COT 5407: Introduction to Algorithms

Giri Narasimhan

ECS 254A; Phone: x3748
giri@cis.fiu.edu

http://www.cis.fiu.edu/~giri/teach/5407S17.html
https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494
Figure 8.10  Quicksort
Partition Algorithm

- Pick a pivot
- Compare each item to a pivot and create two lists:
  - L = list of all items smaller than the pivot
  - R = list of all items larger than the pivot
- One scan through the list is enough, but seems to need extra space
- How to design an **in-place partition algorithm**!
QuickSort\( (array\ A, int\ p, int\ r) \)

1 \hspace{1em} if \hspace{1em} (p < r)

2 \hspace{1em} then \hspace{1em} q \leftarrow \text{Partition}(A, p, r)

3 \hspace{1em} \text{QuickSort}(A, p, q - 1)

4 \hspace{1em} \text{QuickSort}(A, q + 1, r)

To sort array call \text{QuickSort}(A, 1, length[A]).

\text{Partition}(array\ A, int\ p, int\ r)

1 \hspace{1em} x \leftarrow A[r] \quad \triangleright \text{Choose pivot}

2 \hspace{1em} i \leftarrow p - 1

3 \hspace{1em} \text{for} \hspace{1em} j \leftarrow p \text{ to } r - 1

4 \hspace{1em} \hspace{1em} \text{do if} \hspace{1em} (A[j] \leq x)

5 \hspace{1em} \hspace{1em} \hspace{1em} \text{then} \hspace{1em} i \leftarrow i + 1

6 \hspace{1em} \hspace{1em} \text{exchange} \hspace{1em} A[i] \leftrightarrow A[j]

7 \hspace{1em} \text{exchange} \hspace{1em} A[i + 1] \leftrightarrow A[r]

8 \hspace{1em} \text{return} \hspace{1em} i + 1
Time Complexity

Recurrence Relation
- \( T(N) = O(N) + T(N_1) + T(N_2) \)

Average-Case Time Complexity
- On the average, \( N_1 = N_2 = N/2 \)
- \( T(N) = O(N) + 2T(N/2) \)
- Thus, average-case complexity = \( O(N \log N) \)

Worst-Case Time Complexity
- Worst-case: Either \( N_1 \) or \( N_2 = 0 \)
  - Thus, \( T(N) = O(N) + T(N - 1) \)
  - \( T(N) = O(N^2) \)
Variants of QuickSort

• **Choice of Pivot**
  - Random choice
  - Median of 3
  - Median

• **Avoiding recursion on small subarrays**
  - Invoking InsertionSort for small arrays
Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort
**Definitions**

**Abstract Problem**: defines a function from any allowable input to a corresponding output

**Instance of a Problem**: a specific input to abstract problem

**Algorithm**: well-defined computational procedure that takes an instance of a problem as input and produces the correct output

*An Algorithm must halt on every input with correct output.*
Algorithm Analysis

- **Worst-case time complexity**
  - Worst possible time of all input instances of length N
- **(Worst-case) space complexity**
  - Worst possible space of all input instances of length N
- **Average-case time complexity**
  - Average time of all input instances of length N
Upper and Lower Bounds

- **Time Complexity of a Problem**
  - **Difficulty**: Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution**: Define upper bounds and lower bounds within which the time complexity lies.

- **What is the upper bound on time complexity of sorting?**
  - **Answer**: Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point**: Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.

- **What is the lower bound on time complexity of sorting?**
  - **Difficulty**: If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts $N$ items can run in worst-case time $o(f(N))$. 
Surprisingly, it is possible to prove lower bounds for many comparison-based problems.

For any comparison-based problem, for any input of length $N$, if there are $P(N)$ possible solutions, then any algorithm must need $\log_2(P(N))$ to solve the problem.

Binary Search on a list of $N$ items has at least $N + 1$ possible solutions. Hence lower bound is

- $\log_2(N+1)$.

Sorting a list of $N$ items has at least $N!$ possible solutions. Hence lower bound is

- $\log_2(N!) = O(N \log N)$

Thus, **MergeSort is an optimal algorithm.**

- Because its worst-case time complexity equals lower bound!
Beating the Lower Bound

- **Bucket Sort**
  - Runs in time $O(N+K)$ given $N$ integers in range $[a+1, a+K]$.
  - If $K = O(N)$, we are able to sort in $O(N)$.
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.

- **Radix Sort**
  - Runs in time $O(d(N+K))$ given $N$ items with $d$ digits each in range $[1,K]$.

- **Counting Sort**
  - Runs in time $O(N+K)$ given $N$ items in range $[a+1, a+K]$.
Bucket Sort

- **N integer** values in the range \([a..a+m-1]\)
- For e.g., sort a list of 50 scores in the range \([0..9]\).
- **Algorithm**
  - Make \(m\) buckets \([a..a+m-1]\)
  - As you read elements throw into appropriate bucket
  - Output contents of buckets \([0..m]\) in that order
- **Time** \(O(N+m)\)
- **Warning**: This algorithm cannot be used for “infinite-precision” real numbers, even if the range of values is specified.
Stable Sort

- A sort is **stable** if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!
Algorithm

for i = 1 to d do

sort array A on digit i using any sorting algorithm

Time Complexity: $O((N+m) + (N+m^2) + \ldots + (N+m^d))$

Space Complexity: $O(m^d)$
Radix Sort

Algorithm
for i = 1 to d do
    sort array A on digit i using a stable sort algorithm

Time Complexity: $O((n+m)d)$

Warning: This algorithm cannot be used for “infinite-precision” real numbers, even if the range of values is specified.
Counting Sort

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>Counts</th>
<th>Cumulative Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>2 5 3 0 2 3 0 3</td>
<td>2 0 2 3 0 1</td>
<td>2 2 4 7 7 8</td>
</tr>
</tbody>
</table>

• Warning: This algorithm cannot be used for “infinite-precision” real numbers, even if the range of values is specified.
Storing binary trees as arrays
**Heaps (Max-Heap)**

**HEAP** represents a binary tree stored as an array such that:

- Tree is filled on all levels except last
- Last level is filled from left to right
- Left & right child of \( i \) are in locations \( 2i \) and \( 2i+1 \)

**HEAP PROPERTY:**
Parent value is at least as large as child’s value
HeapSort

- First convert array into a heap (BUILD-MAX-HEAP, p133)
- Then convert heap into sorted array (HEAPSORT, p136)
Animation Demos

http://www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html

http://cg.scs.carleton.ca/~morin/misc/sortalg/
HeapSort: Part 1

MAX-HEAPIFY(array A, int i)
▷ Assume subtree rooted at i is not a heap;
▷ but subtrees rooted at children of i are heaps
1  l ← LEFT[i]
2  r ← RIGHT[i]
3  if ((l ≤ heap-size[A]) and (A[l] > A[i]))
4      then largest ← l
5  else largest ← i
6  if ((r ≤ heap-size[A]) and (A[r] > A[largest]))
7      then largest ← r
8  if (largest ≠ i)
9      then exchange A[i] ← A[largest]
10     MAX-HEAPIFY(A, largest)

O(height of node in location i) = O(log(size of subtree))
HeapSort: Part 2

**BUILD-MAX-HEAP**(array A)

1. `heap-size[A] ← length[A]`
2. for `i ← [length[A]/2] downto 1`
3. do **MAX-HEAPIFY**(A, i)
HeapSort: Part 2

**BUILD-MAX-HEAP**(array \( A \))

1. \( \text{heap-size}[A] \leftarrow \text{length}[A] \)
2. \( \text{for} \ i \leftarrow \lfloor \text{length}[A]/2 \rfloor \ \text{downto} \ 1 \)
   3. \( \text{do} \ \text{MAX-HEAPIFY}(A, i) \)

**HeapSort**(array \( A \))

1. \( \text{BUILD-MAX-HEAP}(A) \)
2. \( \text{for} \ i \leftarrow \text{length}[A] \ \text{downto} \ 2 \)
   3. \( \text{do} \ \text{exchange} \ A[1] \leftarrow A[i] \)
   4. \( \text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1 \)
   5. \( \text{MAX-HEAPIFY}(A, 1) \)

Total: \( \mathcal{O}(n \log n) \)
We need to compute:

\[ \sum_{h=0}^{[\log n]} \frac{h}{2^n} \]

We know that

\[
\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}
\]

Differentiating both sides, we get

\[
\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1 - x)^2}
\]

Multiplying both sides by \( x \), we get

\[
\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2}
\]

Setting \( x = 1/2 \), we can show that

\[
\sum_{h=0}^{[\log n]} \frac{h}{2^n} \leq 2
\]