Introduction to Algorithms (CLRS) - Personal Solutions

Morgan Bruce

June 20, 2022

Contents

Chapter 1

The Role of Algorithms in Computing

1.1 Algorithms

1. Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

Solution. Any sort of problem involving organizing probably has sorting. For example, ordering a set of people by any quantitative characteristic (height, weight, test scores, age, their one mile run time, etc.).

Convex hulls are used widely in graphical simulations, for example to detect collisions. □

2. Other than speed, what other measures of efficiency might one use in a real-world setting?

Solution. Computer memory. Programmer time (time to implement algorithm). Energy efficiency. □

3. Select a data structure that you have seen previously, and discuss its strengths and limitations.

Solution. Linked lists have the advantage of being extendible up to the memory limit of your computer. Disadvantage of needing to traverse entire list to find an element (or determine if one isn't there) □

4. How are the shortest-path and traveling-salesman problems given above similar? How are they different?

Solution. Both are problems on weighted graphs involving attempting to minimize some subset of the graph (a path or a cycle). However, the shortest-path problem makes no requirement to end at the starting point, whereas the traveling-salesman problem does. This makes the shortest-path problem in P (solved by several algorithms, such as Dijkstra's), and the traveling-salesman in NP-complete. □

5. Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

Solution. A problem where only the best solution will do is one that requires a fully correct solution, like needing a sorted list for a directory. One where an approximate solution, as discussed earlier in the chapter, would be solving traveling-salesman. □

1.2 Algorithms as a technology

1. Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Solution. Google Maps when giving directions to users uses algorithms to determine the quickest route, factoring in distance, tolls, predicted traffic at the time the user would be at a certain road, any reported accidents, etc. □

2. Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runes in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

Solution. This is asking for what values of n we have that $8n^2 \leq 64n \lg n$. We can simplify this to $2^{n/8} \leq n$. Plugging in values yields the largest valid integer value of n being 43. □

3. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Solution. This is asking what the smallest value of n such that $100n^2 \leq 2^n$ is, which is equivalent to when $n \leq \frac{2^{n/2}}{10}$. The value of n is 15. $\square$

1.3 Problems

1. For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

Solution.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{1e6}	2^{6e7}	$2^{7.2e9}$	$2^{8.64e10}$	$2^{2.59e12}$	$2^{3.15e13}$	$2^{3.15e15}$
$\sqrt{n}$	$1e12$	$3.6e15$	$1.29e19$	$7.46e21$	$6.71e24$	$9.67e26$	$9.67e30$
n	$1e6$	$6e7$	$3.6e9$	$8.64e10$	$2.59e12$	$3.1104e13$	$3.11e15$
$n \lg n$	62746	$2.80e6$	$1.33e8$	$2.75e9$	$7.18e10$	$7.8709e11$	$6.76e13$
n^2	1000	7745	60000	293938	1609968	5577096	55770960
n^3	100	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	17	18

$\square$

Chapter 2

Getting Started

2.1 Insertion sort

1. Using Figure 2.2 as a model, illustrate the operation of `INSERTION-SORT` on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution. First pass: key set to 41. `A[1] = 31 < 41`, so the while loop breaks, and the second element stays put.

Second pass: key set to 59. Same steps as above.

Third pass: Key set to 26. `A[3] = 59 > 26`, so set `A[4] = 59`. `A[2] = 41 > 26`, so set `A[3] = 41`. `A[1] = 31 > 26`, so set `A[2] = 31`. Loop breaks, set `A[1] = 26`.

Fourth pass: Key set to 41. `A[4] = 59 > 41`, so set `A[5] = 59`. `A[3] = 41`, so the while loop breaks. Set `[4] = 41`.

Final pass: Key set to 58. `A[5] = 59 > 58`, so set `A[6] = 59`. `A[4] = 41 < 58`, so the while loop breaks. Set `[5] = 58`. The final sorted array is $\langle 26, 31, 41, 41, 58, 59 \rangle$. $\square$

2. Rewrite the `INSERTION-SORT` procedure to sort into nonincreasing instead of nondecreasing order.

Solution. Rewrite the algorithm as follows.

```
REVERSE-INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] < key // The only change
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

The implementation of this can be found in
`code/<language>/chapter02/reverse_insertion_sort/`. □

3. Consider the **searching problem**:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value `NIL` if v does not appear in A.

Write pseudocode for **linear search**, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Solution. The algorithm is as follows.

```
LINEAR-SEARCH(A,v)
for i = 1 to A.length
    if A[i] == v
        return i
return NIL
```

The loop invariant can be stated as follows: At the start of each iteration of the for loop, the subarray $A[1..i-1]$ does not contain the value v. We will show this invariant has the three necessary properties to prove the correctness of the algorithm. Prior to the first loop, the subarray $A[1..0]$ is an empty array, which clearly does not contain v. Suppose the invariant is true after loop i, that is, $A[1..i]$ does not contain v. Then on loop $i+1$, if $A[i+1] == v$, the algorithm will exit, returning $i+1$. If the algorithm does not exit, the loop iteration completes and thus $A[1..i+1]$ does not contain v. When the loop terminates without the algorithm prematurely returning, that means that v is not contained in $A[1..n]$, and thus `NIL` is returned. So the loop invariant fulfills the necessary properties to prove correctness of the algorithm. □

4. Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

Solution. Define the problems as follows:

Input: Two sequences A, B representing binary numbers, where the sequences are length n and contain only 0 or 1. The endianness of A, B should be defined, as it affects the algorithm to solve it. We will assume big-endian.

Output: A length $n+1$ sequence representing the binary sum $A+B$.

Write the algorithm solving the problem as follows:

```
ADD(A,B)
carry = 0
for i = A.length to 1
    sum = A[i] + B[i] + carry // Binary addition
    if sum % 10 == 1 // Sum either 01 or 11
        S[i+1] = 1
    if sum >= 10 // Sum either 10 or 11
        carry = 1
    else
        carry = 0
S[0] = carry
return S
```

□

2.2 Analyzing algorithms

1. Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Solution. Considering the informal definition of Θ-notation in this chapter, we simply observe that the $n^3/1000$ term is the leading term and dominates for large n, and drop the coefficient, expressing the function as $\Theta(n^3)$. □

2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

Solution.

```
SELECTION-SORT(A)
for j = 1 to A.length - 1
    current_min = stored_val = A[j]
    stored_index = j
    for i = j + 1 to A.length
        if A[i] < current_min
            current_min = A[i]
            stored_index = i
    A[j] = current_min
    A[stored_index] = stored_val
```

See `code/<language>/chapter02/selection_sort/` for implementation of this algorithm.

The loop invariant that the outer loop maintains is that the subarray $A[1..j]$ is sorted. Informally, this holds because the minimum element is selected to the first index, the second smallest element is selected to the second index, and so forth through the first j elements. The loop only needs to run for the first $n-1$ elements because after $n-1$ iterations, the largest element is necessarily in the final index, as the $n-i$th largest elements are already all fixed to the ith indices. The best case runtime is $\Theta(n^2)$, since even with an input list that is already sorted, there is no way to early stop either for loop, as all elements need to be checked to confirm each candidate element is the current minimum. The worst case is equivalently $\Theta(n^2)$, as the same checks as in the best case are still performed, but the current minimum and stored index are updated on each increment of the i index (in Θ notation, this only adds a linear term which is already accounted for by the nested loop). □

3. Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ notation? Justify your answers.

Solution. □

4. How can we modify almost any algorithm to have a good best-case running time?

Solution. □