# CSC 226

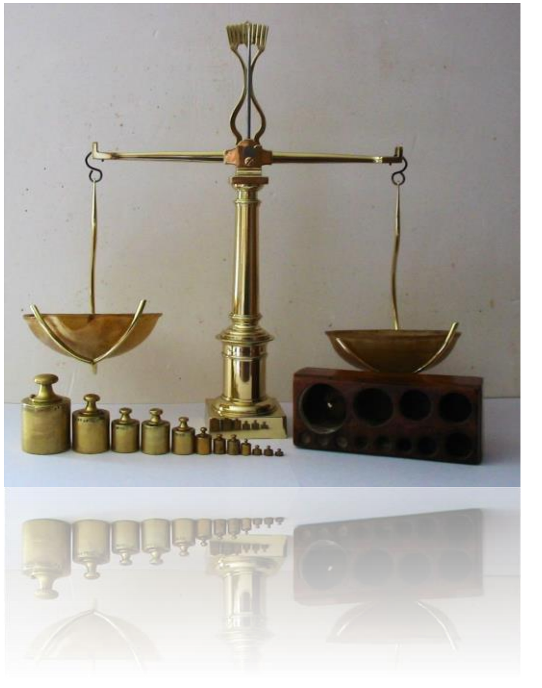
Algorithms and Data Structures: II

Rich Little
rlittle@uvic.ca
ECS 516

# Sorting revisited

- Overview of sorting algorithms

# Different types of Sorting Algorithms



- Comparison Based Sorting
  - sorting algorithm that sorts based only on comparisons
  - elements to be sorted must satisfy total order properties
- Integer Sorting
  - sorting algorithm that sorts a collection of data values by numeric keys, each of which is an integer

# Selection Sort

```
Algorithm selectionSort(A,n):
 Input: Array A of size n
 Output: Array A sorted

 for k ← 0 to n−2 do
    min ← k
    for j ← k+1 to n−1 do
        if A[j] < A[min] then
            min ← j
        end
    end
    swap(A[k], A[min])
 end
end
```

# Properties of Selection Sort

- *Comparisons:* Worst-case = best-case = average-case = $O(n^2)$
- exactly $n - 1$ swaps
- Simple implementation
- *Not adaptive*: The input order has no effect on the efficiency of the algorithm
- *In-place*: only requires a constant amount of (additional) memory

# Linear Insertion/Insertion Sort (Pseudocode)

```
Algorithm insertionSort(A,n):
  Input: Array A of size n
  Output: Array A sorted

  for k ← 1 to n-1 do
     val ← A[k]
     j ← k-1
     while j ≥ 0 and A[j] > val do
        A[j+1] ← A[j]
        j ← j - 1
     end
     A[j+1] = val
  end
end
```

# Complexity of Insertion Sort

- *Comparisons*:
  - Worst-case: $O(n^2)$
  - Best-case: $O(n)$
  - Average-case: $O(n^2)$
- O($n$) space
- *Swaps*:
  - Worst-case: $O(n^2)$
  - Best-case: $O(n)$

# Some Properties of Insertion Sort

- Simple implementation
- Efficient for
  - (quite) small data sets
  - data sets of already substantially sorted sequences: more efficient in practice than most other simple quadratic algorithms (*adaptive*)
- *In-place*: only requires a constant amount of (additional) memory
- *Online*: updates sort as elements are received
- *Natural*: people often use an insertion like sorting in practice

# Quicksort

```
Algorithm quickSort(A, a, b)
  Input Array A, ranks a and b
  Output A with the elements ordered between a and b
   if a ≥ b then return
   p ← A[b]
   l ← a
   r ← b - 1
  while l ≤ r do
    while l ≤ r and A[l] ≤ p do
      l ← l + 1
    while r ≥ l and A[r) ≥ p do
      r ← r - 1
    if l < r then
      swap(A[l], A[r])
  swap(A[l], A[b])
  quickSort(S, a, l - 1)
  quickSort(S, l + 1, b)
```

# Properties of Quicksort

- Worst case time complexity: $O(n^2)$
- Expected: $O(n \log(n))$
  - since, when picked at random, the pivot is likely good and splits up the sequence pretty evenly …
- Best-case: $O(n \log n)$
- Mergesort has $O(n \log(n))$ worst case time complexity, but quicksort is better in practice
- Usually the best choice unless the recursion gets too deep or stability is needed

# Mergesort (Pseudocode)

```
Algorithm mergeSort(S):
  input: Sequence S
  ouput: Sequence S sorted

  if S.size() < 2 then
     return S
  S1, S2 ← divide(S)
  S1 ← mergeSort(S1)
  S2 ← mergeSort(S2)
  S ← merge(S1, S2)
  return S
```

# Merge (Pseudocode)

```
Algorithm merge(S1, S2):
while not(S1.isEmpty() or S2.isEmpty()) do
  if S1.first().key() < S2.first().key() then
     S.insertLast(S1.removeFirst())
  else
     S.insertLast(S2.removeFirst()))
  end
end
while not(S1.isEmpty()) do
  S.insertLast(S1.removeFirst())
end
while not(S2.isEmpty()) do
  S.insertLast(S2.removeFirst())
end
return S
```

# Properties of Mergesort

- Worst-case = best-case = average case = $O(n \log n)$
- $O(n)$ extra space

# Heapsort (Pseudocode)

```
Algorithm heapSort(Array A, int n)
  input: array A of size n
  output: array A sorted

  heapify(A, n) //make A a maxHeap
  end ← n-1
  while end > 0 do
     swap(A[end],A[0])
     end ← end - 1
     bubbleDown(A,0,end)
```

# Properties of Heapsort

- Worst-case = best-case = average case = $O(n \log n)$
- In-place (constant extra space)
- Quick sort still outperforms it in most cases, why?

# Radixsort LSD

- Take the least significant digit (or group of bits) of each key.
- Group the keys based on that digit into buckets, but otherwise keep the original order of keys.
- Concatenate the buckets together in order.
- Repeat the grouping process with each more significant digit.

# Radixsort MSD

- Take the most significant digit of each key.
- Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
- Recursively sort each bucket, starting with the next digit to the right.
- Concatenate the buckets together in order.

# Properties of Radixsort

- Repeated sorting by means of Bucket Sort on *n* keys
  - For each component of the key perform one Bucket Sort
- Implement buckets (*N* of them) as queues
- Let the number of components per key be *d*

- **Theorem. The time complexity of Radix Sort is** $\boxed{O(d(n+N))}$ **or** $\boxed{O(dn)}$ **for large *n*.**

# Shellsort (Pseudocode)

```
Algorithm Shellsort (A,n)
   input: array A of size n
   output: A sorted

   h ← 1
   while h < n/3 do h ← 3h +1
   while h ≥ 1 do
      for i = h to n-1 do
         j ← i
         while j ≥ h and A[j] < A[j-h] do
            swap(A[j],A[j-h])
      h ← h/3
```

# Properties of Shellsort

- The analysis of Shellsort is dependent on the numeric properties of $n$, the size of $h$ and the step size and in some cases is still an open question

- This algorithm decreases $h$ by the sequence $\left(3^k - 1\right)/2$ which has a worst-case time of $O\left(n^{3/2}\right)$

- Is this better than $O(n \log n)$?

# Bubble Sort (Pseudocode)

```
Algorithm bubbleSort(A,n):
  Input: Array A of size n
  Output: Array A sorted

 repeat
   swapped ← false
   for i = 1 to n-1 do
     if A[i-1] > A[i] then
       swap(A[i-1],A[i])
       swapped ← true
   n ← n-1
 until not swapped
```

# Complexity of Bubble Sort

- Worst-case = average-case = $O(n^2)$
- Best-case = $O(n)$

# Shakersort

- Variation of bubble sort: sorts in both directions instead of just one as in bubble sort
- slightly better performance than bubble sort
- worse than insertion sort in practice

| | Type of Sorting Algorithm | Worst Case Time | Best Case Performance | Average Case Performance | Properties |
|---|---|---|---|---|---|
| **Insertion Sort** | Comparison Based Sorting | $O(n^2)$ | $O(n)$ | $O(n^2)$ | adaptive, in place, stable, online |
| **Bubblesort** | Comparison Based Sorting | $O(n^2)$ | $O(n)$ | $O(n^2)$ | in place |
| **Selection Sort** | Comparison Based Sorting | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | in place |
| **Binary Insertion** | Comparison Based Sorting | $O(n^2)$ | $O(n)$ | $O(n^2)$ | adaptive, in place |
| **Shakersort** | Comparison Based Sorting | $O(n^2)$ | $O(n)$ | $O(n^2)$ | stable, in place |
| **Shellsort** | Comparison Based Sorting | $O(n^2)$ | $O(n \log n)$ | | in place |
| **Quicksort** | Comparison Based Sorting | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ | in place |
| **Heapsort** | Comparison Based Sorting | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | in place |
| **Mergesort** | Comparison Based Sorting | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | not in place |
| **Bucketsort** | Integer Sorting | $O(n+k)$ | | $O(n+k)$ | can be implemented such that stable |
| **Radixsort** | Integer Sorting | $O(dn)$ | | | stable |

# Sorting out Sorting

- Visualization and Comparison of Sorting Algorithms
- https://www.youtube.com/watch?v=ZZuD6iUe3Pc

# How fast can we sort? A lower bound for comparison based sorting

- We prove: using a comparison based sorting algorithm, we cannot do better than $O(n \log(n))$ worst case time complexity
- Therefore, $\boldsymbol{\Omega(n \log(n))}$ denotes the **lower bound** for comparison based sorting

## **Theorem:** No comparison based sorting algorithm for $n$ distinct elements has a worst case running time that is better than $O(n \log n)$

Proof:

- Consider a sequence $S$ containing $n$ distinct elements, say $x_0, x_1, x_2, \ldots, x_{n-1}$
- To decide the order of elements, a comparison-based algorithm compares elements pairwise—a sufficient number of times
- In particular, to decide which element of $x_i$ and $x_j$ is smaller, it answers "is $x_i < x_j$?"
- Depending on the outcome—i.e., yes or no—the algorithm performs either no further comparisons or it continues with more comparisons

# Proof (continued)

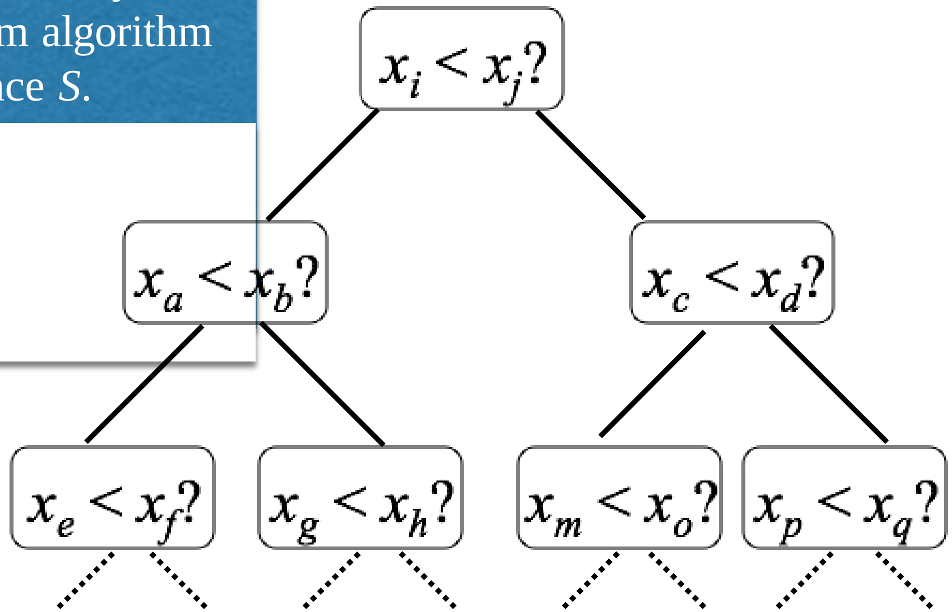
- We want to know: how good is the best of all comparison-based sorting algorithms? (Let's call it the *optimal* algorithm)
- This optimal sorting algorithm requires a certain number of comparisons (at least) to sort *any* sequence (not just the easiest input)
- We ask: How many comparisons are required for an optimal sorting algorithm to sort *n* elements?

# Proof (continued)

- How many comparisons are required for an optimal sorting algorithm to sort $n$ elements?
- We consider our sequence $S$: $x_0, x_1, x_2, \ldots, x_{n-1}$
- The optimal algorithm will pick two elements for a first comparison, and, based on the outcome choose a second, etc.
- This can be depicted in a decision tree

# Decision tree of an optimal sorting algorithm that is sorting a general sequence of elements

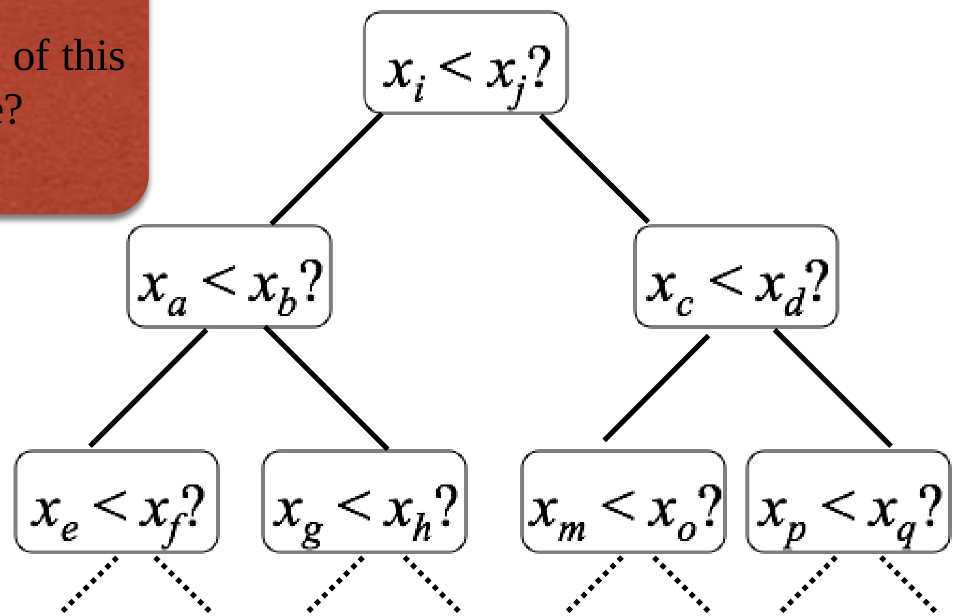
- The decision tree contains every possible path the optimum algorithm might take to sort sequence *S*.
- How many leaves does the tree have at least?




Since we don't know what *S* looks like, any permutation of *S* could be the sorted one. Thus, every permutation of *S* has to be represented by a path from the root to a leaf in the decision tree. Therefore:

# Decision tree of an optimal sorting algorithm sorting a general sequence of elements

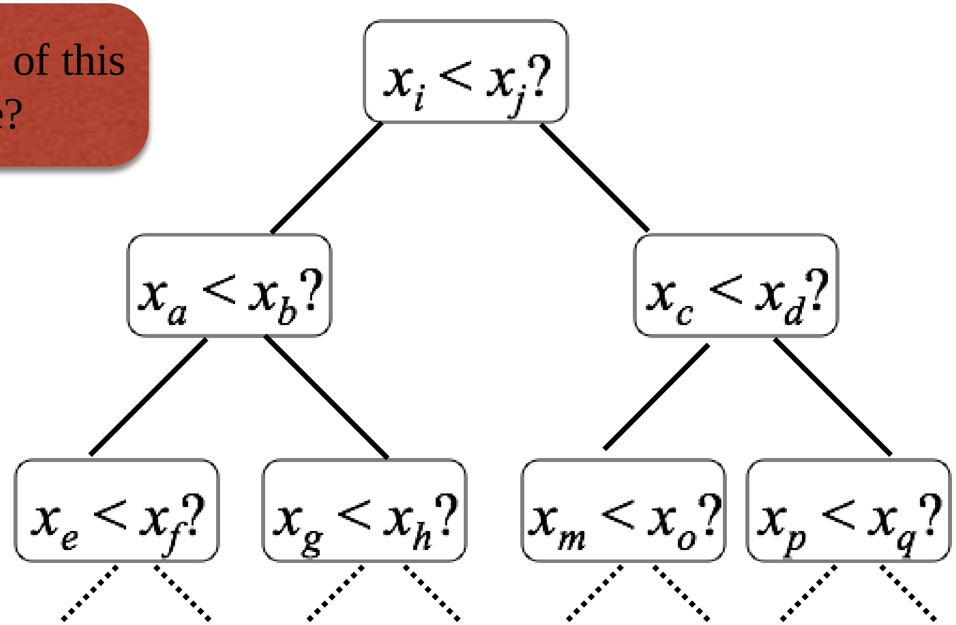
What is the height of this decision tree?




# leaves $\geqq n!$

# Decision tree of an optimal sorting algorithm sorting a general sequence of elements

What is the height of this decision tree?




- # leaves ≧ $n$!
- tree is binary
- height is ≧ log($n$!); can't be less, since the shortest binary tree that has $n$! leaves has a height of log($n$!) [definition of log]

# Proof (continued)

- Since the height of the tree is at least log($n$!), we know that
  - at least log($n$!) worst case comparisons are required by an optimal comparison based sorting algorithm
  - at least log($n$!) worst case comparisons are required by any comparison based algorithm

# Proof (continued)

- What is $\Omega(\log(n!))$?

- $\log(n!) \geqq \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)$
  $\geqq \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n/2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)$
  $\geqq \log((n/2) \cdot \ldots \cdot (n/2))$
  $\geqq \log((n/2)^{(n/2)}) = (n/2) \log (n/2)$

- and therefore $\log(n!) \in \Omega(n \log(n))$

- We conclude: there is no comparison-based sorting algorithm that has a worst-case time complexity that is better than $O(n \log(n))$