This lecture and the next

- Heapsort
  - Heap data structure and priority queue ADT
- Quicksort
  - A popular algorithm, very fast on average

Why Sorting?
- "When in doubt, sort" – one of the principles of algorithm design. Sorting used as a subroutine in many of the algorithms:
  - Searching in databases: we can do binary search on sorted data
  - A large number of computer graphics and computational geometry problems
  - Closest pair, element uniqueness

Why Sorting? (2)
- A large number of sorting algorithms are developed representing different algorithm design techniques.
- A lower bound for sorting $\Omega(n \log n)$ is used to prove lower bounds of other problems

Sorting Algorithms so far
- Insertion sort, selection sort
  - Worst-case running time $\Theta(n^2)$; in-place
- Merge sort
  - Worst-case running time $\Theta(n \log n)$, but requires additional memory $\Theta(n)$; (WHY?)

Selection Sort

```plaintext
Selection-Sort(A[1..n]):
  For i -> n downto 2
  A:    Find the largest element among A[1..i]
  B:    Exchange it with A[i]
```
- A takes $\Theta(n)$ and B takes $\Theta(1)$; $\Theta(n^2)$ in total
- Idea for improvement: use a data structure, to do both A and B in $O(\log n)$ time, balancing the work, achieving a better trade-off, and a total running time $O(n \log n)$

Heap Sort
- Binary heap data structure A
  - array
  - Can be viewed as a nearly complete binary tree
    - All levels, except the lowest one are completely filled
    - The key in root is greater or equal than all its children, and the left and right subtrees are again binary heaps
- Two attributes
  - length[A]
  - heap-size[A]
The method "Heapify" makes a heap of size $n$ rooted at node $i$ is

- determining the relationship between elements: $\Theta(1)$
- plus the time to run Heapify on a subtree rooted at one of the children of $i$, where $2n/3$ is the worst-case size of this subtree.

$T(n) \leq T(2n/3) + \Theta(1)$ \quad \Rightarrow \quad T(n) = O(\log n)$

Alternatively

- Running time on a node of height $h$: $O(h)$
Building a Heap

- Convert an array $A[1...n]$, where $n = \text{length}[A]$, into a heap
- Notice that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)...n]$ are already 1-element heaps to begin with!

**BUILD-HEAP($A$)**
1. for $i \leftarrow \lfloor n/2 \rfloor$ down to 1
2. do HEAPIFY($A[i]$)

Building a Heap: Analysis

- Correctness: induction on $i$, all trees rooted at $m > i$ are heaps
- Running time: $n$ calls to HEAPIFY = $n \cdot O(\log n) = O(n \log n)$
- Good enough for an $O(n \log n)$ bound on Heapsort, but sometimes we build heaps for other reasons, would be nice to have a tight bound
- Intuition: for most of the time, HEAPIFY works on smaller than $n$ element heaps

Building a Heap: Analysis (3)

- How? By using the following "trick"
  \[ \sum_{i=1}^{n} x^i = \frac{1}{1-x} \quad \text{if } |x| < 1 \]
  \[ \sum_{i=1}^{n} x^i = \frac{x}{(1-x)^2} \quad \text{plug in } x = \frac{1}{2} \]
  \[ \sum_{i=2}^{n} i = \frac{1}{2} \cdot \frac{1}{1/4} \]
- Therefore Build-Heap time is $O(n)$

Building a Heap: Analysis (2)

- Definitions
  - height of node: longest path from node to leaf
  - height of tree: height of root

**BUILD-HEAPIFY($A$)**
1. for $i \leftarrow \lfloor n/2 \rfloor$ down to 1
2. do HEAPIFY($A[i]$)

- time to HEAPIFY = $O(\text{height of subtree rooted at } i)$
- assume $n = 2^k - 1$ (a complete binary tree $k = \lfloor \log n \rfloor$)
- $T(n) = O(\sum \frac{\log \log \log \log \ldots \log \log \log n}{\log n})$
  \[ T(n) = O\left(\frac{\log \log \log \log \ldots \log \log \log n}{\log n}\right) = O(1) \]

Heap Sort

- The total running time of heap sort is $O(n \log n) + \text{Build-Heap}(A)$ time, which is $O(n)$
Heap Sort: Summary

- Heap sort uses a heap data structure to improve selection sort and make the running time asymptotically optimal.
- Running time is $O(n \log n)$ – like merge sort, but unlike selection, insertion, or bubble sorts.
- Sorts in place – like insertion, selection or bubble sorts, but unlike merge sort.

Priority Queues

- A priority queue is an ADT (abstract data type) for maintaining a set $S$ of elements, each with an associated value called key.
- A PQ supports the following operations:
  - Insert$(S, x)$: insert element $x$ in set $S$ ($S ← S ∪ \{x\}$).
  - Maximum($S$): returns the element of $S$ with the largest key.
  - Extract-Max($S$): returns and removes the element of $S$ with the largest key.

Priority Queues (2)

- Applications:
  - Job scheduling shared computing resources (Unix).
  - Event simulation.
  - As a building block for other algorithms.
- A Heap can be used to implement a PQ.

Priority Queues (3)

- Removal of max takes constant time on top of Heapify $\Theta(\lg n)$.

Priority Queues (4)

- Insertion of a new element:
  - Enlarge the PQ and propagate the new element from last place "up" the PQ.
  - Tree is of height $\lg n$, running time: $\Theta(\lg n)$.

```
Heap-Insert(A, key)
1 len ← heap-size(A) + 1
2 A[len] ← key
3 current ← len
4 while current > 1 and A[parent(current)] < key
6 current ← parent(current)
7 A[current] ← key
```
Quick Sort

- Characteristics
  - sorts almost in "place," i.e., does not require an additional array, like insertion sort
  - Divide-and-conquer, like merge sort
  - very practical, average sort performance $O(n \log n)$ (with small constant factors), but worst case $O(n^2)$

Quick Sort – the Principle

- To understand quick-sort, let's look at a high-level description of the algorithm
- A divide-and-conquer algorithm
  - Divide: partition array into 2 subarrays such that elements in the lower part $\leq$ elements in the higher part
  - Conquer: recursively sort the 2 subarrays
  - Combine: trivial since sorting is done in place

Quick Sort Algorithm

Initial call \texttt{Quicksort(A, 1, length[A])}

```
Quicksort(A, p, r)
01 if p<r
02 then q ← Partition(A, p, r)
03 Quicksort(A, p, q)
04 Quicksort(A, q+1, r)
```

Partitioning

Linear time partitioning procedure

```
Partition(A, p, r)
01 x ← A[r]
02 i ← p-1
03 j ← r+1
04 while TRUE
05 repeat j ← j-1
06 until A[j] ≤ x
07 repeat i ← i+1
08 until A[i] ≥ x
09 if i<j
10 then exchange A[i] ↔ A[j]
11 else return j
```

Analysis of Quicksort

- Assume that all input elements are distinct
- The running time depends on the distribution of splits
### Best Case
- If we are lucky, Partition splits the array evenly
  \[ T(n) = 2T(n/2) + \Theta(n) \]

### Worst Case
- What is the worst case?
- One side of the partition has only one element
  \[ T(n) = T(1) + T(n-1) + \Theta(n) \]
  \[ = T(n-1) + \Theta(n) \]
  \[ = \sum_{i=1}^{n} \Theta(k) \]
  \[ = \Theta(n^2) \]

### Worst Case (2)
- Suppose the split is 1/10 : 9/10
  \[ T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n) \]

### Using the median as a pivot
- The recurrence in the previous slide works out, BUT......

Q: Can we find the median in linear-time?
A: YES! But we need to wait until we get to Chapter 8......
An Average Case Scenario

Suppose, we alternate lucky and unlucky cases to get an average behavior

\[ L(n) = 2U(n/2) + \Theta(n) \text{ lucky} \]
\[ U(n) = L(n-1) + \Theta(n) \text{ unlucky} \]
we consequently get

\[ L(n) = 2L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \]
\[ = 2L(n/2 - 1) + \Theta(n) \]
\[ = \Theta(n \log n) \]

\[ n \text{ - } \Theta(n) \]

Randomized Quicksort

- Assume all elements are distinct
- Partition around a random element
- Consequently, all splits (1:n-1, 2:n-2, ..., n-1:1) are equally likely with probability 1/n
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity

Next Week

Q: Can we beat the \( \Omega(n \log n) \) lower bound for sorting?
A: In general no, but in some special cases YES!
Ch 7: Sorting in linear time