UNSW SYDNEY

2. PRELIMINARIES

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2023

Table of Contents

- You should be familiar with true-ish statements such as:

  Heap sort is faster than bubble sort.

  Linear search is slower than binary search.

- We would like to make such statements more precise.

- We also want to understand when they are wrong, and why it matters.

- The statements on the previous slide are most commonly made in comparing the *worst case* performance of these algorithms.
- They are actually false in the best case!
- In most problems in this course, we are concerned with worst case performance, so as to be robust to maliciously created (or simply unlucky) instances.
- We will hardly ever discuss the best case.

- In some circumstances, one might accept occasional poor performance as a trade-off for good average performance over all possible inputs (or a large sample of them).

- Analysing the *average case* or *expected* performance of an algorithm requires probabilistic methods and is beyond the scope of this course, *except* where we rely purely on results of this type from prior courses (e.g. hash table operations, quicksort).

- We need a way to compare two functions, in this case representing the worst case runtime of each algorithm.
- A natural starting point is to compare function values directly.

Question

If $f(100) > g(100)$, then does f represent a greater running time, i.e. a slower algorithm?

Answer

Not necessarily! $n = 100$ might be an outlier, or too small to appreciate the efficiencies of algorithm f. We care more about which algorithm *scales better*.

- We prefer to talk in terms of asymptotics, i.e. long-run behaviour.
  - For example, if the size of the input doubles, the function value could (approximately) double, quadruple, etc.
  - A function which quadruples will eventually exceed a function which doubles, regardless of the values for small inputs.
  - We'd like to categorise functions (i.e. runtimes, and therefore algorithms) by their *asymptotic* rate of growth.
- We'll primarily talk about the worst-case performance of algorithms, but the same method of analysis could be applied to the best case or average case performance of an algorithm.

Definition

We say $f(n) = O(g(n))$ if for *large enough* n, $f(n)$ is *at most* a constant multiple of $g(n)$.

- $g(n)$ is said to be an *asymptotic upper bound* for $f(n)$.
- This means that the rate of growth of function f is no greater than that of function g.
- An algorithm whose running time is $f(n)$ *scales at least as well as* one whose running time is $g(n)$.
- It's true-ish to say that the former algorithm is 'at least as fast as' the latter.

- Useful to (over-)estimate the complexity of a particular algorithm.

- For uncomplicated functions, such as those that typically arise as the running time of an algorithm, we usually only have to consider the dominant term.

Example 1

Let $f(n) = 100n$. Then $f(n) = O(n)$, because $f(n)$ is at most 100 times n for large n.

Example 2

Let $f(n) = 2n + 7$. Then $f(n) = O(n^2)$, because $f(n)$ is at most 1 times n^2 for large n. Note that $f(n) = O(n)$ is also true in this case.

Example 3

Let $f(n) = 0.001n^3$. Then $f(n) \neq O(n^2)$, because for any constant multiple of n^2, $f(n)$ will eventually exceed it.

Recall from prior courses that:

- inserting into a binary search tree takes $O(h)$ time in the worst case, where h is the height of the tree, and
- insertion sort runs in $O(n^2)$ time in the worst case, but $O(n)$ in the best case.

Note that these statements are true regardless of:

- the details of how the algorithm is implemented, and
- the hardware used to execute the algorithm.

Both of these issues change the actual running time, but the dominant term of the running time will still be a constant times h, n^2 or n respectively.

Question

The definition states that we only care about the function values for "large enough" n. How large is "large enough"?

Put differently, how small is "small enough" that it can be safely ignored in assessing whether the definition is satisfied?

Answer

Everything is small compared to the infinity of n-values beyond itself.

It doesn't matter how $f(1)$ compares to $g(1)$, or even how $f(1,000,000)$ compares to $g(1,000,000)$. We only care that $f(n)$ is bounded above by a multiple of $g(n)$ *eventually*.

Disclaimer

Of course, how $f(1,000,000)$ compares to $g(1,000,000)$ is also important.

When choosing algorithms for a particular application with inputs of size at most $1,000,000$, the behaviour beyond this size doesn't matter at all!

Asymptotic analysis is *one* tool by which to compare algorithms. It is not the only consideration; the actual input sizes used and the constant factors hidden by the $O(\cdot)$ should also be taken into account.

Definition

We say $f(n) = \Omega(g(n))$ if for *large enough* n, $f(n)$ is *at least* a constant multiple of $g(n)$.

- $g(n)$ is said to be an *asymptotic lower bound* for $f(n)$.
- This means that the rate of growth of function f is no less than that of function g.
- An algorithm whose running time is $f(n)$ *scales at least as badly as* one whose running time is $g(n)$.
- It's true-ish to say that the former algorithm is 'no faster than' the latter.

Big-Omega notation

- Useful to say that any algorithm solving a particular problem runs in at least $\Omega(g(n))$.
- For example, finding the maximum element of an unsorted array takes $\Omega(n)$ time, because you must consider every element.
- Once again, for every function that we care about, only the dominant term will be relevant.

Challenge

We didn't present the formal definitions of $O(\cdot)$ and $\Omega(\cdot)$, but they can be found in either textbook.

Using these formal definitions, prove that if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$.

Definition

We say $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

- $f(n)$ and $g(n)$ are said to have the same asymptotic growth rate.
- An algorithm whose running time is $f(n)$ *scales as well as* one whose running time is $g(n)$.
- It's true-ish to say that the former algorithm is 'as fast as' the latter.

Question

You've previously seen statements such as

bubble sort runs in $O(n^2)$ time in the worst case.

Should these statements be written using $\Theta(\cdot)$ instead of $O(\cdot)$?

Answer

They can, but they don't have to be. The statements

bubble sort runs in $O(n^2)$ time in the worst case

and

bubble sort runs in $\Theta(n^2)$ time in the worst case

are both true: they claim that the worst case running time is *at most* quadratic and *exactly* quadratic respectively.

Answer (continued)

The $\Theta(\cdot)$ statement conveys slightly more information than the $O(\cdot)$ statement. However in most situations we just want to be sure that the running time hasn't been *under*estimated, so $O(\cdot)$ is the important part.

Adding to the confusion, some people write $O(\cdot)$ as a shorthand for $\Theta(\cdot)$! Please be aware of the difference, even though it's often inconsequential.

Fact

If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$.

- The last term $O(g_1 + g_2)$ is often rewritten as $O\left(\max(g_1, g_2)\right)$, since $g_1 + g_2 \leq 2\max(g_1, g_2)$.

- The same property applies if O is replaced by Ω or Θ.

- This property justifies ignoring non-dominant terms:if f_2 has a lower asymptotic bound than f_1, then the bound on f_1 also applies to $f_1 + f_2$.
- For example, if f_2 is linear but f_1 is quadratic, then $f_1 + f_2$ is also quadratic.
- This is useful for analysing algorithms that have two or more stages executed sequentially.
- If f_1 is the running time of the first stage and f_2 of the second stage, then we can bound each stage and add the bounds, *or* simply take the most 'expensive' stage.

Fact

If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.

- In particular, if $f = O(g)$ and λ is a constant (i.e. $\lambda = O(1)$), then $\lambda \cdot f = O(g)$ also.
- The same property applies if O is replaced by Ω or Θ.
- This is useful for analysing algorithms that have two or more nested parts.
- If each execution of the inner part takes f_2 time, and it is executed f_1 many times, then we can bound each part and multiply the bounds.

Table of Contents

Definition

For $a, b > 0$ and $a \neq 1$, let $n = \log_a b$ if $a^n = b$.

Properties

$$a^{\log_a n} = n$$

$$\log_a(mn) = \log_a m + \log_a n$$

$$\log_a(n^k) = k \log_a n$$

Theorem

For $a, b, x > 0$ and $a, b \neq 1$, we have

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

- The denominator is constant with respect to x!
- Therefore $\log_a n = \Theta(\log_b n)$, that is, logarithms of any base are interchangeable in asymptotic notation.
- We typically write $\log n$ instead, suppressing the base.

Table of Contents

- Store *values* indexed by *keys*.
- Hash function maps keys to indices in a fixed size table.
- Ideally no two keys map to the same index, but it's impossible to guarantee this.
- A situation where two (or more) keys have the same hash value is called a *collision*.
- There are several ways to resolve collisions – for example, separate chaining stores a linked list of all colliding key-value pairs at each index of the hash table.

Operations (expected)

- Search for the value associated to a given key: $O(1)$
- Update the value associated to a given key: $O(1)$
- Insert/delete: $O(1)$

Operations (worst case)

- Search for the value associated to a given key: $O(n)$
- Update the value associated to a given key: $O(n)$
- Insert/delete: $O(n)$

- Store (comparable) keys or key-value pairs in a binary tree, where each node has at most two children, designated as *left* and *right*
- Each node's key compares greater than all keys in its left subtree, and less than all keys in its right subtree.

Operations

Let h be the height of the tree, that is, the length of the longest path from the root to the leaf.

- Search: $O(h)$
- Insert/delete: $O(h)$

- In the best case, $h \approx \log_2 n$. Such trees are said to be *balanced*.
- In the worst case, $h \approx n$, e.g. if keys were inserted in increasing order.
- Fortunately, there are several ways to make a *self-balancing* binary search tree (e.g. B-tree, AVL tree, red-black tree).
- Each of these performs rotations to maintain certain invariants, in order to guarantee that $h = O(\log n)$ and therefore all tree operations run in $O(\log n)$.
- Red-black trees are detailed in CLRS, but in this course it is sufficient to write "self-balancing binary search tree" without specifying any particular scheme.

(Binary) Heaps

- Store items in a *complete* binary tree, with every parent comparing $\geq$ all its children.
- This is a max heap; replace $\geq$ with $\leq$ for min heap.
- Used to implement priority queue.

Operations

- Build heap: $O(n)$
- Find maximum: $O(1)$
- Delete maximum: $O(\log n)$
- Insert: $O(\log n)$

Table of Contents

Problem

You are given a *sorted* array A of n integers. Determine whether a value x appears in the array.

Observation

If $A[i] < x$, then for all $j < i$, it is also true that $A[j] < x$.

Algorithm

We recursively apply the following algorithm to the subarray $A[\ell..r]$ in order to search for x.

If the array has no elements (i.e. r is not $\geq \ell$), then x cannot be found.

Otherwise:

- first define $m = \left\lfloor \frac{\ell+r}{2} \right\rfloor$, the midpoint of the subarray,
- then, we take cases on the value at the midpoint, namely $A[m]$.
  - If $A[m] = x$, we have found an occurrence of x and we can exit the recursion.
  - If $A[m] > x$, we recurse on the left subarray $A[\ell..m-1]$.
  - If $A[m] < x$, we recurse on the right subarray $A[m+1..r]$.

Complexity

Worst case: $O(\log n)$. At each step, the search space halves, which can only happen $\log_2 n$ times.

Small modifications allow us to solve related search problems:

- Find the smallest index i such that $A[i] \geq x$, etc.
- Find the range of indices $\ell..r$ such that $A[\ell] = \ldots = A[r] = x$.

- Decision problems are of the form

  Given some parameters including X, can you ...

- Optimisation problems are of the form

  What is the smallest X for which you can ...

- An optimisation problem is typically *much* harder than the corresponding decision problem, because there are many more choices.

- Can we reduce (some) optimisation problems to decision problems?

- For simplicity, we'll assume that X can only be an integer. Similar ideas work for real X.
- Let $f(x)$ be the outcome of the decision problem when $X = x$, with 0 for false and 1 for true.
- In some (but not all) such problems, if the condition holds with $X = x$ then it also holds with $X = x + 1$.
- Thus f is all 0's up to the first 1, after which it is all 1's. So we can use binary search!

- This technique of binary searching the answer, that is, finding the smallest X such that $f(X) = 1$ using binary search, is often called *discrete* binary search.
- Overhead is just a factor of $O(\log A)$ where A is the range of possible answers.
- Question 2 of Tutorial 1 is an example of this technique.

Table of Contents

Problem

You are given an array A consisting of n items. The following operations each take constant time:

- read from any index
- write to any index
- compare two items.

Design an efficient algorithm to sort the items.

Bubble sort, selection sort and insertion sort all take $O(n^2)$ time in the worst case.

Algorithm

1. If $n = 1$, do nothing. Otherwise, let $m = \lfloor (n+1)/2 \rfloor$.
2. Apply merge sort recursively to $A[1..m]$ and $A[m+1..n]$.
3. Merge $A[1..m]$ and $A[m+1..n]$ into $A[1..n]$.

Correctness

Discussed in the previous lecture.

Complexity

- Best case: $O(n \log n)$.
- Worst case: $O(n \log n)$.
- Space: $O(n)$.

Conclusions

- Reliably fast for large arrays.
- Space requirement is a drawback.
- Useful in some circumstances:
  - if a stable sort is required
  - when sorting a linked list
  - with parallelisation.

Algorithm

1. Construct a min heap from the elements of A.
2. Write the top element of the heap to $A[1]$ and pop it from the heap.
3. Repeat the previous step until the heap is empty.

Correctness

Obvious, since the top element of the heap is always the smallest remaining.

Complexity

- Best case: $O(n \log n)$.
- Worst case: $O(n \log n)$.
- Selection sort but with selection in $O(\log n)$ rather than $O(n)$.

Conclusions

- Reliably fast for large arrays.
- No additional space required.
- Constant factor is larger than other fast sorts, so used less in practice.

Algorithm

1. Designate the first element as the *pivot*.
2. Rearrange the array so that all smaller elements are to the left of the pivot, and all larger elements to its right.
3. Recurse on the subarrays left and right of the pivot.

Correctness

Obvious.

Complexity

- The second step (rearranging and partitioning the array) can be done in $O(n)$ without using additional memory (how?).
- However, the performance still greatly depends on the pivot used.
  - In the best case, the pivot is always the median, so the subarrays each have size about $n/2$; like mergesort, this is $O(n \log n)$.
  - In the worst case, the pivot is always the minimum (or maximum), so one subarray is empty and the other has size $n - 1$; like selection sort, this is $O(n^2)$.

Complexity (continued)

- Fortunately, the worst case is rare.
- The average case runtime is $O(n \log n)$.
- Any element could be the pivot! Better pivot selection strategies include:
  - select an array element at random
  - 'median-of-three': among the first, middle and last elements, select the median
  - 'median-of-medians': more on this next week.

Conclusions

- In this course, we prefer mergesort or heapsort for their worst case time complexity.
- However, quicksort is widely used in practice, because the worst case is so rare and the constant factor is small.
- Quicksort forms the basis of the default sort in many programming languages.

- We have seen three sorting algorithms which achieve $O(n \log n)$ time complexity for all or almost all inputs.
- In this course, we are not concerned by the extra space used by merge sort, or the larger constant factor of heap sort.
- However, unless explicitly directed otherwise, we always consider worst case performance, so quicksort's $O(n^2)$ case is unacceptable.
- When designing algorithms in this course which use sorting by comparison, you can simply say 'sort the array using merge sort' or 'using heapsort'.
- However, the other algorithms are useful to understand conceptually and occasionally find some practical application.

Question

Is it possible to design a comparison sort which is asymptotically faster than merge sort in the worst case?

Answer

No!

Claim

Any comparison sort must perform $\Omega(n \log n)$ comparisons in the worst case.

Proof

- There are $n!$ permutations of the array.
- In the worst case, only one of these is the correct sorted order. Our sorting algorithm must find which permutation this is.
- An algorithm which performs k comparisons can get 2^k different combinations of results from these comparisons, and therefore can distinguish between at most 2^k permutations.

Proof (continued)

- We need to perform number of comparisons k such that $2^k \geq n!$, i.e. $k \geq \log_2 n!$.
- Now, $n! = 1 \times 2 \times \ldots \times n$. At least $n/2$ terms of this product are greater than $n/2$, so $n! > \left(\frac{n}{2}\right)^{n/2}$.
- Therefore $k > \frac{n}{2} \log_2 \frac{n}{2}$.
- We can conclude that $k = \Omega(n \log n)$, i.e. any comparison sort performs $\Omega(n \log n)$ comparisons in the worst case.
- So merge sort and heap sort are asymptotically optimal!

- Not all sorting algorithms are based on comparison.
- If we know some information about the items to be sorted, we might be able to design a more specialised algorithm.
- In particular, there are other ways to sort integers.

Problem

You are given an array A consisting of n *integers*, each between 1 and k.

Design an efficient algorithm to sort the integers.

Algorithm

1. Create another array B of size k to store the count of each value, initially all zeros.
2. Iterate through A. At each index i, record one more instance of the value $A[i]$ by incrementing $B[A[i]]$.
3. Write $B[1]$ many ones, then $B[2]$ many twos, etc. into A, from left to right.

Correctness

The final array A is clearly sorted, and it has the same number of occurrences of each value as the original array had, so this algorithm is correct.

Complexity

- Initialising B takes $O(k)$ time.
- Iterating through A takes $O(n)$ time.
- The final step takes $O(\max(n, k))$ time, which is typically written as $O(n + k)$.
- In total the time complexity is $O(n + k)$.
- $O(k)$ additional space is also needed.

Conclusions

- Useful for this particular problem, if the additional space is available.
- Frequency table is a fruitful idea in many contexts.

Question

Does this contradict the earlier $\Omega(n \log n)$ lower bound?

Answer

No! That bound applies only to *comparison sorts*.

Algorithm

1. Distribute the items into buckets $1, \ldots, k$.
2. Sort within each bucket.
3. Concatenate the sorted buckets.

- Not really a sorting algorithm, but rather a template from which we can build sorting algorithms.
- Correctness follows from any item in an earlier bucket comparing $\leq$ any item in a later bucket.

Efficiency

- Depends on several factors:
  - number of buckets used,
  - distribution of items among buckets, and
  - sorting algorithm within buckets.
- Best case has approx n/k items in each of k buckets; worst case has all n items in the same bucket.

Conclusions

- Useful for sorting data that can be easily bucketed with roughly uniform distribution of items to buckets. Examples include:
  - strings, bucketed by first character
  - m-digit integers, bucketed by most significant digit.

Question

Can we sort each bucket using bucket sort?

Answer

Yes, this is MSD (most significant digit) radix sort.

Problem

You are given an array A consisting of n *keys*, each consisting of k symbols.

Design an efficient algorithm to sort the keys lexicographically.

Algorithm

Bucket the keys by their first symbol, and recursively apply the same algorithm to each bucket.

Correctness

Obvious.

Complexity

There are k levels of recursion. In each level, there are a total of n keys to be bucketed, each in constant time. So the total time complexity is $O(nk)$.

Conclusions

- Useful for sorting fixed-length keys, e.g. k-digit or k-bit integers, k-letter words, dates and times.
- Many intermediate buckets to keep track of, and not necessarily stable.

Algorithm

- Sort all keys by their *last* symbol, then sort all keys by their *second last* symbol, and so on.
- The sorting algorithm used in each step must be *stable*, e.g. make buckets and concatenate.

Correctness

Left as an exercise.

Hint: consider two keys which have their first j symbols in common, and differ on the $(j+1)$th symbol.

Hint: stability is important!

Complexity

- Time complexity is again $O(nk)$.
- Space complexity is only $O(n + k)$; no intermediate buckets!

Conclusions

- Stable, space-efficient version of radix sort.
- Same applications: fixed-length keys such as integers and words.

Table of Contents

Definition

A graph is a pair (V, E), where V is the *vertex set* and E is the *edge set*, where each edge connects a pair of vertices.

Variants

- Graphs can be undirected, with edges $\{u, v\}$, or directed, with edges (u, v).
- Graphs can be weighted, where each edge e has an associated weight $w(e)$, or unweighted.

- Instead of $|V|$ and $|E|$, we often simply write V and E for the number of vertices and number of edges.

- A *simple* graph does not have:
  - self-loops, i.e. edges from v to itself, or
  - parallel edges, i.e. multiple identical edges.

  All graphs in this course are simple unless specified otherwise.

- The *degree* of a vertex is the number of edges incident to v.
  - Each vertex of a directed graph has an in-degree and an out-degree.

- Two vertices joined by an edge are said to be *adjacent* or *neighbours*.

- For each vertex v, store a list of edges from v.
- $O(V + E)$ memory

Operations

- Test for edge from v to u: $O(\deg(v))$
- Iterate over neighbours of v: $O(\deg(v))$

Preferable for sparse graphs (few edges per vertex).

- Store a matrix, where each cell stores information about the edge from u to v or lack thereof.
- $O(V^2)$ memory

Operations

- Test for edge from u to v: $O(1)$
- Iterate over neighbours of v: $O(V)$

Preferable for dense graphs (many edges per vertex).

- How to visit all vertices of a graph?
- Two main approaches: depth-first or breadth-first.
- We'll consider undirected graphs, but these methods can be extended to directed graphs also.

From a vertex v:

- mark v as visited, and
- recurse on each unvisited neighbour of v

Time complexity is $O(V + E)$, using a stack.

Many applications, some covered in future weeks.

From a vertex v:

- mark v as visited,
- mark each unvisited neighbour of v as visited,
- mark each of their unvisited neighbours as visited, etc.

Time complexity is $O(V + E)$, using a queue.

Finds shortest paths in unweighted graphs.

Definitions

- An undirected graph is *connected* if every pair of vertices can reach each other by a sequence of one or more edges.
- A *tree* is a connected graph in which:
  - there is a *unique* simple path between every pair of vertices, or
  - there is one fewer edge than the number of vertices, or
  - there are no cycles, or
  - the removal of any edge disconnects the graph.

These definitions are all equivalent!

Table of Contents

On a circular highway there are n petrol stations, unevenly spaced, each containing a different quantity of petrol. It is known that the total quantity of petrol on all stations is enough to go around the highway once, and that the tank of your car can hold enough fuel to make a trip around the highway.

Prove that there always exists a station among all of the stations on the highway, such that if you take it as a starting point and take the fuel from that station, you can continue to make a complete round trip around the highway, never emptying your tank before reaching the next station to refuel.

That's All, Folks!!