



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

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

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

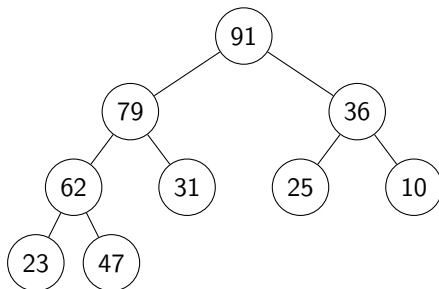
1. Time Complexity
2. Review of Logarithms
3. Revision of Data Structures and Algorithms
4. Problem Solving by Reduction

(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

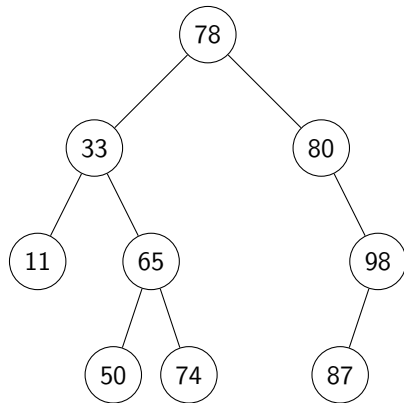
Binary search trees

- ▶ 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

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

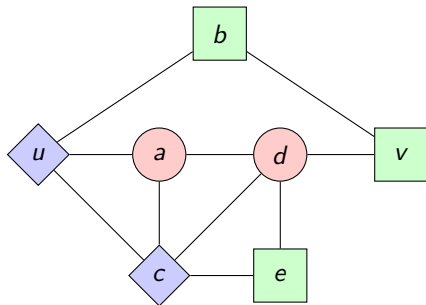
Problem

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.

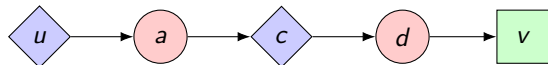
Given the graph $G = (V, E)$ and two distinct vertices $u, v \in V$, design an algorithm that computes the length of the shortest colourful path from u to v .

Example problem: Shortest Colourful Path

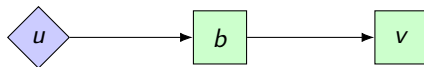
For example, consider the graph G below.



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.

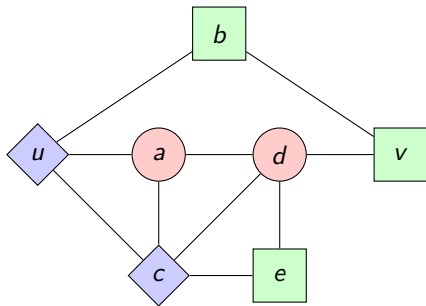


Example problem: Shortest Colourful Path

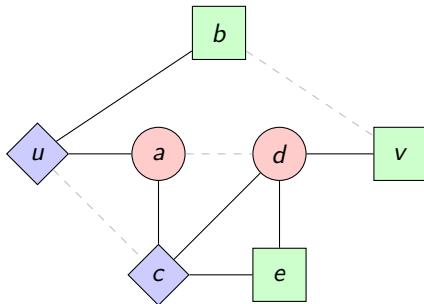
- ▶ If not for the colours, the problem would be to find the shortest path in an unweighted graph.
 - Recall that this is solved by Breadth-First Search (BFS).
- ▶ We cannot apply BFS immediately here because it may use illegal edges, i.e. edges that do not represent a change in colour.
- ▶ There are 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?

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

- ▶ 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.

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.

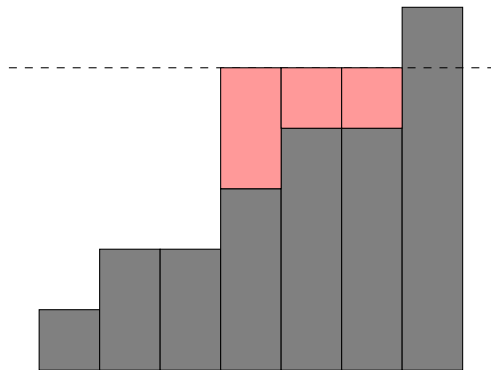
- ▶ But we still don't know how much to increase the numbers in the second half!
- ▶ What if we did know? What if we had a target to aim for, and we just had to report whether it was possible or not?

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.

Problem solving by reduction

- ▶ 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.