CSC 202 Mathematics for Computer Science
Lecture Notes

Marcus Schaefer
DePaul University[1]

Chapter 4

Sorting and Searching in Databases

Imagine you are searching for a student with last name "Johnson". You would go to the `student` table of the database and go through the list of students until you find such a student. How would a computer do this? First of all, let us assume that we have downloaded the data in the database to be used in a computer program (this is sometimes called row-processing, as opposed to set-processing, which is what the database does). The records are stored in an *array* of data, named $R[1]$ through $R[n]$ for n records. (For the student table, $n = 8$.)

For example, let's say we have downloaded the `student` table from our database; then

$R[1]$ = (Brennigan, Marcus, 90421, 987654321, UGRD, COMP-GPH, Evanston),

(assuming the records remain in the same order as they are in the database; an assumption that is not valid in general, but won't harm us). To access a particular field, take `LastName` as an example, we write $R[1]$.`LastName`. If we wanted to test whether the ith record is a student with last name "Johnson", we write

```
if R[i].LastName = "Johnson"
   return i
```

This is pseudocode for a test statement, known as `if` in most programming languages. If the condition: `R[i] = "Johnson"` is true, then the indented statement `return i` is performed, that is, the value of i is returned to whoever asked for it and the program stops; an appropriate action, since we have found "Johnson" and want to return the information in which record. If we want to search for "Johnson" through all records, we have to repeat the test for all possible values of i from 1 to n; in our pseudocode language we use the `for` statement for that task (most programming languages support `for`).

```
for i = 1 to n
   if R[i].LastName = "Johnson"
      return i
return 0 // not found
```

This program looks for "Johnson" in all records from $R[1]$ to $R[n]$. If for some i we find that `R[i].LastName = "Johnson"` the program returns i and stops. Otherwise it will run through all values of i and finish by returning 0, encoding the information that nobody in the database has last name "Johnson".

In general, we are not looking for one particular name only or a name at all, but for some value x in an array $A[1..n]$. The general *search problem* asks for an index i such that $A[i] = x$ or the value 0 if no such index exists. Here is a pseudocode solution to the search problem:

```
for i = 1 to n
   if A[i] = x
      return i
return 0 // not found
```

This algorithm is known as *linear search*, since it searches the array linearly, element by element. If we want to stress the fact that this algorithm is a procedure that can be used by other programs we can give it a name as follows:

```
procedure linear_search(A,x)
   n := length(A) // n = number of elements in A
   for i = 1 to n
      if A[i] = x
         return i
   return 0 // not found
```

A call to `linear_search`(A, x) searches for x in the array A and will return i is x equals $A[i]$ and 0 if there is no such i.

And this is the best you can do if you have no more knowledge of the array. This is not good enough for most applications, not for reasonably large databases and certainly not for web-applications that sit on top of large databases such as Amazon or Google. Imagine each access to the array takes 1ms (one millisecond). Then searching for a record in a million records might take as much as 1000s or, roughly, 16 minutes. Most people won't wait that long.

Check out a comparable situation: you are looking for a name in the Chicago residential phone book, which has about a million entries. You won't take 16 minutes to find somebody with last name "Johnson", even though you can't check 1000 records a second. The difference of course is that the entries in the phone book are sorted by last name (try finding somebody in the phone book by first name or telephone number) and you can use a technique much better than linear search; before you read on, think a minute about the procedure you use to find a name in a phone book.

So what do you do? You open the book somewhere, pick out a name and compare it to the name your are looking for, say "Johnson". If the name you

found when opening the book was "Molnar" you'd know that you have to keep looking for "Johnson" in the pages preceding "Molnar" rather than the pages following it. And that's it: you repeat this procedure until you have homed in on the record you are looking for.

Why is this quicker than looking for the entry linearly? Because in each look-up you are discarding large parts of the phone book that you no longer need to check. If you arrange things optimally you will always look in the middle of the part you haven't checked yet, since then you will be able to discard at least half of the pages you still need to check in each step. In other words you are halving the amount of work that needs to be done in each step. Let us reexpress this mathematically: if there are n entries and you halve the number of entries you need to consider in each step, you will be done in at most k steps, where k is the smallest number such that

$$n(1/2)^k \leq 1,$$

because once you are down to a single entry you are done. Rewriting this gives us

$$n \leq 2^k;$$

the smallest k solving this equation is $k = \lceil \log n \rceil$ (the ceiling operator $\lceil \cdot \rceil$ rounds the number up to the nearest integer, and $\log n$ is the logarithm to base 2 of n).

Let us see the algorithm in detail; it is called *binary search* because it splits the input in half. We write a binary search procedure `binary_search`(A, x, i, j) that searches for element x in array A between (and including) positions i and j.

```
procedure binary_search(A,x,i,j)
   if j < i // no more elements to check
      return 0
   mid := (i+j)/2 // round down
   if x = A[mid]
      return mid
   if x < A[mid]
      return binary_search(A,x,i,mid-1)
   if x > A[mid]
      return binary_search(A,x,mid+1,j)
```

To check whether x occurs in A we call `binary_search`$(A, x, 1, n)$ to get the search started. The algorithm will then check whether we have run out of elements to check (the right boundary, j, has moved to the left of the left boundary, i); if so, we return 0, since x was not found. Otherwise it computes the midpoint into a variable `mid`, then checks whether the midpoint is x. If not, it continues its search on the correct side of `mid`.

Example 4.0.1. Suppose A is the array

$$A = [3, 7, 9, 11, 14, 15, 20, 31, 44, 47, 49, 51, 78, 90, 91, 94].$$

We can present the array more visually as follows:

```
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
-----------------------------------------------------------------
| 3 | 7 | 9 | 11| 14| 15| 20| 31| 44| 47| 49| 51| 78| 90| 91| 94|
-----------------------------------------------------------------
```

Here's a transcript of the calls made to binary search when searching for $x = 14$ in A:

```
binary_search(A,14,1,16)
   mid = 8, A[8] = 31 > 14
binary_search(A,14,1,7)
   mid = 4, A[4] = 11 < 14
binary_search(A,14,5,7)
   mid = 6, A[6] = 15 > 14
binary_search(A,14,5,5)
   mid = 5, A[5] = 14 = 14 -> return 5
```

Here's a transcript of the calls made to binary search when searching for $x = 63$, which is not in the array:

```
binary_search(A,63,1,16)
   mid = 8, A[8] = 31 < 63
binary_search(A,63,9,16)
   mid = 12, A[12] = 51 < 63
binary_search(A,63,13,16)
   mid = 14, A[14] = 90 > 63
binary_search(A,63,13,13)
   mid = 13, A[13] = 78 > 63
binary_search(A,63,13,12)
   j < i -> return 0
```

As we argued earlier, we will repeat the `binary_search` call at most $\lceil \log n \rceil$ times if A contains n elements. This means that for large values of n (a million entries, a billion entries), the time it will take to find an element x will be proportional to $\lceil \log n \rceil$ which is the number of binary digits in n. So searching a database with a million entries using binary search will take time proportional to $\log 1000000$ which is roughly 20. If the database has a billion entries, it will take time proportional to 30. Not that much more. If each call takes about 10ms, say, then we can search a billion entries in 300ms, or 0.3s. Still not good enough for Amazon or Google, but good enough for a desktop database.

For binary search to work on an array, the array needs to be sorted. A quick look at our database shows that this is not the case by default (databases tend to store records in the order that they are entered). How can we sort an array? Actually, this is a task we do perform by hand occasionally, when rearranging a pile of numbered pages we just dropped on the floor or when ordering a hand of cards for a game. How do we order a hand of cards? Think about it for

a moment; typically we glance over the cards looking for the highest one and moving it all the way to the left, we then look for the next highest one and so on. Actually, when sorting a hand of cards we often mix two strategies: selection and insertion. Selection looks for the next card in the rest of the hand, insertion picks a card and looks for the right place to insert it. Both strategies lead to sorting algorithms. We will only see *selection sort* here. The idea then is: find the highest card in the remainder of the hand and move it to the next position.

Example 4.0.2. Before beginning the example, let us explain how to order a deck of cards. Each card has a suit and a rank. There are four suits:

$$\{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\},$$

read Club, Spade, Heart, and Diamond, and thirteen ranks:

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, \text{Jack}, \text{Queen}, \text{King}, \text{Ace}\}.$$

We will abbreviate the card names, e.g. we write $10\spadesuit$ for the ten of spades and $A\diamondsuit$ for the ace of diamonds. The cards are ordered by suit: $\clubsuit > \spadesuit > \heartsuit > \diamondsuit$ and *then* by rank: Ace > King > Queen > Jack > 10 > 9 > 8 > 7 > 6 > 5 > 4 > 3 > 2 (the actual order depends a bit on the game you are playing). So $9\spadesuit > A\diamondsuit$ and $A\clubsuit > K\clubsuit$.

Suppose our hand looks as follows: $9\spadesuit, J\heartsuit, Q\diamondsuit, 3\spadesuit, 5\clubsuit, J\diamondsuit$. The highest card we hold is the five of clubs, so we move it to the left: $5\clubsuit, 9\spadesuit, J\heartsuit, Q\diamondsuit, 3\spadesuit, J\diamondsuit$. In the remaining hand the next highest card is the nine of spades, which already is at the right location. Next is the three of spades, so we move it: $5\clubsuit, 9\spadesuit, 3\spadesuit, J\heartsuit, Q\diamondsuit, J\diamondsuit$. At this point the remaining cards are in the right order and we are done.

Note that we sorted the cards in descending order. For the remaining examples we will sort in ascending order.

To implement the selection sort algorithm we need to be able to find the smallest value in the array. We can do this using the following code:

```
min_val = A[1]
for j = 2 to n
   if A[j] < min_val
      min_val = A[j]
```

Actually, we don't just need the minimum *value* but also at which position it occurred, so we can move it towards the front later. We solve this problem by adding another variable, `min_pos`, that tracks the position of the minimum.

```
min_val = A[1]
min_pos = 1
for j = 2 to n
   if A[j] < min_val
      min_val = A[j]
      min_pos = j
```

At the end of the loop, `min_val` contains the value of the smallest element in A and `min_pos` the position of the smallest value. (We should be precise: if there are duplicate elements, then `min_val` will contain the smallest value and `min_pos` the first position in the array that that value occurs at). All we need to do now is swap `A[1]` with `A[min_pos]` and repeat the whole procedure for the remainder of the array, that is, `A[2 .. n]` and so on:

```
for i = 1 to n-1
   min_val = A[i]
   min_pos = 1
   for j = i + 1 to n
      if A[j] < min_val
         min_val = A[j]
         min_pos = j
   x := A[i]           // code swapping
   A[i] := A[min_pos]  // values in A[i]
   A[min_pos] := x     // and A[min_pos]
```

The code contains two nested loops, that is, it, roughly, has the following structure:

```
for i = 1 to n-1
   for j = i + 1 to n
       body
```

The body of the innermost loop is performed most often (when optimizing code, the innermost loop is the most important one to look at). It is performed, $n-1$ times for $i = 1$ plus $n-2$ times for $i = 2$ and so on, down to 1 time for $i = n-1$. So, overall, it is performed $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$ times (we will derive this formula later). As n gets larger, the most influential term here is n^2. We say, selection sort runs in time $\Theta(n^2)$, which means that ignoring constant factors and smaller terms, the running time of selection sort shows the same order of growth as does n^2.

For a database with a million elements selection sort would take about 30 years to sort it at a rate of one swap in a millisecond. Even if the machine took 1μs (one microsecond) per swap, it would still take about 11 hours to sort the database using selection sort. In practice much faster sorts are needed; selection sort is very good for small arrays though (one reason we use it in real life for sorting hands of cards).

Exercise 4.0.3. 1. Construct a hand of 6 cards that will require the largest number of swaps. How many swaps does it require?

2. Construct a hand of 6 cards that will require the smallest number of swaps. How many swaps does it require?

3. The way we wrote the code for selection sort, we always swap, even if the minimum element is already in its right position. In practice, one would modify the code to only perform the swap when necessary:

```
for i = 1 to n-1
   min_val = A[i]
   min_pos = 1
   for j = i + 1 to n
      if A[j] < min_val
         min_val = A[j]
         min_pos = j
   if i <> min_pos
      x := A[i]           // code swapping
      A[i] := A[min_pos]  // values in A[i]
      A[min_pos] := x     // and A[min_pos]
```

Using this variant of selection sort, answer the following questions:

- What is the smallest number of swaps you could have in an array of n elements; illustrate what the initial array looks like assuming it contains numbers $1, 2, \ldots, n$.
- What is the largest number of swaps you could have in an array of n elements; illustrate what the initial array looks like assuming it contains numbers $1, 2, \ldots, n$.

4.1 Exercises

1. There are algorithms (mergesort, quicksort, heapsort) that can sort an array in time proportional to $n \log n$. Assuming that it takes $1ms$ to perform a swap and the algorithm makes exactly $n \log n$ swaps (logarithm base 2), how long does it take to sort a database with $1,000$ elements? How about $1,000,000$ elements? How about a database with $1,000,000,000$ elements? Redo these computations under the assumption that a swap only takes $1\mu s$.

2. You are given a sorted array that may contain duplicate elements, e.g. $1, 2, 2, 5, 6, 6, 6, 8$. Write pseudocode for an algorithm that performs at most n swaps (where n is the length of the array) and moves all duplicates to the end of the array (in any order). In the example above, your algorithm could return $1, 2, 5, 6, 8, 6, 2$ ($1, 2, 5, 6, 8$ is in order, followed by the duplicates in arbitrary order). This algorithm is useful, since one can now easily resize the array, cutting off the duplicates, yielding $1, 2, 5, 6, 8$. This is exactly what a database needs to do when you request a `SELECT DISTINCT` rather than a `SELECT ALL`.