My algorithms exercises

Evgeny Markin

2023

Contents

Preface

Exercises for Introduction to Algorithms by Cormen et al., 4th ed. It has exercises, that should be written down, moslty in math and whatnot.

Some of the exercises require that you code something (sometimes it's not explicitely required, but that would be nice to code it anyways), and this code is presented in the progs folder. Everything is written in C, because I'm most familiar with it.

Pseudocode is written by using package **algorithm2e**, which does not really correspond to the one, that is used in the book, but it still does the job.

Same rules as usual apply, if you want to use this book for any reason – go right ahead, it's free, just be aware that it is full of mistakes.

Part I

Foundations

Chapter 1

The Role of Algorithms in Computing

1.1 Algorithms

1.1.1

Describe your own real-world example that required sorting. Describe one that required finding the shortest distance between two points.

I've needed both when I was creating 8-puzzle program

1.1.2

Other than speed, what other measures of efficiency migh you need to consideer in a real-world setting?

Memory and parallellability.

1.1.3

Select a data structure that you've seen, and discuss its strengths and limitations.

Linked lists. They are perfect in everything, apart from sorting; but even then you can define any data structure through linked lists, which makes them just perfect (especially omnidirectional ones).

1.1.4

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

Sorting has to be perfect, otherwise it's borderline useless. Estimated time to complete the task can tolerate imperfections.

1.1.5

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time

Traffic on maps does this thing. Sometimes you have all the input, sometimes it changes.

1.2   Algorithms as a technology

1.2.1

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

Path finding on maps will do. It requires to triverse graphs and whatnot.

1.2.2

Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

For

$$8n^2 < 64n \lg n$$

$$n < 8 \lg n$$

$$\frac{n}{lgn} < 8$$

$$n \approx 44$$

cases.

1.2.3

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?

Calculator says 15

Chapter 2

Getting Started

2.1 Inserition sort

2.1.1

Using Figure 2.2 as a model, illustrate the operation of Insertion-Sort on an array initially containing the sequence $[31, 41, 59, 26, 41, 58]$

$$[31, 41, 59, 26, 41, 58]$$

$$[26, 31, 41, 59, 41, 58]$$

$$[26, 31, 41, 41, 59, 58]$$

$$[26, 31, 41, 41, 58, 59]$$

2.1.2

State loop invariant for the Sum-Array procedure.

At the start of each iteration of the for loop, the *sum* variable contains the sum of elements in $A[1:i]$.

Initialization:

Firstly, we've got 0 as the sum. Given that we've summed 0 elements so far, we can conclude that this is indeed a correct value to set it.

Maintenance:

For each iteration of i we've got that we add a i'th element from the array to our sum and incrementing i. Thus before iterating through i we had a sum of all of the elements before i, and after iterating through it we create a sum of elements before i and the i'th element as well. Thus the sum after iterating through i is correct.

Termination:

Given that the array is finite, we follow that because we are incrementing i at each iteration the algoritm will terminate. Because we increment through elements, we follow that we've added every element of the array to the sum at the point of termination.

2.1.3

Rewrite the Insertion-Sort procedure to sort into monotonically decreasing instead of monotonically increasing order.

Done it in the progs section; long story short: reverse the ordering function in the inner loop, replace $A[j] > key$ with $A[j] < key$.

2.1.4

Consider the seqrching problem

Input: A sequence of n numbers $[a_1, ...a_n]$ stored in array $A[1:n]$ and a value x.

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

Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorythm is correct. Make sure that your loop invariant fufills the three necessary properties.

Function Linear-search(A, x)

```
1 for (i = 1 → n) do
2 │   if A[i] = x then
3 │   │   return i ;
4 │   end
5 end
6 return NIL;
```

At each iteration of the for loop, we've got that elements $A[1:i]$ do not contain x.

Initialization: Null case is when we haven't gone through any of the elements yet; because we haven't searched anything, we can follow that we haven't found anything, thus the base case is correct

Maintenance: Suppose that $i = j + 1$. Then we follow that $A[1:j]$ does not contain our element by our induction hypothesis (not sure that we can use this kind of language here, but it's my book and I can do whatever I want). Then i'th element is cheched for the necessary equality and returned in case of the equality; otherwise we increment i and make it so $A[1:j+1]$ is the array of processed values. Thus before and after the loop iteration we have a correct solution.

Termination: We terminate either after going through every element, or some time before it.

Example of this thing in C is presented in progs directory

2.1.5

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{n=0}^{n-1} A[i] * 2^i$ and $b = \sum_{n=0}^{n-1} B[i] * 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{n=0}^{n-1} C[i] * 2^i$. Write a procedure Add-Binary-Integers that takes an input arrays A and B, along with the length n, and returns array C holding the sum.

Function Add-Binary-Integers(A, B, n)

1 define C[0: n + 1] and fill it with zeroes;
   /* carry stores overflow from the previous iteration   */
2 $carry \leftarrow 0$;
3 for $(i = 0 \rightarrow n + 1)$ **do**
     /* Initializing temporary variable with carry bit; we need to sum
       carry, A[i], and B[i], so we can just initialize temp with
       carry   */
4 $r \leftarrow carry$;
     /* if it is not the last bit, where A nor B are not defined; We
       can just define it to be zero with the same result   */
5   **if** $i \neq n + 1$ **then**
6 $r \leftarrow A[i] + B[i] + r$;
7   **end**
     /* If we've got an overflow as the result, set carry bit and
       result of summation appropriately   */
8   **if** $r > 1$ **then**
9 $carry \leftarrow 1$;
10 $r \leftarrow r\%2$;
11 **else**
     /* otherwise zero the carry bit   */
12 $carry \leftarrow 0$;
13 **end**
     /* lastly, put the result of the partial summation into the
       resulting array   */
14 $C[i] \leftarrow r$;
15 end
16 return C;

2.2 Analyzing algorithms

2.2.1

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

$$n^3/1000 + 100n^2 - 100n + 3 \in \Theta(n^3)$$

2.2.2

Consider sorting n numbers strored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Continue in this manner for the first $n-1$ elements of A. Write a pseudocode for this algorithm, which is known as selection sort. What loop invarian 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 worst-case running time fo selection sort in Θ-notation. Is the best-case running time any better?

Function Selection-Sort(A, n)

---

1   **for** $i = 1 \rightarrow n-1$ **do**
2     $minPos \leftarrow i$;
3   **for** $j = i \rightarrow n$ **do**
4   **if** $A[j] < A[i]$ **then**
5         $minPos \leftarrow j$
6   **end**
7   **end**
8     $temp \leftarrow A[i]$;
9     $A[i] \leftarrow A[minPos]$;
10     $A[j] \leftarrow temp$;
11 **end**

(this function in C is located in progs)

Algorithm maintains that every $A[1:i]$ is sorted and any element in $A[i+1:n]$ is greater or equal to every element of $A[1:i]$. It needs to run only for $n-1$ times, because the last element n will be greater or equal then any element in $A[1:n-1]$ and therefore will be places in its rightful place.

Worst-case running time of this algorithm is $\Theta(n^2)$, because we compute that coefficient for c_4 is $\sum_{i=1}^{n} i = \Theta(n^2)$, while all the other coeffitients are linear.

Given coefficient for c_4 is not changed depending on the input, we follow that the best running time and worst running time are the same

2.2.3

Consider linear search again. How many elements of the input array need to be checked on the average, assuming that the element being searched for is equaly likely to be any element in the array? How about the worst case? Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

The low-hanging answer here is that we need to check the first half of the array in order to find the needed input. It is wrong though.

Assuming that $A[i] = x$ with probability p, we follow that we need to check first $1/p$ elements in order to get the desired element (I can be wrong here, I'm not good with probabilities).

Assuming that I'm right about it, we follow that $pn \in \Theta(n)$ for the average case.

The worst case is if A does not contain x, in which case we've got the running time of n. It is also in $\Theta(n)$.

2.2.4

How can you modify any sorting algorithm to have a good best-case running time?

We can throw a check for the case if A is sorted from the start in the start of the sorting algorithm. Thus we can follow, that the best case for any such algorithm is when we get the sorted input, and we get linear best-case running time.

Part II

Appendix: Mathematical Background

Chapter 3

Summations

1-1

Prove that $\sum_{k=1}^{n} O(f_k(i)) = O(\sum_{k=1}^{n} f_k(i))$

Short answer:

$$\sum cg(x) = c \sum g(x)$$

Long answer:

Suppose that $g \in O(f_k(i))$. It follows that there exists n_i and c_i such that $0 \leq g(n) \leq c f_i(n)$. Thus we can pick $n = \max\{n_0, n_1, ...\}$ and $c = \max\{c_0, c_1, ...\}$. We know that both n and c will work all of functions f_k. Therefore by linearity of summations

$$\sum_{k=1}^{n} O(f_k(i)) = \sum_{k=1}^{n} c f_k(i) == c \sum_{k=1}^{n} f_k(i) == O(\sum_{k=1}^{n} f_k(i))$$

(notation is a little abused and there is nothing is rigorously proven, but it'll do).

1-2

Find a simple formula for $\sum_{k=1}^{n} (2k - 1)$.

$$\sum_{k=1}^{n} (2k - 1) = \sum_{k=1}^{n} (2k) - \sum_{k=1}^{n} (1) = 2 \sum_{k=1}^{n} (k) - n = 2\frac{n(n+1)}{2} - n = n(n+1) - n = n^2$$

1-3

Interpret the decimal number $111, 111, 111$ *in light of equation A.6*

$$111,111,111 = \sum_{k=0}^{9} 10^k = \frac{10^{10} - 1}{10 - 1}$$

1-4

Evaluate the infinite series $1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - ...$

The series converges absolutely to 2, so we are free to do anything with it.

$$1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - ... = \sum_{k=0}^{\infty} \frac{1}{2}^{2k} - \sum_{k=0}^{\infty} \frac{1}{2}^{1+2k} = \sum_{k=0}^{\infty} \frac{1}{2}^{2k} - \frac{1}{2} \sum_{k=0}^{\infty} \frac{1}{2}^{2k} = \left(1 - \frac{1}{2}\right) \sum_{k=0}^{\infty} \frac{1}{2}^{2k} =$$

$$= \left(1 - \frac{1}{2}\right) \sum_{k=0}^{\infty} \frac{1}{4}^{k} = \left(1 - \frac{1}{2}\right) \frac{1}{1 - \frac{1}{4}} = \frac{1}{2} * \frac{4}{3} = \frac{2}{3}$$

1-5

Let $c \geq 0$ be a constant. Show that $\sum_{k=1}^{n} k^c = \Theta(n^{c+1})$

$$\sum_{k=1}^{n} k^c = \sum_{k=1}^{n-1} k^c + n^c = n^c \sum_{k=1}^{n} \frac{k^c}{n^c} =$$

Let $f(n) = n^c$. It can be seen from the graph that

$$\int_0^n f(x)dx \leq \sum_{k=1}^{n} k^c \leq \int_0^n f(x+1)dx$$

Thus

$$\int_0^n f(x)dx = \int_0^n x^c = \frac{n^{c+1}}{c+1} \in$$

$$\int_0^n f(x+1)dx = \int_0^n (x+1)^c = \frac{(n+1)^{c+1}}{c+1}$$

Thus we can state that $\sum_{k=1}^{n} k^c = \Theta(n^{c+1})$ (I'm not good enough yet to show that $\frac{(n+1)^{c+1}}{c+1} \in \Theta(n^{c+1})$, but I'm pretty sure that it's true TODO).

1-6

Show that $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ for $|x| < 1$
    We know that for $|x| < 1$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

thus if we differentiate both sides we get

$$\sum_{k=0}^{\infty} k^2 x^{k-1} = \frac{2x}{(1-x)^3} + \frac{1}{(1-x)^2}$$

and then if we multiply all of it by x we'll get

$$\sum_{k=0}^{\infty} k^2 x^k = \frac{2x^2}{(1-x)^3} + \frac{x}{(1-x)^2}$$

thus if we factor all of this jazz we'll get

$$\sum_{k=0}^{\infty} k^2 x^k = -\frac{x(x+1)}{(x-1)^3}$$

and if we tuck this minus into denominator we'll get (which we can do because the power is odd)

$$\sum_{k=0}^{\infty} k^2 x^k = \frac{x(x+1)}{(1-x)^3}$$

as desired.

1-7

Prove that $\sum_{k=1}^{n} \sqrt{k \lg k} = \Theta(n^{3/2} \lg^{1/2} n)$

$$\int \sqrt{k \lg k} =$$

1-9

Show that

$$\sum_{k=0}^{\infty} (k-1)/2^k = 0$$

$$\sum_{k=0}^{\infty}(k-1)/2^k = \sum_{k=0}^{\infty}k/2^k - \sum_{k=0}^{\infty}1/2^k = \sum_{k=0}^{\infty}k/2^k - 2 = 0$$

$$\sum_{k=0}^{\infty}k/2^k - 2 = 0$$

$$\sum_{k=0}^{\infty}k/2^k = 2$$

$$\sum_{k=0}^{\infty}k/2^k - 2 = 0$$

$$\sum_{k=0}^{\infty}\frac{k-2^{k+1}}{2^k} = 0$$

Chapter 4

Sets, Etc.

1-1

Draw Venn diagrams that illustrate the first of the distributive laws (B.1)
    TODO, add picture here

1-2

Prove the generalization of DeMorgan's laws to any finite collection of sets
    Copy from real analysis exercises
    Suppose that $x \in (\cup_{\lambda \in \Lambda} E_\lambda)^c$. It follows, that x is not in the union of given sets. Therefore there is no set E_n such that $x \in E_n$ (because if there would be such a set, then x wouldn't be in $(\cup_{\lambda \in \Lambda} E_\lambda)^c$). Therefore $x \in \cap_{\lambda \in \Lambda} E_\lambda^c$. Therefore

$$(\cup_{\lambda \in \Lambda} E_\lambda)^c \subseteq \cap_{\lambda \in \Lambda} E_\lambda^c$$

The proof of reverse inclusion is the same as with the forward, but in reverse order.
    $x \in (\cap_{\lambda \in \Lambda} E_\lambda)^c$ implies that x is not in every E_n. Therefore there exists $x \in E_n^c$ for some E_n. therefore it is in $\cup_{\lambda \in \Lambda} E_\lambda^c$. The proof of reverse inclusion uses the same argument, but in other direction.

1-3

TODO

1-4

Show that the set of odd natural numbers is countable.

Let us set a function $f : A \to N$, where A denotes the set of odd natural numbers

$$f(n) = (n+1)/2$$

for this function we've got

$$f^{-1}(n) = 2n - 1$$

Both funcitons are injective and therefore f is bijective. Therefore we've got a bijective function betweeen A and N, therefore $A \sim N$, therefore it is conuntable, as desired.

1-5

Show that for any finite set S, the power set 2^S has $2^{|S|}$ elements (that is, there are $2^{|S|}$ distinct subsets of S).

Another copy from real analysis

This proof is dumb, but intuitive:

Every subset is corresponding to a number in binary system: 0 for excluded, 1 for included. Therefore there exist 2^n possible combinations.

For a more concrete proof let's resort to induction.

Base case(s): subsets of $\emptyset$ are $\emptyset$ itseft ($2^0 = 1$ in total). Subsets of set with one element are $\emptyset$ and set itself ($2^1 = 1$ in total).

Proposition is that set with n elements has 2^n subsets.

Inductive step is that for set with $n+1$ elements can either have or hot have the $n+1$'th element. Therefore there exist $2^n + 2^n = 2 * 2^n = 2^{n+1}$ subsets, as desired.

1-6

Give an inductive definition for an n-tuple by extending the set-theoretic definition for an ordered pair.

The tuple is actually just a re-writing of particular set

$$(a_1, a_2, ..., a_n) = \{\{a_1\}, \{a_1, a_2\}, \{a_1, a_2, a_3\}...\{a_1, a_2, a_3, ..., a_n\}\}$$