Part II Sorting algorithms
Why sorting

1. Sometimes the need to sort information is inherent in an application.
2. Algorithms often use sorting as a key subroutine.
3. There is a wide variety of sorting algorithms, and they use rich set of techniques.
4. Sorting problem has a nontrivial lower bound.
5. Many engineering issues come to fore when implementing sorting algorithms.

Sorting Algorithm Animations http://www.sorting-algorithms.com/
Sorting algorithm

- **Insertion sort**:  
  - In place: only a constant number of elements of the input array are even sorted outside the array.

- **Merge sort**:  
  - not in place.

- **Heap sort**:  
  - Sorts $n$ numbers in place in $O(n \log n)$
Sorting algorithm

- **Quick sort**: (chapter 7)
  - worst time complexity $O(n^2)$
  - Average time complexity $O(n \log n)$
- **Decision tree model**: (chapter 8)
  - Lower bound $\Omega (n \log n)$
  - Counting sort $\Theta (k+n)$ if $k=O(n)$ ???
  - Radix sort $\Theta (d(k+n))$ if $d$ is a constant & $k=O(n)$ ???
6. Heapsort

https://zh.wikipedia.org/wiki/%E5%A0%86%E6%8E%92%E5%BA%8F
6.1 Heaps (Binary heap)

- The **binary heap** data structure is an array object that can be viewed as a complete tree.

Parent($i$)  \[ \text{return } \lceil \frac{i}{2} \rceil \]

Left($i$)  \[ \text{return } 2i \]

Right($i$)  \[ \text{return } 2i+1 \]
Heap property

- **Max-heap**: $A[\text{parent}(i)] \geq A[i]$
- **Min-heap**: $A[\text{parent}(i)] \leq A[i]$

The *height of a node* in a tree: the number of edges on the longest simple downward path from the node to a leaf.

- The *height of a tree*: the height of the root
- **The height of a heap**: $\Theta(\lg n)$.

[https://www.youtube.com/watch?v=WCm3TqScBM8](https://www.youtube.com/watch?v=WCm3TqScBM8)
Basic procedures on heap

- Max-Heapify procedure $O(\log n)$
- Build-Max-Heap procedure $O(n)$
- Heapsort procedure $O(n \log n)$
- Max-Heap-Insert procedure
- Heap-Extract-Max procedure
- Heap-Increase-Key procedure $O(\log n)$
- Heap-Maximum procedure

Chapter 6
6.2 Maintaining the heap property

- Heapify is an important subroutine for manipulating heaps. Its inputs are an array $A$ and an index $i$ in the array. When Heapify is called, it is assume that the binary trees rooted at LEFT($i$) and RIGHT($i$) are heaps, but that $A[i]$ may be smaller than its children, thus violating the heap property.
Max-Heapify (A, i)
1  \( l \rightarrow \text{Left}(i) \)
2  \( r \rightarrow \text{Right}(i) \)
3  \textbf{if} \( l \leq \text{heap-size}[A] \) and \( A[l] > A[i] \)
4  \textbf{then} largest \( \leftarrow l \)
5  \textbf{else} largest \( \leftarrow i \)
6  \textbf{if} \( r \leq \text{heap-size}[A] \) and \( A[r] > A[\text{largest}] \)
7  \textbf{then} largest \( \leftarrow r \)
8  \textbf{if} largest \( \neq i \)
9  \textbf{then} exchange \( A[i] \leftrightarrow A[\text{largest}] \)
10 \text{Max-Heapify (A, largest)}

\[ T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1) \Rightarrow T(n) = O(\lg n) \]

Alternatively \( O(h) \) (\( h \): height)

The children's subtrees each have size at most \( 2n/3 \)-the worst case occurs when the last row of the tree is exactly half full-and the running time of MAX-HEAPIFY can therefore be described by the recurrence.
Max-Heapify(A, 2)
heap-size[A] = 10
6.3 Building a heap

Build-Max-Heap(A)

1. heap-size[A] ← length[A]
2. for i ← ⌊ length[A]/2 ⌋ downto 1
3. do Max-Heapify(A, i)
Build-Max-Heap is $O(n \log n)$?

$$
O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}}) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})
$$

$$
\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \left( \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \right)
$$

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

Exercises 6.3-3

Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$ in any $n$-element heap.
An example on heapsort

\[ \begin{array}{cccccccc}
6 & 5 & 3 & 1 & 8 & 7 & 2 & 4
\end{array} \]
6.4 The Heapsort algorithm

Heapsort(A)
1  Build-Max-Heap(A)
2  for i ← length[A] down to 2
4   heap-size[A] ← heap-size[A] - 1
5   Max-Heapify(A, 1)
The operation of Heapsort
Analysis: $O(n \log n)$
6.5 Priority queues

A priority queue is a data structure that maintains a set $S$ of elements, each with an associated value called a key. A max-priority queue supports the following operations:

- **Insert** ($S, x$) $O(\log n)$
- **Maximum** ($S$) $O(1)$
- **Extract-Max** ($S$) $O(\log n)$
- **Increase-Key** ($S, x, k$) $O(\log n)$

https://www.youtube.com/watch?v=-WEku8ZnynU
Heap_Extract-Max(A)

1 if heap-size[A] < 1
2 then error “heap underflow”
3 max ← A[1]
5 heap-size[A] ← heap-size[A] - 1
6 Max-Heapify (A, 1)
7 return max
Heap-Increase-Key \((A, i, \text{key})\)

1. \textbf{if} key < \(A[i]\)
2. \textbf{then error} “new key is smaller than current key”
3. \(A[i] \leftarrow \text{key}\)
4. \textbf{while} \(i > 1\) and \(A[\text{Parent}(i)] < A[i]\)
5. \textbf{do} exchange \(A[i] \leftrightarrow A[\text{Parent}(i)]\)
6. \(i \leftarrow \text{Parent}(i)\)
Heap-Increase-Key
Heap_Insert(A, key)

1. heap-size[A] ← heap-size[A] + 1
2. A[heap-size[A]] ← -∞
3. Heap-Increase-Key (A, heap-size[A], key)