



UNSW
SYDNEY

Module 1: Foundations

Raveen de Silva (K17 202)

Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

- Identify the key features of an algorithm
- Solve problems by creatively applying standard data structures and searching and sorting algorithms
- Communicate algorithmic ideas at different abstraction levels
- Evaluate the efficiency of algorithms and justify their correctness
- Apply the \LaTeX typesetting system to produce high-quality technical documents

1. Time Complexity

2. Revision of Data Structures and Algorithms

3. Proofs

4. Puzzle

- 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.

- Useful to (under-)estimate the complexity of any algorithm solving a particular problem.
- 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 *underestimated*, 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 $O(\max(g_1, g_2))$, 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 $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.

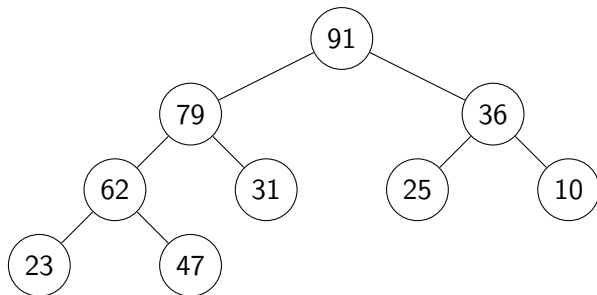
- In particular, if $f = O(g)$ and λ is a constant (i.e. $\lambda = O(1)$), then $\lambda \cdot f = O(g)$ also.
- The same property applies if O is replaced by Ω or Θ .
- This is useful for analysing algorithms that have two or more nested parts.
- If each execution of the inner part takes f_2 time,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

1. Time Complexity
2. Revision of Data Structures and Algorithms
3. Proofs
4. Puzzle

- 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: $O(\log n)$



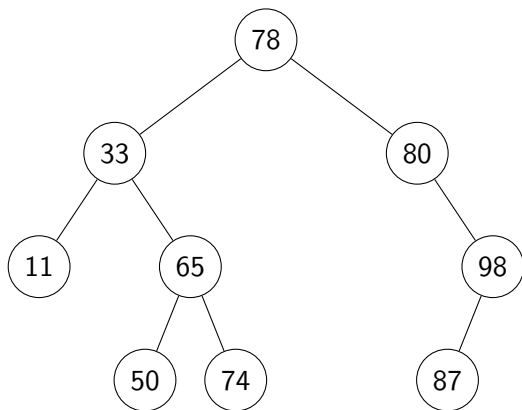
i	1	2	3	4	5	6	7	8	9
$A[i]$	91	79	36	62	31	25	10	23	47

- 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.

Problem

You are given an array A of $n - 1$ elements. Each number from 1 to n appears exactly once, except for one number which is missing. Find the missing number.

Idea

Record whether you've seen each possible value.

- This leads to a linear time, linear space algorithm.
- How could you solve the problem in linear time and *constant* space?

- To count the occurrences of integers from 1 to m , simply make a zero-initialised array of frequencies and increment every time the value is encountered.
- The real question is how to count the occurrences of other kinds of items, such as large integers, strings, and so on.
- Try to do the same thing: map these items to integers from 1 to m , and count the occurrences of the mapped values."

- 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.

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: $O(1)$

- A situation where two (or more) keys have the same hash value is called a *collision*.
- When mapping from a large key space to a small table, it's *impossible* to completely avoid collisions.
- There are several ways to manage collisions – for example, separate chaining stores a linked list of all colliding key-value pairs at each index of the hash table.

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: $O(n)$

1. Time Complexity
2. Revision of Data Structures and Algorithms
3. Proofs
4. Puzzle

- Propositions are true/false statements. Some examples:
 - The sky is orange.
 - Basser Steps consists of more than 100 stairs.
 - Everyone in this room is awake.
- The goal of an argument is usually to establish a relationship between propositions, often also using the following operations.

Notation	Meaning	Formal term
$\neg P$	not P	negation
$P \wedge Q$	P and Q	conjunction
$P \vee Q$	P (inclusive-) or Q	disjunction
$P \rightarrow Q$	if P then Q	implication

- The quantifiers “for all” (\forall) and “for some” (\exists) are also very common.

- Used to prove a sequence of propositions P_1, P_2, P_3, \dots
- If the first proposition is true, and each one implies the next, then they are all true.
- If we make an infinite line of dominoes so that each one can knock over the next, and we tip over the first domino, then all the dominoes will fall.

- Used to prove a proposition P by considering the alternative.
- If the assumption $\neg P$ leads to a contradiction ($Q \wedge \neg Q$ for some other proposition), then $\neg P$ is impossible, so P holds.
- Example: double-move chess, where each side moves twice on their turn.
 - With perfect play, White can at least draw.
 - Suppose otherwise. Then Black must have a winning strategy that works no matter what White does.
 - White could waste their first turn by moving a knight out and back, then implement the winning strategy themselves.
 - So Black's strategy is not winning after all!

- Whenever we present an algorithm, we must justify its correctness and efficiency.
- We need to provide reasoning for the claims that:
 - it always gives the right answer, and
 - it runs in the time complexity we claim.
- Straightforward claims might need only a sentence of reasoning, but for a less obvious claim the burden is higher.

Algorithm

We recursively apply the following algorithm to the subarray $A[\ell..r]$.

If $\ell = r$, i.e. the subarray has only one element, we simply exit. This is the base case of our recursion.

Otherwise:

- first define $m = \lfloor \frac{\ell+r}{2} \rfloor$, the midpoint of the subarray,
- then apply the algorithm recursively to $A[\ell..m]$ and $A[m+1..r]$, and
- finally merge the subarrays $A[\ell..m]$ and $A[m+1..r]$.

- The depth of recursion in mergesort is $\log_2 n$.
- On each level of recursion, merging all the intermediate arrays takes $O(n)$ steps in total.
- Thus, mergesort always terminates, and in fact it terminates in $O(n \log_2 n)$ steps.
- Merging two sorted arrays always produces a sorted array, thus, the output of mergesort will be a sorted array.
 - This is actually a proof by induction on the size of the array!
 - Where we can communicate the idea succinctly, we should do so. An argument can be rigorous without notation for its own sake.

- Sometimes it is *not* easy to gather from the description of an algorithm:
 - that it terminates, rather than entering an infinite loop,
 - how many steps it takes to terminate, and whether this is acceptably small, or
 - whether the answer it reports is actually correct.
- We always need to justify the claims we make about algorithms, and occasionally that requires a detailed proof.

- Suppose you have hired n frontend engineers and n backend engineers to build n apps in pairs.
- Every frontend engineer submits a list of preferences, which ranks all the backend engineers from their most preferred partner to least preferred, *and vice versa*.
- We'd like to make n separate pairs in order to “make everyone happy” in some sense.

Definition

A *matching* is an assignment of engineers so that no-one belongs to two or more pairs.

Definition

A *perfect matching* is a matching involving *all* frontend engineers and *all* backend engineers, i.e. where everyone belongs to exactly one pair.

Our task is to design a perfect matching that somehow satisfies the frontend and backend engineers, with regards to their preferences.

Question

Can we assign every engineer their first preference partner?

Answer

Not necessarily!

- We'll have to lower our expectations. Let's aim for an allocation that doesn't make anyone too unhappy.
- If a frontend engineer and a backend engineer were both very unhappy with their allocated partner, they might quit and go build an app together.

Definition

A *stable matching* is a perfect matching in which there are no two pairs (f, b) and (f', b') such that:

- f prefers b' to b , **and**
- b' prefers f to f' .

Stable matchings are self-enforcing; no *unmatched* pair of frontend engineer f and backend engineer b' will quit.

We will design an algorithm which produces a stable matching.

In these examples, we'll label the engineers for ease of reference.

- Frontend: Flora  and Fred 

- Backend: Betty  and Bilal 

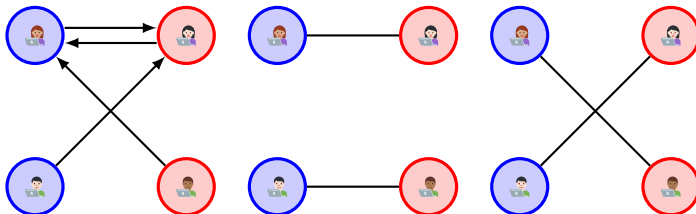
In each example, all four engineers will rank their two counterparts from most preferred to least preferred.

Flora 🧑:	Betty 🧑, Bilal 🧑	Betty 🧑:	Flora 🧑, Fred 🧑
Fred 🧑:	Betty 🧑, Bilal 🧑	Bilal 🧑:	Flora 🧑, Fred 🧑

Preferences

Stable

Not stable

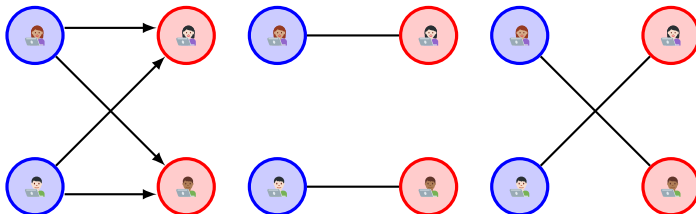


Flora  :	Betty  , Bilal 	Betty  :	Fred  , Flora 
Fred  :	Bilal  , Betty 	Bilal  :	Flora  , Fred 

Preferences

Stable

Stable



Exercise

Up to symmetry, there are two more cases to consider. Solve them.

The following rules cover all situations with two frontend and two backend engineers.

- If Flora and Fred prefer different backend engineers, assign each of them their preferred partner. Neither Flora nor Fred wants to swap, so the matching is stable.
 - The same applies vice versa, i.e. if Betty and Bilal prefer different frontend engineers.
- However, if both Flora and Fred prefer the same partner b **and** both Betty and Bilal prefer the same partner f , then put f with b and pair the two remaining engineers. Neither f nor b wants to swap, so the matching is stable.

Question

Given n frontend and n backend engineers, how many ways are there to match them, without regard for preferences?

Answer

$n! \approx (n/e)^n$ - more than exponentially many in n ($e \approx 2.71$).

Question

Is it true that for every possible collection of n lists of preferences provided by all frontend engineers, and n lists of preferences provided by all backend engineers, a stable matching always exists?

Answer

Perhaps surprisingly, yes!




Question

Can we find a stable matching in a reasonable amount of time?

Answer

Yes, using the *Gale-Shapley algorithm*.

- Produces pairs in stages, with possible revisions
- Frontend engineers will be pitching to backend engineers (arbitrarily chosen). Backend engineers will decide to accept a pitch or not.
- A frontend engineer who is not currently in a pair will be called *solo*.
- Start with all frontend engineers solo.

- While there is a solo frontend engineer who has not pitched to all backend engineers, pick any such solo frontend engineer, say Femi .
- Femi pitches to the highest ranking backend engineer Brad  on his list, ignoring any who he has pitched to before.
 - If Brad is not yet paired, he accepts Femi's pitch (at least tentatively).
 - Otherwise, if Brad is already paired with someone else, say Frida :
 - if Brad prefers Femi to Frida, he turns down Frida and makes a new pair with Femi (making Frida now solo)
 - otherwise, Brad will turn down Femi and stay with Frida.

Claim 1

The algorithm terminates after $\leq n^2$ rounds.


Proof

- In every round of the algorithm, one frontend engineer pitches to one backend engineer.
- Every frontend engineer can make a pitch to a particular backend engineer at most once.
- Thus, every frontend engineer can make at most n pitches.
- There are n frontend engineers, so in total they can make $\leq n^2$ offers.
- Thus there can be no more than n^2 many rounds.

Claim 2

The algorithm produces a perfect matching, i.e. when the algorithm terminates, the $2n$ engineers form n separate pairs.

Proof

- Assume that the algorithm has terminated, but a frontend engineer Felix  is still solo.
- This means that Felix has already pitched to all the backend engineers.
- A backend engineer is not paired only if no-one has pitched to them. Since Felix has pitched to everyone, all n backend engineers must be paired.
- No frontend engineer can have more than one partner, so all n frontend engineers must also be paired, including Felix.
- This is a contradiction, completing the proof.

Claim 3

The matching produced by the algorithm is stable.

- Recall that each frontend engineer pitches in order from their most preferred to their least preferred backend counterpart.
- Therefore, the sequence of backend engineers paired with a particular frontend engineer is in this order also.
- Each backend engineer is initially not paired, then accepts the first pitch made to them, and only ever switches to a different frontend counterpart who they prefer to their current partner.
- Thus, each backend engineer is paired with frontend engineers in order from their least preferred to their most preferred.

Proof

We will prove that the matching is stable using *proof by contradiction*.

Assume that the matching is not stable. Thus, there are two pairs:

- Feng 🧑 and Bina 🧑, and
- Farah 🧑 and Bayo 🧑, such that:
- Feng prefers Bayo over Bina and
- Bayo prefers Feng over Farah.

Proof (continued)

Feng : ..., Bayo , ..., Bina  (circled), ...

Bayo : ..., Feng , ..., Farah  (circled), ...

- Since Feng prefers Bayo over Bina, he must have made a pitch to Bayo before Bina.
- Bayo must have either:
 - rejected Feng because he was already paired with a frontend engineer he prefers to Feng, or
 - accepted Feng only to later rescind this and accept a pitch from someone he prefers to Feng.

Proof (continued)

Feng 🧑: ..., Bayo 🧑, ..., Bina 🧑, ...

Bayo 🧑: ..., Feng 🧑, ..., Farah 🧑, ...

- In both cases Bayo would become paired with someone higher than Feng in his preferences.
- However he ends up with Farah, who he prefers even less than he does Feng.
- This is a contradiction, completing the proof.

1. Time Complexity
2. Revision of Data Structures and Algorithms
3. Proofs
4. 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 amount of petrol from all stations is exactly enough to go around the highway once.

You want to start at a petrol station, take all its fuel, and drive clockwise around the highway once. Each time you reach a petrol station, you will take all its fuel too.¹ The only problem is the possibility that you might run out of fuel before reaching the next petrol station!

Prove that there is always at least one starting petrol station to make it around the highway.

¹Your petrol tank is large enough to hold all the fuel.



That's All, Folks!!