Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017
Outline of Today’s Class

- Sorting in $O(n \lg n)$ Time: Heapsort
Heapsort

- Desired property of Mergesort: Time complexity of $O(n \lg n)$
- Desired property of Insertion sort: Sorting in place
- Heapsort combines these two properties
- Heapsort illustrates a powerful algorithm design technique: a data structure organizes information during execution
Heap Data Structure

- Array object regarded as nearly complete binary tree
- Attributes:
  - length(A): # elements in A
  - heap_size(A): # elements in heap
- Heap property: value of parent ≥ values of children
- Filled from root to leaves, left to right
  - Root of tree is stored in A[1]
  - Given index i of a node:
    - parent: PARENT(i) ← \lfloor i/2 \rfloor
    - left child: LEFT(i) ← 2i
    - right child: RIGHT(i) ← 2i + 1
Heaps as a Balanced Binary Tree

- In a tree:
  - Depth of a node is distance of node from root
  - Depth of a tree is depth of deepest node
  - Height is the opposite of depth
  - Height and depth are often confused

- A binary tree of depth $d$ is balanced if all nodes at depths 0 through $d - 2$ have two children

- Illustration of balanced and unbalanced binary trees:
A balanced binary tree of depth $d$ is left-justified if:

1. It has $2^k$ nodes at depth $k \ \forall k < d$
2. All leaves at depth $d$ are as far left as possible
Back to the Heap Property

- In a max-heap: $A[\text{PARENT}(i)] \geq A[i], \forall i \geq 1$
- In a min-heap: $A[\text{PARENT}(i)] \leq A[i], \forall i \geq 1$
- We will focus on the heap property in max-heaps for sorting:

Blue node has heap property

Blue node has heap property

Blue node does not have heap property

- Leaf nodes automatically have the heap property. Why?
- A binary tree is a heap if all nodes in it have the heap property
- What can you say about the root in a max/min-heap?
Maintaining the Heap Property in a Max-heap

- Given a node that does not have the heap property, one can give it the heap property by exchanging its value with that of the larger child:

  ![Diagram of heap property maintenance](image)

  Blue node does not have heap property
  Blue node has heap property

- Note: Upon the exchange, the heap property may be violated in the subtree rooted at the child.

- The MAX – HEAPIFY subroutine restores the heap property on the subtree rooted at index $i$.

- How? The value at $A[i]$ is floated down in the max-heap.
MAX-HEAPIFY(array A, index i)

1: if i is not a leaf and A[LEFT(i)] or A[RIGHT(i)] > A[i] then
2: let k denote larger child
3: swap(A[i], A[k])
4: MAX-HEAPIFY(A, k)

Time Complexity: MAX-HEAPIFY
- Let H(i) denote running time
- Show H(i) ∈ O(lgn)
- Hint: Down the tree we go
BUILD-MAX-HEAP(A, size n)

1: for \( i \leftarrow \left\lfloor \frac{n}{2} \right\rfloor \) to 1 do
2: \hspace{1em} MAX-HEAPIFY(A, i)

Time: BUILD-MAX-HEAP

- Why is \( \left\lfloor \frac{n}{2} \right\rfloor \) bound sufficient?
- Hint: \# internal nodes in heap?
- Let \( B(n) \) be running time
- Show \( B(n) \in O(n \log n) \)

Figure: Trace on above array.
1. Show that an \( n \)-element heap has depth (and height) \( \lfloor \log n \rfloor \).

2. Show that there are at most \( \lceil n/2^{h+1} \rceil \) nodes of height \( h \).

\[
B(n) = \sum_{i=1}^{n/2} H(i) = \sum_{h=0}^{\lfloor \log n \rfloor} \{ \lfloor n/2^{h+1} \rfloor O(h) \}
\in O(n \sum_{h=0}^{\lfloor \log n \rfloor} h \cdot 2^h)
\]

Note: \( \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} \) (A.8)

Hence:

\[
O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)
\]

Conclusion: A heap can be built in \( O(n) \) time.
HEAPSORT(A)

1: BUILD-MAX-HEAP(A)
2: for $i \leftarrow A\.length$ to 2 do
3: swap([A[1], A[i]])
4: A.heap-size $\leftarrow$ A.heap-size - 1
5: MAX-HEAPIFY(A, 1)

Basic Idea Behind HEAPSORT

- What property holds for root after BUILD-MAX-HEAP?
- Why can we put it at index $i$?
- Why do we need to run MAX-HEAPIFY after swap?

Time: HEAPSORT

- HEAPSORT takes $O(n \lg n)$.
- BUILD-MAX-HEAP runs in linear time - $O(n)$
- There are $n - 1$ calls to MAX-HEAPIFY
- Each call takes $O(\lg n)$ time
- So: $T(\text{HEAPSORT}(A, n)) \in O(n + (n - 1)\lg n) \in O(n \lg n)$
An Important Application of Heaps

- A priority queue maintains a set $S$ of elements, each one associated with a key.
- Max- or min-priority queues help to rank jobs for scheduling.
- Operations for a max-priority queue:
  1. `INSERT(S, x)` inserts element $x$ in the set $S$.
  2. `MAXIMUM(S)` returns the element of $S$ with the largest key.
  3. `EXTRACT-MAX(S)` removes from $S$ and returns the element with the largest key.
  4. `INSERT-KEY(S, x, k)` increases the value of the key of $x$ to the new value $k$, which is assumed to be at least as large as the current value of the key of $x$.
How Fast Can We Sort?

- The sorting algorithms we have seen so far are insertion sort, mergesort, heapsort, and quicksort
- All these sorting algorithms are comparison sorts
- They rely on comparisons to determine the relative order of elements
- The best worst-case running time that we have seen for comparison sorting is $O(n \cdot \log n)$
- \textbf{Is $O(n \cdot \log n)$ the best we can do?}
- We need to employ decision trees to answer this question
Reason for Employing a Decision Tree

Sort \( \langle a_1, a_2, \ldots, a_n \rangle \):

Each internal node is labeled \( i : j \) for \( i, j \in \{1, 2, \ldots, n\} \)

- The left subtree shows subsequent comparisons if \( a_i \leq a_j \)
- The right subtree shows subsequent comparisons if \( a_i > a_j \)
Example of a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle = \langle 9, 4, 6 \rangle$:

Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$
Sort $\langle a_1, a_2, \ldots, a_n \rangle = \langle 9, 4, 6 \rangle$:

Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$
Example of a Decision Tree

Sort \( \langle a_1, a_2, \ldots, a_n \rangle = \langle 9, 4, 6 \rangle \):

Each internal node is labeled \( i : j \) for \( i, j \in \{1, 2, \ldots, n\} \)

- The left subtree shows subsequent comparisons if \( a_i \leq a_j \)
- The right subtree shows subsequent comparisons if \( a_i > a_j \)
Sort $\langle a_1, a_2, \ldots, a_n \rangle = \langle 9, 4, 6 \rangle$:

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots \pi(n) \rangle$ which establishes the ordering $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$.
A decision tree can model the execution of any comparison sort:

- One tree for each input size \( n \)
- View the algorithm as splitting the tree whenever it compares two elements
- The tree contains the comparisons along all possible instruction traces
- The running time of the algorithm is then the length of the actual path taken
- Worst-case running time is the height of the tree
Theorem: Any decision tree that can sort $n$ elements must have height $\Omega(n \cdot \lg n)$

Proof:

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.

A height $h$ binary tree has $\leq 2^h$ leaves

Hence, $n! \leq 2^h$

$$h \geq \lg(n!)
\geq \lg((n/e)^n) \quad \text{– Stirling’s approximation}
= n \cdot \lg n - n \cdot \lg e
\in \Omega(n \cdot \lg n)$$

Corollary: Heapsort and mergesort are asymptotically optimal comparison-based sorting algorithms
1 Outline of Today’s Class

2 Sorting in Linear Time
   - Counting Sort
   - Radix Sort
We can sort faster than $O(n \cdot \log n)$ if we do not compare the items being sorted against each other.

We can do this if we have additional information about the structure of the items.

Examples of Sorting Algorithms that do not compare items:

1. Counting Sort
2. Radix Sort
3. Bucket Sort
Counting Sort: Basic Idea and Pseudocode

**Input:** $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$

**Output:** $B[1 \ldots n]$ sorted

**Auxiliary storage:** $C[1 \ldots k]$

**Note:** all elements are in $\{1, 2, \ldots, k\}$

**Basic Idea:** Count the number of 1’s, 2’s, \ldots, $k$’s.

COUNTINGSORT($A$, $n$)

1. for $i \leftarrow 1$ to $k$ do
2. $C_i \leftarrow 0$
3. for $j \leftarrow 1$ to $n$ do
4. $C[A[j]] \leftarrow C[A[j]] + 1$
   $\Downarrow C[i] = |\{\text{key} = i\}|$
5. for $i \leftarrow 2$ to $k$ do
6. $C[i] \leftarrow C[i] + C[i - 1]$
   $\Downarrow C[i] = |\{\text{key} \leq i\}|$
7. for $j \leftarrow n$ to 1 do
Counting Sort: Trace

A: 4 1 3 4 3
B: 

C: 1 2 3 4
Counting Sort: Trace

```
for i ← 1 to k
    do C[i] ← 0
```
Outline of Today's Class

Sorting in Linear Time

Counting Sort

Radix Sort

Counting Sort: Trace

\[
A: \begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 \\
4 & \quad 1 & \quad 3 & \quad 4 & \quad 3
\end{align*}
\]

\[
C: \begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 \\
0 & \quad 0 & \quad 0 & \quad 1
\end{align*}
\]

\[
B: \quad \text{(empty)}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|
\]
Counting Sort: Trace

\[
\begin{align*}
A: & & 1 & 2 & 3 & 4 & 5 \\
& & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & & \\
C: & & 1 & 2 & 3 & 4 \\
& & 1 & 0 & 0 & 1 \\
\end{align*}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|
\]
Outline of Today’s Class
Sorting in Linear Time

Counting Sort
Radix Sort

Counting Sort: Trace

\[
\begin{align*}
A & : \quad 4 \quad 1 \quad 3 \quad 4 \quad 3 \\
B & : \quad \quad \quad \quad \quad \quad \quad \quad \\
C & : \quad 1 \quad 0 \quad 1 \quad 1 \quad 1
\end{align*}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|
\]
Counting Sort: Trace

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 0 & 1 & 2 & \\
\end{array}
\]

for \( j \leftarrow 1 \) to \( n \)
\[
do C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}| 
\]
Counting Sort: Trace

\[
\begin{align*}
A: & \begin{array}{c}
4 \quad 1 \quad 3 \quad 4 \quad 3
\end{array} \\
B: & \begin{array}{c}
\quad \quad \quad \quad \quad \quad \quad
\end{array} \\
C: & \begin{array}{c}
1 \quad 0 \quad 2 \quad 2
\end{array}
\end{align*}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \ C[i] = |\{\text{key} = i\}|
\]
Counting Sort: Trace

\begin{align*}
A: & \quad 4 \quad 1 \quad 3 \quad 4 \quad 3 \\
B: & \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \\
C: & \quad 1 \quad 0 \quad 2 \quad 2 \\
C': & \quad 1 \quad 1 \quad 2 \quad 2
\end{align*}

\textbf{for} \ i \leftarrow 2 \ \textbf{to} \ k \\
\textbf{do} \ C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \ C[i] = |\{\text{key} \leq i\}|
Counting Sort: Trace

A: 4 1 3 4 3
B: 
C: 1 0 2 2

for $i \leftarrow 2 \text{ to } k$

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

$C[i] = |\{\text{key } \leq i\}|$
Counting Sort: Trace

\[
\begin{align*}
A & : 4 & 1 & 3 & 4 & 3 \\
\text{for } i \leftarrow 2 \text{ to } k & \\
\text{do } C[i] & \leftarrow C[i] + C[i-1] & \quad \triangleright C[i] = |\{\text{key} \leq i\}| \\
B & : & & & & \\
C & : 1 & 0 & 2 & 2 \\
\text{C'} & : 1 & 1 & 3 & 5
\end{align*}
\]
Counting Sort: Trace

\[
\begin{align*}
A: & \begin{array}{c}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3
\end{array} & \quad C: & \begin{array}{c}
1 & 2 & 3 & 4 \\
1 & 1 & 3 & 5
\end{array} \\
B: & \begin{array}{c}
1 & 2 & 3 & 4 & 5 \\
& 3 & & & \\
\end{array} & \quad C: & \begin{array}{c}
1 & 1 & 2 & 5 \\
\end{array}
\end{align*}
\]

\[
\text{for } j \leftarrow n \text{ downto } 1 \\
\text{do } B[C[A[j]]] \leftarrow A[j] \\
C[A[j]] \leftarrow C[A[j]] - 1
\]
Counting Sort: Trace

for $j \leftarrow n$ downto 1
  do $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$
Counting Sort: Trace

for $j ← n$ down to 1
   do $B[C[A[j]]] ← A[j]$
   $C[A[j]] ← C[A[j]] - 1$
Counting Sort: Trace

for \( j \leftarrow n \) downto 1
  do \( B[C[A[j]]] \leftarrow A[j] \)
      \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort: Trace

for $j \leftarrow n$ downto 1
do $B[C[A[j]]] \leftarrow A[j]
    C[A[j]] \leftarrow C[A[j]] - 1
Counting Sort: Running Time Analysis

\[ \Theta(k) \begin{cases} 
\text{for } i \leftarrow 1 \text{ to } k \\
\quad \text{do } C[i] \leftarrow 0 
\end{cases} \]

\[ \Theta(n) \begin{cases} 
\text{for } j \leftarrow 1 \text{ to } n \\
\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 
\end{cases} \]

\[ \Theta(k) \begin{cases} 
\text{for } i \leftarrow 2 \text{ to } k \\
\quad \text{do } C[i] \leftarrow C[i] + C[i-1] 
\end{cases} \]

\[ \Theta(n) \begin{cases} 
\text{for } j \leftarrow n \text{ downto } 1 \\
\quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
\quad \quad C[A[j]] \leftarrow C[A[j]] - 1 
\end{cases} \]

\[ \Theta(n + k) \]
Counting Sort: Running Time Analysis

If \( k \in O(n) \), then counting sort takes \( O(n) \) time.

- But sorting takes \( \Omega(n \cdot \log n) \) time!
- Where is the contradiction?
If $k \in O(n)$, then counting sort takes $O(n)$ time.

- But sorting takes $\Omega(n \cdot \log n)$ time!
- Where is the contradiction?

- *Comparison sorting* takes $\Omega(n \cdot \log n)$
- Counting sort is *not* a comparison sort
- Not a single comparison occurs in counting sort
If \( k \in O(n) \), then counting sort takes \( O(n) \) time.

- But sorting takes \( \Omega(n \cdot \log n) \) time!
- Where is the contradiction?

- *Comparison sorting* takes \( \Omega(n \cdot \log n) \)
- Counting sort is *not* a comparison sort
- Not a single comparison occurs in counting sort
Counting sort is a stable sort because it preserves the input order among equal elements.

What other sorting algorithms have this property?
Radix Sort

- History: Herman Hollerith’s card-sorting machine for the 1890 US Census.
- Radix sort is digit-by-digit sort
- Hollerith’s original (wrong) idea was to sort on most significant digit first
- The final (correct) idea was to sort on the least significant digit first with an auxiliary stable sort
Radix Sort: Correctness

- The proof is by induction on the digit position
- Assume that the numbers are already sorted by their low-order $t - 1$ digits
- Sort on digit $t$
The proof is by induction on the digit position.

Assume that the numbers are already sorted by their low-order \( t - 1 \) digits.

Sort on digit \( t \).

- Two numbers that differ in digit \( t \) are correctly sorted.
Radix Sort: Correctness

- The proof is by induction on the digit position
- Assume that the numbers are already sorted by their low-order \( t - 1 \) digits
- Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted
  - Two numbers equal in digit \( t \) are put in the same order as the input - the correct order
Radix Sort: Running Time Analysis

- Assume counting sort is the auxiliary stable sort
- Sort \( n \) computer words of \( b \) bits each
- Each word can be viewed as having \( b/r \) base-2\(^r\)

\[
\begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array}
\]

**Figure:** Example of a 32-bit word

- \( r = 8 \) means \( b/r = 4 \) passes of counting sort on base-2\(^8\) digits
- \( r = 16 \) means \( b/r = 2 \) passes on base-2\(^{16}\) digits
- How many passes should one make?
Note: Counting sort takes $\theta(n + k)$ time to sort $n$ numbers in the range 0 to $k - 1$.

If each $b$-bit word is broken into $r$-bit pieces, each pass of counting sort takes $\theta(n + 2^r)$ time. Since there are $b/r$ passes, we have:

$$T(n, b) \in \theta\left(\frac{b}{r}(n + 2^r)\right)$$

Choose $r$ to minimize $T(n, b)$

- Increasing $r$ means fewer passes, but as $r >> \lg n$, the time grows exponentially
Minimize $T(n, b)$ by differentiating and setting the first derivative to 0. Recall that this is the technique to find minima or maxima for a function.

Alternatively, observe that we do not want $2^r \gg n$, and so we can safely choose $r$ to be as large as possible without violating this constraint.

Choosing $r = lgn$ implies that $T(n, b) \in \theta(bn/lgn)$

- For numbers in the range 0 to $n^d - 1$, we have that $b = d \cdot lgn$
- Hence, radix sort runs in $\theta(d \cdot n)$ time
In practice, radix sort is fast for large inputs, as well as simple to implement and maintain.

**Example:** 32-bit numbers
- At most 3 passes when sorting \( \geq 2000 \) numbers
- Mergesort and quicksort do at least \( \lceil \log 2000 \rceil \) passes

**Not all Rosy:**
- Unlike quicksort, radix sort displays little locality of reference
- A well-tuned quicksort does better on modern processors that feature steep memory hierarchies