



School of Computer Science and Engineering
COMP3121/9101 Algorithm Design and Analysis

Module 1: Foundations

Raveen de Silva
K17-202

Course Admins:
cs3121@unsw.edu.au
(Liam, Leith, Hugo, Stephen, James, Brendan)

2026 T3

Table of Contents

1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction
5. Proving Claims about Algorithms

Running times

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 will make such statements more precise.
- ▶ We also want to understand when they are wrong.

Best and worst case performance

- ▶ Such statements are most commonly meant to compare 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.

Average case performance

- ▶ 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**, **expected**, **typical**, or **smoothed** performance of an algorithm requires probabilistic methods and is beyond the scope of this course.
- ▶ The only case where we are allowed to use expected running times is where we rely purely on results of this type from prior courses (e.g., hash table operations, quicksort).

Rates of growth

- ▶ 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.

Big-Oh notation

Definition (Big-Oh)

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

Alternative notation: $f(n) = O(g(n))$.

- ▶ $g(n)$ 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.

Big-Oh notation: Examples

Example 1

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

Example 2

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

Example 3

Let $f(n) = 0.001n^3$. Then $f(n) \notin O(n^2)$, 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 have an impact on the actual running time, but the dominant term of the running time will still be a constant times h , n^2 or n respectively.

Big-Oh notation

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

It is sufficient that there exists a number n_0 such that the property holds for all $n \geq n_0$.

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 important in practice.

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

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 $O(\cdot)$ can have a significant impact on practical running times.

Big-Omega notation

Definition (Big Omega)

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

Alternative notation: $f(n) = \Omega(g(n))$.

- ▶ $g(n)$ 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.

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: Look up the more formal definitions of $O(\cdot)$ and $\Omega(\cdot)$ in either textbook and prove that if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$.

Big-Theta notation

Definition (Big Theta)

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

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

- ▶ $f(n)$ 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

Both versions are correct. The statement
bubble sort runs in $O(n^2)$ time in the worst case

asserts that the worst case running time is **at most** quadratic, while the statement

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

asserts that the worst case running time is **exactly** quadratic.

Big-Theta notation

Big Oh versus Big Theta

The $\Theta(\cdot)$ statement conveys 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.

Note that sometimes confusions arise because people write the more commonly used $O(\cdot)$ as a shorthand for the less commonly used $\Theta(\cdot)$. Please be aware of the difference.

Sum property

Fact

Sum property If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(g_1 + g_2)$.

- ▶ 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.

Product property

Fact

Product property If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.

- ▶ In particular, if $f \in O(g)$ and λ is a constant (i.e., $\lambda \in O(1)$), then $\lambda \cdot f \in O(g)$.
- ▶ 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

1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction
5. Proving Claims about Algorithms

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, suppressing the base.

Table of Contents

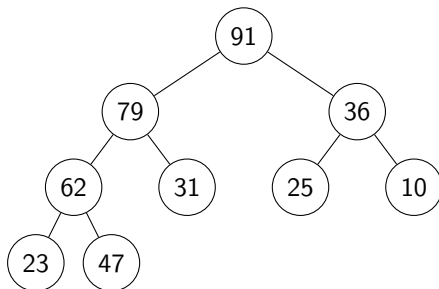
1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction
5. Proving Claims about Algorithms

(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: $O(\log n)$



Array representation:

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

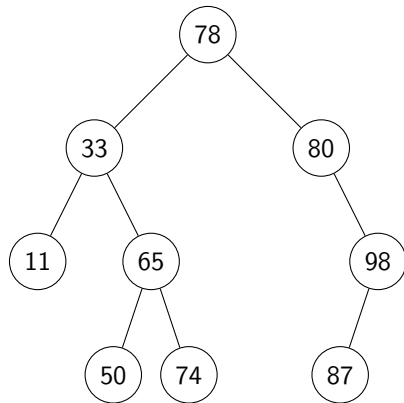
Binary search trees

- ▶ 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 $h \in O(\log n)$ and therefore all tree operations run in $O(\log n)$ time.
- ▶ 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.

Frequency tables

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.



Record whether you've seen each possible value.

This leads to a linear time, linear space algorithm.



Challenge: How could you solve the problem in linear time and **asymptotically less than linear** space?

Problem

You are given an array A of $m > n$ elements. Each entry is a number from 1 to n . Count the number of occurrences of each number.

- ▶ To count the occurrences of integers from 1 to n , simply make a zero-initialised array of frequencies and increment the value at index i every time the value i 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 n , and count the occurrences of the mapped values.

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.

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: $O(1)$

- ▶ When two (or more) keys have the same hash value we have a **collision**.
- ▶ When mapping from a large key space to a small table, it's **impossible** to completely avoid collisions.
- ▶ There are 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: $O(n)$

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 $A[j] < x$.

Algorithm

We search the subarray $A[\ell..r]$ for x recursively.

If the subarray has no elements (i.e., $r < \ell$), then x cannot be found.

Otherwise:

- ▶ first define $m = \lfloor \frac{\ell+r}{2} \rfloor$, the midpoint of the subarray,
- ▶ then, we take cases by comparing to $A[m]$, the value at the midpoint.
 - If $A[m] = x$, we can conclude the search.
 - If $A[m] > x$, we recurse on the left subarray $A[\ell..m-1]$.
 - If $A[m] < x$, we recurse on the right subarray $A[m+1..r]$.

Binary search

Complexity

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

Initially, $\ell = 1$ and $r = n$. At each step, the search space halves, which can only happen $\log_2 n$ times.

Table of Contents

1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction
5. Proving Claims about Algorithms

Problem solving by reduction

- ▶ Recall the example problem “Geese in a Trenchcoat” (formally known as “generalised 2-sum”), in which we applied sorting and binary search to identify whether two numbers from a given collection add to a given target.
- ▶ If you were to write a program to solve this problem, how much code would you write?
 - In most programming languages, you would not implement the sort or the binary search; you would instead call a **library function** for each.
 - We need only write the wrapper that invokes binary search for each element that could make one half of the pair.
 - It is a good habit when programming to rely on proven code, which you can safely assume does not have bugs.
- ▶ For our purposes, this perspective is often helpful in simplifying the process of solving problems.
 - We can reliably invoke well-established subroutines to solve a larger and more complex problem.
- ▶ Presentation of the algorithm is also improved.
 - We can first justify that the subroutine achieves its goals, and then rely on its properties in the analysis of the wider algorithm.

Story

Alice and Bob live together. One day, Alice is cooking dinner, when she accidentally starts a fire in the pan. They quickly stifle the fire, and manage to extinguish it. The dinner is unfortunately ruined, but they order a delicious pizza and carry on with their evening.

The next day, it is Bob's turn to cook dinner, when he has a bright idea. He sets the food on fire, and declares "I have reduced this to a problem that we have solved before! Who's for pizza?"

- ▶ "Reduction" encompasses a family of technical terms, each with specific meanings.
- ▶ We will see these in more detail in the module on *Intractable Problems*; for now, we will focus on the *strategy* and how you can apply it to elegantly solve problems using your pre-existing knowledge.

Example problem: Shortest Colourful Path

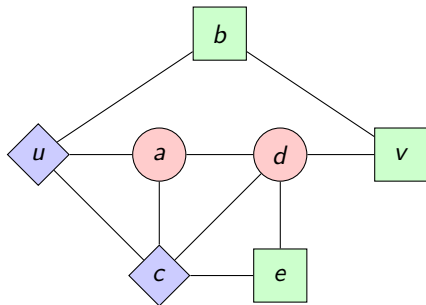
Problem

Let $G = (V, E)$ be an undirected and unweighted graph, where each vertex is coloured either red, green, or blue. We say that a path in G is colourful if it is a path that does not visit two consecutive vertices of the same colour.

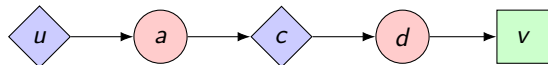
Given the graph $G = (V, E)$ and two distinct vertices $u, v \in V$, design an algorithm that computes the length of the shortest colourful path from u to v .

Example problem: Shortest Colourful Path

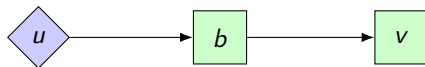
For example, consider the graph G below.



The shortest colourful path from u to v in G is the following path with length 4.



However, the shortest path without the colour restriction is the following path with length 2.

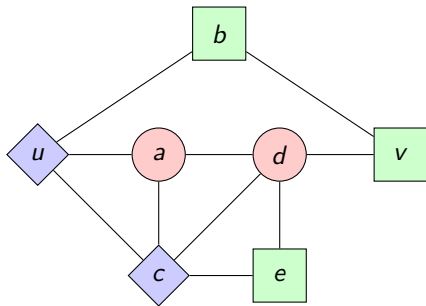


Example problem: Shortest Colourful Path

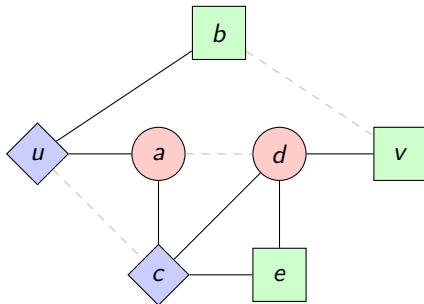
- ▶ If not for the colours, the problem would be to find the shortest path in an unweighted graph.
 - Recall that this is solved by Breadth-First Search (BFS).
- ▶ We cannot apply BFS immediately here because it may use illegal edges, i.e. edges that do not represent a change in colour.
- ▶ There are two main ways to employ BFS in the solution:
 1. **Change the algorithm.**
In the loop over neighbours, skip over those which have the same colour as the current vertex.
 2. **Map the problem instance.**
Remove any edges of graph G whose vertices have the same colour, thus forming a new graph H . A shortest colourful path in G corresponds exactly to a shortest path in H , which can be found using BFS.

Example problem: Shortest Colourful Path

Recall the earlier example graph G .



The new graph H would be as follows.



Example problem: Shortest Colourful Path

- ▶ We prefer the latter approach, mapping instances of Shortest Colourful Path to instances of Shortest Path.
 - In a programming context, it means adapting our problem instance to that which is solved by a library function, rather than implementing a small variation of the library function.
 - Even if no code is to be written, the analysis is simpler to conduct, and the conclusions are **more easily generalised**.
- ▶ Is our algorithm correct?
 - Proof hinges entirely on the correspondence (formally: bijection) between instances of the two problems.
- ▶ What is the time complexity of our algorithm?
 - Time taken to construct the new graph H , plus time taken to run BFS on it.
- ▶ What is the relationship between the two problems?
 - If we have an algorithm for Shortest Path, then we also have an algorithm for Shortest Colourful Path.

Binary search

- ▶ Recall the use of binary search to determine whether a value x in a non-increasing^a array A of length n , in $O(\log n)$ time.
- ▶ This is particularly powerful as a wrapper for other processes!
- ▶ Did we need to have the entire array in advance?
- ▶ There are natural extensions. We can use the same method to find the first array element to compare $\leq x$, or the last to compare $\geq x$.
- ▶ What if our array consisted entirely of 0s and 1s?
 - We could use binary search to find the last 1 or the first 0.

^aDefinition: $A[i] \geq A[i + 1]$ for all $1 \leq i < n$.

Decision problems and optimisation problems

► Decision problems

- Questions of the form: “Given some input, is there an x which ...”
- Answer is “Yes” or “No”.
- Example: given a graph G and an integer x , does G contain a clique of x vertices?

► Optimisation problems

- Questions of the form: “What is the smallest/largest x for which ...”
 - Answer is an object, e.g., a number or a set.
 - Example: given a graph G , what is the size of the largest clique, or what is the largest clique?
- An optimisation problem is typically harder than the corresponding decision problem.
- But we can sometimes solve optimisation problems by invoking an algorithm for the decision problem only a few times.

Example optimisation problem: Maximum Median

Definition (median)

Recall that the **median** of an array is the middle value in sorted order. For example, the median of $[1, 2, 2, 4, 5]$ is 2.

Problem

Ling has an array A consisting of $2n - 1$ integers, sorted from smallest to largest. Note that the median is initially $A[n]$.

She wants her numbers to grow big and strong, so for each of the following k days she adds 1 to one of her numbers.

At the end of the k days, what is the largest possible median that her array can have?

Example

If the starting array is $[1, 2, 2, 4, 5]$ and $k = 3$, the maximum median is 4, achieved by adding 1 to the third number on the first two days and to any number on the last day.

Using all three increases on the third number does not make the median value 5.

Example optimisation problem: Maximum Median

- ▶ How do we solve this problem?
- ▶ Which numbers should we increase, and how many times?

Observation

We are only concerned with the middle number and everything after it. Numbers from the first half of the array can be left small.

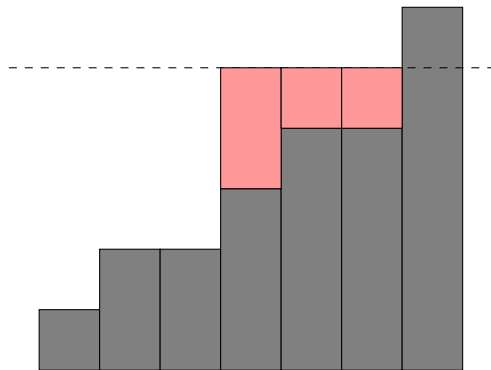
- ▶ But we still don't know how much to increase the numbers in the second half!
- ▶ What if we did know? What if we had a target to aim for, and we just had to report whether it was possible or not?

Example decision problem: Maximum Median

Related decision problem

Given positive integers t and k , Ling wants to know whether she can make the median of her numbers at least the target value t (by adding 1 to one of her numbers each day for k days).

- ▶ How do we get the median to at least t ?
 - Increase the middle value $A[n]$ and everything after it to t ; nothing to do for any values that are $\geq t$ already.
- ▶ Why might we fail?
 - t could be too big for our particular array and number of days k .



Example decision problem: Maximum Median

- ▶ To increase the median to at least t , we will need to increase the values:
 - at index n by $\max(t - A[n], 0)$
 - at index $n + 1$ by $\max(t - A[n + 1], 0)$
 - ...
 - at index $2n - 1$ by $\max(t - A[2n - 1], 0)$.

Algorithm for the related decision problem

Iterate over indices i from n to $2n - 1$ to calculate

$$\sum_{i=n}^{2n-1} \max(t - A[i], 0).$$

If this value is $\leq k$, report yes; otherwise report no.

- ▶ This algorithm clearly runs in $O(n)$ time in the worst case.

Example problem: Maximum Median

Decision problem

Given positive integers t and k , Ling wants to know whether she can make the median of her numbers at least the target value t (by adding 1 to one of her numbers each day for k days).

- ▶ **Solved!** We can now test whether it's possible to reach (or exceed) any target in $O(n)$ time.
- ▶ Can we use this to solve our original optimisation problem?

Optimisation Problem

At the end of the k days, what is the largest possible median t that Ling's array can have?

- ▶ **Solution approach.** Apply our decision algorithm for different target values t . Which values of t do we need to test?

Example problem: Maximum Median

- ▶ **Question.** What possible target values t do we need to test in order to solve the optimisation problem?
 - Anything less than the current median $A[n]$ can't possibly be the maximum median.
 - The median can't become any larger than $A[n] + k$, as we don't have enough days to grow $A[n]$ that high, let alone $A[n + 1]$ and so on.
- ▶ **Solution to optimisation problem.** Apply our decision algorithm to every target value t from $A[n]$ through to $A[n] + k$.
- ▶ **Time complexity.** We test $k + 1$ different target values, each taking $O(n)$ time. This gives a time complexity of $O((k + 1)n) = O(kn)$.
- ▶ **Question.** Can we do better than this? Are there any target values in this range that we can skip?
 - If we test a target t , does the result give us any information about **other** targets?

Example problem: Maximum Median

- ▶ **Question.** If we test a target t , does the result give us any information about other targets?
 - If a target t is achievable, then any smaller target is also achievable!
 - If a target t isn't achievable, then any larger target is also not achievable.
 - **Monotonicity:** targets are all achievable until some cutoff, and not achievable thereafter.
- ▶ **Better solution to optimisation problem.**
 - We can binary search on “is target t achievable?” for t in the range $[A[n], A[n] + k]$.
 - We don't have a pre-computed array of yes/no values; instead we'll compute them as they're needed in the binary search, by using our decision algorithm
- ▶ **Time complexity.** The binary search has $O(\log k)$ steps each taking $O(n)$, so the time complexity is $O(n \log k)$.

Discrete binary search

- ▶ The approach we just used, of solving an optimisation problem by applying binary search over input values for the related decision problem, is called **discrete binary search**.
- ▶ For which optimisation problems can we apply discrete binary search?
 - Suppose we have an optimisation problem that asks to find the largest integer x with some properties.¹
 - Let $f(b)$ be the outcome of the decision problem when $x = b$, with 0 for false and 1 for true.
 - In some (but not all) such problems, if the condition holds with $x = b$ then it also holds with $x = b - 1$.
 - Thus, f is all 1's up to the first 0, after which it is all 0's.
 - Since this is monotonic, the optimisation problem can be rephrased as finding the largest x such that $f(x) = 1$ using discrete binary search.
 - The method is equally applicable if f is non-decreasing ($f(b) \implies f(b + 1)$) instead.

¹Similar ideas work for real x , but termination is complicated; in practice, we can usually stop after a finite number of iterations, by which point we have narrowed the search space to a tiny window.

Discrete binary search

Question. What is the running time of discrete binary search?

- ▶ Suppose each decision problem can be solved in some running time $O(g(n))$.
- ▶ If the range of possible input values we need to test for the decision problem is k , then the number of iterations of the binary search is $O(\log k)$.
- ▶ So the time complexity of the discrete binary search is $O(g(n) \cdot \log k)$.
- ▶ Once again, the key strategic insight is that **if monotonicity applies**, then an algorithm for the decision problem could be invoked as a subroutine of an algorithm for the corresponding optimisation problem. The number of times invoked is logarithmic in the size of the answer space.

Recursive problem-solving

- ▶ Thus far, we have solved unseen problems by invoking known algorithms which solve related instances of a different well-studied problem.
 - ▶ We could likewise relate larger instances of a problem to smaller instances of **the same problem**.
 - ▶ We would then be able to relate those to even smaller instances, and so on.
- ▶ Recursion is a powerful tool, underpinning familiar algorithms including binary search and merge sort.
 - ▶ Recursion will be crucial to two future modules: *Dynamic Programming* and *Divide and Conquer*.

Example problem: Palindrome Testing

Problem

Given a string s consisting of n letters, report whether or not s is a palindrome.

- ▶ While there is a straightforward iterative solution, it is slightly cumbersome to specify clearly.
- ▶ We can instead take a recursive perspective. A string of length $n \geq 2$ is a palindrome if and only if:
 - s_1 and s_n match, and
 - $s_2s_3 \dots s_{n-1}$ is a palindrome.
- ▶ How do we check whether this substring is a palindrome?
 - By again checking its endpoints and reducing to an even smaller substring.
 - If these checks are all successful, where does the recursion end?
 - We cannot recurse further once we are left with a single letter or the empty string, both of which are palindromes; these will be our base cases.

Time complexity

- ▶ We must remember this strategy is not 'free', that is, there are computational costs.
- ▶ We need to account for:
 - the number of times that the subroutine is called,
 - the time complexity of each call, and
 - any other work performed in our algorithm.
- ▶ In Shortest Colourful Path:
 - we called the BFS subroutine only once,
 - its time complexity is $O(V + E)$ where V and E are the number of vertices and edges respectively of the constructed graph (which has as many vertices as, and potentially fewer edges than the original graph), and
 - our algorithm had to also check each original edge for omission from the new construction, taking $O(E)$ time.
- ▶ This analysis can be complicated for some recursive algorithms; we'll see more of this in the *Divide and Conquer* module.

Example problem: Hamiltonian Path

Problem

Given a graph $G = (V, E)$ and two specified vertices $s, t \in V$, is there a path from s to t which visits each vertex exactly once?

- ▶ If such a path exists, it must visit the other vertices in some particular order.
- ▶ Each order is a possible path, and to check a possible path, we need only confirm the presence of each of its edges, which we know how to do.
- ▶ This certainly is an algorithm for Hamiltonian Path, but its time complexity is poor.
- ▶ Later in *Intractable Problems* we will see more (but not much more!) efficient algorithms for this problem.

Table of Contents

1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction
5. Proving Claims about Algorithms

Propositional logic

- ▶ Propositions are true/false statements.

Some examples:

- Clancy the lion is the mascot of UNSW.
 - Mount Everest is less than 500 times the height of Bassar Steps.
 - These slides are written in $\text{\LaTeX}$, using the Beamer document class.
- ▶ The goal of an argument is usually to establish a relationship between propositions, often using logical connectives.

Common logical connectives

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
$P \Leftrightarrow Q$	P if and only if Q	equivalence

The quantifiers “for all” ($\forall$) and “there exists” ($\exists$) are also very common; these are features of *predicate logic*.

Application to computer science

- ▶ Much of our work involves specifying how statements interact with each other.
- ▶ In particular, implications represent our deductions from information given in the question to conclusions that we want to make.

Tree definitions

Let $G = (V, E)$ be an undirected graph.

If G is connected and $|E| = |V| - 1$, then G does not contain any cycles.

Restatement using propositions

$$\left[(G \text{ is connected}) \wedge (|E| = |V| - 1) \right] \Rightarrow (G \text{ has no cycles})$$

Reasoning about algorithms

- ▶ 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.

Direct Proof

- ▶ If we can follow implications from known facts to the desired conclusion, this is usually simplest.

Example: Heapsort

Prove that heapsort runs in $O(n \log n)$ time in the worst case.

- ▶ Recall heapsort:
 1. Add all array elements to a min-heap
 2. n times, remove the top element and append it to the output sequence
- ▶ We can prove the time complexity by accounting for each operation.

Proof

1. Constructing the heap takes $O(n)$ time (although even inserting the entries each in logarithmic time would suffice).
2. Each access to the top of the heap takes constant time, as does each append. Removal however takes logarithmic time.

Therefore the total time complexity is $O(n + n \cdot \log n) = O(n \log n)$.

Proof by contradiction

- ▶ Used to prove a proposition P by considering its negation.
- ▶ If the assumption $\neg P$ leads to a contradiction ($Q \wedge \neg Q$ for some proposition Q), then $\neg P$ is impossible, so P holds.

Example: tree diameter

Define a diameter of a tree to be a longest simple path between any pair of vertices. Consider the following algorithm to find the diameter length of a given tree T .

1. Perform a breadth-first search from any vertex u , and let v be the last vertex visited.
2. Perform a breadth-first search from v , and w be the last vertex visited.
3. Report the distance $v \rightarrow w$ as the diameter length of the tree.

Prove the correctness of this algorithm.

Recall that BFS visits vertices in order of distance, so the last vertex visited is the furthest from the starting vertex.

Example: tree diameter

- ▶ The given algorithm first ran a BFS from u , and labelled the last seen vertex as v .
- ▶ We will first need to establish that there is some diameter with endpoint v .
- ▶ Suppose otherwise:
 - Suppose there is no diameter with endpoint v . Then there must be some path $x \rightarrow y$ that is strictly longer than all paths starting at v .
 - View the tree as being rooted at u . Then v is on the deepest level of the tree.
 - Let l be the lowest common ancestor of x , y and v .
 - The paths $v \rightarrow x$ and $v \rightarrow y$ cannot both avoid l (or else some child of l would be the lowest common ancestor). It follows that $x \rightarrow y$ cannot be longer than both $v \rightarrow x$ and $v \rightarrow y$, contradicting our earlier assumption.
- ▶ Now that some diameter has v as an endpoint, we can find it by running a BFS from v to find the furthest vertex w .

Proof by invariants

- ▶ Used to prove that a certain property P always holds during the execution of an algorithm.
- ▶ If P holds at the start, and each step of the algorithm preserves P , then P holds throughout.
- ▶ Typically used to prove correctness of loops.

Example: insertion sort

- ▶ **Invariant:** At the start of each iteration of the outer loop, the subarray to the left of the current element is sorted.
- ▶ **Initialisation:** This property holds at the start, as a single-element array is sorted
- ▶ **Maintenance:** The invariant is preserved at each iteration of the loop, by inserting the current element into the sorted subarray.
- ▶ **Termination:** Thus, at the end of the algorithm, the entire array is sorted.

Proof by (mono)variants

- ▶ Used to prove that a certain quantity Q changes monotonically during the execution of an algorithm, with some bound to ensure termination.
- ▶ If Q has some non-negative initial value, and each step of the algorithm decreases Q , and Q cannot become negative, then the algorithm terminates.
- ▶ Typically used to prove termination of procedures without an explicit ending condition, or bound the number of iterations.

Example: bubble sort

- ▶ **(Mono)variant:** The number of *inversions*, i.e. pairs of indices $i < j$ such that $A[i] > A[j]$.
- ▶ **Initialisation:** This count is positive or zero initially.
- ▶ **Maintenance:** Each swap performed by bubble sort reduces the number of inversions by one.
- ▶ **Termination:** If there is an inversion, then there is an adjacent inversion (proof left as an exercise). If so, bubble sort will find and resolve it. Therefore the number of inversions decreases one-by-one to zero, at which point the algorithm terminates with the array correctly sorted.

Proof by induction

- ▶ 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.
- ▶ We also make use of strong induction: if the first proposition is true, and the first k always imply the $(k + 1)$ th, then they are all true.

Example: selection sort

Prove that selection sort is a correct sorting algorithm.

Example: selection sort

- ▶ Recall selection sort:
for each i from 1 to n : identify the smallest value in $A[i..n]$ and swap it with $A[i]$.
- ▶ After one round, the algorithm effectively applies selection sort to the subarray $A[2..n]$. This recursive structure is well suited to proof by induction. We will prove that selection sort correctly sorts arrays of size n , for all positive integers n .

Proof

Base case: For $n = 1$, the algorithm clearly reports the single entry as is, which is correct.

Induction hypothesis: Suppose arrays of length k are correctly sorted by selection sort.

Induction step: Now we prove that arrays of length $k + 1$ are correctly sorted by selection sort.

Let A be an array of length $k + 1$. After performing the first swap in selection sort, the elements are rearranged such that $A[1] \leq A[2], \dots, A[k + 1]$. Applying selection sort to $A[2..(k + 1)]$ will sort it by the induction hypothesis, so then $A[2] \leq A[3] \leq \dots \leq A[k + 1]$. Thus array A is correctly sorted.

Conclusion: By induction, arrays of all sizes are correctly sorted by selection sort.

Example: mergesort

To sort an array $A[1..n]$, apply the following recursive algorithm to $(A, 1, n)$.

mergesort(A, ℓ, r)

The algorithm sorts the subarray $A[\ell..r]$ as follows.

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, ℓ, m) and $(A, m+1, r)$, and
- ▶ finally merge the subarrays $A[\ell..m]$ and $A[m+1..r]$ into a sorted subarray $A[\ell..r]$.

merge(A, ℓ, m, r)

The algorithm sorts the subarray $A[\ell..r]$, assuming that $\ell \leq m \leq r$ and that $A[\ell..m]$ and $A[m+1..r]$ are already sorted.

Create a temporary array B of size $r - \ell + 1$ and fill it from left to right.

Use indices i, j pointing to the smallest uncopied elements in each of the two subarrays of A .

While i and j are not out of bounds, the algorithm copies the smaller of $A[i]$ and $A[j]$ into B (ties are broken arbitrarily), and it increments the corresponding index i or j . Afterwards, it copies any remaining elements into B in order.

Finally, overwrite $A[\ell..r]$ with B .

Mergesort: correctness of merge step

To show that merging the sorted subarrays $A[\ell..m]$ and $A[m+1..r]$ into $A[\ell..r]$ produces a sorted array, we use an invariant.

- ▶ **Invariant:** at any point in time the elements copied into B so far are the smallest elements of $A[\ell..r]$ in sorted order.
- ▶ **Initialisation:** This holds at the start as no element has been copied yet.
- ▶ **Maintenance:** Each iteration preserves the invariant as the smallest remaining element from the two subarrays of A is copied into B .
- ▶ **Termination:** When all elements have been copied, the invariant implies that B contains all elements of $A[\ell..r]$ in sorted order.
- ▶ Thus, after overwriting $A[\ell..r]$ with B , the subarray $A[\ell..r]$ is sorted.

Recall:

merge(A, ℓ, m, r)

The algorithm sorts the subarray $A[\ell..r]$, assuming that $\ell \leq m \leq r$ and that $A[\ell..m]$ and $A[m+1..r]$ are already sorted.

Create a temporary array B of size $r - \ell + 1$ and fill it from left to right.

Use indices i, j pointing to the smallest uncopied elements in each of the two subarrays of A .

While i and j are not out of bounds, the algorithm copies the smaller of $A[i]$ and $A[j]$ into B (ties are broken arbitrarily), and it increments the corresponding index i or j . Afterwards, it copies any remaining elements into B in order.

Finally, overwrite $A[\ell..r]$ with B .

Mergesort: correctness

Recall:

mergesort(A, ℓ, r)

The algorithm sorts the subarray $A[\ell..r]$ as follows.

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, ℓ, m) and $(A, m+1, r)$, and
- ▶ finally merge the subarrays $A[\ell..m]$ and $A[m+1..r]$ into a sorted subarray $A[\ell..r]$.

To show that mergesort correctly sorts the array, we use induction on the size of the array.

- ▶ **Base case:** if the array has one element, it is already sorted.
- ▶ **Inductive hypothesis:** assume that mergesort correctly sorts all arrays of size up to k .
- ▶ **Inductive step:** Consider an array of size $k+1$.
 - The two recursive calls sort the two halves of the array, each of size at most k . By the inductive hypothesis, these two halves are sorted correctly.
 - Merging two sorted arrays produces a sorted array.
 - Therefore, mergesort correctly sorts the array of size $k+1$.
- ▶ By induction, the algorithm is correct for all array sizes.

Mergesort: time complexity

Merging sorted arrays $A[\ell..m]$ and $A[m+1..r]$ takes time $O(r - \ell)$

- ▶ Each element of the two subarrays is copied exactly once into the temporary array B , and each such operation takes $O(1)$ time.
- ▶ Thus, the total time taken is linear in the number of elements copied, which is $r - \ell + 1 \subseteq O(r - \ell)$.
- ▶ Overwriting $A[\ell..r]$ with B also takes $O(r - \ell)$ time.
- ▶ Therefore, the merge operation runs in time $O(r - \ell)$.

Mergesort has time complexity $O(n \log_2 n)$.

- ▶ 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.

Note: We'll usually omit the subscript of $\log_2$.

Stable Matching Problem

- ▶ Suppose you are running an individual project course at CSE, with n student developers and n clients.
- ▶ Every developers submits a list of preferences, which ranks all the clients from their most preferred to least preferred, **and vice versa**.
- ▶ We'd like to make n separate pairs in order to “make everyone happy” in some sense.

Stable Matching Problem

Definition (matching)

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

Definition (perfect matching)

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

Our task is to design a perfect matching that somehow satisfies the developers and clients, with regards to their preferences.

Question

Can we assign every party their first preference partner?

Answer

Not necessarily!

- ▶ We'll have to lower our expectations. We would still like to guarantee some fairness.
- ▶ If a developer and a client were both very unhappy with their allocated partner, they might quit our system and work together externally.

Stable Matching Problem

Definition (stable matching)

A **stable matching** is a perfect matching in which there are no two pairs (d_1, c_1) and (d_2, c_2) such that:

- ▶ developer d_1 prefers client c_2 to c_1 , and
- ▶ client c_2 prefers developer d_1 to d_2 .

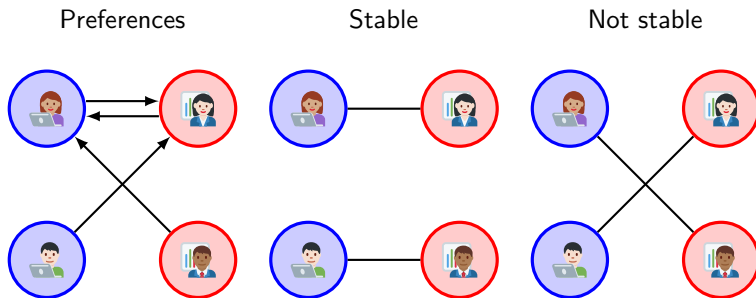
Stable matchings are self-enforcing; no **unmatched** pair of developer d_1 and client c_2 will quit together.

We will design an algorithm that produces a stable matching.

Stable Matching Problem: Example 1

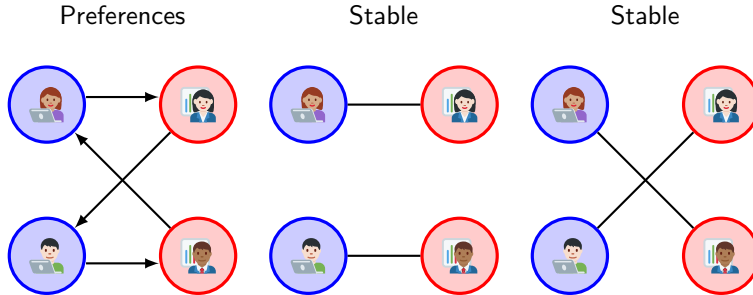
Each developer (Daphne  and David ) ranks each client (Cynthia  and Christos ) from most preferred to least preferred, and vice-versa.

Daphne  :	Cynthia  ,	Christos 		Cynthia  :	Daphne  ,	David 
David  :	Cynthia  ,	Christos 		Christos  :	Daphne  ,	David 



Stable Matching Problem: Example 2

Daphne : Cynthia , Christos  | Cynthia : David , Daphne 
 David : Christos , Cynthia  | Christos : Daphne , David 



Stable Matching Problem: Gale-Shapley algorithm

Question

How many ways are there to match n developers and n clients, without regard for preferences?

Answer

$n! \approx (n/e)^n$ with $e \approx 2.718$: more than exponentially many in n .

Question

Is there a stable matching for every collection of $2n$ preferences provided by all clients and developers?

Answer

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
- ▶ Developers make pitches to clients. Clients will decide to reject or tentatively accept a pitch.
- ▶ A developer who is not currently in a pair will be called **unemployed**.
- ▶ Start with all developers unemployed.

Stable Matching Problem: Gale-Shapley algorithm

Outline of algorithm

- ▶ While there is a unemployed developer who has not pitched to all clients, pick any such unemployed developer, say David .
- ▶ David  pitches to the highest ranking client Christos  on his list, ignoring any client who he has already pitched to.
 - If Christos  is not yet paired, he accepts David's  pitch (at least tentatively).
 - Otherwise, if Christos  is already paired with someone else, say Daphne :
 - if Christos  prefers David  to Daphne , he turns down Daphne  and makes a new pair with David  (Daphne  becomes unemployed)
 - otherwise, Christos  turns down David  and stays with Daphne .

Stable Matching Problem: Gale-Shapley algorithm

Question

Is the algorithm correct? Is it efficient?

To answer this, we will prove 3 claims:

Claim 1

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

Claim 2

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

Claim 3

The matching produced by the algorithm is stable.

Stable Matching Problem: Gale-Shapley algorithm

Claim 1

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

Proof.

- ▶ In each round of the algorithm, one developer pitches to one client.
- ▶ Each developer can make a pitch to a particular client at most once.
- ▶ Thus, each developer can make at most n pitches.
- ▶ There are n developers, so in total they can make $\leq n^2$ pitches.
- ▶ Thus, there can be no more than n^2 rounds.



Stable Matching Problem: Gale-Shapley algorithm

Claim 2

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

Proof.

- ▶ For the sake of contradiction, assume that the algorithm has terminated, but a developer **Daphne**  is unemployed.
- ▶ This means that Daphne  has already pitched to all the clients and was rejected each time.
- ▶ A client is not paired only if no-one has pitched to them. Since Daphne  has pitched to everyone, all n clients must be paired.
- ▶ Since no developer can have more than one partner, all n developers must also be paired, including Daphne . So, **Daphne**  is not unemployed.
- ▶ This is a contradiction, completing the proof.



Stable Matching Problem: Gale-Shapley algorithm

Claim 3

The matching produced by the algorithm is stable.

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

Stable Matching Problem: Gale-Shapley algorithm

Proof (1/2).

We use a proof by contradiction to show that the matching is stable.

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

- ▶ David  and Cynthia , and
- ▶ Daphne  and Christos ,

such that:

- ▶ David  prefers Christos  over Cynthia  and
- ▶ Christos  prefers David  over Daphne .

Since David  prefers Christos  over Cynthia , he must have made a pitch to Christos  before Cynthia .

Stable Matching Problem: Gale-Shapley algorithm

Proof (2/2).

David : ..., Christos , ..., Cynthia  (circled), ...

Christos : ..., David , ..., Daphne  (circled), ...

- ▶ Christos  must have either:
 - rejected David  because he was already paired with a more preferred developer, or
 - accepted David  only to later rescind this and accept a pitch from someone he prefers to David .
- ▶ In both cases Christos  is paired with someone more preferred than David .
- ▶ However he ends up with Daphne , who he prefers even less than David .
- ▶ This is a contradiction, completing the proof.



Stable Matching Problem: Gale-Shapley algorithm

Claims 1–3 establish that the Gale-Shapley algorithm is correct terminates after $\leq n^2$ rounds.

? **Exercise:** Can you set up data structures so that each round takes $O(1)$ time?



That's All, Folks!!