

Week 3 Applied Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the applied classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to these problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Problems

Problem 1. Describe a simple modification that can be made to any comparison-based sorting algorithm to make it stable. How much space and time overhead does this modification incur?

Solution

Assume that the sequence is $\{a_1, a_2, \dots, a_n\}$ where a_i represents the element at i^{th} position in the sequence (e.g., element at index i in the array). To stabilise a comparison-based sorting algorithm, we can replace each element of the sequence a_i with a pair (a_i, i) . Since each index is unique, the pairs are unique. We can then apply any comparison-based sorting algorithm on these pairs where two pairs (a_i, i) and (a_j, j) are first compared based on the values a_i and a_j and, if these values are equal (i.e., $a_i = a_j$), they are then compared on their index positions i and j . Our comparison ensures that they will retain their relative order, i.e., a_i always appears before a_j if $a_i = a_j$ and $i < j$, therefore, they will be sorted stably.

This modification increases the auxiliary space complexity by $\Theta(n)$ since we store an additional integer for each of the n elements, but does not affect the time complexity since it only adds a constant overhead to each comparison and $\Theta(n)$ additional preprocessing time.

Problem 2. A subroutine used by Mergesort is the merge routine, which takes two sorted lists and produces from them a single sorted list consisting of the elements from both original lists. In this problem, we want to design and analyse some algorithms for merging many lists, specifically $k \geq 2$ lists.

- (a) Design an algorithm for merging k sorted lists of total size n that runs in $O(nk)$ time
- (b) Design a better algorithm for merging k sorted lists of total size n that runs in $O(n \log(k))$
- (c) Is it possible to write a comparison-based algorithm that merges k sorted lists that is faster than $O(n \log(k))$?

Solution

To solve part (a), we can just do the naive algorithm. For all n elements, let's loop over all k of the lists and take the smallest element that we find. We must remember to keep track of where we are up to in each list.

```
1: function KWISSE_MERGE( $A[1..k][1..n_i]$ )
2:   Set  $pos[1..k] = 1$ 
3:   Set  $result[1..n]$ 
4:   for  $i = 1$  to  $n$  do
5:     Set  $min = 0$ 
6:     for  $j = 1$  to  $k$  do
```

```

7:         if pos[j] ≤ nj and (min = 0 or A[j][pos[j]] < A[min][pos[min]]) then
8:             min = j
9:         result[i] = A[min][pos[min]]
10:        pos[min] = pos[min] + 1
11:    return result

```

The value n_i denotes the length of the i^{th} list. This solution has worst-case complexity $\Theta(nk)$ since there are n elements in total and we spend $\Theta(k)$ time looking for the minimum in each iteration. There are multiple ways to write a faster algorithm for part (b). Let's have a look at two of them.

Option 1: Divide and Conquer

We can speed up the merge by doing divide and conquer. Given a sequence of k lists to merge, let's recursively merge the first $k/2$ of them, the second $k/2$ of them and then merge the results together.

```

1: function KWISE_MERGE(A[1..k][1..ni])
2:     if k = 1 then
3:         return A[1][1..n1]
4:     else
5:         Set list1 = KWISE_MERGE(A[1..k/2][1..ni])
6:         Set list2 = KWISE_MERGE(A[k/2 + 1..k][1..ni])
7:         return MERGE(list1, list2) // The ordinary 2-way merge algorithm from Mergesort

```

We recurse to a depth of $\Theta(\log(k))$ and in each level of the recursion tree we perform $\Theta(n)$ total work by doing the usual 2-way merges among the smaller subproblems in that level. In total this algorithm runs in $\Theta(n \log(k))$ time.

Option 2: Use a priority queue

The second option is to speed up the naive algorithm by using a heap / priority queue. In the naive algorithm, we spend $\Theta(k)$ time searching for the minimum element to add next. Let's just do this step by maintaining a priority queue of the next values, so that this step takes $\Theta(\log(k))$ time.

```

1: function KWISE_MERGE(A[1..k][1..ni])
2:     Set pos[1..k] = 1
3:     Set result[1..n]
4:     Set queue = PriorityQueue(1..k, key(j) = A[j][pos[j]])
5:     for i = 1 to n do
6:         Set min = queue.pop()
7:         result[i] = A[min][pos[min]]
8:         pos[min] = pos[min] + 1
9:         if pos[min] ≤ nmin then
10:            queue.push(min, key = A[min][pos[min]])
11:    return result

```

We now find the minimum element in $\Theta(\log(k))$ time hence the total complexity is $\Theta(n \log(k))$.

Finally, we cannot write an algorithm for k -wise merging that runs faster than $O(n \log(k))$ in the *comparison model*. Suppose that we have a sequence of length n and split it into n sequences of length 1 and merge them. This is just going to sort the list, which we know has a lower bound of $\Omega(n \log(n))$, and hence a merge algorithm that ran faster than $O(n \log(k))$ with $k = n$ would surpass this lower bound.

Problem 3. Consider an application of radix sort to sorting a sequence of non-empty strings of lowercase letters a to z in alphabetical order (each character of the strings can be interpreted as being base-26 for running the internal counting sort rounds). Radix sort is traditionally applied to a sequence of equal length elements, but we

can modify it to work on variable length strings by simply padding the shorter strings with empty characters at the end.

- (a) What is the time complexity of this algorithm? In what situation is this algorithm very inefficient?
- (b) Describe how the algorithm can be improved to overcome the problem mentioned in (a). The improved algorithm should have worst-case time complexity $O(n)$, where n is the sum of all of the string lengths, i.e. it should be optimal.

Solution

Let ℓ denote the length of the longest string and k the number of strings. Since we scan each string ℓ times, the complexity of this algorithm is $\Theta(\ell k)$. This is inefficient if there are many short strings in the list and only a few long ones. For example, if there were one million strings of length 10 and a single string of length one million, the algorithm would perform one trillion operations, which would take far too long.

To improve this to $\Theta(n)$ where n is the total length of all strings, we want to avoid looking at the short strings before they actually matter. Note that a string of length i only needs to be looked at in the final i iterations. To achieve this, let's first sort the strings by length. This can be done in $\Theta(\ell + k)$ using counting sort with the string's length as the key. We need to sort them shortest to longest, since we want shorter strings to come before longer strings if they share a prefix.

Then let's keep a pointer j such that the strings $S[j \dots k]$ have length $\geq i$. Each iteration when we decrement i , we decrement j while $\text{length}(S[j-1]) \geq i$ to account for strings that have just become worth considering. By doing so, we will only perform the inner sort on the strings that actually have characters in position j , hence we do not waste any time sorting on non-existent characters. This improves the complexity to $\Theta(n)$ as required. An example implementation in pseudocode is shown below.

```

1: function RADIX_SORT( $S[1 \dots k]$ )
2:   Set  $\ell = \max(\text{length}(S[1 \dots k]))$   //  $\ell$  is the length of the longest string
3:   Sort  $\text{length}(S[1 \dots k])$  by ascending length using counting sort  // Takes  $O(\ell + k)$  time
4:   Set  $j = k$ 
5:   for  $i = \ell$  to 1 do
6:     while  $j > 1$  and  $\text{length}(S[j-1]) \geq i$  do
7:        $j -= 1$ 
8:     Set  $\text{count}[a \dots z] = 0$ 
9:     for  $p = j$  to  $k$  do
10:       $\text{count}[S[p][i]] += 1$ 
11:     Set  $\text{pos}['a'] = 1$ 
12:     for  $\text{char} = 'b'$  to  $'z'$  do
13:      Set  $\text{pos}[\text{char}] = \text{pos}[\text{char}-1] + \text{count}[\text{char}-1]$ 
14:     Set  $\text{temp}[1 \dots k-j+1] = \text{null}$ 
15:     for  $p = j$  to  $k$  do
16:       $\text{temp}[\text{pos}[S[p][i]]] = S[p]$ 
17:       $\text{pos}[S[p][i]] += 1$ 
18:     Swap( $S[j \dots k]$ ,  $\text{temp}$ )

```

Problem 4. Write an in-place algorithm that takes a sequence of n integers and removes all duplicate elements from it. The relative order of the remaining elements is not important. Your algorithm should run in $O(n \log(n))$ time and use $O(1)$ auxiliary space (i.e. it must be in-place).

Solution

The easiest way to remove duplicates from a sequence would be to store them in a data structure that doesn't hold duplicates, like a binary search tree or hashtable. This would use extra space though and hence would not be an in-place solution. Instead, let's make use of the fact that when a sequence is sorted, all duplicate elements are guaranteed to be next to each other.

If we sort the sequence, we can iterate over it, and check whether an element $A[i]$ is a duplicate by checking whether $A[i] = A[i - 1]$. We will maintain an index j into the sequence such that $A[1..j]$ contains the non-duplicate elements that we have seen. Whenever we see a non-duplicate, we will copy that element into $A[j + 1]$ and increment j . This way, we will overwrite duplicates with the succeeding non-duplicates. At the end, the contents of $A[1..j]$ will contain all of the non-duplicate elements, and $A[j + 1..n]$ will contain leftovers that we will delete. An example implementation might look like this.

```
1: function REMOVE_DUPLICATES( $A[1..n]$ )
2:   sort( $A[1..n]$ )
3:    $j = 1$ 
4:   for  $i = 2$  to  $n$  do
5:     if  $A[i] \neq A[i - 1]$  then
6:        $A[j + 1] = A[i]$ 
7:        $j = j + 1$ 
8:   delete  $A[j + 1..n]$ 
```

Notice that we do not create any additional data structures and only use a constant number of variables, hence this solution is in-place provided that the sorting algorithm we use is also in-place. Let's say that we use Heapsort since it is in-place and has $\Theta(n \log(n))$ complexity. The complexity of our algorithm is dominated by the sorting, hence it takes $\Theta(n \log(n))$ time and is in-place as required.

Problem 5. The engineers in your company believe they have created a phone that is ultra-resistant against falls (perhaps even falls from as high as 150m). To test that hypothesis, your company built two prototypes and asked you to perform the test to determine the maximum height in meters (without considering fractions) that the phone can be dropped from without breaking.

There is an unknown integer value $0 \leq x \leq 150$ such that if the prototype is dropped from heights up to x meters it will not break, while if it is dropped from heights of $x + 1$ meters or higher it will break. Your job is to determine x . If one prototype is broken, it cannot be used in the tests anymore.

As a computer science student, you want to optimise your work. For doing so, you plan to develop an algorithm for determining the heights you should drop the prototype from at each iteration. No matter what is the value of $0 \leq x \leq 150$, your algorithm should be correct and determine the value of x . For an algorithm A and an unknown integer $0 \leq x \leq 150$, let $\text{Drop}(A, x)$ denote the number of times you need to drop a prototype if you use algorithm A and x is the threshold. Let $\text{Drop}(A)$ denote the worst-case performance of A , which is given by the maximum of $\text{Drop}(A, x)$ over all possible $0 \leq x \leq 150$.

For an optimal deterministic algorithm A that has $\text{Drop}(A)$ as small as possible, what is the value of $\text{Drop}(A)$?

[Hint: Since you have two prototypes, it is possible to do better than linear search. But as you only have two prototypes, you need to be very careful if only one prototype remains intact (and thus the search cannot be as efficient as binary search).]

Solution

Let h_1, h_2, h_3, \dots denote the sequence of heights that you will drop the first prototype from until you either break it or test the height 150m without breaking the prototype. We assume without loss of generality that this is an increasing sequence.

One might initially think that the ideal strategy would be to have h_1, h_2, h_3, \dots equally spaced, but is that really an optimal strategy?

If you break the first prototype while dropping it from height h_i for $i > 1$, what you now know is that $h_{i-1} \leq x < h_i$ and that you haven't dropped a prototype from any height in between h_{i-1} and h_i . As moving forward you would only have one prototype left, and your algorithm needs to find the correct value of x in all situations, you would need to move cautiously and do a linear search on heights $h_{i-1} + 1, h_{i-1} + 2, h_{i-1} + 3, \dots, h_i - 1$ until you either break the second prototype (but by breaking it, you are able to determine the correct value of x at that moment), or you finish testing all this sequence without breaking the second prototype (in which case $x = h_i - 1$). In the edge case that the first prototype is broken already when dropped from h_1 , you would need to do a linear search on $1, 2, 3, \dots, h_1 - 1$ with the second prototype.

So let's analyse what is the worst-case behaviour on each of the possible scenarios for when the first prototype is broken:

1. If you break the first prototype when dropping it from height h_1 , then you are doing 1 drop of the first prototype, plus $|h_1 - 1|$ drops of the second prototype in the worst case, for a total of $1 + |h_1 - 1|$ drops.
2. If you break the first prototype when dropping it from height h_2 , then you are doing 2 drops of the first prototype, plus $|h_2 - h_1 - 1|$ drops of the second prototype in the worst case, for a total of $2 + |h_2 - h_1 - 1|$ drops.
3. If you break the first prototype when dropping it from height h_3 , then you are doing 3 drops of the first prototype, plus $|h_3 - h_2 - 1|$ drops of the second prototype in the worst case, for a total of $3 + |h_3 - h_2 - 1|$ drops.
4. If you break the first prototype when dropping it from height h_4 , then you are doing 4 drops of the first prototype, plus $|h_4 - h_3 - 1|$ drops of the second prototype in the worst case, for a total of $4 + |h_4 - h_3 - 1|$ drops.
5. And so on.

So if you are looking to minimise $\text{Drop}(A)$, you want to balance those cases out and pick the sequence of heights such that the values $|h_1 - 1|, |h_2 - h_1 - 1|, |h_3 - h_2 - 1|, |h_4 - h_3 - 1|, \dots$ form a decreasing sequence in which each element is smaller than the previous by 1 (apart from borderline conditions for the last difference, when you only need to drop at most from the height of 150m).

Therefore if you let $h_1 = n$, then $h_2 = n + (n - 1)$, $h_3 = n + (n - 1) + (n - 2)$, $h_4 = n + (n - 1) + (n - 2) + (n - 3)$, and so on.

Summing it up, the problem is essentially about finding the smallest n such that $n + (n - 1) + (n - 2) + (n - 3) + \dots + 1 > 150$.

Using the formula for the sum of arithmetic series, you want to find the smallest n such that $\frac{n(n+1)}{2} > 150$, which is $n = 17$.

The sequence of heights for the first prototype is 17, 33, 48, 62, 75, 87, 98, 108, 117, 125, 132, 138, 143, 147, 150 (you still have unused capacity to test for heights up to 153m with the same amount of worst-case drops by continuing this initial sequence with 152 and 153). If at some point the first prototype is broken, you just use the second prototype to linearly test all heights between the previous element of the sequence and the current one. You can check that no matter what is the value of x you would always find it with at most 17 drops and only 2 prototypes by using this strategy.

Problem 6. Devise an efficient online algorithm¹ that finds the smallest k elements of a sequence of integers. Write pseudocode for your algorithm. [Hint: Use a data structure that you have learned about in a previous unit]

¹In this case, online means that you are given the numbers one at a time, and at any point you need to know which are the smallest k .

Solution

An offline method to solve this problem would be to find the k^{th} smallest element of the sequence and then take all elements less than it, but this solution would not be online since the k^{th} smallest element may change if we increase the input size.

Each time we see a new number, we need to know if this should go in our set of k smallest numbers or not. This means we need to compare it to the *maximum* element in our set. If our new number is smaller than the existing max, we should remove the max, insert our new number into our smallest k elements, and then *still* have fast access to whatever the new maximum is (so that we are ready to do this process again for the next number). For these operations (get-max, delete-max, insert) we should think of a heap.

Since we need fast access to the max, we will use a max-heap. Whenever we encounter a new element, we know that it should become part of the solution if it is smaller than the current k^{th} smallest element, i.e. the largest value in the max-heap. We can check this in $\Theta(1)$ with a max-heap, and swap out the maximum value with the new value in $\Theta(\log(k))$ time if it is smaller.

We can write a solution that looks like this.

```
1: function K_MINIMUM_ELEMENTS( $A[1\dots]$ ,  $k$ )
2:   Initialize an empty MaxHeap named  $k\_smallest$ 
3:   for each incoming element  $e$  do
4:     if  $k\_smallest.size() < k$  then
5:        $k\_smallest.push(e)$ 
6:     else if  $e < k\_smallest.max\_element()$  then
7:        $k\_smallest.pop()$ 
8:        $k\_smallest.push(e)$ 
9:   report  $k\_smallest$ 
```

Since we can add more elements to the sequence A at any time and the algorithm will still work, this solution is online. It runs in $\Theta(n \log(k))$ time assuming that the heap is a binary heap with operations costing $\Theta(\log(k))$ for a heap of size k (where n is the number of elements processed).

Supplementary Problems

Problem 7. Write pseudocode for insertion sort, except instead of sorting the elements into non-decreasing order, sort them into non-increasing order. Identify a useful invariant of this algorithm.

Solution

We write the usual insertion sort algorithm, except that when performing the insertion step, we loop as long as $A[j] < \text{key}$ (rather than $A[j] > \text{key}$), so that larger elements get moved to the left.

```
1: function INSERTION_SORT( $A[1..n]$ )
2:   for  $i = 2$  to  $n$  do
3:     Set  $\text{key} = A[i]$ 
4:     Set  $j = i - 1$ 
5:     while  $j \geq 1$  and  $A[j] < \text{key}$  do
6:        $A[j + 1] = A[j]$ 
7:        $j = j - 1$ 
8:      $A[j + 1] = \text{key}$ 
```

A useful invariant is that at the end of iteration i , the sub-array $A[1..i]$ is sorted in non-increasing order.

Problem 8. Consider the following algorithm that returns the minimum element of a given sequence A . Identify a useful invariant that is true at the beginning of each iteration of the **for** loop. Prove that it holds, and use it to show that the algorithm is correct.

```

1: function MINIMUM_ELEMENT( $A[1..n]$ )
2:    $\text{min} = A[1]$ 
3:   for  $i = 2$  to  $n$  do
4:     if  $A[i] < \text{min}$  then
5:        $\text{min} = A[i]$ 
6:   return  $\text{min}$ 

```

Solution

A useful invariant is that at the start of every iteration, the value of min is the minimum element in the subarray $A[1..i-1]$.

Proof: At the start of the first iteration, for the invariant to hold we need min to be the minimum element of the subarray $A[1..1]$. This is just one element, namely $A[1]$, and min is initialised to $A[1]$, so the invariant holds at the start of the first iteration.

Suppose the invariant holds at the start of some iteration, namely, that min is the minimum element of the subarray $A[1..i-1]$. We want to show that the invariant still holds at the start of the next iteration, namely, that min is the minimum element of the subarray $A[1..i]$. During each iteration, the algorithm compares $A[i]$ to min . If $A[i] < \text{min}$, then we know that $A[i]$ is the minimum element of the subarray $A[1..i]$, since it is less than min and min is the minimum element of the subarray $A[1..i-1]$ by the invariant. So after we set $\text{min} = A[i]$, the invariant holds. If $A[i]$ is not less than min , then since min was already the minimum of the subarray $A[1..i-1]$ and $A[i]$ is not less than it, min is also the minimum element of the subarray $A[1..i]$. So we do not need to do any work to preserve the invariant, and indeed the algorithm does not modify anything in this case, so the invariant holds. Since we know the invariant holds at the start of the first loop, by induction it holds for all values of i , including at the termination of the loop when $i = n + 1$.

To prove the algorithm is correct, we need to show that at loop termination, min is the minimum element in A . The invariant tells us that min is the minimum of the subarray $A[1..i-1]$, but at loop termination, $i = n + 1$, so min is the minimum element in $A[1..n]$ which is all of A . Therefore the algorithm is correct.

Problem 9. Consider the problem of finding a target value in a sequence (not necessarily sorted). Given below is pseudocode for a simple linear search that solves this problem. Identify a useful loop invariant of this algorithm and use it to prove that the algorithm is correct.

```

1: function LINEAR_SEARCH( $A[1..n]$ ,  $\text{target}$ )
2:   Set  $\text{index} = \text{null}$ 
3:   for  $i = 1$  to  $n$  do
4:     if  $A[i] = \text{target}$  then
5:        $\text{index} = i$ 
6:   return  $\text{index}$ 

```

Solution

In this case, a useful loop invariant would be that at the end of iteration i , index is equal to the largest $j \leq i$ such that $A[j] = \text{target}$, or **null** if target is not in $A[1..i]$. In the last iteration of the loop, $i = n$ and hence index is equal to some j such that $A[j] = \text{target}$, or **null** if target is not in $A[1..n]$, which is correct.

Problem 10. Write an iterative Python function that implements binary search on a sorted, non-empty list, and returns the position of the key, or None if it does not exist.

- (a) If there are multiple occurrences of the key, return the position of the **final** one. Identify a useful invariant of your program and explain why your algorithm is correct
- (b) If there are multiple occurrences of the key, return the position of the **first** one. Identify a useful invariant of your program and explain why your algorithm is correct

Solution

For part (a), the implementation of binary search in the lecture notes satisfies this property. Note that since the invariant is $\text{array}[\text{lo}] \leq \text{key}$ and $\text{array}[\text{hi}] > \text{key}$, lo will point to the final element of array that is not greater than key . This means that if there are multiple occurrences of key , lo will point to the last one.

For part (b), we make a slight adjustment to the algorithm in the lecture notes. We modify the binary search such that we maintain the following similar invariant, $\text{array}[\text{lo}] < \text{key}$ and $\text{array}[\text{hi}] \geq \text{key}$. To do so, we must change the initial values of lo and hi to 0 and n respectively, change the condition of the **if** statement appropriately and change the final check to $\text{array}[\text{hi}] = \text{key}$. This works because when the algorithm terminates, hi is now pointing to the first element that is at least as large as the key. This means that if there are multiple occurrences of key , hi will point to the first one. Here is some pseudocode for concreteness.

```

1: function BINARY_SEARCH(array[1..n], key)
2:   Set lo = 0 and hi = n
3:   while lo < hi - 1 do
4:     Set mid =  $\lfloor (\text{lo} + \text{hi})/2 \rfloor$ 
5:     if key > array[mid] then lo = mid
6:     else hi = mid
7:   if array[hi] = key then return hi
8:   else return null

```

Problem 11. Devise an algorithm that given a sorted sequence of distinct integers a_1, a_2, \dots, a_n determines whether there exists an element such that $a_i = i$. Your algorithm should run in $O(\log(n))$ time.

Solution

Since the elements of a are sorted and distinct, it is true that $a_i \leq a_{i+1} - 1$. We are seeking an element such that $a_i = i$, which is equivalent to searching for an element such that $a_i - i = 0$. Since $a_i \leq a_{i+1} - 1$, the sequence $(a_i - i)$ for all i is a non-decreasing sequence, because

$$a_i - i \leq a_{i+1} - 1 - i = a_{i+1} - (i + 1).$$

Therefore the problem becomes searching for zero in the non-decreasing sequence $a_i - i$, which can be solved using binary search. See the pseudocode implementation below.

```

1: function VALUE_INDEX(a[1..n])
2:   if a[n] - n = 0 then return True
3:   if a[n] - n < 0 then return False
4:   if a[1] - 1 > 0 then return False
5:   // Invariant: a[lo] - lo ≤ 0, a[hi] - hi > 0
6:   Set lo = 1, hi = n + 1
7:   while lo < hi - 1 do
8:     Set mid =  $\lfloor (\text{lo} + \text{hi})/2 \rfloor$ 
9:     if a[mid] - mid ≤ 0 then lo = mid
10:    else hi = mid
11:   if a[lo] = lo then return True
12:   else return False

```


Problem 12. Consider the following variation on the usual implementation of insertion sort.

```

1: function FAST_INSERTION_SORT( $A[1..n]$ )
2:   for  $i = 2$  to  $n$  do
3:     Set  $\text{key} = A[i]$ 
4:     Binary search to find max  $k < i$  such that  $A[k] \leq \text{key}$ 
5:     for  $j = i$  downto  $k + 1$  do
6:        $A[j] = A[j - 1]$ 
7:      $A[k] = \text{key}$ 

```

- What is the number of comparisons performed by this implementation of insertion sort?
- What is the worst-case time complexity of this implementation of insertion sort?
- What do the above two facts imply about the use of the comparison model (analysing a sorting algorithm's complexity by the number of comparisons it does) for analysing time complexity?

Solution

For (a), note that we perform a binary search for each element of the sequence. The worst-case complexity of performing those n searches is $\Theta(n \log(n))$ comparisons.

For (b), note that in the worst case we will have to swap every element all the way to the beginning of the sequence. This means that we will perform $\Theta(n^2)$ swaps and hence the worst-case complexity will be $\Theta(n^2)$.

Comparing (a) and (b) reveals an interesting fact. Although we perform just $\Theta(n \log(n))$ comparisons which is optimal in the comparison model, we still take $\Theta(n^2)$ time to sort due to the swaps required. This shows that the comparison model is not always suitable for proving upper bounds on the running times of sorting algorithms, but rather, its main purpose is for proving lower bounds. This algorithm might still be useful if the items being sorted were expensive to compare but fast to swap however. For example, it would perform okay for sorting a sequence of large strings since the comparisons would be the bottleneck, and the swaps could be done fast (assuming an implementation that allows moving strings in $O(1)$).

Problem 13. (Advanced) Consider the problem of sorting one million 64-bit integers using radix sort.

- Write down a formula in terms of b for the number of operations performed by radix sort when sorting one million 64-bit integers in base b .
- Using your preferred program (for example, Wolfram Alpha), plot a graph of this formula against b and find the best value of b , the one that minimises the number of operations required. How many passes of radix sort will be performed for this value of b ?
- Implement radix sort and use it to sort one million randomly generated 64-bit integers. Compare various choices for the base b and see whether or not the one that you found in Part (b) is in fact the best.

Solution

Recall that the complexity of radix sort for n integers with k digits in base b is

$$\Theta(k(n + b)).$$

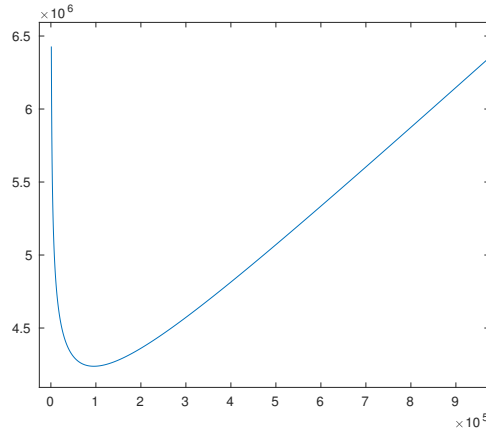
Since our integers are 64 bits, the number of digits k will be equal to

$$k = \frac{64}{\log_2(b)},$$

and hence the number of operations will be proportional to

$$\frac{64}{\log_2(b)} (10^6 + b).$$

A plot of this function looks like:



Your favourite mathematics software will then tell you that the minimum of this function occurs for $b = 95536$. Since $\log_2(95536) \approx 16.5$, and the closest divisor of 64 to that is 16, a base of $2^{16} = 65536$ should perform optimally.

My test implementation found the following results for various bases. Your results should hopefully look similar.

Base (b)	In base-2	Time (seconds)
4	2^2	23.52
16	2^4	11.77
256	2^8	6.19
65536	2^{16}	3.38
1048576	2^{20}	4.63
16777216	2^{24}	14.87

This seems to confirm our prediction that $b = 2^{16}$ would be the optimal base. Of course, the bases 2^{20} and 2^{24} are naturally disadvantaged since 20 and 24 do not divide 64, but attempting to test base 2^{32} caused my computer to run out of RAM.

Problem 14. (Advanced) When we analyse the complexity of an algorithm, we always make assumptions about the kinds of operations we can perform, how long they will take, and how much space that we will use. We call this the *model of computation* under which we analyse the algorithm. The assumptions that we make can lead to wildly different conclusions in our analysis.

An early model of computation used by computer scientists was the *RAM*, the Random-Access Machine. In the RAM model, we assume that we have unlimited memory, consisting of *registers*. Each register has an *address* and some contents, which can be any integer. Additionally, integers are used as pointers to refer to memory addresses. A RAM is endowed with certain operations that it is allowed to perform in constant time. The total amount of space used by an algorithm is the total size of all of the contents of the registers used by it.

- State some unrealistic aspects of the RAM model of computation.
- Explain why the definition of an in-place algorithm being those which use $O(1)$ auxiliary space is near worthless in this description of the RAM model.

- (c) In the Week 2 Applied, we discussed fast algorithms for computing Fibonacci numbers. In particular, we saw that $F(n)$ can be computed using matrix powers, which can be computed in just $O(\log(n))$ multiplications. If a RAM is endowed with all arithmetic operations, then we can compute $F(n)$ in $O(\log(n))$ time using this algorithm. If instead the RAM is only allowed to perform addition, subtraction, and bitwise operations in constant time, we can still simulate multiplication using any multiplication algorithm (for example, Karatsuba multiplication). Explain why in this model, it is impossible to compute $F(n)$ in $O(\log(n))$ time with this algorithm.

A model that is more commonly used in modern algorithm and data structure analysis is the *word RAM* (short for word Random-Access Machine). In the word RAM model, every word is a fixed-size w -bit integer, where w is a parameter of the model. We can perform all standard arithmetic and bitwise operations on w -bit integers in constant time. The total amount of space used by an algorithm is the number of words that it uses

- (d) What is the maximum amount of memory that can be used by a w -bit word RAM?
- (e) Suppose we wish to solve a problem whose input is a sequence of size n . What assumption must be made about the model for this to make sense?
- (f) Discuss some aspects that the word RAM still fails to account for in realistic modern computers
- (g) Does the word RAM model allow us to compute $F(n)$, the n^{th} Fibonacci number, in $O(\log(n))$ time?

Solution

The RAM model

- (a) The RAM model is unrealistic for several reasons. Here is a non-exhaustive list of possible reasons:
- It has an infinite amount of memory. Although this is unrealistic, most models of computation are similar, since if we restrict ourselves to a finite amount of memory, then technically every problem is solvable in $O(1)$ time since there are only a finite number of possible inputs to the problem and the input size is constant.
 - It can store arbitrarily large integers in a single register and operate on them in constant time. This is the most overpowered part of the model. In fact, under certain assumptions, it is possible to show that certain problems like integer sorting can be solved in $O(1)$ in this model! This is because the machine can cheat by concatenating arbitrarily many integers into a single register and then operating on them in constant time. If you are interested in these kinds of strange problems, you should read Michael Brand's PhD thesis *Computing with Arbitrary and Random Numbers*.
- (b) If we define an in-place algorithm as one that takes $O(1)$ space, then in the RAM model we are instantly dead. A problem that takes as input a sequence of size n requires pointers of size $O(\log(n))$ to refer to those elements, which is not constant. It is therefore common in complexity theory to use different definitions of in-place. Examples including defining in-place as $O(\log(n))$ space, as $O(1)$ space not counting pointers, or as $o(n)$ space (that's a little- O , not a big- O , meaning asymptotically smaller than. For example $\log(n)$, \sqrt{n} , and $n^{0.99}$ are all $o(n)$).
- (c) Computing Fibonacci numbers using matrix multiplication takes $O(\log(n))$ multiplications. If multiplications can be performed in constant time then we are happy and can compute $F(n)$ in just $O(\log(n))$ time. Otherwise, things get trickier. If multiplication cannot be done in constant time, then it needs to be done manually using a multiplication algorithm, whose running time is dependant on the length of the integers being multiplied. Since the Fibonacci numbers grow exponentially, specifically like ϕ^n where $\phi \approx 1.618$ is the golden ratio, they have $\Theta(\log(\phi^n)) = \Theta(n)$ digits. Even if we had a linear time multiplication algorithm (we do not), this means that it would take $\Omega(n \log(n))$ time to compute $F(n)$ since those multiplications cost $\Omega(n)$.

The word-RAM model

- (d) The largest integer that can be written with w bits is 2^w . Since we use integers as pointers to address memory, we can address at most 2^w distinct registers and hence we cannot use more than $O(2^w)$ memory.
- (e) To address a sequence of size n , we need to be able to store pointers to at least n different registers. A pointer referring to address n requires $\log_2(n)$ bits to store, hence we require that $w \geq \log_2(n)$.
- (f) While the word-RAM addresses many of the failings of the RAM, it still does not account for the following:
 - Parallel computation
 - Memory hierarchies - the differences in speed between reading from disk, reading from main memory or reading from cache
- (g) We cannot compute $F(n)$ in $O(\log(n))$ time on a word-RAM. Even though we can multiply w -bit integers in constant time (the contents of a single register), since $F(n)$ has $\Theta(n)$ bits, it would have to be split across $\Theta(n/w)$ different registers, and multiplication would have to be performed using a multiplication algorithm that would take $\Omega(n/w)$ time.