Sorting
The Sorting Problem

• Input is a sequence of \( n \) items \((a_1, a_2, \ldots, a_n)\)
• The mapping we want is determined by a “comparison” operation, denoted by \( \leq \)
• Output is a sequence \((b_1, b_2, \ldots, b_n)\) such that:
  - \( \{a_1, a_2, \ldots, a_n\} = \{b_1, b_2, \ldots, b_n\} \)
    (i.e. output is a permutation of the input sequence)
  - \( b_1 \leq b_2 \leq \ldots \leq b_n \)
Solutions so far…

- Insertion sort
  - Requires $O(n^2)$ comparisons in worst/average case
- Bubble Sort (homework)
  - Requires $O(n^2)$ comparisons in all cases
- Merge Sort
  - Requires $O(n \log n)$ comparisons in the worst case
  - Unfortunately, must copy data

Can we sort in place in $O(n \log n)$ time?
A Useful Data Structure: Heaps

- Complete binary tree representation of an array
  - Well, nearly complete… (array lengths not exactly $2^h$)
  - Notation: Data contained in an array $A[1..n]$
    - $\text{Parent}(i) = \lfloor i / 2 \rfloor$, $\text{Left}(i) = 2i$, $\text{Right}(i) = 2i + 1$
- Max Heap Property: $A[i] \leq A[\text{Parent}(i)]$
  - $\text{Build-Max-Heap}$ imposes the property on unsorted $A$
    - Runs in time $O(n)$
  - $\text{Max-Heapify}$ is a subroutine called by $\text{Build-Max-Heap}$
    - Runs in time $O(\log n)$
- Read the textbook for info on Priority Queues
The Max-Heapify Subroutine

Max-Heapify(A[1..n], i )
    L = left(i), R = right(i)
    largest = i
        largest = L
    if R ≤ n and A[R] ≰ A[largest]
        largest = R
    if largest ≠ i
        exchange A[i] with A[largest]
    Max-Heapify(A[1..n], largest)
Building a Max Heap

• Start from the “leaves” and build upwards
  ▪ No need to actually heapify the leaves, as they are already valid max heaps… so we will skip them

Build-Max-Heap(A[1..n])
  for i = ⌊ n/2 ⌋ downto 1
  Max-Heapify(A, i)
Heapsort

• Can use Max-Heaps to sort efficiently, in-place

Heapsort(A[1..n])
  Build-Max-Heap(A)  // O(n)
  for i = n downto 2
    Max-Heapify(A[1..i-1], 1)  // O(log n)

• Worst case runtime is O(n log n) (due to the loop)
• See the textbook for detailed proofs and analysis
Quicksort

• Often performs best in the real world
  ▪ Small constants + many optimized implementations
• Uses divide and conquer, like Mergesort
• Unlike Mergesort, can sort in place

QuickSort(A[1..n])
  if n > 1
    q = Partition(A[1..n])
    QuickSort(A[ 1 .. q-1 ])
    QuickSort(A[ q+1 .. n ])

• The Partition routine does all the work…
Partitioning

Partition(A[1..n])

\[
x = A[n], \ i = 0 \quad \text{// } x \text{ is known as the “pivot”}
\]

for j = 1 to n-1

\[
\text{if } A[j] \leq x \quad \text{// Other elements compared to the pivot}
\]

\[
i = i + 1
\]

exchange A[i] with A[j]

exchange A[i+1] with A[n]

return i+1

- Observe that \(O(n)\) comparisons are performed
- If \(q = \text{Partition}(A[1..n])\) then \(A[1..q-1] \leq A[q] \leq A[q+1..n]\)
Quicksort: Correctness

Quicksort(A[1..n])
  if n > 1
    q = Partition(A[1..n])
    Quicksort(A[1..q-1])
    Quicksort(A[q+1..n])

• Assuming Partition is correct, can argue correctness of Quicksort via induction, with base case n = 1
  ▪ Inductive Hypothesis: Quicksort correctly sorts up to size n-1
  ▪ Use correctness of Partition to complete the inductive step:
    If q=Partition(A[1..n]) then A[1..q-1] ≤ A[q] ≤ A[q+1..n]
• Detailed analysis of Partition given in CLRS
QuickSort: The Worst Case

QuickSort(A[1..n])
  if n > 1
    q = Partition(A[1..n])
    QuickSort(A[ 1 .. q-1 ])
    QuickSort(A[ q+1 .. n ])
  
  \[ T(n) = \max_{1 \leq q \leq n} [T(q-1) + T(n-q)] + O(n) \]
  
  Can use proof by induction to show this \(O(n^2)\)
  
  Alternatively, use a graphical argument…
QuickSort: Expected Case (1)

QuickSort(A[1..n])
  if n > 1
    q = Partition(A[1..n])
    QuickSort(A[1..q-1])
    QuickSort(A[q+1..n])

• Method 1: Use indicator random variables
  ▪ Let $z_1 \leq z_2 \leq \ldots \leq z_n$ denote the sorted elements of A
  ▪ Let $X_{ij}$ indicate a comparison between $z_j$ and $z_i$
  ▪ Total number of comparisons: $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
  ▪ Find $E[X]$ by computing $Pr\{X_{ij}\}$ (see CLRS for details)
QuickSort(A[1..n])
  if n > 1
    q = Partition(A[1..n])
    QuickSort(A[ 1 .. q-1 ])
    QuickSort(A[ q+1 .. n ])

• Method 2: Solve the recurrence for expected case
  ▪ Observe that pivot location will be random
  ▪ Observe that recursive inputs are also randomly ordered
  \[ T(n) = O(n) + \left( \sum_{i=0}^{n-1} T(i) \right) / n \]
  = \( O(n) + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \)
  ▪ Solve using Telescoping and Transformations (somewhat hard)
    • Take notes! This will help you for the homework.
Quicksort: Expected Case (3)

QuickSort(A[1..n])
if n > 1
    q = Partition(A[1..n])
    QuickSort(A[ 1 .. q-1 ])
    QuickSort(A[ q+1 .. n ])

• Method 3: Recursion Trees
  ▪ Much easier, but still needs some stuff from probability
  ▪ Not done in CLRS… see the handout to be distributed in class

• Conclusion: Expected runtime is $O(n \log n)$
Lower Bounds for Sorting

• The worst case runtime for our sorting algorithms so far was $O(n \log n)$ or worse. Can we do better?
  ▪ What is the best that can be done? Should we even try?
• Assume that we can only compare by using the $\leq$ operator. How many times must we apply it, in the worst case, if we always want a correct output?
• Observation: There are $n!$ possible input orderings
• Observation: The $\leq$ operator has a binary output
• Consider the “decision tree” of the program
Decision Tree using Comparisons

• Result of each \( \leq \) operation determines a single “left” or “right” branch in the program flow
• Model program branching with a tree
  ▪ Only two choices per operation, so it is a binary tree
• Final outputs (leaves) are the sorting permutations applied to the input sequence. Requires at least \( n! \) leaves to correctly sort each of the \( n! \) possible input orderings
• The height of the tree is the number of comparisons performed in the worst case. What is the minimum height of a binary tree with \( n! \) leaves?
  ▪ A binary tree of height \( h \) has at most \( 2^h \) leaves, and we require at least \( n! \) leaves, so we have \( 2^h \geq n! \), and thus \( h = \Omega(n \log n) \)
• Thus, worst case requires \( h = \Omega(n \log n) \) comparisons
Breaking the $\Omega(n \log n)$ Barrier

- **Counting Sort**
  - Runtime is always $\Theta(n)$.
  - Sorts integers in the range $0..k$ for $k = O(n)$

- **Bucket Sort (aka Binsort)**
  - Expected runtime is $\Theta(n)$
  - Sorts floats uniformly distributed over a fixed range

- **Radix Sort**
  - Runs in $\Theta(n)$ time
  - Sorts $d$-digit numbers, where each digit is in the range $0..k-1$, for any constant $d$ and for $k = O(n)$
Counting Sort

Counting-Sort(A[1..n], k)

\[ B[1..n], C[0..k] = \{0, \ldots, 0\} \]

for j = 1 to n

\[ C[A[j]] = C[A[j]] + 1 \]

for i = 1 to k

\[ C[i] = C[i] + C[i-1] \]

for j = n downto 1


\[ C[A[j]] = C[A[j]] - 1 \]

return B[1..n]
Bucket Sort

• Without much loss of generality, assume inputs are in the interval \([0,1)\)

Bucket-Sort(A[1..n])

Let B[0..n-1] be an array of (initially empty) lists
for i = 1 to n
  insert A[i] into B[\lfloor n \times A[i] \rfloor]
for i = 0 to n-1
  sort list B[i] with insertion sort
return concatenation of sorted lists B[1], … , B[n-1]

• If inputs are uniformly distributed, expect \(\Theta(n)\) runtime
Radix Sort

• Simple, but counter-intuitive algorithm
  ▪ Used for sorting d-digit numbers written in base k
  ▪ Naming convention: digit 1 is the LEAST significant

Radix-Sort(A[1..n], d)
  for i = 1 to d
    use a stable sort to sort A[1..n] on digit i

• Runtime depends on which stable sort is used
  ▪ Can use a counting sort, and if k=O(n) that will take O(n) time
    per iteration of the loop
  ▪ Since d is a constant, looping d times is irrelevant to asymptotic
    runtime. Thus we can achieve an O(n) overall runtime.