Compsci 330 Design and Analysis of Algorithms
Assignment 1, Spring 2024 Duke University

Example Solution

Due Date: Thursday, January 25, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Asymptotics). For each of the following statements, decide whether the statement is true or false, and briefly explain your reasoning.

(a) $3n$ is $O(n)$.

(b) $n^2 + 2n$ is $\Omega(n)$.

(c) $n^2 + 2n$ is $\Theta(n)$.

(d) $n^2 \log(n)$ is $O(n^3)$.

(e) $n \log(n^3)$ is $\Theta(n \log(n))$.

(f) 2^n is $\Theta(n^2)$.

(g) If Algorithm A has runtime complexity $\Theta(n)$ and Algorithm B has runtime complexity $\Theta(n)$, then the empirical runtime of the algorithms will be approximately equal for a given input size n.

(h) If Algorithm A has runtime complexity $\Theta(n^2)$, then doubling the input size should roughly double the empirical runtime of the algorithm for large input sizes n.

Solution 1.

(a) **True**, $3n$ is $O(n)$. We drop constant terms in asymptotic notation, noting that 3 suffices as a coefficient in the definition of O notation for the statement.

(b) **True**, $n^2 + 2n$ is $\Omega(n)$. Note that Ω is the asymptotic *lower bound* (as compared to the asymptotic upper bound using O notation). The lowest order term is the linear $2n$.

(c) **False**, $n^2 + 2n$ is **not** $\Theta(n)$. Note for the statement to be true, $n^2 + 2n$ would have to be **both** $\Omega(n)$ and $O(n)$; that is, would have to be asymptotically lower and upper bounded by linear functions. While the lower bound holds, the quadratic term n^2 is *not* asymptotically upper bounded by a linear function.

(d) **True**, $n^2 \log(n)$ is $O(n^3)$. This follows from observing that $\log(n) \leq n$ for all positive n.

(e) **True**, $n \log(n^3)$ is $\Theta(n \log(n))$. This follows from observing that $\log(n^3) = 3 \log(n)$.

(f) **False**, 2^n is **not** $\Theta(n^2)$. In particular, the exponential function 2^n is not $O(n^2)$.

(g) **False**, the empirical runtimes will **not** necessarily be approximately equal. They will exhibit the same growth, but may have different constant factor coefficients. For example, it is possible that Algorithm A is one million times faster than Algorithm B at all input sizes.

(h) **False**, doubling the input size should **more than** roughly double the empirical runtime of the algorithm for large input sizes n. In particular, we would expect the runtime on an input of size $2n$ scale as $(2n)^2 = 4n^2$ so roughly 4 times that as with an input of size n. It is important that the input size is large to ensure that the quadratic term is dominating over possible lower linear or constant terms.

Problem 2 (Iteration and Induction). Consider an array A storing n integers. Consider the following algorithm for sorting from least to greatest. Note that it sorts the same array passed as input in place; there is no return value.

```
1: procedure SORT(A)
2:     for i = 0 to n − 2 do
3:         m = i
4:         for j = i + 1 to n − 1 do
5:             if A[j] < A[m] then
6:                 m = j
7:         temp = A[i]
8:         A[i] = A[m]
9:         A[m] = temp
```

(a) Derive the asymptotic runtime complexity of the SORT algorithm as a function of n. Briefly explain your answer.

(b) Prove that the SORT procedure is correct using mathematical induction. You may assume that for a given iteration i of the outer loop, lines $3-9$ swap the minimum value from $A[i], \ldots, A[n-1]$ with $A[i]$.

Solution 2.

(a) The asymptotic runtime complexity is $O(n^2)$. All individual operations are constant time so we just need to count the total number of iterations. On iteration i of the outer for loop, the inner for loop runs $n-i-1$ iterations. So in total there are $(n-1)+(n-2)+\cdots+1 = \frac{(n-1)n}{2}$ is $O(n^2)$ iterations.

(b) We will argue by induction that after iteration i, the elements $A[0], \ldots, A[i]$ are the $i+1$ smallest elements of A in sorted order.

- *Base Case.* After iteration $i = 0$, the single minimum element of the entire array is swapped to $A[0]$.
- *Inductive Hypothesis (IH).* Suppose that after iteration $i-1$, the elements $A[0], \ldots, A[i-1]$ are the i smallest elements of A in sorted order.
- *Inductive Step.* Since $A[0], \ldots, A[i-1]$ are the i smallest elements of A by the IH, the $i+1$ smallest element of A is the smallest element among $A[i], \ldots, A[n-1]$. That element is swapped to position i during the iteration i, at which point $A[0], \ldots, A[i]$ are the $i+1$ smallest elements of A in sorted order.

Problem 3 (Tree Recursion and Induction). Consider a binary search tree T storing n integers. Each node r has attributes r.value storing the integer element at that node, as well as references r.left and r.right to the left and right child nodes in the tree respectively. If there is no child node, the reference will be NULL.

(a) Given integers a and b where $a < b$, write an efficient recursive algorithm that returns the number of elements of T between a and b (inclusive).

(b) Use induction to prove that your algorithm is correct. Consider using the size of a tree for your induction variable, and note that a subtree of a binary search tree is itself a binary search tree.

(c) Derive the worst-case asymptotic runtime complexity of your algorithm as a function of n, the size of T. Make no assumptions about the shape of the binary search tree or the number of elements of T between a and b.

(d) Let m be the number of elements in T between a and b (inclusive). Suppose that T is perfect, that is, every internal node has 2 children and the leaf nodes are all at the same depth (intuitively, a perfectly balanced tree). Derive the runtime complexity of your algorithm as a function of n and m. For an efficient algorithm, this runtime should be smaller than in the previous step when m is much smaller than n.

Solution 3.

(a) Our algorithm operations by checking whether the root of a subtree is in the range between a and b. If so, the element is counted and added to the number of elements in the range in the left and right subtrees. Otherwise, only the subtree that could include elements from the range is recursively searched. We define the procedure details below.

```
1: procedure RANGE(r, a, b)
2:     if r is NULL then
3:         return 0
4:     else if r.value < a then
5:         return RANGE(r.right, a, b)
6:     else if r.value > b then
7:         return RANGE(r.left, a, b)
8:     return 1 + RANGE(r.left, a, b) + RANGE(r.right, a, b)
```

(b) We want to prove that Range(r, a, b) returns the number of elements between a and b (inclusive) in a binary search tree T rooted at r. We will prove this by induction on n, the number of nodes in the tree rooted at r.

For the base case, consider a tree of size 1 (r is the only node). Any recursive calls on r.left or r.right return 0 (lines 1-2). If r.value is outside of the range, then we return 0. If r.value is inside of the range, then we will return 1+ Range(r.left, a, b) + Range(r.right, a, b) = 1.

For the inductive hypothesis (IH): Suppose Range(r, a, b) returns the correct number of elements between a and b (inclusive) in all binary search trees of size at most $n-1$. Consider a binary search tree T rooted at r of size n. There are three cases:

- If r.value $< a$, then all values in the subtree rooted at r.left are also less than a, by virtue of the binary search tree property. Therefore, all elements in the range are in the

subtree rooted at r.right. Since that subtree has at most $n-1$ elements, Range(r.right, a, b) returns the correct value by the IH.

- The r.value $> b$ case is symmetric with the first case: all elements in the range must be in the left subtree.
- If r.value is between a and b inclusive, then the number of total elements in the range is 1 (counting r itself) plus the number in the subtree rooted at r.left, plus the number in the subtree rooted at r.right. As each of of these subtrees has size at most $n-1$, Range(r.left, a, b) and Range(r.right, a, b) return the correct values by the IH.

(c) The runtime complexity is $O(n)$ because, even in the worst case, each of the n nodes are visited (or recursed on) at most once, and all other operations have constant runtime complexity.

(d) The runtime complexity is $O(m + \log(n))$, which is much smaller than $O(n)$ when $m << n$.

It is clear from the correctness that the algorithm visits every node in the range from a to b, thus the linear term m. It remains to argue that the algorithm visits only $O(\log(n))$ nodes that are are *not* in the range from a to b.

Let $l(n)$ be the number of nodes with values less than a visited by the algorithm called on the root of a tree with n nodes. Note that in any case of the algorithm, we make at most one recursive call on a subtree that may contain values less than a, and a subtree has at most $n/2$ nodes since the tree is perfect / balanced. We can therefore write the recurrence $l(n) \leq l(n/2) + c$ for some constant c, the solution to which is $l(n)$ is $O(\log(n))$. The same argument can be made for the number of nodes we visit with values greater than b.

Problem 4 (Hash Table and Probability). Consider a hash table T with m positions indexed from 0 to $m-1$ and a hash function $h(x) = |x| \mod m$. The hash table stores an element x in position $h(x)$ and resolves collisions by linear chaining (that is, by storing a linked list at each position for all elements mapped to that index). You also have an array A of n integers, not necessarily unique. Consider the following algorithm for counting the number of unique integers in A using T.

```
1: procedure CountUnique(A, T)
2:     c = 0;
3:     for i = 0 to n − 1 do
4:         x = A[i]
5:         if x ∉ T[h(x)] then
6:             c = c + 1
7:             Add x to T[h(x)]
8:     return c
```

(a) What is the *worst-case* asymptotic runtime complexity of the CountUnique algorithm? Do not make any assumptions about the values in A. Briefly explain your answer.

(b) Suppose that A consists of n integers drawn independently and uniformly at random from $\{0, 1, \ldots, km-1\}$ for some integer $k > 0$. For a given index $0 \le i \le m-1$, what is the probability that *none* of the n elements of A are hashed to $T[i]$? What is the probability that *all* of the the n elements of A are hashed to $T[i]$? Briefly explain your answers.

(c) Again suppose that A consists of n integers drawn independently and uniformly at random from $\{0, 1, \ldots, km-1\}$ for some integer $k > 0$. For a given index $0 \le i \le m-1$, what is the *expected* number of elements of A (counting possible duplicates) that hash to $T[i]$? Based on your answer, derive the expected asymptotic runtime complexity of the CountUnique(p)rocedure as a function of n and/or m, under this random input assumption.

Solution 4.

(a) $\Theta(n^2)$. For example, suppose A contains the elements $0, m, 2m, \ldots, (n-1)m$. Then $h(x) = 0$ for all $x \in A$. So all elements will be hashed to the linked list at position 0 in the hash table. All elements are unique as well, so will be added to the list. The runtime complexity to search a linked list is linear in the size of the list, so there will be n searches with total complexity proportional to $1 + 2 + 3 + \cdots + n$ which is $\Theta(n^2)$.

(b) The probability that none hash to i is $(1 - 1/m)^n$. Let $X_{i,j}$ be an indicator random variable equal to 1 if the j'th element of A hashes to index i, and 0 otherwise. Then $\Pr(X_{i,j} = 1) = 1/m$ because the j'th element is drawn uniformly at random. Then the probability that *none* of the elements hash to a position i is $\Pr(X_{i,0} = 0 \wedge X_{i,1} = 0 \wedge \cdots \wedge X_{i,n-1} = 0)$. Since the elements are drawn independently, this is $(1 - 1/m)^n$.

The probability that all hash to i is $(1/m)^n$, given by $\Pr(X_{i,0} = 1 \wedge X_{i,1} = 1 \wedge \cdots \wedge X_{i,n-1} = 1) = (1/m)^n$.

(c) The expected runtime under the random input assumption is $O\left(n + \frac{n^2}{m}\right)$.

Define $X_{i,j}$ as in the previous part. Then the expected number of elements of A that hash to

$T[i]$ is

$$\mathbb{E}[\sum_{j=0}^{n-1} X_{i,j} = \sum_{j=0}^{n-1} \mathbb{E}[X_{i,j}] = \frac{n}{m}.$$

There are n iterations to the algorithm and the runtime of a given iteration is some constant (call it c) plus time proportional to the length of the linked list to search at a given position in the hash table. Under the random input assumption, the expected length after i iterations is i/m, so the total expected runtime is proportional to

$$\left(c + \frac{1}{m}\right) + \left(c + \frac{2}{m}\right) + \cdots + \left(c + \frac{n-1}{m}\right) \text{ is } O\left(n + \frac{n^2}{m}\right).$$

Problem 5 (Applied Problem). Given a sorted array A of size N ($1 \leq N \leq 5 \cdot 10^5$), and two numbers x, y ($0 \leq x \leq y \leq N$), determine the number of elements of A in the range $[x, y]$ (inclusive). Language-specific details follow. You can use whichever of Python or Java you prefer.

Your solution's efficiency will be checked by comparing if its empirical runtime is within constant factors of an $O(\log(N))$ reference solution. Your submission will be automatically graded against test cases for correctness and efficiency. You will be able to see your score and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `range.py` to the Gradescope item "Homework 1 (Python)." The file should define (at least) a top level function `range_count` that looks like:
  - `def range_count(A:[int], x:int, y:int)`

  and returns the number of elements of `A` in the range `[x,y]`
- **Java.** You should submit a file called `Range.java` to the Gradescope item "Homework 1 (Java)." The file should define (at least) a top level function `rangeCount` that looks like:
  - `public int rangeCount(int[] A, int x, int y)`

  where `A` is a sorted `int[]` and `[x,y]` is the range of our query. Return the number of elements of `A` in the range `[x,y]`

Solution 4. We do not provide full code solutions. The idea for this problem is as follows: Binary search for the *first* index whose value lies in the range (call this i_f, as well as the *last* index whose value lies in this range (call this i_l). Be careful to check the case where there are no elements in the range. Otherwise, return $i_l - i_f + 1$.

The standard implementation of binary search usually just guarantees to return *any* index, so the algorithm will require some modification. In particular, one cannot necessarily return immediately when a valid index is located, but must continue searching to see if there is a smaller or larger valid index.