Week 3 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial 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 the tutorial 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, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. Write psuedocode 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 $A[j] < \texttt{key}$, rather than $A[j] > \texttt{key}$, 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 key = A[i]
4:          Set j = i - 1
5:          while j ≥ 1 and A[j] < key do
6:              A[j+1] = A[j]
7:              j = j - 1
8:          end while
9:          A[j+1] = key
10:     end for
11: end function
```

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

Problem 2. Consider the following algorithm that returns the number of occurrences of *target* in the sequence A. Identify a useful invariant that is true at the beginning of each iteration of the **while** loop. Prove that it holds, and use it to prove that the algorithm is correct.

```
1:  function COUNT(A[1..n], target)
2:      count = 0
3:      i = 1
4:      while i ≤ n do
5:          if A[i] = target then
6:              count = count + 1
7:          end if
8:          i = i + 1
9:      end while
10:     return count
11: end function
```

Solution

A useful invariant is that, at the start of iteration i, `count` is equal to the number of occurrences of `target` in $A[1..i-1]$, where we consider $A[1..0]$ to be an empty list.

Note: To prove this loop invariant, we will use induction. First we show that the invariant holds at initialisation, at the start of the first iteration of the loop. This is our base case. Next we assume that the invariant holds at the start of some iteration of the loop, and show that it still holds at the start of the next iteration. At this point we are done, since we have shown that the invariant holds at the start of the first loop, and that if it holds at the start of loop i, it also holds at the start of loop $i+1$. This means it holds at the start of every loop, and importantly, that it holds at the start of the loop where the loop condition is false, i.e., it holds when the loop ends.

Proof: At the start of the first iteration, $i = 1$. Also, `count` $= 0$, so `count` is equal to the number of occurrences of `target` in $A[1..0]$, since $A[1..0]$ is an empty list. So the invariant is true at initialisation.

Assume that the invariant holds at the start of k-th iteration. So `count` is equal to the number of occurrences of `target` in $A[1..k-1]$. Call this number of occurrences c. During this iteration of the loop, $i = k$. If $A[k] =$ `target`, we will increment `count`, so `count` will equal $c+1$, which is the number of occurrences of `target` in $A[1..k]$. If $A[k] \neq$ `target`, then `count` will not be changed, so `count` will equal c, which is equal to the number of occurrences of `target` in $A[1..k]$. Either way the invariant holds at the start of $k+1^{th}$ iteration, that is, `count` is equal to the number of occurrences of `target` in $A[1..k]$. Since we know the invariant holds at the start, by induction it holds for all values of i, including when $i = n+1$, so the invariant holds.

To prove the algorithm is correct, we need to show that at loop termination, `count` is equal to the number of occurrences of `target` in A. The invariant tells us that `count` is equal to the number of occurrences of target in $A[1..i-1]$, but at loop termination, $i = n+1$, so `count` is equal to the number of occurrences of target in $A[1..n]$ which is all of A. Therefore the algorithm is correct.

Tutorial Problems

Problem 3. What are the complexities of these two approaches to stable counting sort outlined in lecture 2? What are the advantages and disadvantages of each?

Solution

Position array count sort (approach 1): The total space required is $O(B+N)$, since we need to create two arrays of size B (count and position) and we need to create the output which is size N. The time required is also $O(B+N)$, since we need to iterate through the input to create `count` ($O(N)$), then iterate through `count` to create `position` ($O(B)$), then iterate through the input again and place each input element in the correct position in the output ($O(N)$).

Buckets count sort (approach 2): The total space required is $O(B+N)$ where B is the base (since we need an array of size B) and N is the number of elements to be sorted (since we need to store every element in one of the linked lists that make up the buckets). The time required is also $O(B+N)$, since we need to create the array ($O(B)$), iterate through the input, appending each element to one of the linked lists ($O(N)$), then iterate through each of the linked lists and append all the elements to the output list ($O(N)$).

Though the complexities are identical, there are differences between the two approaches. Approach 2 is conceptually simpler as it requires less bookkeeping. However, it is less efficient in practice. It stores $O(N)$ data in B linked lists, and linked lists have more overhead than arrays (since they need to store pointers as well as data). Additionally, linked lists are not contiguous in memory, which means that traversing a

linked list results in many more cache misses than traversing an array.

Problem 4. 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:     min = A[1]
3:     for i = 2 to n do
4:         if A[i] < min then
5:             min = A[i]
6:         end if
7:     end for
8:     return min
9: end function
```

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 $A[i] <$ `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 `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` 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 5. 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 $\{a_1, a_2, \cdots, a_n\}$ 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. Since these pairs are unique, any sorting algorithm will sort them correctly, and 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 6. 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] ≠ A[i − 1] then
6:              A[j + 1] = A[i]
7:              j = j + 1
8:          end if
9:      end for
10:     delete A[j + 1..n]
11: end function
```

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 $O(n\log(n))$ complexity. The complexity of our algorithm is dominated by the sorting, hence it takes $O(n\log(n))$ time and is in-place as required.

Problem 7. Think about and discuss with those around you why auxiliary space complexity is a useful metric. Why is it often more informative than total space complexity? For the purpose of this problem, define auxiliary space complexity as the amount of space required by an algorithm, excluding the space taken by the input, and define total space complexity as the space taken by an algorithm, including the space taken by the input.

Solution

Auxiliary space complexity is useful to measure since it helps us to distinguish between algorithms that use at most the same amount of space as the input. If two algorithms take an input of size $O(n)$ and have $O(n)$ space complexity, we can not actually tell how much additional memory they use from this measurement. An algorithm that uses $O(1)$ additional space in addition to the input uses much less memory than one that uses $O(n)$ space, but the two are indistinguishable if we only measure total space.

Problem 8. Devise an efficient online algorithm[1] that finds the smallest k elements of a sequence of integers. Write psuedocode for your algorithm. [Hint: Use a data structure that you have learned about in a previous unit]

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 $O(1)$ with a max-heap, and swap out the maximum value with the new value in $O(\log(k))$ time if it is smaller.

We can write a solution that looks like this.

```
1:  function K_MINIMUM_ELEMENTS(A[1...], 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:          end if
10:         report k_smallest
11:     end for
12: end function
```

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 $O(n\log(k))$ time assuming that the heap is a binary heap with operations costing $O(\log(k))$ for a heap of size k (where n is the number of elements processed).

Problem 9. 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

[1] In 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

(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 (Chapter 1) satisfies this property. Note that since the invariant is `array[lo]` $\leq$ `key` and `array[hi]` $>$ `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, `array[lo]` $<$ `key` and `array[hi]` $\geq$ `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 `array[hi] = 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 psuedocode 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 = ⌊(lo + hi)/2⌋
5:          if key > array[mid] then lo = mid
6:          else hi = mid
7:      end while
8:      if array[hi] = key then return hi
9:      else return null
10: end function
```

Problem 10. 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 KWISE_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 = 1
6:          for j = 1 to k do
7:              if pos[j] ≤ n_j and (pos[min] = n_j + 1 or A[j][pos[j]] < A[min][pos[min]]) then
8:                  min = j
9:              end if
10:         end for
11:         result[i] = A[min][pos[min]]
```

```
12:         pos[min] = pos[min] + 1
13:     end for
14:     return result
15: end function
```

The value n_i denotes the length of the i^{th} list. This solution has complexity $O(nk)$ since there are n elements in total and we spend $O(k)$ time looking for the minimum 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..n_i])
2:     if k = 1 then
3:         return A[1][1..n_1]
4:     else
5:         Set list1 = KWISE_MERGE(A[1..k/2][1..n_i])
6:         Set list2 = KWISE_MERGE(A[k/2+1..k][1..n_i])
7:         return MERGE(list1, list2)    // The ordinary 2-way merge algorithm from Mergesort
8:     end if
9: end function
```

We recurse to a depth of $O(\log(k))$ and perform n work at each step by doing the usual 2-way merge. In total this algorithm runs in $O(n\log(k))$ 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 $O(k)$ 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 $O(\log(k))$ time.

```
1: function KWISE_MERGE(A[1..k][1..n_i])
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] ≤ n_min then
10:            queue.push(min, key = A[min][pos[min]])
11:        end if
12:    end for
13:    return result
14: end function
```

This implementation is the same as the first one except that we now find the minimum element in $O(\log(k))$ time hence the total complexity is $O(n\log(k))$.

Finally, we can not 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 than ran faster than $O(n\log(k))$ with $k = n$ would surpass this lower bound.

Supplementary Problems

Problem 11. Consider the problem of finding a target value in sequence (not necessarily sorted). Given below is psuedocode 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], target)
2:     Set index = null
3:     for i = 1 to n do
4:         if A[i] = target then
5:             index = i
6:         end if
7:     end for
8:     return index
9: end function
```

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 $A[j] =$ 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 $A[j] =$ target, or **null** if target is not in $A[1..n]$, which is correct.

Problem 12. Devise an algorithm that given a sorted sequence of distinct integers $a_1, a_2, ..., a_n$ 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 psuedocode implementation below.

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

Problem 13. 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 key = A[i]
4:        Binary search to find max k < i such that A[k] ≤ key
5:        for j = k + 1 to i do
6:            A[j] = A[j − 1]
7:        end for
8:        A[k] = key
9:    end for
10: end function
```

(a) What is the number of comparisons performed by this implementation of insertion sort?

(b) What is the worst-case time complexity of this implementation of insertion sort?

(c) 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. Each binary search takes $O(\log(n))$ comparisons, hence we perform $O(n\log(n))$ 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 $O(n^2)$ swaps and hence the worst-case complexity will be $O(n^2)$.

Comparing (a) and (b) reveals an interesting fact. Although we perform just $O(n\log(n))$ comparisons which is optimal in the comparison model, we still take $O(n^2)$ 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 14. (Advanced) Consider the problem of integer multiplication. For two integers with n digits, the standard multiplication algorithm that you learned in school runs in $O(n^2)$ time. Interestingly, we can do much better than this. We will consider integers written in binary and suppose that n is a power of 2, but the techniques described here work in any base. Let's analyse a divide-and-conquer approach in which we split each of the integers into two halves and recursively multiply the two halves. Given two integers X and Y of n bits each, we write X_L, X_R, Y_L, Y_R for the left and right halves of X and Y respectively. X and Y are decomposed as follows

$$X = (2^{\frac{n}{2}} X_L + X_R), \qquad Y = (2^{\frac{n}{2}} Y_L + Y_R),$$

and hence we can write their product as

$$XY = (2^{\frac{n}{2}} X_L + X_R)(2^{\frac{n}{2}} Y_L + Y_R) = 2^n X_L Y_L + 2^{\frac{n}{2}}(X_L Y_R + X_R Y_L) + X_R Y_R. \tag{1}$$

Suppose we implement a divide-and-conquer algorithm that uses this formula directly to compute the product. Note that multiplication by a power of two takes $O(n)$ time since this is just a shift operation, and keep in mind that adding two n bit numbers takes $O(n)$ time.

(a) Write a recurrence relation for the time complexity of such an algorithm

(b) Solve the recurrence relation and write the time complexity in big-O notation

The above method requires four recursive calls since there are four products to compute in the given formula. Suppose instead that we make use of the following formula

$$(X_L Y_R + X_R Y_L) = (X_L + X_R)(Y_R + Y_L) - X_R Y_R - X_L Y_L.$$

If we use this formula for evaluating Equation (1), we can reduce the required number of multiplications to three, so an algorithm based on this formula should be faster.

(c) Write a recurrence relation for the time complexity of such an algorithm

(d) Solve the recurrence relation and write the time complexity in big-O notation

(e) Implement this algorithm in Python. Compare it against the naive $O(n^2)$ algorithm for very large numbers and measure the difference in running time

This algorithm is known as Karatsuba multiplication and is what Python uses to multiply very large numbers. Faster algorithms based on the Fast Fourier Transform exist (see the Schönhage–Strassen algorithm), but these are rarely used in practice due to having very large constant factors.

Solution

For part (a) and (b), such an algorithm would make 4 recursive calls after performing linear work for the additions and shifting (multiplication by a power of two). Each recursive call deals with numbers with half as many digits, hence a recurrence for the running time is

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + cn & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

By telescoping (see the Week 2 tutorial sheet if unfamiliar), we observe that the solution satisfies

$$T(n) = 4^k T\left(\frac{n}{2^k}\right) + (1 + 2 + \ldots + 2^{k-1})cn = 4^k T\left(\frac{n}{2^k}\right) + 2^k cn.$$

Letting $k = \log_2(n)$, we obtain

$$T(n) = 4^{\log_2(n)} T(1) + 2^{\log_2(n)} cn = n^2 + cn^2 = O(n^2),$$

hence this algorithm is no faster than the ordinary multiplication algorithm.

For part (c) and (d), this improvement allows us to perform only three recursive multiplications ($(X_L + X_R) \times (Y_R + Y_L)$, $X_R \times Y_R$, and $X_L \times Y_L$), hence a recurrence for the running time is

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + cn & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

By telescoping just as we did above, we find that a solution satisfies

$$T(n) = 3^{\log_2(n)} + cn\left(1 + \frac{3}{2} + \frac{3^2}{2^2} + \ldots + \frac{3^{k-1}}{2^{k-1}}\right) = n^{\log_2(3)} + 2cn(n^{\log_2\left(\frac{3}{2}\right)} - 1).$$

Since $\log_2(3) \approx 1.585$ and $\log_2\left(\frac{3}{2}\right) \approx 0.585$, we have

$$T(n) = O(n^{1.585}),$$

which is asymptotically faster than the ordinary multiplication algorithm.

Problem 15. (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 computed 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.

(a) State some unrealistic aspects of the RAM model of computation

(b) 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 tutorial, 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 shown in Problem 14). 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 can not 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

(a) The largest integer that can write 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 can not use more than $O(2^w)$ memory.

(b) To address a sequence of size n, we need to be able 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 $w \geq \log_2(n)$.

(c) 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

(d) We can not 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.