Homework 1

Gleici Pereira, Rachel Collier

Feb 2023

1

Write a recursive function to find the minimum element of an array A with n elements. Assume no duplicates and the array is not sorted.

1.1 minElem Algorithm

Algorithm 1 Linear search
Input: An array A with n elements
Output: Minimum element of A

```
function minElem(A[1...n])
    if n = 1 then
        return A[n]
    else
        mid = n/2
        min1 = minElem(A[1 ... mid])
        min2 = minElem(A[mid+1 ... n])
    end if
    if min1 ≤ min2 then
        return min1
    else
        return min2
    end if
end function
```

1.2 Time Complexity

The worst-case scenario for a recursive linear search is $\Theta(n)$.
Proof. **Recurrence relation:**

$$T(n) = T(\frac{n}{2}) \ + T(\frac{n}{2}) \ + 1 \tag{1}$$

$$T(n) = 2T(\frac{n}{2}) \ + 1 \tag{2}$$

That is because the costly operations for this algorithm are the comparisons; thus, for each recursive call the program divides array A in half and performs two comparisons plus the assignment operation which is constant.

Solving the above recurrence relation by analyzing the level-by-level series of the recurrence tree, it shows the series decays exponentially, every term is a constant factor smaller than the previous term, and the sum is dominated by the root of the tree. If the level-by-level series is decreasing exponentially, then $T(n) = \theta(f(n))$.

Therefore, for the above recurrence $T(n) = \theta(n)$.

If this algorithm is converted to an iterative approach, the loop invariant would be the fact that the *min* variable would always hold the minimum value found so far. This condition is what makes the algorithm be correct because it is true before the loop begins, before each iteration and after the loop ends, thus returning the correct value for min.

References:
`https://www2.cs.uh.edu/~panruowu/2022f_cosc3320/lec2_recursion.pdf`
Pandurangan, Gopal. (2022). *Algorithms: An Intuitive Approach*

2

Write a Quicksort algorithm that has two subscripts (Hoare's version) for the partition routine and that can handle duplicate values. Estimate and contrast the number of comparisons and swaps between Hoare's and Lemuto's. Show the lop invariant for the partition function.

2.1 Partition Algorithm

Algorithm 2 Algorithm

for $i \leftarrow 1\ n-1$ **do**

  if $A[i] < A[n]$ **then** $\leftarrow +1$
swap $A[] \leftrightarrow A[i]$ swap $A[n] \leftrightarrow A[+1]$
return $+1$

2.2 Analysis

Hoare proposed a more complicated two partitioning algorithm that has some partitioning algorithm that has some practical advantages over Lemuto's algorithm. Depending on the quickest algorithm you find, Hoare's algorithm is

better. This quicksort algorithm has two subscripts, one that goes to the left to the right. The other one that goes from the right to the left, then they cross each other. It has less comparisons and less assignments. Hoare's version, the time complexity doesn't change anything, it depends on how you pick the algorithm.

3

Slow and fast primality checking: to check if a number n is prime. Clarify O() of input size (n), depending on the programming language.

3.1

Solution

Both IsPrime and FastIsPrime algorithms from Gopal's textbook are created under the mathematical definition of a prime number. The only difference is that while IsPrime checks primality by attempting to divide every i by n (starting from $i = 2$ up to n), FastIsPrime makes use of the theorem that if a number i divides n, then $\frac{n}{i}$ also divides n. Therefore, there is no need to check $\frac{n}{i}$ because the maximum value one needs to check is when $\frac{n}{i} = i$. In this case, solving for i yields $\sqrt{n}$. As a result, FastIsPrime checks divisors of n from $i =$ 2 to $\sqrt{n}$.

The difference in methods for checking primality improves the algorithm performance which for IsPrime was $\theta(n)$ and for FastIsPrime is now $\theta(\sqrt{n})$. This improvement takes place because the number of comparisons is reduced from one algorithm to another. For that reason, implementing either of these algorithms with different programming languages will not change the time complexity of either of them since what has changed was only the reduction in comparisons, and O( ) is only a measurement of any algorithm's performance as the input size (n) grows large. Moreover, if FastIsPrime had changed its algorithm design technique rather than reducing the number of comparisons we could actually see an improvement in performance depending on which programming languages support that technique. Evidently, the ones that don't support that specific technique would not benefit from performance improvement in the algorithm.

References:
Pandurangan, Gopal. (2022). *Algorithms: An Intuitive Approach*

4

Show an example of input array and an algorithm version where *Quicksort* has time complexity $\theta(n^2)$.Show an example where *Quicksort* has time complexity O($n$log(n)).

4.1 n^2 Quick Sort

Algorithm 3 Quick Sort has time complexity O(n2)

```
1: function QuickSort(arr)
2:     if length(arr) ≤ 1 then
3:         return arr
4:     end if
5:     pivot ← arr[0]
6:     left ← []
7:     right ← []
8:     for i ← 1 to length(arr) do
9:         if arr[i] < pivot then
10:            append(left, arr[i])
11:        else
12:            append(right, arr[i])
13:        end if
14:    end for
15: end function=0
```

4.2 n^2 Quick Sort Analysis

The first element of the array is always chosen as the pivot. The for loop in the implementation iterates over the remaining elements of the array and puts each element in either the left or right subarray, depending on whether it is less than or greater than the pivot.

This implementation will have time complexity $0(n^2)$ when the input array is already sorted or almost sorted in reverse order. In such a case, the pivot will be the largest element at each partitioning step, resulting in only one element being added to the left subarray and the rest to the right subarray. This leads to each recursive call reducing the size of the subarray by only one, resulting in a time complexity of $0(n^2)$.

4.3 O($nlog(n)$) Quick Sort

Algorithm 4 Quick Sort has time complexity O(nlog(n))

```
if length(arr) ≤ 1 then
    return arr
end if
pivot ← arr[random index between 0 and length(arr) − 1]
left ← []
right ← []
for i ← 0 to length(arr) − 1 do
    if arr[i] < pivot then
        left.append(arr[i])
    else
        right.append(arr[i])
    end if
end for
```

4.4 O($nlog(n)$) Quick Sort Analysis

The pivot is chosen randomly from the array rather than always being the first element. This helps ensure that the pivot is likely to be near the median value of the array, which in turn helps ensure that the recursive calls will be roughly balanced. The implementation has time complexity O(n log(n)) in the average case, because each recursive call partitions the array into two roughly equal parts, and the depth of the recursion is roughly log(n) (where n is the size of the input array). The total time complexity is therefore O(n log(n)). In the worst case, the implementation still has time complexity O(n2), because it is possible for the pivot to be the minimum or maximum element of the array at each partitioning step, leading to only one element being added to the left or right subarray.

5

Write MergeSort in fully recursive functions (no loops). Does $\theta(f(n))$ change? Show derivation of T(n) with recurrences. Write P, Q, for the merge function.

5.1 Algorithm

Algorithm 5 Recursive MergeSort
Input: An unsorted array A with n elements, low, high indices
Output: Sorted array A

```
function recurseMergeSort(A[1...n], low, high)
    if low = high then
        return A[low]
    else if low < high then
        m = (low+high)/2
        A = recurseMergeSort(low, m)
        B = recurseMergeSort(m + 1, high)
        merge(A, B, m, high)
    end if
end function
function merge(A, B, m, high):
    i ← 1; j ← 1; k ← 1

    if i ≤ m & j ≤ high then
        if A[i] < B[j] then
            C[k] ← A[i] + merge(A[i + 1], B[j])
        else
            C[k] ← B[i] + merge(A[i], B[j + 1])
        end if
        k ← k + 1
    end if
    return C
end function
```

5.2 Analysis and Time Complexity

The time complexity for the recursiveMergeSort has not changed since it is still $\theta(nlog(n))$.

That is because the costly operation is the sorting and not the merge. The merge part of the algorithm are only comparison operations and copying elements from two sorted subarrays to a third array C. For the first function, the program recursively divides array A by two on each recursive call and sorting elements once the procedure has finished. The second function takes both sorted subarrays as parameters and only copies elements to a third array. The comparisons made by the second function are proportional to the sum of the sizes of each subarrays A and B. Therefore, if T(n) is the total cost of recursiveMergeSort, the sorting function is T(n) = 2T($\frac{n}{2}$) and merging is T(n) = n.

Moreover,

P = the array A to be sorted contains n (n is at least 1) elements that can be compared and sorted
Q = the program outputs a merged array that is sorted in increasing order

Recurrence relations

$$T(n) = T(\frac{n}{2}) \ + T(\frac{n}{2}) \ + n \tag{2}$$

$$T(n) = 2T(\frac{n}{2}) \ + n \tag{2}$$

Solving recurrences

By analyzing the level-by-level series of the recursion tree formed by this recurrence relation, it shows all terms of the series are equal which implies that $T(n) = \theta(nlog(n))$.

Therefore, recursiveMergeSort has a time complexity of $\theta(nlog(n))$.

References:
`https://www2.cs.uh.edu/~panruowu/2022f_cosc3320/lec2_recursion.pdf`
Pandurangan, Gopal. (2022). *Algorithms: An Intuitive Approach*

6

Write a recursive binary search function that can handle duplicates (if key or other elements are repeated) on array A. It returns the first and last subscript of the key (if found) as a list of 2 elements (they can be returned as an array). Show derivation of T (n) with recurrences, counting number of comparisons. Write P, Q. Notice there is no I because there is no loop.

7

Recursive function to generate all 2^n subsets of a set of n numbers, bottom-up or top down. what is its $\theta(f(n))$?. How can you justify all possible subsets are generated?

7.1 Algorithm

Algorithm 6 PowerSet Backtracking
Input: set A, n elements, and an initially empty set P
Output: subset P containing all power sets of A

```
function powerSet(A, n, P)
    P ← ∅

    if n = 0 then
        return P
    else if n = 1 then
        P.push(n)
        return P
    else
        for i ← 1 to n do
            P.push(i)
            powerSet(A, n, P)
            P.pop( )
            i ← i + 1
        end for
    end if
end function
```

8

You have an array of n=1M elements already sorted. How would you add σ =100k new elements (unsorted) and sort it again? O()? You can assume the array has enough space in main memory. Compute T(σ) where σ is a fraction of n.

Solution

One way to add σ = 100k new elements to a sorted array of n = 1M elements and sort it efficiently without sorting the entire array again is to use a modified form of merge sort.

We can split the sorted array into two halves, and merge the σ new elements with each half separately. This will create two subarrays of size (n+σ)/2, each containing the original sorted elements and half of the new unsorted elements. We can then merge the two subarrays to obtain the final sorted array of size n+σ.

This approach has a time complexity of O(nlogn + σlogn), where the first term represents the time required for merging the original sorted array of n elements, and the second term represents the time required for merging the σ new unsorted elements with the sorted halves of the array.

Alternatively, we can use a variation of insertion sort that takes advantage of the fact that the original array is already sorted. We can insert each of the σ

new unsorted elements into the correct position in the original sorted array by performing binary search to find the appropriate position, and then shifting the existing elements to make room for the new element. This approach has a time complexity of O(σlogn + n), where the first term represents the time required for binary search, and the second term represents the time required for shifting the elements.

9

Backtracking: Problem 5A Jeff's textbook. Describe a recursive backtracking algorithm to determine whether X is a subsequence of Y . For example, the string PPAP is a subsequence of the string PENPINEAPPLEAPPLEPEN.

10

Greedy: Huffman codes for a long string of repeated symbols to compress it. Give an example on a string with length at least 30 and 6 symbols.

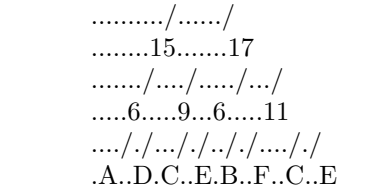
10.1 Solution

Suppose we have a string "AABBBBBBCCCCCCCCCCCCCCDDDDDEEEEEE" of length 30, with 6 letters A, B, C, D, E, and F.

We first count the frequency of each letter: A is 2, B is 6, C is 13, D is 5, E is 6, and F is 0.

Next, we create a binary tree where each leaf node represents a letter and the weight of each leaf node is its frequency. We combine the two smallest weight nodes into a new node with weight equal to the sum of the two nodes. We repeat this process until there is only one node left, which represents the entire string. The result is the following tree.

The "." is used as spacing for drawing the tree in this pdf. It is not part of the tree.



Assign the code '0' to the left branch and '1' to the right branch of each node in the tree, starting from the root. Then the following code is: A is 000, B is 01, C is 11, D is 001, E is 10, and F is none.

AABBBBBBCCCCCCCCCCCCCCDDDDDEEEEEE
000000010101010101010111111111111110000000000010101010

This binary representation has only 58 bits, while the original string had 240 bits. Therefore, Huffman coding has compressed the string to less than 1/4 of its original size.

11

Write a recursive $O(n^2)$ function(s) to sort a doubly linked list. Use next() and prev() to get the next and previous nodes as necessary. Derive T(n). Transform it into two nested loops and show the loop invariant I.

12

Backtracking: Problem 6B Jeff's textbook. Symmetric binary B-trees (Red-black trees). Show a recursive algorithm to insert nodes into the tree, with or without backtracking. Compare O() with AVL trees. Symmetric binary B-trees are another self-balancing binary trees, first described by Rudolf Bayer in 1972; these are better known by the name red-black trees, after a somewhat simpler reformulation by Leo Guibas and Bob Sedgwick in 1978. A red-black tree is a binary search tree with the following additional constraints: Every node is either red or black. Every red node has a black parent. Every root-to-leaf path contains the same number of black nodes. Describe a recursive backtracking algorithm to construct an optimal red-black tree for a given set of search keys and frequencies.

12.1 Solution

If the tree is empty, create a new black node and make it the root. Otherwise, find the appropriate position for the new node as in a regular binary search tree. Insert the new node as a red leaf node. If the new node's parent is black, then we're done.

If the new node's parent is red, then there are two cases. If the parent's sibling is also red, we color the parent and its sibling black, and color the grandparent red. Then we recursively fix the violation at the grandparent. If the parent's sibling is black or null, then we need to do a rotation to bring the parent and new node into a straight line with the grandparent. If the parent and new node are both left children or both right children, we perform a single rotation. If they are on opposite sides, we perform a double rotation. After the rotation, we color the parent black and the grandparent red.

The worst-case time complexity of inserting a node into a red-black tree is O(log n), which is the same as the worst-case time complexity of inserting a node into an AVL tree. However, red-black trees have a lower constant factor than AVL trees, making them more efficient in practice. Additionally, red-black trees are

easier to implement and maintain than AVL trees. Red-black trees tend to have a slightly slower average performance than AVL trees due to the additional color information that needs to be maintained.

13

Is it possible to program MergeSort in a programming language that does not support recursion? Requirements: partition and merging, programming language has functions and arrays, computer has a stack (hint!).

13.1 Solution

Yes. It is possible to implement MergeSort in a programming language that does not support recursion by storing sub-arrays for each partition with the help of the stack. The steps for this non-recursive alternative are:
1. Start to partition the array.
2. Store each sub-array in a stack data structure sorting them as it needed.
3. Merge each sorted sub-array into a new array.

14

$O(n^2 + n + 2^n + log(n) + n!) =?, O(2^n) = O(3^n)?, O((log(n))^2) > O(n)?$

14.1 Solution

$O(n^2 + n + 2^n + log(n) + n!) = O(n!)$ because n! grows faster than any polynomial function (including 2^n).

$O(2^n) = \mathrm{O}(3^n)$ is true because the big O notation is the upper bound of a function, which means it only guarantees to give approximate values for the time complexity of an algorithm. Therefore, $O(2^n)$ and $O(3^n)$ are both equivalent which means that they have the same growth rate.

$O(log(n))^2$ is less than O(n) because the growth rate of the logarithmic function is slower than that of a linear function. Therefore, even if we square the logarithmic function, it will still be smaller than the linear function when n is sufficiently large.

15

Explain all potential algorithms to solve the n-Queens problem, contrasting O() and why they guarantee no queen attacks another queen.

15.1 Solution

The objective of the n-Queens problem is to place n queens on a n x n board such that no two queens attack each other. To solve such a problem, there is a major algorithm design technique that can be used, which is backtracking and the other is an analytical approach described on Gopal's textbook.

Backtracking considers a solution as a sequence of moves. It starts with a partial solution, attempts every configuration (legal move) possible, places each move/configuration in a vector and backtracks to attempt next possible move. By using backtracking to solve the n-queens problem, initially an empty chess board is considered a partial solution. Each queen is then placed on the board one row at a time, starting from the first row, while attempting to place the next nth-queen on the nth-row and ignoring squares that would be attacked by earlier queens. Whenever there are no more legal choices, the algorithm backtracks to a previous legal move and tries different configurations once again. This algorithm is equivalent to a depth-first search traversal of a tree, which starting from a root node branches to every possible configuration until all choices are exhausted.

An analytical approach for this problem is described on Gopal's textbook and consists in analyzing the pattern that exists on the solution when n = 4 and its configuration. There is a pattern on the placements of the queens that is basically placing half of the queens on the right of the diagonal and the other half on the left of the diagonal, thus avoiding the diagonal itself. The solution by this analytical approach works for every n that leaves a remainder 0, 1, 4 and 5, and for the values that do not work it can be extended or adjusted according to the size of n.

The time complexity for the backtracking solution is O($n!$) because there are n choices for the first queen, $n-1$ choices for the second queen on the second row, $n-2$ choices for the third queen on the third row and so on, which results in a total of $n*(n-1)*(n-2)...*2*1 = n!$ possible choices. On the other hand, the time complexity for the analytical approach is O(n), which is linear because there are no costly operations in this algorithm, only analysis and placement of queens which can be done linear depending on the size of n.

The difference between both algorithms is clearly the time complexity due to the fact that bactracking computes every possible solution in an exhaustive manner while the analytical approach copies the pattern that exists in previ-

ous solutions already computed. However, they are similar in that they both guarantee no two queens attack each other because solutions every solution is constructed incrementally one piece at a time, always making sure no constraints are being violated and only legal moves are being added to the full solution.https://www.overleaf.com/project/63e179588cc928291b635666

16

Fibonacci numbers: write a function with forward recursion, going from 1 to n. Compute $\theta(f(n))$. It this smart recursion (i.e. equivalent to dynamic programming)?

16.1 Algorithm

Algorithm 7 Fibonacci numbers

```
function FIBONACCI(n)
    if n == 1 or n == 2 then
        return 1
    else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
    end if
end function
```

16.2 Time Complexity

The time complexity of this function can be expressed as T(n) = T(n-1) + T(n-2) + O(1), since each recursive call requires two additional calls until it reaches the base case, and the constant time operations are negligible. Using the recursive tree method, we can see that the number of recursive calls grows exponentially with n. This is not considered a smart recursion or equivalent to dynamic programming, as it does not utilize any memorization or storage of previously computed values to avoid redundant computations. Dynamic programming involves breaking a problem down into smaller subproblems and solving each subproblem only once, storing the solutions in a table or array for future use. This can significantly reduce the number of redundant computations and improve the time complexity of the algorithm.

17

Based on the definitions, justify why $\theta(2^n) < \theta(4^n)$.

17.1 Solution

Based on the definition of Big-θ, a function $f(n)$ is Big-θ of g(n) if and only if there are constants c1 and c2, and n0, such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$, which means $f(n)$ is bounded above and bounded below by $g(n)$.

In other words, Big-θ is a tight/average bound providing the exact notation and exact time complexity for an algorithm compared to Big-O and Big-ω, which gives upper and lower bounds of a function respectively.

That means, the exact bound for $\theta(4^n)$ grows much faster than $\theta(2^n)$ as the input size n becomes larger. For that reason, $\theta(2^n) < \theta(4^n)$ because $\theta(2^n)$ grows slower and has a better time complexity than $\theta(4^n)$.

Additionally,

$$1 < log(n) < \sqrt{(n)} < n < log(n) < n^2 < n^3 < ... < 2^n < 3^n < ... < n^n \qquad (2)$$

18

Is it possible to program binary search using only linked lists? Is it possible to program merge sort using only linked lists?

18.1 Solution

Yes, it is possible to implement binary search using only linked lists. The basic idea is to traverse the linked list like how we traverse an array, but instead of using an index, we use pointers to the head and tail of the linked list to determine the middle element. Then, we compare the search key with the middle element and repeat the search on the appropriate half of the linked list. This process continues until either the search key is found, or the search range becomes empty. Similarly, it is possible to implement merge sort using only linked lists. The basic idea is to recursively divide the linked list into halves, sort the halves separately, and then merge the sorted halves into a single sorted list. This involves breaking the linked list into two parts, recursively applying merge sort on each part, and then merging the two sorted linked lists. The merge operation involves comparing the elements in the two linked lists and creating a new linked list that contains the elements in sorted order.

19

Show a step by step example aligning two sequences of 3 symbols with a Naive, recursive and smart recursion (dynamic prog.) algorithms from Gopal's textbook (Ch.6).

20

Compare algorithms to solve the n-Queens problem for n = 4..10 with backtracking and the faster solution based on 6t + 1, 6t + 4, 6t + 5 and its generalization to 6t + 2, 6t + 3. Show solutions for n = 9, 10.

20.1 Solution

The n-Queens problem is the task of placing n queens on an nxn chessboard so that no two queens attack each other. There are several algorithms to solve the n-Queens problem, but we'll compare backtracking and the faster solution based on 6t + 1, 6t + 4, 6t + 5 and its generalization to 6t + 2, 6t + 3.

For n = 9, the backtracking algorithm requires searching through 9! (362,880) possible solutions. This takes a significant amount of time and is not practical for larger values of n. On the other hand, the faster solution based on 6t + 1, 6t + 4, 6t + 5 and its generalization to 6t + 2, 6t + 3 can solve the n-Queens problem for n = 9 almost instantaneously.

For n = 10, the backtracking algorithm requires searching through 10! (3,628,800) possible solutions, which is significantly larger than the previous case. The time required to find a solution using backtracking becomes prohibitive for larger values of n. The faster solution based on 6t + 1, 6t + 4, 6t + 5 and its generalization to 6t + 2, 6t + 3 can solve the n-Queens problem for n = 10 in a reasonable amount of time.

In summary, the backtracking algorithm can solve the n-Queens problem for small values of n, but its performance degrades rapidly as n increases. The faster solution based on 6t + 1, 6t + 4, 6t + 5 and its generalization to 6t + 2, 6t + 3 can solve the n-Queens problem for larger values of n in a reasonable amount of time.