## COMP 102: Computers and Computing <br> Lecture 9: Sorting

Instructor: Kaleem Siddiqi (siddiqi@cim.mcgill.ca)

Class web page: www.cim.mcgill.ca/~siddiqi/102.html

## On the usefulness of sorting

- Recall our example last week about finding the minimum.
- At the end of class, we talked about the number of operations.
- If the list is in increasing order:
- MinValue gets assigned only once.
- If the list is in decreasing order:
- MinValue gets assigned K times.
- If the list is in random order:
- A bit harder to estimate how many times we might need to assign MinValue.


If we are going to use the list many times, it may be better to sort it first!

## Sock Matching

- We've got a basketful of mixed up pairs of socks.
- We want to pair them up reaching into the basket as few times as we can.



## Sock Sorter A

- Strategy: Repeat until basket is empty
- Grab a sock.
- Grab another.
- If they don't match, toss them back in the basket.
- Will this procedure ever work?
- Will it always work?


## Measuring Performance

- Let's say we have 8 pairs of socks.
- How many times does this strategy reach into the basket?
- Min?
- Max?
- Average?
- How do these values change with increasing numbers of pairs of socks?


## Sock Sorter B

- Strategy: Repeat until basket is empty
- Grab a sock.
- Is its match already on the bed?
- If yes, make a pair.
- If no, put it on the bed.


## Measuring Performance

- Once again, assume we have 8 pairs of socks.
- How many times does this strategy reach into the basket?
- Min?
- Max?
- Average?
- How do these values grow with increasing numbers of pairs of socks?
- How does this compare with Sock Sorter A?


## Comparing Algorithms

## Repeat For Each Sock

## sockA

 sockB

- Do you have a matching pair? Set it aside.
- Do you have a nonmatching pair? Put them back in the basket.
- Is there a match on the table? Pair them and set the pair aside.
- Otherwise, find an empty place on the table and set the sock down.


## Notable if No Table

- Sock Sorter B seems like it is faster.
- One disadvantage of Sock Sorter B is that you must have a big empty space.
- What if you can only hold 2 socks at a time?


## Sock Sorter C

- Strategy: Repeat until basket empty
- Grab a sock.
- Grab another.
- Repeat until they match:
- Toss second sock into the basket.
- Grab a replacement.


## Measuring Performance

- Once again, let's imagine we have 8 pairs of socks.
- How many times does this strategy reach into the basket?
- Min?
- Max?
- Average?
- How do these values grow with increasing numbers of pairs of socks?


## Comparing Algorithms

## Round \#2

## sockA



## sockC

- Do you have a matching pair? Set it aside.
- Do you have a nonmatching pair? Put them both back in the basket.
- Do you have a matching pair? Set it aside.
- Do you have a nonmatching pair? Put one back in the basket.


## Analysis of Sock Sorter C

- Roughly the same number of matching operations as Sock Sorter A, but since it always holds one sock, roughly half the number of socks taken out of the basket.


## Algorithms

- Sock Sorter A, Sock Sorter B and Sock Sorter C are three different algorithms for solving the problem of sock sorting.
- Different algorithms can be better or worse in different ways.
- Number of operations
E.g. total \# of times reaching into basket, total \# of comparisons.
- Amount of memory
E.g. \# of socks on the bed (or in the hand) at any given time.


## Lessons Learned

- Given a notion of "time" (\# instructions to execute) and "space" (amount of memory), you can compare different algorithms.
- It's important to use a good algorithm!
- It's especially important to think how "time" and "space" change, as a function of the size of the problem (i.e. \# pairs of socks).


## Sorting Lists

- Many problems of this type! This is an important topic in CS.
- Sorting words in alphabetical order.
- Ranking objects according to some numerical value (price, size, ...)


## Unsorted / Sorted

```
262, 201, 918, 301, 187, 762, 397, 277, 645, 306,
765, 798, 689, 867, 276, 402, 124, 545, 907, 569,
259, 152, 399, 481, 977, 947, 774, 727, 292, 285,
173, 599, 464, 212, 147, 696, 242, 559, 155, 569,
806, 784, 415, 321, 820, 126, 469, 225, 646, 438
124, 126, 147, 152, 155, 173, 187, 201, 212, 225,
242, 259, 262, 276, 277, 285, 292, 301, 306, 321,
397, 399, 402, 415, 438, 464, 469, 481, 545, 559,
569, 569, 599, 645, 646, 689, 696, 727, 762, 765,
774, 784, 798, 806, 820, 867, 907, 918, 947, 977
```


## Sorting web pages



Web
Sorting algorithm - Wikipedia, the free encyclopedia
In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical ...
en.wikipedia.org/wiki/Sorting_algorithm - 90 k - Cached - Similar pages
Quicksort - Wikipedia, the free encyclopedia
Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that .... One
advantage of parallel quicksort over other parallel sort algorithms is ...
en.wikipedia.org/wiki/Quicksort - 74k - Cached - Similar pages

## Sorting Algorithms Demo

The following applets chart the progress of several common sorting algorithms while sorting an array of data using in-place algorithms. ...
www.cs.ubc.ca/~harrison/Java/sorting-demo.html - 11k - Cached - Similar pages

## Sorting Algorithms

Description, source code, algorithm analysis, and empirical results for bubble, heap,
insertion, merge, quick, selection, and shell sorts.
linux.wku.edu/~lamonml/algor/sort/sort.html -9k - Cached - Similar pages

## Sorting Algorithms

Shows the number of comparisons, performed by the sorting algorithm. ... 4. Shows the
code listing of the performed sorting algorithm. ...
maven.smith.edu/~thiebaut/java/sort/demo.html - 3 k - Cached - Similar pages

## Sorting Algorithms

Overview of many sorting techniques and corresponding links.
www.softpanorama.org/Algorithms/sorting.shtml - 67 k - Cached - Similar pages

## Sorting arrays

- Consider an array containing a list of names:

| Lindsey |
| :---: |
| Christopher |
| Nicholas |
| Erica |
| Rahul |
| Joelle |

- How can we arrange them in alphabetical order?


## A simple way to sort: Bubble sort

- Compare the first two values. If the second is lower, then swap.
- Continue with the 2nd and 3rd values, and so on.
- When you get to the end, start again.
- Repeat until no values are swapped.



## Let's think about Bubble sort

- Is this a good way to sort items?
- Simple to implement. This is good!
- Guaranteed to find a fully sorted list. This is good too!
- How do we decide whether it's a good method?


## Useful things to consider

- How long will it take?
- How much memory will it take?
- Is there a way we can measure this?
- Best criteria:
- number of basic machine operations: move, read, write data
- amount of machine memory
- Can we think of something similar, at a higher level?


## Predicting the "cost" of a sorting program

- Number of pairwise comparisons

For Bubble sort:

- Let's say $n$ is the number of items in the array.
- Need $n-1$ comparisons on every pass through the array.
- Need $n$ passes in total (at most).
- So $n^{*}(n-1)$ pairwise comparisons.
- Amount of memory we need (in addition to the original array)

For Bubble sort:

- Everything happens within the original array.
- Need to keep track of the index of the current item being compared.
- Need to keep track, during each pass, of whether a swap was done.
- So only need to remember two quantities, an integer and a boolean.


## A more intuitive sort method: Selection sort

- Scan the full array to find the first element, and put it into 1st position.
- Repeat for the 2nd position, the 3rd, and so on until array is sorted.



## What is the "cost" of Selection sort?

- Number of pairwise comparisons
- Let's say $n$ is the number of items in the array.
- Need $n$ - 1 comparisons on the 1 st pass through the array.
- Need $n-2$ comparisons on the 2nd pass through the array.
- And so on until we reach the last two elements.
- So in total: $(n-1)+(n-2)+(n-3)+\ldots+1=n *(n-1) / 2$
- This is better than Bubble sort! (But only by a factor of 2 .)
- Amount of memory we need (in addition to the original array)
- Everything happens within the original array.
- Need to keep track of the index of the current item being compared.
- Need to keep track, during each pass, of the index of the best value found so far.
- So only 2 integers in extra memory. Almost the same as Bubble sort.


## Why do we care about the "cost"?

- Need to know whether we can use our program or not!
- Can we use Selection sort to alphabetically sort the words in the

English Oxford dictionary?

- About 615,000 entries in the 2nd edition (1989).
- So we would need 189 trillion pairwise comparisons!
- What if we try to sort websites according to hostnames:
- About 62.4 million active domain names (as of December 2007).
- So we would need $1.95 * 10^{15}$ pairwise comparisons!
- Fortunately, not much "extra" memory is needed :-))


## Let's find a better way: Merge sort

- Divide-and-Conquer! (This is our old friend "Recursion".)
- Main idea:

1. Divide the problem into subproblems.
2. Conquer the sub-problems by solving them recursively.
3. Merge the solution of each subproblem into the solution of the original problem.

- What does this have to do with sorting?


## Merge sort

- Example:
- Sort an array of names to be in alphabetical order.
- Algorithm:

1. Divide the array into left and right halves.
2. Conquer each half by sorting them (recursively).
3. Merge the sorted left and right halves into a fully sorted array.

## Merge sort: An example



## Another example of Merge sort

- Consider sorting an array of numbers:



## Let's think about Merge sort

- Possibly harder to implement than Bubble sort or Selection sort.
- Number of pairwise comparisons:
- How many times we divide into left/right sets? At most $\log _{2}(n)$.
- How many items to sort once everything is fully split? None!
- How many comparisons during merge, if subsets are sorted?
- Need about $n$ comparisons if sorted subsets have $n / 2$ items each.
- So in total: $n$ comparisons per level ${ }^{*} \log _{2}(n)$ levels $=n * \log _{2}(n)$
- This is better than Bubble sort and Selection sort (by a lot)!
- Amount of memory we need (in addition to the original array):
- Every time we merge 2 lists, we need extra memory.
- For the last merge, we need a full $n$-item array of extra memory.
- This is worse than Bubble sort and Selection sort, but not a big deal.
- We also need 2 integers ( 1 for each list) to keep track of where we are during merging.


## Merge sort is a bargain!

- Using Merge sort to alphabetically sort the words in the English Oxford dictionary.
- Recall: about 615,000 entries in the 2nd edition (1989).
- So we would need 11.8 million pairwise comparisons.
- Versus 1.89 trillion if using Selection sort!
- Using Merge sort to organize websites according to hostnames:
- Recall: about 62.4 million active domain names (as of December 2007).
- So we would need 1.6 billion pairwise comparisons.
- Versus $1.95^{*} 10^{15}$ if using Selection sort!


## Quick recap on the number of operations

- Number of operations (y) as a function of the problem size (n)
- Constant:
- Linear:
- Log-linear: $\quad y=n * \log _{2}(n)$
- Quadratic:
- Exponential:

Bubble sort and Selection sort take a quadratic number of comparisons.

- This is as bad as it gets, for sorting algorithms.
- Merge sort takes a linear*log number of comparisons.
- This is as good as it gets, for sorting algorithms.
- This is a worst-case analysis (i.e. maximum number of operations.)


## A word about memory

- Merge sort uses twice as much memory as Selection sort.
- This is not a big deal. If you can store the array once, you can probably store it twice.
- But computers have 2 types of memory:
- RAM (Rapid-access memory) and Disk memory.
- RAM is much faster, but usually there is less of it.
- As long as everything fits into RAM, no problem!
- If array is too large for RAM, then you need to worry about:
- Number of times sections of the array are copied / swapped to and from disk.


## Take-home message

- Sorting is one of the most useful algorithms.
- Applications are everywhere.
- There are many ways to solve a problem.
- For sorting: Bubble sort, Selection sort, Merge sort, and many more.
- Some methods use $n^{*} \log _{2}(n)$ comparisons and no extra memory!
- When choosing an algorithm to solve a problem, it's important to think about the cost (time and memory) of this algorithm.
- It's also useful to think about how "easy" the algorithm is to program (more complicated $=$ more possible mistakes), but this is harder to quantify.


## Final comments

- Some material from these slides was taken from:
- http://www.sable.mcgill.ca/~clump/comp202/
- http://www.mcb.mcgill.ca/~blanchem/250/
- http://www.cs.rutgers.edu/~mlittman/courses/cs442-06/

