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.

Usefull stuff

0.0.1 Triangular numbers

Triangular numbers refer to the sum

$$1 + 2 + 3 + 4...$$

They are called so because when we stack 1, 2, 3, etc, things on top of the other, we get a triangle. For a useful visual reference, google (or just imagine) bowling pins.

The sum evaluates to

$$S(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

(for a proof I'll sudgest using induction, or you can look a pretty interesting proof in the book "Concrete mathematics" by Knuth et. al.) for which it is true that

$$\Theta(S(n)) = \Theta(n^2)$$

.

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 ← 0;
3  for (i = 0 → 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 ← 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 ≠ n + 1 then
6          r ← 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 ← 1 ;
10         r ← r%2;
11     else
           /* otherwise zero the carry bit                             */
12         carry ← 0;
13     end
       /* lastly, put the result of the partial summation into the
          resulting array                                              */
14     C[i] ← 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 → n − 1 do
2  |  minPos ← i;
3  |  for j = i → n do
4  |  |  if A[j] < A[i] then
5  |  |  |  minPos ← j
6  |  |  end
7  |  end
8  |  temp ← A[i];
9  |  A[i] ← A[minPos];
10 |  A[j] ← 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.

2.3 Designing algorithms

2.3.1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence

$$[3, 41, 52, 26, 38, 57, 9, 49]$$

$$[3, 41, 52, 26][38, 57, 9, 49]$$

$$[3, 41][52, 26][38, 57][9, 49]$$

$$[3][41][52][26][38][57][9][49]$$

$$[3, 41][26, 52][38, 57][9, 49]$$

$$[3, 26, 41, 52][9, 38, 49, 57]$$

$$[3, 9, 26, 38, 41, 49, 52, 57]$$

2.3.2

The test in line 1 of the Merge-Sort procedure reads "if $p \geq r$" rather than "if $p \neq r$". If Merge-Sort is called with $p > r$, then the subarray $A[p : r]$ is empty. Argue that as long as the initial call of Merge-Sort(A, 1, n) has $n \geq 1$, the test "if $p \neq r$" suffices to ensure that no recursive call has $p > r$.

For a general case when $p \leq r$ we've got that

$$p = 2p/p = (p + p)/2 \leq \lfloor (p + r)/2 \rfloor = q$$

thus $p \leq q$ and call on line 4 will be done with the condition that $p \leq q$.

Focusing our attension on r, we get that by condition on lines 1 and 2 we follow that the function returns if $p = r$. Thus we follow that $p \neq r$, and by our assumption we get that by the line 3 we get if $p < r$. Given that both p and r are integers, we follow that

$$(p + r)/2 \leq (p + p + 1)/2 = p + 1/2$$

thus

$$\lfloor (p + r)/2 \rfloor \leq \lfloor p + 1/2 \rfloor = p$$

Therefore we follow that

$$q \leq r - 1$$

$$q + 1 \leq r$$

thus we can follow that the call to the Merge-Sort on line 5 happens also with the condition that $p \leq r$.

Thus we can state that given that $p \leq r$ we can follow that all the other calls to Merge-Sort, that happen inside of it will have the same restriction. Therefore we follow that if initical call to Merge-Sort happens with $n \geq 1$, then the test "if $p \geq r$" is equivalent to the test "if $p \neq r$".

2.3.3

State a loop invariant for the while loop of lines 12-18 of the Merge procedure. Show how to use it, along with the while loops 20-23 and 24-27, to prove that the Merge procedure is correct.

At the start of each iteration of the while loop of lines 12-18, the subarray $A[1 : k]$ consists of th elements, originally in $L[1 : i]$ or $R[1 : j]$ in sorted order and any element in $A[1 : k]$ is less or equal to any element of $L[i + 1, n_L]$ and $R[i + 1, n_R]$.

Base case is trivial, sinse none of the subarrays contain any elements.

For maintenance we've got that we take the lowest element of $L[i : n_L]$ or $R[j : n_R]$ (which happens to be the lowest of the first elements of those subarrays, given that both of those arrays are sorted), and placing them at $A[k]$. Because of our restrictions on $L[i+1, n_L]$

and $R[+1, n_R]$ we follow that $A[k]$ will be less or equal then any element of $L[i+1, n_L]$ and $R[j+1, n_R]$. And given that it is originally from $L[i+1, n_L]$ or $R[i+1, n_R]$, we follow that $A[k]$ is greater or equal then any of $A[1 : k-1]$. Thus we can follow that $A[1 : k]$ is sorted. Thus at the end of the loop we've got our loop invariant.

Termination happens whenever we run out of elements in L or R, at which point we still have our invariant.

We follow then that after our loop, one of the arrays L or R will be empty, and will contain elements, that are greater or equal then $A[1 : k]$. Thus we just append remaining elements of either of the arrays to A.

Given that L and R are two subarrays of $A[k]$, whose disjoint union is A, we follow that merge indeed merges two subarrays of $A[k]$ into one sorted array, as desired.

2.3.4

Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 \text{ if } n = 2 \\ 2T(n/2) + n \text{ if } n > 2 \end{cases}$$

is $T(n) = n \lg(n)$.

Our domain here is $(n_k) = 2^k$

For base case $k = 1 \to n = 2$ we've got

$$T(2) = 2 = 2 * 1 = 2 \lg(2)$$

Our hypothesis is that for $k - 1 < n$ and n_{k-1} we've got that $T(n_{k-1}) = n_{k-1} \lg(n_{k-1})$

Thus we follow that if n is an exact power of 2 and $n \geq 2 > 0$ we've got that $n/2 < n$. Thus we follow that

$$T(n) = 2T(n/2) + n = 2n/2 \lg(n/2) = n \lg(n/2) + n = n(\lg(n/2) + 1) =$$

$$= n(\lg(n * 2^{-1}) + 1) = n(\lg(n) - 1 + 1) = n \lg(n)$$

Therefore for $x \in (n_k)$ we've got that $T(x) = x \lg(x)$, as desired.

2.3.5

You can also think of insertion sort as a resursive algorithm. In order to sort $A[1 : n]$, recursively sort the subarray $A[1 : n-1]$ and then insert $A[n]$ into the sorted subarray $A[1 : n-1]$. Write a pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

For the worst-case we've got

$$T(n) = T(n-1) + n$$

```
Function Recursive-insertion-Sort(A, n)
   /* Ensure termination and sane inputs                              */
 1 if n ≤ 1 then
 2 |  return;
 3 end
   /* Sort everything in A[1:n - 1]                                   */
 4 Recursive-insertion-Sort (A, n - 1);
 5 i ← n − 1 ;
 6 while A[i] > A[n] and i ≥ 1 do
 7 |  i ← i − 1
 8 end
   /* Swapping A[n] and A[i]                                          */
 9 temp ← A[i] ;
10 A[i] ← A[n] ;
11 A[n] ← temp ;
```

which reduces to

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

2.3.6

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg(n))$

For this one we conclude that its worst-case is $\Theta(\lg n)$ by the same logic as in merge, but we go through only one branch.

2.3.7

The while loop of lines 5-7 of the Insertion-Sort procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 : j-1]$. *What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to* $\Theta(n \lg(n))$*?*

If we use binary search instead of the linear search, then we'll undoubtably have a better running time on a real machine, given that the the binary search uses less instructions then the linear search.

```
Function Binary-Search(A, x, s, f)
1  if s = f then
2  |  return NIL;
3  end
4  q ← ⌊(s + f)/2⌋;
5  if A[q] = x then
6  |  return q;
7  else
8  |  if A[q] < x then
9  |  |  return Binary-Search (A, x, q + 1, f) ;
10 |  else
11 |  |  return Binary-Search (A, x, s, q - 1) ;
12 |  end
13 end
```

But in order to put the required value in the needed place, we'll still have to go through the entire array in a worst-case scenario (array sorted in reverse order). Thus it can be argued that the overall asympomatic approximation will be the same.

2.3.8

Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum exactly to x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

The obvious case with just going through the whole array will not do, as it will give us worst-case scenario of $\Theta(n^2)$.

So the main strategy here will be to sort the whole array and then make a linear search for the desired values.

Thus we'll get something that looks like Sum-of-ints

The last while iterates once through the array, therefore we can (non-rigorously) follow that everything except for the search runs at $\Theta(n)$. Thus the whole thing goes through the array at time

$$\Theta(n) + \Theta(n \lg(n)) = \Theta(n \lg(n))$$

(or at the very least I think so; once again we've got non-rigorous approach here, but I'm sure that in subsequent chapters we'll have some more sophisticated tools at our hands to prove me right).

Function Sum-of-ints(A, n, x)

```
1  Merge-Sort(A) ;
2  i ← 1;
3  j ← n;
4  while i ≠ j do
5  |  if A[i] + A[j] > x then
6  |  |  j ← j − 1 ;
7  |  else
8  |  |  if A[i] + A[j] > x then
9  |  |  |  i ← i + 1 ;
10 |  |  else
11 |  |  |  return [A[i], A[j]];
12 |  |  end
13 |  end
14 end
```

2.4 Problems

2.4.1 Insertion sort on small arrays in merge sort

(a) Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time

We know that the insertion sort sorts an array with worst-case performance of $\Theta(n^2)$, and therefore its worst case performance for the array of length k is $\Theta(k^2)$. Thus we can follow that it sorts n/k sublists of the original array at time

$$n/k * \Theta(k^2) = n/k * ck^2 = cnk = \Theta(nk)$$

(b) Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time

We can probably get it by applying standart merge in pairs, and then recursively. More explicitely we first merge first and the second, third and fourth, and so on, getting in the end roughly $n/2k$ sorted subarrays. Then we can relabel resulting arrays, so that the result of the merge on first and second subarray becomes first array, result of merging third and fourth becomes second array and so forth.

By doing that we iterate through the whole array at every inetration, and we go through roughly $\lg(n/k)$ iterations, because every time we reduce the number of subarrays in half.

(c) Given taht the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standart merge sort, in terms of Θ-notation?

With our rough definition of Θ-notation, we get that the approximate times of the

algorithms are roughly proportional to the times inside Θ. Thus there must exist c_1, c_2 such that

$$c_1(nk + n\lg(n/k)) = c_2(n\lg(n))$$
$$c_1 n(k + \lg(n/k)) = c_2 n\lg(n)$$
$$c_1(k + \lg(n/k)) = c_2\lg(n)$$
$$c_1(k + \lg(n) - \lg(k)) = c_2\lg(n)$$
$$k - \lg(k) = (c_1 - c_2)\lg(n)$$
$$\lg(2^k) - \lg(k) = (c_1 - c_2)\lg(n)$$
$$\lg(2^k/k) = (c_1 - c_2)\lg(n)$$
$$2^k/k = 2^{(c_1 - c_2)}n$$

I'm sure that I've made a mistake here somewhere, but it's the best that I can do.

(d) How should you choose k in practice?

By trial and error, I suppose. I'd stick to one sort in general and not overcomplicate the things, but if we try to overcomplicate things, then we can go through the cases when the linear sort outperform the merge, get somewhat concrete number and divide the first sort based on this number.

2.4.2 Correctness of bubblesort

(a) Let A' denote the array A after Bubblesort(A, n) is executed. To prove that Bubblesort is correct, you need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq ... \leq A'[n]$$

In order to show that Bubblesort actually sorts, what else do you need to prove?

That A' consists of elements, that were originally in A.

(b) State precicely a loop invariant for the for loop in lines 2-4, and prove that this loop invariant holds.

$A[j]$ is the smallest element of $A[j : n]$. Elements of $A[i + 1 : n]$ remain to be the same, but their order might change.

Initialization: $A[j : n]$ consists of one element, therefore $A[j]$ is the smallest element of it. No manipulations to $A[i + 1 : n]$ were applied, therefore the elements of it are the same.

Maintenence: If $A[j - 1]$ is greater then $A[j]$, then we swap their places. Thus we follow that before the next iteration $A[j]$ will be the smallest element of $A[j : n]$. The only performed operation is the possible swap, therefore we can follow that elements remain to be the same.

Termination: Loop terminates, because at each iteration we reduce j by 1. Nothing happens to the loop invariant, therefore it still holds.

(c) Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1-4 that allows you to prove inequality (2.5). Your proof should use the structure of the loop-invariant proof presented in this chapter.

$A[1 : i-1]$ are the smallest values of $A[1 : n]$, but in sorted order. Values of $A[1 : n]$ remain the same, but their place may change.

Initialization: $A[1 : i-1]$ is empty, therefore first part holds. No manipulations were performed on the array, therefore the second condition also holds.

Maintenence: As we've discussed earlier, inner for makes it so that we push the smallest element of $A[i : n]$ to $A[i]$. Given that $A[1 : i-1]$ has the smallest values, we follow that the smallest value of $A[i : n]$ is greater or equal to any value in $A[1 : i-1]$. THus we follow that $A[1 : i-1]$ will be sorted before the next iteration. Thus the first part holds.

Conditions, that are imposed in loop invariant of inner for guarantees us the second condition of the loop invariant.

Termination: We increment i at every iteration and n remains unchanged, therefore we follow that loop terminates. All of the requirements of loop invariant are implied by maintenence clause.

(d) What is the worst-case running time of Bubblesort? How does it compare with the running time of Insertion-Sort?

Worst-case is once again the reversely sorted array. In that case we need to execuse line 4

$$\Theta(\sum_{j=1}^{n} j) = \Theta(n^2)$$

times. In the best case we've got the same case, but now with the line 3.

This sort has the same asymptotic time as the Insertion sort, although the practice shows, that it it absolutely horrible when it comes to the machine test.

2.4.3 Correctness of Horner's rule

(a) In terms of Θ-notation, what is the running time of this procedure?

Linear with respect to n ($\Theta(n)$).

(b) Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare with Horner?

Worst case running time of this algorithm is a triangular number (beccause of the inner for), therefore it is $\Theta(n^2)$.

With a Horner it comperes badly.

(c) Consider the following look invariant for the procedure Horner:

Function Naive-poly-eval(A, n, x)

```
1 p = 0;
2 for i = 0 → n do
3 |   pw ← 1 ;
4 |   for j = 0 → i do
5 |   |   pw = pw * x;
6 |   end
7 |   p = A[i] * pw + p;
8 end
9 return p;
```

At the start of each iteration of the foor loop of lines 2-3,

$$p = \sum_{k=0}^{n-(i+1)} A[k+i+i] * x * k$$

Interpret a summation with no terms as equating 0. Use this loop invariant to show taht, at terminatiom, $p = \sum A[k] * x * k$.

Initialization: Since $p = 0$, we can follow that the loop invariant holds for the case before the initialization.

Maintenence: Following the Horner rule definition, we can see taht after completing another loop cycle, we get to the next parenthesis in the Horner rule's parenthesization. Thus we've got the desired result at the output.

Termination: Algorihm terminates, since we don't modify i in the loop itself. Thus, according to our loop invariant, this procedure indeed returns the $\sum A[k] * x * k$, as desired.

(Not sure that this is right, but got no intentions to do a proper one.)

2.4.4

(a) List the five inversions of the array $\{2.3, 8, 6, 1\}$

$$(6,1), (8,1), (3,1), (2,1), (8,6)$$

(b) What array with elements from the set $\{1, 2, ...n\}$ *has the most inversions? How many does it have?*

I think that the reversel sorted one gets the cake. It has a triangular number of inversions (obvious, when counting from the end).

(c) What is the relationship between the running time of the insertion sort and the number of inversions in the input array? Justify your answer.

It's proportional to the number of inversions, since it essentially corrects them one by one in decreasing order of the second number of the pair.

(d) Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg(n))$ worst-case time.

TODO