HeapSort loop invariant

Build - Max - Heap(A)
1 heap-size[A] = length[A]
2 for i = [length[A]/2] downto 1
3 MAX-HEAPIFY(A, i)

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the for loop of lines 2–3, each node i + 1, i + 2, …, n is the root of a max-heap.

---

Sorting

Mergesort

Heapsort $O(n \log n)$ time

General sorts seem to have running times of $\Omega(n \log n)$

Nice to say "Any sorting alg. takes $\Omega(n \log n)$ time"

Equiv. Say an alg is $\Omega(n \log n)$ means that for infinitely many $n$, there exists an input of size $n$ that takes $\Omega(n \log n)$ time.
Want to say: All algorithms that sort
take \( \mathcal{O}(n \log n) \) time.

- take all numbers
- sort every \( 3'd \)
- look up page A(7) in N+Times
- Double it

Think about sorts, they do comparisons & swaps.

Restrict attention to algorithms that only access the data via comparisons.
- Conveneince: all elements are distinct, \( \leq \)
Any permutation of the data \( a_1, \ldots, a_n \)

- \( a_i \leq a_j \)
- \( a_i > a_j \)

How many comparisons must I make to sort correctly?

Any algorithm that correctly sorts 3 items takes \( \geq 3 \) comparisons.
In order to correctly sort, a decision tree must have \( \geq n! \) leaves.

Height of a tree is the length of the longest root-leaf path.

\[
2^h \geq \text{#leaves} \geq n!
\]

\[
2h \geq n!
\]

\[
2^h \geq n!
\]

\[
h \geq \log(n!)
\]

\[
h = \Omega(n \log n)
\]

Sterling's Approx.

\[
n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \left(1 + o(1)\right)
\]

\[
lg(n!) \leq n \left(\frac{n}{e}\right)^n \leq \Theta(n \log n)
\]
Any alg. that only access the data via comparisons must take $\Omega(n\log n)$ time.

Probably implies: Any alg. that sorts any kind of ordered data takes $\Omega(n\log n)$ time.

Maybe: if we know something about our data, we might be able to sort faster.

$A[i]$, and use that to compute position.

- Array $A[1..n]$ — holds input
- Array $C[1..k]$ — $C[j]$ holds number of elements of $A$ less than or equal to $j$

Example:

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Questions:
- How do we compute $C$?
- We need to be careful dealing with duplicates (stability)
Counting Sort

$\text{Counting - Sort}(A, B, k)$

1. for $i = 0$ to $k$
   2. do $C[i] = 0$
3. for $j = 1$ to $\text{length}[A]$
   4. do $C[A[j]] \leftarrow C[A[j]] + 1$
5. $\triangleright C[i]$ now contains the number of elements equal to $i$.
6. for $i = 1$ to $k$
   7. do $C[i] \leftarrow C[i] + C[i - 1]$
8. $\triangleright C[i]$ now contains the number of elements less than or equal to $i$.
9. for $j = \text{length}[A]$ downto $1$

$O(k)$

$O(n)$

$O(n + k)$

---

$10^{10}$ ints from 1 to 9

$O(10^{10} + 9)$ better than $O(10^{10} \log(10^k))$

ints from 1 to $10^3$

$O(10^3 + 10^n)$ vs. $O(10^3 \log(10^3))$

$O(10^{10})$ mem. vs. $O(10^3)$ mem.
Work for data that can be mapped to ints.

E.g. English words <-> base int

\[ \text{cliff} = 2 \times 27^4 + 12 \times 27^3 + 9 \times 27^2 + 6 \times 27 + 6 \]

Alphabetic class:

\[ 18 \quad \mathfrak{e} \quad 27 \quad \mathfrak{g} \quad 10 \]

\[ 18 \quad \mathfrak{e} \quad 27 \quad \mathfrak{g} \quad 10 \]

Radix 1..b^d \quad \mathcal{O}(d(n+b))

Names

100 1.27^8 C.S. 27^8+100

R.S. 18(100+22)

R.S. useful for large ints, words