9. Sorting

Please read Part II Sorting and Order Statistics in (Corman et al., 2009).

Ineffective Sorts

Sorting — Basics

Problem 4: Sorting

Decision Problem: Given a list of keys, is it sorted?
Function Problem: Given a list of keys, sort it into ascending (or descending) order.

Pretend you are given a *file of records* containing *keys* that can be ordered. That is, the keys are from a totally ordered set where the

common relations and functions

$$(<), (\leq), (=), (\geq), (>), (\neq), \text{min}, \text{max}$$

are defined on the keys.

The function `sorted` *witnesses* that a list is sorted in ascending order or not.

What is the time complexity of `sorted`?

Listing 23: The Sorted Decision Problem

106 ⟨*Sorted list or not?* 106⟩≡

```
sorted :: (Ord a) => [a] -> Bool
sorted []      = True
sorted [x]     = True
sorted (x:y:ys) = (x <= y) && sorted (y:ys)
```

The `sorted` function has time complexity $O(n)$ and this shows the `Sorted` can be decided in polynomial time. That is, `Sorted` is in NP, the class of decision problems that can be solved in polynomial time using a non-deterministic Turing machine. Of course, the function problem: Sort this list, can be solved in polynomial time and is in P,

Two factors dominate the time spent sorting by comparing keys.

1. The number of comparisons made, and
2. The amount of data moved

When an algorithm requires duplicating the records, space complexity can become an issue too.

- Some comparison-based sorting algorithms have $O(n^2)$ worst case time complexity. For example: bubble, insertion, and selection sorts.
- Other comparison-based sorting algorithms have $O(n \lg n)$ worst case time complexity. For example: merge and heap sorts.
- Quicksort is famous for almost always being fastest, but in some races it does not win.

Sentinels: To terminate a sort, some algorithms benefit from *sentinels* at an ends of records ($A[0]$ or $A[n+1]$). Sentinels are typically below or above every valid value.

Comparison sorts: The most common sorts require comparing keys. The lower bound time complexity for comparison sorts is $\Omega(n \lg n)$.

Sorts without Compares: Time complexity can be reduced when data properties are known. There are several algorithms that sort in $O(n)$ time without comparing keys. For example, counting, radix, and bin sorts.

Internal and External Sorts: Sorting algorithms can also be classified by the size of the file to be sorted. Internal sorting processes files that fit into main memory. External sorting sorting processes files too large to fit in main memory. The files are stored in external memory: magnetic tapes, disk, or on a network (in the cloud).

Stability: A sorting algorithm is *stable* if it preserves relative order of equal keys. For example, if an alphabetized file of names is sorted by salary, those names with the same salary will remain in alphabetical order.

Sorting Algorithms

The following algorithms are presented and analyzed in these notes.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Merge Sort
- Quicksort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket (Bin) Sort

Some nice demonstrations of several sorting algorithms can be found at this external site. Know thy complexities! Below is a list of sorting algorithms and their complexities. I found it here. The value n is the length of the list (number of keys in the file) to be sorted.

I don't know Timsort, perhaps you can educate me. See Algorithmics on 201.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Timsort	$O(n)$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Heapsort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$O(n)$	$O((n\log(n))^2)$	$O((n\log(n))^2)$	$O(1)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

A main takeaway from the following notes should be: Small case analysis can lead to general expressions and understanding.

Bubble Sort

The heuristic is: Repeatedly pass through the records exchanging adjacent elements that are out of order. When no exchanges are needed

Bubble sort is infamous for being clear but slow. Although bubble sort is maligned, it is a useful example to learn about algorithm analysis.

the list is sorted. The insight for bubblesort is the `bubble` function that maps a list into a tuple containing a *bubbled* list and a Boolean flag indicating whether an exchange was made or not.

Bubble

Before learning to sort, learn to bubble. The `bubble` function will move the smallest value in a list to the head.

Listing 24: Functional Bubbling

A pattern for defining a function is shown in the code outline.

108a ⟨*Bubble smallest to head* 108a⟩≡
⟨*Define the function's type* 108b⟩
⟨*Define base computations* 108c⟩
⟨*Define the recursion* 108d⟩

The `bubble` function maps a list `[a]` of values from a totally ordered set to an ordered pair: A permutation of the input list and a Boolean flag that signals if the input was changed or not. Not explicit in the type declaration is the post condition: The smallest value in the input list is the head of the output list.

108b ⟨*Define the function's type* 108b⟩≡

```
bubble :: Ord a => [a] -> ([a], Bool)
```

As base cases, the empty list and a singleton are sorted and need no bubbling or changes to the list.

108c ⟨*Define base computations* 108c⟩≡

```
bubble [] = ([], False)
bubble [x] = ([x], False)
```

Otherwise, on a longer list `(x:xs)`, there are two objectives: Permute the input, if necessary, so its smallest value becomes the head of the list. And, record whether or not the input list was changed.
To bubble the smallest value to the head in a list `(x:xs)`, bubble the tail `xs` returning a permuted list `(y:ys)` where `y` is the smallest value in `xs`. If the permutation is the identity, the `changed` flag is `False`, otherwise `changed = True`.
Now that `x` and `y` are the two smallest values in `(x:xs)`, they can be ordered, setting and returning the `changed` flag appropriately.

108d ⟨*Define the recursion* 108d⟩≡

```
bubble (x:xs) =
  let (y:ys, changed) = bubble xs
  in if x > y
  then (y:x:ys, True)
  else (x:y:ys, changed)
```

It may or may not be clear that the time complexity of `bubble` is

Inductive and equational reasoning is useful here. Trace the executions

- bubble [2, 5] =
  - let ([5], False) = bubble [5]
  - in if 2 > 5
  - then ([2, 5], False)
- bubble [7, 2, 5] =
  - let ([2, 5], False) = bubble [2, 5]
  - in if 7 > 2
  - else ([2, 7, 5], True)
- bubble [4, 7, 2, 5]
  - let ([2, 7, 5], True) = bubble [7, 2, 5]
  - in if 4 > 2
  - then ([2, 4, 7, 5], True)

Convince yourself: One call to `bubble zs` places the smallest value in `zs` at the head of the list.
A second call to `bubble zs` places the next smallest value in `zs` in the second position of the list.
Each subsequence call places another value it it correct position.

$O(n)$ where n is the length of the input list.

The algorithm shows when `bubble` is called on a list of size n, it recursively calls itself on a list of size $n-1$. The additional work takes constant time: Comparing x and y; and inserting values into the head head of `ys`. The time complexity of `bubble` can be described by the recurrence

$$T(n) = T(n-1) + c, \quad T(0) = c$$

Which has solution

$$T(n) = c(n+1) = O(n)$$

By induction: If $T(n) = c(n+1)$, then $T(0) = c = (0+1)c$. And, by substitution, $T(n-1) = cn$ for $n \geq 1$. Therefore, the equation $T(n) = c(n+1) = cn + c = T(n-1) + c$ is satisfied.

Now `bubble` can be used repeated to sort a list.

Listing 25: Functional Bubble Sort

109a ⟨*Functional bubble sort* 109a⟩≡

```
bubble_sort :: Ord a => [a] -> [a]
bubble_sort xs =
   let (zs, changed) = bubble xs
   in if changed
      then bubble_sort zs
      else zs
```

Each call to `bubble` places at least one element where it belongs. Therefore, `bubble_sort` will be called at most n times on a list of length n.

- In the best case, `bubble_sort xs` calls `bubble xs` which returns `(zs, False)`, and `bubble_sort` immediately returns `zs`. In this case the time complexity is $T(n) = c(n+1)$.
- In the worst case, `bubble xs` returns `(zs, True)` each of $n-1$ times and `(zs, False)` on the last pass. In this case, the time cost is $T(n) = cn(n+1) = O(n^2)$

Here is an imperative implementation of bubble sort. It is naive in that it does not halt once it determines the array is sorted

Listing 26: Imperative Bubble Sort

109b ⟨*Bubble sort: C version* 109b⟩≡

```
#include <stdio.h>

void bubblesort(int A[], int n)
{
  int tmp; // for swapping
  for (int i = n-1; i > 0; i-) {
```

```
      for (int j = 1; j <= i; j++) {
        if (A[j-1] > A[j]) {
           tmp = A[j-1];
           A[j-1] = A[j];
           A[j] = tmp;
        }
      }
    }
  }

  int main () {
     int A[10] = {21, 8, 13, 55, 34, 5, 3, 2, 0, 1};
     bubblesort(A, 10);
     for (int i = 0; i < 10; i++) {
        printf("A[%d] = %d \n", i, A[i]);
     }
  }
```

Bubble Sort – Analysis of Complexity

- Bubble sort uses about $n^2/2$ compares and $n^2/2$ data exchanges in the worst and average cases.
- The comparison `A[j-1] > A[j]` is always executed inside the `for` loops on `i` and `j`. The cost of the compares can be calculated using summation notation

$$\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}$$

  Bubble sort always makes $O(n^2)$ compares.
- In the worst case for swaps, the file is in reverse order, and $O(n^2)$ swaps are required.
- In the best case for swaps, the file is in sorted order, and no swaps are required.
- In the average case for swaps, we need the probability that the `if` test evaluates to `True`.

Example: Bubble sort operations

Consider the six permutations of $\{2, 4, 7\}$. *Pretend each permutation of the keys occurs with equal likelihood* $1/6$.

$i = 2$, $j = 1$		$i = 2$, $j = 2$		$i = 1$, $j = 1$		
order	*swaps*	*reorder*	*swaps*	*reorder*	*swaps*	*reorder*
2 5 7	*0*		*0*		*0*	
2 7 5	*0*		*1*	*2 5 7*	*0*	
5 2 7	*1*	*2 5 7*	*0*		*0*	
5 7 2	*0*		*1*	*5 2 7*	*1*	*2 5 7*
7 2 5	*1*	*2 7 5*	*1*	*2 5 7*	*0*	
7 5 2	*1*	*5 7 2*	*1*	*5 2 7*	*1*	*2 5 7*
$P(swap) = \frac{1}{2}$		$P(swap) = \frac{2}{3}$		$P(swap) = \frac{1}{3}$		

There are (3+4+2) = 9 swaps for the 6 cases. The average number of swaps is

$$\frac{9}{6} = \frac{3}{2} = \frac{3(3-1)}{4} = \frac{n(n-1)}{4}$$

Let's see if this example provides the general rule.

- On the first pass of the outer loop $i = (n-1)$ and $j = 1, \ldots, (n-1)$. the `if` test will be `True` at any given $1 \leq j \leq i$ if and only if

$$\max\{\{A[0], A[1], \ldots, A[j-1]\} > A[j]$$

- This will be `True` if the largest value from the set

$$\{\{A[0], A[1], \ldots, A[j]\}$$

  is in any of the first `j` of `j+1` positions

- Therefore, the probability that the `if` test evaluates to `True`, during the first pass on `i=n-1` for $j = 1, 2, \ldots, (n-1)$ is

$$\frac{j}{j+1} = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots, \frac{n-1}{n}$$

- On the next pass of the outer loop $i = (n-2)$ and the `if` test will be `True` if and only if

$$\max\{\{A[0], A[1], \ldots, A[j-1]\} > A[j]$$

  and

$$\max\{\{A'[0], A'[1], \ldots, A'[j]\} > \mathtt{A}'[j+1]$$

  where the primed values refer to the original array values

- The probability that the `if` test evaluates to `True` on this second pass is

$$\begin{aligned}\frac{j+1}{j+2}\cdot\frac{j}{j+1} &= \frac{j}{j+2}\\ &= \frac{1}{3},\frac{2}{4},\frac{3}{5},\ldots,\frac{n-2}{n} \quad \text{for } j=1,\ldots,n-2\end{aligned}$$

- In general, the probability that the `if` test evaluates to `True` on the kth pass $k = 1, 2, \ldots, (n-1)$ where $i = n-k$ is

$$\frac{j}{j+k}, \quad \text{for } j=1,\ldots,n-k$$

- Average case cost is computed by the formula

$$\sum_{\text{all cases}} \text{Prob(case)}\cdot\text{Work(case)}$$

- The average number of swaps is

$$\sum_{i=1}^{n-1}\sum_{j=1}^{i}\frac{j}{j+(n-i)}(1) = \begin{array}{lllllll|l}
\frac{1}{2} & + & \frac{2}{3} & + & \frac{3}{4} & + & \cdots + & \frac{n-1}{n} & i=n-1,\ j=1,\ldots(n-1)\\
 & + & \frac{1}{3} & + & \frac{2}{4} & + & \cdots + & \frac{n-2}{n} & i=n-2,\ j=1,\ldots(n-2)\\
 & & & + & \frac{1}{4} & + & \cdots + & \frac{n-3}{n} & i=n-3,\ j=1,\ldots(n-3)\\
 & \vdots & & & & & & & \\
 & & & & & & + \frac{1}{n-1} + & \frac{2}{n} & i=2,\ j=1,2\\
 & & & & & & + & \frac{1}{n} & i=1,\ j=1
\end{array}$$

$$\begin{aligned}
&= \frac{1}{2}+\frac{1}{3}(1+2)+\frac{1}{4}(1+2+3)+\cdots\frac{1}{n}(1+2+3+\cdots+(n-1))\\
&= \frac{1}{2}+\frac{1}{3}\left(\frac{2\cdot 3}{2}\right)+\frac{1}{4}\left(\frac{3\cdot 4}{2}\right)+\cdots\frac{1}{n}\left(\frac{n(n-1)}{2}\right)\\
&= \frac{1}{2}(1+2+3+\cdots+(n-1))\\
&= \frac{n(n-1)}{4}
\end{aligned}$$

Exercises

1. Consider the insertion sort algorithm below. Assume $A[0]$ is a *sentinel*, smaller every other element in the array.

Listing 27: Imperative Insertion sort

113 ⟨*Insertion sort* 113⟩≡

```
void InsertionSort(int A[], int n)
{
  int i, j, v;
  for (i = 2; i < n; i++) {
    v = A[i];
    j = i;
    while (A[j-1] > v) {
      A[j] = A[j-1];
      j = j - 1;
    }
    A[j] = v;
  }
}
```

(a) Compute the average number of comparisons (executions of the `while` loop) on the 6 permutations of $\{0, 2, 5, 7\}$, where 0 is held fixed as the first element, the sentinel. Construct a table, similar to the table in example to show your computations.

(b) Find a general formula for the average number of comparisons in the inserttion sort algorithm.

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15.

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* C *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147], [page 163]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355.

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857.

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index