ALGORITHMS – COMTEK3,CCT3 & ESD3

Sorting Algorithms

Ramoni Adeogun

Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

Outline

- Sorting
- Sorting algorithms
  - Heap sort
  - Quick sort
- Linear time sorting algorithms
  - Counting sort
  - Radix sort
  - Bucket sort
- Summing up
- Exercises – Group rooms

Sorting

- Sorting **re-arrange elements** of an array into a list in which the elements are in **increasing (decreasing)** order.

- For instance,
  - sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9.
  - sorting the list d, h, c, a, f (using alphabetical order) produces the list a, c, d, f, h.

- Sorting problems arises in several applications/contexts:
  - generating a telephone directory requires alphabetic ordering of subscriber names
  - creating a useful dictionary requires ordering of words

Sorting

- The sorting problem:
  - **Precondition**: Sequence of n numbers $\{a_1, a_2, \cdots, a_n\}$
  - **Postcondition**: re-ordered sequence of n numbers $\{a'_1, a'_2, \cdots, a'_n\}$ such that $\{a'_1 \leq a'_2 \leq \cdots \leq a'_n\}$

- Several sorting algorithms have been developed over the years:
  - Bubble sort
  - Insertion sort
  - Heap sort
  - Counting sort
  - Bucket sort
  - Merge sort, etc

Sorting algorithms

- A **sorting algorithm** is made up of a series of instructions that
  - takes an array as input,
  - performs specified operations on the array and
  - outputs a sorted array

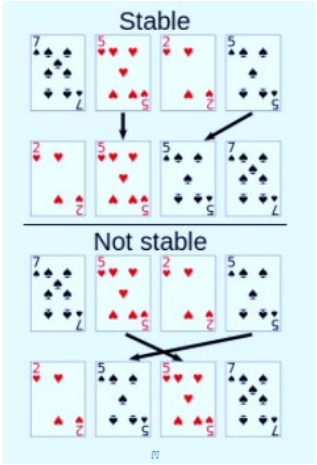
- A **sorted array** is an array that is in a particular order example,
  - [a,b,c,d] is sorted alphabetically,
  - [1,2,3,4,5] is a list of integers sorted in increasing order, and
  - [5,4,3,2,1] is a list of integers sorted in decreasing order.

- Sorting algorithms are broadly classified into two:
  - comparison sorts
  - integer sorts

Properties of Sorting Algorithms

- Before applying an algorithm to a sorting problem, it is important to be aware of the:
  - **time-complexity**
  - **space complexity**
  - **stability**

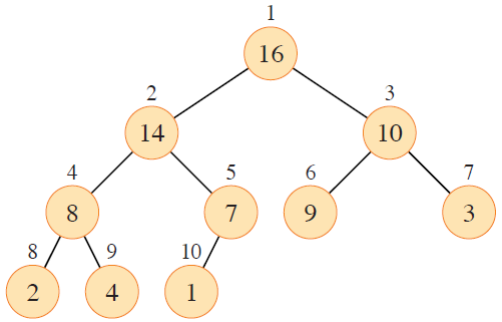
Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)



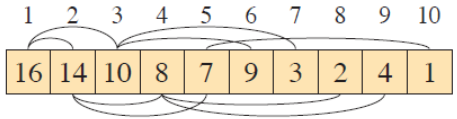
Heapsort

Heap

- Heap A (not garbage-collected storage) is a nearly complete binary tree.
  - Height of node = # of edges on a longest simple path from the node down to a leaf.
  - Height of heap = height of root = $\Theta(\log_2(n))$



- A heap can be stored as an array A.
  - Root of tree is $A[1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Computing is fast with binary representation implementation



Heap property

- Max-heaps (largest element at root),
  - **max-heap property**: for all nodes i, excluding the root, $A[PARENT(i)] \geq A[i]$
- Min-heaps (smallest element at root),
  - **min-heap property**: for all nodes i, excluding the root, $A[PARENT(i)] \geq A[i]$
- By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is at the root.
  - Similar argument for min-heaps.
- The heapsort algorithm we'll show uses max-heaps.
- **Note**: In general, heaps can be k-ary tree instead of binary.

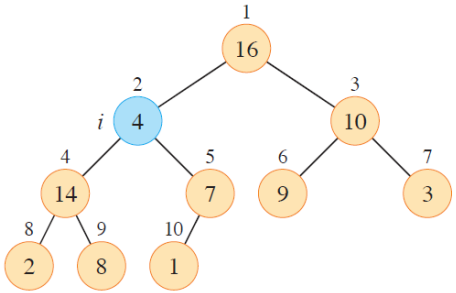
Maintaining the heap property

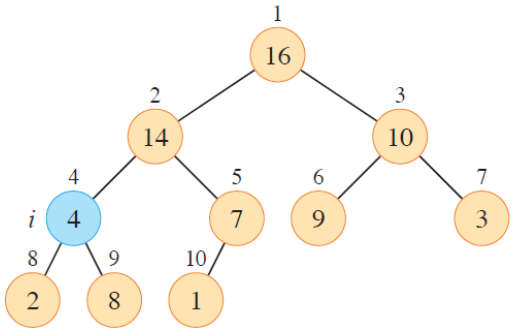
- **MAX-HEAPIFY** is important for manipulating max-heaps. It is used to maintain the max-heap property.
  - Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
  - Assume left and right subtrees of i are max-heaps.
  - After MAX-HEAPIFY, subtree rooted at i is a max-heap.

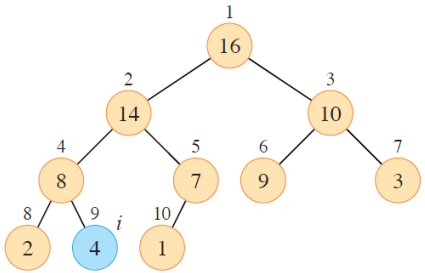
```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

How MAX-HEAPIFY works

- How **MAX-HEAPIFY** works:
  - Compare $A[i]$ $A[\text{LEFT}[(i)]$ and $A[\text{RIGHT}(i)]$
  - 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.







The complexity of Max-heapify algorithm

- The execution time for the MAX-HEAPIFY is as follows
  - $\Theta(1)$ to restore the relations between $A[i], A[\text{LEFT}(i)]$ and $A[\text{RIGHT}(i)]$, plus
  - the time it takes to execute MAX-HEAPIFY on the sub-nodes
    - Each individual nodes maximally has $2n/3$ sub-nodes
      - *Why $2n/3$ is left as an exercise for you to find out*

- *Therefore,* the complexity of this algorithm can be expressed as

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

- We have seen something similar to this before and it becomes

$$T(n) = O(\log_2(n))$$

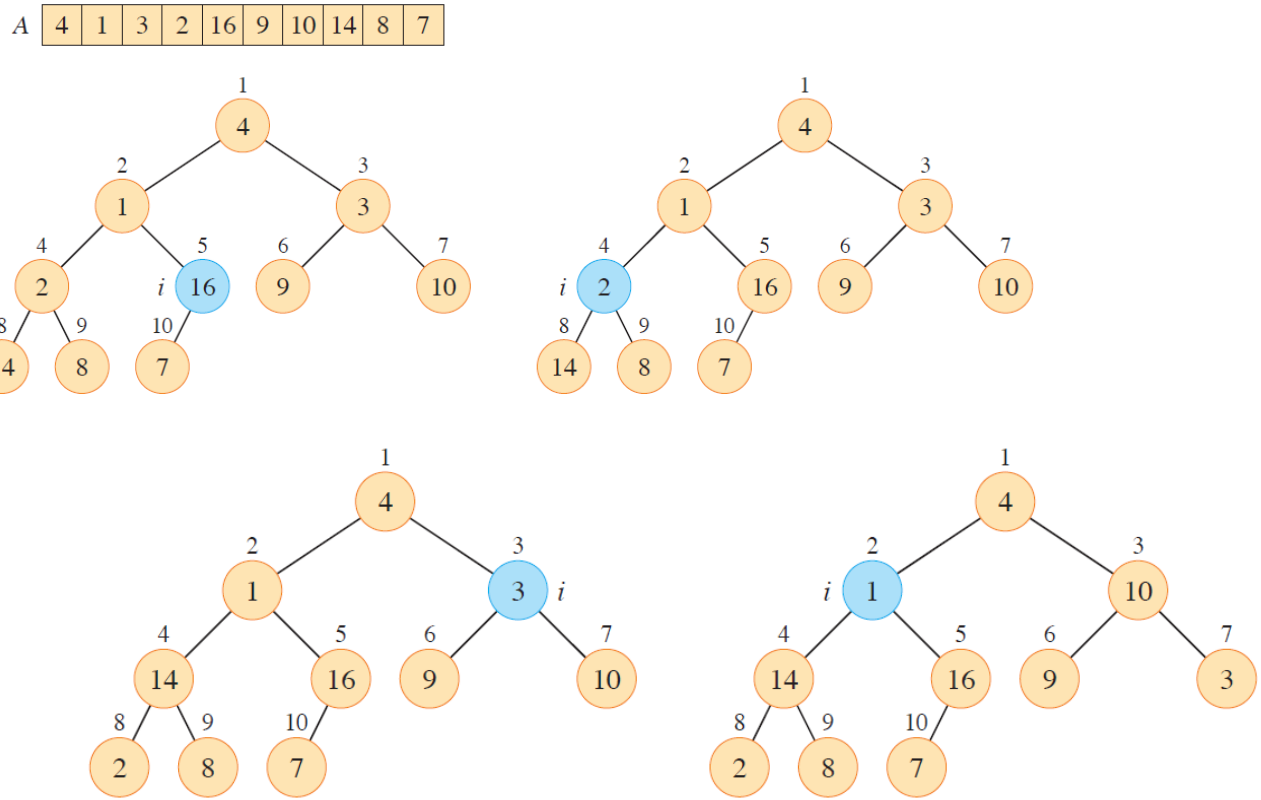
Building a heap

- The following procedure, given an unordered array, will produce a max-heap.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- Example: Building a max heap from the following unsorted array results in the first heap example.
  - i starts off as 5.
  - MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): $16, 2, 3, 1, 4$.

Building a heap - Example



Correctness of BUILD-MAX-HEAP

- **Loop invariant**: At the start of every iteration of for loop, each $i+1, 1+2, \cdots, n$ is root of a max-heap.
- **Initialization**: We know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the for loop, the invariant is initially true.
- **Maintenance**: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, 1+2, \cdots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.
- **Termination**: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

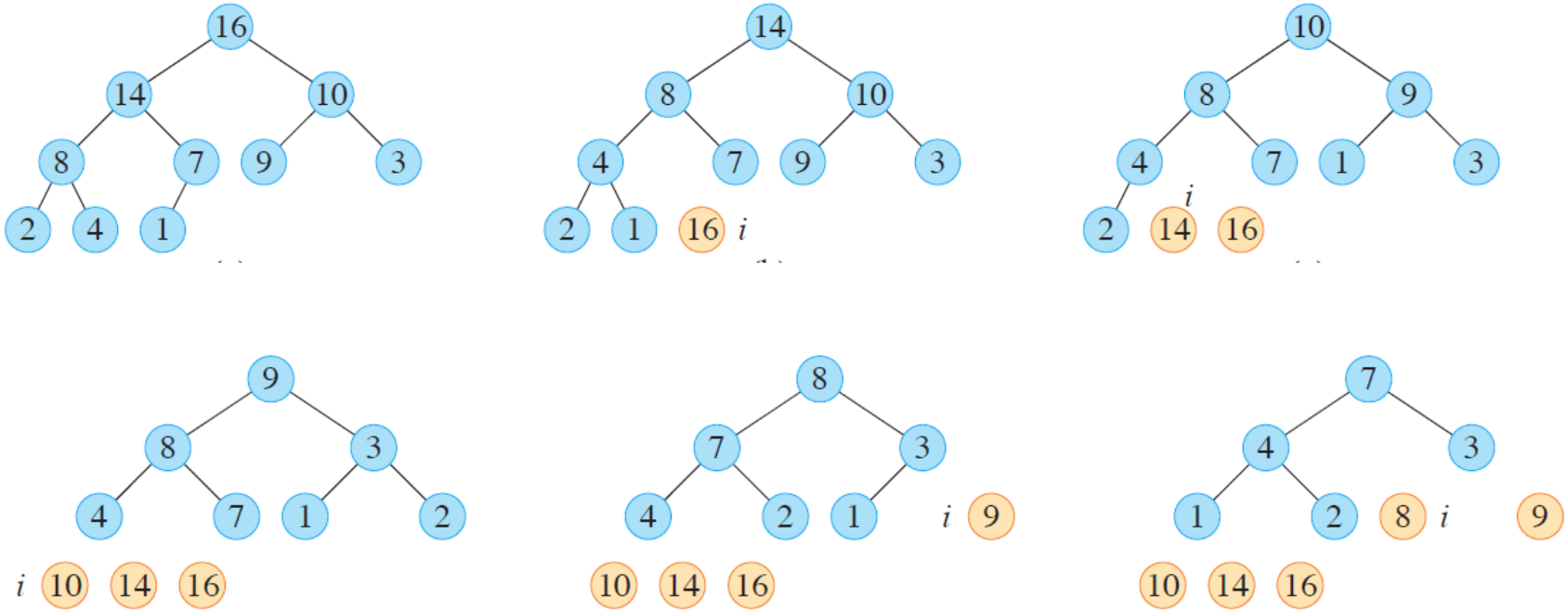
The Heapsort algorithm

Given an input array, the heapsort algorithm acts as follows:

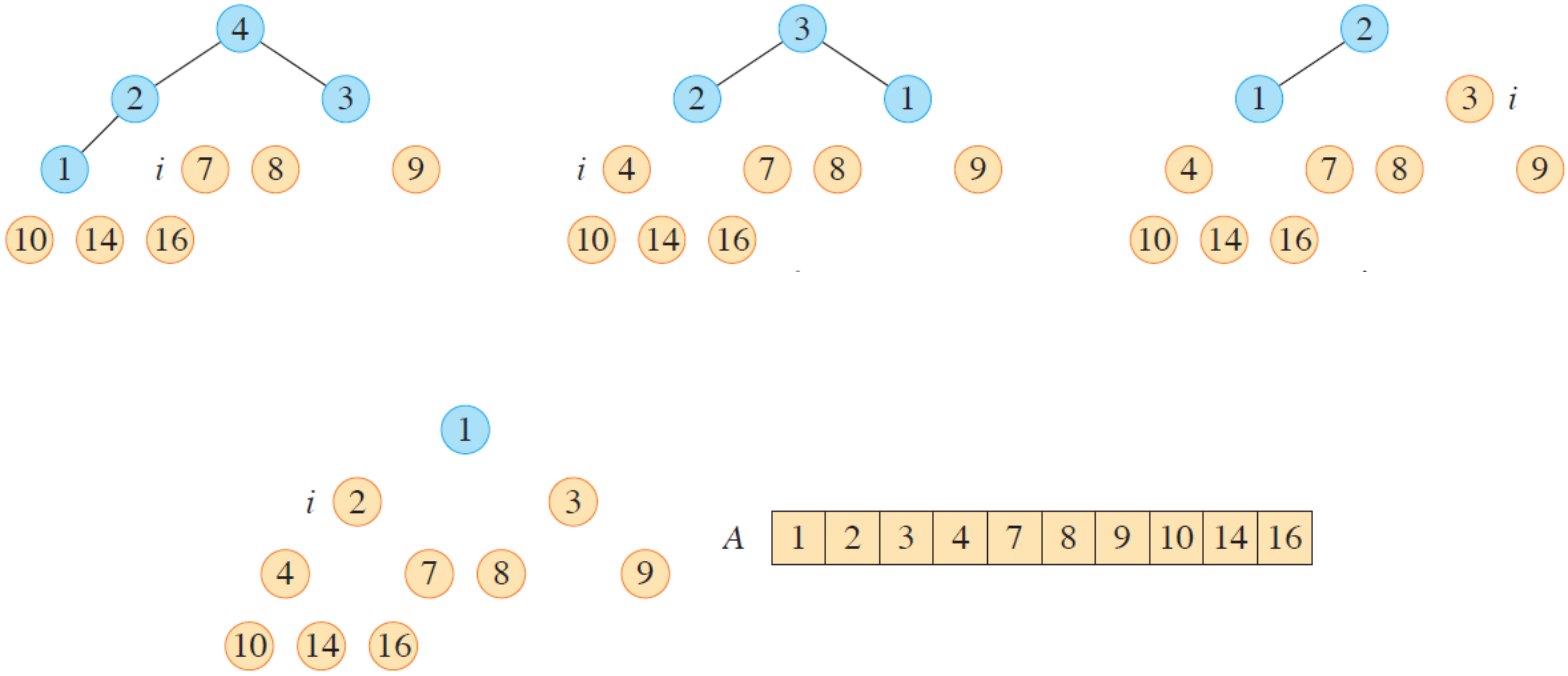
- Builds a **max-heap from the array**.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling **MAX-HEAPIFY** on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

The Heapsort algorithm-example



The Heapsort algorithm-example



The Heapsort algorithm - Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(n \log_2(n))$
- Total time: $O(n \log_2(n))$
- **Note**: Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.

Quicksort

- Worst-case running time: $\Theta(n^2)$.
- Expected running time: $\Theta(n \log_2(n))$.
- Constants hidden in $\Theta(n \log_2(n))$ are small.
- Sorts in place.

Quicksort

- Quicksort is based on the three-step process of divide-and-conquer.
- To sort the subarray $A[p, \ldots, r]$
  - **Divide**: Partition $A[p, \ldots, r]$ into two (possibly empty) subarrays $A[p, \ldots, q-1]$ and $A[q+1, \ldots, r]$ such that each element in the first subarray $A[p, \ldots, q-1]$ is $\leq A[q]$ and $A[q] \leq$ each element in the second subarray $A[q+1, \ldots, r]$
  - **Conquer**: Sort the two subarrays by recursive calls to QUICKSORT.
  - **Combine**: No work is needed to combine the subarrays, because they are sorted in place.
- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2        // Partition the subarray around the pivot, which ends up in A[q].
3        q = PARTITION(A, p, r)
4        QUICKSORT(A, p, q − 1)  // recursively sort the low side
5        QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Partitioning

- Partition subarray $A[p, \ldots, r]$ by the following procedure:

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- PARTITION always selects the last element A[r] in the subarray $A[p, \ldots, r]$ as the pivot—the element around which to partition.
- **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.

Correctness of PARTITION

- Use the loop invariant to prove the correctness of PARTITION:
  - **Initialization**: Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot and the subarrays $A[p, \dots, i]$ and $A[i + 1 + 1, \dots, j - 1]$ are empty.
  - **Maintenance**: While the loop is running, if $A[j] \leq$ pivot, then $A[j]$ and $A[i + 1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then only j is incremented.
  - **Termination**: When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p, \dots, i] \leq$ pivot, $A[i + 1, \dots, r - 1] >$ pivot, and $A[r] =$ pivot.

- The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i + 1]$ and $A[r]$.

Performance of Quicksort

- The running time of quicksort depends on the partitioning of the subarrays:
  - If the subarrays are balanced, then quicksort can run as fast as mergesort.
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.

- Worst case
  - Occurs when the subarrays are completely unbalanced.
  - Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
  - The recurrence:

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

  - Same running time as insertion sort.
    - In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Performance of Quicksort

- **Best case:**
  - Occurs when the subarrays are completely balanced every time.
  - Each subarray has $\leq n/2$ elements.
  - The recurrence:

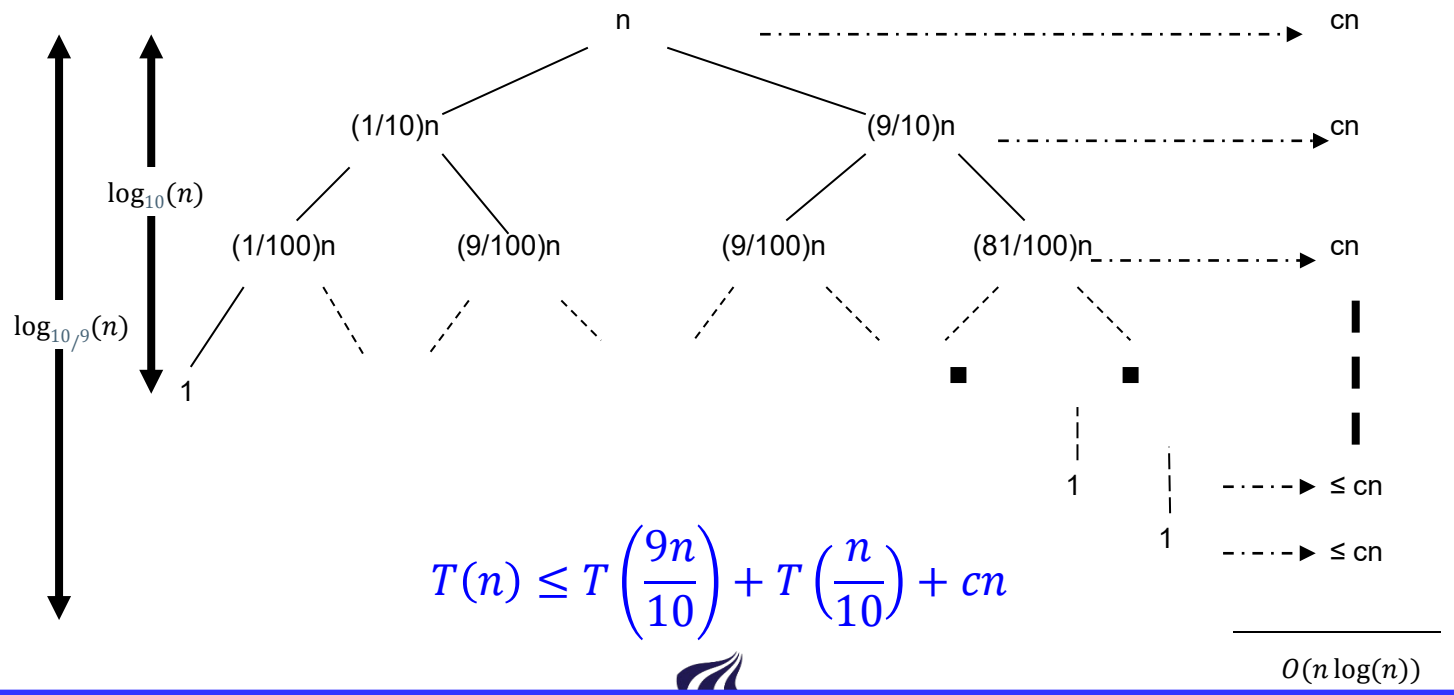
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- **Average case:**
  - Quicksort's average running time is much closer to the best case than to the worst case.
  - Imagine that PARTITION always produces a 9-to-1 split.
  - The recurrence:

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) = O(n \log_2(n))$$

Average performance of Quicksort

- Claim: the average performance is closer to the best case than to the worst case



Randomized version of Quicksort

- We have assumed that all input permutations are equally likely.
  - This is not always true.
- To correct this, we add randomization to quicksort.
  - We could randomly permute the input array.
  - Instead, we use random sampling, or picking one element at random.
- Don't always use A[r] as the pivot.
  - Instead, randomly pick an element from the subarray that is being sorted.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

- 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

Sorting in Linear Time

- **How fast can we sort?** We will prove a lower bound, then beat it by playing a different game.

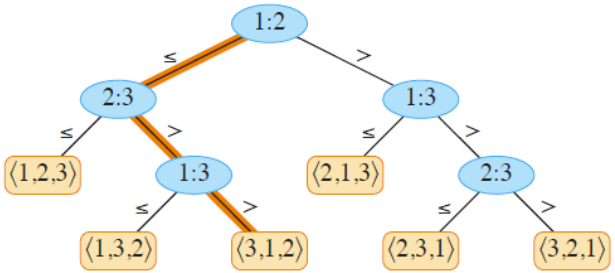
Comparison sorting

- The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
- All sorts seen so far are comparison sorts: insertion sort, selection sort, mergesort, quicksort, heapsort, treesort.

Lower bounds for sorting

- Lower bounds
  - $\Omega(n)$ to examine all the input
  - All sorting algorithms discussed so far are $\Omega(n \log_2(n))$ (the lower bound)
- Decision tree can be used as an abstraction of any comparison sort
  - Represents the comparisons made => everything else is ignored

  Example: Decision tree for insertion sort on three elements



- How many leaves on the decision tree?

Lower bounds for sorting

- The execution of a sorting algorithm based on comparisons is the same as searching through a binary tree
  - At each point we do a comparison, e.g. $a_i > a_j, a_i \leq a_j$
- This gives a set of possible terminating leaves
  - To be exact, $n!$ **leaves**
- The height of the decision tree gives our worst-case situation for a sorting algorithm
  - Because each branch leads to or contains a comparison
  - A lower boundary of complexity can be found
- The conclusions are that heap-sort and merge-sort are asymptotically optimal algorithms

Counting sort

- Depends on a key assumption: numbers to be sorted are integers in $\{0,1,\ldots,k\}$
- **Precondition**: $A[1..n]$ where $A[j] \in \{0,1,\ldots,k\}$. Array A and values n and k are given as parameters.
- Postcondition: Sorted array $B[1..n]$. B is assumed to be already allocated and is given as a parameter.
- **Auxiliary storage**: $A[0..k]$

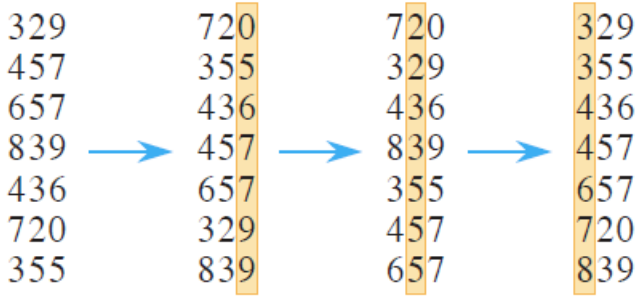
```
COUNTING-SORT(A, n, k)
1  let B[1:n] and C[0:k] be new arrays
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to n
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 // Copy A to B, starting from the end of A.
11 for j = n downto 1
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14 return B
```

- Counting sort is **stable** ☺
- Running time: $\Theta(n+k)$ which is $\Theta(n)$ if $k = O(n)$.

Radix sort

- Radix sort is based on sorting numbers by digit by digit
  - First to sort numbers by the least digit (ones)
  - Then the next digit (tens)
  - With this up to the highest digit (hundreds)

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1 : n] on digit i
```



- If counting sort is used, then sorting the radix can be done in $\Theta(d(n+k))$
- Given d is constant and $k = O(n)$ then radix sort in linear time!

- Given n b-bit numbers and for every positive integer $r \leq b$, then radix-sort sorts correctly in $\Theta((b/r)(n+2r))$ time!!

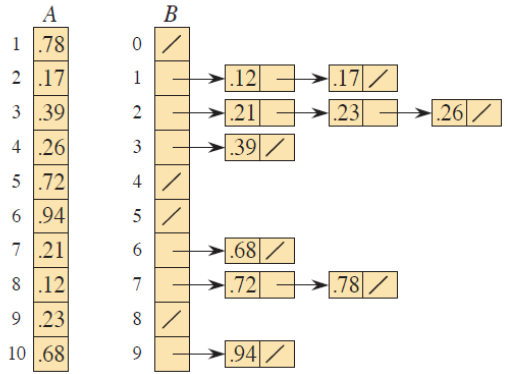
Bucket sort

Bucket sort: useful for sorting array with uniformly distributed values over a range

Idea

- Divide $[0,1)$ into n equal-sized buckets.
- Distribute the n input values into the buckets.
- Sort each bucket.
- Then go through buckets in order,
  - listing elements in each one.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```



Analysis of Bucket sort

- Assume that inputs are numbers created by a stochastic process which distribute data uniformly in the interval [0, 1)
- All lines of code takes $O(n)$ time with the exception of Insertion-sort call which takes $O(n^2)$ time
- But, then, how can we claim an algorithm runs in linear time $O(n)$ when it relies on something in quadratic time $O(n^2)$?
- Well, principally we cannot, but we can say it runs in expected linear time, i.e. $\Theta(n)$

Summary and conclusion

- Sorting algorithms in non-linear time
  - Heap sort - $O(n \log_2(n))$
  - Quick sort - $\Theta(n^2)$, but typically $O(n \log_2(n))$
- Sorting algorithms in linear time
  - Counting sort - $\Theta(n + k)$, but typically $O(n)$
  - Radix sort (dependent on sub-sorting algorithms used, with e.g. counting sort, one can achieve $T(n) = O(n)$)
  - Bucket sort - in average $O(n)$, but actually $O(n^2)$
- Remember there are more metrics involved and not just time, e.g. **memory usage**
- **Next lecture -**