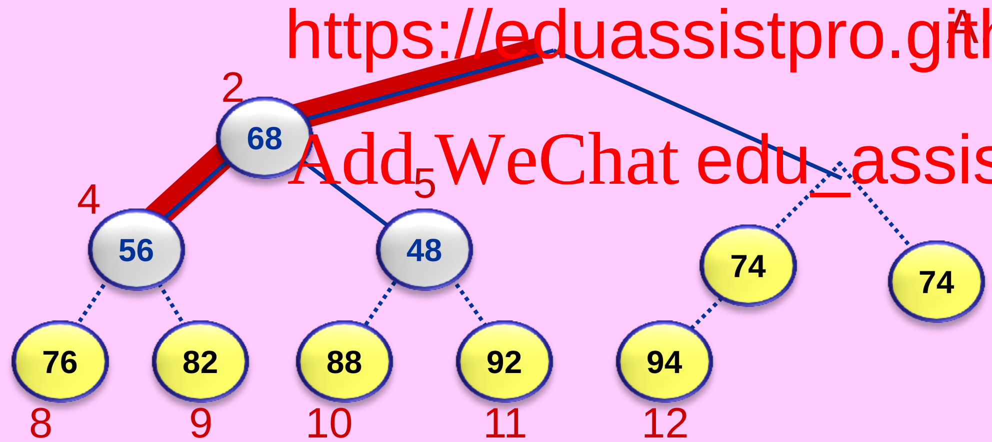
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	68	68	56	48	74	74	76	82	88	92	94

size[A] = 5



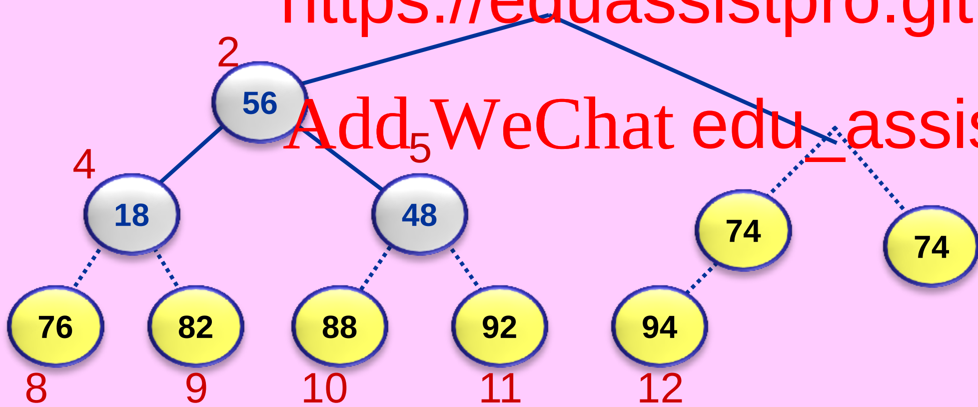
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
68	56	68	18	48	74	74	76	82	88	92	94

Max Item

swap

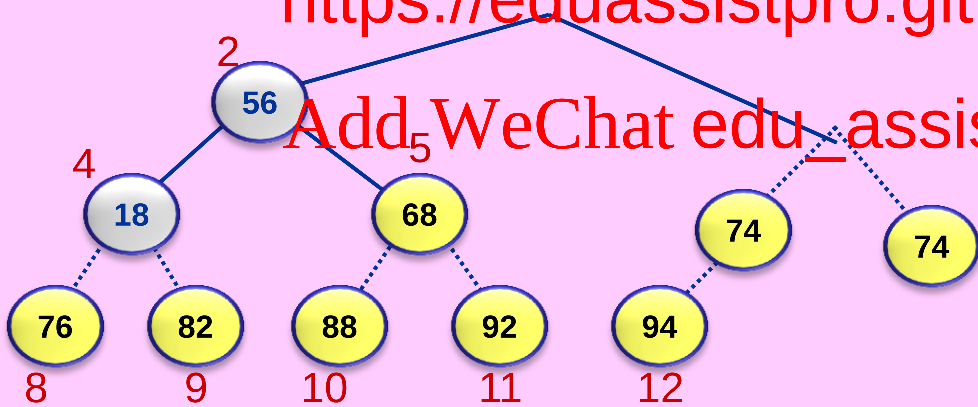
size[A] = 5



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	56	68	18	68	74	74	76	82	88	92	94

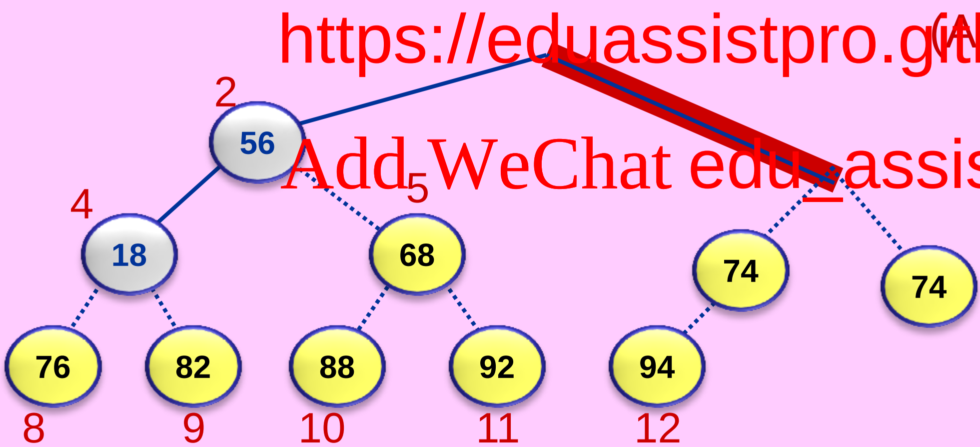
size[A] = 5



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	56	68	18	68	74	74	76	82	88	92	94

size[A]
= 4



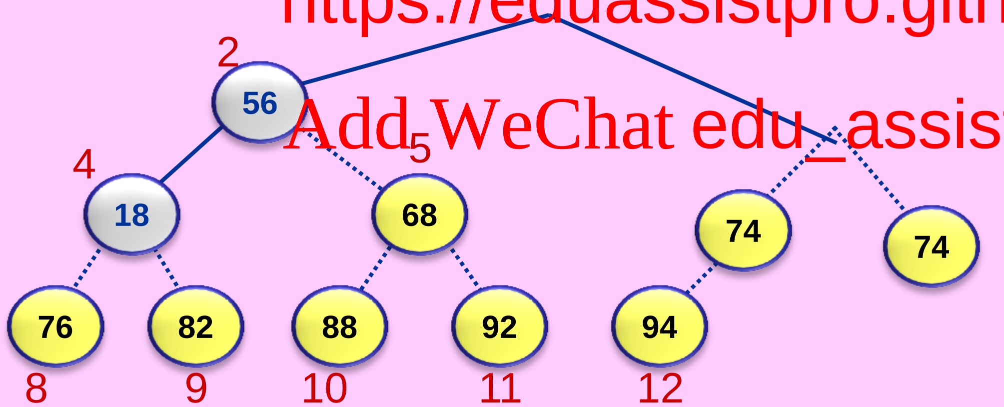
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
68	56	48	18	68	74	74	76	82	88	92	94

Max Item

swap

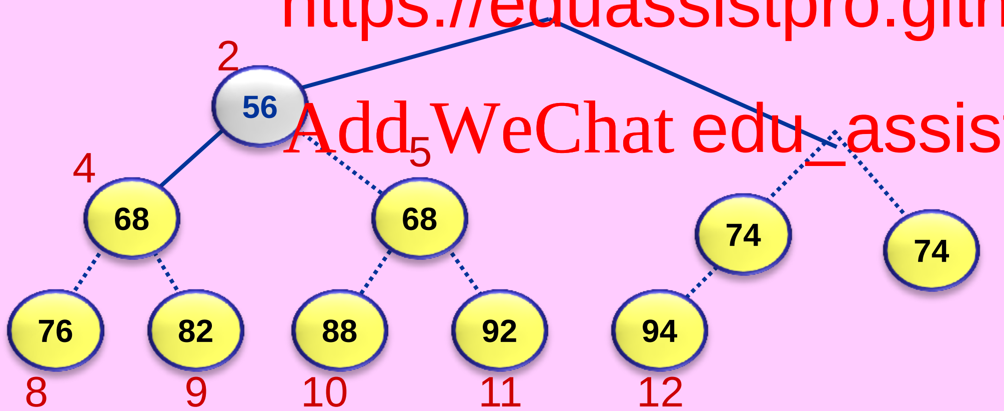
size[A] = 4



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	56	48	68	68	74	74	76	82	88	92	94

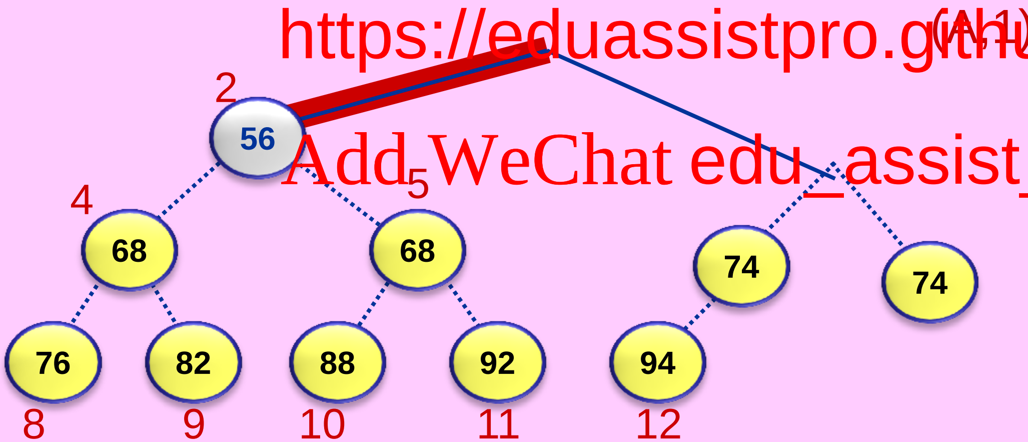
size[A] = 4



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	56	48	68	68	74	74	76	82	88	92	94

size[A] = 3



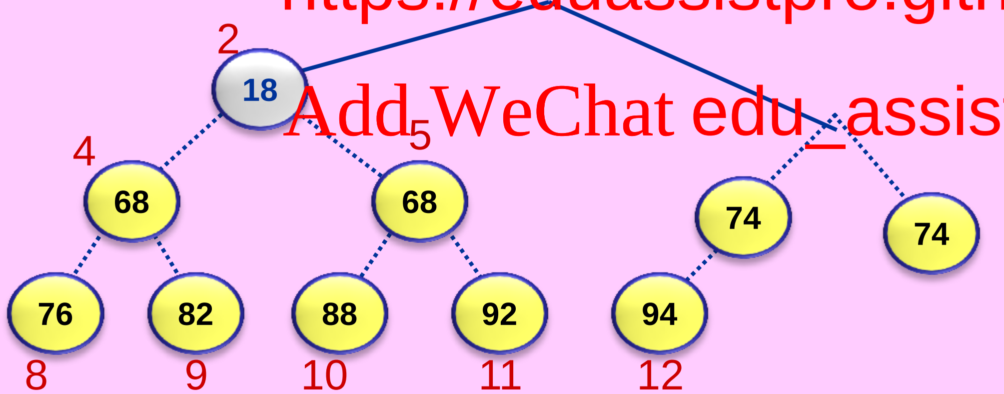
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	18	48	68	68	74	74	76	82	88	92	94

Max Item

swap

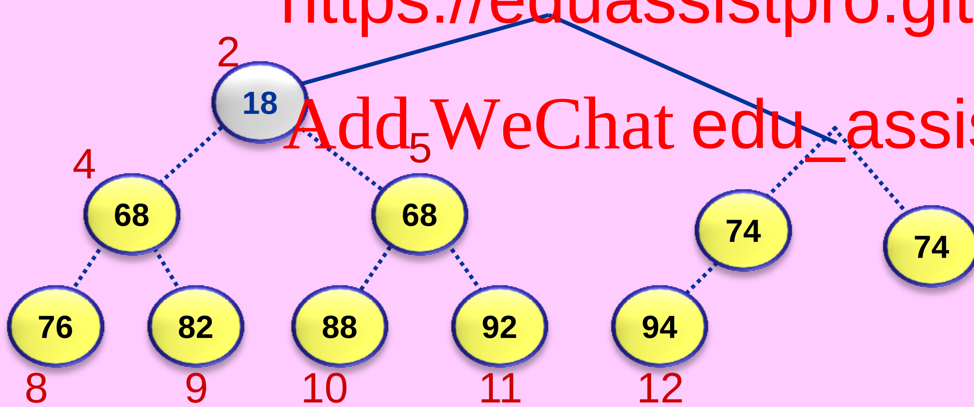
size[A] = 3



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	18	56	68	68	74	74	76	82	88	92	94

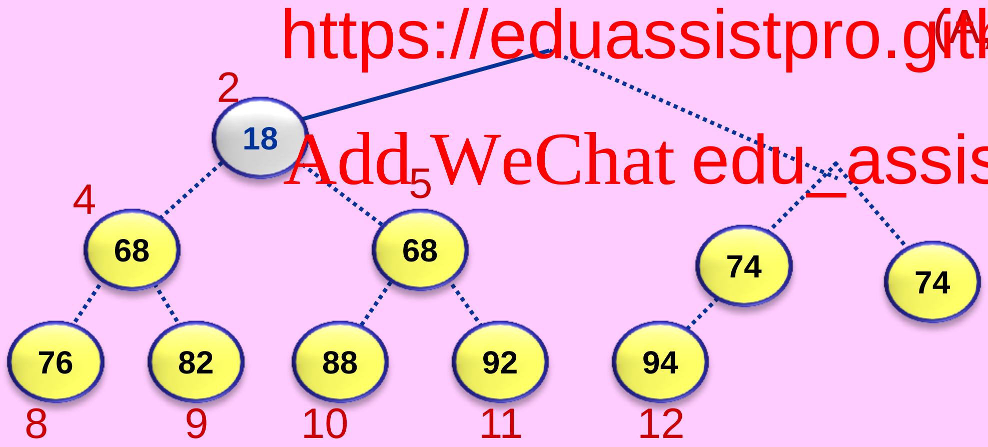
size[A] = 3



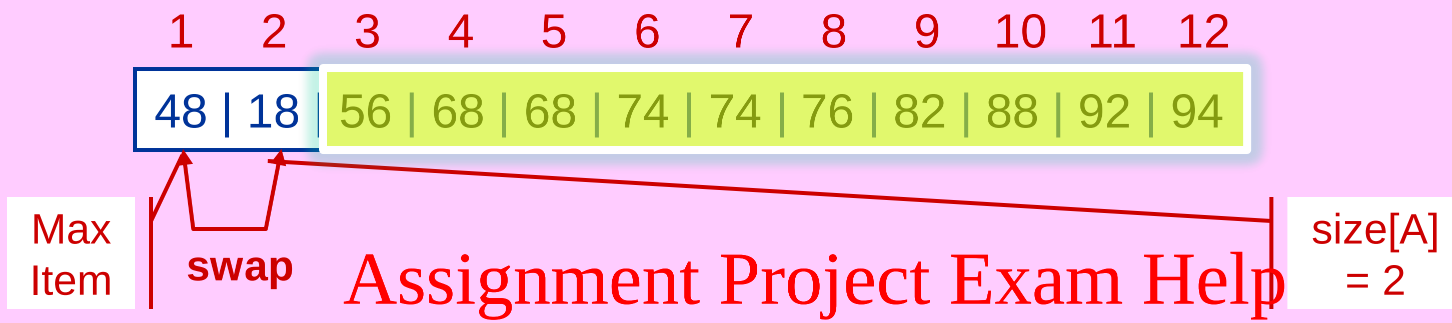
HeapSort Example

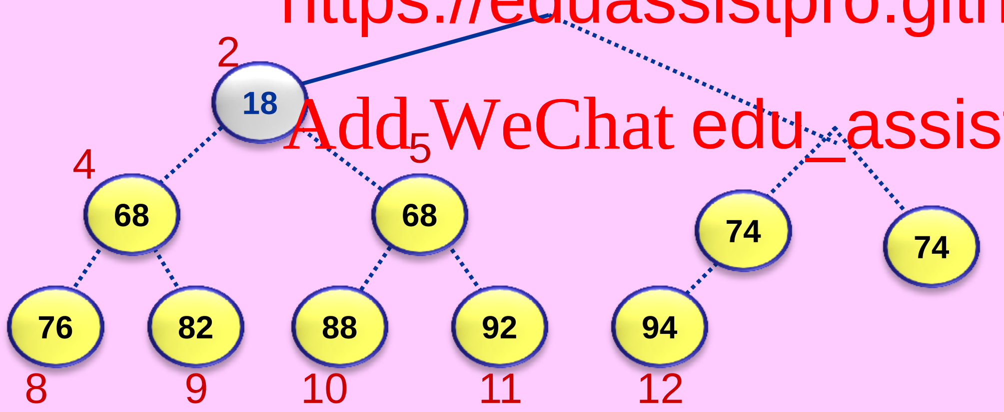
1	2	3	4	5	6	7	8	9	10	11	12
48	18	56	68	68	74	74	76	82	88	92	94

size[A] = 2



HeapSort Example

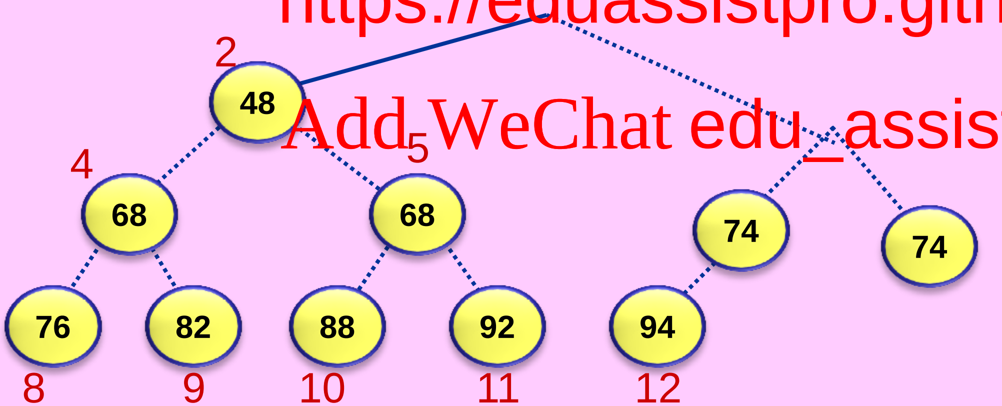




HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	48	56	68	68	74	74	76	82	88	92	94

size[A] = 2

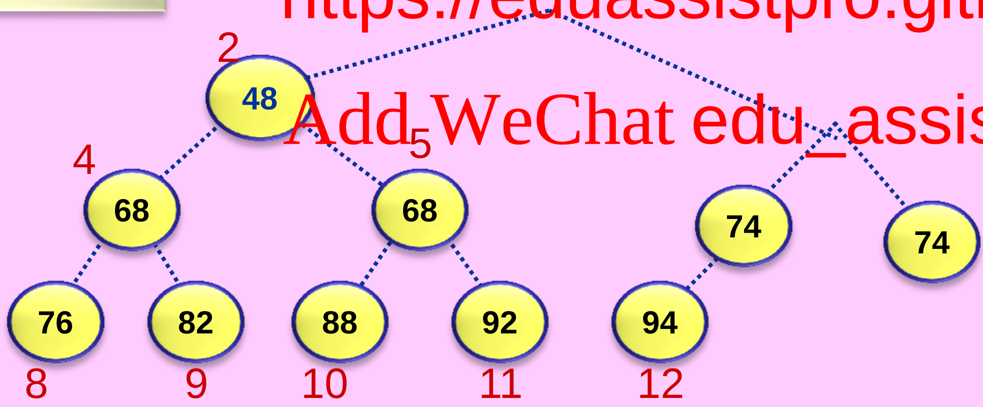


HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	48	56	68	68	74	74	76	82	88	92	94

size[A] = 1

SORTED ARRAY



SORTING
LO ND

My friend, it's impossible for one person to build this ship in one month, using only wood, saw, hammer and nail!

n Black Boxes

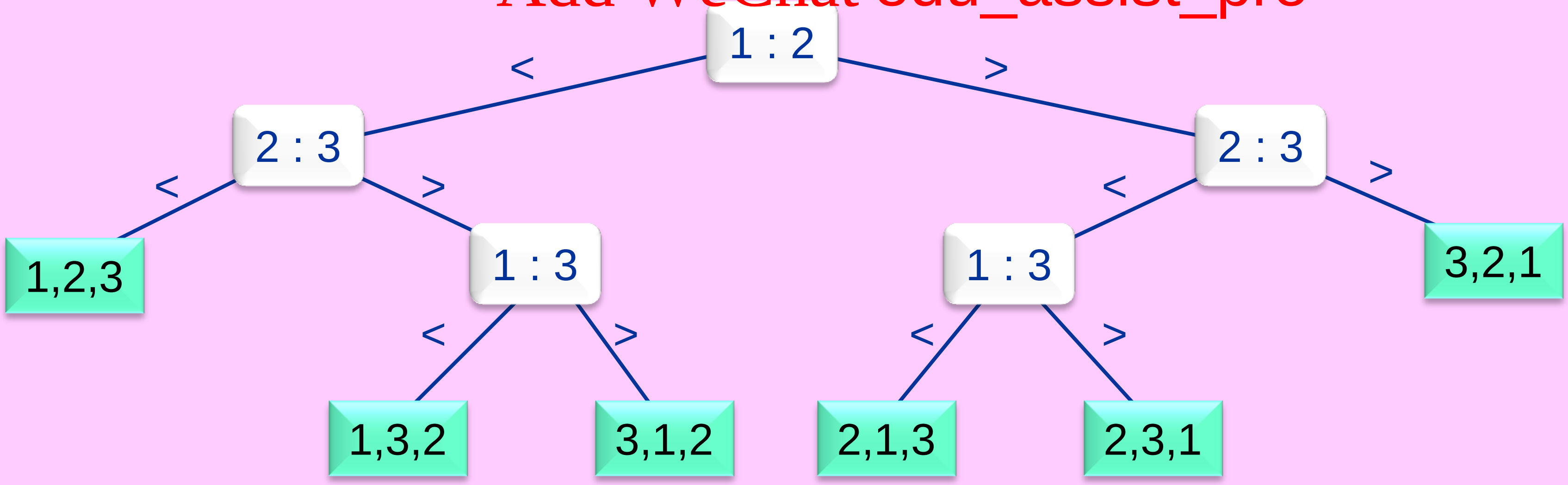
Box 1 **Box 2** **Box 3**

The **adversary shows** you 3 black boxes.
Each contains a distinct number.
Your task is to order these boxes in increasing order of the numbers they contain.
You are not allowed to open the boxes or examine their contents in any way.
You can only **ask the oracle** questions of the type:

"Is the number in Box x < the number in Box y?" ("x : y" for short)

where x and y are in { ice.

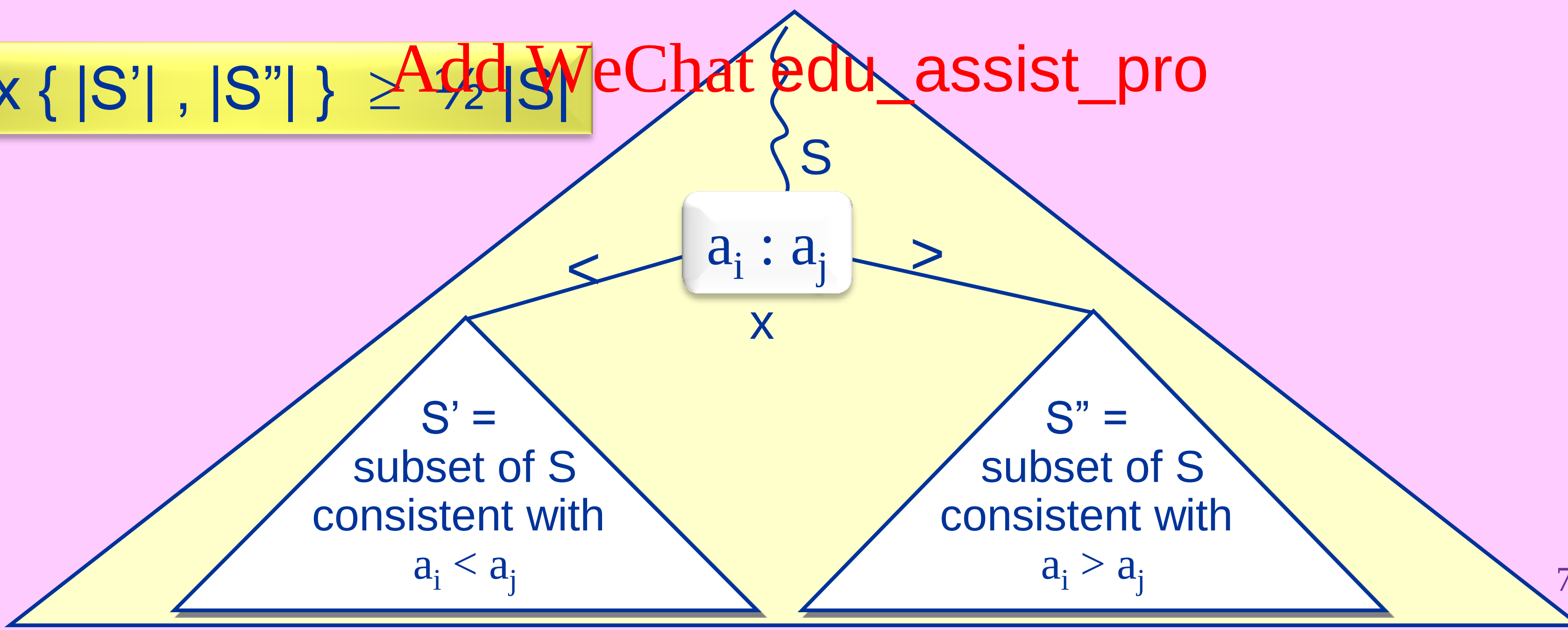
In the worst case, **how m** ou need to ask?
Below is a possible scenario called **the deci** strategy you might use:



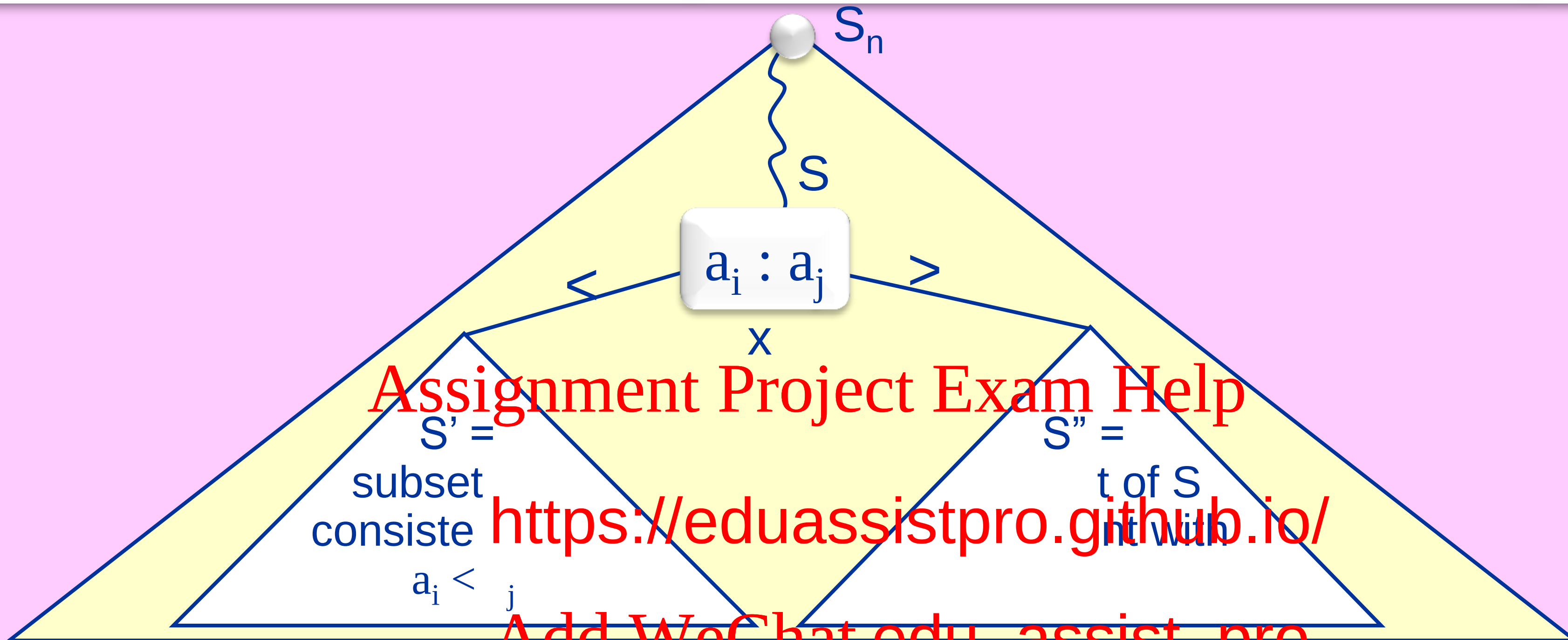
The Decision Tree Model

- General Sorting: no assumption on item type.
- We only know there is a linear ordering defined on the items.
- Comparison types: =, < , >   (or any question with binary (yes/no) answer)
- Relative ordering information is gained by comparing pairs of items.
- For simplicity first assume all input items are distinct.
- n! possible permutations of input items $\langle a_1, a_2, \cdots, a_n \rangle$
- S_n = set of all n! permutations.
- S = set of possi   p $(a_i : a_j)$ at node x.

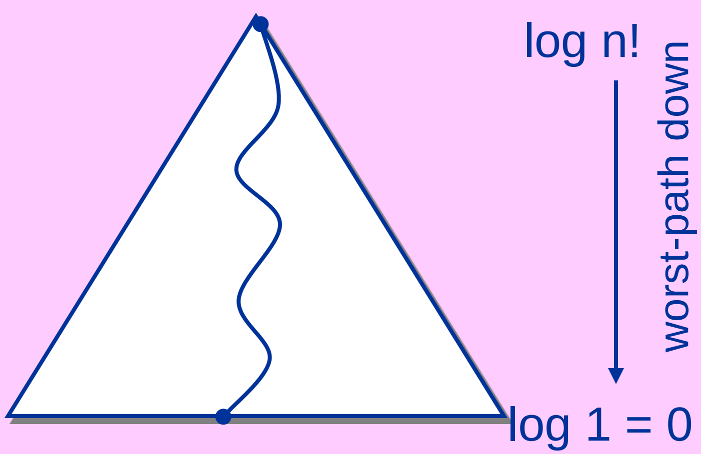
max { |S'| , |S"| } $\geq$ ½ |S|



Information Theory Lower Bound on Sorting



log (max { |S'| , |S"| S|) – 1

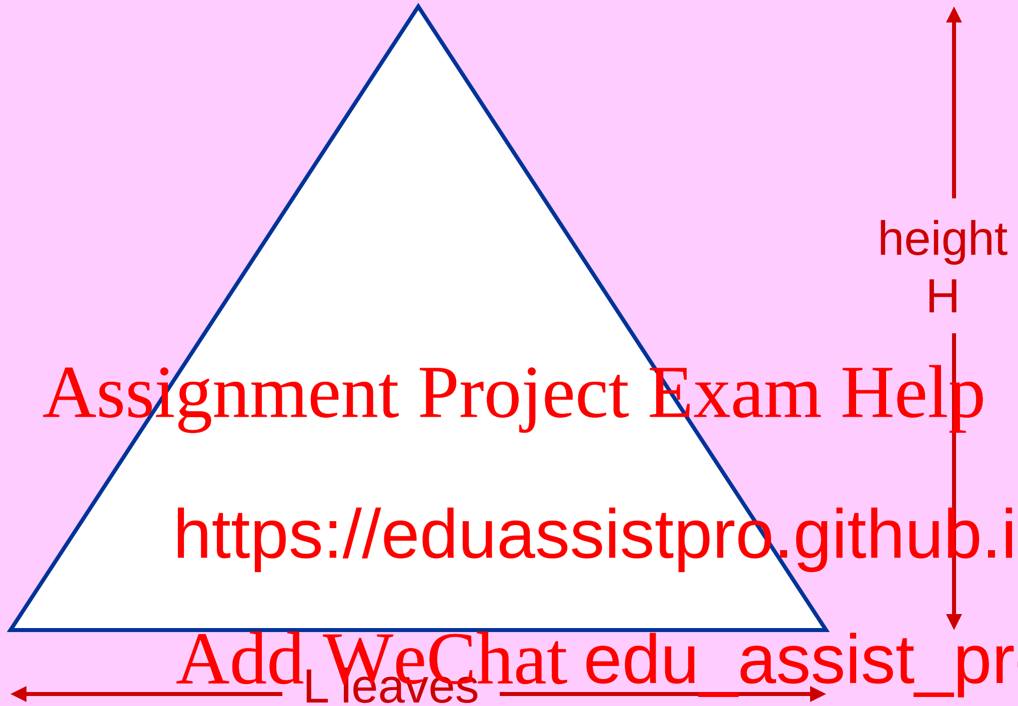


Sorting Lower Bound:

length of worst path
down the decision tree is
$\geq$ log n!

$\geq \Omega$(n log n)

Information Theory Lower Bound



# of possible outcomes $\le L \le 2^H$

In a 3-way decision tree, it’s log base 3.

Worst- case # of comparisons $\ge H \ge \log$ (# possible outcomes).

More Decision Tree Lower Bounds

FACT 0: Any comparison based sorting algorithm requires at least Ω(n log n) comparisons in the worst-case.

FACT 1: Any comparison based sorting algorithm requires at least Ω(n log n) comparisons even in the average-case.

Proof: # Leaves in T = m_T = m_L + m_R .

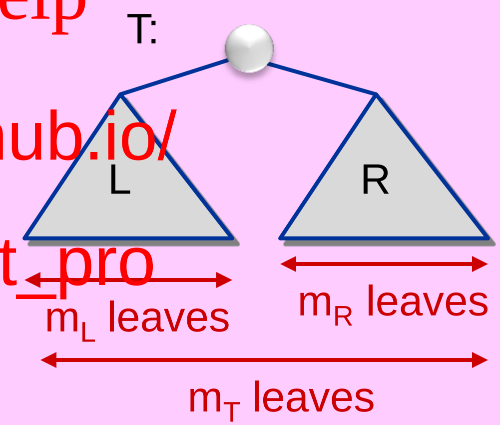
Average leaf depth = Sum of leaf depths / # leaves.

Proof by induction that

Sum of leaf depths

Basis (one leaf): Trivial.

Induction: Sum of leaf depths in T



$m_T \geq n!$

$$\begin{aligned}
&= \Sigma\{\, depth(x, T) : x \text{ is a leaf in } T\} \\
&= \Sigma\{\, depth(x, T) : x \text{ is a leaf in } L\} \\
&\quad + \Sigma\{\, depth(x, T) : x \text{ is a leaf in } R\} \\
&= \Sigma\{\, 1 + depth(x, L) : x \text{ is a leaf in } L\} \\
&\quad + \Sigma\{\, 1 + depth(x, R) : x \text{ is a leaf in } R\} \\
&= m_T + \Sigma\{\, depth(x, L) : x \text{ is a leaf in } L\} \\
&\quad\quad + \Sigma\{\, depth(x, R) : x \text{ is a leaf in } R\} \\
&\geq m_T + m_L \log m_L + m_R \log m_R && \text{(by Induction Hypothesis)} \\
&\geq m_T + (\tfrac{1}{2} m_T)\log(\tfrac{1}{2} m_T) + (\tfrac{1}{2} m_T)\log(\tfrac{1}{2} m_T) && \text{(m log m is convex:} \\
&= m_T \log m_T && \text{min at } m_L = m_R = \tfrac{1}{2} m_T)
\end{aligned}$$

More Decision Tree Lower Bounds

FACT 0: Any comparison based sorting algorithm requires at least $\Omega(n \log n)$ comparisons in the worst-case.

FACT 1: Any comparison based sorting algorithm requires at least $\Omega(n \log n)$ comparisons even in the average-case.

FACT 2: Any comparison based Merging algorithm of two sorted lists, each with n elements, requires at least $\Omega(n)$ comparisons in the worst-case.

Proof: There are 2n out

le outcomes?

The outcome is uniquely determined t spots that the elements of the first list occupy.

How many possibilities are there to pi of 2n spots?

Answer: # outcomes =

log (# outcomes) = log (2n)! – 2 log n! ≈ 2n – ½ log n ≥ $\Omega(n)$.

Use eq. (3.18): approximation formula for n!, on page 57 of [CLRS]

Algebraic Decision Tree

- Suppose we want to sort n **real** numbers.
- Why should our computation be restricted to only element comparisons?
- What about something like:

  Is $(3a_1 + 6a_2)*(7a_5 - 6a_9) - 15\, a_3*a_4 - 8 < 0$?

- **Michael Ben Or [1983]**, using an earlier result of [Petrovskiĭ, Oleĭnik, Thom, Milnor], addressed that concern by developing

  the more powerf **e Model**.

- In this tree model, inter
  - a computation node: it does arith put items & constants.
  - a decision node: it asks whether a computed quantity is =0, or <0, or >0.

- He showed that (even if we assume the cost of computation nodes are free of charge) there must be paths with many decision type nodes.
- For instance, he showed the following result:

FACT 3: Sorting requires at least $\Omega(n \log n)$ comparisons in the worst-case, even in the Algebraic Computation Tree Model.

Ben Or's Lower Bound Method

- A sequence $\langle x_1, x_2, \cdots, x_n \rangle$ of n real numbers can be interpreted as a point in $\Re^n$, the n dimensional real space.
- Similarly, for any permutation π of $\langle 1,2, \ldots, n\rangle$, $\langle x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)}\rangle$ is a point in that space.
- Permutation $\langle x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)}\rangle$ is the sorted order if and only if the input point $\langle x_1, x_2, \cdots, x \rangle$ falls in the following subset of the space:

$S(\pi) = \{ \langle x_1, x_2$ … $= 1..n\text{-}1\}$.

- The entire n dimens … o such regions.
- Each algebraic expression computed … is sign invariant in a subset of $\Re^n$.

- The intersection of these subsets must fall within a unique $S(\pi)$ as a certificate that π is the correct sorted permutation of the input.
- The lower bound argument is that we need many such intersections to achieve the goal.

More Algebraic Computation Tree Lower Bounds

FACT: The following problems have worst-case **$\Omega(n \log n)$** lower bounds in the Algebraic Computation Tree Model.

Element Uniqueness:
Are any two elements of the input sequence $\langle x_1, x_2, \cdots, x_n \rangle$ equal?

Set Intersection:

Given two sets $\{x_1, x$ … $y_n\}$, do they intersect?

Set Equality:
Given two sets $\{x_1, x_2, \cdots, x_n\}$ and $\{y$ … $\}$, are they equal?

3SUM:
Given a set $\{x_1, x_2, \cdots, x_n\}$, does it contain 3 elements x_i, x_j, x_k, such that $x_i + x_j + x_k = 0$?

Note that the decision tree model cannot give such results, since each of these problems has only two possible outcomes: YES or NO.

SPECIAL PURPOSE

ALGOR S

- General Purpose Sorting:
  Comparisons, Decision Tree, Algebraic Decision or Computation Tree …

- Suppose you want to sort n numbers
  with the pre-condition that each number is 1, 2, or 3.
  How would you sort them?
  We already saw that 3-Partition can sort them in-place in O(n) time.

- Pre-cond: … Given universe of item values is finite (preferably "small").

  - E.g., integers i it integers in base B]
    (B are given constants).
  - We can use special purpose sorting
  - They break the Ω(n log n) lower bo
  - They are outside the algebraic computation/comparison model.
  - E.g., they use floor/ceiling integer rounding, use value as array index, …

- Examples of Special Purpose Sorting Algorithms:
  Bucket Sort, Distribution Counting Sort,
  Radix Sort, Radix Exchange Sort, …

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K)

Pre-Cond: input is array A[1..n], each A[t] $\in$ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

1. **for** v $\leftarrow$ 0 .. K-1 **do** Count[v] $\leftarrow$ 0 § initialize
2. **for** t $\leftarrow$ 1 .. n **do** Count[A[t]] $\leftarrow$ Count[A[t]] + 1 § increment

   § Count[v] = # input items A[t] = v

   . . . More steps to come

end

Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Count =

0	1	2	3
2	3	2	3

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] $\in$ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2

3. **for** v ← 1 .. K-1 **do** Count[v] ← Count[v] + Count[v –1] § aggregate

§ Now Count[v] = # input items A[t] ≤ v

§ Count[v] = Last output index for item with value v

. . .

end

Example: A =

	1	2	3	4	5	6				
A =	0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =										

	2	5	7	10
Count =	2	3	2	3
	0	1	2	3

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] $\in$ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   - Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   - Count[A[t]] ← Count[A[t]] – 1 § decrement to preceding position

   end-for

   . . .

end

Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =

									3

9 = Temp array position for next 3

Count =

0	1	2	3
2	5	7	~~10~~

Distribution Counting Sort

Algorithm CountingSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   - Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   - Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position

   end-for

   . . .

end

Example: A = n = 10, K = 4

	1	2	3	4	5	6			9	
A =	0	3	1	3	2	0	2	1	1	3
Temp =					1					3

		4		
Count =	2	~~5~~	7	9
	0	1	2	3

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) **§ O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position
   end-for

. . .

end

1 2 3 4 5 6 9

Example: A = 0 | 3 | 1 | 3 | 2 | 0 | 2 | 1 | 1 | 3   n = 10, K = 4

Temp = | | | 1 | 1 | | | | | 3

3

Count = 2 | ~~4~~ | 7 | 9

0 1 2 3

Distribution Counting Sort

Algorithm CountingSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position
   end-for

. . .

end

1 2 3 4 5 6 9

Example: A = | 0 | 3 | 1 | 3 | 2 | 0 | 2 | 1 | 1 | 3 | n = 10, K = 4

Temp = | | | | 1 | 1 | | 2 | | | 3 |

6

Count = | 2 | 3 | ~~7~~ | 9 |

0 1 2 3

Distribution Counting Sort

Algorithm CountingSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   - Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   - Count[A[t]] ← Count[A[t]] – 1 § decrement to preceding position

   end-for

   . . .

end

Example: n = 10, K = 4

	1	2	3	4	5	6			9	
A =	0	3	1	3	2	0	2	1	1	3
Temp =		0		1	1		2			3

	0	1	2	3
Count =	~~2~~ 1	3	6	9

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position
   end-for

. . .

end

Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =

	0		1	1	2	2			3

Count =

0	1	2	3
1	3	~~6~~ 5	9

Distribution Counting Sort

Algorithm CountingSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

```
     Steps 1, 2, 3
4.   for t ← n downto 1 do                 § stable sort
        Temp[Count[A[t]] ← A[t]            § place items in final position Temp array
        Count[A[t]] ← Count[A[t]] − 1      § decrement to preceding position
     end-for
      . . .
end
```


Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =

	0		1	1	2	2		3	3

Count =

0	1	2	3
1	3	5	~~9~~ 8

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   Temp[Count[A[t]]] ← A[t] § place items in final position Temp array
   Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position
   end-for

. . .

end

Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =

	0	1	1	1	2	2		3	3

Count =

0	1	2	3
1	~~3~~ 2	5	8

Distribution Counting Sort

Algorithm CountingSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   - Temp[Count[A[t]] ← A[t] § place items in final position Temp array
   - Count[A[t]] ← Count[A[t]] −1 § decrement to preceding position

   end-for

   . . .

end

Example: A =

	1	2	3	4	5	6			9	
A =	0	3	1	3	2	0	2	1	1	3
Temp =		0	1	1	1	2	2	3	3	3

n = 10, K = 3

	0	1	2	3
Count =	1	2	5	~~8~~ 7

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K)　　　**§ O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do**　　　§ stable sort
    Temp[Count[A[t]]] ← A[t]　　　§ place items in final position Temp array
    Count[A[t]] ← Count[A[t]] −1　　　§ decrement to preceding position
  end-for

  . . .

end

Example: A =

1	2	3	4	5	6			9	
0	3	1	3	2	0	2	1	1	3

n = 10, K = 4

Temp =

0	0	1	1	1	2	2	3	3	3

Count =

0	1	2	3
~~1~~ 0	2	5	7

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3

4. **for** t ← n downto 1 **do** § stable sort
   Temp[Count[A[t]]] ← A[t] § place items in final position Temp array
   Count[A[t]] ← Count[A[t]] − 1 § decrement to preceding position
   end-for

. . .

end

Example:

	1	2	3	4	5	6			9	
A =	0	3	1	3	2	0	2	1	1	3
Temp =	0	0	1	1	1	2	2	3	3	3

n = 10, K = 4

	0	1	2	3
Count =	0	2	5	7

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

Steps 1, 2, 3, 4

5. **for** t ← 1 .. n **do** A[t] ← Temp[t]] § copy items back into A

. . .

end

Example: A =

1	2	3	4	5	6			9	
0	0	1	1	1	2	2	3	3	3

n = 10, K = 4

Temp =

0	0	1	1	1	2	2	3	3	3

Count =

0	1	2	3
0	2	5	7

Distribution Counting Sort

Algorithm **CountingSort**(A[1..n], K) **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] $\in$ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

1. **for** v ← 0 .. K-1 **do** Count[v] ← 0 § initialize
2. **for** t ← 1 .. n **do** Count[A[t]] ← Count[A[t]] + 1 § increment

    § Count[v] = # input items A[t] = v
3. **for** v ← 1 .. K-1 **do** v –1] § aggregate

    § Now Count[v] =

    § Count[v] = Last output index for ite
4. **for** t ← n downto 1 **do** § s

    Temp[Count[A[t]] ← A[t] § place items in final position Temp array

    Count[A[t]] ← Count[A[t]] –1 § decrement to preceding position

    end-for
5. **for** t ← 1 .. n **do** A[t] ← Temp[t]] § copy items back into A

end

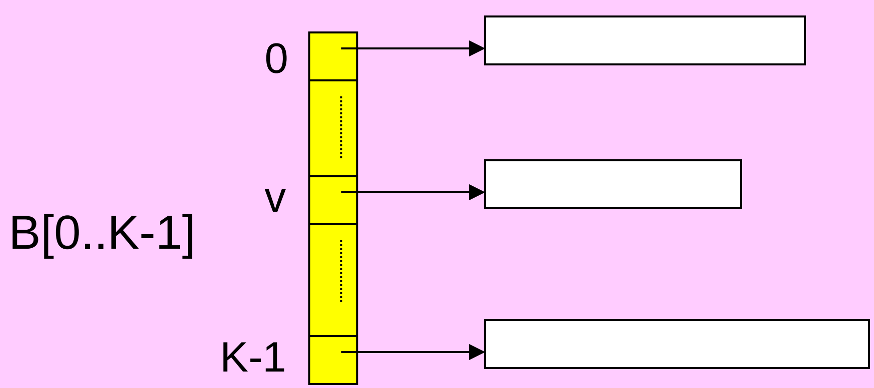
Bucket Sort: Deterministic Version

Algorithm BucketSort(A[1..n], K) § **O(n + K) Time & Space**

Pre-Cond: input is array A[1..n], each A[t] ∈ [0 .. K-1], t = 1..n

Post-Cond: A is rearranged into sorted order

```
    for v ← 0 .. K-1 do Empty(B[v])                    § initialize
    for t ← 1 .. n do Enqueue(A[t], B[A[t]])           § fill buckets
    t ← 1
    for v ← 0 .. K-1 do            § empty buckets back into array in order
        while B[v] not
            A[t] ← D
            t ← t +1
        end-while
end
```

Use K buckets B[0..K-1].
Each bucket B[v] is a queue
for stable sorting.

Radix Sort

Sorting on digits from <u>least</u> significant to <u>most</u> significant digit position.

Algorithm **RadixSort**(A[1..n], D, B) § **O(D(n + B)) Time**

Pre-Cond: input is array A[1..n], A[t] is a D digit integer in base B, t = 1..n

Post-Cond: A is rearranged into sorted order

for d ← 0 .. D-1 **do** **stable sort** A[1..n] on digit d

rt)

LI: A[1..n] is so igits

end

4 5 3	→	4 5 3	→	3 0 5	→	1 9 8
5 6 8		8 6 4		4 5 3		3 0 5
1 9 8		3 0 5		8 6 4		4 5 3
8 6 4		5 6 8		5 6 8		5 6 8
3 0 5		1 9 8		1 9 8		8 6 4

Radix Sort: Proof of Loop Invariant

Loop Invariant by Induction on d:

Induction Hypothesis:
at the end of iteration d:
X appears before Y $\Rightarrow$ X[d..0] $\leq$ Y[d..0]

X[d..0]	……	X_d	X_{d-1} … X_0
Y[d..0]	……	Y_d	Y_{d-1} … Y_0

Proof: X appears before Y $\Rightarrow$ X[d] $\leq$ Y[d] iteration d)

Case 1: X[d] < Y[d] $\Rightarrow$ X[d..0] < Y[d..0]

Case 2: X[d] = Y[d] $\Rightarrow$ X[d-1..0] appeared before Y[d-1..0] in previous iteration (by stable sorting)

$\Rightarrow$ X[d-1..0] $\leq$ Y[d-1..0] (by Induction Hypothesis) (null for the basis)

$\Rightarrow$ X[d..0] $\leq$ Y[d..0].

Radix Sort: the wrong way

4 5 3	1 9 8	3 0 5	4 5 3
5 6 8	3 0 5	4 5 3	8 6 4
1 9 8	4 5 3	5 6 8	3 0 5
8 6 4			5 6 8
3 0 5			1 9 8

Explain why it’s **NOT** working correctly.

Radix Exchange Sort

Sorting on <u>most</u> significant digit first á la QuickSort Partitioning.
First call: RadixExchangeSort(A[1..n], D-1, B).

Algorithm RadixExchangeSort(A[s..t], d, B) § O(D(n + B)) Time

Pre-Cond: array A[s..t] is sorted on digits d+1 .. D-1.

Post-Cond: A[s..t] is sorted order on all digits 0 .. d, d+1 .. D-1

if d < 0 or s ≥ t **then return**

Partition A[s..t] into B contiguous (possibly empty) subarrays A[s(v) .. t(v)],

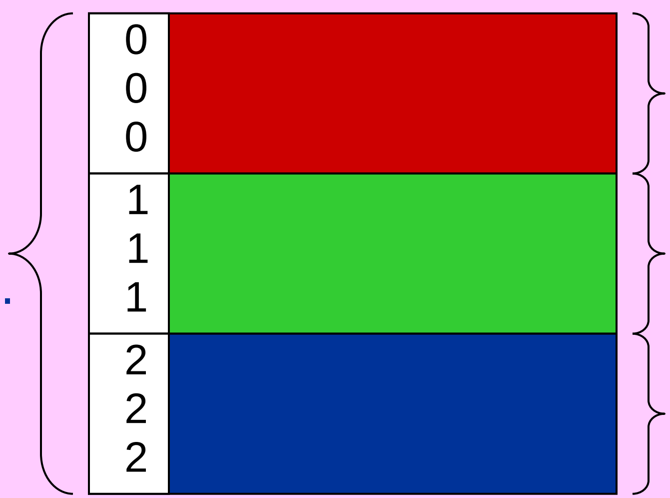
v = … tem in A[s(v) .. t(v)] is v.

for v ← 0 .. B-1 **do** …)], d-1, B)

end

d d-1 …..

First partition on digit d position.



Then **separately** sort each of these sub-arrays on lower order digits recursively.

Radix Exchange Sort: Binary Example

1 0 0 1	0 1 0 0	0 0 0 1	0 0 0 1	0 0 0 1
0 1 0 0	0 0 0 1	0 1 0 0	0 1 0 0	0 1 0 0
1 1 1 1	0 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1
0 0 0 1	0 1 1 1		1 1 1	0 1 1 1
0 1 0 1	1 0 0 1		0 0 1	1 0 0 1
1 1 1 0	1 1 1 1	1 0 1 1	1 1	1 0 1 0
1 0 1 1	1 1 1 0	1 0 1 0	1 0 1 0	1 0 1 1
0 1 1 1	1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 0
1 0 1 0	1 0 1 0	1 1 1 0	1 1 1 0	1 1 1 1

Quiz Question

INPUT: array A[1..n], each element is a positive integer at most n^2.
OUTPUT: array A in sorted order.

QUESTION: How fast can this be done?

ANSWER: CountingSort or BucketSort will take $\Theta(n^2)$ TIME and SPACE.
That's bad. ☹

Use Ra

& B to minimize:

Time = O

$X \in [1 .. n^2]$ can be thought of as 2 digits in base n:

$$X - 1 = (X_1 X_0)_n = n X_1 + X_0$$

$$X_1 = (X - 1) \text{ div } n$$

$$X_0 = (X - 1) \text{ mod } n$$

Choose D = 2, B = n. Time = O(D(n+B)) = O(n).

SELECTION

Find me the **median** pro [illegible] n Ontario

and the **top 10%** student SAT scores in North America

The Selection Problem

INPUT: A sequence $A = \langle a_1, a_2, \cdots, a_n \rangle$ of n arbitrary numbers, and an integer K, $1 \le K \le n$.

OUTPUT: The K^{th} smallest element in A. That is, an element $x \in A$ such that there are at most K-1 elements $y \in A$ that satisfy $y < x$, but there are at least K elements $y \in A$ that satisfy $y \le x$.

Example: $A = \langle 12, 15, 43, 15, 62, 88, 76 \rangle$.

	5	6	7
K^{th} s	62	76	88

Applications:

1. Order Statistics, Partitioning, Cluste
2. In divide-and-conquer: divide at the median value, i.e., with $K = \lceil n/2 \rceil$.

Solution 1: Sort A, then return the element at position K in the sorted order. This takes $\Omega(n \log n)$ time in the worst case.

O(n) time solution for some special values of K, e.g.,

$K=1 \longrightarrow$ minimum element. $K=n \longrightarrow$ maximum element.

WLOGA: $K \le \lceil n/2 \rceil$ (the median value): K^{th} smallest = $(n-1-K)^{th}$ largest.

Solution 2

- How would you find the 2^{nd} smallest element in A, i.e., with K = 2?
- Incrementally scan A and keep track of the K smallest items (1^{st} & 2^{nd} smallest).
- Generalize this idea using a heap:
  Incrementally scan A[1..n] and rearrange the items so that A[1..K] is a MAX HEAP holding the K smallest items seen so far.

1 K t n

A = K smallest in A[1..t] | | Not yet scanned

Max Heap

If the next item A[t] is tss than A[1] (the K^{th} smallest so far), remove A[1] from lace A[t] in the heap:

Algorithm **Select**(A[1..n], K

ConstructMaxHeap(A[1..K]) § O(K) time, heap size = K
for t ← K+1 . . n **do** § O(n) iterations
  if A[t] < A[1] **then** A[t] ↔ A[1]
    DownHeap(A,1) § O(log K) time
return A[1]
end

Time = O(n log K). This is O(n) if K = O(1).

Solution 3

- Turn A[1..n] into a MIN HEAP of size n, then do K successive DeleteMins. The K smallest items will come out in sorted order.

Algorithm Select(A[1..n], K)

```
    ConstructMinHeap(A[1..n])        § O(n) time, heap size = n
    for t ← 1 .. K do                § K iterations
        x ← DeleteMin(A)             § O(log n) time
    return x
end
```


er reason why we want Construct Heap in linear time

Time = O(n + K log n).

This is O(n) if K = O(n/log n).

Getting closer to covering up to the median range $K = \lceil n/2 \rceil$ ☺

Solution 4: Randomized QuickSelect

- Hoare: Adapt the QuickSort strategy. Luckily we need to do only one of the two possible recursive calls! That saves time on average.

Algorithm **QuickSelect**(S, K)

§ Assume $1 \le K \le |S|$. Returns the K^{th} smallest element of S.

if $|S| = 1$ **then return** the unique element of S

$p \leftarrow$ a random element in S

Partition S:

$>$

Assume adversary picks
e larger recursive call

if $|S_<| \ge K$ **then return** Quick

else if $|S| - |S_>| \ge K$ **then return** p

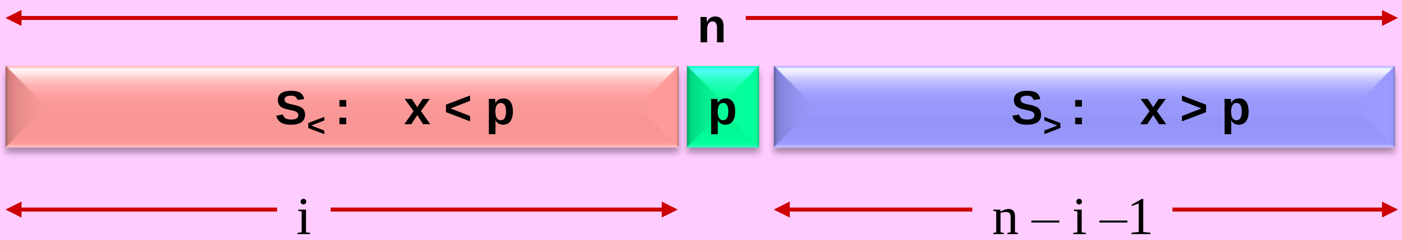
else return QuickSelect$(S_> , K - |S| + |S_>|)$

end

$S_<$: $x < p$	$S_=$: $x = p$	$S_>$: $x > p$

$T(|S|) = \max\{ T(|S_<|) , T(|S_>|) \} + \Theta(|S|)$, $T(n) = \Theta(1)$, for n=0,1.

QuickSelect Running Time



WLOG Assume: $|S_=| = 1$. If it's larger, it can only help!

$T(n) = \max\{T(i)$, … $\Theta(1)$, for n=0,1.

Worst-Case:

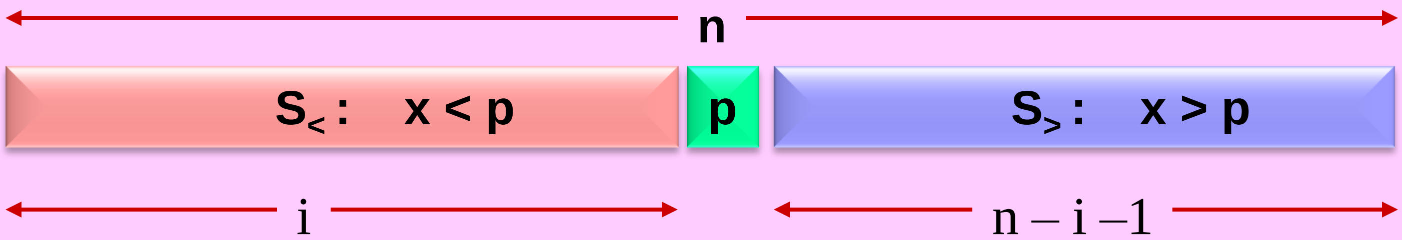
$$T(n) = \max_i \{ \max\{T(i) , T(n- \ldots : i = 0 .. n-1 \}$$

$$= \max_i \{ T(i) + \Theta(n) : i = \lceil n/2 \rceil .. n-1 \}$$

$$= T(n-1) + \Theta(n)$$

$$= \Theta(n^2)$$

QuickSelect Running Time



WLOG Assume: $|S_=| = 1$. If it's larger, it can only help!

$T(n) = \max\{T(i)$, (1), for n=0,1.

Best-Case:

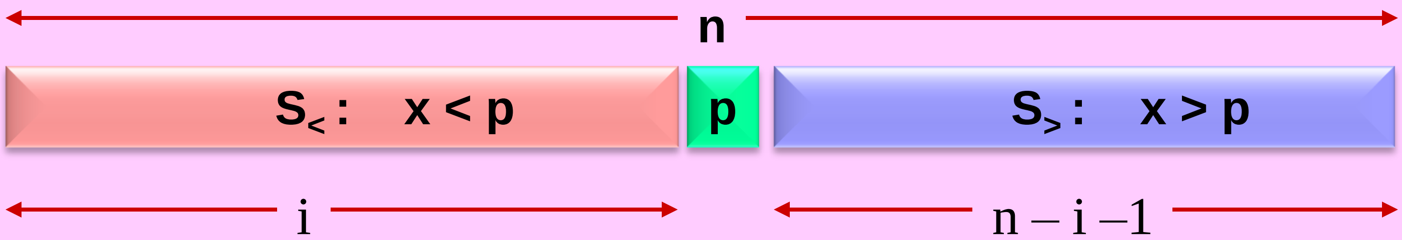
$T(n) = \min_i \{ \max\{T(i), T(n\text{-}i \quad : \quad i = 0 \,..\, n-1 \}$

$= \min_i \{ T(i) + \Theta(n) \; : \; i = \lceil n/2 \rceil \,..\, n-1 \}$

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

$= \Theta(n)$

QuickSelect Running Time



WLOG Assume: $|S_=| = 1$. If it's larger, it can only help!

T(n) = max{T(i) , ... $\Theta(1)$, for n=0,1.

Expected-Cas

$$T(n) = \text{ave}_i \{ \max\{T(i), T(n-i ... : i = 0 .. n-1 \}$$

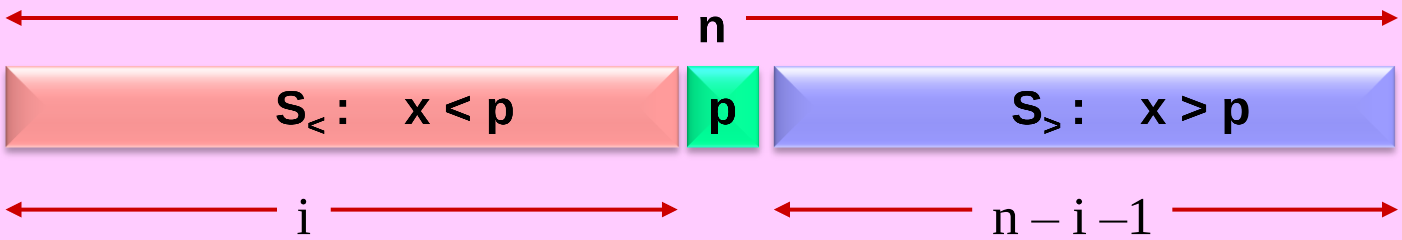
$$= \frac{1}{n}\left(\sum_{i=0}^{\lfloor n/2 \rfloor} T(n-i- ... \sum_{i=1+\lfloor n/2 \rfloor} ... \right)$$

differs from QuickSort

$$= \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) + \Theta(n)$$

$$= \Theta(n)$$ (use guess-&-verify method)

QuickSelect Running Time



Wors 2

Expe

Best-Case: T(n

The following question persisted for a while:

Is Selection as hard as Sorting in the worst case?

Does finding the median require Ω(n log n) time in the worst case?

But then came along the next development ☺ …………….. P.T.O.

Solution 5: Improved QuickSelect

$S_<$: $x < p$	$S_=$: $x = p$	$S_>$: $x > p$

$T(|S|) = \max\{ T(|S_<|), T(|S_>|) \} + \Theta(|S|)$, $T(n) = \Theta(1)$, for n=0,1.

- Ensure that neither of the two potential recursive calls is too large. How?
- Pick the pivot more carefully rather than completely at random. How?
- If the pivot is the median element, then the recursive call has size at most n/2.

  But that's like hitting th
- Make sure pivot is no way from median. How?

- Use a deterministic sampling technique:

  Let's say, a 20% sample (i.e., one out of e
  That's n/5 element sample set.
  Let the pivot be median of this sample set, found recursively in $T(n/5)$ time.

$(\alpha n) + T(\beta n) + \Theta(n)$

$\alpha + \beta > 1$.

how make $\alpha + \beta < 1$?

s) give us $T(n) = \Theta(n)$!

P.T.O. ☺

- How close is this pivot to the true median of all the n elements?
- Its ranking can vary from 10% to 90% (depending on the chosen sample). Why?
- So, the larger recursive call can be as bad as $T(9n/10)$.
- $T(n) = T(9n/10) + T(n/5) + \Theta(n)$ gives $T(n) = \Theta(n^{1+\varepsilon}) = \omega(n \log n)$.
- Any hope with this strategy?

Solution 5: Improved QuickSelect

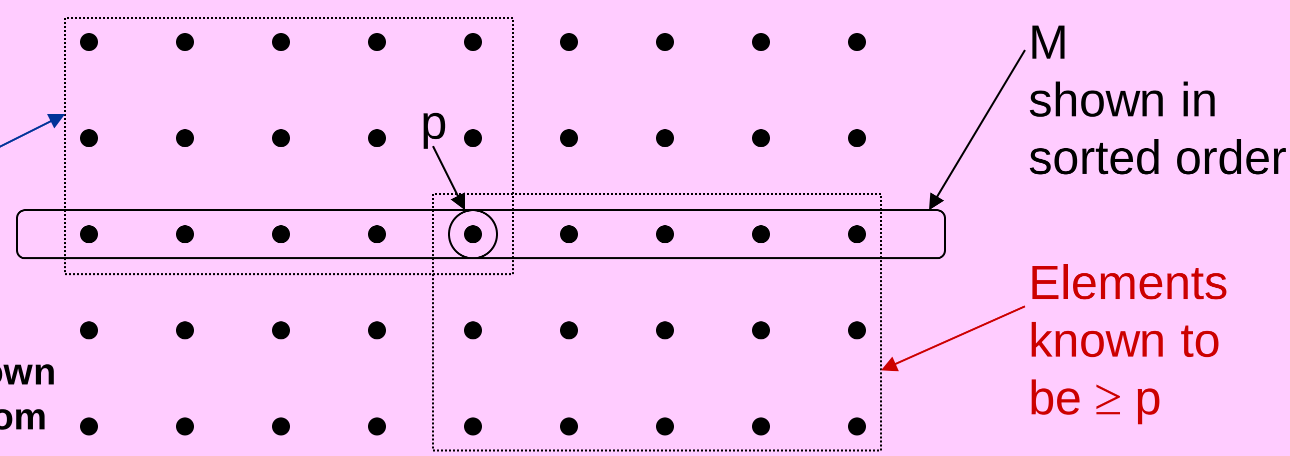
- Pick the 20% sample itself more carefully. How?
- Scan through the n elements and pick one out of each 5 elements.
- Consider one of these 5 element groups.
  If you were to pick one out of this 5, which one would you pick?
  Pick median of this 5 element group in O(1) time.
- So, in O(n) time we can pick our sample set M of n/5 group medians.
- Now recursively Select pivot p to be the median of the sample set M.

$S_<$: x < p **$S_>$: x > p**

CLAIM: With this pivot p, we have max{ $|S_<|$, $|S_>|$ } ≤ 3n/4.

Proof:

Elements known to be ≤ p



Each 5 item group shown as a column, sorted from top to bottom.

Solution 5: Improved QuickSelect

- Pick the 20% sample itself more carefully. How?
- Scan through the n elements and pick one out of each 5 elements.
- Consider one of these 5 element groups.

  If you were to pick one out of this 5, which one would you pick?

  Pick median of this 5 element group in O(1) time.
- So, in O(n) time we can pick our sample set M of n/5 group medians.
- Now recursively Select pivot p to be the median of the sample set M.

$S_<$: x < p **$S_>$: x > p**

CLAIM: With this pivot p, we have [illegible] $_>$| } ≤ 3n/4.

We have improved 90% to 75%, that is, 9n/10 to 3n/4.
The new recurrence is:

75% + 20% = 95% < 100%

Why **5** is chosen as group size?

$$T(n) = T(3n/4) + T(n/5) + \Theta(n)$$

Which gives the solution $T(n) = \Theta(n)$.

The complete algorithm is shown on the next slide.

Solution 5: Improved QuickSelect

[Blum, Floyd, Pratt, Rivest, Tarjan, 1972]

Algorithm Select(S , K) § O(n) worst-case time

§ Assume $1 \le K \le |S|$. Returns the K^{th} smallest element of S.

$\Theta(1)$ { **if** $|S| < 100$ **then** sort S; **return** K^{th} element in sorted sequence S

$\Theta(n)$ {
$g \leftarrow \lfloor |S|/5 \rfloor$ § ------------ sample size
divide S into g groups $M_1, M_2, \ldots, M_g$ of 5 elements each (some leftover)
for $t \leftarrow 1 .. g$
$M \leftarrow \{m_1, m_2,$ … sample set of size g

$T(n/5)$ { $p \leftarrow$ Select(M , $\lceil g/2 \rceil$) § ------ median of sample set M

$\Theta(n)$ { Partition S: $S_< \leftarrow \{ x \in S : x < p \}$
$S_> \leftarrow \{ x \in S : x > p \}$

$T(3n/4)$ {
if $|S_<| \ge K$ **then return** Select ($S_<$, K)
else if $|S| - |S_>| \ge K$ **then return** p
else return Select($S_>$, $K - |S| + |S_>|$)

end

Upper Bound

L nd

Techniques

Upper Bound & Lower Bound Techniques

Upper Bound Methods: Algorithm Design Methods (so far):

Iteration
Recursion
Incremental
Iterative or Recursive Doubling

Lower Bound Methods:

Information Theory
Decision Tree
Algebraic Computation Tree
Problem Reduction
Adversary Argument

Prune-&-Search Method

- Examples:
  Binary Search
  Linear Time Selection algorithm.
  More appear in Exercises at the end of this slide & later in the course….

- Search: Perform just enough computation to detect at least a certain fraction of the input data as irrelevant whose removal will not affect the outcome.

- Prune: remove thes nsideration.

- Recur: repeat the process on the remai .

- Time: With each iteration, # of remaining data items is shrinking geometrically. So, the time for the first iteration dominates the entire process.

- When: This method can be applied to problems with small output, e.g., a single data item like the K^{th} smallest, rather than an elaborate output structure.

Reduction Lower Bound Technique

Algorithm for "old" problem A

In_A → input transform → In_B → Any Algorithm for "new" → Out_B → output transform → Out_A

If the input & output transforms can be done "fast enough" so that they are not the bottleneck, then

a lower-bound on time complexity of the "old" problem A
implies
a lower-bound on time complexity of the "new" problem B.

Example Reduction

A) Element Uniqueness: Are any two elements of input sequence $\langle x_1, x_2, \cdots, x_n \rangle$ equal?

We already know this has Ω(n log n) lower bound in the Algebraic Computation Tree Model.

B) Minimum Gap: Are there any two elements of the input sequence $\langle x_1, x_2, \cdots, x_n \rangle$ whose absolute value difference is $\leq$ a given number D?

We can “reduce” problem A t

The argument for this reduction is quite si

if we could solve B in less than Ω(n log n) t t-case,
then we could also solve A in less than Ω(n log n) time in the worst-case
by simply calling the “fastest” algorithm for B with input parameter D = 0.

Therefore, The Minimum Gap Problem also has the worst case time lower bound Ω(n log n).

Reduction Lower Bound Technique

Algorithm for "old" problem A

In_A → input transform → In_B → Any Algorithm for "new" → Out_B → output transform → Out_A

Example:
Problem A: Element Uniqueness.
Problem B: Closest Pair of points in the plane.
Lower Bound: Ω(n log n)

Selection Lower Bound by Adversary Argument

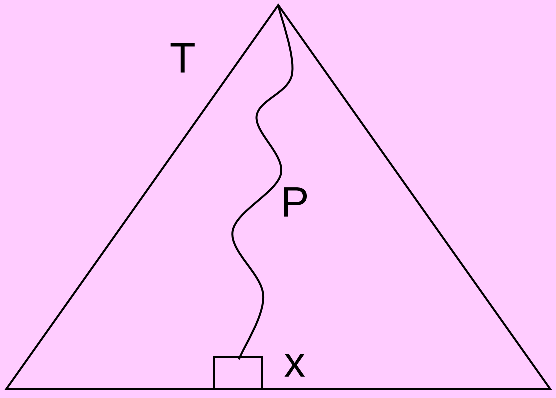
FACT: **Selection always needs $\geq$ n-1 comparisons in the decision tree model.**

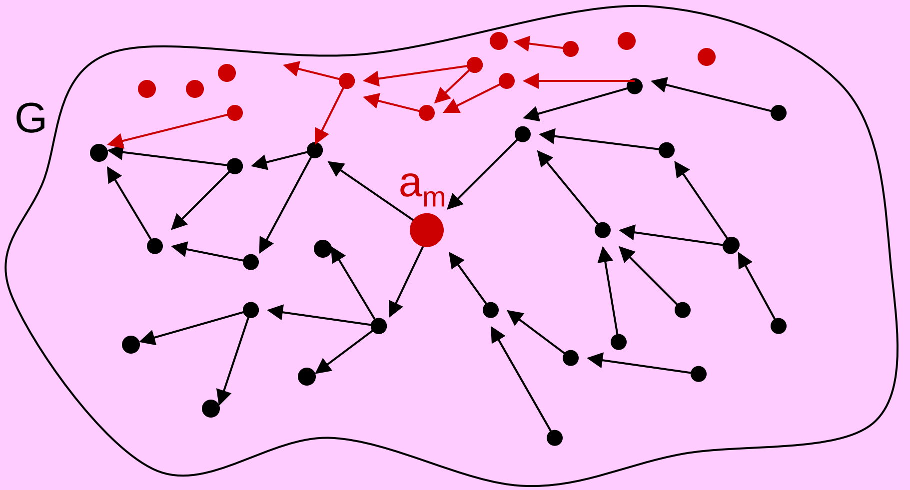
Proof: we want to find the K^{th} smallest item in sequence $S = \langle a_1, a_2, \cdots, a_n \rangle$.

1. Let T be a correct decision tree for that task.
   Assume elements of S are all distinct. T must work correctly on such S.
2. **CLAIM:** every leaf in T must have depth at least n-1.
3. Let X be a leaf in T that declares item a_m is the K^{th} smallest element of S.

4. Let P be the ancestral path of X in T.
5. We have to show tha have been made along P.

6. We will show this by with vertex set S,
   where comparisons along P appear as e
   Then argue that G must have at least n-





Selection Lower Bound

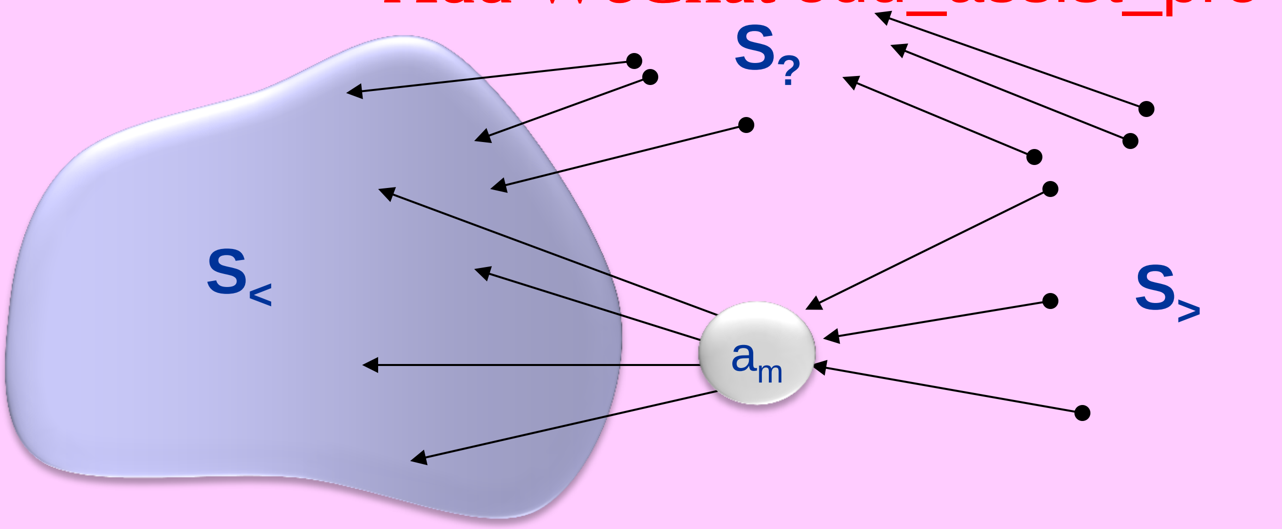
7. Construct the directed graph G = (S, E) as follows:

   $E = \{ (a_i , a_j) \mid \exists$ comparison on path P that declares $a_j < a_i$, or $a_j \leq a_i \}$.
8. Clearly G is acyclic (contains no directed cycles) since S has distinct elements.
9. Remove from E every edge that is implied by transitivity.
10. Define:

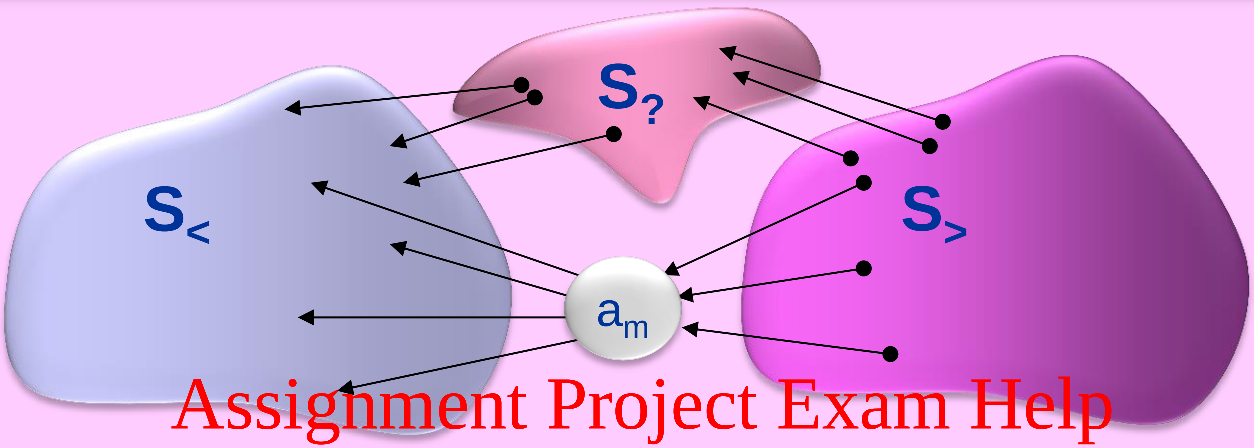
    $S_< = \{ a_i \mid$ there is a directed path of length at least one in G from a_m to $a_i \}$.

    $S_> = \{ a_i \mid$ there is a di e in G from a_i to $a_m \}$.

    $S_? = S - \{a_m\} - S_< -$
11. These 3 subsets are m of them.



Selection Lower Bound



12. **CLAIM:** $S_? = \varnothing$. He …

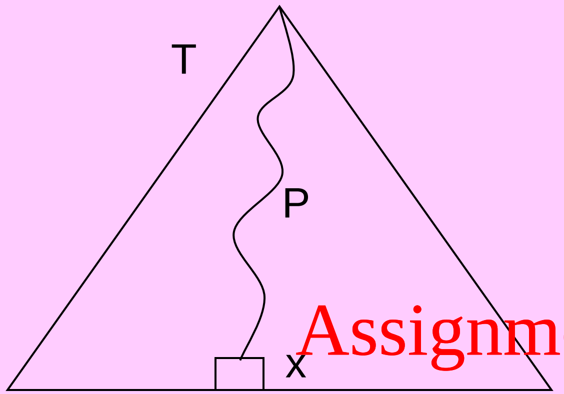
Suppose $S_? \neq \varnothing$. Consider an element … nimizes $\Delta = | a_i - a_m |$.

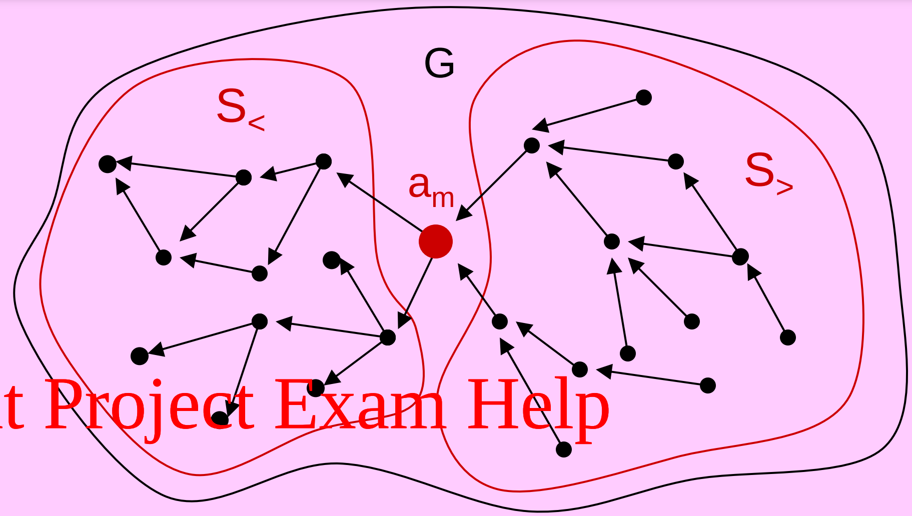
The adversary can shift the value of a_i to … $+ \varepsilon$ (for a very small positive real ε). The relative ordering of every element in S has remained the same, except that now a_i has moved to the opposite side of a_m.

All comparisons in T would still yield the same result, decisions would follow path P, and we would end up at leaf X, declaring that a_m is K^{th} smallest in S.

This can't be correct both times. Why?

Selection Lower Bound





13. Since $S_< = \{ a_i \mid$ t … 1 in G from a_m to $a_i \}$, every vertex in $S_<$ must have i… st 1.
14. Since $S_> = \{ a_i \mid$ there is a path of length at least 1 in G from a_i to $a_m \}$, every vertex in $S_>$ must have out-degree at least 1.
15. These imply that there must be at least $|S_<| + |S_>| = n-1$ edges in G.
16. Therefore, depth of X in T is at least n-1.

This completes the proof.

Median finding: improved lower bound

- The best algorithm uses slightly less than 2.95n comparisons in the worst case.
- The best lower bound is slightly more than 2n comparisons. (See Bibliography.)
- Here we prove a weaker lower bound: 1.5(n – 1).
- Call a comparison between x and y **crucial**

  if either (x < y and y ≤ median) or (x > y and y ≥ median).

- We showed that any sel n –1 crucial comparisons.
- We now give an advers edian finding algorithm

  must also make at least (n –1)/2 non-crucial The idea is:

  - first, the adversary chooses some valu tem) to be the median.
  - Then it assigns a label {N, L, S} to each item, and also values, as appropriate.

    N = never participated in any comparison

    L = larger than the chosen median value

    S = smaller than the chosen median value
  - Initially each item is assigned an “N” label.

The Adversary's Strategy

Algorithm compares	Adversary responds
(N,N)	assign one item to be larger than the median, one smaller; result is (L,S)
(S,N) or (N,S)	a [illegible] an the median; result is (S,L) or (L, [illegible]
(L,N) or (N,L)	assign the N-item to be smaller than the median; result is (L,S) or (S,L)
(S,L) or (L,S) or (S,S) or (L,L)	consistent with previously assigned values

Count Comparisons

- This strategy continues until (n-1)/2 S's (or (n-1)/2 L's) have been assigned.
- If at some point (n-1)/2 S's are assigned, then the adversary assigns the remaining items to be greater than the median, except for one, which IS the median.
- A similar thing is done if (n-1)/2 L's have been assigned.
- In any case, the last item assigned is always the median.

- **CLAIM:** This strategy al rform

  at least (n-1)/2

Proof: Any time an N-item is compared, a n parison is done (except at the very end, when a crucial comparison ith the median itself).

The least number of comparisons with N-items that can be done is (n-1)/2.

- **The total number of comparisons is therefore at least**

$$\mathbf{n - 1 + (n - 1)/2 = 1.5(n - 1).}$$

Bibliography

- C.A.R. Hoare, *“Quicksort,”* Computer Journal, 5(1):10-15, 1962.

- J. W. J. Williams, *“Algorithm 232 (HEAPSORT),”* Communications of the ACM, 7:347-348, 1964.

- Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, Robert E. Tarjan, *“Time Bounds for Selection,”* Journal of Computer and System Sciences, 7(4):448-461, 1973.

- Dorit Dor, Uri Zwick, *“Medi* *parisons,”* SIAM Journal on Discrete Mathematics, 14(

Assignment Project Exam Help

s

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

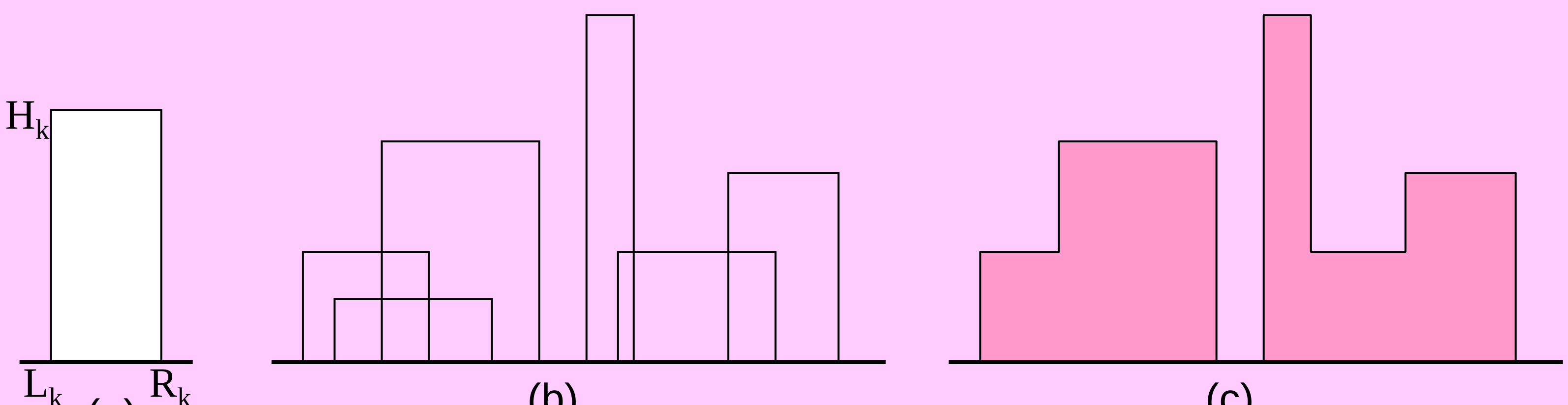
1. **The SkyLine Problem:**
Given the exact locations and shapes of n rectangular buildings, in arbitrary order in a city, compute the skyline (in two dimensions) of these buildings, eliminating hidden lines.
An example of a building is shown in Fig (a) below; an input is shown in Fig (b); and its corresponding output is shown in Fig (c). We assume the bottoms of all the buildings lie on a horizontal line (the horizon).
Building B(k) is represented by a triple of real numbers (L_k , H_k, R_k). See Fig (a).
L_k and R_k denote the left and right x coordinates of the building, respectively, and H_k denotes the height of the building.
A **skyline** is a list of (alternating) x coordinates and heights connecting them arranged in order from left to right. For instance, the buildings in Fig (b) correspond to the following input:

(1, **6,** 5), (2, **3,** 7), (3, **9,** 9), (11, **13,** 13), (12, **5,** 20), (16, **7,** 22).

The skyline in Fig (c) is represented by the sequence: (1, **6,** 3, **9,** 9, **0,** 11, **13,** 13, **5,** 16, **7,** 22, **0**).
[Note: all **bold red** number

a) Design and analyze a lgorithm to solve the Skyline Problem.

b) Now suppose the input list of buildings is giv der of their left x coordinate, i.e., $L_1 \leq L_2 \leq L_3 \leq \ldots \leq L_n$. Can the skyline faster in this case? Explain.



(a) (b) (c)

2. **k-way Merge:** Give an O(n log k) time algorithm to merge k sorted lists of total size n into one sorted list of size n. [Hint: use a min heap for k-way merging.]

3. **Iterative MergeSort:** We described an implementation of MergeSort by recursive divide-&-conquer. MeregeSort can also be implemented iteratively. The idea is to initially consider each of the n input elements as sorted lists of size 1. Then in a round-robin fashion merge these sorted lists in pairs into larger size but fewer lists until there is only one sorted list remaining. That is the output sorted list. Give an iterative implementation of this version of MergeSort by keeping the sorted lists in a queue. Analyze the worst-case time complexity of your implementation.

4. **Median of 2 sorted arrays:** Let X[1..n] & Y[1..n] be two n-element sorted arrays. Give an O(log n) worst-case time algorithm to find the median of the 2n elements that appear in X and Y. [Hint: use divide-&-conque

5. **Stack depth for QuickSor** ed contains two recursive calls. Compilers usually execute recursive procedures by contains pertinent information (that includes the parameter values) called ***stack fr*** rsive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its stack frame is pushed onto the stack; when it terminates, its stack frame is popped. Let's assume we want to sort array A[1..n]. The stack frame corresponding to a subarray to be sorted is the pair of extreme indices of that subarray, and takes O(1) memory cell space. The ***stack depth*** is the maximum number of stack frames on the stack at any time during the computation.
   a) Describe a scenario in which the stack depth for QuickSort, as described in the course, is $\Theta(n)$.
   b) Modify the code for QuickSort so that the worst-case stack depth is $\Theta(\log n)$. You must maintain the O(n log n) expected running time of the algorithm. [Hint: decide which of the 2 recursive calls to invoke first.]

6. Searching and Selection in a partially sorted matrix:

We are given a matrix A[1..n, 1..n] of n×n real numbers such that elements in each row of A appear in non-decreasing sorted order, and elements in each column of A also appear in non-decreasing sorted order. (This is a specific partial ordering of the n^2 matrix elements.) Design and analyze efficient algorithms for the following:

a) Find the number of negative entries in matrix A. [O(n) time is possible.]

b) Search in A for a given real number x: Find the maximum matrix entry A[i,j] that is less than or equal to x, or report that all elements of A are larger than x. [O(n) time is possible.]

c) Select the K^{th} smallest element of A, where K is a given positive integer $\leq n^2$.
[We know $O(n^2)$ time is achievable even without the partial ordering. Any faster here?]

7. Weighted Median:

For n distinct elements $x_1, x_2, \ldots, x_n$ with positiv , …, w_n such that $w_1 + w_2 + \ldots + w_n = 1$, the weighted (lower) medi x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2} \quad \text{and} \quad \sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

a) Argue that the median of $x_1, x_2, \ldots, x_n$ is their weighted median with $w_i = 1/n$ for i = 1..n.

b) Show how to compute the weighted median of n elements in O(n log n) worst-case time using sorting.

c) Show how to compute the weighted median of n elements in O(n) expected time similar to QuickSelect.

d) Show how to compute the weighted median in O(n) worst-case time using a linear time (un-weighted) median finding algorithm such as the one we discussed in the course.
[Hint: use prune-&-search.]

8. **Heap Delete:** The heap operation Delete(A,t) deletes the item in node t from heap A. Give an implementation of this operation that runs in O(log n) time on a max heap of size n.

9. **d-ary Heap:** A d-ary heap is like a binary heap, but non-leaf nodes have d children instead of 2.
    a) How would you represent a d-ary heap in an array?
    b) What is the height of a d-ary heap with n elements in terms of n and d?
    c) Give an efficient implementation of DeleteMax in a d-ary max-heap. Analyze its worst-case running time in terms of d and n.
    d) Give an efficient implementation of Insert in a d-ary max-heap. Analyze its worst-case running time in terms of d and n.

    e) Give an efficient implementation of IncreaseKey(A, t, K), which first sets A[t] $\leftarrow$max{A[t], K structure appropriately. Analyze its worst-ca

10. **Sorting grid points:**

Given n points on the n-by-n integer grid in the pl t these points by their distance from the origin. What is the worst-case time complexity of your algorithm?
[Hint: use RadixSort.]

11. **Small decision tree:**

    a) Show that every comparison based (i.e., decision tree) algorithm that sorts 5 elements, makes at least 7 comparisons in the worst-case.

    b) Give a comparison based (i.e., decision tree) algorithm that sorts 5 elements using at most 7 comparisons in the worst case.

12. Average Gap:

We are given an arbitrary (unsorted) array A[1..n] of n >1 distinct real numbers.
The k^{th} gap of A, denoted $G_k(A)$, is the difference between the $(k+1)^{st}$ smallest and k^{th} smallest elements of A, for k =1..n-1.
The **minimum gap** of A, denoted $G_{min}(A)$, is the minimum $G_k(A)$ over all k=1..n-1.
The **average gap** of A, denoted $G_{ave}(A)$, is the average of $G_k(A)$ over all k=1..n-1.
That is, $G_{ave}(A) = (G_1(A) + G_2(A) + \dots + G_{n-1}(A)) / (n-1)$.
We clearly see that $G_{min}(A) \le G_{ave}(A)$.

a) Describe an efficient algorithm to compute $G_{min}(A)$. What is the running time?
b) Show that $G_{ave}(A) = (\max(A) - \min(A)) / (n-1)$.
c) Use part (b) to show that $G_{ave}(A)$ can be computed in O(n) time.
d) Design and analyze of elements (A[i],A[j]), i ≠ j, of $| \le G_{ave}(A)$.
(The answer may no
[Hint: use median selection and prune-&-se

13. Merging Lower Bound:

We discussed the information theory lower bound on merging two sorted lists, each containing n elements, and showed the lower bound of ≈ 2n – ½ log n.

a) A tighter decision tree lower bound can be achieved as follows:
Using an adversary argument, show that if two elements are consecutive in the output sorted order and are from opposite lists, then they must be compared.

b) Use your answer to part (a) to show a lower bound of 2n –1 comparisons.

14. The decision tree model for comparison-based sorting of a partially ordered set:

In this problem, we consider sorting n distinct keys which are already partially sorted as described below. The only way we are allowed to sort the keys is by performing a comparison between a pair of keys, where the only information obtained from a comparison is which of the two keys is larger. (Thus, we are not allowed to look at the values of the keys.)

Let the given keys be $x_1, x_2, \ldots, x_n$. Let $A = \{x_1, x_2, x_3\}$, and $B = \{x_4, x_5, \ldots, x_n\}$. Suppose that A has been completely sorted (and without loss of generality assume $x_1 < x_2 < x_3$), and that B has been completely sorted (and without loss of generality assume $x_4 < x_5 < \ldots < x_n$), but there have not been any comparisons made between any key in A and any key in B.

a) Exactly how many possible orderings among all n keys are still possible given the information above, i.e. [illegible] $x_2, x_3\}$ as a triple fit into the ordering among the n- [illegible] t positions)?

b) From part (a), derive a [illegible] c) on the remaining number of comparisons necessary to completely sort all n [illegible] gument you use to derive your lower bound.

c) From part (b), what is the lower bound on the number of comparisons to complete the sorting when n is 6? Give your lower bound as a positive integer.

d) Give a decision tree for the case when n is 6 for which the worst case number of comparisons is exactly the lower bound on the number of comparisons given in part (c).

15. [CLRS Problem 8-4, pp. 206-207,] Red-Blue Water Jugs:

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

It is your task to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This will tell you whether the red or the blue jug holds more water, or if they are of the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a) Describe a determini sons to group the jugs into pairs.

b) Prove a lower bound risons an algorithm solving this problem must take.

c) Give a randomized algorithm whose expecte arisons is O(n log n), and prove that this bound is correct. What is the of comparisons for your algorithm?

16. Show that the 2nd smallest of n given elements can be found with $n + \lceil \log n \rceil - 2$ comparisons in the worst-case. [Hint: Also find the smallest element. Play elimination tournament among elements.]

17. Show that $\lceil 3n/2 \rceil - 2$ comparisons are necessary and sufficient in the worst case to find both the maximum and the minimum of n elements. [Hint: Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts. Use labels similar to the ones we used for the median finding adversarial lower bound argument.]

18. Show how the worst-case running time of QuickSort can be improved to O(n log n) using selection.

19. In the comparison based model, show that any selection algorithm that finds the K^{th} smallest element, can also find the er elements without any additional comparisons.

20. Suppose that you have a “black-box” worst-case li finding subroutine. Give a simple, linear-time algorithm that solves the select bitrary order statistics K using the black-box as a subroutine.

21. The K^{th} quantiles of an n-element sequence are the K-1 order statistics that divide the sorted permutation of the sequence into K equal-sized (to within ±1) contiguous subsequences. Give an O(n log K) time algorithm to list the K^{th} quantiles of a given unsorted sequence of n items.

22. Describe an O(n) time algorithm that, given a set S of n distinct numbers and a positive integer $K \le n$, determines the K numbers in S that are **closest** in value to the median of S.
Example:
Let S = { 9,2,7,3,8,1,12 }. Median(S) = 7, and the 3 items with closest values to 7 are {7,8,9}.

23. The facility Location Problem (FLP): We are given a set $P = \{p_1 , p_2 , \ldots, p_n\}$ of n points in the d dimensional space. Each point is given by its d coordinates as real numbers. We want to compute the location of the optimum facility point q whose sum of distances to the input points is minimized. That is, find a point q that minimizes

$$d(q, p_1) + d(q, p_2) + d(q, p_3) + \ldots + d(q, p_n),$$

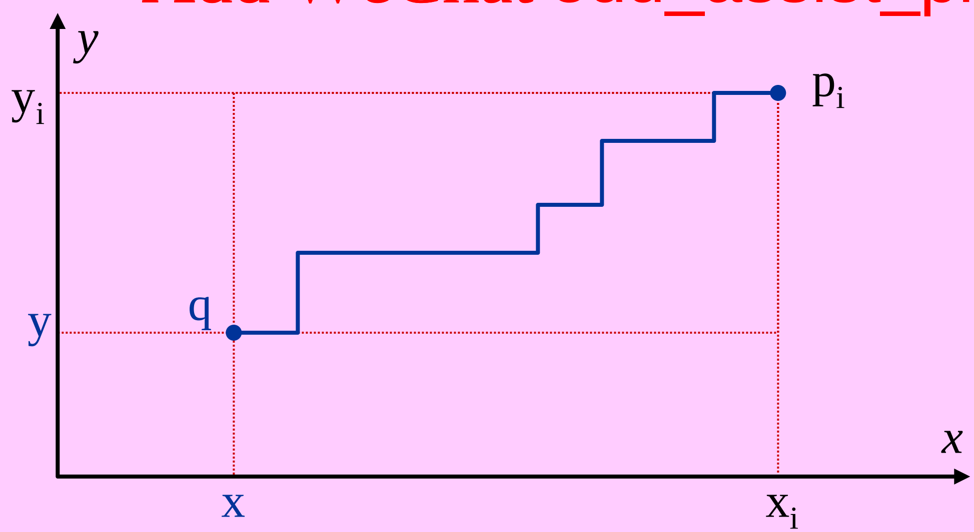
where $d(q, p_i)$ denotes the distance between q and p_i .

Imagine we want to establish a communication center (the facility location q) and run lines from that center to each of the users (the given points). The objective is to minimize the total length of the communication lines between the center and the users.

a) Consider the one dimensional version of FLP (d = 1). How do you characterize the solution point? Show one dimensional FLP can be solved in O(n) time.

b) How would you solve the problem for d = 2 (the planar case) assuming the communication lines are only allowe e segments with bends in between. (Suppose th anhattan should follow along the streets which are anhattan metric, the distance between points $q = (x,y)$ and $p_i = (x_i , y_i)$ is $_i \mid + \mid y - y_i \mid$.

(See the illustrative figure below.)



24. The Majority Problem (MP): We are given a sequence S of n elements. A majority value, if it exists, is one that appears more than n/2 times in the input sequence. The problem is to determine if S has a majority element, and if so find it.

a) Suppose we are allowed to compare pairs of elements of S using the comparisons from the set { = , ≠ , < , ≤ , > , ≥ }. Within this model we can solve MP in O(n log n) worst-case time by first sorting S. Describe the rest of the process.

b) Within the same comparison based model as in part (a), show the problem can be solved in O(n) worst-case time using Median Selection.

c) Now suppose the only comparisons we are allowed to make are from the set { = , ≠}. So, we cannot sort. Show how to solve MP in worst-case O(n log n) time in this model using divide-&-conquer.

d) Within the same co w MP can be solved in O(n) worst-case time. [Hi l chip testing problem in LS4.]

e) Here's another divide-and-conquer approach .
First assume $n = 2^k$ where k is a non-negativ
Pair up elements of A arbitrarily, to get n/2 p pair: If the two elements are different, discard both of them; if they are the same, keep just one of them.
Show that after this procedure there are at most n/2 elements left, and that they have a majority element if A does. Turn this idea into an O(n) time algorithm that finds the majority element of A if there is one.

If n is not an integer power of 2, then we have the following:
Counter-example: (1,1, 1,1, 2,2, 2) has majority → (1,1,2,2) has no majority.

Revise your algorithm so that it works correctly for all n.

25. **A Loading Problem:** We are given a set S of n items with weights specified by positive real numbers w_i, i=1..n, and a truck with load weight capacity specified by a positive real number L. We want to load the truck with as many items as possible without exceeding its load capacity. That is, we want to find a maximum cardinality subset $C \subseteq S$ such that the sum of the item weights in C does not exceed the truck load capacity L.

(a) Briefly describe an O(n) time algorithm to solve this problem if S is given in sorted order.

(b) Describe an O(n) time algorithm to solve this problem if S is given in arbitrary unsorted order. [Hint: use median selection and prune-&-search.]

26. **Significant Inversions:** We are given a sequence of n arbitrary but distinct real numbers $\langle a_1, a_2, \cdots, a_n \rangle$. We define a **significant inversion** to be a pair i < j such that $a_i > 2\, a_j$. Design and analyze an O mber of significant inversions in the given sequence.

27. **Lower Bound on the Closest Pair Problem:**
The Closest Pair Problem asks to find the closest ng a given set of n points in the plane. In the previous slide we claimed that the worst-case time complexity of this problem is $\Omega(n \log n)$. Prove this lower bound using the reduction technique.

28. **Lower Bound on BST Construction:**

(a) Given a Binary Search Tree (BST) holding n keys, give an efficient algorithm to print those keys in sorted order. What is the running time of the algorithm?

(b) Within the decision tree model derive a lower bound on the BST construction problem, i.e., given a set of n keys in no particular order, construct a BST that holds those n keys.

29. Lower Bound on Priority Queues:

Is there a priority queue that does both Insert and DeleteMin in at most O(log log n) time? Explain.

30. Lower Bound on Sorting Partially Sorted List:

Let A[1..n] be an array of n elements such that we already know A[1 .. n-100] is sorted and A[n-99 .. n] is also sorted. Establish a worst-case decision tree lower bound for sorting the entire array A[1..n].

31. Time Bound on A Special Sorting Problem:

You are given a sequenc distinct integer values (but we don’t know which intege

a) Design an algorithm to sort this sequence in e.

[Hint: use a balanced Binary Search Tree wh s a distinct key and a pointer to the list of items with value equal to that key.]

b) Why does this run-time not violate the comparison based sorting lower bound of Ω(n log n) comparisons?

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro