Sorting Algorithms (Part II)

Slightly modified definition of the sorting problem:

**input**: A collection of \( n \) data items \(<a_1, a_2, ..., a_n>\) where data item \( a_i \) has a *key*, \( k_i \), drawn from a linearly ordered set (e.g., ints, chars)

**output**: A permutation \(<a'_1, a'_2, ..., a'_n>\) of the input sequence such that \( k_1 \leq k_2 \leq ... \leq k_n \)

- In practice, one usually sorts 'objects' according to their key (the non-key data is called *satellite data*.)
- If the records are large, we may sort an array of pointers based on some key associated with each record.

• A sorting algorithm is *comparison-based* if the sorting operation depends on a pair-wise comparison of keys.
• A sorting algorithm is *in-place* if only a constant number of elements of the input array are ever stored outside the array.

### Running Time of Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( w-c )</th>
<th>( a-c )</th>
<th>( b-c )</th>
<th>In place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>( n^2 )</td>
<td>( n^2 )</td>
<td>( n )</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>( n^2 )</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>yes</td>
</tr>
</tbody>
</table>

Tree Terminology

The height of a tree node \( i \) is the number of edges on the longest path from node \( i \) to a leaf.

The level (depth) of a tree node starts with 0 for the root level and is incremented by 1 for each downward edge.

In a complete binary tree, every level, except possibly the last, is completely filled, and all bottom nodes are as far left as possible. Heaps are examples of complete binary trees.

A full binary tree (sometimes called a proper binary tree) is a tree in which every node other than the leaves has two children. A perfect binary tree is a proper binary tree in which every level is filled.

Heapsort (Ch. 6)

**heap concept**

- Complete binary tree.
- Values in the nodes satisfy the "heap properties".

**heap as an array implementation**

- store heap as a binary tree in an array
- *heapsize* is number of elements in heap
- *length* is number of elements in array

Max-Heap Definition

A max-heap is a binary tree with one key assigned to each node such that the tree meets the following 2 requirements:

1. **Shape requirement**: complete binary tree.
2. **Parental dominance requirement**: the key at each node is greater than or equal to the keys at its children.

Max-Heap

In the array representation of a max-heap, the root of the tree is in \( A[1] \). Given the index \( i \) of a node,

\[
\begin{align*}
\text{Parent}(i) &= (i/2) \quad \text{LeftChild}(i) = (2i) \quad \text{RightChild}(i) = (2i + 1) \\
\end{align*}
\]

### Diagrams

- [Tree Terminology Diagram](https://example.com/tree-diagram)
- [Heapsort (Ch. 6) Diagram](https://example.com/heapsort-diagram)
- [Max-Heap Definition](https://example.com/max-heap-diagram)
- [Max-Heap](https://example.com/max-heap-array-diagram)
Min-Heap Definition

A min-heap is a binary tree with one key assigned to each node such that the tree meets the following 2 requirements:

1. Shape requirement: complete binary tree.
2. Child dominance requirement: the key at each node is less than or equal to the keys at its children.

Min-Heap

Min-heaps are commonly used for priority queues in event-driven simulators.

Parent(i)  LeftChild(i)  RightChild(i)
return (i/2)  return (2i)  return (2i + 1)

2     3      5     4    6     9   11  10  15  20   18
1     2     3     4     5    6    7    8    9  10   11

Heapsort

Input: An n-element array A[1...n].
Output: An n-element array A in sorted order, smallest to largest.

Heapsort(A)
1. Build-Heap(A)  /* put all elements in heap */
2. for i = A.length downto 2
4. A.heap-size = A.heap-size - 1
5. Max-Heapify(A,1)  /* restore heap property */

Running time:
Line 1: c_1 (?????) Need to know running time of Build-Heap
Line 2: c_2 n
Line 3: c_3(n - 1)
Line 4: c_4(n - 1)
Line 5: c_5(n - 1) * (?????) Need to know running time of Max-Heapify

Max-Heapify: Maintains the Max-Heap Property

Input: An unordered array A[1...n] of comparable items and the index i of a node whose subtrees are max-heaps, but i might violate max-heap property.
Output: A permutation of A such that i is the root of a max-heap

Max-Heapify(A, i)
1. left = 2i; right = 2i + 1  /* indices of left & right children of A[i] */
2. largest = i
3. if left <= heap-size(A) and A[left] > A[i]
4. largest = left
5. if right <= heap-size(A) and A[right] > A[largest]
6. largest = right
7. if largest != i
8. swap(A[i], A[largest])
9. Max-Heapify(A, largest)

How many calls does Heapsort make to Max-Heapify?
How many swaps are made in lines 7, 8, and 9 (worst-case)?
When does the recursion end and why?

Max-Heapify: Running Time

Running Time of Max-Heapify
• every line is o(1) time, except the recursive call
• in worst-case, last row of binary tree is half empty, so the sub-tree rooted at left child has size at most (2/3)n
So we get the recurrence
T(n) ≤ T(2n/3) + o(1)
which, by case 2 of the master theorem, has the solution
T(n) = Θ(lg n)

(or, Max-Heapify takes O(h) time when node A[i] has height h in the heap)
A heap is a complete binary tree, hence must process O(lg n) levels, with constant work at each level.
Heapify: More pseudo-codish version of Max-Heapify

Input: An unordered array A[1...n] of comparable items and the index i of a node whose subtrees are max-heaps, but i might violate max-heap property.
Output: A permutation of A such that i is the root of a max-heap

Heapify(A, i)
1. if node i is not a leaf and A[i] < A[2i] and/or A[2i+1]
2. Let j be the maximum value child of i
4. Heapify(A, j)

Build-Heap

Intuition: use Heapify in a bottom-up manner to convert A into a max-heap

Build-Heap(A)
1. for i = A.length downto 2 // diff. in book
2. Heapify(A, i)

Rough running Time of Build-Heap:
• About n calls to Heapify (O(n) calls)
• Simple upper bound: Each call to Heapify takes O(lgn) time → O(nlgn) time total.
• However, we can prove that Build-Heap runs in O(n) time, meaning that the O(nlgn) bound is correct but is not tight.

Lemma 1: The number of nodes at level i in a perfect binary tree is 2^i

Basis: level=0. 2^0 = 1, so the lemma holds because there is only a root.
Inductive Step: Assume true for level k and show true for level k + 1.

By the IHOP, there are 2^k nodes at level k of a perfect binary tree. By the definition of a perfect binary tree, level k+1 must be full, meaning that every node at level k has 2 children. Therefore, the number of nodes at level k+1 = 2(2^k) = 2^{k+1}.

Lemma 2: A perfect binary tree (pbt) of height h contains 2^{h+1} - 1 nodes.

By Lemma 1, there are 2^k nodes at level k of a perfect binary tree. So the number of nodes in a perfect tree of height h is:

\[ \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \]

This is a geometric series.

Notation you have seen before...

- Floor: \( \lfloor x \rfloor \) = the largest integer \( \leq x \)
- Ceiling: \( \lceil x \rceil \) = the smallest integer \( \geq x \)
- Geometric series: \( \sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1} \)
- Harmonic series: \( H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} = \ln n + \gamma \)
- Telescoping series: \( \sum_{i=0}^{n} (a_{i+1} - a_i) = a_{n+1} - a_0 \)
- \( \sum_{i=0}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1} - \frac{1}{n+1} \)

Build-Heap - Tighter bound

Build-Heap(A)
1. for i = A.length downto 2
2. Heapify(A, i)

We use the following fact to prove the linear running time of Build-Heap:

FACT 1:

For any positive integer k, \( 2^k - 1 = 2^{k-1} + 2^{k-2} + ... + 2^2 + 2^1 + 2^0 \).

Let T(n) denote the worst-case complexity of algorithm Build-Heap on any input of size n. We consider a perfect binary tree with height h = \( \lceil \lg(n) \rceil \) and look at the total amount of work completed for all nodes at each height. At height 0 (level h) in the tree, no work is done since those nodes are leaves. At height 1 (level h-1), 2^{h-1} nodes need to be “fixed” via a call to Heapify. This requires no more than O(1) comparisons for each of these nodes.
Visual Proof of Build-Heap - Tighter bound

The amount of work done at level h-1 is denoted by the left-most blue rectangle in the figure below.

At level h-2, O(2) units of work are required for all 2^{h-2} nodes and so on.

At level 0, O(h) units of work are required for a single node, the root.

The amount of work done at level h-1 is denoted by the left-most blue rectangle in the figure below.

Inserting Heap Elements

Assume A.length > heapsize (the array is large enough to expand the heap)

1. Increment heapsize and insert new element in the highest numbered position of array. Assume items that were already in the heap form a max-heap.
2. Walk up the tree from new leaf to root. Insert input key when a parent key larger than the input key is found

Max-Heap-Insert(A, key)
1. heapsize(A) = heapsize(A) +1
2. i = heapsize(A)
3. while i > 1 and A[parent(i)] < key
5. i = parent(i)
6. A[i] = key

Running time of Max-Heap-Insert:
- time to traverse leaf to root path = height = O(lg n)

Heapsort Time and Space Usage

- An array implementation of a heap uses O(n) space
  - one array element for each node in heap
- Heapsort is an in-place algorithm
- Running time is as good as Mergesort, O(n lg n) in worst case. Note that the running time of Heapsort and Max-Heapify dominate the O(n) running time of Build-Heap.

Suppose we used a top-down approach to build the heap instead of the bottom-up Build-Heap. How would this algorithm work? What would the running time be?

Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated key such that the element with highest priority is always accessed quickly. A max-priority-queue gives priority to keys with larger values and supports the following operations:

1. insert(S, x) inserts the element x into set S.
2. max(S) returns element of S with largest key.
3. extract-max(S) removes and returns element of S with largest key.
4. increase-key(S, x, k) increases the value of element x's key to new value k (assuming k is at least as large as current key's value).

Priority Queues: Application for Heaps

An application of max-priority queues is to schedule jobs on a shared processor. Need to be able to
- check current job's priority
- remove job from the queue
- insert new jobs into queue
- increase priority of jobs

Heap-Maximum(A)
Heap-Extract-Max(A)
Max-Heap-Insert(A, key)
Heap-Increase-Key(A, i, key)

Initialize PQ by running Build-Heap on an array A. A[1] holds the maximum value after this step.

Heap-Maximum(A) - returns value of A[1] but leaves heap as-is.
Heap-Extract-Max(A) - Saves A[1] and then, like Heap-Sort, puts item in A[heapsize] at A[1], decrements heapsize, and uses Max-Heapify(A, 1) to restore heap property.
Heap-Increase-Key(A, i, key) - If key is larger than key at parent(i), floats node with new key up heap until heap property is restored.
Min-Heap Application

An application for a min-heap priority queue is an event-driven simulator, where the key is an integer representing the number of seconds (or other discrete time unit) from time zero (starting point for simulation).