Ch6 - Sorting

In the next few chapters:

1. We look at a number of algorithms that let us sort in time \( O(n \lg n) \), deterministically or probabilistically.
2. We also provide partial answers to the existence of sorting methods that use more detailed information about the data-sets: are there ways to sort faster than \( O(n \lg n) \)? What (if anything) do we have to trade to achieve the speedup?
3. We also look at Order Statistics: how fast can we compute the largest, smallest, \( k^{th} \), median elements of a set?
4. We introduce the notion of Priority Queue: how can we maintain a changing set where we need to access the highest priority (largest, smallest) element? We need a fast algorithm, as insensitive as possible to the size of the set.

Some Algorithms and their Time-Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
<th>Average-case/expected running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>Merge sort</td>
<td>( \Theta(n \lg n) )</td>
<td>( \Theta(n \lg n) )</td>
</tr>
<tr>
<td>Heapsort</td>
<td>( O(n \lg n) )</td>
<td>( - )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n \lg n) ) (expected)</td>
</tr>
<tr>
<td>Counting sort</td>
<td>( \Theta(k + n) )</td>
<td>( \Theta(k + n) )</td>
</tr>
<tr>
<td>Radix sort</td>
<td>( \Theta(d(n + k)) )</td>
<td>( \Theta(d(n + k)) )</td>
</tr>
<tr>
<td>Bucket sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) ) (average-case)</td>
</tr>
</tbody>
</table>
What is a MAX-HEAP?

The MAX-HEAP property requires $A[\text{PARENT}(i)] \geq A[i]$.

The MIN-HEAP property requires $A[\text{PARENT}(i)] \leq A[i]$.

The largest element in a MAX-HEAP is the root.

The smallest element in a MIN-HEAP is the root.
Ch6 – Sorting - HEAPSORT

What is a MAX-HEAP?
How do you make a MAX-HEAP? Start with the assumption that the binary trees rooted at LEFT[i] and RIGHT[i] are MAX-HEAPS. How do you ensure that the tree rooted at i is a MAX-HEAP?

**MAX-HEAPIFY(A, i)**

1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. if \( l \leq A.\heapsize \) and \( A[l] > A[i] \)
   4. \( \text{largest} = l \)
   5. Else \( \text{largest} = i \)
6. if \( r \leq A.\heapsize \) and \( A[r] > A[\text{largest}] \)
   7. \( \text{largest} = r \)
8. if \( \text{largest} \neq i \)
   9. exchange \( A[l] \) with \( A[\text{largest}] \)
10. MAX-HEAPIFY(A, largest)

---

**Figure 6.2** The action of MAX-HEAPIFY(4, 2), where \( A.\heapsize = 10 \). (a) The initial configuration, with \( A[2] \) at node 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging \( A[2] \) with \( A[4] \), which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY(4, 4) now has \( i = 4 \). After swapping \( A[4] \) with \( A[8] \), as shown in (c), node 4 is fixed up, and the recursive call MAX-HEAPIFY(4, 9) makes no further change to the data structure.

---
Ch6 – Sorting - HEAPSORT

What is a MAX-HEAP?

1. Assume we apply MAX-HEAPIFY to a node $i$ at the root of a tree of $n$ nodes.
2. The cost of determining the relationship among $i$, $\text{LEFT}[i]$ and $\text{RIGHT}[i]$ is $\Theta(1)$.
3. Then we may have a recursion (or not). Bound on cost?
   1. If the three rooted at $i$ has $n$ nodes, what are the sizes of the subtrees?
   2. Notice that the original tree is a complete binary tree so the maximum disparity between the subtrees occurs when the left subtree has a full bottom row, while the right one has an empty one.
   3. The largest subtree has $(n - 1) \times \frac{2}{3} \leq \frac{2n}{3}$ nodes
   4. A useful recursion relation is $T(n) = T(2n/3) + \Theta(1)$.
   5. By Case 2 of the Master Theorem $T(n) = O(\lg n)$. (Also corroborated by the fact that the height of the root node ($h$) is $O(\lg n)$).

Ch6 – Sorting - HEAPSORT

What is a MAX-HEAP?

How do we build a MAX-HEAP?

\begin{verbatim}
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = floor(A.length/2) downto 1
3      MAX-HEAPIFY(A, i)
\end{verbatim}

Create the heap from the bottom up! How do we prove that we end with a MAX-HEAP? A loop invariant!

We observe that,
1. At initialization, all the nodes in positions $\lfloor A.length/2 \rfloor + 1, \ldots, n$ are roots of MAX-HEAPS (leaves: they have no children).
2. At termination, the root node (with index 1) must be the root of a MAX-HEAP.
3. Maintenance condition: what must hold for every execution of the for loop?
Ch6 – Sorting - HEAPSORT

**MAX-HEAP: the Maintenance Condition.**

1. We first observe that the children of node \( i \) are numbered higher than \( i \). By the loop invariant they must be roots of MAX-HEAPS.

2. This is the condition required for the call MAX-HEAPIFY\((A, i)\) to make node \( i \) a MAX-HEAP root.

3. The MAX-HEAPIFY call preserves the properties that the nodes \( i+1, i+2, \ldots, n \) are all roots of MAX-HEAPS.

4. Decrementing \( i \) in the for loop update re-establishes the loop invariant for the next iteration.

---

**Heaps – Time**

It is easy to show that constructing a heap with \( n \) elements costs \( O(n \lg n) \). But we can do better: we will show linear time.

We start with two observations:

1. An \( n \)-element heap has height \( \lfloor \lg n \rfloor \). (Ex. 6.1-2)

2. An \( n \)-element heap has at most \( \lceil n/2^{h+1} \rceil \) nodes of height \( h \). (Ex. 6.3-3)

The time required by MAX-HEAPIFY when called on a node of height \( h \) is \( O(h) \). The total cost of BUILD-MAX-HEAP is bounded above by

\[
\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} O(h) = O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} h \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n^2) = O(n).
\]

A complementary series of results applies to BUILD-MIN-HEAP.
Ch6 – Sorting - HEAPSORT

Time-Out: Some Math

Claim: \( \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \) for \( |x| < 1 \).

Proof: we first prove \( S(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \) for \( |x| < 1 \).

Let \( S_n(x) = \sum_{k=0}^{n} x^k \).

\[
S_n(x) - xS_n(x) = 1 - x^{n+1} \\
S_n(x)(1-x) = 1 - x^{n+1} \\
S_n(x) = \frac{1-x^{n+1}}{1-x}
\]

Taking the limit as \( n \to \infty \) we have \( S(x) = 1/(1 - x) \).

Ch6 – Sorting - HEAPSORT

Time-Out: Some Math

Claim: \( \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \) for \( |x| < 1 \).

Proof: Differentiate \( S(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \) for \( |x| < 1 \).

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

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

The result follows.
Heaps - example

Heaps – how do you sort?

Heapsort(A)
1 build-max-heap(A)
2 for i = A.length downto 2
4 A.heap-size = A.heap-size - 1
5 max-heapify(A, 1)

Time: $O(n \log n)$. 
Ch6 – Sorting - **HEAPSORT**

**HeapSort Example**

This is an abstract data types that is useful for simulation: each item has a priority, and the highest priority item must be served first. Time-stamped data packets, processes, or whatever, can be managed via Priority Queues. They support the following operations:

- **INSERT(S, x)** inserts the element *x* into the set *S*, which is equivalent to the operation *S* = *S* ∪ {x}.

- **MAXIMUM(S)** returns the element of *S* with the largest key.

- **EXTRACT-MAX(S)** removes and returns the element of *S* with the largest key.

- **INCREASE-KEY(S, x, k)** increases the value of element *x*’s key to the new value *k*, which is assumed to be at least as large as *x*’s current key value.
Ch6 – Sorting - HEAPSORT

Priority Queues – Implementation

HEAP-MAXIMUM(A)
1 return A[1]

Time: $O(1)$ – no modifications.

HEAP-EXTRACT-MAX(A)
1 if A.heap-size < 1
2 error “heap underflow”
3 max = A[1]
5 A.heap-size = A.heap-size – 1
6 MAX-HEAPIFY(A, 1)
7 return max

Time: $O(lg n)$ – heap needs to be rebuilt after extraction.

Ch6 – Sorting - HEAPSORT

Priority Queues – Implementation

HEAP-INCREASE-KEY(A, i, key)
1 if key < A[i]
2 error “new key is smaller than current key”
3 A[i] = key
4 while i > 1 and A[PARENT(i)] < A[i]
5 exchange A[i] with A[PARENT(i)]
6 i = PARENT(i)

Increase the priority of an item in the PQ, and rebuild the PQ. Time: $O(lg n)$. 

Ch6 – Sorting - HEAPSORT

Priority Queues – Implementation

MAX-HEAP-INSERT (A, key)
1 $A.heap-size = A.heap-size + 1$
2 $A[A.heap-size] = -\infty$
3 HEAP-INCREASE-KEY (A, A.heap-size, key)

First prepare the space (with a fictitious key), and then rebuild the heap, with the correct value for the key. Time: $O(\log n)$.

---

Ch6 – Sorting - HEAPSORT

Priority Queues – Example

(a) 
(b) 
(c) 
(d)