CS 4102: Algorithms
Workshop 12: Sorting Algorithms

David Wu
Fall 2020
Homework

Hashing/Sorting Exercise 2 due October 5

• Written (use LaTeX!)
• Sorting and selection algorithms
Important properties of sorting algorithms:

**Run Time**
- Asymptotic Complexity
- Constants

**In Place**
- Only requires constant additional space

**Adaptive**
- Faster if list is nearly sorted

**Stable**
- Equal elements remain in original order

**Parallelizable**
- Runs faster with many processors

**Relaxed definition:** only need to copy a constant number of elements.
Merge Sort

Divide:
• Break $n$-element list into two lists of $n/2$ elements

Conquer:
• If $n > 1$: Sort each sublist recursively
  • If $n = 1$: List is already sorted (base case)

Combine:
• Merge together sorted sublists into one sorted list

Run Time?
$O(n \log n)$
Optimal!

In Place?
No
Adaptive?
No
Stable?
Yes
Parallelizable?
Yes
**Bubble Sort**

**Idea:** Iterate through list, swapping adjacent elements if out of order, repeat until sorted
Bubble Sort

**Idea:** Iterate through list, swapping *adjacent elements* if out of order, repeat until sorted

- **In Place?** Yes
- **Adaptive?** No

**Run Time?** $O(n^2)$
( Constants worse than insertion sort)

“Compared to straight insertion [...], bubble sorting requires a more complicated program and takes about twice as long!” – Donald Knuth
How Adaptive is Bubble Sort?

Idea: Iterate through list, swapping adjacent elements if out of order, repeat until sorted

Only makes one “pass” if list is already sorted

After one “pass:”

Still requires $n$ passes, thus is $\Omega(n^2)$
**Bubble Sort**

**Run Time?**

\( O(n^2) \)

( Constants worse than insertion sort)

**Idea:** Iterate through list, swapping adjacent elements if out of order, repeat until sorted

**In Place?** Yes  

**Adaptive?** No  

**Stable?** Yes  

**Parallelizable?** No

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems" – Donald Knuth, *The Art of Computer Programming*
Worst case running time is the longest execution path (measures number of comparisons) – this is the **height** of the decision tree.
Show that finding the minimum
of an unordered list requires $\Omega(n)$ comparisons

Suppose (toward contradiction) that there is an algorithm for that does fewer than $n/2 = \Omega(n)$ comparisons.

This means there is at least one element that was not looked at.
We have no information on whether this element is the minimum or not!
Lower Bound for Merging Sorted Lists

Given two sorted lists of length \( n \), how many comparisons are necessary to merge them into a single sorted list?

\[
\begin{array}{cccccc}
1 & 3 & 7 & 9 & 10 \\
2 & 6 & 8 & 12 & 13 \\
1 & 2 & 3 & 6 & 7 & 8 & 9 & 10 & 12 & 13
\end{array}
\]
Given two sorted lists of length $n$, how many comparisons are necessary to merge them into a single sorted list?

**Claim:** these elements must be compared

**Otherwise, we can swap**

**Implication:** adjacent elements that are “split” must be compared
Array is $k$-sorted if it can be divided into $k$ blocks of equal size such that all elements in each block are larger than elements in all previous blocks.

$k$-Sorting Algorithm:
- Find median of list
- Partition the list about the median element
- $\left(\frac{k}{2}\right)$-sort the left and right sublists

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Base case: $n \leq n/k$

$$T(n) = \Theta(n \log k)$$
**k-Sorting Lower Bound**

Worst case running time is the longest execution path (measures number of comparisons) – this is the **height** of the decision tree.

\[
\log(n!) \quad \Omega(n \log n)
\]

Possible execution path

<table>
<thead>
<tr>
<th>Possible comparison</th>
<th>Result of comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&gt;)</td>
<td>(\neq)</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>(\neq)</td>
</tr>
</tbody>
</table>

Permutation of original list

| [1,2,3,4,5] | [2,1,3,4,5] | ... | [5,2,4,1,3] | ... | [5,4,3,2,1] |

How many permutations?
$k$-Sorting Lower Bound