CS 590: Algorithms

Lecture 5 – More Sorting Algorithms

Outline

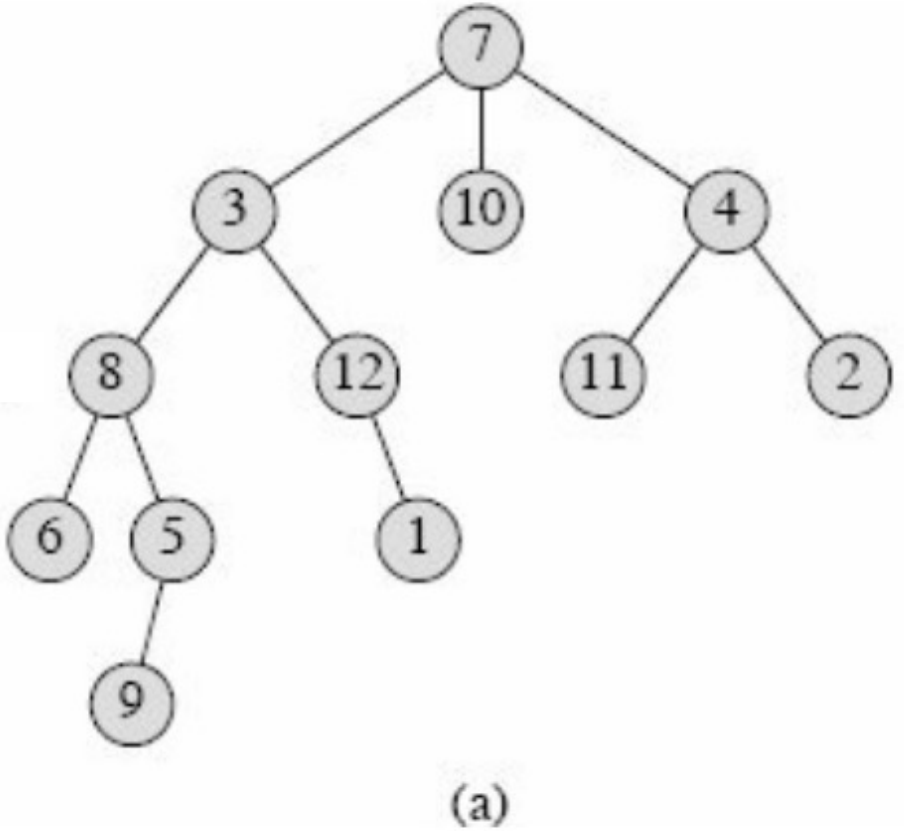
5.1. Heapsort

5.1.1. Tree and Binary Tree

Tree:

- Tree is a graphical way of algorithm.

Rooted tree:
a tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the **root** of the tree. A vertex of a rooted tree as a **node**.

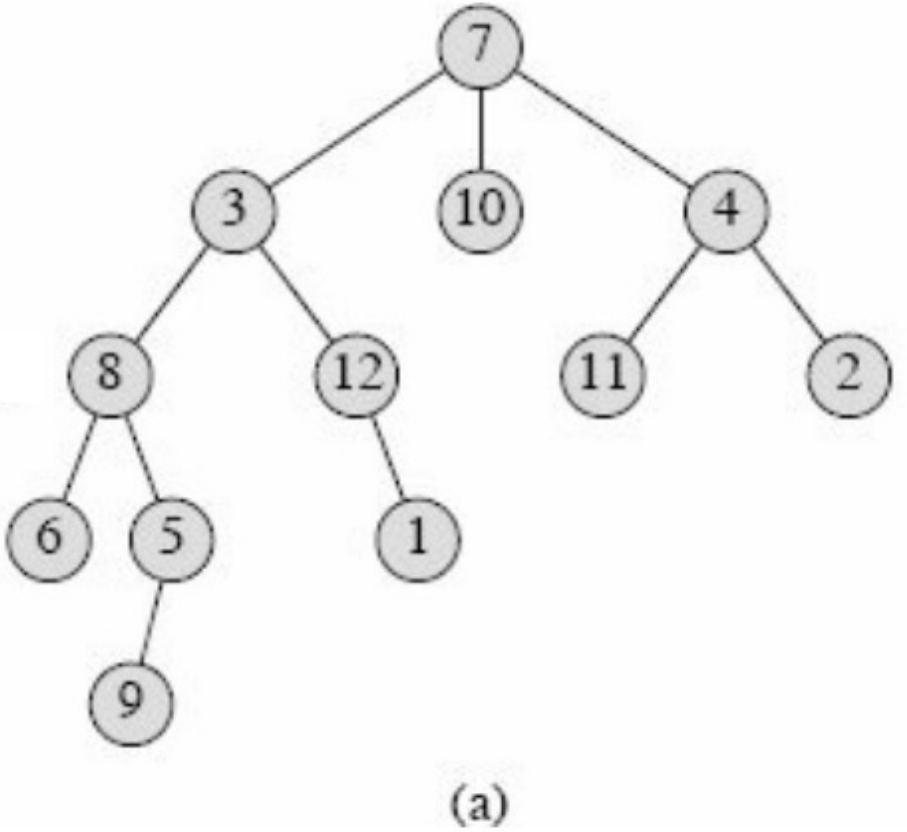


5.1.1. Tree and Binary Tree

Properties:

Consider a node x in a rooted tree T with root r.

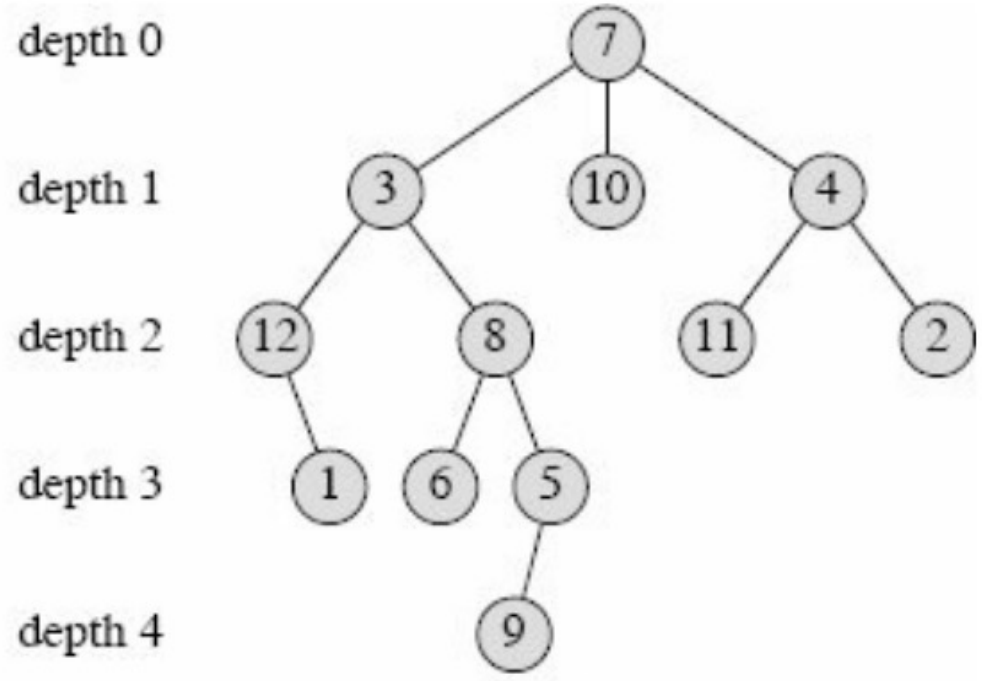
- **Ancestor**: Any node y on the unique simple path from r to x, y is an **ancestor** of x.
- **Descendant**: If y is an ancestor of x, then x is a descendant of y.
- Every node is both an ancestor and a descendant of itself.
- **Subtree** rooted at x: A tree induced by descendant of x.
- In a simple path from r to x is (y, x), y is the **parent** of x, and x is a **child** of y.
- If two nodes have the same parent, they are **siblings**.



5.1.1. Tree and Binary Tree

Characteristics:

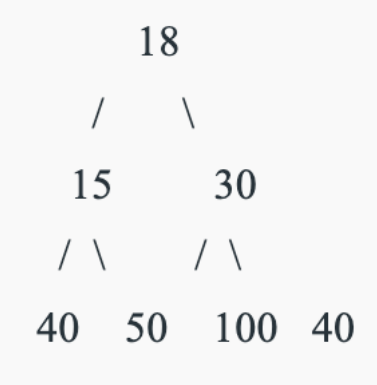
- **Degree** of x: the number of children of a node x.
- **Depth** of x: the length of the simple path.
- **Level**: consists of all nodes at the same depth.
- **Height**: the number of edges on the **longest simple downward path** from the node to a leaf. It is also equal to the largest depth of any node in the tree.

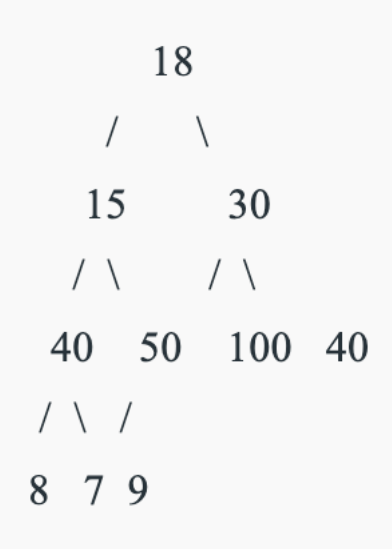


5.1.1. Tree and Binary Tree

A **binary tree** is a structure defined on a finite set of nodes that either

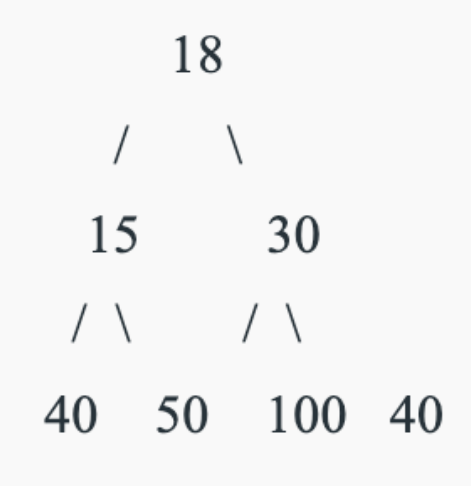
- contains no does or
- composed of three disjoint set of nodes – a root node, left subtree, right subtree.

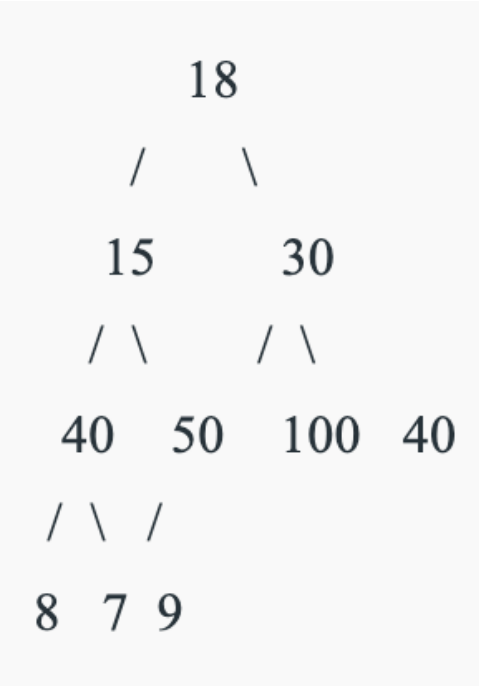




5.1.1. Tree and Binary Tree

- A **binary tree** is not simply an ordered tree in which each node has degree at most of 2.
- **Full binary tree**: each node is either a leaf or has degree of exactly 2. (there is no degree-1 nodes).
- **Perfect binary tree**: a full binary tree of height h with exactly $2^h - 1$ nodes.
- **Complete binary tree**: a perfect binary tree through $h - 1$ with some extra leaf nodes at depth h, all toward the left.



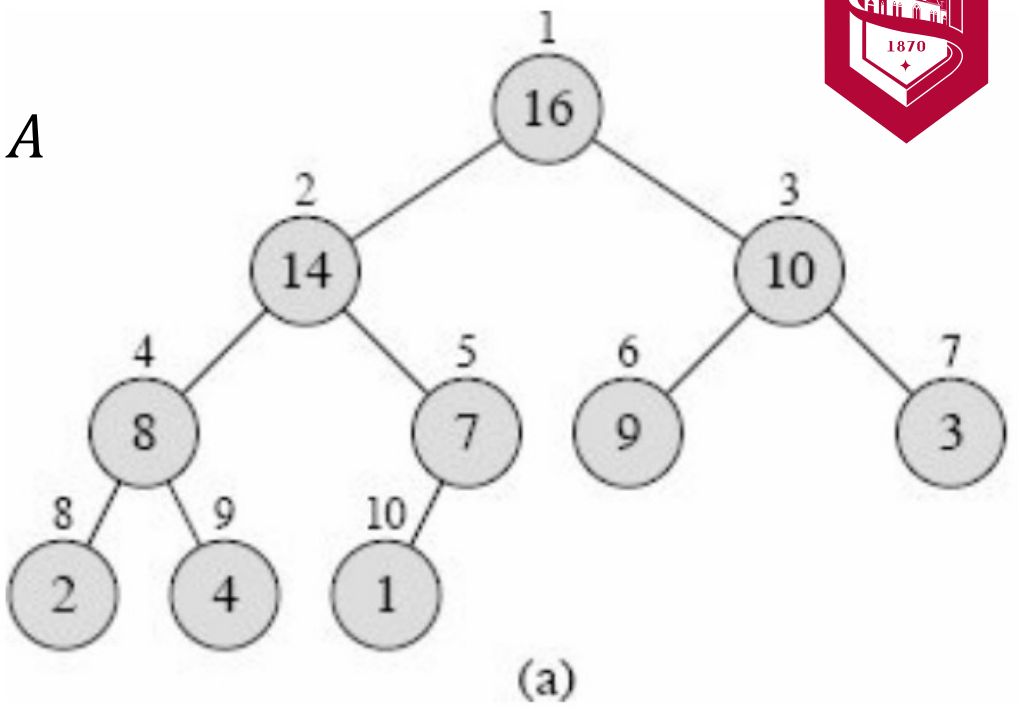


5.1.2. Heap

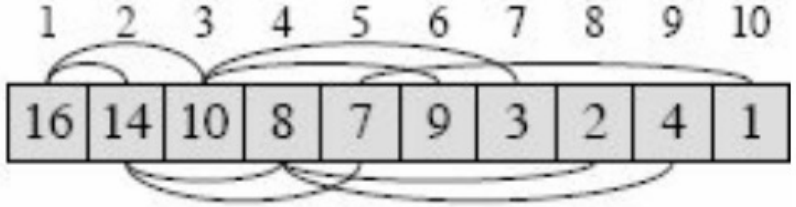
A **heap** (or binary heap) data structure is an array object A that is a complete binary tree with following properties.

- The root of the tree is $A[1]$
- Every subtree is a heap. (e.g., $0 \leq A.heapsize \leq A.length$)

The indices of its parent, left child, and right child are $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right], A[2i], A[2i+1]$, respectively.



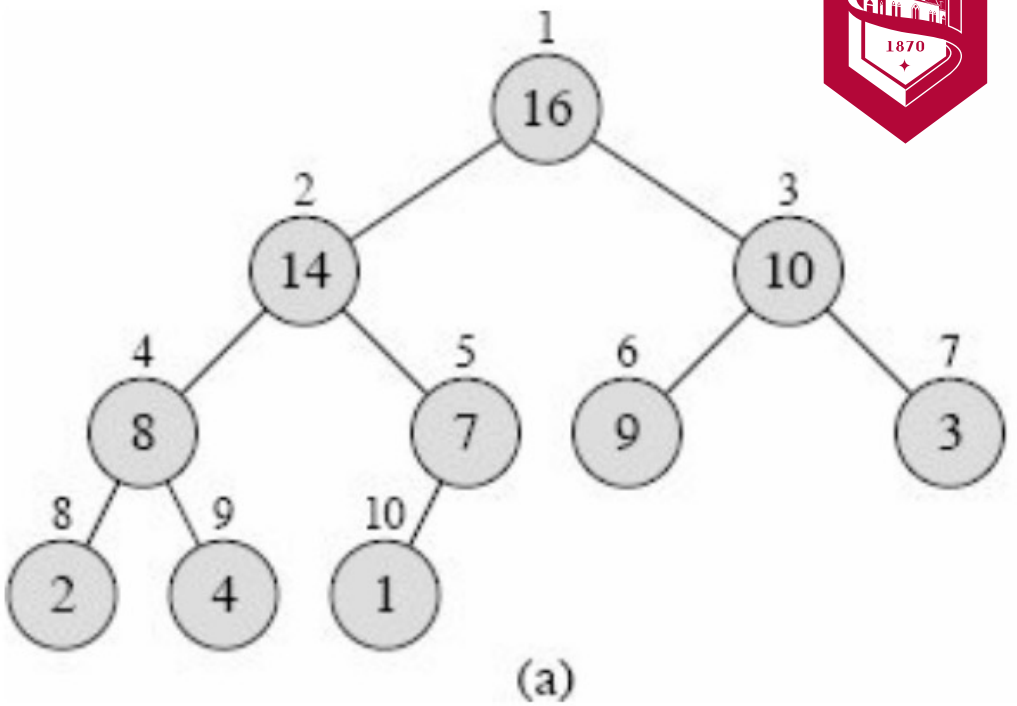
(a)

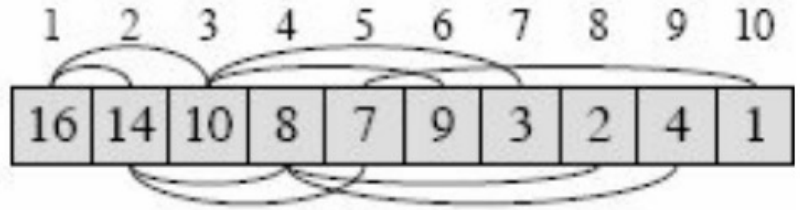


5.1.2. Heap

Two kinds of binary heaps:

- Max-heap:
  - The value of a node is the most the value of its parents, $A[Parent(i)] \geq A[i]$.
  - The largest element in a max-heap is stored at the root.
- Min-heap:
  - Opposite of Max-heap: $A[Parent(i)] \leq A[i]$.
  - The smallest element in a min-heap is at the root.





5.1.2. Heap

The height of heap is the height of its root.

- A heap with n elements based on a complete binary tree runs $\Theta(\lg n)$ time.

Proof:

Given an n-element heap of height k and is an almost-complete binary tree,

- it has at most $2^{k+1} - 1$ elements and at least 2^k elements (if the lowest level has just 1 element and the other levels are complete).
- then $2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$.
- Thus, $k \leq \lg n < k + 1$. Since k is an integer, $k = \lfloor \lg n \rfloor$.

General Algorithmic Procedure of Heapsort:

1. The MAX-HEAPIFY procedure is the key to maintaining the max-heap property.
2. The BUILD-MAX-HEAP procedure produces a max-heap from an unordered input array.
3. The HEAPSORT sorts an array in place.
4. The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM allow the heap data structure to implement a priority queue. (Not going to discuss)

5.1.2. Heap

The MAX-HEAPIFY assumption:

- the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps
- but that $A[i]$ might be smaller than its children and violates the max-heap property.
- It lets have value at $A[i]$ "float down" in the max-heap – so the subtree rooted at index i obeys the max-heap property.

Algorithm (MAX-HEAPIFY(A,i,n))

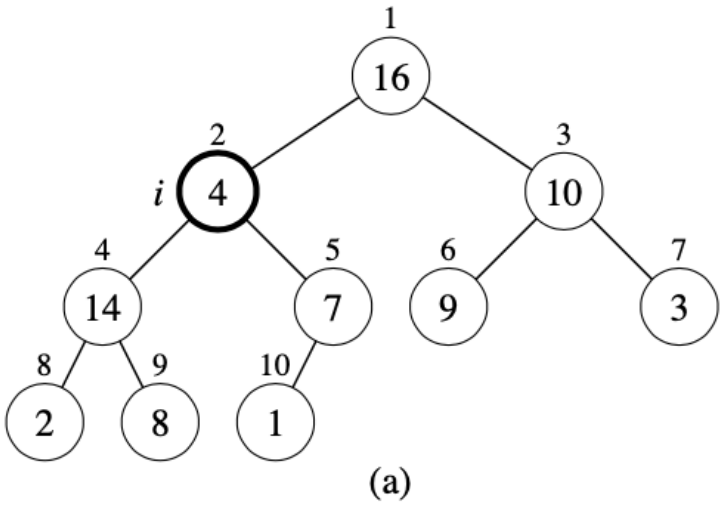
```
(1)  l = LEFT(i)  r = RIGHT(i)
(2)  if (l ≤ n and A[l] > A[i]) then
(3)     largest = l
(4)  else
(5)     largest = i
(6)  if (r ≤ n and A[r] > A[largest]) then
(7)     largest = r
(8)  if (largest ≠ i) then
(9)     swap A[i] and A[largest]
(10)    MAX-HEAPIFY(A, largest, n)
```

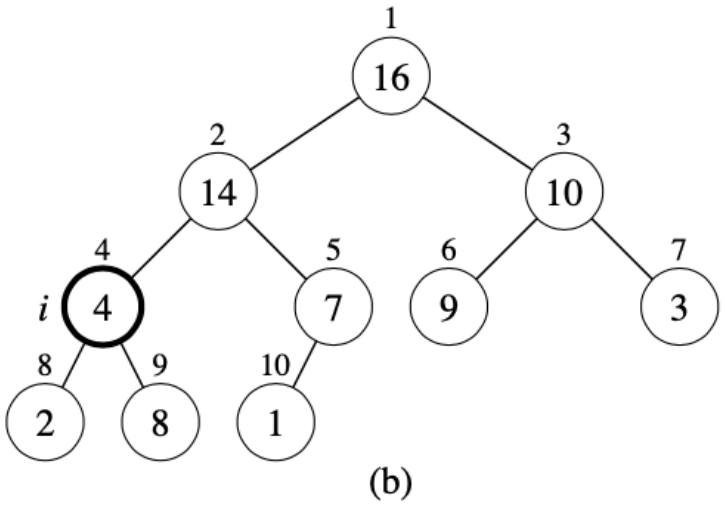
- Compare $A[i], A[LEFT(i)]$, and $A[RIGHT(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

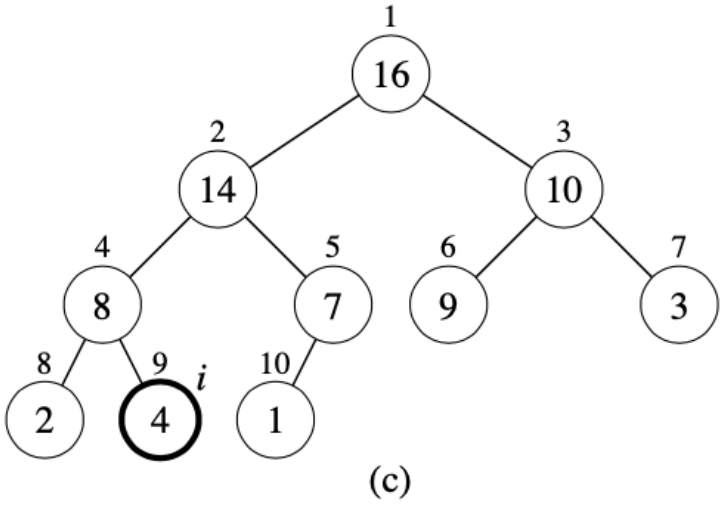
5.1.2. Heap

- Heap is almost-complete binary tree.
- Compares 3 items and maybe swapping 2.
- Fixing up the relationships among the element $A[i], A[LEFT(i)]$, and $A[RIGHT(i)]$ at a given node i takes $\Theta(1)$ time.
- The worst case occurs at the bottom level of the tree.
- The children's subtrees each have size at most $2n/3$. (only half full case)
- The running time $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$.
- Using MT2 ($T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$, we find that $\log_{\frac{3}{2}} 1 = 0$, so $n^0 = 1$.
- With $k = 0$, we can see that $\lg n \geq 1$ if $n \geq 2$. Therefore, we can say that $T(n) = O(\lg n)$.

5.1.2. Heap


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(a)


1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1
(c)

5.1.2. Heap

Building a heap

- Using MAX-HEAPIFY in a *bottom-up* manner to convert an array $A[1 \ldots n]$ into a max-heap.
- $A\left[\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \ldots n\right]$ are all leaves of the tree so each is a 1-element heap to begin with.

Algorithm (BUILD-MAX-HEAP(A,n))

```
(1)  for (⌊n/2⌋ ≥ i ≥ 1) do
(2)      MAX-HEAPIFY(A, i, n)
```

Correctness:

- Initialization – since $i = \left\lfloor \frac{n}{2} \right\rfloor$ before the first iteration of the for loop, the invariant is initially true.
- Maintenance – Decrementing i reestablishes the loop invariant at each iteration.
- Termination – When $i = 0$, the loop terminates.

5.1.2. Heap

Building a heap – Analysis

- **simple bound:** Each call to MAX-HEAPIFY makes $O(\lg n)$ and BUILD-MAX-HEAP makes $O(n)$. Thus, the running time is $O(n \lg n)$.
- **tighter analysis:** Can observe that the running time for MAX-HEAPIFY is linear in the height of the node its running on. Most nodes have small heights.
- We have $\leq \left\lceil \frac{n}{2^{k+1}} \right\rceil$ nodes of height k and height of heap is $\lfloor \lg n \rfloor$.

How?

- The tree leaves (nodes at height 0) are at depth K and $K - 1$.
- They consist of all notes at depth K, and the nodes at depth $K - 1$ that are not parents of depth-K nodes.

5.1.2. Heap

Building a heap - Analysis

- Let x be the number of nodes at depth K in the bottom level. (if n is odd, x is even or vice versa).
- If x is even, there are x/2 nodes at $K-1$ that are parent of K depth nodes.
- Hence, $2^{K-1} - x/2$ nodes at $K-1$ depth are not parents of K depth nodes.
- The total nodes $= x + 2^{K-1} - \frac{x}{2} = \frac{2^K+x}{2} = \left\lceil \frac{(2^K+x-1)}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$.
- If x is odd, using the similar argument, the total nodes $= x + 2^{K-1} - \frac{x+1}{2} = \frac{2^K+x-1}{2} = \left\lceil \frac{n}{2} \right\rceil$.
- Let n_h be the number of nodes at height h in the n-node tree T.
- Considering the tree T' formed after removing the leaves of tree T.
- Then it has $n' = n - n_h$ nodes. Since we know that $n_h = \left\lceil \frac{n}{2} \right\rceil$ from the base case, $n' = \left\lfloor \frac{n}{2} \right\rfloor$.

5.1.2. Heap

Building a heap - Analysis

- Let n'_{h-1} be the number of nodes at height $k-1$ in T'.
- Then $n_h = n'_{h-1}$. We can bound that $n_h = n'_{h-1} \leq \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor n/2 \rfloor}{2^k} \right\rceil \leq \left\lceil \frac{n/2}{2^k} \right\rceil = \left\lceil \frac{n}{2^{k+1}} \right\rceil$.

- The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{k+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \left(\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} \right) \right) = 2$$

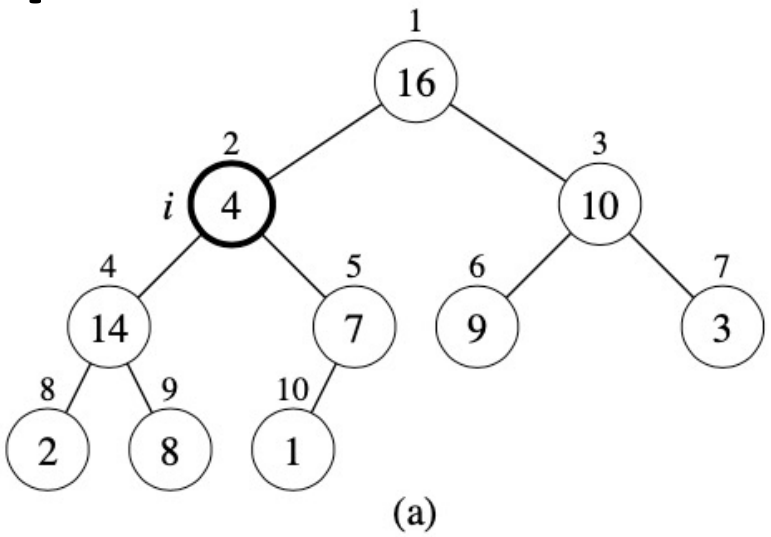
- Thus, the running time of BUILD-MAX-HEAP is $O(n)$.
- The same argument works for min-heap and MIN-HEAPIFY.

5.1.3. Heapsort

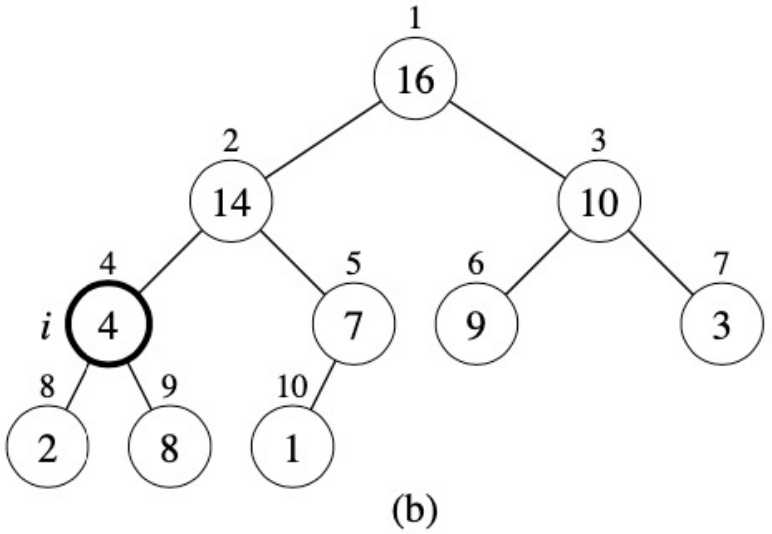
For a given input array, the Heapsort works as follows:

- We build a max-heap from the array.
- We start with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- We discard this last node by decreasing the heap size and call MAX-HEAPIFY on the new and possibly incorrectly placed root.
- We repeat this process until only one node (the smallest element) remains, which is in the correct place.

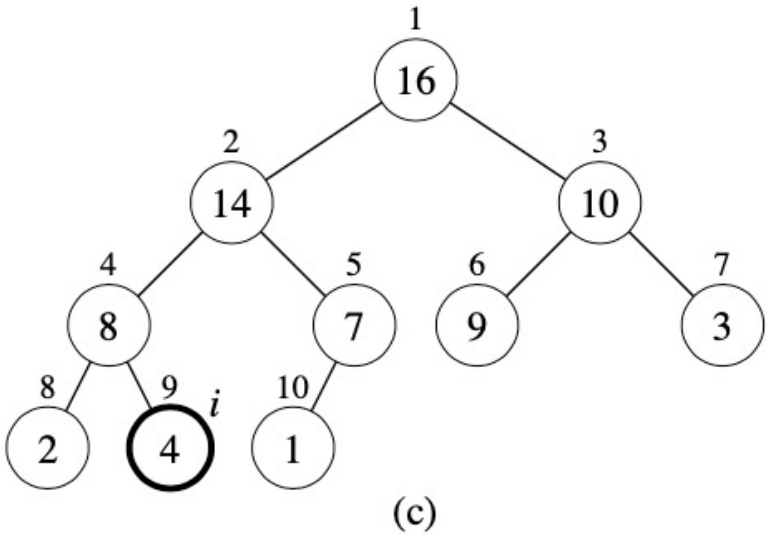
5.1.3. Heapsort



(a)



(b)



(c)

5.1.3. Heapsort

Algorithm (HEAPSORT(A,n))

```
(1)  BUILD-MAX-HEAP(A, n)
(2)  for (n ≥ i ≥ 2) do
(3)     swap A[1] and A[i]
(4)     MAX-HEAPIFY(A, 1, i − 1)
(5)  od
```

- BUILD-MAX-HEAP: $O(n)$
- for loop: $n-1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

The running time is $O(n \lg n)$.

- Compare to Insertion Sort $\Theta(n^2)$ and Merge-Sort $\Theta(n \lg n)$
- The heapsort is slower than the merge sort. **But it does not require massive recursion or multiple arrays to work.**

Outline

5.2. Quick Sort

Quicksort is another sorting algorithm based on the divide-and-conquer process.

- Divide: the partition $A[p \dots r]$ into two subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in $A[p \dots q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ in each element in $A[q+1 \dots r]$.
- Conquer: We sort the two subarrays by recursive calls to QUICKSORT.
- Combine: No need to combine the subarrays, because they are sorted in place.

Algorithm (QUICKSORT(A,p,r))

```
(1) if (p < r) then
(2)     q = PARTITION(A, p, r)
(3)     QUICKSORT(A, p, q − 1)
(4)     QUICKSORT(A, q + 1, r)
(5) fi
```

*Initial call is $\text{QUICKSORT}(A, 1, n)$

5.2. Quick Sort

Algorithm (PARTITION(A,p,r))

```
(1)  x = A[r]
(2)  i = p − 1
(3)  for (p ≤ j ≤ r − 1) do
(4)     if (A[j] ≤ x) then
(5)        i = i + 1
(6)        SWAP(A[i], A[j])
(7)     fi
(8)  od
(9)  SWAP(A[i + 1], A[r])
(10) return i + 1
```

- PARTITION rearranges the subarray in place.
- PARTITION always selects the last element $A[r]$ in the subarray $A[p \ldots r]$ as the pivot (the element around which to partition).
- As the procedure executes, the array is partitioned into four regions, some which may be empty:

5.2. Quick Sort

Algorithm (PARTITION(A,p,r))

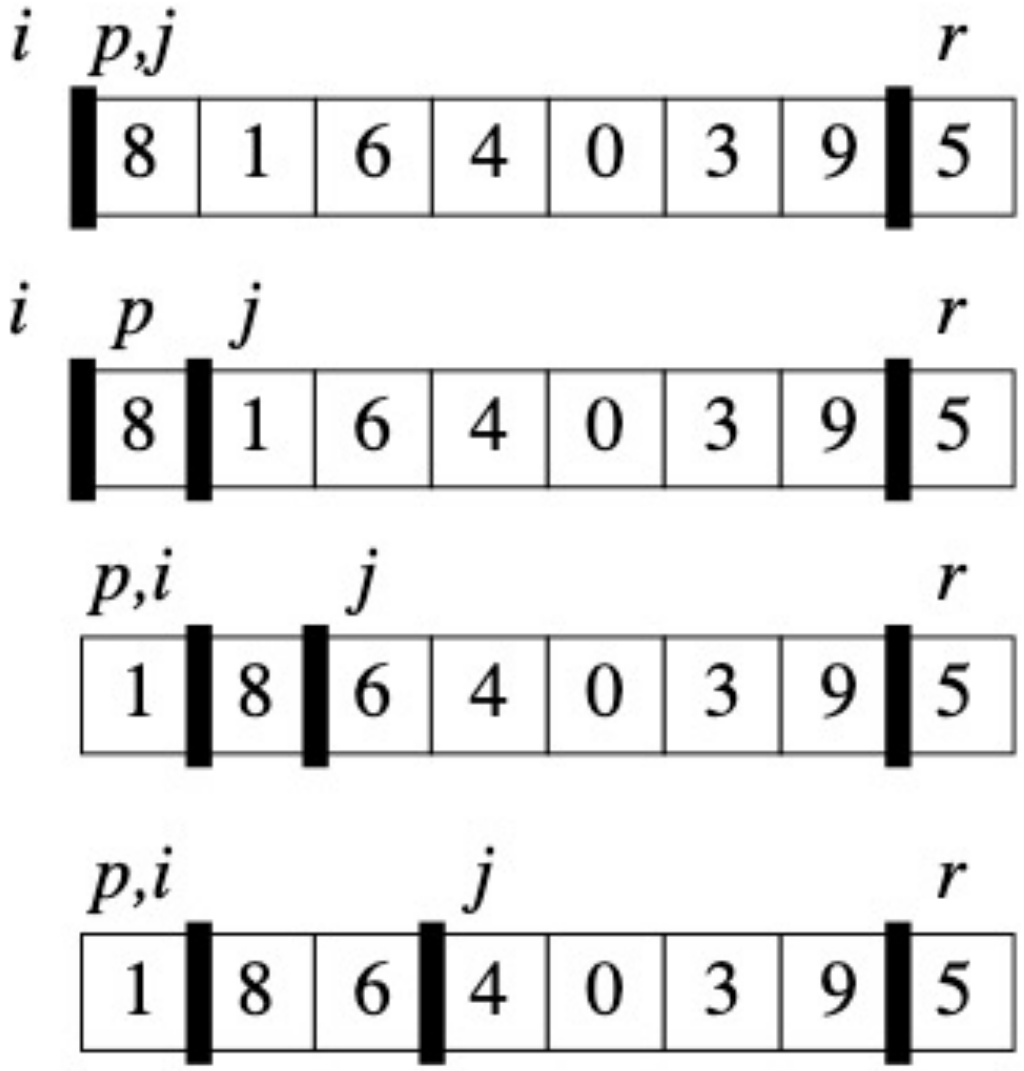
```
(1)  x = A[r]
(2)  i = p − 1
(3)  for (p ≤ j ≤ r − 1) do
(4)     if (A[j] ≤ x) then
(5)        i = i + 1
(6)        SWAP(A[i], A[j])
(7)     fi
(8)  od
(9)  SWAP(A[i + 1], A[r])
(10) return i + 1
```

Loop invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot.
3. $A[r]$ =pivot.

The additional region $A[j \ldots r-1]$ consists of elements that have not yet been processed. We do not yet know how they compare to the pivot element.

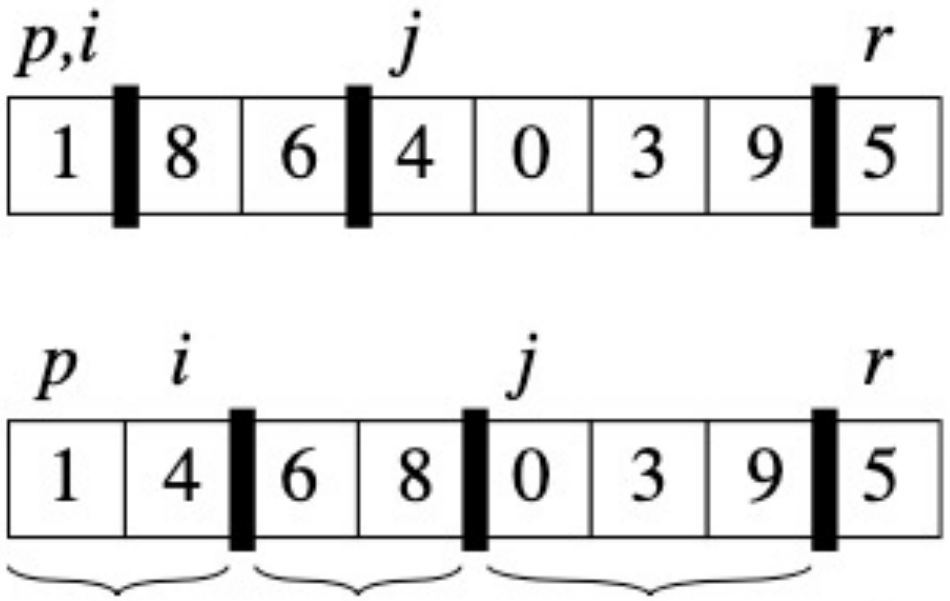
5.2. Quick Sort



Algorithm (PARTITION(A,p,r))

```
(1)  x = A[r]
(2)  i = p − 1
(3)  for (p ≤ j ≤ r − 1) do
(4)     if (A[j] ≤ x) then
(5)        i = i + 1
(6)        SWAP(A[i], A[j])
(7)     fi
(8)  od
(9)  SWAP(A[i + 1], A[r])
(10) return i + 1
```

5.2. Quick Sort



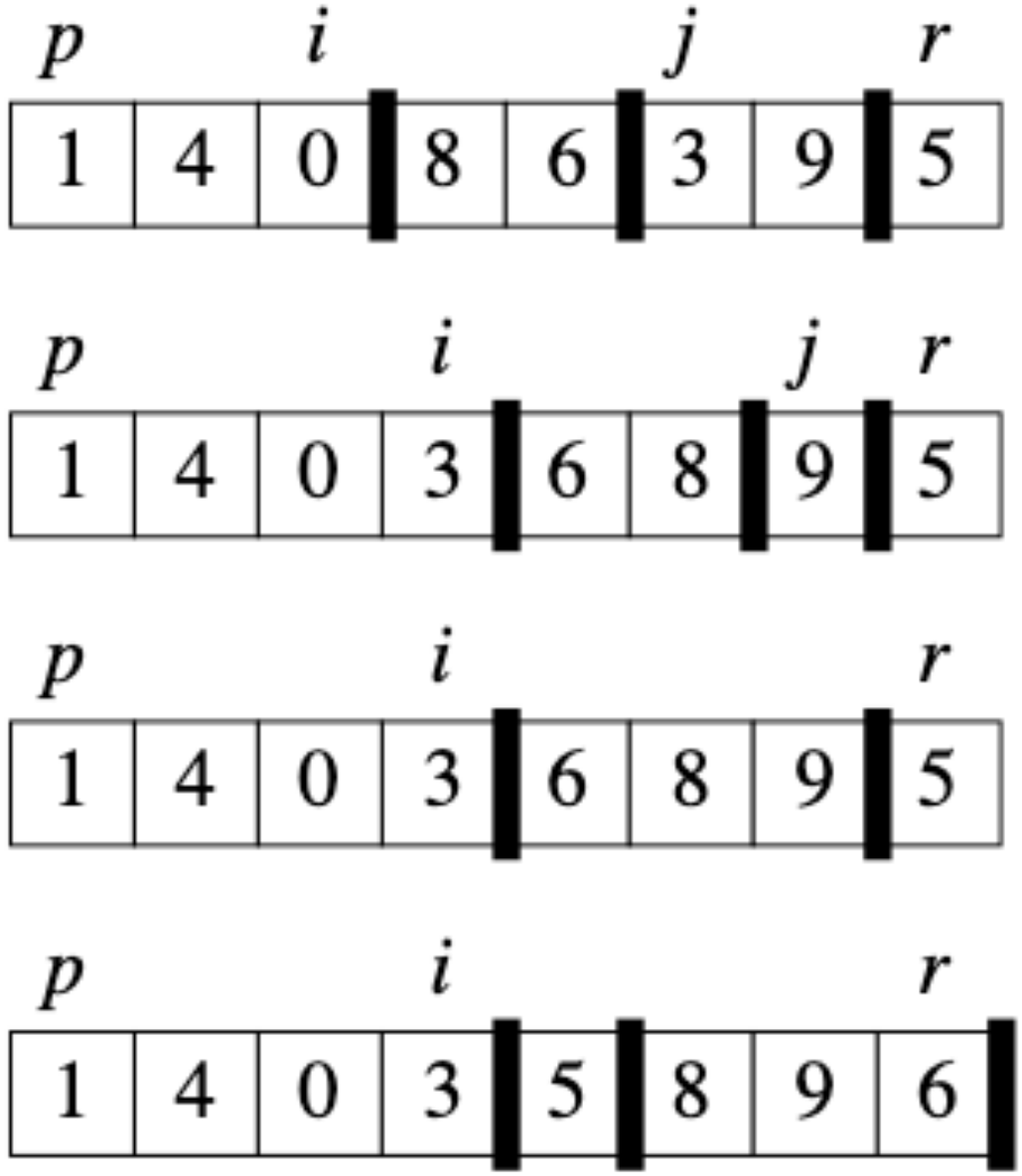
Algorithm (PARTITION(A,p,r))

```
(1)   x = A[r]
(2)   i = p − 1
(3)   for (p ≤ j ≤ r − 1) do
(4)      if (A[j] ≤ x) then
(5)         i = i + 1
(6)         SWAP(A[i], A[j])
(7)      fi
(8)   od
(9)   SWAP(A[i + 1], A[r])
(10)  return i + 1
```

5.2. Quick Sort

Algorithm (PARTITION(A,p,r))

```
(1)  x = A[r]
(2)  i = p − 1
(3)  for (p ≤ j ≤ r − 1) do
(4)     if (A[j] ≤ x) then
(5)        i = i + 1
(6)        SWAP(A[i], A[j])
(7)     fi
(8)  od
(9)  SWAP(A[i + 1], A[r])
(10) return i + 1
```



5.2. Quick Sort

- **PARTICIFICATION Correctness: We use the loop invariant.**
- **Initialization:** The subarray $A[p \dots i]$ and $A[i+1 \dots j-1]$ are empty before the start of the loop. The loop invariant conditions are therefore satisfied.
- **Maintenance:** Inside of the loop – we swap $A[j]$ and $A[i+1]$ and increase i if $A[j] \leq$ pivot element. The loop counter j is incremented.
- **Termination:** Loop terminates for $j = r$. We have three regions $A[p \dots i] \leq$ pivot, $A[i+1 \dots r-1] >$ pivot and $A[r] =$ pivot element.
- At the end of PARTITION we swap the pivot element in its correct position and then return its position.
- Time: $\Theta(n)$ to partition the subarray of size n.

5.2. Quick Sort

The running time of QUICKSORT depends on the partitioning of the subarrays.

- QUICKSORT is fast as MERGE-SORT if the partitioned subarrays are balanced (even sized).
- QUICKSORT is slow as INSERTION SORT if the partitioned subarrays are unbalanced (uneven sized).

Worst case: Subarrays completely unbalanced.

- Have 0 elements in one subarray and $n-1$ elements in the other subarray.
- The recurrence running time:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= \Theta(n^2)$$

- The running time is like INSERTION-SORT.

5.2. Quick Sort

Best case: Subarrays are always completely balanced.

- Each subarray has $\leq n/2$ elements.
- The recurrence running time:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

5.2. Quick Sort

Balanced partitioning:

- We assume that PATITION always produces a 9 to 1 split.
- Then the recurrence is

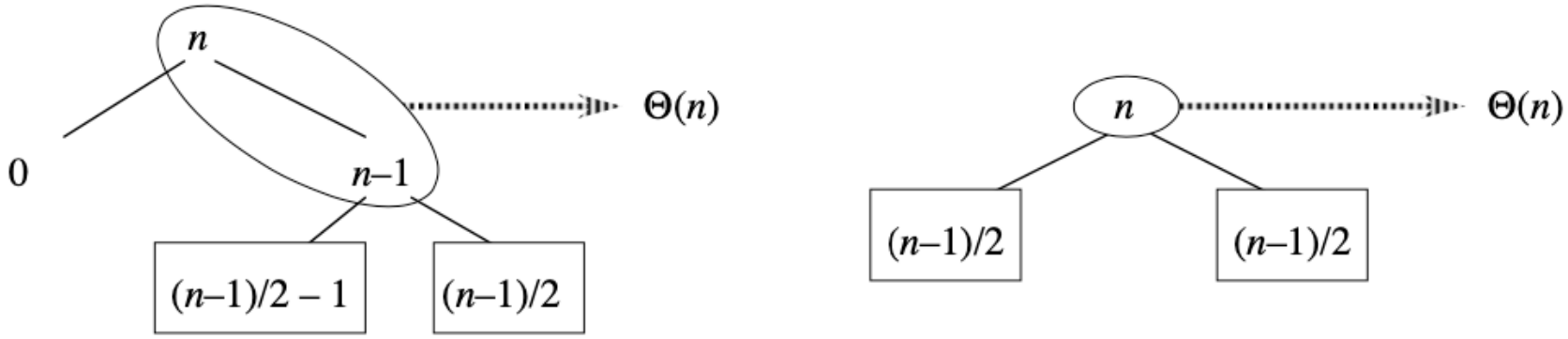
$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$$
$$= O(n \lg n)$$

- In the recursion tree, we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty.
- As long as it's a constant, the base of the log does not matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

5.2. Quick Sort

Average case:

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of “good” and “bad” splits throughout the recursion tree.
- It does not affect the asymptotic running time – assume that levels alternate between best- and worst-case splits.
- The bad split only adds to constant hidden in Θ notation.
- The same number of subarrays to sort but twice as much work is needed to this point.
- Both splits result in $\Theta(n \lg n)$ time, though the constant on the bad split is higher.



5.2. Quick Sort Randomized Quicksort

- We assumed so far that all input permutation are equally likely which is not always the case.
- We introduce randomization in order to improve on the quicksort algorithm.
- One option would be to use a random permutation of the input array.
- We use random sampling instead which is picking one element at random.
- Instead of using $A[r]$ as the pivot element we randomly pick an element from the subarray.

RANDOMIZED-PARTITION(A, p, r)
1 $i = \text{RANDOM}(p, r)$
2 $EXCHANGE\ A[r] \leftrightarrow A[i]$
3 **return** $\text{PARTITION}(A, p, r)$

RANDOMIZED-QUICKSORT(A, p, r)
1 **if** $p < r$
2 $q = \text{RANDOMIZED} - \text{PARTITION}(A, p, r)$
3 $\text{RANDOMIZED} - \text{QUICKSORT}(A, p, q - 1)$
4 $\text{RANDOMIZED} - \text{QUICKSORT}(\text{A}, \text{q} + 1, \text{r})$

*Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

*Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized quicksort, but not in randomized-quicksort.

5.3. Counting Sort

5.3. Counting Sort

Insertion Sort as Tree View:

- We have 1 tree for each n.
- We view the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.
- The decision tree models all possible execution traces.

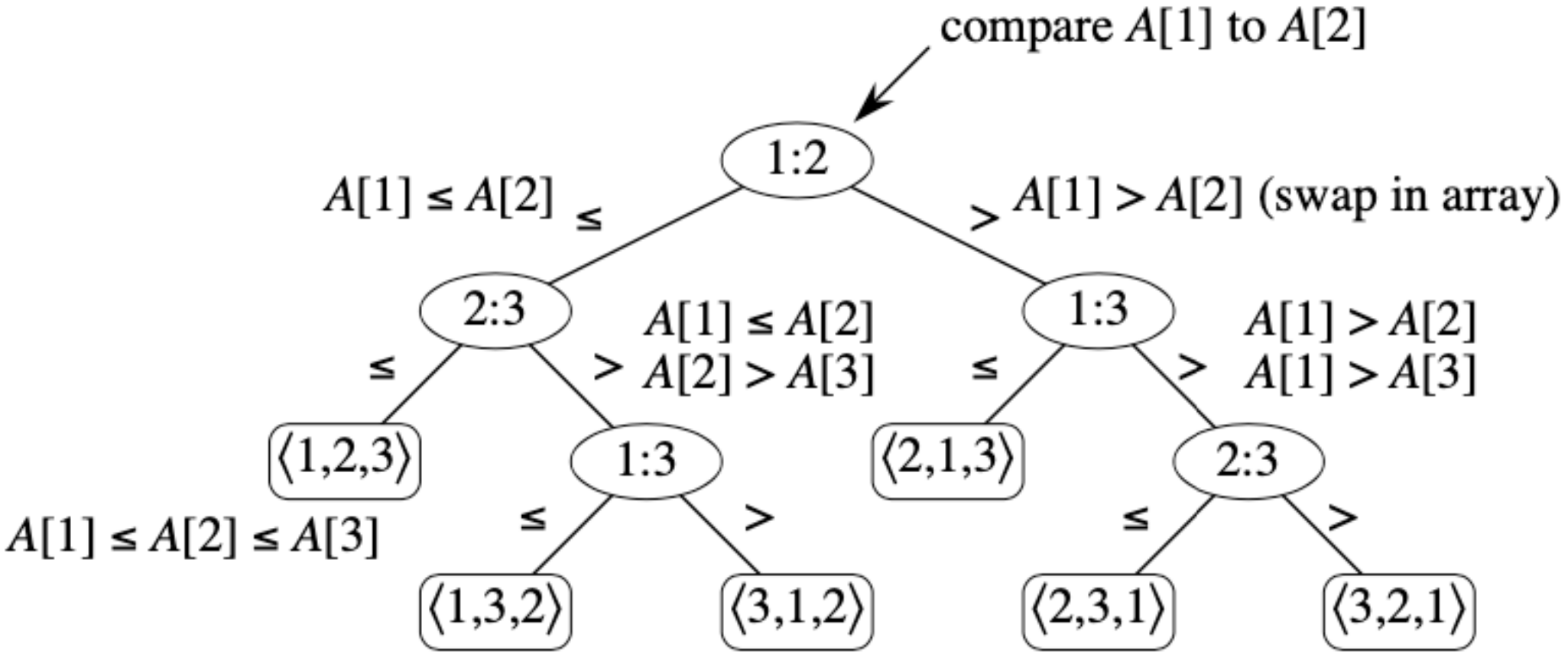
Do not confuse the decision tree with a recursion tree!

Length of the longest path?

- Measured from the root to a leaf.
- The length depends on the sorting algorithm.
- Insertion sort is $\Theta(n^2)$ and Merge sort is $\Theta(n \lg n)$.

5.3. Counting Sort:

For insertion sort on 3 elements:



5.3. Counting Sort:

Lemma

- Any binary tree of height h has $\leq 2^h$ leaves.

Theorem

- Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

5.3. Counting Sort:

Proof (Theorem)

- We have reachable l leaves, $l \geq n!$, by lemma $n! \leq l \leq 2^h$ or $2^h \geq n!$.
- After taking logarithms we get $h \geq \lg(n!)$.
- Using the Stirling approximation: $n! > \left(\frac{n}{e}\right)^n$,

$$h \geq \lg\left(\frac{n}{2}\right)^n = n\lg\left(\frac{n}{e}\right) = n\lg n - n\lg e = \Omega(n\lg n)$$

5.3. Counting Sort:

Proof (Lemma)

- Basis: For $h = 0$ we have a tree with just one leaf $2^h = 1$.
- Inductive step: Assumption holds for height $h - 1$. Extend this tree by making as many new leaves as possible. Each leaf will now be parent of to two new leaves (binary tree).
- # of leaves for height $= 2 \cdot (\#\text{ of leaves for height } h - 1) = 2 \cdot 2^{h-1} = 2^h$.

We can state that heapsort and merge sort are asymptotical optimal comparison sort algorithms.
⇒ But how can we then sort in linear time?

5.3. Counting Sort:

Counting sort: Non-comparison sorting algorithm.

Our key assumption is that the numbers to be sorted are integers from the set $\{0,1, \dots, k\}$.

Input: $A[1 \dots n]$, where $A[j] \in \{0, \dots, k\}$ for $j = 1, \dots, n$. The array A, values n and k are given as parameters.
Output: Sorted array $B[1, \dots, n]$. B is a parameter and already allocated.
Auxiliary storage: $C[0 \dots k]$.

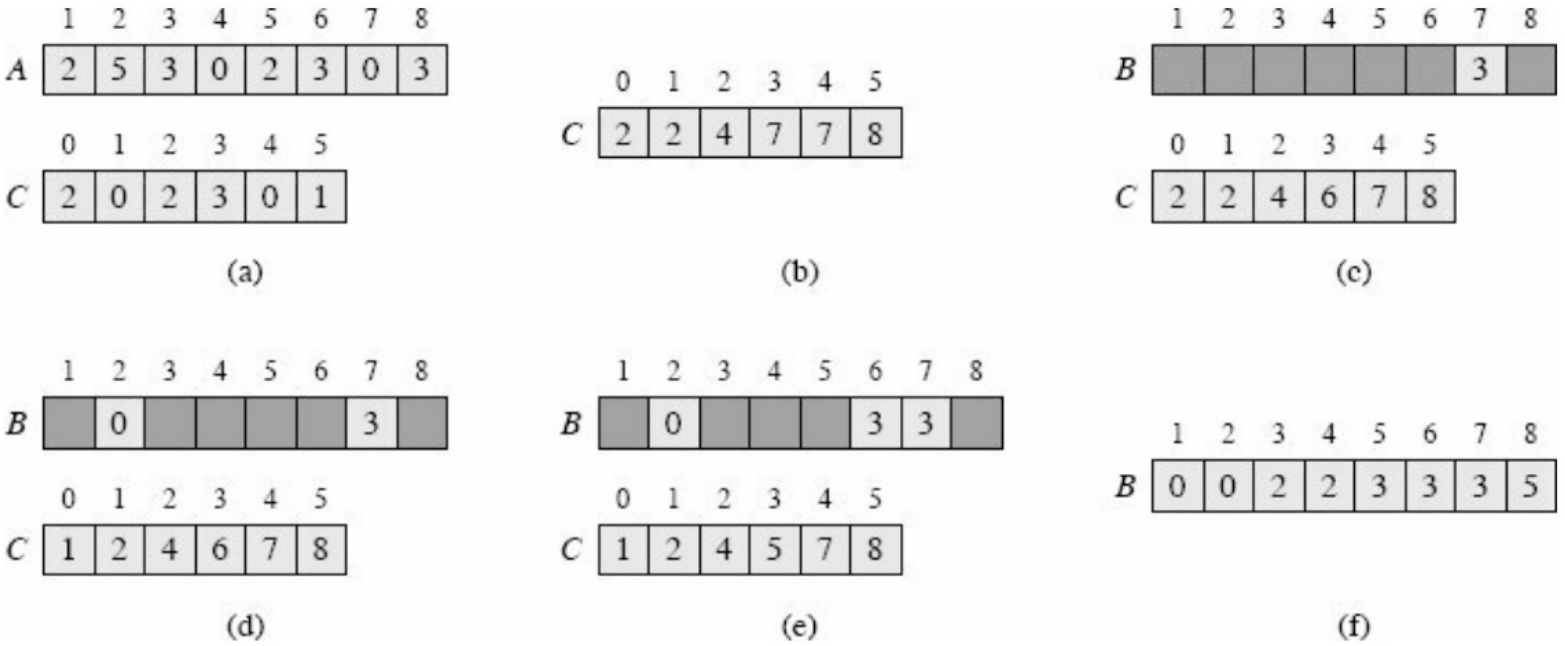
5.3. Counting Sort:

Algorithm (COUNTING-SORT(A,B,n,k))

```
(1)  create new array C[0...k]
(2)  for (0 ≤ i ≤ k) do
(3)     C[i] = 0
(4)  for (1 ≤ j ≤ n) do
(5)     C[A[j]] = C[A[j]] + 1
(6)  for (1 ≤ i ≤ k) do
(7)     C[i] = C[i] + C[i − 1]
(8)  for (n ≥ j ≥ 1) do
(9)     B[C[A[j]]] = A[j]
(10)    C[A[j]] = C[A[j]] − 1
```

- Counting sort is stable.
- Means that keys with the same value appear in the same order in the output as the appeared in the input.
- The last loop in the algorithm ensures this property.
- (Insertion sort, Merge sort)

5.3. Counting Sort:



- Counting sort is stable.
- Means that keys with the same value appear in the same order in the output as the appeared in the input.
- The last loop in the algorithm ensures this property.
- (Insertion sort, Merge sort)

5.3. Counting Sort:

Analysis:

- Running time $\Theta(n + k)$ which is $\Theta(n)$ if $k = \mathrm{O(n)}$.

Practical?

- Not a good idea to use it to sort 32-bit values.
- Might not be a good idea for 16-bit values.
- Probably a good idea for 8-bit or 4-bit value. Strongly depends on the number of values n.

Memory consumption can be a problem.

- The auxiliary storage C necessary for goes from 0 to k.
- The 32-bit integers we need 16GB of auxiliary storage. We need a 32-bit counter for each of the 0 to $k = 2^{32} - 1$.
- $\Rightarrow$ We will use counting sort within radix sort.

5.4. Radix Sort

5.4. Radix Sort

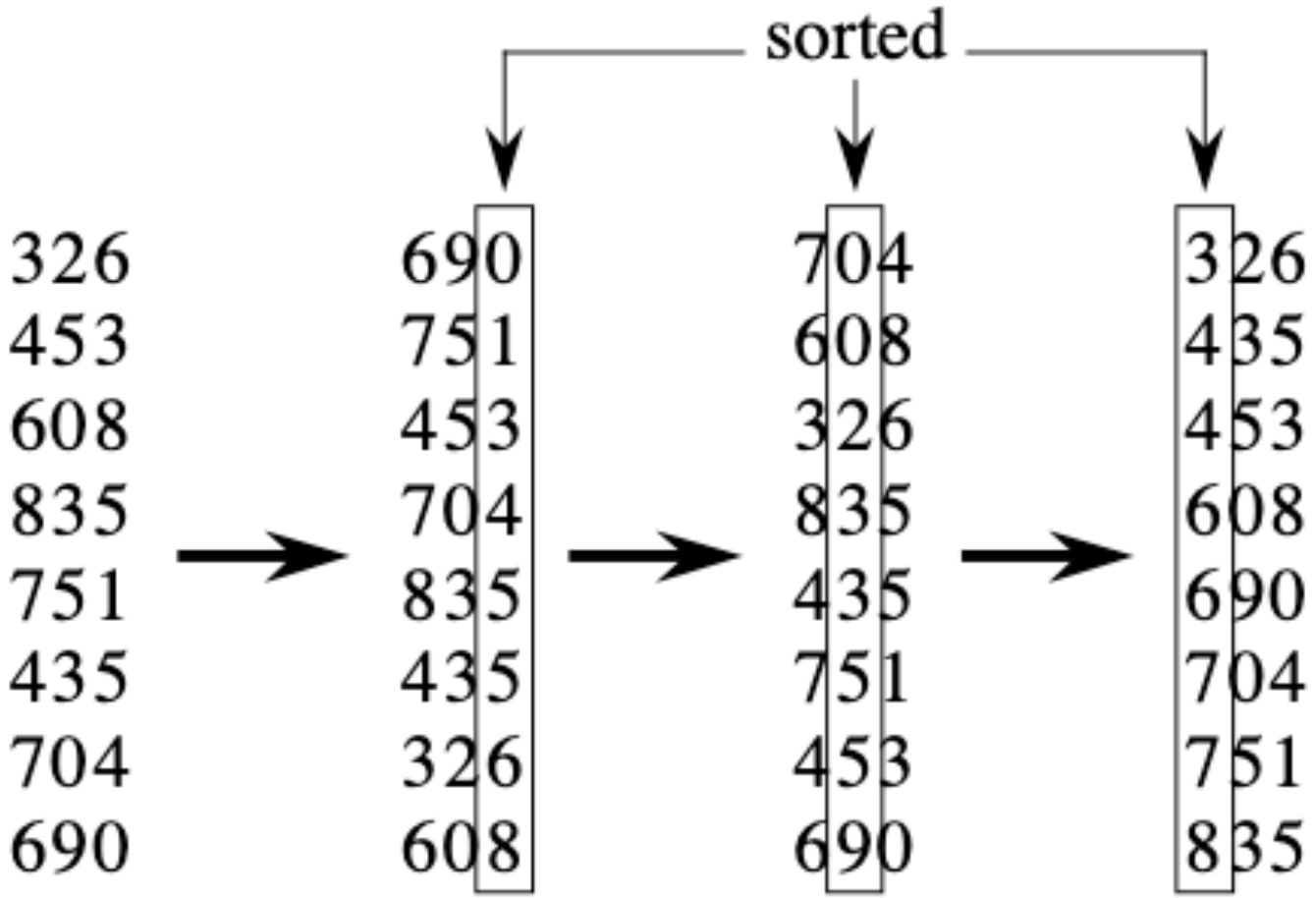
- Goes back to IBM and census in early 1900.
- Card sorters would work on one column at a time.
- Key idea: Sort least significant digits first.

```
RADIX-SORT(A, d)
1       for (1 ≤ i ≤ d) do
2           sort Array A on digit i using stable sorting algorithm
```

D is the highest-order digit

5.4. Radix Sort

Example:



5.4. Radix Sort

Correctness:

- We use induction on the number of passes (loop variable i).
- We assume that the digits $1, \dots, i-1$ are sorted.
- We show that a stable sorting algorithm on digit i leaves the digits $1, \dots, i-1$ sorted.
  - If 2 digits in position i are different, ordering by position i is correct, and positions $1, \dots, i-1$ are irrelevant.
  - If 2 digits in position i are equal, then the numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

Analysis:

- We assume that we use counting sort as the intermediate sort.
- Running time $\Theta(n+k)$ per pass (digits in range $0, \dots, k$).
- We have to do d passes (for loop).
- The total running time is $\Theta\big(d(n+k)\big) \Rightarrow \Theta(dn)$, if $k = O(n)$.

5.4. Radix Sort

Analysis:

- We assume that we use counting sort as the intermediate sort.
- Running time $\Theta(n + k)$ per pass (digits in range $0, \ldots, k$).
- We have to do d passes (for loop).
- The total running time is $\Theta\big(d(n + k)\big) \Rightarrow \Theta(dn)$, if $k = O(n)$.

5.4. Radix Sort

Break values into digits:

- We have n words with b bits per word.
- We break into r-bit digits $\Rightarrow d = \left\lceil \frac{b}{r} \right\rceil$.
- We use counting sort with $k = 2^r - 1$.
- Example: 32-bit words, 8-bit digits. $b = 32, r = 8, d = 4, k = 2^8 - 1 = 255$.
- Running time: $\Theta\left(\frac{b}{r}(n + 2^r)\right)$ for Radix sort.

5.4. Radix Sort

How do we choose r?

- We have to balance $\frac{b}{r}$ and $n + 2^r$.
- Choose $r \approx \lg n \Rightarrow \Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta\left(\frac{bn}{\lg n}\right)$ running time.
- Choose $r < \lg n$, then $\frac{b}{r} > \frac{b}{\lg n}$ and the term $n + 2^r$ does not improve.
- Choose $r > \lg n$, then the term $n + 2^r$ increases. For $r = 2 \lg n$ we get $2^r = 2^{2 \lg n} = \left(2^{\lg n}\right)^2 = n^2$.

Example: If we sort 2^{16} numbers of size 32-bit, we use $r = \lg 2^{16} = 16$ bits. We perform $\left\lceil \frac{b}{r} \right\rceil = 2$ passes.

5.4. Radix Sort

How does radix sort compare to merge sort and quicksort?

- For 1 million ($\approx 2^{20}$) 32 bit integers.
- Radix sort performs $\left\lceil \frac{32}{20} \right\rceil = 2$ passes, whereas merge or quicksort perform $\lg n = 20$ passes.
- One radix sort is really 2 passes ⇒ one to take and one to move data.

How radix sort violates comparison sort rules?

- Using counting sort allows us to gain information without directly comparing 2 keys.
- Use the keys as array indices.

5.5. Bucket Sort

Bucket Sort

We assume that the input is generated at random and the elements are distributed uniformly over $[0,1)$.

Idea:

- We divide interval $[0,1)$ into n equal-sized buckets.
- We distribute the n input values into the buckets.
- We sort each bucket afterwards.
- And then we go through the buckets in order, listing the elements in each one.

Bucket Sort

Algorithm (BUCKET-SORT(A,d))

```
(1) create new array B[0 ... n − 1] of lists
(2) for (0 ≤ i ≤ n − 1) do
(3)     Let B[i] be an empty list
(4) for (1 ≤ i ≤ n) do
(5)     Put (insert) A[i] into bucket (list) B[⌊n · A[i]⌋]
(6) for (0 ≤ i ≤ n − 1) do
(7)     sort list B[i] with insertion sort
(8) concatenate lists B[0], ..., B[n − 1] together in order
(9) return concatenated lists
```

Correctness:

- We consider $A[i], A[j]$ (assume $A[i] \leq A[j]$).
- $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$ follows.
- $A[i]$ is placed in same or in bucket with lower index than $A[j]$.
  - same bucket ⇒ insertion sort fixes order.
  - earlier bucket ⇒ concatenation of list ensures order.

5.5. Bucket Sort

Example:

A[1…10]=[0.89, 0.13, 0.45, 0.2, 0.54, 0.53, 0.7, 0.85, 0.51, 0.49]

- Initialize B[0…9]=0,…,0
- Put A[i] into the 10 buckets:

B[0]	B[1]	B[2]	B[4]	B[5]
0	0.13	0.2	0.45, 0.49	0.54, 0.53, 0.51
B[6]	B[7]	B[8]	B[9]	
0	0.7	0.89, 0.85	0	

5.5. Bucket Sort

- Sort each of the 10 buckets:

B[0]	B[1]	B[2]	B[4]	B[5]
0	0.13	0.2	0.45, 0.49	0.51, 0.53,0.54
B[6]	B[7]	B[8]	B[9]	
0	0.7	0.85, 0.89		

- Concatenate all of the buckets: B[0…9]=0.13, 0.2, 0.45, 0.51, 0.53, 0.54, 0.7, 0.85, 0.89

5.5. Bucket Sort

Analysis:

- Algorithm relies on the fact that no bucket is getting too many values (uniformly distributed).
- Total running time (except the insertion sort) is $\Theta(n)$.
- Intuition: Each bucket gets a constant number of elements. Sorting then takes constant time for each bucket.
- Using our intuition we then get $O(n)$ sorting time for all buckets.
- On average we have 1 element per bucket ⇒ we expect each bucket to have few elements.

⇒ We need to do a careful analysis.

5.5. Bucket Sort

- Define a random variable n_i = the number of elements placed in bucket $B[i]$.
- Insertion sort runs in quadratic time, therefore the recurrence for bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

- We take the expectations on both sides:

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} E\left[O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right)$$

5.5. Bucket Sort

Claim: $E[n_i^2] = 2 - \frac{1}{n}$ for $i = 0, \ldots, n-1$.

Proof (claim):

We define indicator random variables

- $X_{ij} = l\{A[j] \text{ falls in bucket } i\}$.
- $\Pr\{A[j]\ falls\ in\ bucket\ i\} = \frac{1}{n}$.
- $n_i = \sum_{j=1}^{n} X_{ij}$.

Then

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right]$$

5.5. Bucket Sort

- linearly of expectations

$$= \sum_{j=1}^{n} E\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} E\left[X_{ij}X_{ik}\right]$$

$$E\left[X_{ij}^2\right] = 0^2 \cdot \Pr\{A[j] \text{ does not fall in bucket } j\} + 1^2 \cdot \Pr\{A[j] \; falls \; in \; bucket \; j\}$$

$$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} = \frac{1}{n}.$$

- $E\left[X_{ij}X_{ik}\right]$ for $j \neq k$:

X_{ij} and X_{ik} are independent random variables because of $j \neq k$.

$$E\left[X_{ij}X_{ik}\right] = E\left[X_{ij}\right] \cdot E\left[X_{ij}\right] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}.$$

5.5. Bucket Sort

- Therefore:

$$E\left[n_i^2\right] = \sum_{j=1}^{n} \frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \frac{1}{n^2} = n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2}$$

$$= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n}$$

$$= 1 + 1 - \frac{1}{n} = 2 - \frac{1}{n}.$$

5.5. Bucket Sort

Analysis:

Using our claim, we get

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right)$$
$$= \Theta(n) + O(n) = \Theta(n)$$

- We use a function of key values to index into an array ⇒ Not a comparison sort.
- Performance analysis relies on a uniform distribution on $[0,1)$.

We did a probabilistic analysis. Running time depends on the distribution of inputs. Different from randomized algorithm where we impose a distribution.

5.6. Conclusion

5.6. Conclusion

1. 5 different sorting algorithms are discussed.
2. Each algorithm has its own characteristics and benefits.
3. Stable sorting algorithms.