2. REVISION

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

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2022

Table of Contents

Fast and slow algorithms

- 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.

Rates of growth

- We need a way to compare two functions, in this case representing the runtime of each algorithm. However, comparing values directly is surprisingly fraught:
  - Outliers
  - Implementation details
- We prefer to talk in terms of asymptotics.
  - For example, if the size of the input doubles, the function value could (approximately) double, quadruple, etc.
  - A function which quadruples will eventually 'beat' a function which doubles, regardless of the values for small inputs.

"Big-Oh" notation

Definition

We say $f(n) = O(g(n))$ if

there exist positive constants C and N such that $0 \leq f(n) \leq C\,g(n)$ for all $n \geq N$.

- $g(n)$ is said to be an *asymptotic upper bound* for $f(n)$.

- Informally, $f(n)$ is eventually (i.e. for large n) controlled by a multiple of $g(n)$, i.e. $f(n)$ grows "no faster than $g(n)$".

- Useful to (over-)estimate the complexity of a particular algorithm, e.g. insertion sort runs in $O(n^2)$ time.

Big Omega notation

Definition

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

there exist positive constants c and N such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq N$.

- $g(n)$ is said to be an *asymptotic lower bound* for $f(n)$.
- Informally, $f(n)$ eventually (i.e. for large n) dominates a multiple of $g(n)$, i.e. $f(n)$ grows "no slower than $g(n)$".

- Useful to say that any algorithm solving a particular problem runs in at least $\Omega(g(n))$, e.g. finding the maximum element of an unsorted array takes $\Omega(n)$ time, as you must read every element.

Landau notation

- $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.

- There are strict versions of Big-Oh and Big Omega notations: namely little-oh (o) and little omega (ω) respectively.

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.

Properties of Landau notation

Sum property

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

Product property

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, then $\lambda \cdot f = O(g)$ also.

The same properties hold if O is replaced by Ω, Θ, o or ω.

Table of Contents

Logarithms

Definition

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

Properties

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

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

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

Change of Base Rule

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.

Table of Contents

Arrays

- Store items with consecutive indices (always 1-based in this course).
- We'll only talk about static arrays (fixed size) but can extend to dynamic arrays.

Operations

- Random access: $O(1)$
- Insert/delete: $O(n)$
- Search: $O(n)$ (more on this later)

Linked lists

- Store each items with pointers to next and previous items.
- We will use doubly linked lists; we aren't concerned by the $2\times$ overhead

Operations

- Access next/previous: $O(1)$
- Insert/delete: $O(1)$
- Search: $O(n)$

Stacks

- Store items LIFO (last in first out).

Operations

- Access top: $O(1)$
- Insert/delete at top: $O(1)$

Queues

- Store items FIFO (first in first out).

Operations

- Access front: $O(1)$
- Insert at back: $O(1)$
- Delete at front: $O(1)$

Hash tables

- 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.

Hash tables

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: $O(n)$

Binary search trees

- 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: $O(h)$

Self-balancing binary search trees

- 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

Search

Problem

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

Algorithm

Linear search: simply check each entry of A, with early exit if x is found.

Linear search

Complexity

Worst case: $O(n)$.

Without any further information this is the best we can do; we can't avoid looking at all array elements.

- Finding the maximum or minimum entry of an array also requires linear search.

Searching a sorted array

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 $A[j] < x$.

Binary search

Algorithm

BINARY-SEARCH(A, x, ℓ, r) *searching $A[\ell..r]$ for x*

1. $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
2. **if** $A[m] = x$
3. **then return** Yes
4. **else if** $A[m] > x$
5. **then return** BINARY-SEARCH$(A, x, \ell, m-1)$
6. **else if** $A[m] < x$
7. **then return** BINARY-SEARCH$(A, x, m+1, r)$

Binary search

Complexity

Worst case: $O(\log n)$.

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 and optimisation problems

- 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?

Discrete binary search

- 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.

Table of Contents

Comparison sorting

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

Algorithm

1. For each pair of adjacent elements, compare them and swap them if they are out of order (i.e. if $A[i] > A[i+1]$).
2. Repeat until no swaps are performed in an entire pass.

Correctness

- The first pass finds the largest element and stores it in $A[n]$.
- Each subsequent pass fixes the next largest element.
- After at most n passes, the algorithm must terminate.
- It is clear that the resulting array is sorted.

Bubble sort

Complexity

- Best case: $O(n)$ (sorted array).
- Worst case: $O(n^2)$ (reverse sorted array).
- Average case: $O(n^2)$.

Conclusions

- Generally slow for large arrays.
- Fast only if the array is nearly sorted, i.e. very few passes required.
- Hardly any practical applications, but the logic of swapping adjacent elements in order to sort an array is useful in other proofs.

Selection sort

Algorithm

1. Perform a linear search to find the smallest element, and swap it with $A[1]$.
2. Repeat on $A[2..n]$, and so on.

Correctness

Obvious.

Selection sort

Complexity

- Best case: $O(n^2)$.
- Worst case: $O(n^2)$.

Conclusions

- Always slow.
- Good only if swaps (i.e. writes) are much more expensive than comparisons, because it is guaranteed to do no more than n swaps.
- Intuitive for humans sorting small arrays, e.g. a hand of cards.

Insertion sort

Algorithm

1. Make an empty linked list B, which will be kept in sorted order.
2. For each element of A:
   - compare it to each element of B from back to front until its correct place is found.
   - insert it at that place.
3. Copy the entries of B back into A.

Correctness

Obvious.

Insertion sort

Complexity

- Best case: $O(n)$ (sorted array).
- Worst case: $O(n^2)$ (reverse sorted array).
- Average case: $O(n^2)$.
- Space: $O(n)$ (can be improved to $O(1)$).

Conclusions

- Generally slow for large arrays.
- Fast only if the array is nearly sorted, i.e. very few comparisons needed for each element of A.
- Often used to sort small arrays as the last step of a more complicated sorting algorithm.

Merge sort

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.

Merge sort

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.

Heapsort

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.

Heapsort

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.

Quicksort

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.

Quicksort

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 $O(n^2)$.

Quicksort

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.

Efficient comparison sorts

- 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.

Efficient comparison sorts

Question

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

Answer

No!

Asymptotic lower bound on comparison sorting

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.

Asymptotic lower bound on comparison sorting

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!

Non-comparison sorts

- 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.

Counting sort

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.

Counting sort

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.

Counting sort

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 $\Omega(n \lg n)$ lower bound?

Answer

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

Bucket sort

Algorithm

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

- Not really a single sorting algorithm, but rather a template for certain sorting algorithms.
- Correctness follows from any item in an earlier bucket comparing $\leq$ any item in a later bucket.

Bucket sort

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.

Bucket sort

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.

MSD 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.

MSD radix sort

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.

LSD radix sort

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!

LSD radix sort

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

Graphs

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.

Graph terminology

- 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*.

Graph representations: adjacency list

- 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).

Graph representations: adjacency matrix

- 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).

Graph traversals

- 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.

Depth-first search

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.

Breadth-first search

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.

Trees

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

Puzzle

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!!