CSE 101 - ALGORITHMS - SUMMER 2000
Lecture Notes 3

Monday, July 10, 2000.

Chapter 3

Sorting

We saw in Chapter 1 that the sorting problem can be formulated as follows:

SORTING

INPUT: A sequence A of n numbers $\langle a_1, a_2, ..., a_n \rangle$.

OUTPUT: A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots a'_n$.

We also presented and analysed two common sorting algorithms, insertion and merge sort, the first running in $\Theta(n)$ best-case time and $O(n^2)$ worst-case time, while the second running in $\Theta(n \log n)$ (both best-case and worst-case) time.

In this chapter we'll list a series of classical sorting algorithms, organized in three categories depending on their running times: $O(n^2)$, $\Theta(n \log n)$, and linear time. The algorithms in linear time presented in Section 3.3 assume that the input elements satisfy auxiliary properties. Section 3.4 approaches two problems related to sorting, namely that of finding an arbitrary element a in an array and that of finding the i-th element in increasing order in an arbitrary array. Once the array is sorted, an element can be found in $O(\log n)$ by sa called binary search and the i-th element can be found in $\Theta(1)$ time, so the immediate solutions for these problems would run in the same time as ordinary sorting, that is, $O(n^2)$ or $\Theta(n \log n)$; we'll see that the second problem can be solved much better, in linear time.

You may want to additionally read the section 8.3.1 on sorting in the text book for usual trade-offs one should be aware of when choosing a sorting algorithm in practice.

3.1 Sorting in $O(n^2)$ Time

It would be an interesting experiment to look at sorting algorithms designed by students for the first time, without any background in algorithms. They often end up writing correct algorithms which run in $O(n^2)$ time, some of them even descovering insertion sort or selection sort by themselves. I remember (with nostalgy) my first sorting algorithm; as a mathematician, I designed it following rigorously the formal definition of a non-decreasingly sorted array: $A[1, ..., n]$ is *sorted* if and only if for any two indexes $1 \leq i, j \leq n$, if $i < j$ then $A[i] \leq A[j]$. Then an obvious algorithm is to generate all pairs of indexes and test the condition above; if it is not satisfied by two indexes then enforce it, that is, swap the elements associated with the two indexes. This is what I did:

TRIVIAL-SORT(A)

for $i \leftarrow 1$ **to** n **do** (n is the length of A)
  for $j \leftarrow 1$ **to** n **do**
    if $i < j$ and $A[i] > A[j]$ **then** $swap(A[i], A[j])$

The running time of this algorithm is obviously $\Theta(n^2)$, but is it correct?

Exercise 3.1 Show that TRIVIAL-SORT(A) is correct, that is, that it indeed sorts the array A.

In fact, I'd say now that I was lucky that my algorithm was correct! That's because I actually had in mind the following algorithm:

WRONG-SORT(A)
for all $1 \leq i, j \leq n$ **do**
  if $i < j$ and $A[i] > A[j]$ **then** $swap(A[i], A[j])$

and I only chose a particular way to generate all the pairs of indexes $1 \leq i, j \leq n$, which fortunately yielded a correct algorithm.

Exercise 3.2 Show that WRONG-SORT(A) may not be correct for some enumerations of the n^2 pairs of indexes $1 \leq i, j \leq n$.

3.1.1 Selection Sort

Selection sort is one of the most intuitive sorting algorithms, still pretty efficient in practical situations. It can be simply described in english words as follows: *select* the smallest element of A, output it, remove it from A, and then iterate until there are no elements left in A. More formally,

GENERIC-SELECTION-SORT(A)
for $i \leftarrow 1$ **to** n **do**
  $A'[i] \rightarrow minimum(A)$
  $remove\_minimum(A)$
return A'

Various selection sort algorithms can be obtained depending on how the minimum element of A is found and how it is removed, that is, depending on what data structure is used. In Subection 3.2.2 we'll see a version of this algorithm, called heap sort, which runs in $\Theta(n \log n)$ where these operations are implemented by a heap. For now, let un consider a more common implmentation of selection sort, which can be easily adapted from the TRIVIAL-SORT algorithm presented before, noticing that the pairs of indexes (i, j) with $i \geq j$ are not needed:

SELECTION-SORT(A)
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    if $A[i] > A[j]$ **then** $swap(A[i], A[j])$

Notice that the auxiliary array A' is not needed, because we can reuse A.

Exercise 3.3 Show that the running time of SELECTION-SORT is $\Theta(n^2)$.

3.1.2 Insertion Sort

Insertion sort is one of the simplest examples of *incremental insertion*, a usual technique in the design of algorithms where in order to create a complex data structure of n elements, one first creats it for $n - 1$ elements and then makes appropriate changes to insert the n-th element.

We have already seen two versions of insertion sort in the first lecture, both running in worst-case $O(n^2)$, but one in best-case $O(n^2)$ while the other in best-case $O(n)$. For the sake of diversification, we'll next present another one, running in the same worst-case time and also in $O(n)$ best-case time:

INSERTION-SORT(A)
for $i \leftarrow 1$ **to** $length[A] - 1$ **do**
  $j \leftarrow i + 1$
  while $j > 1$ and $A[j] < A[j - 1]$ **do** $swap(A[j], A[j - 1])$

Exercise 3.4 Show that the best-case running time of INSERTION-SORT is $O(n)$ and that its worst-case running time is $O(n^2)$.

3.1.3 Quick Sort

Quick sort proves to bethe fastest sorting algorithm in practice, even though its worst-case running time is $\Theta(n^2)$. That's because its average-case running time is $O(n \log n$ with a very low constant. Like merge sort, quick sort is a divide and conquer algorithm:

QUICK-SORT(A, i, k)
if $i \geq k$ **then return**
$j \leftarrow$ PARTITION(A, i, k)
QUICK-SORT(A, i, j)
QUICK-SORT$(A, j + 1, k)$

where PARTITION(A, i, k) partitions the array $A[i, ..., k]$ by rearanging it in two subarrays $A[i, ..., j]$ and $A[j+1, ..., k]$ with the property that $A[j]$ is larger than or equal to each element of $A[i, ..., j]$ but smaller than or equal to each element of $A[j + 1, ..., k]$. With other words, PARTITION(A, i, k) places an element at its correct position in the sorted array and then returns its index. The following is a possible pseudocode which places $A[i]$ at its appropriate position:

PARTITION(A, i, k)
$j \rightarrow i$
for $p \leftarrow i + 1$ **to** k **do**
  if $A[p] < A[i]$ **then** $\{j \leftarrow j + 1;\ swap(A[j], A[p])\}$
$swap(A[i], A[j])$
return j

Obviously, the running time of PARTITION(A, i, k) is $\Theta(n)$, where $n = k - i$ is the number of partitioned elements.

Then QUICK-SORT$(A, 1, n)$ correctly sorts the entire array A, where n is the length of A. It is worth mentioning that the average-case running time of quick sort is $O(n \log n)$. That means that in practical situations we expect quick sort to run intime $O(n \log n)$, exactly like merge sort. An advantage of quick sort is that, unlike merge sort, it sorts the array *in place*, that is, within the given memory, A. Another advantage of quick sort in constrast with merge sort is that it visits the elements almost sequentially, thus being very suitable for actual computer systems intensively using various levels of cache memory and virtual memory.

A common variation of quick sort which proves to be efficient in practice is to *randomize* its PARTITION procedure, by adding a command of the form $swap(A[i], A[random(i, k)])$ at the beginning.

Exercise 3.5 Illustrate the execution of PARTITION$(\langle 17, 21, 8, 27, 3, 15, 21, 9, 7\rangle, 1, 9)$.

Exercise 3.6 Show that the worst-case running time of QUICK-SORT is $O(n^2)$.

Exercise 3.7 Show that the best-case running time of QUICK-SORT is $O(n \log n)$ (Hint: The best-case situation is when PARTITION splits the array in two balanced subarrays).

3.2 Sorting in $\Theta(n \log n)$ Time

- show that this is the best worst-case time in the comparison model ...

3.2.1 Merge Sort

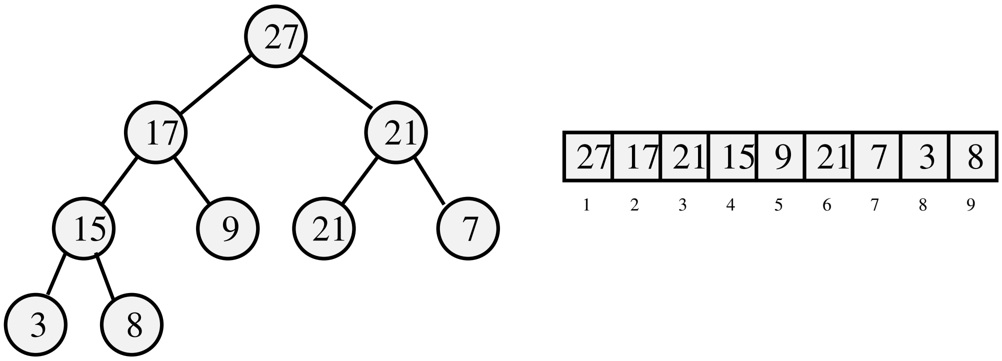
The merge sort algorithm and its running time analysis were presented in the first two chapters.

3.2.2 Heap Sort

We first introduce some preliminary notes.

Heap

A **heap** data structure is a complete binary tree having the *heap property*, that the value of each node is larger than or equal to the values of its two descendents; a heap is implemented as an array as follows:



We assume that for a given index i in the array, $left(i)$, $right(i)$ and $parent(i)$ are the corresponding indexes of its left descendent, right descendent and its parent, respectively. These three functions can be very efficiently implemented by $left(i) = 2i$, $right(i) = 2i + 1$, and $parent(i) = \lfloor i/2 \rfloor$. Then the heap property becomes: $A[parent(i)] \geq A[i]$ for each i different form 1.

Heapify

HEAPIFY is a procedure which has a crucial role in what follows. It takes as input a complete binary tree represented as an array with the property that its root's left and right subtrees are already heaps, and returns only one heap, after inserting the root at its appropriate place. The following is a pseudocode for it, where i is an index in A giving a complete subtree (as a subarray) on which the procedure is applied:

HEAPIFY(A, i)
$max \leftarrow i$
if $left(i) \leq heap\_size$ and $A[left(i)] > A[max]$ **then** $max \leftarrow left(i)$
if $right(i) \leq heap\_size$ and $A[right(i)] > A[max]$ **then** $max \leftarrow right(i)$
if $max \neq i$ **then** $\{swap(A[i], A[max])$; HEAPIFY$(A, max)\}$

where $heap\_size$ is a global variable equal to the length of A. Thus, if the root element is smaller than one of his children then it swaps the root with the largest children and recurently heapifies the subtree rooted in that children. For example, if $A = \langle 27, 8, 21, 17, 9, 21, 7, 3, 15 \rangle$ then HEAPIFY$(A, 2)$ first swaps 8 and 17 and then it swaps 8 and 15.

Exercise 3.8 Show that the running time of HEAPIFY(A, i) is $O(\log n)$, where n is the number of elements in the subtree rooted in i.

Building a Heap

In many applications, including heap sort, it is very profitable to first build a heap from an array. Once we have the procedure HEAPIFY, this task becomes easy:

BUILD-HEAP(A)
$heap\_size = length(A)$
for $i \leftarrow \lfloor length(A)/2 \rfloor$ **downto** 1 **do**
    HEAPIFY(A, i)

Exercise 3.9 Argue that after the execution of BUILD-HEAP(A), $A[1]$ is the largest element of A.

Exercise* 3.10 Show that the running time of BUILD-HEAP(A) is $\Theta(n)$, where n is the number of elements in the array A.

Heap Sort

Following a similar approach (but dual) to selection sort, we can use heaps to device a sorting algorithm as follows:

HEAP-SORT(A)
1 BUILD-HEAP(A)
2 **for** $i \leftarrow length(A)$ **downto** 2 **do**
3 $\quad swap(A[1], A[heap\_size])$
4 $\quad heap\_size \leftarrow heap\_size - 1$
5 $\quad$ HEAPIFY($A, 1$)

Step 1 runs in $O(n)$ time. Then for each index $2 \leq i \leq n$ the procedure HEAPIFY is called once on an input of size i.

Exercise 3.11 Ilustrate the execution of HEAP-SORT on the array $A = \langle 17, 21, 8, 27, 3, 15, 21, 9, 7\rangle$.

Exercise 3.12 Show that the running time of heapsort is $\Omega(n \log n)$.

Proof: It is enough to exhibit a particular input on which the running time of heapsort in $\Omega(n \log n)$. Consider an input array sorted in nonincreasing order. On this input the construction of the heap still takes $O(n)$ time. Moreover, at each step of the for loop, the current last element of the heap is moved to the root and then the procedure Heapify is run on the current heap. Notice that when the input is an array sorted in a nonincreasing order, the current last element of the heap is also the current minimum element, and thus the running time of the procedure Heapify is $\log i$, where i is the number of the iteration of the for loop executed. Thus, the total running time executed by Heapsort is

$$\Omega(n) + \sum_{i=2}^{n} \log i,$$

which is $\Omega(n) + \log(n!) = \Omega(n \log n)$.

Exercise 3.13 Prove that the running time of HEAP-SORT is $\Theta(n \log n)$.

3.3 Sorting in Linear Time

It can be shown that *any* sorting algorithm asimptotically runs in $\Omega(n \log n)$ time, so merge and heap sort are optimal. However, it is often the case in practice that the input elements are not completely random, so better sorting algorithms can be implemented.

3.3.1 Counting Sort

Suppose that the sorting problem is modified as follows:

k-BOUNDED SORTING

INPUT: A sequence $A = \langle a_1, a_2, ..., a_n\rangle$ of n numbers between 1 and k.

OUTPUT: A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n\rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots a'_n$.

Then we can write a sorting algorithm, called *counting sort*, which runs in $O(n+k)$ time; when $k \in O(n)$, in particular when $k = n$, it runs in linear time. Suppose that the elements are sorted in another array B of the same size as A, and that there is an array $F[1, ..., k]$ available which contains only zeros. The array F is used for the frequencies of the elements in A:

COUNTING-SORT(A, B, k)
for $i \leftarrow 1$ **to** n **do** $F[A[i]] \leftarrow F[A[i]] + 1$
for $i \leftarrow 2$ **to** k **do** $F[i] \leftarrow F[i] + F[i-1]$
for $i \leftarrow n$ **downto** 1 **do**
$\quad B[F[A[i]]] \leftarrow A[i]$
$\quad F[A[i]] \leftarrow F[A[i]] - 1$

Step 1 calculates in F the number of occurences of each number. Step 2 calculates for each number $A[i]$ the largest index on which the number $A[i]$ occurs in the sorted array B; the array F is reused to save memory. Then the steps 3, 4, and 5 place each number $A[i]$ on its appropriate position in B.

An important feature of the counting sort algorithm is that it is *stable*, that is, elements with the same value occur in the output in the same order as they appear in the input. This property is particularly important when the elments are more complex structures which are sorted using only a special value, or key, in that structure. For example, if the elements are points in the two dimensional plane, then they can be sorted only by one of the coordinates.

Exercise 3.14 Ilustrate the execution of COUNTING-SORT on the input $\langle 5,3,2,8,4,5,9,3,6,8,5,2,4\rangle$, when $k=9$.

Exercise 3.15 Show that the running time of COUNTING-SORT is $\Theta(n+k)$.

Exercise⋆ 3.16 Prove that COUNTING-SORT is stable.

3.3.2 Radix Sort

Radix sort is a linear sorting algorithm when the elments to be sorted can be represented on a certain number d of fields, that is, $a=(a_d,a_{d-1},...,a_1)$, having the property that for any elements $a=(a_d,a_{d-1},...,a_1)$ and $b=(b_d,b_{d-1},...,b_1)$, $a\leq b$ if and only if $a_d<b_d$ or $(a_d=b_d)$ and $(a_{d-1},a_{d-2},...,a_1)\leq(b_{d-1},b_{d-2},...,b_1)$. A typical representation is the digital representation of numbers.

The idea is to sort the n numbers by their digits, starting with the least significant digit. The following is an example:

4851		4851		6412		5117		1412
3217		2751		1412		7164		1751
6412		1751		3217		3217		2751
2751		6412		8217		8217		3217
8217		1412		5117		6412		4542
1412	$\Rightarrow$	4542	$\Rightarrow$	4542	$\Rightarrow$	1412	$\Rightarrow$	4851
1751		7164		4851		4542		5117
5117		3217		2751		2751		6412
7164		8217		1751		1751		7164
4542		5117		7164		4851		8217
$\uparrow$		$\uparrow$		$\uparrow$		$\uparrow$		

Thus, the radix sort algorithm can be more formally described as follows:

RADIX-SORT(A)
1 **for** $i\leftarrow 1$ **to** d **do**
2 sort the n elements by their i-th field using a *stable* sorting algorithm.

Counting sort is usually the best choice in step 2 because the digits are bounded in general.

Exercise 3.17 Analyse RADIX-SORT.

Exercise 3.18 Why must the sorting algorithm used by radix sort in step 2 be stable?

Exercise 3.19 Given a fixed $k>0$, show how to sort any n integers in the range 1 to n^d in $O(n)$ time. (Hint: do it for $d=2$ first).

Proof: We present the general solution. The idea is to represent the numbers in base n and then use radix sort.

Let's substract 1 from each number (it can be done in linear time), so that the numbers are in the range 1 to n^d-1. Each such number m can be uniquely decomposed as $m=m_1+m_2\cdot n+m_3\cdot n^2+\cdots m_d\cdot n^{d-1}$, where m_1 is the rest of the division of m by n, m_2 is the rest of the division of $m-(m-m_1)/n$ by n, etc. All the numbers $m_1,m_2,...,m_d$ are in the range 0 to $n-1$; we can add 1 to each in linear time. Thus each of the n numbers in the range 1 to n^d can be uniquely represented as a tuple of d numbers in the range 1 to n, $(m_d,m_{d-1},...,m_2,m_1)$.

But this is exactly the hypothesis of radix sort! Therefore, we can simply use radix sort for d "digits", with counting sort applied d times to sort the numbers by each digit. Since the digits range between 1 and n, counting sort runs in $\Theta(n+n)=\Theta(n)$ time, so the whole alogithms runs in $\Theta(d\cdot n)$, that is, $\Theta(n)$.