Notes for CSC 225 Labs - Summer 2014

Bill Bird
Department of Computer Science,
University of Victoria
Victoria, BC, Canada
bbird@uvic.ca

May 9, 2014

Contents

Part I

Algorithm Analysis

1 Introduction

For most students, CSC 225 is the first course that explores the theoretical aspects of computer science. While previous courses (such as introductory programming or architecture courses) may tangentially refer to theoretical results, CSC 225 is focused on providing a theoretical basis to the problems and techniques covered in first and second year computer science courses, and providing a foundation for more advanced courses.

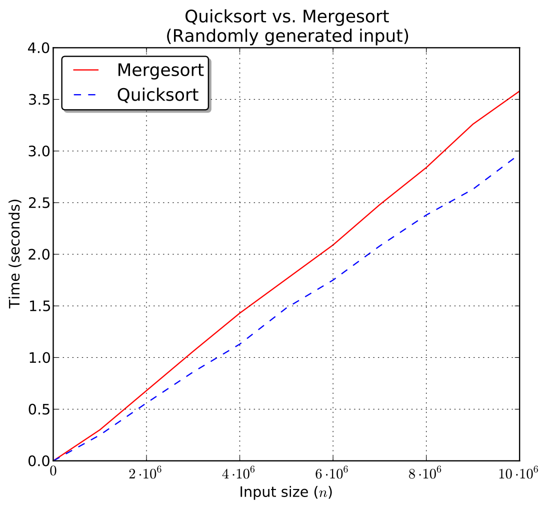
Programming skills and knowledge of particular architectures can be easily obtained and demonstrated without a college or university credential. The goal of advanced education in Computer Science is to produce well-rounded professionals whose skills are flexible and not dependent on specific (and potentially ephemeral) trends in the field. While some applications and platforms (such as Unix, the C programming language, variants of the `x86` architecture, and the TCP/IP networking stack) have been relatively long-lived, many aspects of Computer Science practice change constantly, and it can be dangerous to invest too much time into what may become passing fads in the industry (unless, of course, you are paid to do so). The advantage of the analysis skills taught in this course is that they are universally applicable to all problems within the currently-accepted model of computing, independent of language, platform or application. The algorithms and data structures studied in this course are similarly language and platform agnostic.

2 Basic Algorithm Analysis

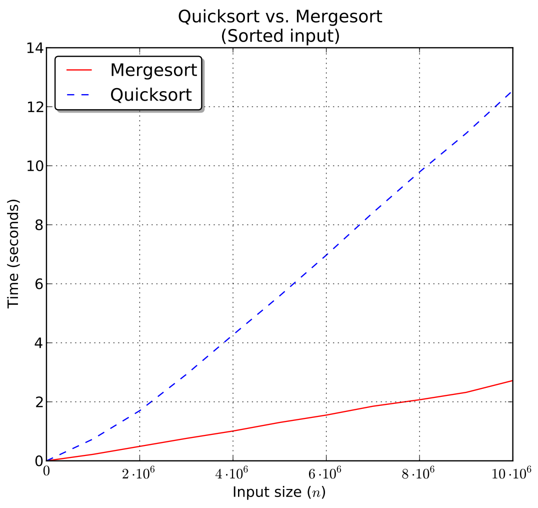
Prior to taking an algorithms course, most students have an intuitive grasp of the concepts of running time and memory consumption, and may even use analysis techniques without realizing it. While the formal notation introduced in this course can be considered a natural extension of the intuitive analysis used by most programmers (even at an introductory level), it can be difficult to adapt to such a rigorous model.

2.1 The Trouble with Timings

To measure the practical performance of an algorithm, the running time of the algorithm on a representative set of test inputs can be measured. This is a form of scientific experiment, and to ensure that two algorithms can be compared accurately, it is necessary to control all variations in the testing environment: the timings of different algorithms can only be compared objectively if they are run on the same machine with the same test inputs.



(a) Randomly generated input data



(b) Sorted input data

Figure 1: Plots of timing observations of an in-place Quicksort implementation and a Mergesort implementation on the same input data (consisting of lists of n integers).

The plots in figure 1 summarize timing data collected from two sorting implementations on two input datasets. The underlying algorithms used are Quicksort and Mergesort, both of which are widely used in practical applications (and will be studied later in the course). The first dataset, containing randomly-generated lists of integers, simulates the 'average' input a sorting algorithm might be expected to process. Both algorithms perform well on the randomly-generated data. The second dataset contains lists of integers sorted in descending order. While the Mergesort implementation has similar performance on both datasets, the Quicksort implementation suffers from a dramatic drop in performance on the second dataset.

The benchmark of figure 1 illustrates several of the reasons that timing data is inadequate to gauge the general performance or suitability of a given algorithm. Since the timings were collected from a specific implementation running on a specific machine, it is not possible to tell whether the drop in performance on the sorted data is due to bad programming, a hardware or architecture issue[1], or the algorithm itself. Moreover, it is not possible to generalize the performance of an algorithm on all possible inputs from data collected on a subset of possible inputs. The input data used in figure 1b was selected to showcase a known weakness of the Quicksort algorithm. In general, weaknesses like this can only be found through direct analysis of the algorithm, not through experimental observation.

Like any scientific experiment, a timing benchmark only provides observations, not absolute truth or certainty. Additionally, the observed timings of a particular implementation of an algorithm on a particular platform do not generalize to other implementations on other platforms. As a result, timing information is generally unreliable for a broad comparison of 'performance'. However, timing data can be very useful when choosing between different implementations of an algorithm or different architectures.

1. In some cases, a 'fast' or 'efficient' algorithm can suffer from a performance drop on certain hardware due to architectural limitations. For example, certain patterns of memory access can dramatically reduce virtual memory or cache performance. In many cases, these patterns can be avoided by a careful programmer, and therefore are not a fault of the underlying algorithm. Optimizations for memory performance are discussed in detail in architecture and compiler courses.

To measure algorithms independent of a specific architecture or implementation, we consider an ideal computer whose underlying architecture is similar to real computers, and classify algorithms by their (notional) performance on this machine. A defining feature of this machine is the ability to access all words of memory in the same amount of time, as opposed to having some memory locations take longer to access than others. Accordingly, the machine is called a 'Random-Access Machine'. Note that although modern computers are essentially random-access machines (since a program's data is stored in random-access memory), the virtual memory infrastructure of modern computers (which uses the hard drive to emulate extra RAM) is not necessarily random-access[2]. In general, position-related hard drive latency is negligible, so it is reasonable to call hard drives 'random-access' in this context. Historically, high-latency storage media (e.g. magnetic tape) was often needed for primary storage when very little core memory was available, which is why the distinction between random-access and non-random-access machines was necessary.

The random access machine is assumed to have a familiar system layout (that is, a Von Neumann architecture), and a CPU instruction set similar to modern machines. In CSC 225, we do not define the specific instructions more formally, since we can assume that all instructions are 'simple' and take a fixed (and small) amount of time. Instead, we think of algorithms in terms of a high-level 'pseudo-code' with a syntax similar to C and Java. It is important to keep in mind that pseudo-code is only a loosely defined notational convenience, and there is no 'standard' for pseudo-code. Some sources do define the specific instruction set for their ideal machine. One famous example is *The Art of Computer Programming* by D. E. Knuth, which defines a complete assembly language for a hypothetical machine called `MIX` (or `MMIX` in newer editions) and uses assembly listings for algorithm analysis.

2.2 Simple Control Flow Analysis

For the time being, all program analysis will be assumed to consider only the *worst case* time taken by an algorithm. While it can be useful to know the best case or expected case, the only indicator of whether an algorithm is 'airtight' in terms of efficiency is worst case performance.

Our first sample problem will be linear search for a value k over an unsorted integer array A of size n. If $A[i] = k$ for some index i, the algorithm returns i. Otherwise, the return value is -1.

Algorithm 1 Linear Search 1

```
procedure LinearSearch1($A, n, k$)
    for $i \leftarrow 0, \ldots, n-1$ do
        if $A[i] = k$ then
            return $i$
        end if
    end for
    return $-1$
end procedure
```

It should be clear that without any assurances about the structure of A (that is, assuming that

2. Specifically, a hard drive which uses physical disks is not random-access, since disk locations closer to the read-write head can be accessed faster than other locations. A solid-state hard drive does not have this limitation. Additionally, when a virtual memory system is used, accessing pages of memory stored on disk (instead of in RAM) incurs significant overhead. Distinctions like these are unreasonably pedantic for theoretical purposes, which is why an ideal machine is so convenient.

the elements of A are in an arbitrary order), it will never be possible to avoid looking at all n elements of A in the worst case. For algorithm 1, the worst case occurs when k is not an element of A.

To analyze algorithm 1 in terms of primitive operations, it is useful to decompose the `for` loop into a while loop, giving algorithm 2:

Algorithm 2 Linear Search 2

```
procedure LinearSearch2($A, n, k$)
    $i \leftarrow 0$
    while $i < n$ do
        if $A[i] = k$ then
            return $i$
        end if
        $i \leftarrow i + 1$
    end while
    return $-1$
end procedure
```

There is no performance difference between a `for` loop and an equivalent `while` loop in C and Java, and a compiler for either language will produce essentially identical code for both loops. The number of primitive operations in algorithm 2 in the worst case is $5n + 3$. A common mistake while counting primitive operations is to neglect the comparison made just before a loop terminates. In algorithm 2, the `while` loop iterates over the values $0, \ldots, n - 1$. However, the loop only terminates after the comparison $i < n$ fails, so a total of $n + 1$ comparisons are necessary.

The second example algorithm is binary search. Given an array A of n integers, which is assumed to be sorted in ascending order (that is, smaller values first), and a value k to search for, the binary search algorithm repeatedly bisects A until either k is found at the bisection point or no further bisections are possible.

Algorithm 3 Binary Search

```
procedure BinarySearch($A, n, k$)
    lo $\leftarrow 0$
    hi $\leftarrow n - 1$
    while lo $<$ hi do
        mid $\leftarrow \frac{\texttt{lo}+\texttt{hi}}{2}$
        if $A[\texttt{mid}] = k$ then
            return mid
        else if $A[\texttt{mid}] < k$ then
            lo $\leftarrow$ mid $+ 1$
        else
            hi $\leftarrow$ mid $- 1$
        end if
    end while
    return $-1$
end procedure
```

When an algorithm contains conditionals and assumptions are made about the results of comparisons for the analysis, it is important that these assumptions are justified. The worst case running time for algorithm 3 corresponds to the case where the `while` loop runs for as long as possible. Therefore, we will assume that the `Return` statement inside the loop is never executed, and, accordingly, that $A[\texttt{mid}]$ is never equal to k. This again corresponds to the case where k is not in A. Under this assumption, the conditional inside the `while` loop will either execute the assignment

$$\texttt{lo} \leftarrow \texttt{mid} + 1$$

or

$$\texttt{hi} \leftarrow \texttt{mid} - 1$$

Both of these statements require 2 primitive operations. Although we assume that $A[\texttt{mid}]$ never equals k, we still must perform the comparison $A[\texttt{mid}] = k$ at every iteration. This gives a total of 10 primitive operations inside the `while` loop (including the loop condition). There are 4 primitive operations outside the `while` loop.

Unlike the previous example, the number of iterations before the `while` loop terminates is not immediately clear to the reader. The loop condition is $\texttt{lo} < \texttt{hi}$, and since every value assigned to either variable lies between `lo` and `hi`, it will never be the case that $\texttt{lo} > \texttt{hi}$. Therefore, the loop will terminate only when $\texttt{lo} = \texttt{hi}$.

At each iteration, the size of the range $[\texttt{lo}, \texttt{hi}]$ of candidate indices has size $\texttt{hi} - \texttt{lo} + 1$, and at each bisection, the size of the range decreases by a factor of 2. When the range has size 1, $\texttt{lo} = \texttt{hi}$ and the loop terminates. The initial range is the entire array, which has size n. Therefore, the total number of iterations is logarithm in base 2 of n, rounded down if not an integer, denoted $\lfloor \log_2 n \rfloor$. In the context of algorithm analysis, the notation $\log n$ is often assumed to mean $\log_2 n$. For asymptotic notation (see section 3), it is not necessary to distinguish between different (constant) bases of logarithms.

Using the above argument, the total running time of algorithm 3 is found to be $10\lfloor \log_2 n \rfloor + 5$. It is important to remember that a valid analysis consists not only of a formula for the running time, but a valid and concise justification of any assumptions made. For the binary search algorithm, it is necessary to provide the complete justification of the $\log n$ iteration count of the `while` loop in addition to the final total.

2.3 Recursive Control Flow Analysis

The search algorithms in section 2.2 can also be implemented recursively. Algorithm 4 gives a recursive linear search algorithm. The parameter `start_idx` contains the index of the first array element to search.

Algorithm 4 Recursive Linear Search

```
procedure RecursiveLinearSearch($A, k$, start_idx)
    if start_idx $=$ length($A$) then
        return $-1$
    end if
    if $A$[start_idx] $= k$ then
        return start_idx
    else
        return RecursiveLinearSearch($A, n, k$, start_idx $+ 1$)
    end if
end procedure
{Initial Recursive Call - Not part of the algorithm itself}
RecursiveLinearSearch($A, k, 0$)
```

Since algorithm 4 is recursive, the number of primitive operations executed during an invocation of the algorithm cannot be obtained by simply counting the operations in each line. Instead, we denote the running time by an unknown function $T(n)$, where $n = \text{length}(A) - \texttt{start\_idx}$ is the number of array indices left to search. When $n = 0$ (i.e. nothing is left to search), the comparison $\texttt{start\_idx} = \text{length}(A)$ evaluates to true, causing a return statement to be executed. This requires 2 operations, so

$$T(0) = 2$$

As with the iterative linear search algorithm, the worst case occurs when k is not an element of A, so we assume that the comparison $A[\texttt{start\_idx}] = k$ always evaluates to false. Therefore, whenever $n \geq 1$, the recursive call is executed, taking $T(n-1)+1$ units of time. Adding the operations required to evaluate the two comparisons and addition before the recursive call, as well as the return statement, gives

$$T(n) = 6 + T(n-1)$$

for $n \geq 1$. We can evaluate $T(n)$ for any $n \geq 0$ by repeated substitution. Techniques for obtaining a closed form or asymptotic bound on recurrence relations will be studied later in the course.

Algorithm 5 gives a recursive binary search algorithm. The values `lo` and `hi`, which define the range of indices to search, are now parameters to the function.

Algorithm 5 Recursive Binary Search

```
procedure RecursiveBinarySearch($A, k,$ lo, hi)
    if lo > hi then
        return $-1$
    end if
    mid ← $\frac{\texttt{lo+hi}}{2}$
    if $A[\text{mid}] = k$ then
        return mid
    else if $A[\text{mid}] < k$ then
        return RecursiveBinarySearch($A, k,$ mid $+ 1,$ hi)
    else
        return RecursiveBinarySearch($A, k,$ lo, mid $- 1$)
    end if
end procedure
{Initial Recursive Call - Not part of the algorithm itself}
RecursiveBinarySearch($A, k, 0, \text{length}(A) - 1$)
```

We again denote the running time by $T(n)$, where $n = \texttt{hi} - \texttt{lo} - 1$ is the number of elements in the search range (which may not be the total size of the array). In the case where $n = 0$ (that is, $\texttt{lo} = \texttt{hi} + 1$, indicating that the search range is empty), only two primitive operations are required, since the conditional at the beginning of the algorithm causes the return statement to be executed. So we have

$$T(0) = 2$$

When $n \geq 1$, the condition $\texttt{lo} > \texttt{hi}$ evaluates to false. Both of the recursive calls to RecursiveBinarySearch take $1 + T(\lfloor \frac{n}{2} \rfloor)$ time[3], so the total number of primitive operations for $n \geq 1$ is

$$T(n) = 11 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

3 Asymptotic Notation

To compare the running times (or other performance metrics) of two algorithms, it is not sufficient to use regular inequalities such as $\leq$ or $\geq$. Consider two algorithms with running times $f(x)$ and $g(x)$, where

$$f(x) = x + 2$$
$$g(x) = e^x$$

From a graph of the two functions (see figure 2), it is clear that in the long run, $f(x)$ will be much smaller than $g(x)$. However, we cannot say that $f(x) \leq g(x)$ since $f(0) = 2 > 1 = g(0)$. We also cannot say that $f(x) \geq g(x)$, since $f(2) = 4 < e^2 = g(2)$.

3. When the array is bisected, the two parts may not have the same size. For example, an array of size $n = 6$ would be split into a part of size 3 and a part of size 2 (since the element `mid` is excluded). If the recursive call searches the part of size 2, the number of operations would be $T(2) = T(\lfloor \frac{n-1}{2} \rfloor)$, not $T(\lfloor \frac{n}{2} \rfloor) = T(3)$. However, since we're interested in the worst case time, we always assume that $T(\lfloor \frac{n}{2} \rfloor)$ operations are needed.

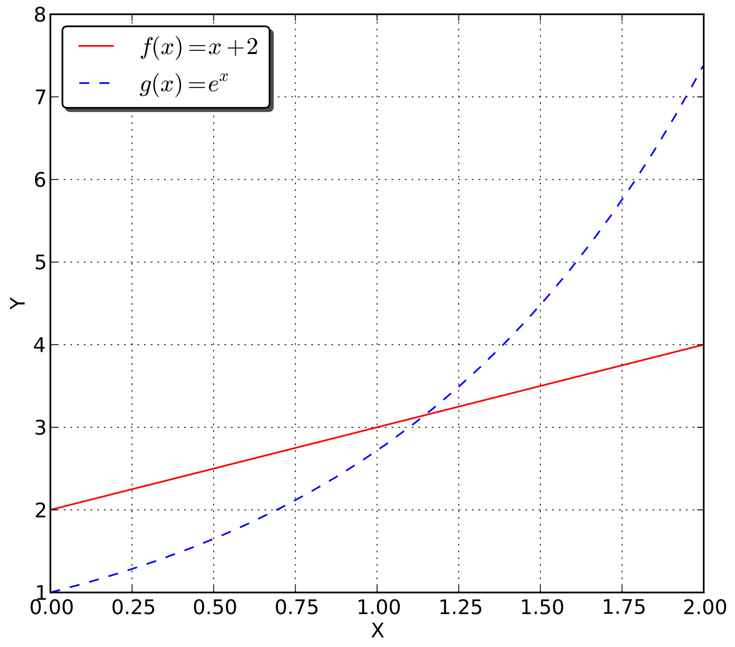


Figure 2: Graph of $x + 2$ and e^x over $[0, 2]$.

Asymptotic notation provides a mechanism for comparing the general behavior of functions as inputs become arbitrarily large. In the example above, the fact that $f(0) > g(0)$ is significant, but for large inputs, f is clearly much smaller than g, so we write $f(n) \in O(g(n))$ and say 'f is big-O of g'. Asymptotic notation is widely used in mathematics outside of computer science, since it is a very convenient and compact way to represent the scale of a quantity while ignoring unimportant details.

The table below summarizes the intuitive generalization of inequalities between numbers a, b to the asymptotic relationships of functions $A(n)$ and $B(n)$.

Inequality	Asymptotic Relationship	Name
$a < b$	$A(n) \in o(B(n))$	Little-O
$a \leq b$	$A(n) \in O(B(n))$	Big-O
$a = b$	$A(n) \in \Theta(B(n))$	Big-Theta
$a \geq b$	$A(n) \in \Omega(B(n))$	Big-Omega
$a > b$	$A(n) \in \omega(B(n))$	Little-Omega

3.1 Big-O

Big-O notation can be considered the asymptotic analogue of the $\leq$ operator. Formally, a function $f(n)$ is an element of $O(g(n))$ if and only if there exists a real constant $c > 0$ and a value n_0 such that for every $n \geq n_0$, $f(n) \leq cg(n)$.

Using this definition, we can prove that $x+2 \in O(e^x)$ by taking $c = 1$ and $n_0 = 2$. Note that it is not necessary to use the minimum possible n_0 to satisfy the definition, and it is often preferable to choose a more convenient value. For the functions $f(x) = x + 2$ and $g(x) = e^x$, the minimum possible n_0 is $\tilde{n}_0 \approx 1.146$, which cannot be expressed analytically, so it is easier to take $n_0 = 2$

when applying the definition[4].

For the analysis of algorithms, functions are normally assumed to take integer arguments. However, when the functions f and g are defined (and reasonably well-behaved) on the real numbers, an equivalent definition can be stated in terms of limits. Specifically, $f \in O(g(n))$ if the ratio between f and g is bounded as n goes to infinity.

If the limit

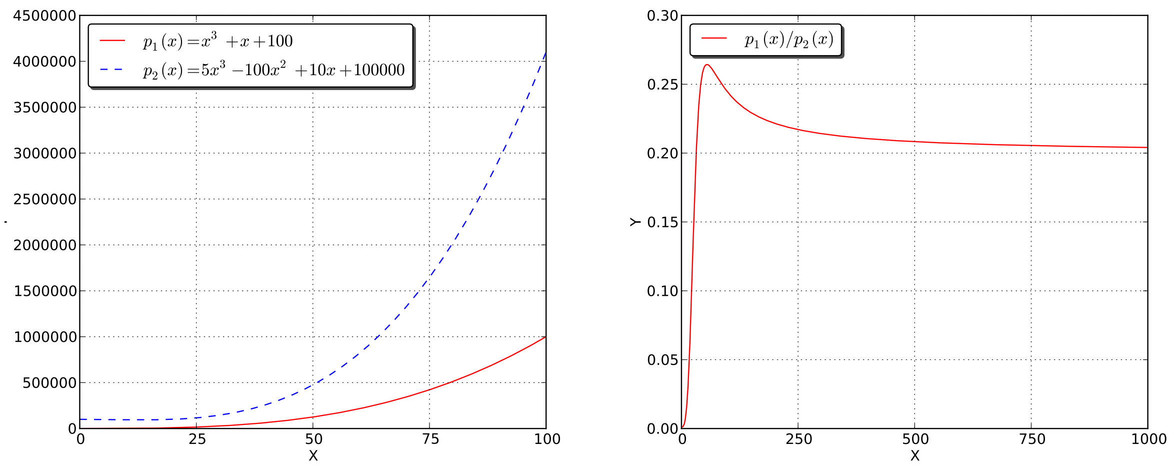
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists, then $f(n) \in O(g(n))$. This is not a necessary condition for big-O, however, since there are cases where $f(n) \in O(g(n))$ but the limit fails to exist[5]. Even so, this limit condition can be very convenient for establishing asymptotic bounds.

Using the limit condition, we can prove that $x + 2 \in O(e^x)$ by applying L'Hôpital's rule:

$$\begin{aligned}\lim_{x\to\infty} \frac{x+2}{e^x} &= \lim_{x\to\infty} \frac{\frac{d}{dx}(x+2)}{\frac{d}{dx}e^x} \\ &= \lim_{x\to\infty} \frac{1}{e^x} \\ &= 0\end{aligned}$$

This actually proves that $x + 2 \in o(e^x)$ (see section 3.4), which implies $x + 2 \in O(e^x)$.



(a) Function Values (b) Ratio

Figure 3: Graphs of two asymptotically equivalent polynomials.

Consider the two polynomials

$$\begin{aligned}p_1(x) &= x^3 + x + 100 \\ p_2(x) &= 5x^3 - 100x^2 + 10x + 100000\end{aligned}$$

4. We could also take n_0 to be any other value in $[\tilde{n}_0, \infty)$
5. A more advanced type of limit, the 'limit superior' of $\frac{f(n)}{g(n)}$ as $n \to \infty$, will exist if and only if $f(n) \in O(g(n))$.

Figure 3 plots these functions, and their ratio small values of x. While it is important to remember that plots prove nothing about a function's asymptotic behavior (or even its behavior in the plotted region), the plot in figure 3b does suggest that the ratio $p_1(x)/p_2(x)$ converges to a constant, which (if proven) would imply that the two functions are asymptotically equivalent.

We can prove that the ratio converges to a constant with the limit definition:

$$\begin{aligned}\lim_{x\to\infty}\frac{p_1(x)}{p_2(x)} &= \lim_{x\to\infty}\frac{x^3+x+100}{5x^3-100x^2+10x+100000}\\ &= \lim_{x\to\infty}\frac{3x^2+1}{15x^2-200x+10}\\ &= \lim_{x\to\infty}\frac{6x}{30x-200}\\ &= \lim_{x\to\infty}\frac{6}{30}\\ &= \frac{1}{5}\end{aligned}$$

This implies that $p_1(x) \in O(p_2(x))$. We can similarly prove that $p_2(x) \in O(p_1(x))$. These two relationships imply that $p_1(x) \in \Theta(p_2(x))$ and $p_2(x) \in \Theta(p_1(x))$ (see section 3.3).

Normally, we want to classify the asymptotic behavior of a function in the simplest terms possible. To classify an algorithm with running time $p_1(n)$ or $p_2(n)$, we would say that both are $O(n^3)$. We can prove this using either definition. For example, using $p_2(n)$ and the inequality definition,

$$\begin{aligned}5n^3-100n^2+10n+100000 &\le 5n^3+10n+100000 && \text{(Remove negative terms)}\\ &\le 5n^3+10n^3+100000n^3 && \text{(Change all terms to } n^3\text{)}\\ &= 100015n^3\end{aligned}$$

Taking $c = 100015$ and $n_0 = 1$, we have $p_2(n) \le cn^3$ for $n \ge n_0$, giving $p_2(n) \in O(n^3)$.

The hierarchy of asymptotic classes commonly encountered in computer science is summarized below.

$$\begin{aligned}O(0) &\subseteq O(1)\\ O(1) &\subseteq O(\log^* n)\\ O(\log^* n) &\subseteq O(\log n)\\ O(\log n) &\subseteq O(n^k) \text{ for } k>0\\ O(n) &\subseteq O(n\log n)\\ O(n\log n) &\subseteq O(n^k) \text{ for } k>1\\ O(n^k) &\subseteq O(n^{k+\varepsilon}) \text{ for } k>0, \varepsilon>0\\ O(n^k) &\subseteq O(q^n) \text{ for } q>1\\ O(q^n) &\subseteq O(n!)\\ O(n!) &\subseteq O(n^n)\end{aligned}$$

3.2 Big-Omega

Big-Omega is the asymptotic analogue of the $\ge$ operator. The following statements are equivalent:

- $f(n) \in \Omega(g(n))$
- There exist constants c and n_0 such that for all $n \geq n_0$, $f(n) \geq cg(n)$
- $g(n) \in O(f(n))$ (This is analogous to the fact that $a \geq b$ is equivalent to $b \leq a$).

3.3 Big-Theta

Big-Theta is the asymptotic analogue of the $=$ operator. The following statements are equivalent:

- $f(n) \in \Theta(g(n))$
- There exist constants c_1, c_2 and n_0 such that for all $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
- $g(n) \in \Theta(f(n))$

3.4 Little-O

Little-O is the asymptotic analogue of the $<$ operator. The following statements are equivalent:

- $f(n) \in o(g(n))$
- For every constant $c > 0$, there exists an n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$
- The limit
$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$
converges to 0. Unlike big-O, the limit condition is a necessary condition for little-O, so if the limit does not exist or converges to a non-zero value, then $f(n)$ is not $o(g(n))$.
- $f(n) \in O(g(n))$ and $f(n) \notin \Theta(g(n))$

3.5 Little-Omega

Little-Omega is the asymptotic analogue of the $>$ operator. The following statements are equivalent:

- $f(n) \in \omega(g(n))$
- For every constant $c > 0$, there exists an n_0 such that for all $n \geq n_0$, $g(n) \leq cf(n)$
- The limit
$$\lim_{n \to \infty} \frac{g(n)}{f(n)}$$
converges to 0. Similar to little-O, the limit condition is necessary for little-Omega.
- $g(n) \in O(f(n))$ and $g(n) \notin \Theta(f(n))$
- $g(n) \in o(f(n))$

4 Recursive Algorithm Analysis

To revisit the analysis of recursive algorithms, we will consider algorithms to evaluate two mathematical sequences, both of which can be defined either with a closed form or a recurrence relation.

4.1 Fibonacci Sequence

The Fibonacci sequence is defined by the recurrence

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n \geq 2 \end{aligned}$$

Code to evaluate this sequence is often used as an introductory example of recursion in programming courses. We will consider an iterative and a recursive method to evaluate F_n for a given n. Algorithm 6 describes the iterative method.

Algorithm 6 Fibonacci Numbers (iterative)

```
procedure FibonacciIterative($n$)
    if $n = 0$ then
        return 0
    end if
    last2 ← 0
    last1 ← 1
    for $i \leftarrow 2, \ldots, n$ do
        next ← last1 + last2
        last2 ← last1
        last1 ← next
    end for
    return last1
end procedure
```

For $n \geq 2$, the number of primitive operations required by algorithm 6 is $6 + 7(n-1) = 7n - 1$, which is clearly $\Theta(n)$. The recursive method is given in algorithm 7.

Algorithm 7 Fibonacci Numbers (recursive)

```
procedure FibonacciRecursive($n$)
    if $n = 0$ then
        return 0
    else if $n = 1$ then
        return 1
    end if
    return FibonacciRecursive($n - 1$) + FibonacciRecursive($n - 2$)
end procedure
```

Denoting the operation count of algorithm 7 by $T(n)$, we can obtain the unsurprising result

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 2 \\ T(n) &= 8 + T(n-1) + T(n-2) \qquad \text{for } n \geq 2 \end{aligned}$$

It should be clear that $F_n \leq T(n)$ for all $n \geq 0$. Although this may seem obvious, it would still be necessary to prove it (for example, with induction), before using it in any formal justifications of the running time of algorithm 7. Since $F_n \leq T(n)$, the running time of algorithm 7 on an input n is at least F_n for sufficiently large n (we can also say that $T(n) \in \Omega(F_n)$, since $F_n \leq T(n)$ is a stronger condition).

It can be shown (using techniques beyond the scope of CSC 225) that F_n has the closed form

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

and consequently, that

$$\lim_{n\to\infty} F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

which gives

$$F_n \in \Theta(\varphi^n)$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Since $T(n) \in \Omega(F_n)$, we have $T(n) \in \Omega(\varphi^n)$, so the running time of algorithm 7 is at least exponential.

4.2 Binomial Coefficients

The binomial coefficient

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

is the number ways to select an unordered combination of size k from a collection of n distinct objects. To evaluate these values precisely, we must use integer arithmetic, since floating point computation can introduce error into the result. First, we can simplify the closed form above slightly, by observing that $\frac{n!}{(n-k)!} = n(n-1)\dots(n-k+1)$, giving

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k!}$$

An algorithm to evaluate binomial coefficients with this closed form will take $\Theta(n)$ time. However, even when the result is relatively small, arithmetic overflow may occur when evaluating the numerator, and there is no straightforward way to avoid evaluating the entire numerator before doing any division (since we are using integer division, we cannot allow intermediate results to have a fractional part). For example,

$$\binom{18}{9} = \frac{18 \cdot 17 \cdot \ldots \cdot 10}{9!} = 48620$$

which is obviously a manageable quantity, but

$$18 \cdot 17 \cdot \ldots \cdot 10 = 17643225600$$

which exceeds 2^{32}, so an overflow would occur in an implementation using 32-bit unsigned arithmetic[6].

It can be shown that $\binom{n}{k} < 2^n$, so it is reasonable to expect that an algorithm to evaluate binomial coefficients with 32-bit arithmetic should produce correct results for all $n \leq 32$, but using the closed form directly does not guarantee this, as the example above illustrates. To

6. Obviously, we could always use 64- or 128-bit arithmetic, but the same problem would still occur for larger values of n and k

remedy this problem, we consider two alternative algorithms whose intermediate computations will never exceed the final result, guaranteeing that if the binomial coefficient can be stored without overflow, the algorithms will produce the correct value.

Binomial coefficients can also be defined by a recurrence,

$$\begin{aligned}\binom{n}{0} &= \binom{n}{n} = 1\\ \binom{n}{k} &= \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for } 1 \le k < n\end{aligned}$$

Algorithm 8 gives a recursive method for evaluating this recurrence.

Algorithm 8 Binomial Coefficients (recursive)

```
procedure BinomialRecursive($n, k$)
    if $n = k$ or $k = 0$ then return 1
    end if
    return BinomialRecursive($n - 1, k - 1$) + BinomialRecursive($n - 1, k$)
end procedure
```

Direct analysis of algorithm 8 yields a recurrence relation for the running time $T(n, k)$ in terms of primitive operations. As with the Fibonacci algorithms in section 4.1, the recurrence is very similar to the recurrence for binomial coefficients

$$\begin{aligned}T(n, n) &= T(n, 0) = 4\\ T(n, k) &= 10 + T(n - 1, k - 1) + T(n - 1, k) \qquad \text{for } 1 \le k < n\end{aligned}$$

Rather than attempt to solve this recurrence, we will just observe that $T(n, k) \ge \binom{n}{k}$ for all $n, k \ge 0$. A proof of this fact can be obtained by using induction and comparing the recurrences. This gives

$$T(n, k) \in \Omega\left(\binom{n}{k}\right)$$

It can be shown that $\binom{n}{k} \in \Omega(2^n)$, which implies that algorithm 8 is exponential, however the mathematical techniques to prove this are beyond the scope of the course. For later comparison, we can easily show that $T(n, k) \in \Omega(n^2)$, since $\binom{n}{2} = \frac{n(n-1)}{2} \in \Theta(n^2)$ and $T(n, k) \in \Omega(\binom{n}{2})$.

Unlike the Fibonacci sequence, we cannot compute binomial coefficients with a 'moving window' approach like that used in algorithm 6. However, we can consider Pascal's Triangle, which is a triangular array where row n contains the binomial coefficients $\binom{n}{0}$ through $\binom{n}{n}$, and each entry is the sum of the two entries above it. The first 5 rows of Pascal's Triangle are given below.

$$\begin{array}{lccccccccc|ccccccccc}
n = 0: & & & & & \binom{0}{0} & & & & & & & & & 1 & & & & \\
n = 1: & & & & \binom{1}{0} & & \binom{1}{1} & & & & & & & 1 & & 1 & & & \\
n = 2: & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & & 1 & & 2 & & 1 & & \\
n = 3: & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & 1 & & 3 & & 3 & & 1 & \\
n = 4: & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & 1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

Since each entry can be computed from previous entries with a single addition, and since the first n rows of Pascal's triangle can be stored in an $n \times n$ array, we can compute a binomial coefficient $\binom{n}{k}$ in $\Theta(n^2)$ time with an iterative algorithm such as algorithm 9. Since the triangular shape of Pascal's triangle is unnatural for tabular representation, the algorithm stores each row 'left-justified', and ignores all array elements after the end of each row. For example, the first five rows would be stored as

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

Algorithm 9 Binomial Coefficients (iterative)

```
procedure BinomialIterative($n, k$)
    Create an $(n+1) \times (n+1)$ array $A$.
    $A[0][0] \leftarrow 1$
    for $i \leftarrow 1, \ldots, n$ do
        $A[i][0] \leftarrow 1$
        $A[i][i] \leftarrow 1$
        for $j \leftarrow 1, \ldots, i-1$ do
            $A[i][j] \leftarrow A[i-1][j-1] + A[i-1][j]$
        end for
    end for
    return $A[n][k]$
end procedure
```

Analysis of algorithm 9 gives a running time asymptotic to

$$\sum_{i=1}^{n} i$$

which is $\Theta(n^2)$ (see section 9).

Most of the complexity in algorithm 9 lies in the computation of the table of values. When similar methods are used to compute binomial coefficients in real applications, the table is often pre-computed and stored, so that the values can be retrieved in constant time. The $(n+1) \times (n+1)$ array requires $\Theta(n^2)$ space, and for large enough n, it may be impossible to store the entire array in memory. Since only the last row is needed, it is possible to modify algorithm 9 to require $\Theta(n)$ space by storing only one row at a time.

5 The Tower of Hanoi

The famous Tower of Hanoi puzzle consists of a set of N disks of increasing diameter stacked on one of three pegs, with the goal being to move the entire stack to another peg, one disk at a time, without ever placing a disk on top of a smaller disk. Figure 4 shows the steps of a solution for the case $N = 3$.

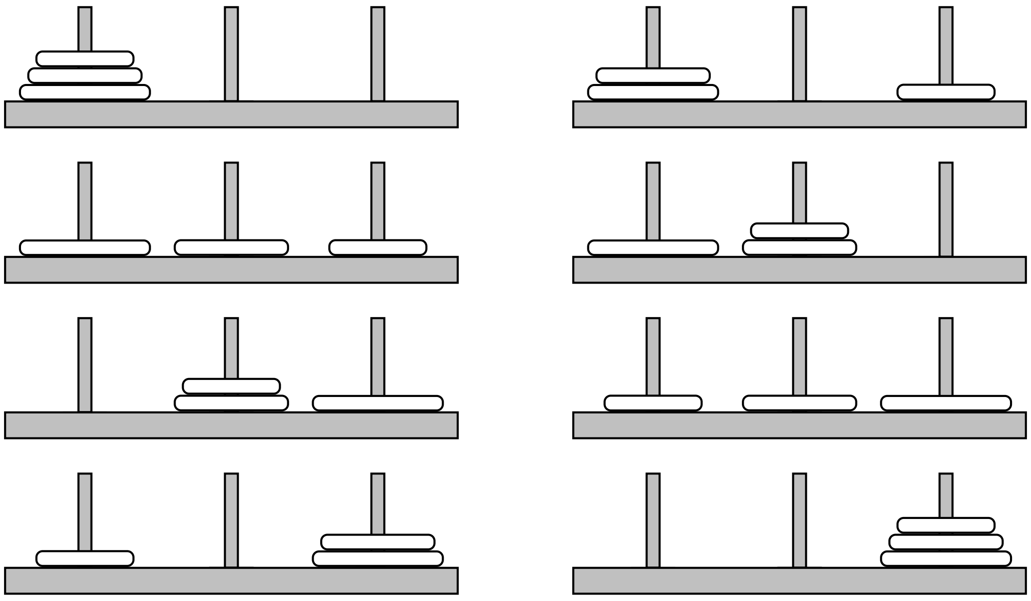
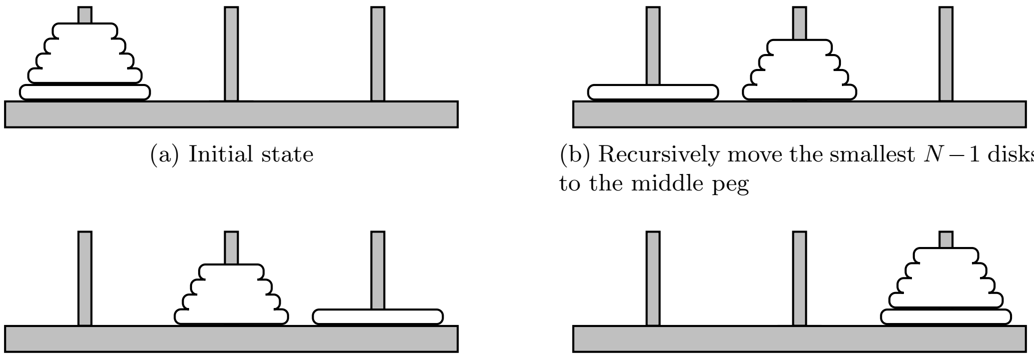


Figure 4: Steps of a solution to the Tower of Hanoi with 3 disks.

An algorithm to solve the Tower of Hanoi problem must produce a correct solution for all values of N. A solution is considered 'correct' if the entire stack of N disks is moved to the destination peg and only legal moves are made (that is, only one disk is moved at a time, and no disk is ever placed on top of a smaller disk). The only way to assert that an algorithm is correct for all input values is to produce a mathematical proof. Beyond the correctness constraint, it is also desirable that the produced solution use as few moves as possible and that the algorithm uses as few computational resources (e.g. time and memory) as possible.

Recursion often provides a convenient way to address all three criteria. Since recursive algorithms tend to be very concise (due to the underlying stack operations being handled implicitly by recursive calls), it can be much easier to write proofs about recursive algorithms than their iterative equivalents. The Tower of Hanoi problem has an elegent recursive formulation which is often presented in first-year programming courses. There are also various iterative solutions, but the proofs for iterative methods are more involved and often require more advanced mathematical knowledge[7]. The diagrams in figure 5 summarize the recursive solution. The first $N-1$ disks on the initial stack of N disks are first moved to the middle peg, allowing the N^{th} disk to be moved to the destination peg before the $N-1$ smaller disks are stacked on top of it.

7. These iterative solutions are often discussed in upper-level courses in combinatorics, coding theory and algorithm analysis.



(a) Initial state

(b) Recursively move the smallest $N-1$ disks to the middle peg

(c) Move the bottom disk to the third peg

(d) Recursively move the smallest $N-1$ disks to the third peg

Figure 5: Outline of the recursive solution to the Tower of Hanoi problem.

Algorithm 10 Tower of Hanoi Solution

```
procedure MoveTower(tower_size, source_peg, dest_peg, temp_peg)
    if tower_size = 0 then
        return 0
    end if
    {Move the first tower_size − 1 disks onto the temporary peg.}
    count ← MoveTower(tower_size − 1, source_peg, temp_peg, dest_peg)
    {Now move the largest disk from the source peg to the destination.}
    $d$ ← Pop(source_peg)
    Push $d$ onto dest_peg
    count ← count + 1
    Output the current state of each peg
    {Now move the first tower_size − 1 disks onto the largest disk.}
    {The original source peg is provided to the recursive invocation as temporary storage.}
    count ← count + MoveTower(tower_size − 1, temp_peg, dest_peg, src_peg)
    return count
end procedure
procedure TowerOfHanoi($N$)
    {Initialize each peg.}
    Let Peg1, Peg2, Peg3 be empty stacks.
    {Put all of the disks on the initial peg.}
    Push the sequence $N, N-1, N-2, \ldots, 2, 1$ onto Peg1
    {Move the entire tower from peg 1 to peg 3, using peg 2 as temporary storage.}
    totalMoves ← MoveTower($N$, Peg1, Peg3, Peg2)
    Output totalMoves
end procedure
```

Algorithm 10 gives pseudocode to solve the Tower of Hanoi problem for any N, outputting the state of each peg at the end of every move and the total number of moves made when the algorithm finishes. The MoveTower function moves the first `tower_size` disks on a given source peg to a given destination peg and returns the total number of moves. In this implementation, the pegs are represented by stacks, and each disk is represented by an integer denoting its size.

The MoveTower function assumes that the pegs provided to it contain legal sequences of disks, and that the first `tower_size` on the source peg are the smallest disks among the entire collection (that is, there are no disks on any other pegs which are smaller than any of the disks being moved). If these assumptions are satisfied, then the move made inside MoveTower is always a legal move.

5.1 Proof of Correctness

To prove that algorithm 10 always produces a correct solution, we must assert that it always finishes (since if the algorithm enters an infinite loop, it will never generate a solution) and that it never makes an illegal move. A proof of correctness can take several forms, but in this case, an inductive proof is a simple and concise approach. Induction is often a natural fit for proofs of results involving recursion.

We will make use of the two assumptions made about the input to MoveTower in the previous section:

1. Every input peg contains a legal sequence of disks (larger disks are below smaller disks).
2. The first `tower_size` disks on the source peg are the smaller than any disks on the other peg.

Basis

When `tower_size` $= 0$, the algorithm simply terminates, which is sufficient since moving a tower of size 0 requires no moves.

Induction Hypothesis

Suppose MoveTower correctly moves a tower of size `tower_size` $= n$ for some $n \geq 0$.

Induction Step

Consider $n + 1$.

By the induction hypothesis, the two recursive calls in MoveTower correctly move the first n disks from the source peg to the temporary peg, and from the temporary peg to the destination peg. It is therefore sufficient to establish that the move of disk $n + 1$ from the source peg to the destination peg is legal. When the move is made, the state of the destination peg is the same as it was when the MoveTower function was called (although the destination peg may have been used by the first recursive call as temporary storage, the induction hypothesis implies that when the recursive call finished, the destination peg's initial state had been restored), and by assumption 2 above, all of the disks on the destination peg are larger than disk $n + 1$. Additionally, by assumption 1, all of the disks that haven't been moved by MoveTower are in legal positions. Therefore, the move of disk $n + 1$ is legal, so MoveTower correctly moves a tower of size $n + 1$.

Therefore, by induction, MoveTower correctly moves all towers of size $n \geq 0$.

5.2 Number of Moves in the Solution

When `tower_size` $= 1$, MoveTower takes a constant amount of time[8], since both of the recursive calls are for towers of size 0. Whenever `tower_size` ≥ 1, exactly one disk is moved directly. Therefore, the running time of the algorithm is proportional to the total number of moves made (plus a constant to cover recursion overhead), so the asymptotic performance of the MoveTower function in algorithm 10 can be determined by finding a formula for the

8. In general, when the input of an algorithm is fixed, it takes a fixed (constant) amount of time, since the running time is a function of the input.

total number of disk moves, which is conceptually simpler than counting operations in the pseudocode.

Denoting the total number of moves made to solve the Tower of Hanoi for a tower of size n by $M(n)$, and given the base case

$$M(0) = 0$$

the recursive structure of algorithm 10 yields the recursive form

$$M(n) = 2M(n-1) + 1$$

for all $n \geq 1$.

The running time $T(n)$ of MoveTower is then $\Theta(M(n)+1)$, where the extra 1 ensures that $T(0)$ is $\Theta(1)$ instead of $\Theta(0)$.

A hypothetical closed form of the recurrence $M(n)$ can be obtained by repeated substitution:

$$\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2\left[2M(n-2) + 1\right] + 1 = 2^2 M(n-2) + 2 + 1 \\
&= 2^2\left[2M(n-3) + 1\right] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1 \\
&= 2^3\left[2M(n-4) + 1\right] + 2^2 + 2 + 1 = 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1 \\
&\;\;\vdots \\
&= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2^2 + 2^1 + 2^0 \\
&= 2^i M(n-i) + \sum_{j=0}^{i-1} 2^j
\end{aligned}$$

Setting $i = n$ in the last equation (to allow the base case $M(0) = 0$ to be substituted for the recursive use of M) gives

$$\begin{aligned}
M(n) &= 2^n M(0) + \sum_{j=0}^{n-1} 2^j \\
&= 2^n \cdot 0 + \sum_{j=0}^{n-1} 2^j \\
&= \sum_{j=0}^{n-1} 2^j
\end{aligned}$$

Using the identity

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

yields the closed form

$$M(n) = 2^n - 1$$

This closed form is only a hypothesis, since repeated substitution proves nothing by itself (since repeated substitution is only a heuristic tool to find patterns in expanded recurrences). To prove the correctness of the closed form, it is necessary to give a mathematical proof, such as the following induction proof that $M(n) = 2^n - 1$.

Basis

When $n = 0$, the base case of the recurrence is 0, which is equal to $2^0 - 1$.

Induction Hypothesis

Suppose $M(n) = 2^n - 1$ for some $n \geq 0$.

Induction Step

Consider $n + 1$.

$$\begin{aligned}
M(n+1) &= 2M(n) + 1 && \text{(By the definition of } M\text{)} \\
&= 2\left[2^n - 1\right] + 1 && \text{(By the induction hypothesis)} \\
&= 2^{n+1} - 2 + 1 \\
&= 2^{n+1} - 1
\end{aligned}$$

The final equation corresponds to the hypothesized closed form of $M(n + 1)$. Therefore, the closed form $2^n - 1$ is equal to $M(n)$ for all $n \geq 0$ by induction.

The running time of MoveTower is therefore $\Theta(M(n) + 1) = \Theta(2^n - 1 + 1) = \Theta(2^n)$ for all $n \geq 0$.

6 Selection Sort

Sorting is a fundamental problem in computer science. One important class of sorting algorithms are the *comparison sorts*, which sort a sequence of values $a_1, \ldots, a_n$ using only binary comparisons such as $a_i \leq a_j$. Comparison sorting can be used for any type of object for which a well-behaved $\leq$ operator can be defined[9].

Selection sort is a simple comparison sorting algorithm which builds a sorted array by repeatedly moving the minimum element of the unsorted part of the array to the front. The move operation is normally implemented with a swap rather than a vector insertion. The behavior of the algorithm on a sample array is conceptualized in the diagram below.

Iteration	Unsorted Array					
0	9	16	1	25	4	36
1	**1**	16	9	25	4	36
2	**1**	**4**	9	25	16	36
3	**1**	**4**	**9**	25	16	36
4	**1**	**4**	**9**	**16**	25	36
5	**1**	**4**	**9**	**16**	**25**	36
6	**1**	**4**	**9**	**16**	**25**	**36**
	Sorted Array					

An iterative method for selection sort is given in algorithm 11. The input array A and its size n are provided as parameters.

9. When the comparison relation is only a partial order, the 'sorted' representation of the sequence may not be unique. These 'sorted' representations of partially-ordered sets can be produced with *topological sorting*, which is covered later in the course in the context of graph algorithms.

Algorithm 11 Selection Sort (iterative)

```
procedure SelectionSortIterative($A, n$)
    for $i \leftarrow 0, \ldots, n-2$ do
        min $\leftarrow i$
        for $j \leftarrow i+1, \ldots, n-1$ do
            if $A[j] < A[\texttt{min}]$ then
                min $\leftarrow j$
            end if
        end for
        if min $\neq i$ then
            Swap $A[\texttt{min}]$ and $A[i]$
        end if
    end for
end procedure
```

Analysis of algorithm 11 gives a running time asymptotic to

$$\sum_{i=1}^{n-2} n-i-2 = \left(\sum_{i=1}^{n-2} n-i\right) - 2(n-1) = \sum_{i=2}^{n-1} i - 2n + 2 = \frac{n(n-1)}{2} - 2n + 1$$

which is $\Theta(n^2)$ (see section 9).

Selection sort can also be implemented recursively. Algorithm 12 gives a recursive formulation. The parameter `start_idx` tracks the first index in the unsorted part of the input array.

Algorithm 12 Selection Sort (recursive)

```
procedure SelectionSortRecursive($A$, start_idx, $n$)
    if start_idx $= n-1$ then
        return
    end if
    min $\leftarrow$ start_idx
    for $j \leftarrow$ start_idx $+ 1, \ldots, n-1$ do
        if $A[j] < A[\texttt{min}]$ then
            min $\leftarrow j$
        end if
    end for
    if min $\neq$ start_idx then
        Swap $A[\texttt{min}]$ and $A[\texttt{start\_idx}]$
    end if
    SelectionSortRecursive($A$, start_idx $+ 1, n$)
end procedure
```

Analysis of algorithm 12 gives a running time asymptotic to the recurrence

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + n \qquad \text{for } n \geq 2 \end{aligned}$$

This recurrence can be proven to be equivalent to the expression

$$\sum_{i=0}^{n-1} i + 1 = 1 + 2 + \ldots + n$$

which is $\Theta(n^2)$.

7 Efficiency of Recursive Implementations

The two selection sort algorithms in the previous section are asymptotically equivalent, and have comparable primitive operation counts. However, recursive algorithms are often assumed to be inferior to equivalent iterative algorithms, even when running times are similar.

As sections 4.1 and 4.2 show, recursive algorithms which appear to be very simple may have extremely poor asymptotic performance, and this may lead programmers to be wary of using recursion when iterative methods are available. Additionally, recursive calls, like any function call, normally take longer than regular intra-function branches (like those found in a loop).

In a language like C or Java, a function call must allocate memory for the local storage of the function (and various other information, such as a return address). As a result, a sequence of n recursive calls may require $O(n)$ space. Extremely deep recursion may even exhaust the available space on the call stack, causing a crash[10].

However, not all recursive algorithms have these problems. The recursive selection sort method given by algorithm 12 uses a pattern called *tail recursion*, where the recursive call is the last operation (besides possibly a return statement). In a tail-recursive function, it is not necessary to allocate new storage for the local variables of the recursively-called function, since the local variables of the caller can be recycled. The compiler can also optimize away some of the other function-call infrastructure, since a recursive call is conceptually similar to an intra-function branch. As a result, modern compilers are able to optimize tail-recursive functions, such as algorithm 12 into iterative code. This is referred to as *tail call optimization*. Figure 6 shows C code for algorithm 12 before and after tail call optimization. The `gcc` C compiler (with the optimization flag `-O3`) and the Java virtual machine both support tail call optimization.

10. The default stack size is often relatively small. For example, Ubuntu 12.04 uses a default size of 8 megabytes. To view the active stack size in kilobytes on a linux machine, use the command `ulimit -s` (for `bash`) or `limit s` (for `csh`). When very deep recursion is necessary (or when functions require large amounts of stack space), it is possible to change the amount of stack space allocated for new processes, subject to global restrictions. For example, `ulimit -s 65536` will increase the limit to 64 megabytes.

```
void selectionsort(int* A, int n, int start_idx){
    int j;

    if (start_idx == n){
        return;
    }
    int min = start_idx;
    for(j = start_idx; j < n; j++)
        if (A[j] < A[min])
            min = j;
    if (min != start_idx){
        j = A[min];
        A[min] = A[start_idx];
        A[start_idx] = j;
    }

    selectionsort(A,n,start_idx+1);
}
```

(a) Original Code

```
void selectionsort(int* A, int n, int start_idx){
    int j;
L1:
    if (start_idx == n){
        return;
    }
    int min = start_idx;
    for(j = start_idx; j < n; j++)
        if (A[j] < A[min])
            min = j;
    if (min != start_idx){
        j = A[min];
        A[min] = A[start_idx];
        A[start_idx] = j;
    }
    start_idx++;
    goto L1;
}
```

(b) Compiler Optimized Code

Figure 6: Recursive selection sort in C, before and after tail call optimization. **Don't use `goto` statements like this in your own code: let the compiler handle optimization for you.**

8 Evaluation Proves Nothing

Consider the recurrence

$$\begin{aligned} T(0) &= 1 \\ T(n) &= T(n-1) + \frac{n(n-1)}{2} + 1 \qquad \text{for } n \geq 1 \end{aligned}$$

Evaluating the first few terms of this recurrence gives

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 2 \\ T(2) &= 4 \\ T(3) &= 8 \end{aligned}$$

Based on this information, it is reasonable to hypothesize that $T(n) = 2^n$ is a closed form for the recurrence. Often, the correct closed form can be obtained on the first guess. However, no matter how obvious the answer seems, without a formal proof for all values of n, no hypothesized closed form can be considered correct.

For the recurrence $T(n)$, the hypothesized closed form of 2^n is incorrect, as the evaluation of $T(4) = 15$ proves. The correct closed form is $T(n) = \frac{1}{6}n^3 + \frac{5}{6}n + 1$, which can be proved by induction. Figure 7 compares two functions. It is possible to create polynomials which mimic the behavior of other functions for any (finite) number of data points, so even though $T(4)$ provides a counterexample to $T(n) = 2^n$ in this case, it would be possible to derive a polynomial $p(n)$ such that $p(n) = 2^n$ for an arbitrary set of integers n, where a counterexample would be more difficult to find[11].

11. Polynomials created for this purpose are called *interpolating polynomials* and can be useful for approximating functions which are difficult to evaluate. Interpolation methods are covered in detail in a numerical analysis course.

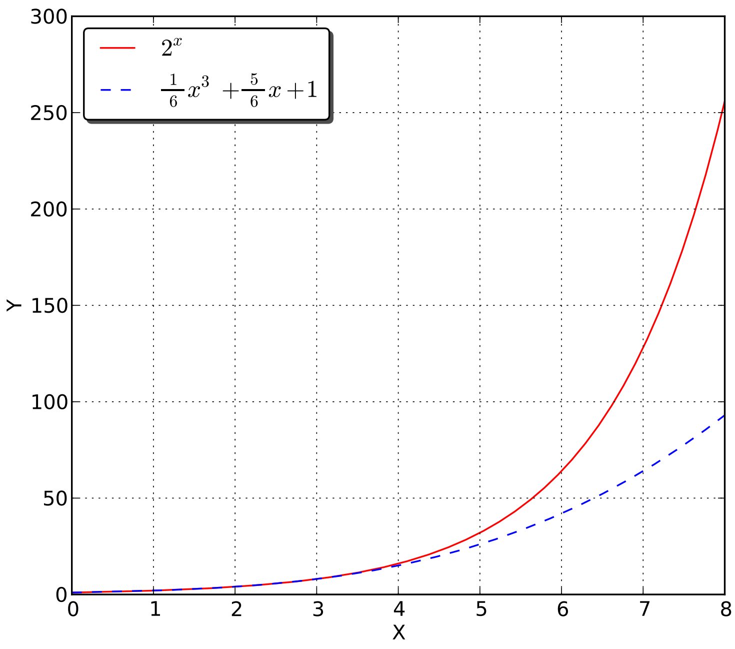


Figure 7: Comparison of the hypothesized solution 2^n and actual solution $\frac{1}{6}n^3 + \frac{5}{6}n + 1$ to the recurrence $T(n)$ on $[0, 8]$

9 'Triangular' Nested Loops

Analysis of nested loops of the form

```
for $i \leftarrow 1, \ldots, n$ do
    for $j \leftarrow i, \ldots, n$ do
        {Statements requiring $O(1)$ time}
    end for
end for
```

gives a running time asymptotic to

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

which is $\Theta(n^2)$ due to the identity

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

Similarly, the recurrence

$$\begin{aligned} T(0) &= c \\ T(n) &= \Theta(n) + T(n-1) \qquad \text{for } n \geq 1 \end{aligned}$$

which often appears in the analysis of the recursive version of the above loops, is also asymptotic to the sum $1 + \ldots + n$ and therefore $\Theta(n^2)$.

In practice, iterative algorithms containing these nested loops, or their recursive equivalent, are often encountered in applications involving symmetric matrices (e.g. the adjacency matrix of an undirected graph), iteration over the lower- or upper-triangle of a square matrix (e.g. a matrix transpose algorithm), or iteration over all unordered pairs of some collection of elements (e.g. searching for duplicates in a collection of objects without a totally ordered comparison operator).

The identity

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

can be proven in several different ways. The inductive proof is often used as an introductory example in first year courses:

Basis

When $n = 0$, $\sum_{i=1}^{n} i = 0$, and $\frac{n(n+1)}{2} = 0$, so the identity holds.

Induction Hypothesis

Suppose $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for some $n \geq 0$.

Induction Step

Consider $n + 1$.

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= (n+1) + \sum_{i=1}^{n} i \\
&= (n+1) + \frac{n(n+1)}{2} && \text{(By the induction hypothesis)} \\
&= \frac{2n + 2 + n^2 + n}{2} \\
&= \frac{n^2 + 3n + 2}{2} \\
&= \frac{(n+1)(n+2)}{2}
\end{aligned}$$

Therefore, the identity holds for $n+1$, and by induction, the identity holds for all $n \geq 0$.

Using the fact that

$$\frac{n(n+1)}{2} = \binom{n+1}{2}$$

a simple combinatorial argument can also be used to prove the identity. The value $\binom{n+1}{2}$ counts the number of ways to choose an unordered pair $\{a, b\}$ of items from the collection $\{1, \ldots, n+1\}$. If we assume that $a < b$ (which is permitted since order is irrelevant), we can also count the possibilities for each value of b. When $b = 1$, no pairs are possible, since there are no choices for a. When $b \geq 2$, there are $b - 1$ choices for a, so the total number of possibilities is

$$\sum_{b=2}^{n+1} (b-1) = \sum_{b=1}^{n} b$$

Since we counted the same quantity in two different ways, the two counts must be equal. Therefore,

$$\sum_{i=1}^{n} i = \binom{n+1}{2} = \frac{n(n+1)}{2}$$

While it is superfluous to do so here, since we have a closed form for the sum $\sum_{i=1}^{n} i$, we can also use an 'reduction' argument to assert that $\sum_{i=1}^{n} i \in \Theta(n^2)$. Recall that the *transpose* of an $n \times n$ matrix A, denoted A^T, is the reflection of A across its forward diagonal:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, \quad A^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}$$

Pseudo-code for transposing an $n \times n$ matrix A is given in algorithm 13.

Algorithm 13 Matrix Transpose

```
procedure TRANSPOSE($A, n$)
    for $i \leftarrow 0, \dots, n-1$ do
        for $j \leftarrow i+1, \dots, n-1$ do
            Swap $A[i][j]$ and $A[j][i]$
        end for
    end for
end procedure
```

The diagonal entries of A can be ignored by a transpose algorithm, but the remaining $n^2 - n$ entries must be accessed at least once. Since the number of memory accesses of the algorithm cannot exceed its running time, we can conclude that algorithm 13 is $\Omega(n^2)$ (assuming, of course, that the algorithm is correct). We can also see that algorithm 13 contains the familiar nested loops, and since the swap operation is $O(1)$, we know from the earlier analysis that the running time is asymptotic to

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Since algorithm 13 is $\Omega(n^2)$, we can conclude that

$$\sum_{i=1}^{n} i \in \Omega(n^2)$$

Similarly, since

$$\sum_{i=1}^{n} i \leq \sum_{i=1}^{n} n = n^2$$

we can conclude that algorithm 13 is $O(n^2)$, and therefore that it must be $\Theta(n^2)$. This not only proves that the algorithm is asymptotic to n^2, but that the sum $1 + \ldots + n$ is as well.

10 Squeeze Law for Asymptotics

For three functions $f_1(x), f_2(x)$ and $g(x)$, if

$$f_1(x) \in O(g(x)), \quad g(x) \in O(f_2(x)), \quad f_2(x) \in O(f_1(x))$$

then $\Theta(f_1(x)) = \Theta(f_2(x)) = \Theta(g(x))$ by the definition of Θ. This property can be useful when the asymptotic behavior of g is difficult to determine with more direct methods.

Consider the function

$$g(n) = \sum_{i=1}^{n} i^k = n^k + (n-1)^k + \ldots + 2^k + 1^k$$

The value of $g(n)$ is the sum of the k^{th} powers of all integers less than or equal to n. Geometrically, since $i^k = 1 \cdot i^k$, each term i^k can be thought of as a rectangle of width 1 and height i^k, and the value of $g(n)$ can be thought of as the combined area of the set of rectangles[12].

Let

$$f_1(x) = x^k, \quad f_2(x) = (x+1)^k$$

Then for all i, $f_1(i) \leq i^k \leq f_2(i)$, and the area of a rectangle with width 1 and height i^k can be bounded by integrating each function,

$$\int_{i-1}^{i} x^k \mathrm{d}x \leq \int_{i-1}^{i} i^k \mathrm{d}x \leq \int_{i-1}^{i} (x+1)^k \mathrm{d}x$$

$$\frac{i^{k+1} - (i-1)^{k+1}}{k+1} \leq i^k \leq \frac{(i+1)^{k+1} - i^{k+1}}{k+1}$$

Applying this relationship to the entire sum $\sum_{i=1}^{n} i^k$ gives

$$\int_{0}^{n} x^k \mathrm{d}x \leq \sum_{i=1}^{n} i^k \leq \int_{0}^{n} (x+1)^k \mathrm{d}x$$

$$\frac{n^{k+1}}{k+1} \leq g(n) \leq \frac{(n+1)^{k+1}}{k+1}$$

Giving

$$n^{k+1} \in O(g(n)), \quad g(n) \in O((n+1)^{k+1})$$

and since it can be shown that $n^{k+1} \in \Theta((n+1)^{k+1})$, we have

$$g(n) \in \Theta(n^{k+1})$$

Figure 8 shows the terms of the sum (green bars) compared to the two functions f_1 and f_2.

12. One way to formalize this is to rewrite the discrete sum $\sum_{i=1}^{n} i^k$ as the integral $\int_0^n \lceil x \rceil^2 \mathrm{d}x$.

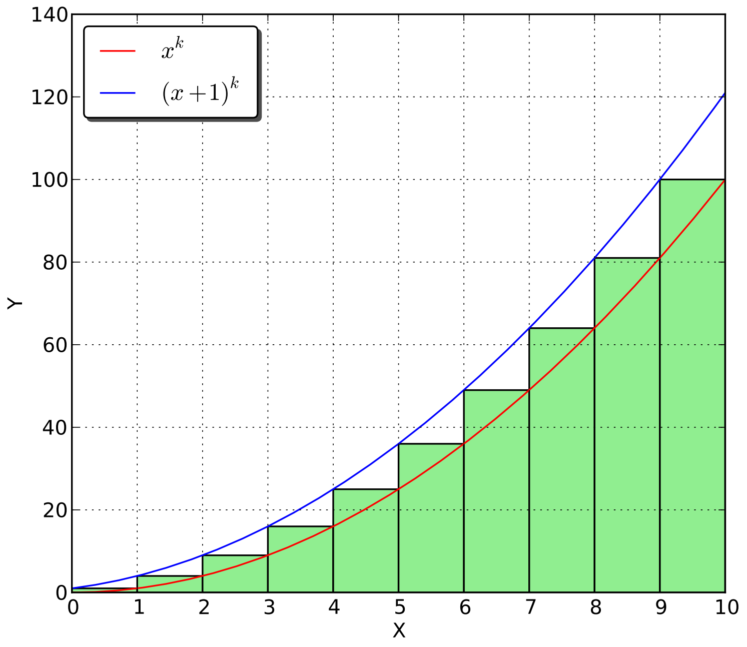


Figure 8: Comparison of the terms of the sum $\sum_{i=1}^{n} i^k$, represented by green bars, with the functions $f_1(x)$ and $f_2(x)$

In general, if $F(n)$ is a function of the form

$$F(n) = \sum_{i=0}^{n} f(i)$$

and a function $g(n)$ can be found where $f(i) \in O(g(i))$, then

$$F(n) \in O\left(\int_0^n g(x)\mathrm{d}x\right)$$

and the predictable generalization to Θ and Ω also holds.

Part II

Searching and Sorting

11 Selection

Given an unordered sequence A of size n and an integer k, a *selection* algorithm finds the k^{th}-smallest element[13] in A. Since the k^{th}-smallest element in A is simply the k^{th} element in the sorted ordering of A, one solution to the selection problem is to sort A and return the k^{th} element. This takes $O(n \log n)$ time in general[14]. Unlike comparison sorting, there is no

13. Note that for 'k^{th}-smallest element' to be defined, we must assume that the elements of A have a unique sorted ordering. This constraint also applies to comparison sorting algorithms (such as merge sort).

14. That is, when no assumptions can be made about the input data which allow the sorting method to break the $\Omega(n \log n)$ bound.

$\Omega(n \log n)$ bound on selection, only a trivial $\Omega(n)$ bound (since every element must be examined at least once) which can be attained with the specialized selection algorithms described in this section.

Since sorting can be used to solve the selection problem, it is natural to use sorting algorithms as the basis for selection algorithms. The merge sort and heap sort algorithms are particularly attractive since they both have $O(n \log n)$ worst case performance.

The initial partition of an input array A by the merge sort algorithm does not examine any of the elements, and therefore the k^{th}-smallest element may lie in either part. Additionally, at the first recursive step, it is unknown what the relative position of the desired element is within the partitioned array, so the merge sort algorithm must fully sort both parts before the k^{th}-smallest element can be identified. As a result, a merge-based selection algorithm with running time better than $O(n \log n)$ in the worst case cannot be easily derived from merge sort.

11.1 Heap Select

Algorithm 14 gives pseudocode for a selection algorithm based on heap sort. The heap H can be built from the elements of A in $O(n)$ time using one of the algorithms from section 13, such as algorithms 17 or 18. The k heap-removal operations each take $O(\log n)$ time, giving a total running time of $O(n + k \log n)$.

Algorithm 14 Heap-based selection algorithm

```
procedure HeapSelect($A, k$)
    $H \leftarrow$ Copy of $A$
    Convert $H$ to an array-based min-heap
    {Discard the smallest $k - 1$ elements in the heap.}
    for $i = 0, \ldots, k - 2$ do
        Remove the minimum element from the heap $H$.
    end for
    {The minimum element left on the heap is the $k^{\text{th}}$-smallest in $A$.}
    element $\leftarrow$ Minimum element in the heap $H$.
    return element
end procedure
```

If k is constant[15], then the running time of algorithm 14 is $O(n)$ in the worst case. For example, the 10th-smallest element can be found in $O(n + 10 \log n) = O(n)$ time on all inputs. Modifying algorithm 14 to use a max-heap instead of a min-heap would find the k^{th}-largest element instead.

When $k \in \Theta(n)$, such as the case $k = \frac{n}{2}$ which occurs when finding the median of the elements of A, the overall running time of algorithm 14 is $O(n \log n)$.

11.2 Quick Select

Although the worst case running time of deterministic quick sort is $O(n^2)$, the quick sort algorithm has several properties that warrant its consideration for a fast selection algorithm. First, in practice, quick sort tends to be highly efficient, since there are relatively few worst case inputs. Second, randomization is very effective at preventing the worst case behavior from occurring.

15. Or, more generally, if $k \in O(\frac{n}{\log n})$.

Finally, the mechanics of quick sort are, in a broad sense, a mirror-image of the mechanics of merge sort: the comparison step, along with most of the actual sorting, takes place before the recursive call, and almost no work is needed after recursion returns. This is advantageous for selection, since it is possible to know before recursion begins exactly which part of the partition contains the desired element. The digram in figure 9 illustrates this property for the case $k = \frac{n}{2}$ (which finds the median of the input array). After pivoting about an element p, there are fewer than $\frac{n}{2}$ elements less than or equal to p, so index $\frac{n}{2}$ in the sorted array must be an element greater than p. Accordingly, it is only necessary to recurse on the sequence containing elements larger than p, adjusting the value of k to compensate for the elements in the other parts of the partition. If there are m elements less than or equal to p, then the desired value will be the $(k-m)^{\text{th}}$-smallest element among those greater than p.

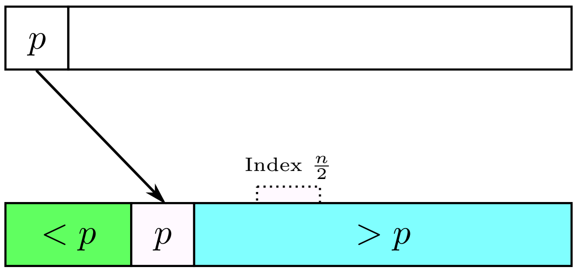


Figure 9: Illustration of a single pivoting operation in quick select. It is only necessary to recurse into the blue sub-array to find the median value (index $\frac{n}{2}$ in the sorted array).

Algorithm 15 gives pseudocode for quick select, without specifying a pivot selection scheme. Figure 10 shows the steps of quick select on a sample input when the first element is used as the pivot at each step.

Algorithm 15 Quick select

```
procedure QUICKSELECT($A, k$)
    if length($A$) $= 1$ then
        return $A[0]$
    end if
    $p \leftarrow$ Pivot value from $A$
    Partition $A$ into $L$, $E$ and $G$ using the pivot $p$.
    if $k <$ length($L$) then
        return QUICKSELECT($L, k$)
    else if $k \geq$ length($L$) + length($E$) then
        return QUICKSELECT($G, k -$ length($L$) $-$ length($E$))
    else
        return $p$
    end if
end procedure
```

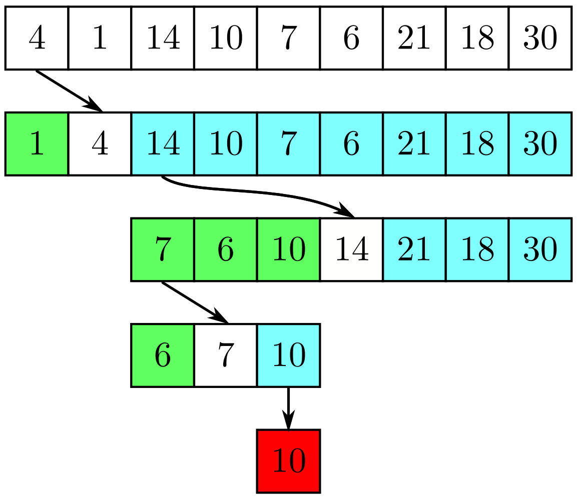


Figure 10: Steps of quick select on a sample array when the first element is used as the pivot at each step.

The running time of algorithm 15 on an input of size $n > 1$ in the worst case is

$$T(n) = O(n) + T(\max(|L|, |G|))$$

When $\max(|L|, |G|) \geq \frac{n}{c}$ for some constant $c > 1$, the recurrence can be bounded above:

$$\begin{aligned} T(n) &\leq O(n) + T(\frac{n}{c}) \\ &= O(n) + \left[O(\frac{n}{c}) + T(\frac{n}{c^2})\right] \\ &= O(n) + O(\frac{n}{c}) + \left[O(\frac{n}{c^2}) + T(\frac{n}{c^3})\right] \\ &\quad \vdots \\ &\leq O(n) \cdot \sum_{i=0}^{\infty} \frac{1}{c^i} \end{aligned}$$

The infinite sum in the last equation converges to a constant[16], so $T(n) \in O(n)$ when both L and G can be guaranteed to contain at least a certain constant fraction of the input array.

However, simple pivot selection rules offer no such guarantee. The pivoting rule used in the example in figure 10, in which the first element of the input array is used as the pivot, may select the smallest (or largest) element in the array as a pivot at any step, giving $|L| = 0$ and $|G| = n - 1$. Since we cannot assume that this case occurs infrequently (since for worst case performance we must consider the worst possible pivot selection at every step), the recurrence above becomes

$$T(n) = O(n) + T(n - 1)$$

which is $O(n^2)$.

One remedy to the pivot selection problem which is effective in practice is to choose a random element as the pivot at each iteration. In quick sort, randomization gives an expected running

16. One simple way to prove this is to observe that since $c^{-x} \leq c^{-x/2}$ for all real $x \geq 0$ the sum $\sum_{i=0}^{\infty} c^{-i}$ is bounded above by the integral $\int_{x=0}^{\infty} c^{-x/2} \mathrm{d}x$, which converges to $2/\log(c)$. Another option is to take the limit of the geometric series identity $\sum_{i=0}^{n} c^i = \frac{1-c^{n+1}}{1-c}$ as n approaches infinity to get the exact value $1/(1-c)$.

time of $O(n \log n)$, and in quick select, it gives an expected running time of $O(n)$. In practice, randomized quick select will likely have better performance than any other selection algorithm. Asymptotically, however, randomization does not affect the worst case running time, since there is no way to assert that the sequence of random pivots does not cause the worst case behavior (although it can be shown that such a sequence of pivots is extremely unlikely).

A pivot selection scheme which guarantees that both L and G will contain a minimum fraction of the total elements at each step was proposed by Blum, Floyd, Pratt, Rivest and Tarjan in 1973[17]. The BFPRT algorithm uses quick select recursively to choose a pivot which loosely approximates a median of the input array, guaranteeing that the chosen pivot will be both greater than and less than a certain fraction of the total items.

To select a pivot from the input array A, the n elements of A are first divided into subarrays of a constant size. For quick select to achieve linear time performance, it is necessary to use subarrays of size greater than 3, and to avoid confusion over which element to take as a median, it is desirable to use subarrays of an odd size[18]. Therefore, the examples in this section will assume subarrays have size 5.

Dividing the input array A into subarrays $A_1, \ldots, A_m$ of size 5 (where $m = \frac{n}{5}$) requires $O(n)$ time in the worst case[19]. Each subarray is then sorted, which requires $O(1)$ time per subarray since the size of each subarray is fixed. The total time required to sort all subarrays is therefore $O(n)$. After sorting, the element $A_i[2]$ (using 0-based indexing) will be the median of the subarray A_i. The medians of each subarray are then collected into an array M of size $m = \frac{n}{5}$. Quick select is then used recursively to find the median of M, which is used as the pivot for the input array A. Algorithm 16 gives pseudocode for the BFPRT pivot selection algorithm.

In cases where the length n of the input array A is not a multiple of 5, it is acceptable to simply ignore the last $n \bmod 5$ elements (that is, only consider the first $n-(n \bmod 5)$ elements, effectively rounding n down to the nearest multiple of 5). The number of discarded elements is always bounded by a constant (since at most 4 elements will ever be discarded), so discarding them does not affect the viability of the selected pivot (although the analysis below will assume that n is a multiple of 5 to simplify the proof). In cases where $n < 5$, the entire array can be sorted in constant time (since the array's size is bounded by a constant) and the median value chosen as the pivot.

Algorithm 16 BFPRT pivot selection using subarrays of size 5

```
procedure PivotBFPRT($A$)
    Partition $A$ into arrays $A_1, \ldots, A_m$ of size 5
    Sort each $A_i$
    {Create an array $M$ of the medians of the $A_i$}
    Let $M = [A_1[2], A_2[2], \ldots, A_m[2]]$
    {Return the median of $M$}
    return Quickselect($M$, $\frac{m}{2}$)
end procedure
```

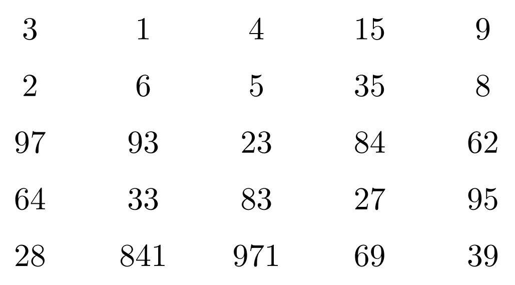
17. See Blum, Floyd, Pratt, Rivest and Tarjan, *Time bounds for selection*, J. Comput. System Sci., vol. 7 no. 4 (1973) 448-461.

18. Obviously, we can easily make up a rule for which element of an array of even size we will take as the median, but the differing number of elements on either side of the median complicates the analysis.

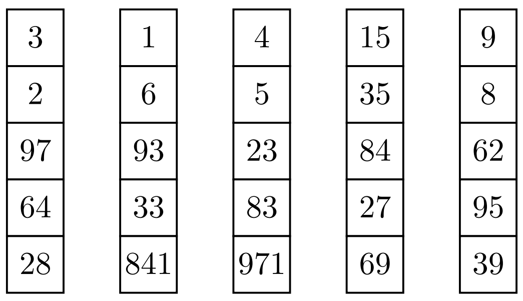
19. In an actual implementation, the partitioning step may take $O(1)$ time since it is possible to reuse the storage of A and implicitly group elements.

Since algorithm 16 makes a recursive call to algorithm 15, it is not possible to analyze the running time of algorithm 16 until the properties of the pivot it selects are known. It is, however, possible to observe that, besides the recursive call to Quickselect, algorithm 16 is $O(n)$, and that the recursive call takes $T(\frac{n}{5})$ time, where $T(n)$ is the running time of the quick select algorithm.

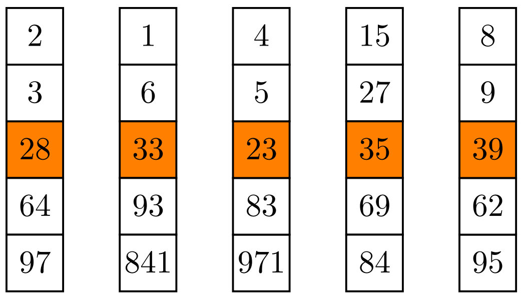
Although the pivot selected by algorithm 16 is not the median of A, we can prove that it lies both above and below at least $\frac{3}{10}$ of the elements of A. A pivot p selected by algorithm 16 is the median of the array M, which contains the medians of all $\frac{n}{5}$ subarrays of size 5. As the median of M, p is greater than $\frac{1}{2} \cdot |M| = \frac{1}{2} \cdot \frac{n}{5} = \frac{n}{10}$ of the other medians in M, and therefore greater than at least 3 elements (including the median) in the subarrays containing those medians. Therefore, p is greater than at least $\frac{3n}{10}$ other elements in A. Similarly, p is less than at least $\frac{3n}{10}$ elements in A. As a result, both of the sequences L and G in the quick select algorithm will have size at most $n - \frac{3}{10} = \frac{7}{10}n$. Figure 11 shows the steps of the BFPRT pivot selection algorithm on a sample input array.



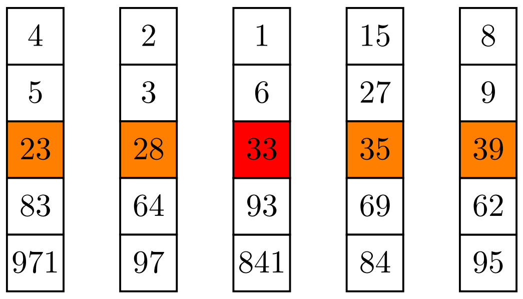
(a) Input data



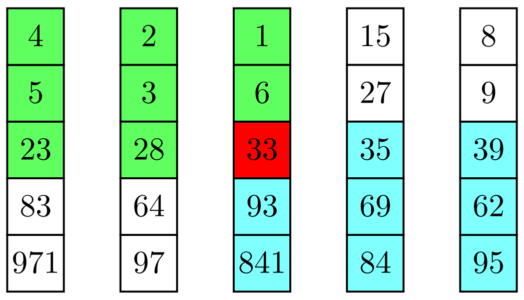
(b) After subdivision into $\frac{n}{5}$ subarrays



(c) After sorting each subarray (with medians highlighted)



(d) The pivot (highlighted in red) is the median of the set of medians.



(e) The pivot is guaranteed to be greater than all of the elements in the upper left quadrant (shaded in green) and less than all of the elements in the lower right quadrant (shaded in blue). There will always be at least $\frac{3n}{10}$ values in each quadrant.

Figure 11: Steps of the BFPRT pivot selection process on a sample array. Note that the rearrangement of subarrays in the last two steps is purely for visual clarity: the algorithm itself does not sort the list of medians.

Using the selected pivot, the running time of algorithm 15 with the pivot selection scheme of algorithm 16 for an array of size n is

$$\begin{aligned}
T(n) &= O(n) + T\left(\frac{n}{5}\right) + T(\max(|L|, |G|)) \\
&\leq cn + T\left(\frac{n}{5}\right) + T(\max(|L|, |G|)) \qquad \text{for some constant } c \\
&\leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\
&\leq cn + T\left(n\frac{2}{10}\right) + T\left(n\frac{7}{10}\right)
\end{aligned}$$

The constant c can be chosen such that the base case $T(1) \leq c$. Before continuing, we must prove that

$$T(a) + T(b) = T(a+b)$$

The proof is by contradiction. Suppose the relation does not hold and consider the minimum pair[20] of values (a, b) which violates the identity. Since neither a nor b can be the base case (which is a constant), we can expand all terms in the above recurrence:

$$\begin{gathered}
T(a) + T(b) \neq T(a+b) \\
\left[ca + T\left(a\frac{2}{10}\right) + T\left(a\frac{7}{10}\right)\right] + \left[cb + T\left(b\frac{2}{10}\right) + T\left(b\frac{7}{10}\right)\right] \neq \left[c(a+b) + T\left((a+b)\frac{2}{10}\right) + T\left((a+b)\frac{7}{10}\right)\right] \\
c(a+b) + T\left(a\frac{2}{10}\right) + T\left(b\frac{2}{10}\right) + T\left(a\frac{7}{10}\right) + T\left(b\frac{7}{10}\right) \neq c(a+b) + T\left((a+b)\frac{2}{10}\right) + T\left((a+b)\frac{7}{10}\right)
\end{gathered}$$

Since the pair (a, b) was chosen to be the minimum pair which violated the identity, and every argument to T on the left hand side of the last equation is smaller than both a and b, the identity holds for all of them and we can combine the terms:

$$\begin{gathered}
c(a+b) + T\left(a\frac{2}{10} + b\frac{2}{10}\right) + T\left(a\frac{7}{10} + b\frac{7}{10}\right) \neq c(a+b) + T\left((a+b)\frac{2}{10}\right) + T\left((a+b)\frac{7}{10}\right) \\
c(a+b) + T\left((a+b)\frac{2}{10}\right) + T\left((a+b)\frac{7}{10}\right) \neq c(a+b) + T\left((a+b)\frac{2}{10}\right) + T\left((a+b)\frac{7}{10}\right)
\end{gathered}$$

Both sides of the inequality are the same, which is clearly a contradiction. Therefore, the identity

20. In this case, (x, y) is a 'minimum pair' if there is no pair (x', y') with either value smaller than either value of (x, y)

holds. We can now use the identity to simplify the recurrence:

$$\begin{aligned}
T(n) &\leq cn + T\left(n\frac{2}{10}\right) + T\left(n\frac{7}{10}\right) \\
&= cn + T\left(n\frac{9}{10}\right) \\
&= cn + \left[cn\frac{9}{10} + T\left(n\left(\frac{9}{10}\right)^2\right)\right] \\
&= cn + cn\frac{9}{10} + \left[cn\left(\frac{9}{10}\right)^2 + T\left(n\left(\frac{9}{10}\right)^3\right)\right] \\
&\leq cn\sum_{i=0}^{\infty}\left(\frac{9}{10}\right)^i
\end{aligned}$$

The infinite series in the last line converges to a constant, so $T(n) \in O(n)$.

The BFPRT pivot selection mechanism can also be used to choose pivots for quick sort, which produces a sorting algorithm with a deterministic worst case running time of $O(n \log n)$ (however, we consider the resulting algorithm to be distinct from quick sort, which we still characterize as having $O(n^2)$ running time in the worst case). In practice, the overhead of the BFPRT process makes a randomized quick select much faster in the vast majority of cases.

12 Trees

In general, a *tree* is a connected graph on n vertices[21] with $n-1$ edges. A *rooted tree* has a distinguished root vertex. Trees without a distinguished root are sometimes called *free trees*. Algorithms which work with trees often use a rooted representation, even when working with free trees. Figure 12 shows two trees which are equivalent when considered as free trees, but distinct when considered as rooted trees.

In a rooted tree, the *depth* of a node is the distance from that node to the root, where the root itself is considered to have depth 0. The *height* of a tree is the maximum depth of any node. The *parent* of a node v is the unique node connected to v which has a smaller depth, and the *children* of v are the nodes connected to v which have larger depths. A node is called a *leaf* if it has no children.

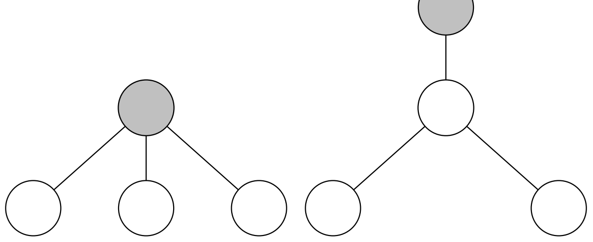


Figure 12: Two trees on 4 vertices. As free trees, they are equivalent, but when considered as rooted trees (where the root is shaded), they are distinct.

21. The terms ‘vertex’ and ‘node’ as applied to trees are used interchangeably in this document.

12.1 Binary Trees

A *binary tree* is a tree in which all nodes have either $0, 1$ or 2 children[22]. When binary trees are covered in first-year programming courses, they are usually implemented with linked data structures, since a tree node can be conveniently encapsulated with object-oriented programming, and the linked representation of binary trees is conceptually similar to other simple data structures such as linked lists. A binary tree can also be implemented with array-based storage, by numbering the nodes left-to-right, top to bottom, as shown in figure 13

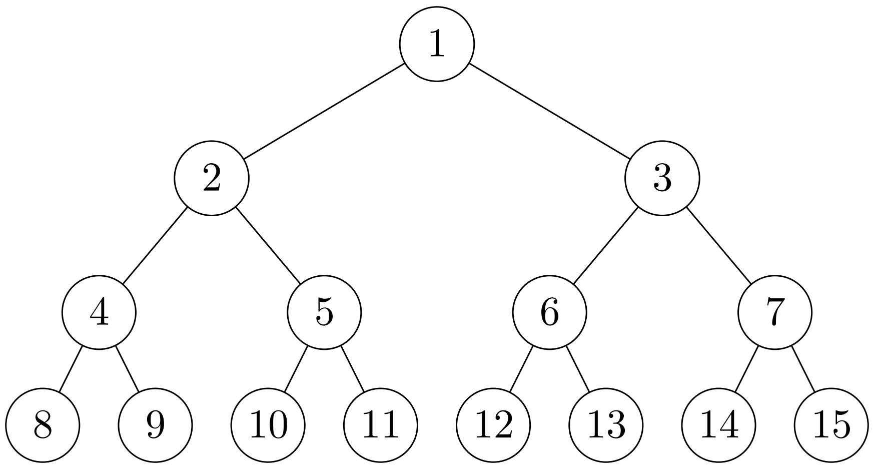


Figure 13: A binary tree, where each node is labeled with its index in an array based representation.

This straightforward indexing scheme has several useful mathematical properties. Unlike a linked representation, where it is necessary to store pointers to the parent and children of each node, the index i of a node is enough to determine the indices of the parent and children of node i. Specifically,

$$\begin{aligned}
\textsc{Parent}(i) &= \lfloor \frac{i}{2} \rfloor = i/2 \text{ with integer division (for } i > 1) \\
\textsc{LeftChild}(i) &= 2i \\
\textsc{RightChild}(i) &= 2i + 1
\end{aligned}$$

Since there is no need to store pointers, the array-based representation can be very compact compared to the linked representation. In addition, using an array to store the contents of the tree ensures that the set of nodes is relatively clustered in memory compared to the linked representation, which may improve cache performance. It should be noted that there is no inherent performance advantage to using indices over pointers, since an array index is fundamentally the same as a pointer, only on a smaller scale.

As with other data structures which have both linked and array-based representations, the best choice of representation depends on the application. For a tree with depth d on n vertices, the space required for an array-based representation is $O(2^{d+1})$. In cases where $\log_2 n$ may be significantly less than d, a linked representation may be a better choice, since an array-based representation would have significant wasted space. Applications which insert nodes into the

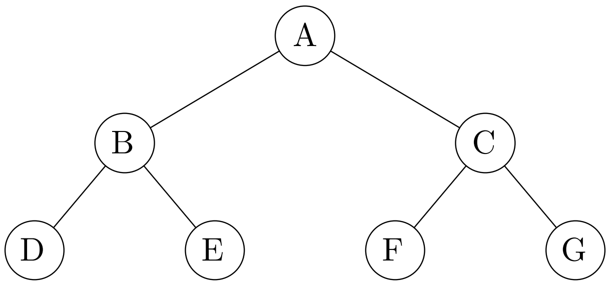
22. In computer science, the term ‘binary tree’ is always assumed to refer to a rooted tree unless otherwise specified.

middle of the tree, or rearrange entire subtrees, benefit from a linked representation, where these operations are $O(1)$, instead of an array representation where they are $O(n)$. However, applications in which the tree is expected to be very dense, and insertions are expected only at the bottom level of the tree, such as a heap, benefit from the array representation, which lacks the overhead of the linked structure.

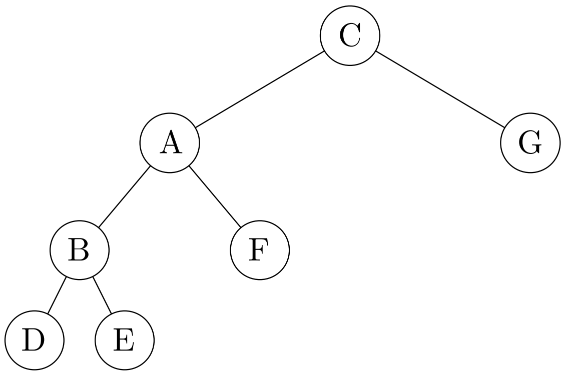
It should be noted that the indexing scheme above is valid for all k-ary trees where k is fixed.

12.2 Rotation in Binary Trees

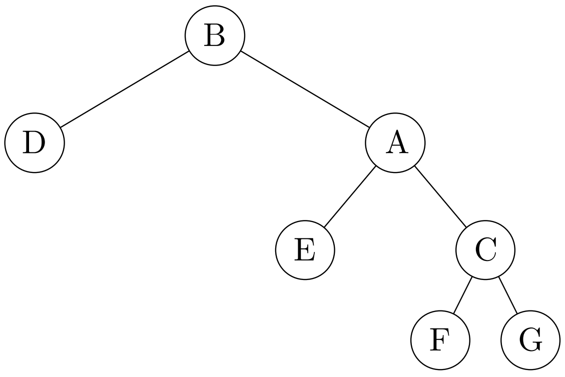
With respect to a node v of a binary tree (which may not be the root of the tree), a *rotation* shifts v into the position of one of its children, while preserving the ordering of an in-order traversal of the tree. A *left rotation* shifts v into the position of its left child, and moves the right child of v into the position previously occupied by v. A *right rotation* is defined similarly. Figure 14 shows a tree and the results of left- and right-rotation about its root.



(a) Initial Tree



(b) After left-rotation



(c) After right-rotation

Figure 14: A binary tree before and after left- and right-rotation about vertex A.

Since rotation can be performed in constant-time on a linked representation of a binary tree, it provides a mechanism for efficiently adjusting the heights of subtrees without compromising the overall ordering of nodes in an in-order traversal. It is often encountered as part of the rebalancing step of self-balancing binary search trees, such as AVL trees (see section 20).

13 Heap Sort

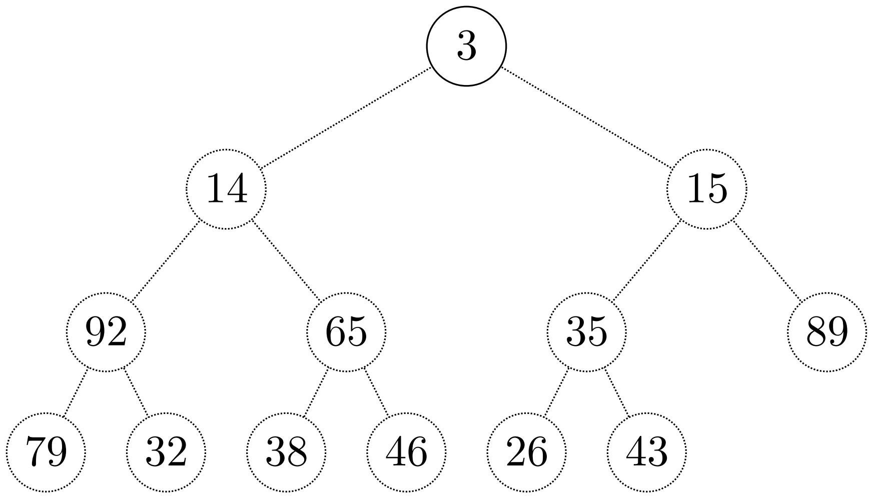
The heap sort algorithm uses the $O(\log n)$ insertion and remove-minimum operations of a heap to sort an array of n elements in $O(n \log n)$ time. The simplest implementation of heap sort, particularly in languages whose standard library provides a heap data structure, takes an array A of n elements, creates a heap, inserts all n elements into the heap, then removes successive minimum elements and inserts each one back into A. This approach requires $O(n)$ extra space, since the heap does not share storage with A.

An in-place version of heap sort, which uses the input array as the heap storage, can be implemented by considering the input array to be an array-based binary tree, then rearranging the elements to satisfy the heap property. For convenience, we use a *max-heap*, in which the value of each node is greater than or equal to the values of its children[23].

The first phase of in-place heap sort takes the input array and transforms it into the array-based representation of a heap. There are two ways to do this. The *top-down* approach starts with a heap of size 1 (containing the first element of the array) and gradually expands the heap to contain successive elements of the array, mimicking a sequence of n heap insertions. Figures 15 through 27 illustrate the top-down heap construction on the following array A:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	14	15	92	65	35	89	79	32	38	46	26	43

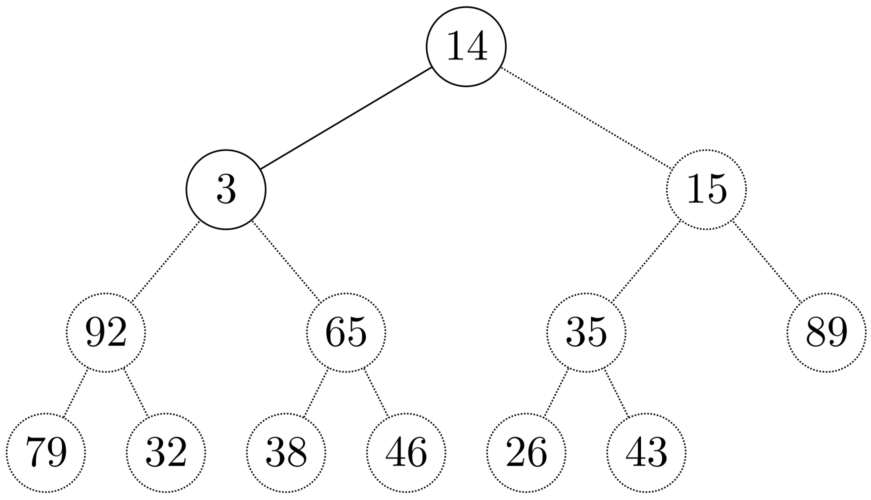
Note that we use 1-based indexing for this example. In practice, it is not necessary to shift the elements of the input array to allow 1-based indexing, since indices can be implicitly shifted. It is important to remember that we assume that the input array *already* represents a binary tree, and that the top-down construction only modifies this tree to produce a valid heap. In the diagrams in this section, nodes with a solid outline are part of a valid heap, while nodes with a dotted outline are not. In the array listing for each diagram, the array elements which are part of a valid heap are shown in bold.



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**3**	14	15	92	65	35	89	79	32	38	46	26	43

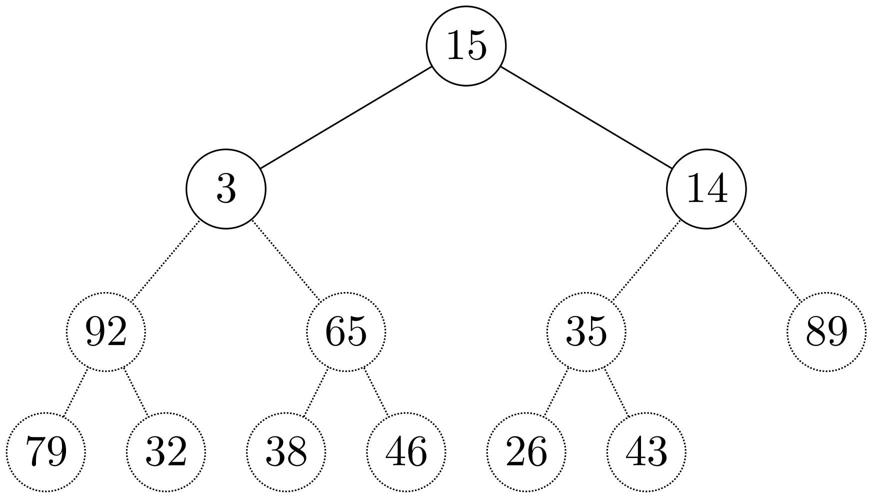
Figure 15: Top-down heap construction - Step 1

23. It is also possible to use a min-heap with tedious but constant-time reindexing.



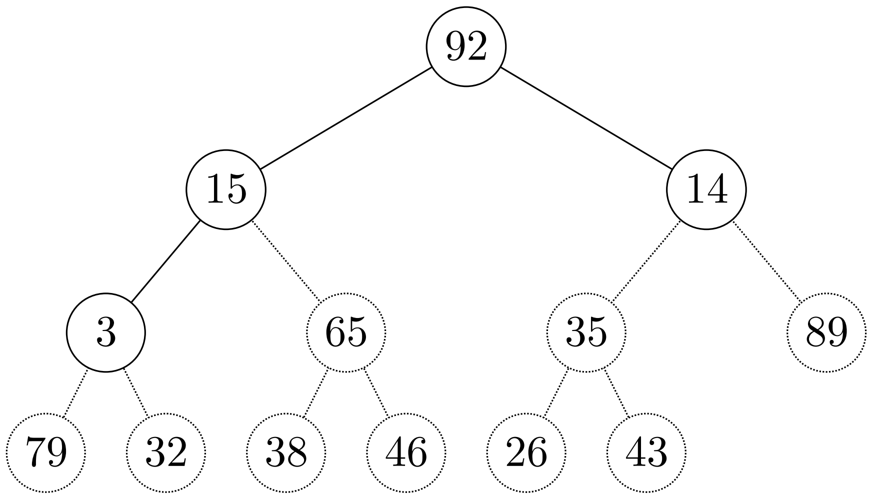
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**14**	**3**	15	92	65	35	89	79	32	38	46	26	43

Figure 16: Top-down heap construction - Step 2



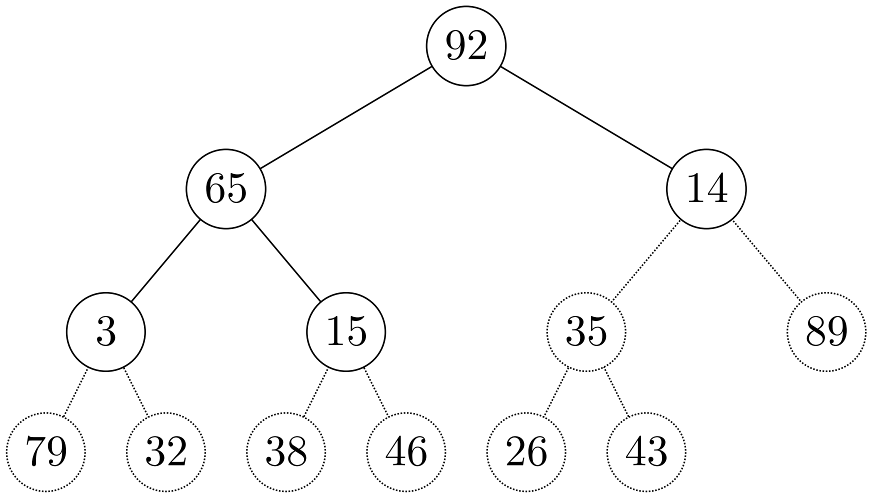
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**15**	**3**	**14**	92	65	35	89	79	32	38	46	26	43

Figure 17: Top-down heap construction - Step 3



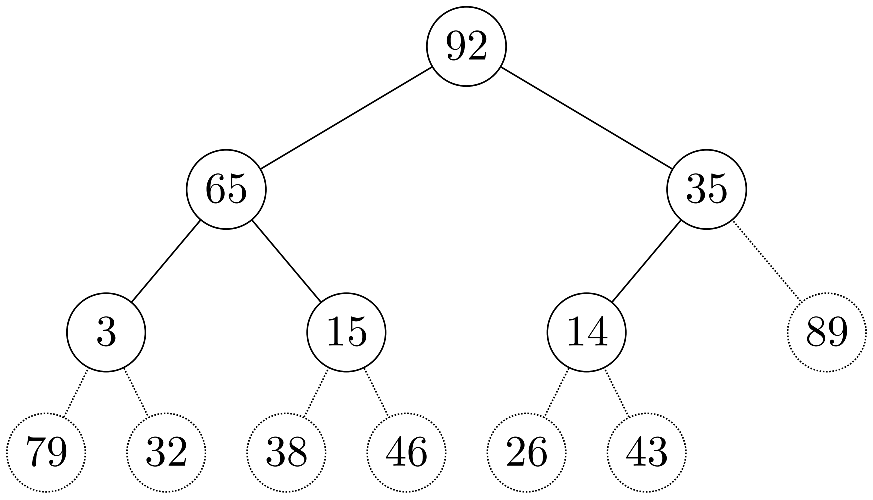
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**15**	**14**	**3**	65	35	89	79	32	38	46	26	43

Figure 18: Top-down heap construction - Step 4



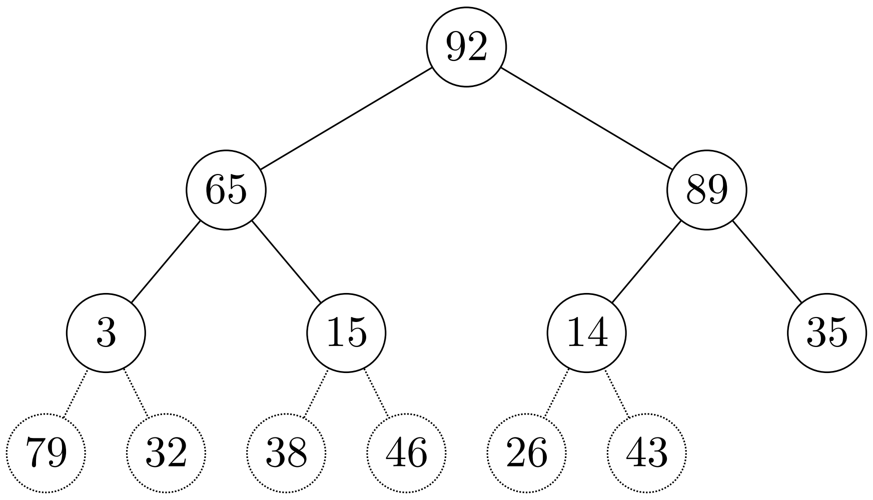
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**65**	**14**	**3**	**15**	35	89	79	32	38	46	26	43

Figure 19: Top-down heap construction - Step 5



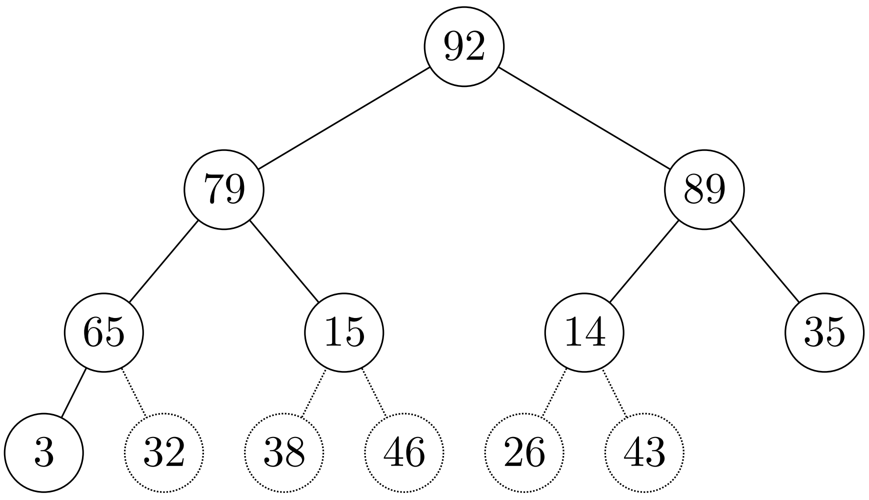
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**65**	**35**	**3**	**15**	**14**	89	79	32	38	46	26	43

Figure 20: Top-down heap construction - Step 6



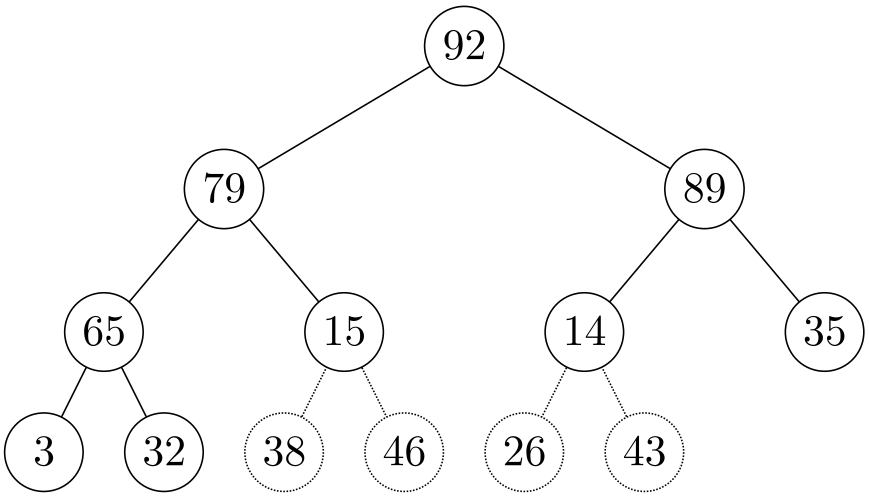
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**65**	**89**	**3**	**15**	**14**	**35**	79	32	38	46	26	43

Figure 21: Top-down heap construction - Step 7



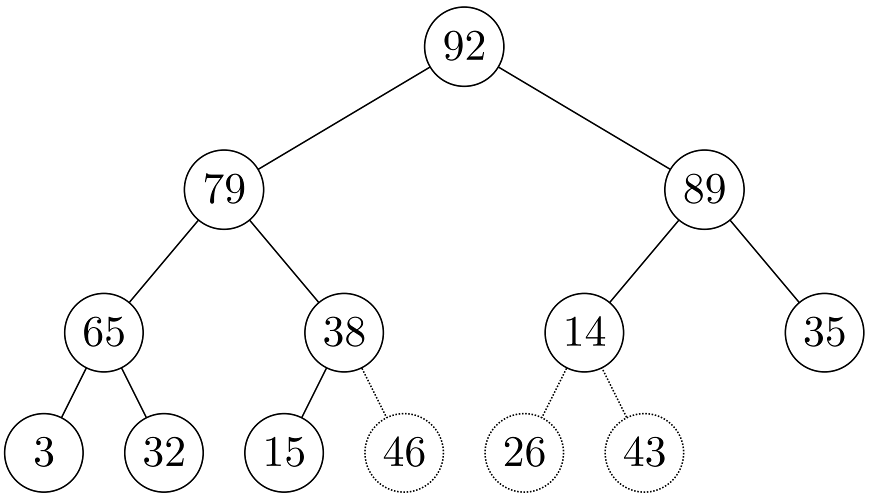
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**15**	**14**	**35**	**3**	32	38	46	26	43

Figure 22: Top-down heap construction - Step 8



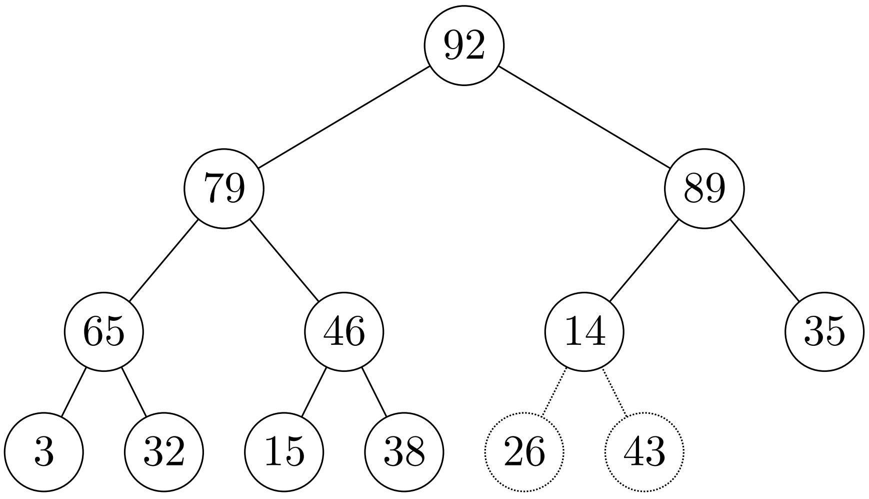
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**15**	**14**	**35**	**3**	**32**	38	46	26	43

Figure 23: Top-down heap construction - Step 9



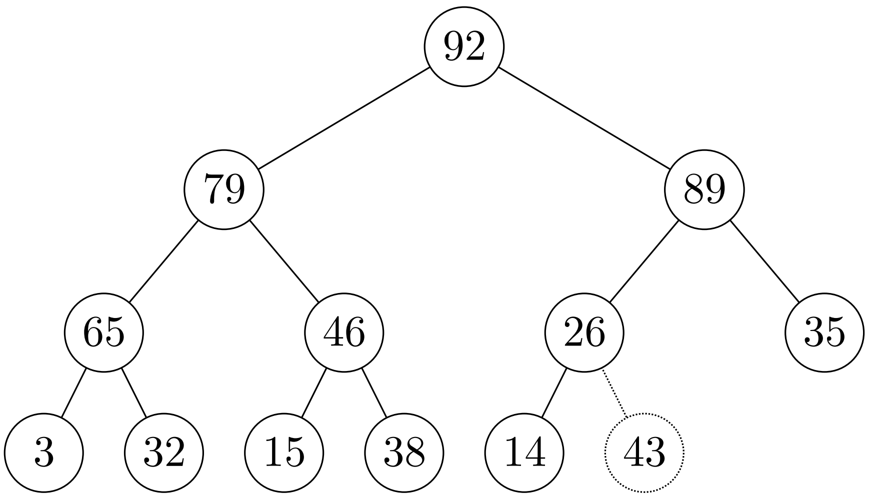
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**38**	**14**	**35**	**3**	**32**	**15**	46	26	43

Figure 24: Top-down heap construction - Step 10



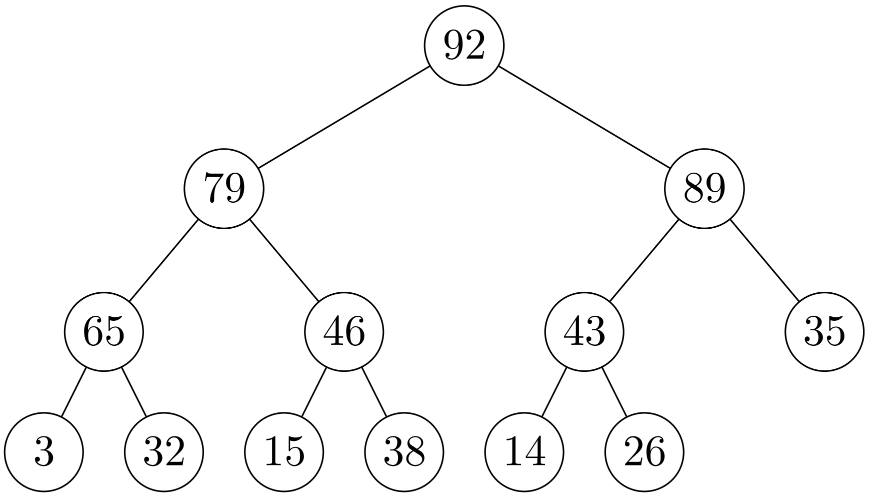
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**46**	**14**	**35**	**3**	**32**	**15**	**38**	26	43

Figure 25: Top-down heap construction - Step 11



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**46**	**26**	**35**	**3**	**32**	**15**	**38**	**14**	43

Figure 26: Top-down heap construction - Step 12



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**65**	**46**	**43**	**35**	**3**	**32**	**15**	**38**	**14**	**26**

Figure 27: Top-down heap construction - Step 13

The running time of the top-down construction is $O(n \log n)$, since the process is identical to a sequence of n heap insertions. Another approach to transforming the input array A into a heap is the *bottom-up* approach. Rather than build a heap by inserting a sequence of elements, the bottom-up approach recursively transforms each subtree into a heap, then merges the resulting heaps together until the entire array is a valid heap. If the two subtrees T_1 and T_2 of a node v are both heaps, then the subtree rooted at v can be converted to a heap with a single 'bubble-down'[24] operation. Algorithm 17 gives a recursive method for the bottom-up construction. Algorithm 17 builds a max-heap, but can easily be adapted to build a min-heap instead.

Algorithm 17 Bottom-up Max-Heap Construction (recursive)

```
procedure BubbleDownRecursive($A, n, i$)
    if $2i > n$ then
        {Node $i$ is a leaf.}
        return
    end if
    left ← $2i$
    right ← $2i + 1$
    max ← left
    if right $\leq n$ and $A$[max] $<$ $A$[right] then
        max ← right
    end if
    if $A[i] < A$[max] then
        Swap $A[i]$ and $A$[max]
        BubbleDownRecursive($A, n, max$)
    end if
end procedure
procedure HeapifyRecursive($A, n, i$)
    if $2i > n$ then
        {Node $i$ is a leaf, which is already a valid heap of size 1.}
        return
    end if
    {Recursively heapify the left subtree}
    HeapifyRecursive($A, n, 2i$)
    {Recursively heapify the right subtree}
    HeapifyRecursive($A, n, 2i + 1$)
    {'Bubble-down' the element at index $i$}
    BubbleDownRecursive($A, n, i$)
end procedure
{Initial Recursive Call - Not part of the algorithm itself}
HeapifyRecursive($A, n, 1$)
```

Algorithm 18 gives an iterative version of algorithm 17. To ease the analysis process, the 'bubble-down' mechanism has been integrated into the inner loop of algorithm 18.

24. Also called 'Heap Down', 'Sift Down', 'Shuffle Down', 'Sink', etc.

Algorithm 18 Bottom-up Max-Heap Construction (iterative)

```
procedure HeapifyIterative($A, n$)
    {The leaves of the tree are already valid heaps of size 1,}
    {so we can start from the last internal node.}
    {If the index of the last internal node cannot be easily}
    {computed, we can also take last to be $n$}
    last ← Index of last internal node
    for $i = \texttt{last}, \texttt{last} - 1, \dots, 1$ do
        {At this step, both subtrees of node $i$ are heaps, so we}
        {'bubble-down' the element at index $i$.}
        $j \leftarrow i$
        while true do
            left ← $2j$
            right ← $2j + 1$
            if left $> n$ then
                {If left $> n$, then $j$ is a leaf}
                Break
            end if
            max ← left
            if right $\leq n$ and $A$[max] $<$ $A$[right] then
                max ← right
            end if
            if $A[j] <$ $A$[max] then
                Swap $A[j]$ and $A$[max]
                $j \leftarrow$ max
            else
                {If no swap is needed, the heap property has been restored.}
                Break
            end if
        end while
    end for
end procedure
```

At first glance, the nested loops of algorithm 18 may appear to imply a $O(n^2)$ running time. However, the inner loop implements the 'bubble-down' operation, which is $O(\log n)$ in the worst case, so it should be reasonably intuitive that algorithm 18 is $O(n \log n)$. With a more thorough analysis, we can establish that algorithm 18 is $O(n)$.

In the worst case, every element i processed by the outer loop of algorithm 18 must be brought to the bottom of the heap, taking $O(\log q)$ time, where q is the distance between i and the lowest level of the tree. For example, this worst case occurs when the input array is already sorted in ascending order (or, more generally, if the input array corresponds to a min-heap instead of a max-heap). Let $d = \lfloor \log_2 n \rfloor$ be the depth of the tree. At level k of the tree[25], there are 2^{k-1} nodes and the distance to the lowest level of the tree is $d - k$, so the total number of swaps needed if every node on level k must be lowered to the bottom of the tree is $2^{k-1}(d-k)$. Adding

25. Where the root is considered to be at level 0, following the convention of section 12.1.

up the counts for all levels $k = 0, \dots, d-1$ gives

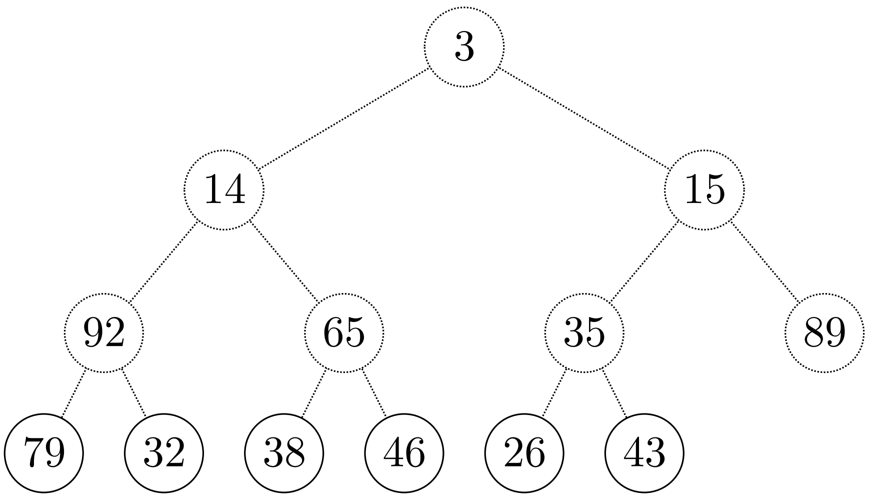
$$\sum_{i=0}^{d-1} 2^i(d-i) = \sum_{i=1}^{d} i2^{d-i} = 2^d \sum_{i=1}^{d} \frac{i}{2^i}$$

The value 2^d is the total number of internal nodes of the heap, which is $O(n)$, and due to the identity

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

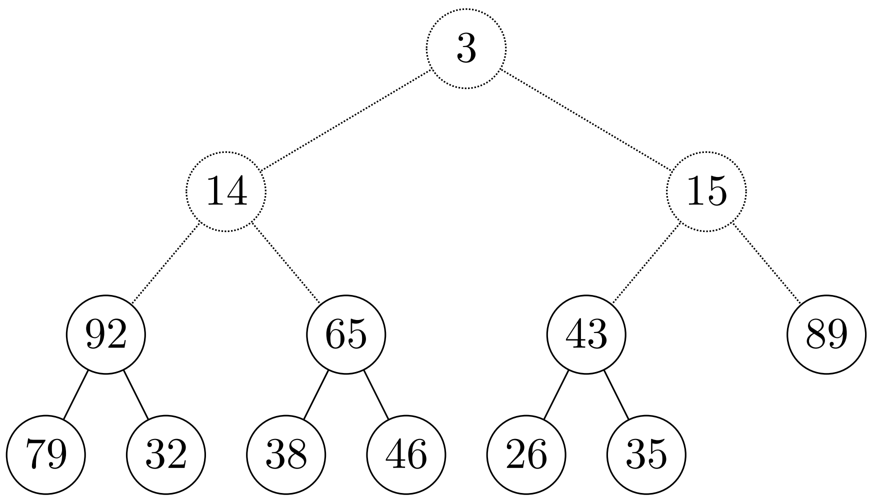
we can conclude that algorithm 18 is $O(n)$.

Figures 28 through 31 show the bottom-up construction on the input array A from the earlier example.



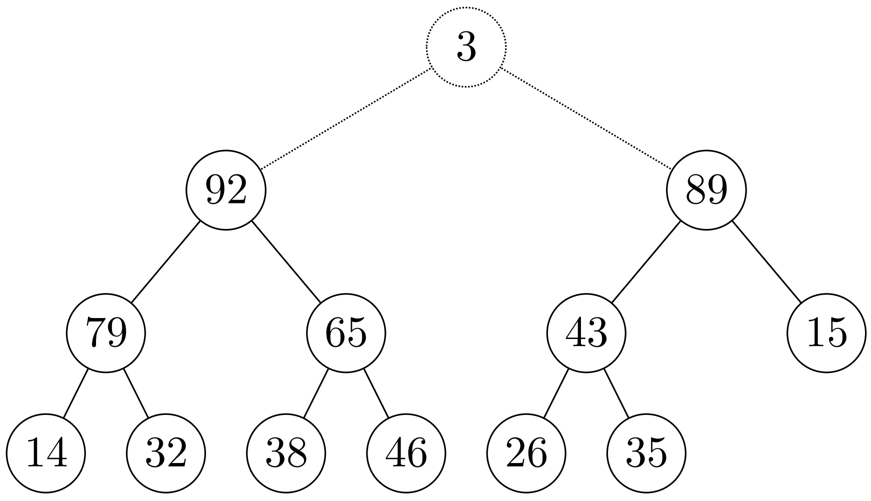
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	14	15	92	65	35	89	**79**	**32**	**38**	**46**	**26**	**43**

Figure 28: Bottom-up heap construction - Step 1. The nodes at the last level are all valid heaps of size 1.



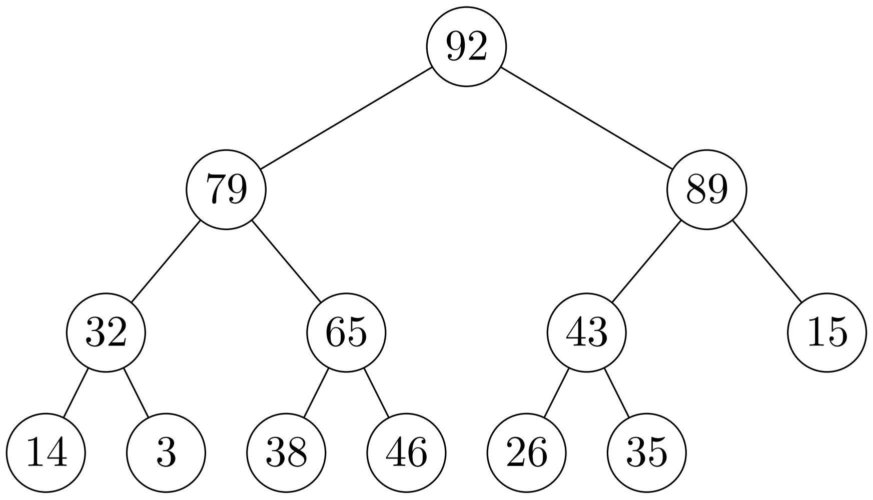
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	14	15	**92**	**65**	**43**	**89**	**79**	**32**	**38**	**46**	**26**	**35**

Figure 29: Bottom-up heap construction - Step 2. The subtrees on the last 2 levels are now valid heaps.



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	3	**92**	**89**	**79**	**65**	**43**	**15**	**14**	**32**	**38**	**46**	**26**	**35**

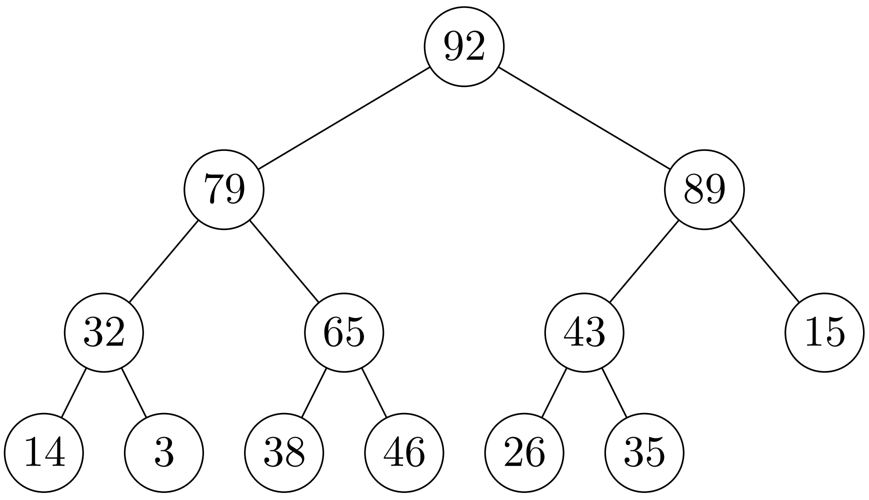
Figure 30: Bottom-up heap construction - Step 3. The subtrees on the last 3 levels are now valid heaps.



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**32**	**65**	**43**	**15**	**14**	**3**	**38**	**46**	**26**	**35**

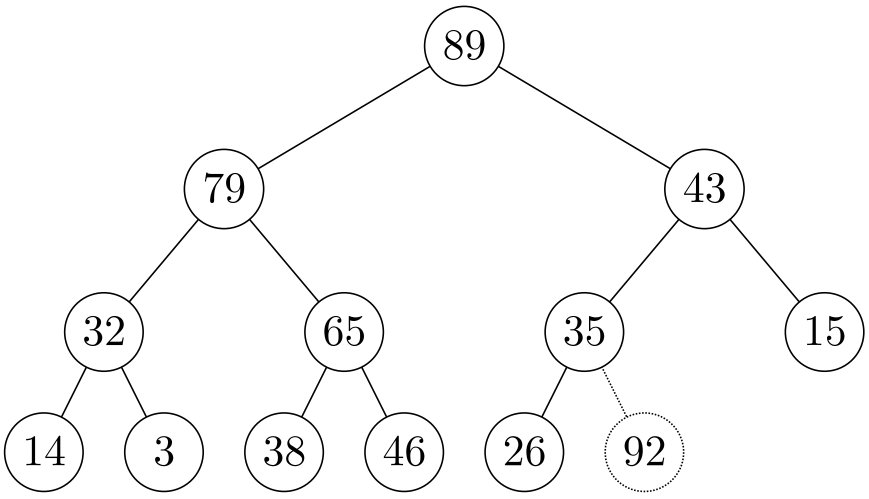
Figure 31: Bottom-up heap construction - Step 4. The entire tree is now a valid heap.

After the input array is transformed into a valid max-heap, a sequence of n remove-maximum operations is used to extract successive maximum objects from the heap. Since heaps are always complete binary trees, after every removal operation the array cell corresponding to the lower-right leaf will become free, and we can store the removed maximum element in this cell. Since the value of the maximum element in the heap decreases after every removal, the set of array cells filled by the removed maximum elements will be a sorted sequence, and when the process finishes, the array will be sorted. Since each heap removal operation takes $O(\log n)$ time, the total time complexity of the sequence of remove operations is $O(n \log n)$. Figures 32 through 44 illustrate the removal process, starting from the heap in figure 31 produced by the bottom-up construction.



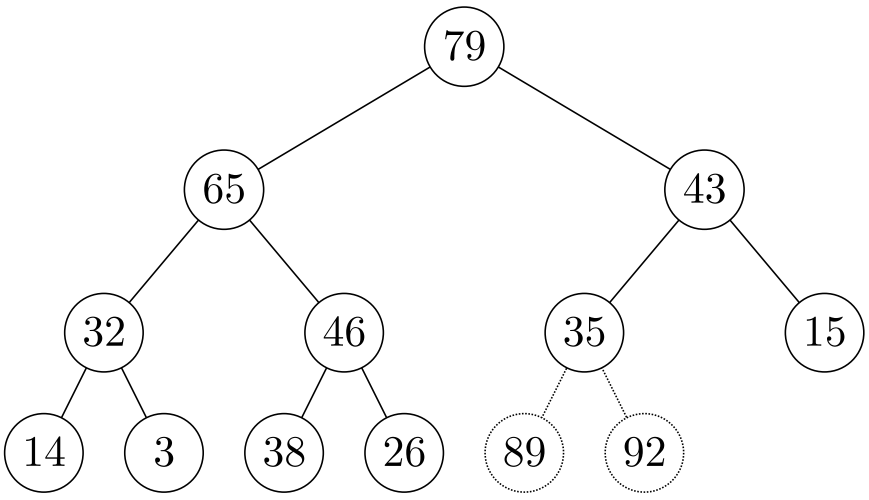
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**92**	**79**	**89**	**32**	**65**	**43**	**15**	**14**	**3**	**38**	**46**	**26**	**35**

Figure 32: Heap sort - Step 1



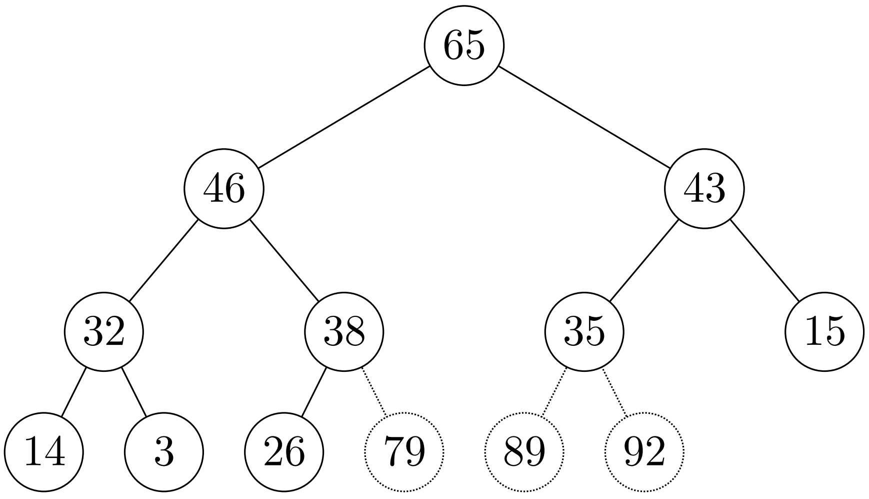
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**89**	**79**	**43**	**32**	**65**	**35**	**15**	**14**	**3**	**38**	**46**	**26**	92

Figure 33: Heap sort - Step 2



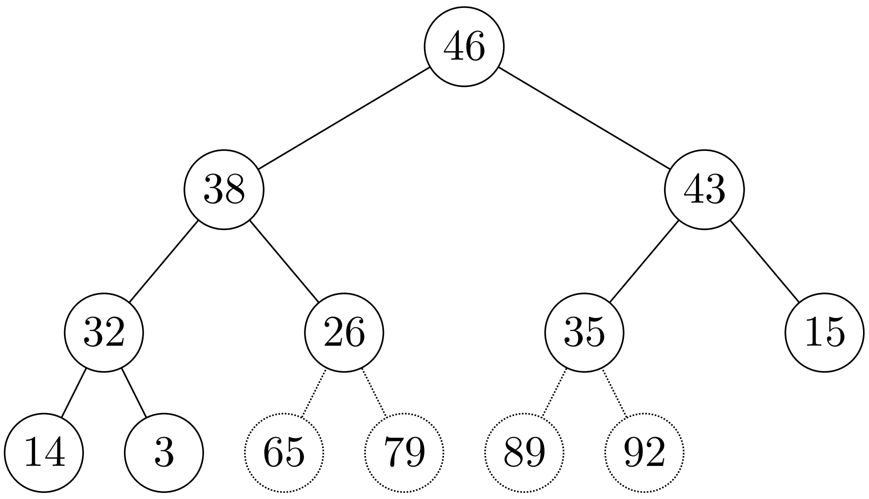
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**79**	**65**	**43**	**32**	**46**	**35**	**15**	**14**	**3**	**38**	**26**	89	92

Figure 34: Heap sort - Step 3



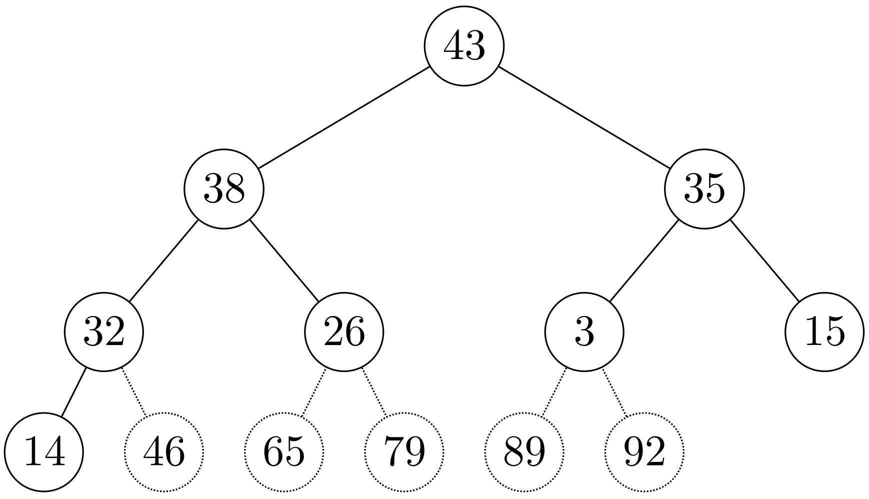
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**65**	**46**	**43**	**32**	**38**	**35**	**15**	**14**	**3**	**26**	79	89	92

Figure 35: Heap sort - Step 4



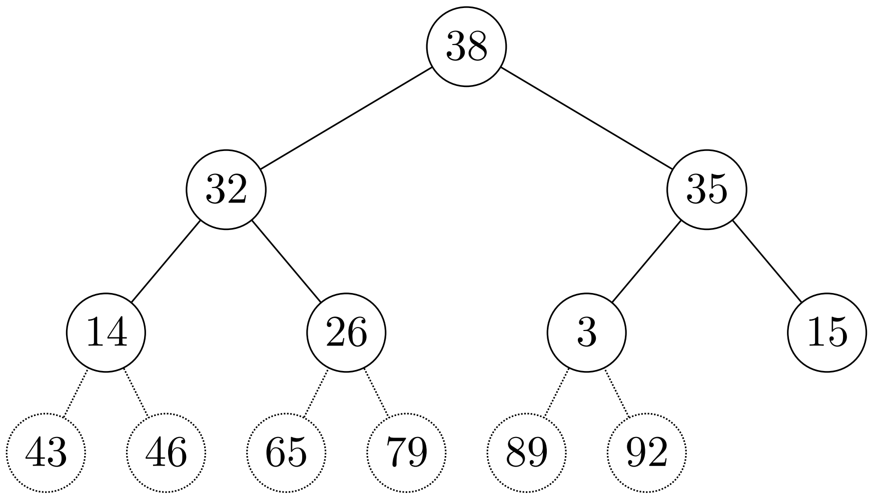
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**46**	**38**	**43**	**32**	**26**	**35**	**15**	**14**	**3**	65	79	89	92

Figure 36: Heap sort - Step 5



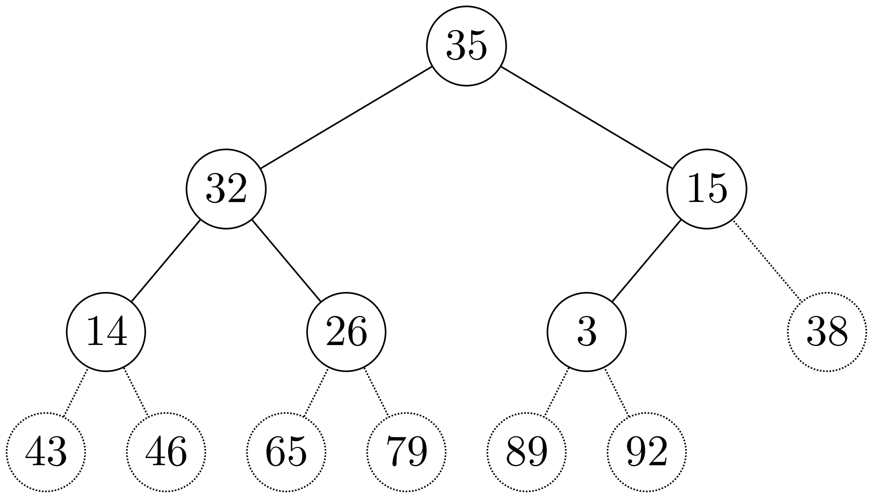
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**43**	**38**	**35**	**32**	**26**	**3**	**15**	**14**	46	65	79	89	92

Figure 37: Heap sort - Step 6



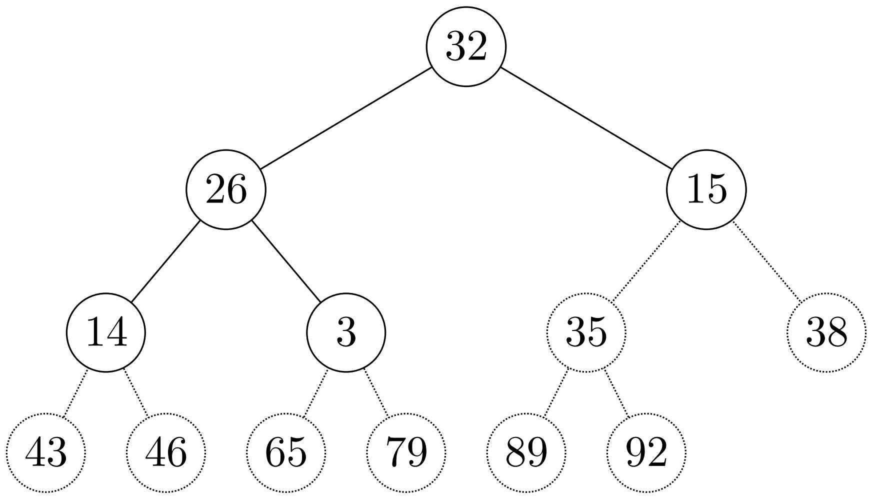
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**38**	**32**	**35**	**14**	**26**	**3**	**15**	43	46	65	79	89	92

Figure 38: Heap sort - Step 7



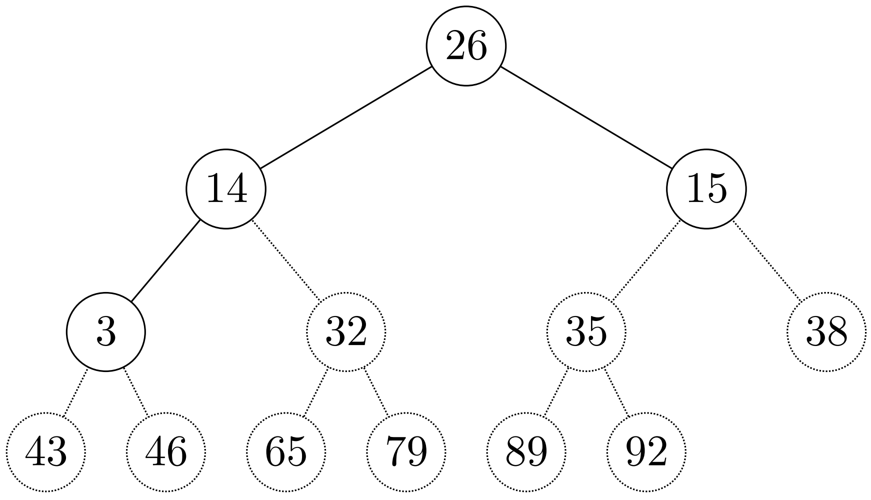
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**35**	**32**	**15**	**14**	**26**	**3**	38	43	46	65	79	89	92

Figure 39: Heap sort - Step 8



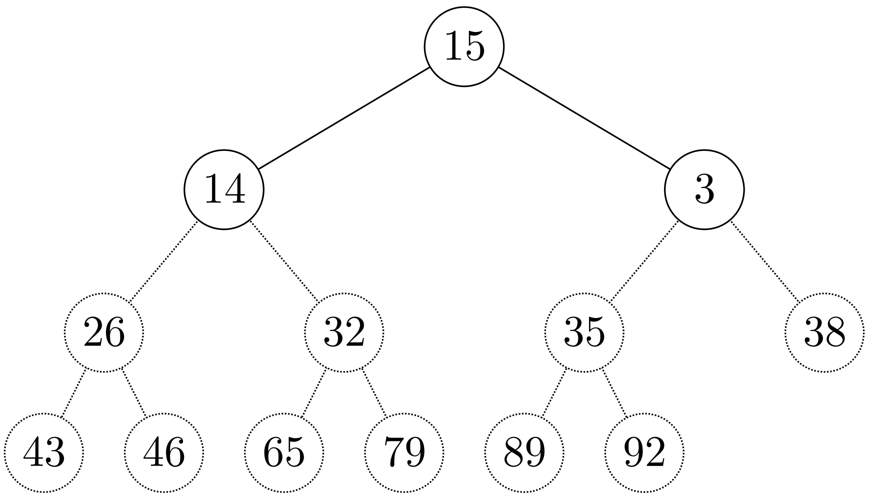
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**32**	**26**	**15**	**14**	**3**	35	38	43	46	65	79	89	92

Figure 40: Heap sort - Step 9



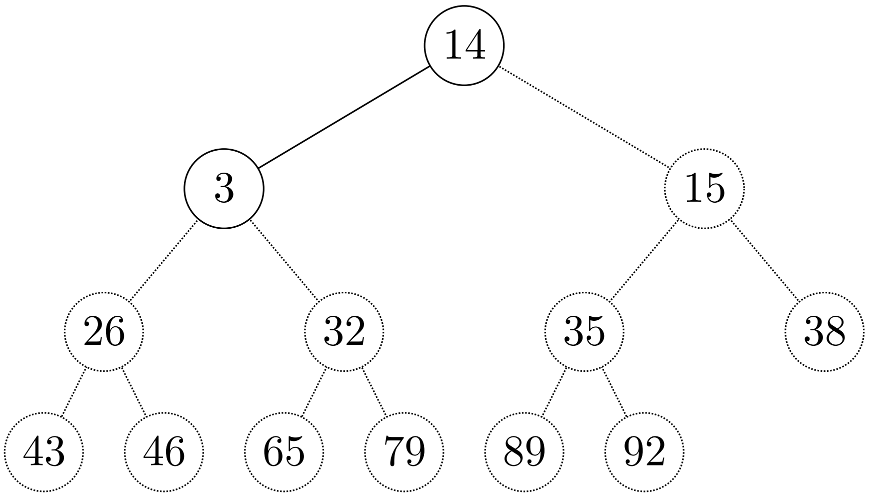
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**26**	**14**	**15**	**3**	32	35	38	43	46	65	79	89	92

Figure 41: Heap sort - Step 10



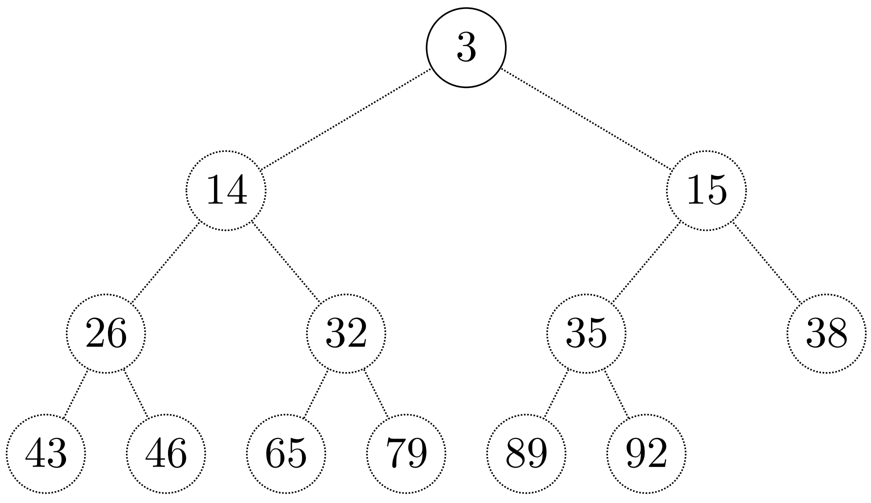
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**15**	**14**	**3**	26	32	35	38	43	46	65	79	89	92

Figure 42: Heap sort - Step 11



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**14**	**3**	15	26	32	35	38	43	46	65	79	89	92

Figure 43: Heap sort - Step 12



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	**3**	14	15	26	32	35	38	43	46	65	79	89	92

Figure 44: Heap sort - Step 13

14 k-way Merging

The merge sort algorithm gets its name from the 2-way merge process used to combine the recursively sorted sub-lists to produce the sorted output array. For sub-lists with a combined length of n, the merge step is known to have a $O(n)$ running time. We consider the generalized problem of merging an arbitrary number k of sub-lists with a total of n elements.

The simplest way to perform a k-way merge is to extend the 2-way merge to k arrays, as given in algorithm 19. The parameters $L_1, \ldots, L_k$ are the set of sub-lists, and it is assumed that removal from the front of each list is a $O(1)$ operation. The parameter A is the output array.

Algorithm 19 k-way Merge (simple)

```
procedure KWaySimple($n, A, L_1, \ldots, L_k$)
    for $i = 0, \ldots, n-1$ do
        minIdx $\leftarrow 0$
        for $j = 1, \ldots, k$ do
            if $L_j$ is empty then
                Continue
            end if
            if minIdx $= 0$ or Front($L_j$) $<$ Front($L_{\texttt{minIdx}}$) then
                minIdx $\leftarrow j$
            end if
        end for
        $A[i] \leftarrow$ RemoveFront($L_{\texttt{minIdx}}$)
    end for
end procedure
```

Algorithm 19 is $O(nk)$. In practice, the number of sublists k tends to be fixed, and an implementation of algorithm 19 for a fixed k will technically be $O(n)$.

The inner loop in algorithm 19 finds the minimum of a set of k elements. In general, this problem is $O(k)$ (since each element must be examined at least once). However, for k-way merging, the set of elements inspected by the inner loop on each iteration of the outer loop differs from the set inspected on the previous iteration by only one value (since only one sub-list is changed on each iteration of the outer loop). The relative ordering of the first elements of the unchanged lists remains the same between iterations. Using an auxiliary data structure, it is possible to avoid having to inspect every sub-list on every iteration, giving a $O(n \log k)$ algorithm for the k-way merge problem.

At each iteration of the outer loop, we want to find the sub-list whose first element is minimum. While there are several data structures that will perform this task in $O(\log k)$ time, the simplest and most obvious choice is a heap. In particular, we create a heap to store pairs (e, j), where $e = \text{FRONT}(L_j)$, for $j = 1, \ldots, k$. We use a heap comparator which only considers the value e, and the value j is kept only as a 'tag' to identify the source of e.

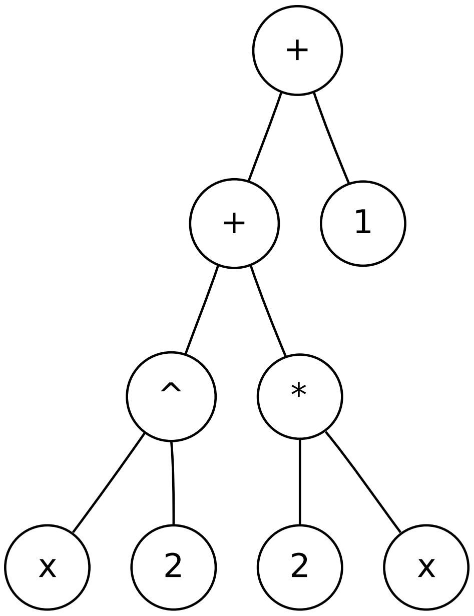
Before the merge process starts, we insert a pair (e, j) for all sub-lists. At each iteration of the merge process, we remove the pair (e, j) with the minimum e in $O(\log k)$ time from the heap, then add a new pair (e', j) containing the next element in L_j to the heap in $O(\log k)$ time. Each sub-list is accessed only when one of its elements is merged into the output array, so the total number of sub-list accesses is $O(n)$, giving a total running time of $O(n \log k)$. Algorithm 20 gives pseudocode for this procedure.

Algorithm 20 k-way Merge (fast)

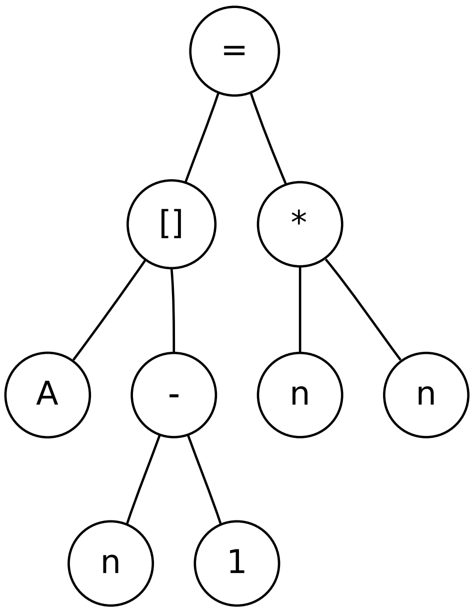
```
procedure KWAYFAST($n, A, L_1, \ldots, L_k$)
    Create a heap $H$ to contain pairs $(e, j)$, with the comparator considering only $e$.
    for $j = 1, \ldots, k$ do
        if $L_j$ is empty then
            Continue
        end if
        $e \leftarrow$ REMOVEFRONT($L_j$)
        Add the pair $(e, j)$ to $H$
    end for
    for $i = 0, \ldots, n - 1$ do
        Remove the minimum $(e, j)$ pair from $H$
        if $L_j$ is not empty then
            $e' \leftarrow$ REMOVEFRONT($L_j$)
            Add the pair $(e', j)$ to $H$
        end if
        $A[i] \leftarrow e$
    end for
end procedure
```

15 Parse Trees

Parsers for programming languages and query engines break a complex expression, such as "$x^2 + x + 1$" or "`A[0] = 10`", into a series of binary operations[26], forming a parse tree. Figure 45 shows parse trees for a mathematical expression and a C/Java statement.



(a) The expression $x^2 + 2x + 1$



(b) The C/Java statement `A[n-1] = n*n`

Figure 45: Parse trees for two types of expressions.

While the parsing itself can be difficult, evaluating the expression is usually very simple, since the operation at each node can be evaluated by recursively evaluating the subtrees and applying the operator to the result. Object oriented languages are particularly well suited to concise recursive evaluation code, since a separate class can be used for each node type, with an overridden `eval` method which defines the specific behavior of the operator. Ignoring the type information of methods, a base class for a parse tree node might resemble the following:

```
class TreeNode{
    TreeNode left, right;
    ...
    //eval() - Evaluate the expression rooted at this node and
    //return the result.
    //This method will be overridden by each subclass.
    eval();

}
```

A subclass of `TreeNode` which models the behavior of the `[]` operator in figure 45b, which

26. Some languages have ternary operators, which cannot be parsed as binary operations. One such operator is the ternary conditional operator in Java and C (seen in expressions such as `(x > y)? x: y`, which evaluates to the maximum of `x` and `y`). To accommodate these operators in a binary parse tree, one workaround is to treat the operator as a pair of binary operators, and add an extra step in the parsing process which verifies that the two operators always appear together.

indexes the array specified by the right subtree with the numerical value specified by the left subtree, would then be implemented with code similar to the following:

```
class SquareBracketOperator extends TreeNode{
    ...
    eval(){
        array = left.eval();
        index = right.eval();
        val = array[index];
        return val;
    }

}
```

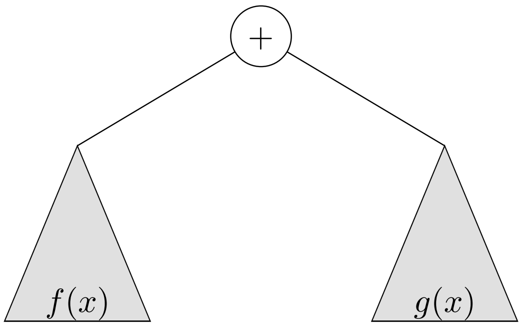
While the above code is sufficient to evaluate the expression, a C or Java compiler uses parse trees to generate code which evaluates an expression. However, besides the obvious added complication of generating assembly (or bytecode) instructions, the process is very similar.

16 Symbolic Algebra

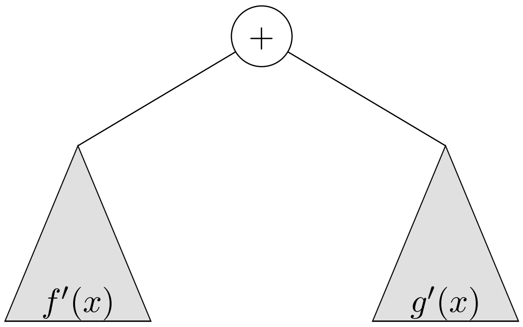
The parse tree in figure 45a describes a mathematical expression, and section 15 illustrates how a simple recursive procedure can be used to evaluate parse trees. We can also use parse trees for more advanced mathematical operations. Consider the basic rules for differentiation:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}(c) &= 0 \\
\frac{\mathrm{d}}{\mathrm{d}x}(f(x)+g(x)) &= f'(x)+g'(x) \\
\frac{\mathrm{d}}{\mathrm{d}x}(f(x)g(x)) &= f'(x)g(x)+f(x)g'(x) \\
\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{f(x)}{g(x)}\right) &= \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2} \\
\frac{\mathrm{d}}{\mathrm{d}x}\left(f(x)^k\right) &= kf'(x)f(x)^{k-1}
\end{aligned}$$

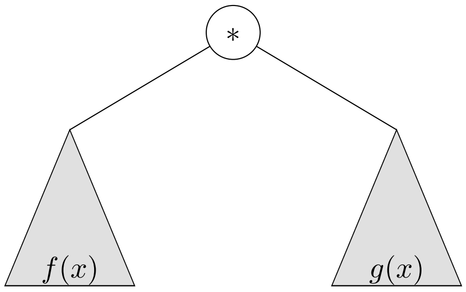
Since these differentiation rules are all ‘local’ to their respective operators (for example, the product rule requires only the two operands of the multiplication operator), we can define rules for each operator in a parse tree of a function $F(x)$ which produce a new tree for the derivative $F'(x)$. The tree transformations for each differentiation rule are given in figure 46.



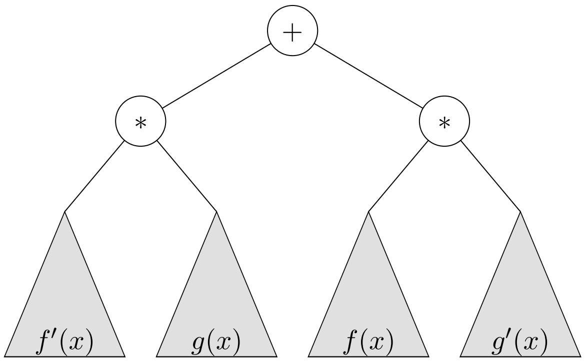
(a) $F_1(x) = f(x) + g(x)$



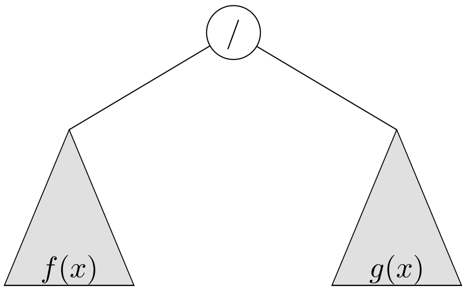
(b) $F_1'(x) = f'(x) + g'(x)$



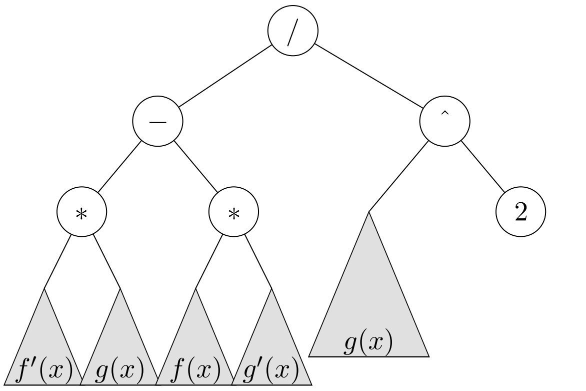
(c) $F_2(x) = f(x)g(x)$



(d) $F_2'(x) = f'(x)g(x) + f(x)g'(x)$



(e) $F_3(x) = f(x)/g(x)$



(f) $F_3'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$

Figure 46: Parse trees for various expressions and their derivatives. Shaded triangles denote subtrees.

Using the `TreeNode` interface defined in section 15 as a base class, the differentiation operation for the + operator can be implemented with code similar to the following:

```
class PlusOperator extends TreeNode{
    TreeNode left, right;
    ...
    PlusOperator(TreeNode left, TreeNode right){
        this.left = left;
        this.right = right;
    }
    ...
    differentiate(){
        d_left = left.differentiate();
        d_right = right.differentiate();
        return new PlusOperator(d_left,d_right);
    }
}
```

17 Tries

A prefix tree, or *trie*[27], stores a set S of n strings of length at most k in a tree of depth k such that for any prefix p of a string in S, there is a unique path from the root of the tree to the root of a subtree containing every string in S prefixed by p. Every non-root node stores a single letter, and the non-root nodes in a path of length l from the root to a leaf form a string in S. Prefix trees provide an efficient means of finding the set of strings which share a common prefix. One application of prefix trees is the auto-complete feature of command shells.

A trie for some common Unix commands is given in figure 47. If the user types 'g' and requests auto-completion, the three strings 'grep', 'gcov' and 'gcc' would be returned as possible matches.

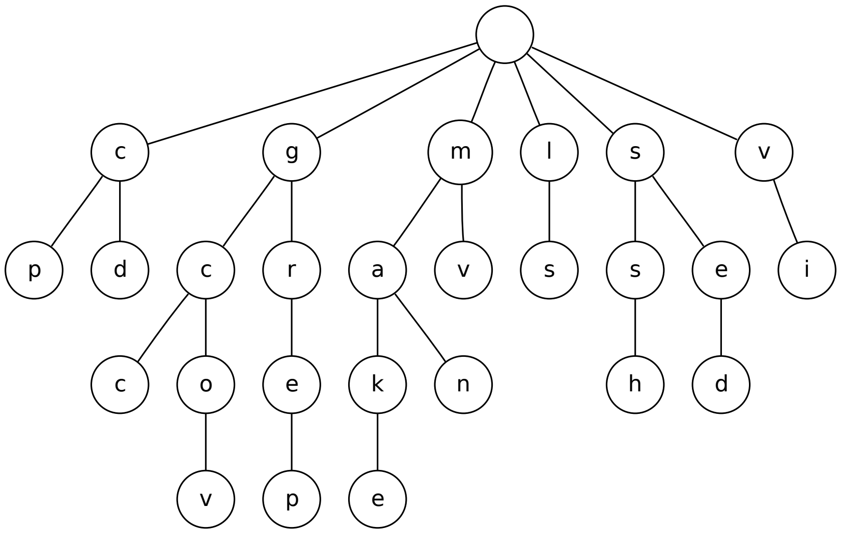


Figure 47: A prefix tree for some common Unix commands.

18 Binary Space Partitions

An important problem in 3d graphics applications is how to efficiently determine which parts of a (fixed) scene are visible from a given point in the scene, and how to efficiently obtain a 'front-to-back' ordering of the objects in the scene from the perspective of that point. Figure 48 shows the floor plan of a 3d environment and a point p denoting an observer of that environment[28].

27. Pronounced 'try', even though the term 'trie' comes from the word 'retrieval'
28. Note that the direction the observer is facing is not given here, so we consider all possible orientations of the observer.

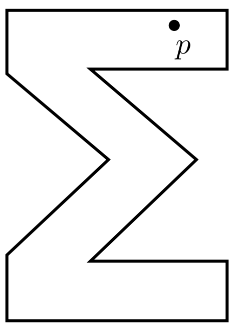


Figure 48: The floor plan of a 3d scene and the position of an observer p.

Determining the exact set of points visible from p is time consuming, but it is desirable to exclude regions which are definitely not visible, to avoid processing parts of the scene which will never be drawn. For example, an observer at p will never have a line of sight to any points in the lower area of the scene.

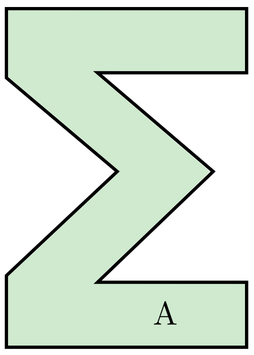
Additionally, it is important that when the scene is rendered, objects are drawn in the right order. That is, we want objects which are closer to p to be drawn in front of objects which are further away. The naive solution to this problem is to sort every object in the scene by the distance between it and p. However, since the position of p may change, this is not feasible when the number of objects is large.

A binary space partition is a set of lines (or planes) in space which subdivide a region into convex sub-regions. The relationship of lines and regions can be modeled as a binary tree, called a BSP-tree, and the region containing a point p can be identified in time proportional to the depth of the tree.

In applications where the geometry of the environment is known in advance (such as 3d games with pre-compiled maps), a pre-processing phase can be used to compute visibility information for each convex region in the binary space partition. At run-time, the region containing the observer can be quickly identified and the associated visibility information retrieved.

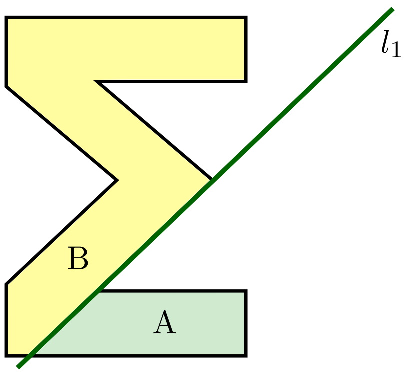
The structure of a binary space partition also provides information about the relative ordering of objects in neighbouring regions, which partially solves the front-to-back ordering problem.

To construct a binary space partition and the associated BSP-tree, the region is repeatedly divided by splitting planes, which become the internal nodes of the BSP-tree. The regions formed on each side of the splitting plane become leaves of the BSP-tree. Any plane can be used as a splitting plane, and certain choices for splitting planes may produce better results. Algorithms which compute a BSP-tree automatically from a polygon normally use the edges of the polygon (that is, the walls of the scene) as splitting planes. Figure 49 shows the region from figure 48 and associated BSP-tree before and after the first division.

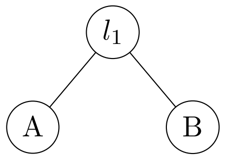


(a) Region before first division

(b) BSP-tree before first division



(c) Region after first division

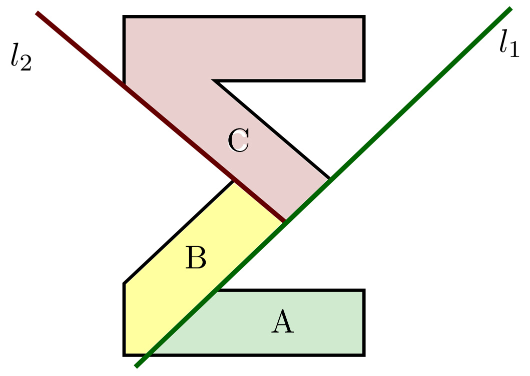


(d) BSP-tree after first division

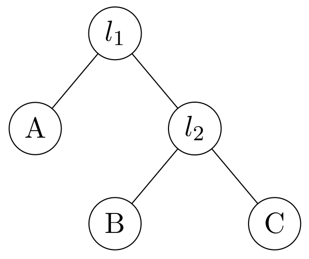
Figure 49: The first division of a binary space partition.

For each splitting plane, we define a 'right' side and 'left' side. For general BSP-trees, this can be done by choosing a reference normal vector to the plane, and defining the 'left' side to contain all points on the same side of the plane as the normal vector. To avoid overcomplicating things, we will define the 'left' side of our splitting lines to be all points below the line, and the 'right' side of our splitting lines to be all points on or above the line. Using this definition, we can find the region containing a point p by starting at the root of the BSP-tree and navigating to a leaf. At each internal node, corresponding to a splitting line l, we follow the left path if p is on the left side of l and the right path if p is on the right side of l. For the BSP-tree given in figure 49d and the point p in figure 48, this traversal arrives at region B.

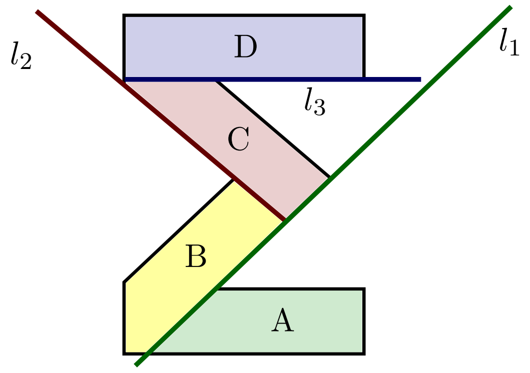
To build the complete binary space partition, we recursively split each region until all leaves of the BSP-tree are convex regions. Figure 50 shows construction of the completed BSP-tree.



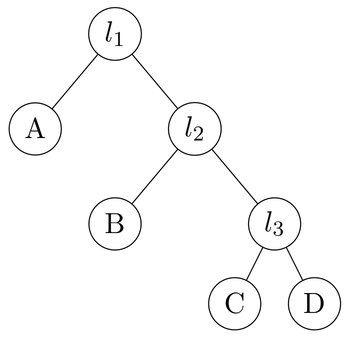
(a) Region after second division



(b) BSP-tree second second division



(c) Region after third division



(d) BSP-tree after third division

Figure 50: The second and third divisions of a binary space partition.

If this binary space partition is intended for use in a 3d game, we can now compute a visibility table, which gives the set of regions R' visible from each region R. We consider R' to be visible from R if there exists any point in R from which a point in R' can be seen. This computation can be very tedious, but it can be performed in advance (such as when the scene is designed). For the convex regions in figure 51, we have the visibility table

	A	B	C	D
A	1	1	1	0
B	1	1	1	1
C	1	1	1	1
D	0	1	1	1

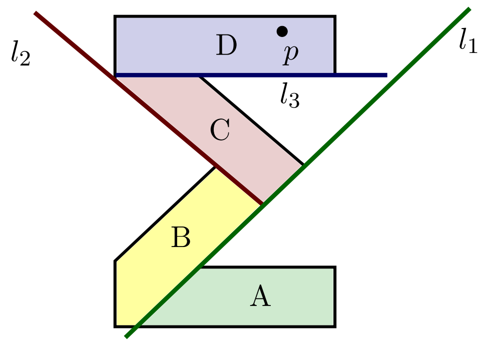


Figure 51: The floor plan of a 3d scene and the position of an observer p.

Figure 51 shows the point p from figure 48 in the completed binary space partition. To find the region containing p, we traverse the BSP-tree:

- We start at the root, which is an internal node corresponding to line l_1.
- p is above line l_1, so we take the right path and arrive at node l_2.
- p is above line l_2, so we take the right path to l_3.
- p is above line l_3, so we take the right path to node D.
- D is a leaf, so it is the convex region containing p.

To draw the scene from the perspective of point p, we can check the visibility table to determine which regions are visible from p. Since A is not visible from p, we can completely ignore A during the rendering process. To determine which order to draw the remaining regions, we can traverse the BSP-tree starting at node D and moving outward in both directions. Since region C appears before region B in such a traversal, we know that C should appear in front of B.

19 The $\Omega(n \log n)$ Bound on Comparison Sorting

An input sequence

$$a_1, a_2, \ldots, a_n$$

of distinct objects from a totally-ordered collection S, has a unique sorted ordering

$$a_{k_1} < a_{k_2} < \ldots < a_{k_n}$$

where the sequence of indices k_i is a permutation of $\{1, \ldots, n\}$. Informally, the values k_i give the index of the i^{th} smallest value in the input sequence.

Since every a_i is distinct, we can assume that the input sequence $a_1, \ldots, a_n$ is a permutation of the integers $\{1, \ldots, n\}$, since the behavior of the comparator is identical[29]. For clarity, we can use the functional notation

$$A(i) = a_i, K(i) = k_i$$

to describe the sequences a_i and k_i.

Since $A(K(i)) = a_{k_i} = i$ for all i, it must be the case that $K(i) = A^{-1}(i)$, so the permutation $k_1 k_2 \ldots k_n$ is the inverse of the permutation $a_1 a_2 \ldots a_n$. We can similarly establish that the permutation $a_1 a_2 \ldots a_n$ is the inverse of $k_1 k_2 \ldots k_n$, either by proving that sorting the sequence $k_1, k_2, \ldots, k_n$ gives $k_{a_1}, k_{a_2}, \ldots, k_{a_n}$, or by using mathematical properties of permutations[30].

From the above, we can conclude that the process of sorting is at least as complicated as the process of finding the inverse of an arbitrary permutation. There are $n!$ permutations of $\{1, \ldots, n\}$, and since the inverse of a permutation is unique, a sorting algorithm must be able to uniquely discriminate between every possible ordering of input values. A comparison sorting algorithm is unable (in general) to use any information besides the result of a comparison of two elements to identify the ordering of input values. Each comparison of the form $a \leq b$ has two possible outcomes, '`true`' and '`false`', so a sequence of k comparisons can therefore identify at most 2^k distinct orderings. Since there are $n!$ possible permutations, it must be the case that

$$2^k \geq n!$$

29. More generally, any finite total order on n objects is isomorphic to the totally ordered set $\{1, \ldots, n\}$.

30. Specifically, the set of permutations of $\{1, \ldots, n\}$, denoted S_n, is a mathematical group. One property of a group is that each element has exactly one inverse, and that the inverse of an element's inverse is the element itself (i.e. $(A^{-1})^{-1} = A$).

Solving this inequality for k gives

$$k \geq \log(n!) = \sum_{i=1}^{n} \log i$$

Sections 19.1 and 19.2 give two different proofs that $\log(n!) \in \Theta(n \log n)$, but to prove the lower bound on comparison sorting it is only necessary to establish the $\Omega(n \log n)$ bound. In either case, since $k \in \Omega(n \log n)$, any comparison sorting algorithm must take $\Omega(n \log n)$ time in the worst case.

19.1 Proof Using Stirling's Approximation

Stirling's approximation gives

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

as n becomes large.[31] Taking the logarithm of both sides gives

$$\begin{aligned} \log n! &\approx \log \left[\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right] \\ &= \frac{\log(2\pi n)}{2} + n \log \left(\frac{n}{e}\right) \\ &= n \log n - n \log e + \frac{1}{2} \log(2\pi n) \end{aligned}$$

Since $n \log n - n \log e + \frac{1}{2} \log(2\pi n) \in \Theta(n \log n)$, we conclude that $\log n! \in \Theta(n \log n)$. Since Stirling's approximation is well-established, this version of the proof is very short and concise since there is no need to prove the approximation. However, to generate this proof during an exam it is necessary to have the identity memorized, whereas the proof in section 19.2, while more tedious, can be generated with logarithm rules and basic algebra.

19.2 Proof Using Laws of Logarithms

To prove that $\log(n!) \in \Theta(n \log n)$, we establish that $\log(n!) \in O(n \log n)$ and $\log(n!) \in \Omega(n \log n)$ separately[32].

$$\begin{aligned} \log n! &= \log(1 \cdot 2 \cdot \ldots n) \\ &= \log(1) + \log(2) + \ldots + \log(n) \\ &\leq \overbrace{\log(n) + \log(n) + \ldots + \log(n)}^{n} \\ &= n \log n \end{aligned}$$

31. The notation '$f(x) \approx g(x)$' is interpreted as 'f approximates g'. While this notation may be used loosely in other contexts, in this document we assume that the statement '$f(x) \approx g(x)$' always implies $f(x) \in \Theta(g(x))$. Other sources may use different notation for this property; one popular alternative is the '$\sim$' symbol, and the statement '$f(x) \sim g(x)$' is interpreted to mean 'f is asymptotically equal to g'. In a formal context, the $\approx$ symbol is considered 'stronger' than the $\sim$ symbol, since approximations are not typically permitted to differ from the target function by an arbitrary constant factor.

32. Recall from section 3.3 that $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Therefore, $\log(n!) \le n \log n$, so $\log(n!) \in O(n \log n)$.

$$\begin{aligned}
\log n! &= \log(1 \cdot 2 \cdot \ldots n) \\
&= \log(1) + \log(2) + \ldots + \log(\frac{n}{2}) + \ldots + \log(n) \\
&\ge \log(\frac{n}{2}) + \ldots + \log(n) \\
&\ge \overbrace{\log(\frac{n}{2}) + \log(\frac{n}{2}) + \ldots + \log(\frac{n}{2})}^{\frac{n}{2}} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&= \frac{1}{2} n \log n - \frac{1}{2} n \log(2)
\end{aligned}$$

Therefore, $\log(n!) \ge \frac{1}{2} n \log n - \frac{1}{2} n \log(2)$. The second term is asymptotically less than the first term (since the first term is $\Theta(n \log n)$ and the second term is $\Theta(n)$) and can therefore be ignored for asymptotic purposes[33], giving

$$\log(n!) \in \Omega(\frac{1}{2} n \log n) = \Omega(n \log n)$$

20 AVL Trees

Recall that a binary search tree is an AVL tree if and only if for every node v, the height of the left subtree of v differs from the height of the right subtree of v by at most 1. The performance of AVL trees is ensured by asserting that the height condition is invariant under all of the tree operations. That is, while none of the standard operations checks or enforces the condition, all operations are designed to guarantee that if the condition holds for all nodes before the operation, it will continue to hold afterwards.

Obviously, the tree operations which do not modify the tree (such as searching or traversal) will not affect whether the condition holds. For AVL trees, the only operations which modify the tree are insertion and removal. Even when the tree is modified, there is no guarantee that the height condition will necessarily be violated. For example, no insertion into the tree in figure 52 will create an imbalance at the root node, since even when the height of one subtree increases, the condition will hold (note that it is not possible for an insertion to decrease the height of a subtree).

The diagrams in this section use grey triangles to represent arbitary subtrees, which may be empty. In the context of AVL trees, it is assumed that the subtrees represented by grey triangles are balanced unless otherwise stated. Although the examples given here describe imbalances created at the root of the depicted tree, the examples are equally valid when the depicted tree is a subtree of some larger tree (that is, when the root of the depicted tree has a parent).

33. The limit condition from 3.1 can be used to verify that $\frac{1}{2} n \log n - \frac{1}{2} n \log(2) \in \Theta(\frac{1}{2} n \log n)$

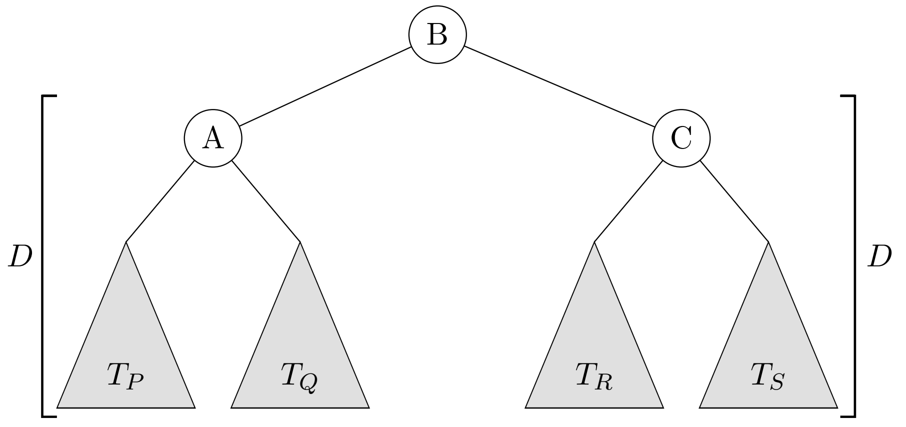


Figure 52: An AVL tree in which no insertion can create an imbalance at B.

The first step of an insertion into an AVL tree is to perform a binary search tree insertion operation. Afterwards, the height condition is checked at each node by tracing from the newly created node to the root of the tree, and a rebalancing operation is performed at the first imbalanced node encountered (that is, the node with the largest depth). As the example in figure 52 shows, rebalancing is not always necessary. When rebalancing is necessary, no more than one rebalance operation, requiring one or two rotations, is needed after an insertion. As section 20.4 illustrates, multiple rebalancing operations may be necessary after a removal operation. In any case, rebalancing operations are only performed for nodes along a path from a leaf to the root, and since AVL trees are balanced, such a path has length $O(\log n)$.

Consider the AVL tree in figure 53 and suppose that an insertion creates an imbalance at node B (without unbalancing any lower node). Since the subtree rooted at node A has height D, and the subtree at C has height $D+1$, the imbalance must be result of an insertion into either T_R or T_S.

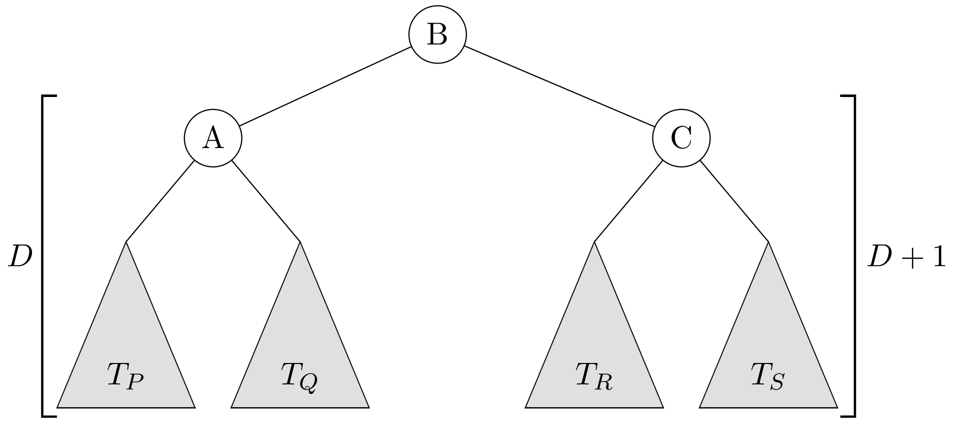


Figure 53: An AVL tree.

Section 20.1 describes the rebalancing required after an insertion into T_S and section 20.2 describes the rebalancing required after an insertion into T_R. Insertions into T_P and T_Q require similar rebalancing.

20.1 Single Rotation Rebalancing

Suppose that an insertion into the subtree T_S of the tree in figure 53 creates an imbalance at B (without creating an imbalance at any lower node). Since such an insertion must increase the height of T_S, the inserted node x must lie below any existing level of T_S, as shown in figure 54.

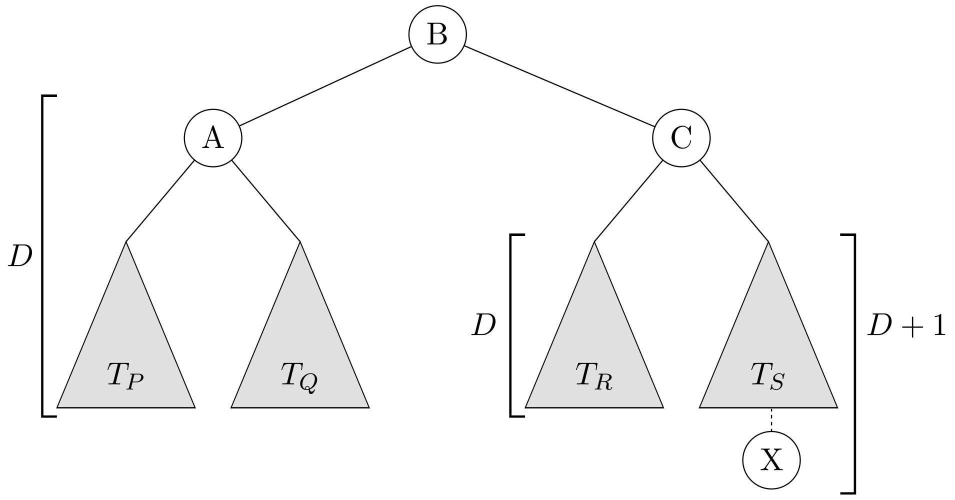


Figure 54: The tree from figure 53 after an insertion into T_S

After insertion of x, the height of the subtree rooted at C becomes $D+2$. Since the tree was balanced before the insertion, the height of T_R is at most D. To correct the imbalance at B, we must rearrange nodes to restore the height condition. Since the node x is responsible for the violation, and since it is the only node at depth $D+2$, the imbalance can be corrected by moving x to a lower depth without lowering any other node to depth $D+2$. This can be accomplished by a left rotation[34] about B, making C the new root and raising the subtree T_S up by one level. The resulting tree, shown in figure 55, is balanced.

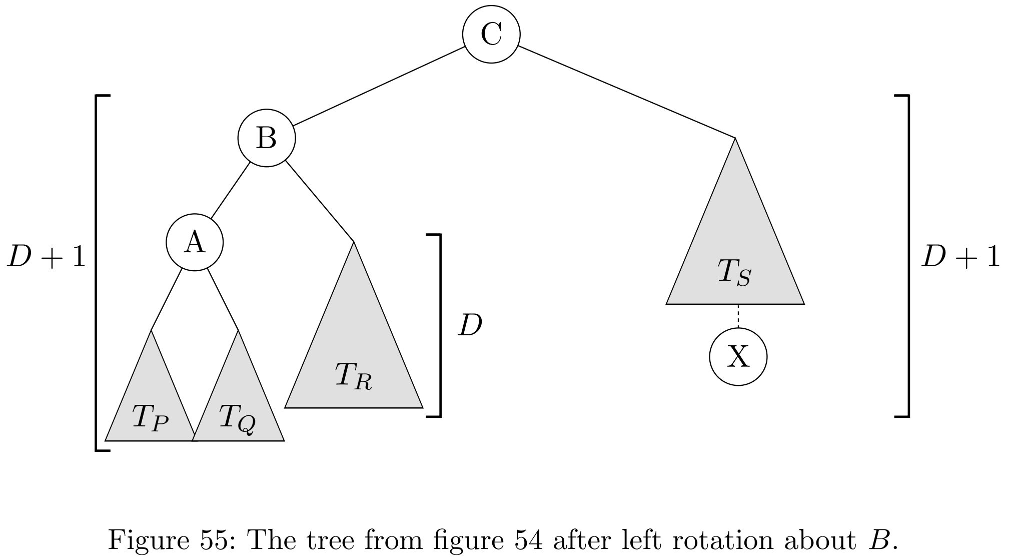


Figure 55: The tree from figure 54 after left rotation about B.

In general, if the subtree T_S is 'too-heavy', a single rotation is sufficient to rebalance the tree.

34. For an overview of rotation in binary trees, see section 12.2.

20.2 Double Rotation Rebalancing

Suppose that an insertion into the subtree T_R of the tree in figure 53, shown in figure 56, creates an imbalance at node B.

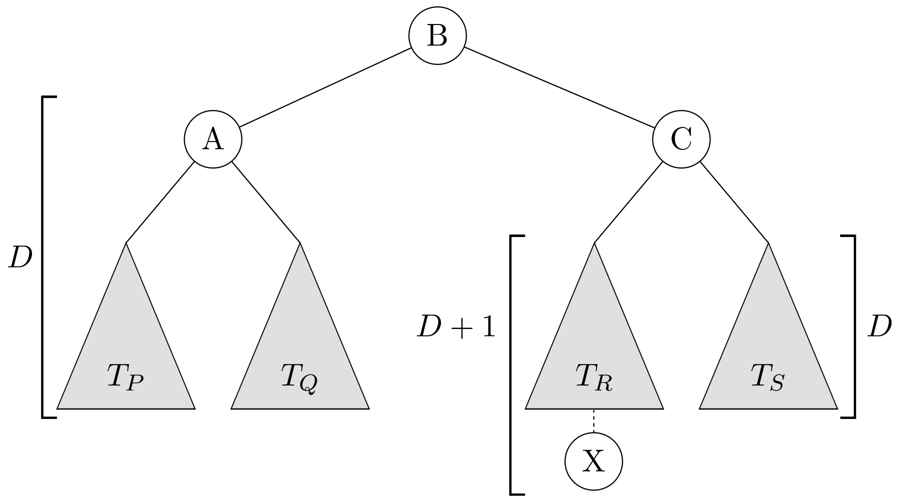


Figure 56: The tree from figure 53 after an insertion into T_R.

When an insertion into T_R creates an imbalance at B, the single rotation approach in section 20.1 is not sufficient to rebalance the tree, since a single rotation about B does not change the level of any nodes in T_R. Instead, we must adjust the subtree rooted at C first. Figure 57 shows an expanded view of figure 56. Note that the inserted node x may lie in either T_{R_1} or T_{R_2}.

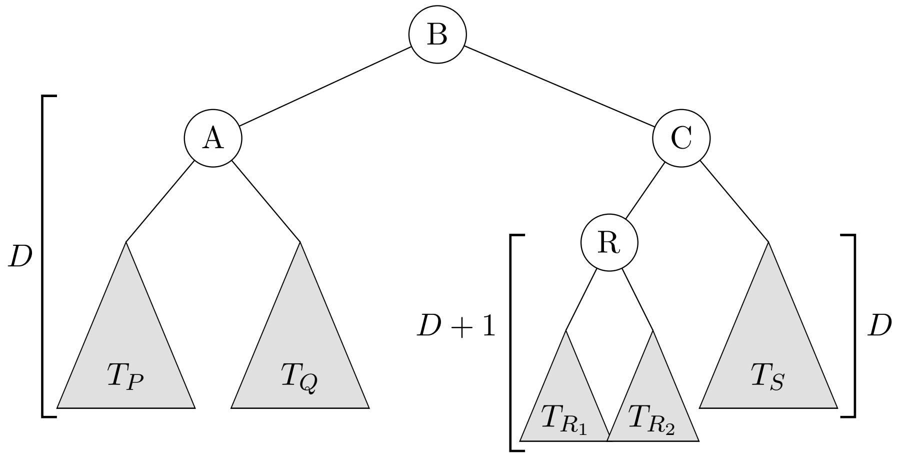


Figure 57: An expanded view of figure 56.

A right rotation about node C gives the tree in figure 58. Observe that the structure of the resulting tree is broadly similar to the tree in figure 54.

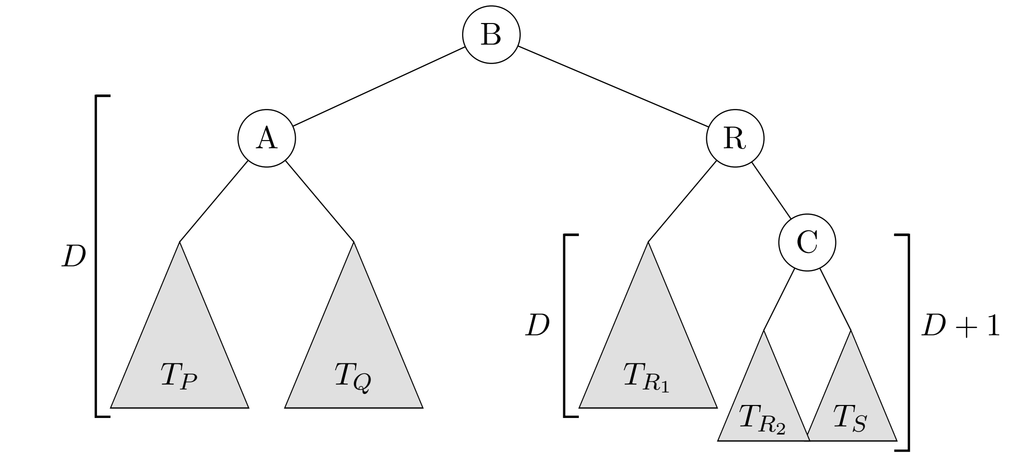


Figure 58: The tree of figure 57 after right rotation about C.

Since the shape of the tree now agrees with that of figure 54, the rebalancing can be completed by a left rotation about B.

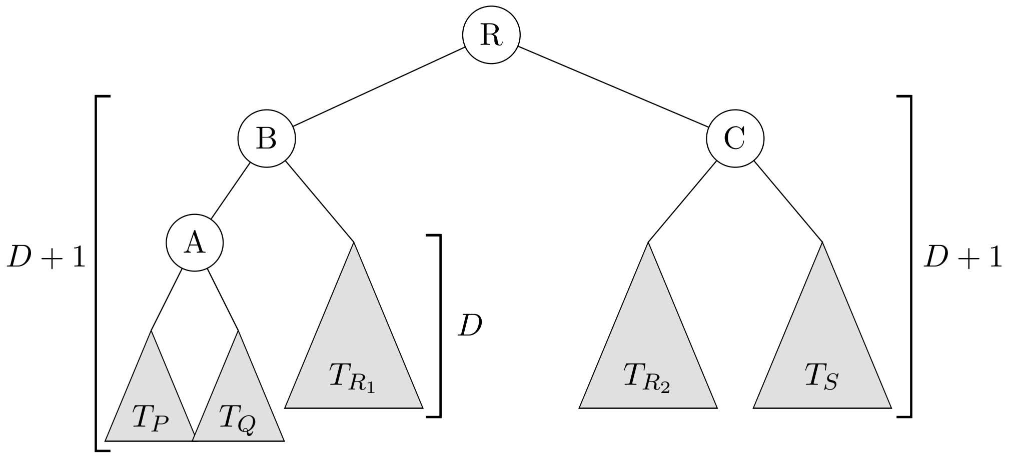


Figure 59: The tree of figure 58 after left rotation about B.

In general, the single rotation approach is sufficient to rebalance the tree when the subtree of an ‘outer’ grand-child of the root has the larger height. The double rotation approach, which is necessary when the subtree of an ‘inner’ grand-child causes an imbalance, first rotates about a child of the root to shift the extra height into an outer grand-child, then uses a single rotation to rebalance the tree.

20.3 Examples

Figure 60 shows an AVL tree on 10 vertices.

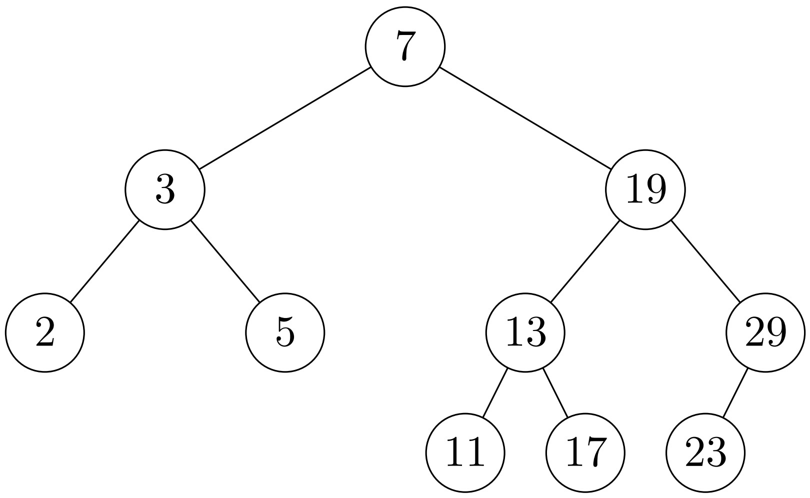


Figure 60: An AVL tree on 10 vertices.

Figures 61 and 62 show the result of inserting the values 1 or 30 into the tree of figure 60. Neither of these insertions create an imbalance, so no rebalancing is needed.

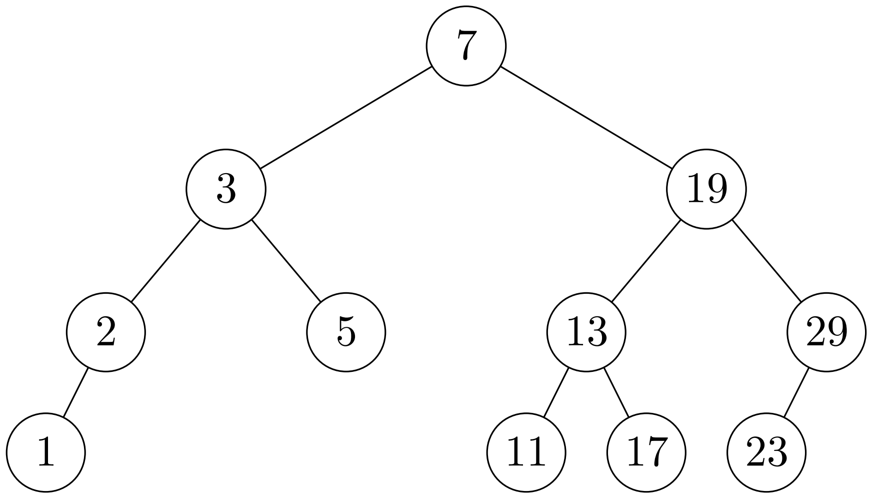


Figure 61: The tree of figure 60 after inserting 1.

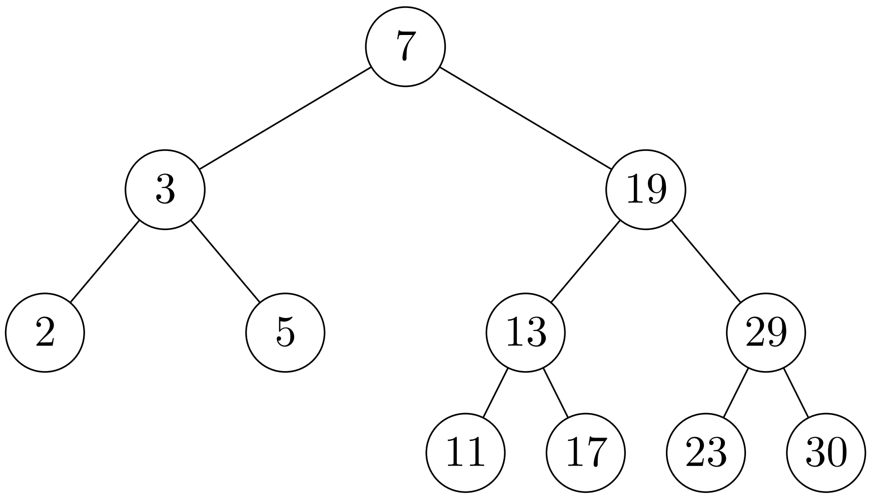
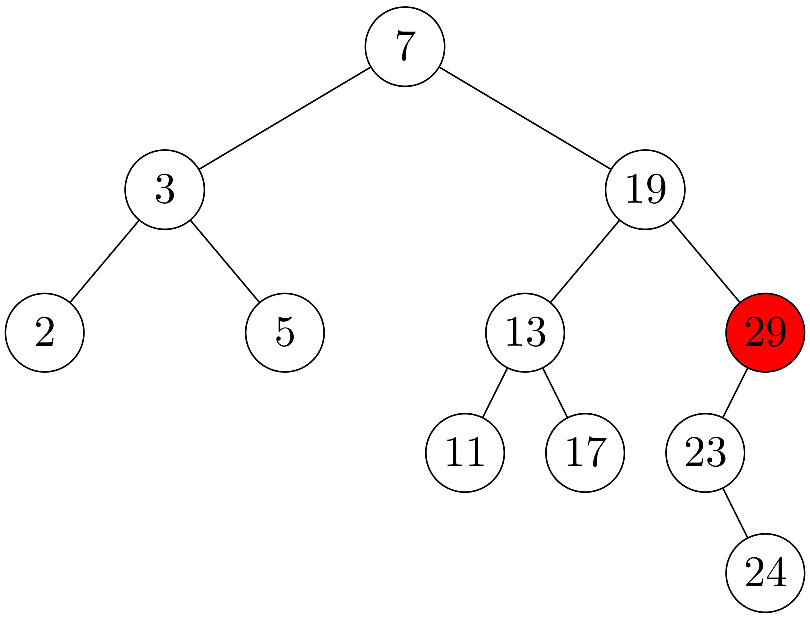
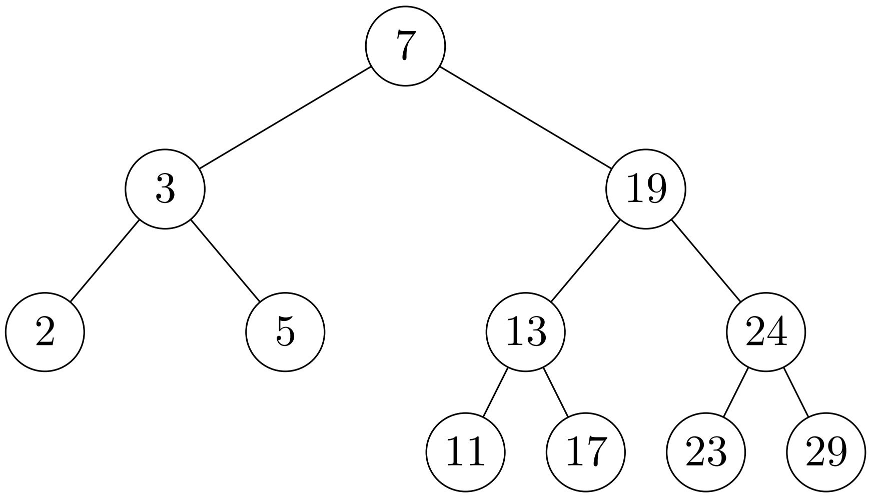


Figure 62: The tree of figure 60 after inserting 30.

Figure 63 shows the process of inserting the value 24 into the tree of figure 60. The insertion creates an imbalance at node 29, requiring a double rotation rebalancing.



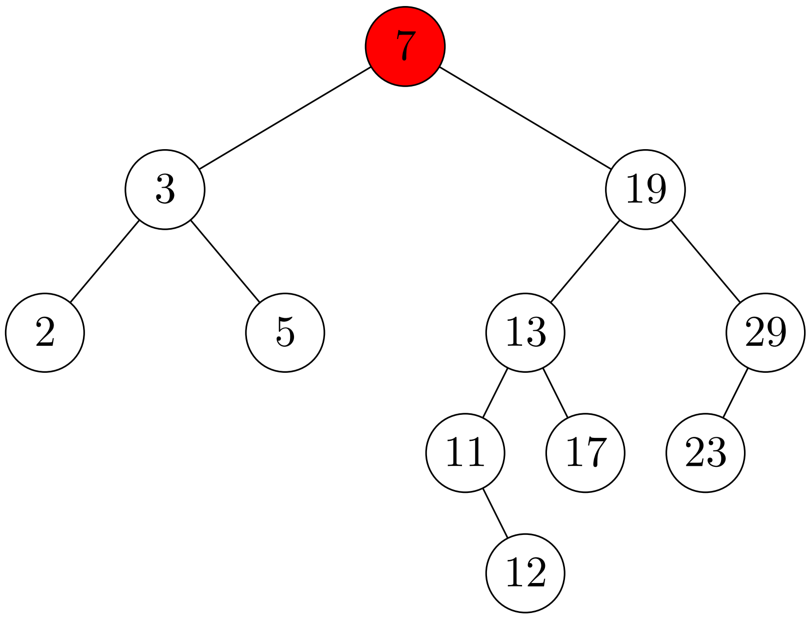
(a) After insertion, the subtrees of 29 are unbalanced



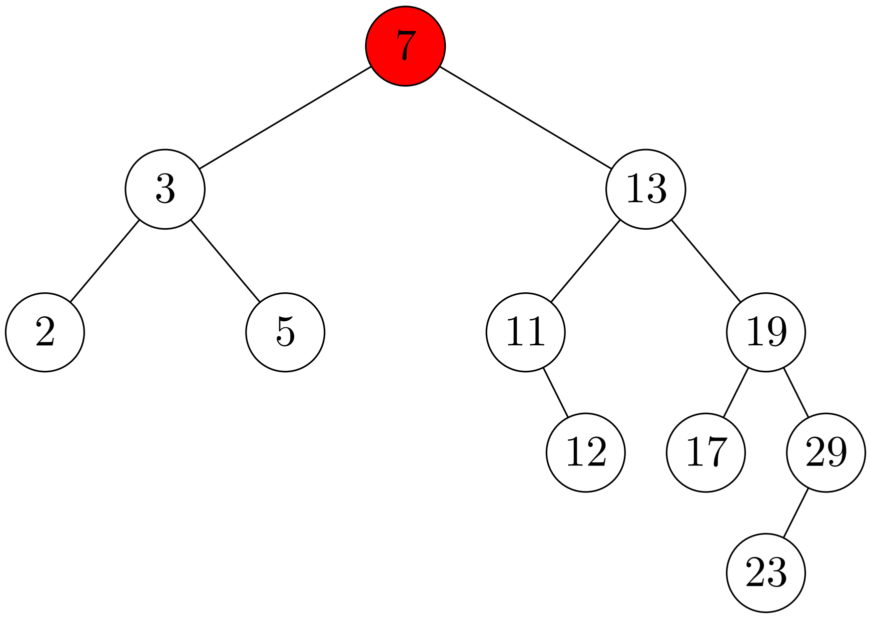
(b) After rebalancing

Figure 63: The tree of figure 60 after inserting 24 and rebalancing.

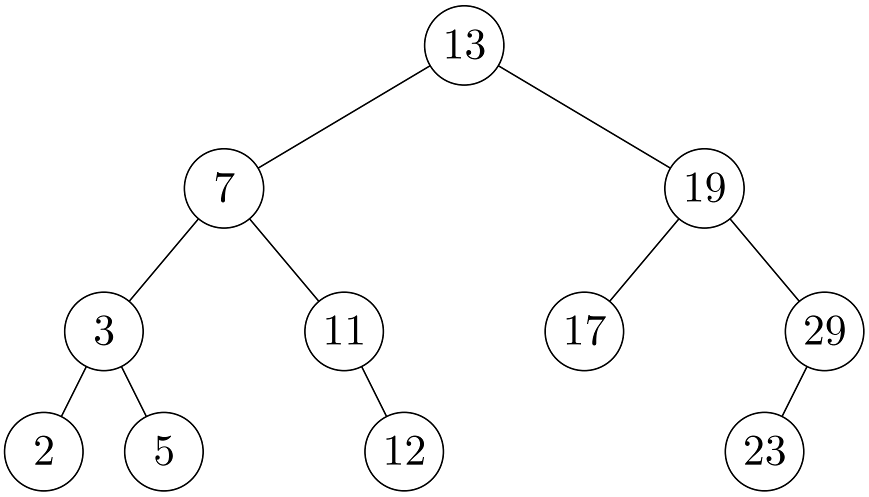
Figure 64 shows the process of inserting the value 12 into the tree of figure 60. The insertion creates an imbalance at the root, which is resolved by a double rotation. First, the subtree rooted at 19 is rotated right to shift nodes into the 'outer' grand-child of the root. Second, the tree is rotated left about the root to complete the rebalancing.



(a) After Insertion, the root is unbalanced



(b) After a right rotation about 19



(c) After a left rotation about 7

Figure 64: The tree of figure 60 after inserting 12 and rebalancing.

20.4 Cascading Rebalances on Removal

A single rebalancing operation, comprising either a single or double rotation, is sufficient to rebalance the tree after an insertion. Since rotations are a constant-time operation[35], the worst case running time of an insertion is $O(\log n)$, since the insertion itself takes $O(\log n)$ time, and the time required to trace a path from the newly created leaf to the root and correct any imbalances is $O(\log n)$. After a removal, however, there is no guarantee that a single rebalancing operation will suffice to restore the global balance of the tree.

The removal operation of an AVL tree begins with a regular binary tree removal, and, as with an insertion, traces up from the site of the removal to the root, rebalancing any unbalanced nodes encountered along the way. Since the path to the root has length at most $O(\log n)$, at most $O(\log n)$ rebalances are necessary. Since rebalancing is a constant-time operation, the total running time of the removal process is $O(\log n)$. Figure 65 shows an AVL tree with a 'tilted' internal layout. After removing the element 12 from this tree, an imbalance is created at 11, as shown in figure 66. After a rebalancing operation at node 11, the root becomes unbalanced (note that the balance property holds at the root before 11 is rebalanced), as shown in figure 67. A second rebalancing operation must be applied at the root to restore the global balance of the tree, as shown in figure 68. The structure of the tree in figure 65 can be generalized to produce a tree which may require $O(\log n)$ rebalancing operations on a particular removal.

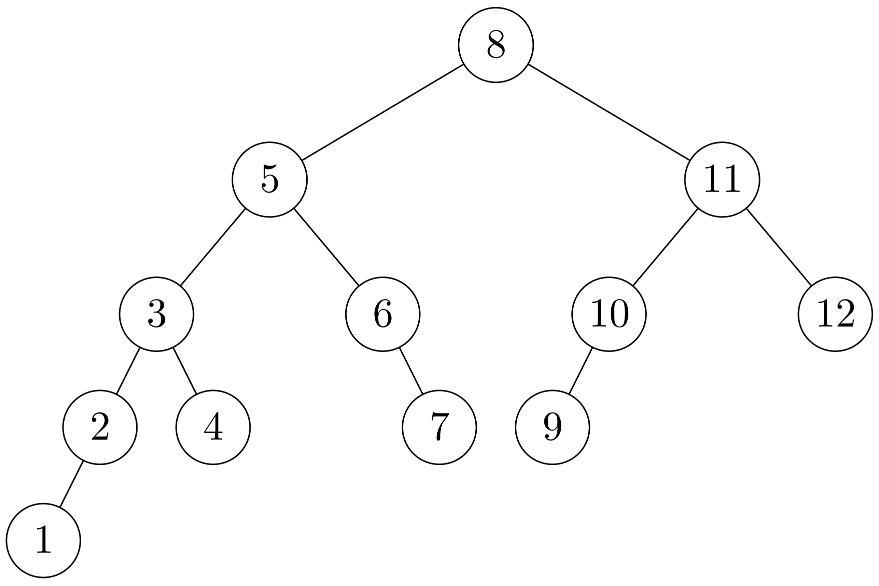


Figure 65: A 'tilted' AVL tree.

35. When a linked representation is used, a rotation can be performed by modifying a constant number of pointer values. If an array-based representation without indirection, such as the implicit indexing scheme in section 12.1, rotation is a $O(n)$ operation. For this reason, it is assumed that AVL trees are always implemented with a linked representation.

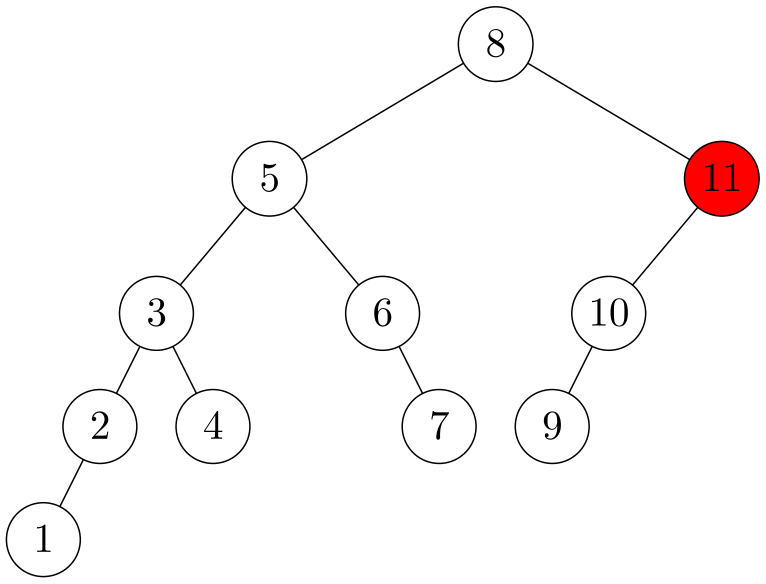


Figure 66: After removing 12, an imbalance occurs at 11.

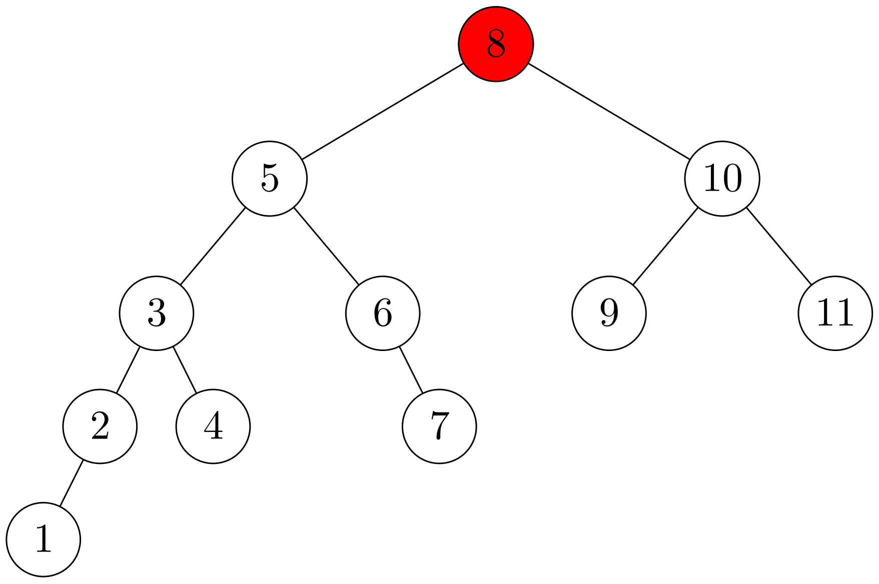


Figure 67: After rebalancing at 11, an imbalance occurs at the root.

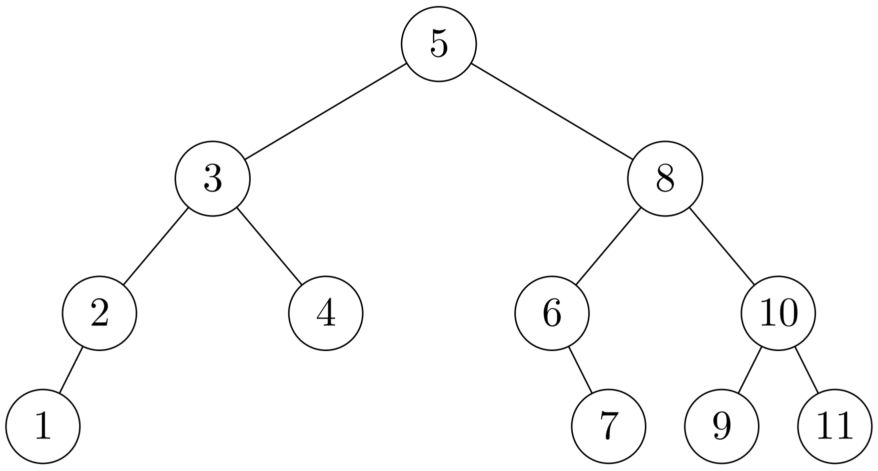


Figure 68: After rebalancing at the root, the tree is balanced.

Part III

Graphs

21 Definitions About Graphs

A graph G is an ordered pair (V, E) comprising a vertex set V and an edge set E, where elements of E are pairs of vertices in V. When the elements of E are ordered pairs (u, v), $u, v \in V$, G is a *directed graph* and the element $(u, v) \in E$ is referred to as an *arc* from u to v. If the elements of E are unordered pairs, then G is undirected. The sets V and E are often denoted by the functional notation $V(G)$ and $E(G)$.

A *loop* in a graph is an edge from a vertex to itself. A *simple* graph has no loops and a maximum of one edge between any pair of vertices. In this course, unless otherwise stated, all graphs will be assumed to be simple.

The *degree* of a vertex $v \in V$, denoted $d(v)$ or $\deg(v)$, is the number of edges incident with v.

A *subgraph* of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. The *complement* of a graph G, denoted $\overline{G}$, has an edge $uv \in E(\overline{G})$ if and only if $uv \notin E(G)$.

The *complete graph* on n vertices, denoted K_n, has an edge between every pair of vertices. The complement of a complete graph has no edges. Figure 69 shows the complete graph on 5 vertices and its complement.

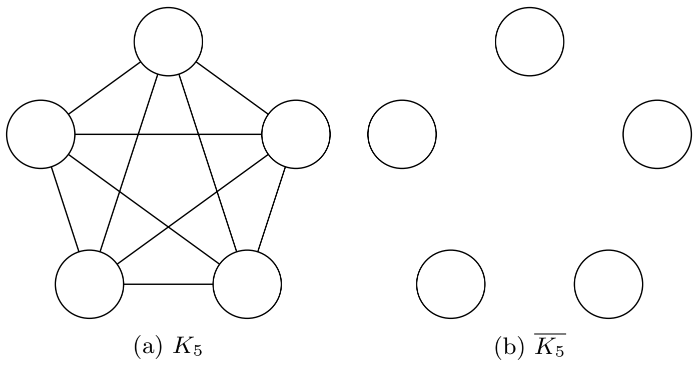

(a) K_5 (b) $\overline{K_5}$

Figure 69: The complete graph K_5 and its complement.

A *clique* of size n (or 'n-clique') in a graph G is a set of n vertices of G with an edge between every pair of vertices (i.e. a copy of K_n). An *independent set* (also called a *stable set*) of size n is a set of n vertices with no edges between any pair of vertices (i.e. a copy of $\overline{K_n}$). Figure 70 shows a graph with a 4-clique (blue) and a 4-independent set (red).

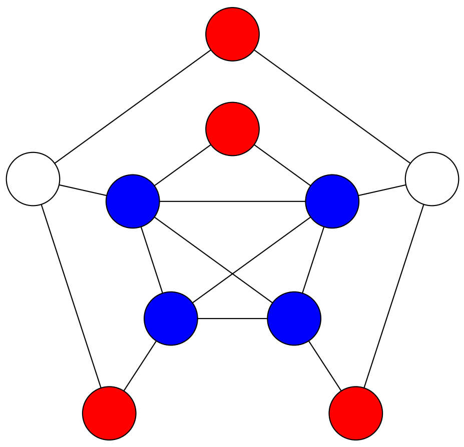


Figure 70: A graph containing a 4-clique (blue) and 4-independent set (red).

22 Adjacency List vs. Adjacency Matrix

For a graph G with n vertices and m edges, an adjacency list requires $\Theta(m + n)$ space and an adjacency matrix requires $\Theta(n^2)$ space. When G is simple, the number of edges is bounded above by the number of possible pairs of vertices. For an undirected graph, this bound gives

$$m \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

As a result, the worst case space complexity of an adjacency list representation is $\Theta(n^2)$, identical to an adjacency matrix. However, if space is the primary concern, there are cases in which both representations have significant advantages.

Two common implementations of adjacency lists are linked representations, where each vertex is an object with a list of pointers to its neighbours, and array-based representations, where vertices are indexed with integers and the list of neighbours is stored as a list of indices. For a linked representation, the minimum amount of space needed to store the data for each vertex is $w(1 + \deg(v))$, where w is the word size in bytes (which is assumed to be both the size of a pointer and of an integer value), since $\deg(v)$ words are needed to store a pointer to each neighbour, and at least 1 extra word is needed to manage the list of pointers[36].

An adjacency list for a graph with the maximum number $\binom{n}{2}$ of edges would then require $wn(1 + (n - 1)) = wn^2$ bytes of storage. Note that $\deg(v) = n - 1$ for every vertex in such a graph. On a machine with 32-bit words, a total of $4n^2$ bytes would be required[37]. If the array based representation is used, the space consumption can be reduced (at the cost of extra processing when values are retrieved from the list) by taking $w = \log_2 n$, since only $\log_2 n$ bits are necessary to store an index into an array of size n.

Since an adjacency matrix only stores 0 and 1 values, n^2 bits are needed in all cases[38]. Therefore, a graph with $\binom{n}{2}$ edges can be represented with $\frac{n^2}{8}$ bytes, and, as a result, the adjacency matrix representation is more space-efficient than an adjacency list.

To derive a general inequality to compare the space requirement of adjacency lists with adjacency matrices, we use the following identity (sometimes called the Handshake Lemma)

$$\sum_{v \in V(G)} \deg(v) = 2m$$

and obtain the total space consumption of the adjacency list for a graph, using the formula given above for the space consumption of each vertex:

$$\sum_{v \in V(G)} w(1 + \deg(v)) = wn + w \sum_{v \in V(G)} \deg(v) = wn + 2wm$$

An adjacency matrix requires less space when

$$n^2 < wn + 2wm$$

which is equivalent to

$$m > \frac{n^2 - wn}{2w}$$

For a graph on 100 vertices on a machine with a 32-bit word size ($w = 4$), the above gives

$$m > 1200$$

36. For example, if the pointers are stored in an array, one word would be needed to store the length of the array. Alternatively, a linked list could be used to store the pointers, but this would require $\deg(v)$ extra words of overhead to store pointers between elements of the list.

37. Note that on a 64-bit architecture, the C `int` type may be 32-bits, as on a 32-bit machine. However, all pointers are 64-bits on most current 64-bit architectures.

38. It is important to note that we can *store* an adjacency matrix in a more space-efficient format by leveraging certain properties of the graph, but doing so compromises the running time of some of the graph operations summarized at the end of this section. For example, in an undirected simple graph, the diagonal entries `A`$[i][i]$ are always 0, and since the matrix is symmetric, only the upper (or lower) triangle needs to be stored, requiring $\frac{1}{2}(n^2 - n)$ bits. Similarly, in a directed simple graph, although both the upper and lower triangle must be kept, the diagonal can be ignored, so only $n^2 - n$ bits are necessary. In this section, we assume that adjacency matrices are always stored as square 2-dimensional arrays, since we want to consider a representation which is usable by graph algorithms, rather than a space-efficient way to store the graph. In practice, large graphs are often stored in triangular adjacency matrices, which are often compressed with a general compression algorithm (such as `gzip` or `lzma`) for archival or distribution.

so an adjacency matrix is more space-efficient for graphs of more than 1200 edges.

Depending on the application, an adjacency list representation may use sorted lists of neighbours. When vertices are indexed by integers, maintaining sorted lists of neighbours allows retrieval with binary search. However, to do so, it is necessary to store the neighbour list in an array rather than a linked list (to allow direct indexing). The table below summarizes the worst-case running times of common graph operations in unsorted adjacency lists, sorted adjacency lists and adjacency matrices.

	Adjacency List (unsorted)	Adjacency List (sorted)	Adjacency Matrix
Find degree of v	$O(\deg(v))$	$O(\deg(v))$	$O(n)$
Find all neighbours of v	$O(\deg(v))$	$O(\deg(v))$	$O(n)$
Test whether edge vu exists	$O(\deg(v))$	$O(\log(\deg(v)))$	$O(1)$
Insert edge vu	$O(1)$	$O(\deg(u) + \deg(v))$	$O(1)$
Remove edge vu	$O(\deg(u) + \deg(v))$	$O(\deg(u) + \deg(v))$	$O(1)$

It is also possible to store the neighbour list for each vertex with a more advanced data structure (such as a binary search tree or hash table), which may reduce the running time of some operations.

23 Applications of Traversals

Simple modifications of the basic graph traversal algorithms can be used to find various important properties of graphs. This section contains pseudocode for several common modifications. Except for the minimum-length path algorithm (algorithm 23), all of the algorithms are presented as adaptations of DFS, but can also be implemented by modifying BFS. Consistent with commonly used graph notation, in algorithms dealing with a single graph G, the values n and m will be assumed to equal $|V(G)|$ and $|E(G)|$, respectively. For simplicity, it is assumed vertices of the graph are stored as integer values in the range $\{0, \dots, n\}$, allowing vertex values to be used as array indices.

Algorithm 21 finds a path between two given vertices u and v. Algorithm 22 finds a spanning tree of a connected input graph G rooted at a given vertex u and returns the spanning tree as a parent array P, where for a vertex v, the value of $P[v]$ is the parent of v in the spanning tree. Note that a path between u and another vertex v can easily be found by walking up from v to the root of the spanning tree.

Algorithm 21 Find a path from a vertex u to a vertex v.

```
procedure FindUVPathDFS($G, u, v, S,$ covered, current)
    covered[current] $\leftarrow$ true
    Push current onto $S$
    if current $= v$ then
        {$S$ now contains a $u - v$ path}
        Output the contents of $S$
        return
    end if
    for each neighbour $w$ of current do
        if covered[$w$] $=$ true then
            {$w$ has already been visited}
            Continue
        end if
        FindUVPathDFS($G, u, v, S,$ covered, $w$)
    end for
    Pop and discard the top value of $S$
end procedure

procedure FindUVPath($G, u, v$)
    covered $\leftarrow$ Array of size $n$, initialized to false
    $S \leftarrow$ Empty Stack
    {Start DFS on the initial vertex $u$}
    FindUVPathDFS($G, u, v, S,$ covered, $u$)
end procedure
```

Algorithm 22 Find a spanning tree of G rooted at vertex u. Note that if G is not connected, the resulting tree will only span the connected component containing u.

```
procedure FindSpanningTreeDFS($G$, covered, $P$, current)
    covered[current] ← true
    for each neighbour $v$ of current do
        if covered[$v$] = true then
            {$v$ has already been visited}
            Continue
        end if
        {If $v$ has not been visited yet, add the edge from current}
        {to $v$ to the spanning tree}
        $P[v]$ ← current
        FindSpanningTreeDFS($G$, covered, $P$, $v$)
    end for
end procedure

procedure FindSpanningTree($G$, $u$)
    covered ← Array of size $n$, initialized to false
    {Create an array to store the parent of each node in the tree}
    $P$ ← Array of size $n$
    {Set the parent of the root $u$ to an invalid value}
    $P[u] \leftarrow -1$
    {Start DFS on the initial vertex $u$}
    FindSpanningTreeDFS($G$, covered, $P$, $u$)
    return $P$
end procedure
```

Algorithm 21 is guaranteed to find a $u - v$ path if one exists. However, the path it finds may contain more than the minimum number of edges. Reimplementing the same algorithm with BFS guarantees that the resulting path will be of minimum length. Algorithm 23 gives pseudocode to find a minimum length $u - v$ path with BFS. Unlike algorithm 21, algorithm 23 cannot build a path recursively and store the partially-formed result in a stack, since BFS does not traverse paths sequentially. Instead, a spanning tree is created and the path from v to the root is returned. As a result, algorithm 23 also provides an alternative to algorithm 22.

Algorithm 23 Find a minimum length $u - v$ path.

```
procedure MinLengthPath($G, u.v$)
    covered ← Array of size $n$, initialized to false
    {Create an array to store the parent of each node in the tree}
    $P$ ← Array of size $n$
    {Set the parent of the root $u$ to an invalid value}
    $P[u] \leftarrow -1$
    {Create a queue for BFS}
    $Q$ ← Empty queue
    Enqueue $u$ in $Q$
    while $Q$ is non-empty do
        current ← Dequeue($Q$)
        covered[current] ← true
        for each neighbour $w$ of current do
            if covered[$w$] = true then
                {$w$ has already been visited}
                Continue
            end if
            {If $w$ has not been visited yet, add the edge from current}
            {to $w$ to the spanning tree}
            $P[w]$ ← current
            Enqueue $w$ in $Q$
        end for
    end while
    {Create a sequence to store the path}
    $S$ ← Empty sequence
    {Walk to the root of the spanning tree, adding each vertex to $S$}
    current ← $v$
    Add $v$ to $S$
    while $P$[current] $\neq -1$ do
        current ← $P$[current]
        Insert current at the beginning of $S$
    end while
    return S
end procedure
```

Algorithm 21 can be modified to find a cycle containing a given vertex u. A related problem is determining whether a graph G contains a cycle at all. Checking for cycles is often a pre-processing step of algorithms which require acyclic graphs (for example, topological sorting). One simple method to check for cycles is to run DFS and check for back edges, since the existence of a back edge in a DFS tree implies the existence of a cycle. Algorithm 24 gives pseudocode for such an approach. With relatively minor modifications to store the current branch of the DFS tree, it is also possible to use algorithm 24 to explicitly find the vertices in a cycle.

Algorithm 24 Test whether G contains a cycle.

```
procedure TestForCycleDFS($G$, covered, current)
    covered[current] $\leftarrow$ true
    for each neighbour $v$ of current do
        if covered[$v$] = true then
            {$v$ has already been visited}
            {Therefore, (current, $v$) is a back edge, so a cycle exists}
            return true
        end if
        if TestForCycleDFS($G$, covered, $v$) = true then
            return true
        end if
    end for
    return false
end procedure

procedure TestForCycle($G$)
    covered $\leftarrow$ Array of size $n$, initialized to false
    $u \leftarrow$ An arbitrary vertex of $G$
    return TestForCycleDFS($G$, covered, $u$)
end procedure
```

24 Connectivity

An undirected graph G is *connected* if for all pairs of vertices $u, v \in V(G)$ there exists a path between u and v. The *connected components* of G are the set of maximal connected subgraphs of G. G is connected if and only if it has exactly one connected component.

A graph has a spanning tree if and only if it is connected, so DFS and BFS can be used to test connectivity in undirected graphs. The set of connected components of a graph can be computed with a sequence of traversals, with each traversal identifying a single component. Algorithm 25 gives pseudocode for finding the connected components of a graph, using DFS as a basis. If the graph contains k connected components, algorithm 25 associates each vertex v with the index `component`[v] $\in \{0, \dots, k-1\}$ of its component.

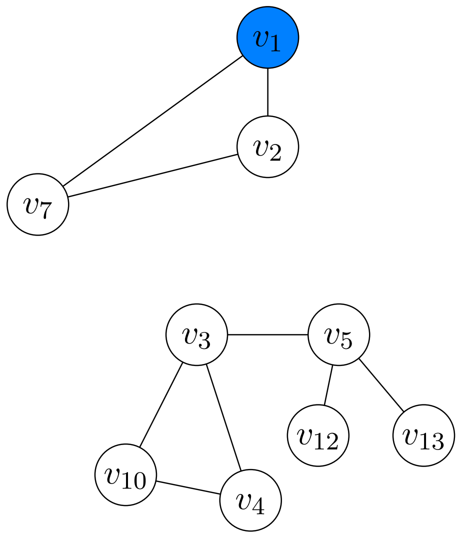
Algorithm 25 Find the connected components of G.

```
procedure ComponentDFS($G$, component, current, componentIndex)
    component[current] $\leftarrow$ componentIndex
    for each neighbour $v$ of current do
        if component[$v$] $\geq 0$ then
            {$v$ has already been visited during traversal of the current component}
        else
            ComponentDFS($G$, component, $v$, componentIndex)
        end if
    end for
end procedure

procedure FindConnectedComponents($G$)
    component $\leftarrow$ Array of size $n$, initialized to $-1$
    nextComponentIndex $\leftarrow 0$
    for each vertex $u \in V(G)$ do
        {Check if $u$ has already been assigned to a component, and if so, ignore it.}
        if component[$u$] $\geq 0$ then
            Continue
        end if
        {Otherwise, create a new component and assign all vertices reachable from $u$ to it.}
        ComponentDFS($G$, component, $u$, nextComponentIndex)
        nextComponentIndex $\leftarrow$ nextComponentIndex $+ 1$
    end for
    return component
end procedure
```

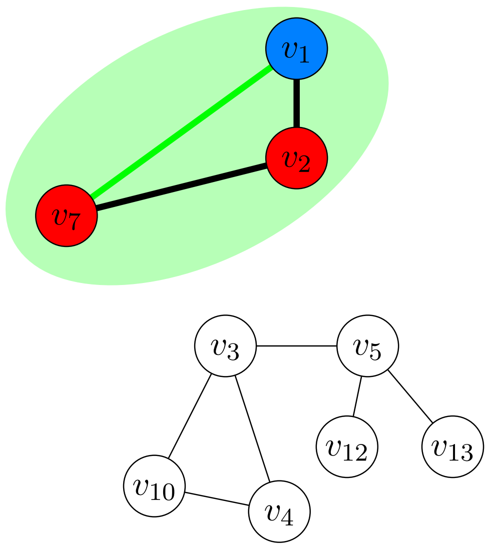
The ComponentDFS procedure in algorithm 25 traverses the connected component containing a given vertex and marks all vertices found as being in the component. Since such a traversal implicitly constructs a spanning tree of the component, the algorithm can easily be modified to find a spanning tree of each connected component. The for-loop in the FindConnectedComponents procedure ignores all vertices that have already been assigned to a component, so the total number of calls to ComponentDFS is equal to the number of components. Each vertex and edge of G is visited by only one of the calls to ComponentDFS, so the entire algorithm runs in $O(n + m)$ time, which is asymptotically optimal, since every vertex and edge must be examined at least once to make any assertions about connectivity.

Figures 71 through 75 show the progression of algorithm 25 on a sample input. In the diagrams, edges in the spanning tree of each component are darkened and back edges are shaded bright green. The node provided to the initial call to ComponentDFS for each component is shaded in blue, and the components (after they are found) are shaded light green.



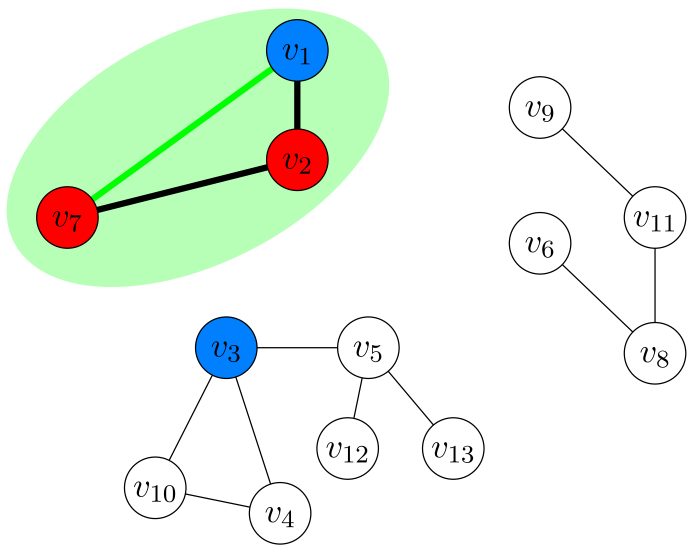
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
`component`	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

Figure 71: Computing Connected Components - Step 1. The node v_1, which is not yet associated with a component, is passed to ComponentDFS.



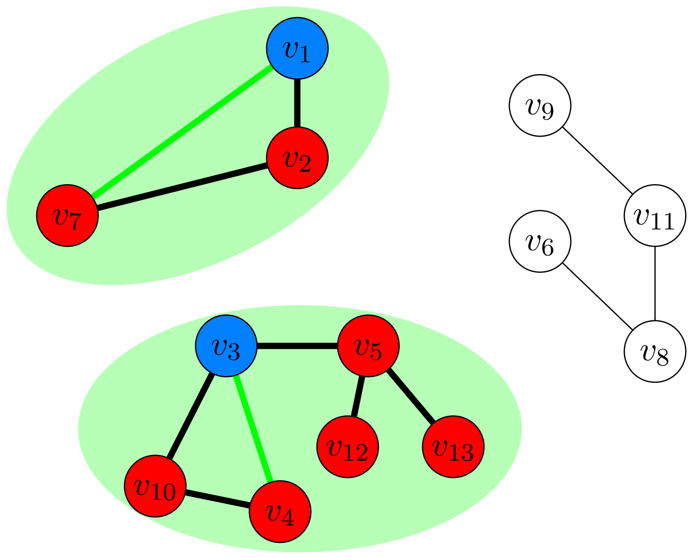
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
`component`	**0**	**0**	−1	−1	−1	−1	**0**	−1	−1	−1	−1	−1	−1

Figure 72: Computing Connected Components - Step 2. After ComponentDFS returns, all vertices reachable from v_1 have been added to component 0.



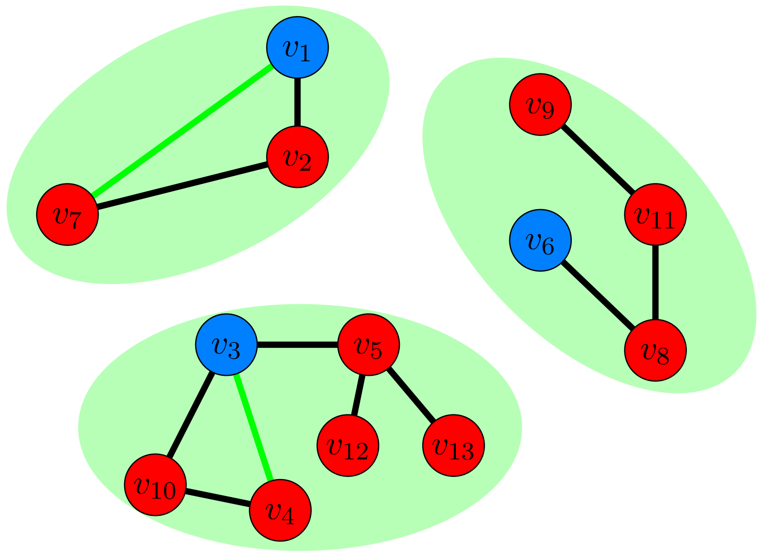
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
`component`	0	0	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1

Figure 73: Computing Connected Components - Step 3. The outer loop of FINDCONNECTED-COMPONENTS skips vertex v_2 since it already belongs to component 0. Vertex v_3 is passed to COMPONENTDFS.



Index	1	2	3	4	5	6	7	8	9	10	11	12	13
`component`	0	0	**1**	**1**	**1**	-1	0	-1	-1	**1**	-1	**1**	**1**

Figure 74: Computing Connected Components - Step 4. When COMPONENTDFS returns, the component containing v_3 has been fully identified. The next unassociated vertex is v_6.

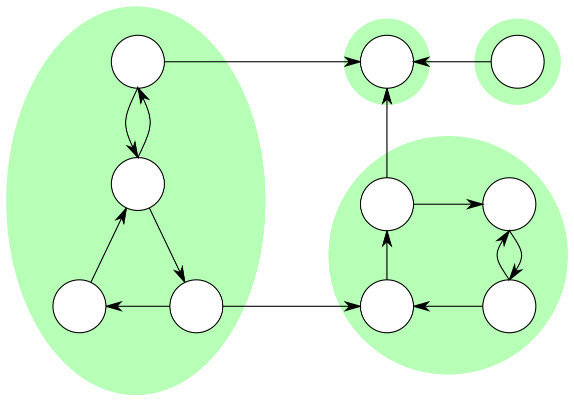


Index	1	2	3	4	5	6	7	8	9	10	11	12	13
`component`	0	0	1	1	1	**2**	0	**2**	**2**	1	**2**	1	1

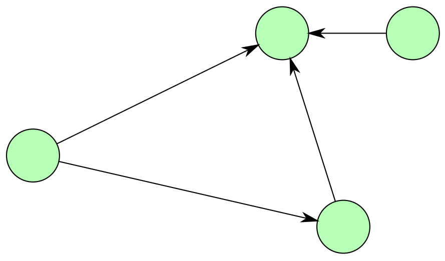
Figure 75: Computing Connected Components - Step 5. The component containing v_6 has been computed. All vertices are now associated with a component.

When G is a directed graph, we say that it is *weakly connected* if the underlying undirected graph is connected, and *strongly connected* if for all pairs of vertices $u, v \in V(G)$, there is a directed path from u to v. The *strongly connected components* of G are the set of maximal strongly connected subgraphs of G. A directed graph is *strongly connected* if it has only one strongly connected component.

If each strongly connected component of G is merged into a single vertex, the resulting graph is a directed acyclic graph (DAG). Figure 76 illustrates this transformation for a sample graph.



(a) Original graph G with strongly connected components shaded light green.



(b) Collapsed graph G'.

Figure 76: When the strongly connected components (shaded in green) of a graph G (left) are merged into single vertices (right), the result is a directed acyclic graph.

Unlike the undirected case, whether or not G is strongly connected cannot be determined as a side effect of a single directed DFS or BFS traversal, since a traversal rooted at a vertex v only finds vertices reachable by directed paths from v. A naive algorithm for testing strong connectivity is to simply run a traversal rooted at every vertex in G, and report that the graph is strongly connected if every traversal visits every vertex. This approach requires $O(n(n+m)) = O(n^3)$ time in the worst case.

The spanning tree generated by directed DFS (or BFS) rooted at a starting vertex $v \in V(G)$ gives a set of 'outbound' paths from v to other vertices $u \in V(G)$. A set of 'inbound' paths from other vertices $w \in V(G)$ to v can be obtained by running DFS (or BFS) on the graph G^{T} obtained by reversing the direction of all arcs in G. This graph is called the *transpose graph* of G because the adjacency matrix of G^{T} is the transpose of the adjacency matrix of G. If there exists a path from w to v and a path from v to u, then a wu-path runs through v. If G is strongly connected, there must exist a wv-path and a vu-path for all $w, v \in V(G)$, so strong connectivity can be tested by simply verifying that all vertices in G and G^{T} are visited by traversals rooted at an arbitrary starting point v. Algorithm 26 gives pseudocode for this test. Instead of explicitly computing G^{T}, the traversal function ConnectivityDFSTransposed implicitly traverses G^{T} by walking backward along arcs of G. The ConnectivityDFS and ConnectivityDFSTransposed functions return the total number of vertices visited, which must equal n on both traversals for the graph to be strongly connected.

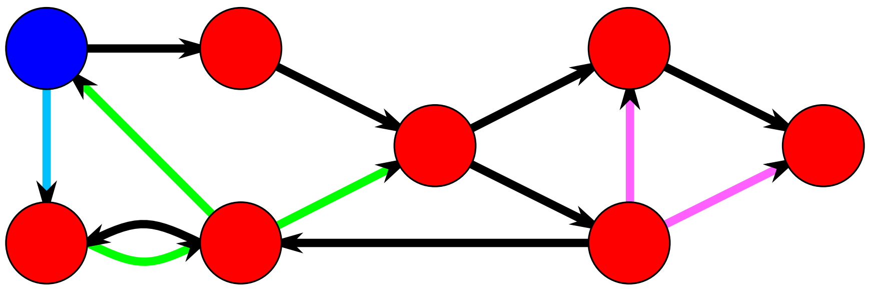
Algorithm 26 Test whether a directed graph G is strongly connected.

```
procedure ConnectivityDFS($G$, covered, current)
    covered[current] $\leftarrow$ true
    totalCovered $\leftarrow 1$
    for each outbound neighbour $v$ of current do
        if covered[$v$] $=$ true then
            {$v$ has already been visited}
            Continue
        end if
        totalCovered $\leftarrow$ totalCovered$+$ ConnectivityDFS($G$, covered, $v$)
    end for
    return totalCovered
end procedure

procedure ConnectivityDFSTransposed($G$, covered, current)
    covered[current] $\leftarrow$ true
    totalCovered $\leftarrow 1$
    for each inbound neighbour $v$ of current do
        if covered[$v$] $=$ true then
            {$v$ has already been visited}
            Continue
        end if
        totalCovered $\leftarrow$ totalCovered$+$ ConnectivityDFSTransposed($G$, covered, $v$)
    end for
    return totalCovered
end procedure

procedure TestStrongConnectivity($G$)
    covered $\leftarrow$ Array of size $n$, initialized to false
    $n \leftarrow |V(G)|$
    $u \leftarrow$ An arbitrary vertex of $G$
    {Traverse arcs from $u$ in $G$}
    if ConnectivityDFS($G$, covered, $u$) $< n$ then
        return false
    end if
    Reset every value of covered to false
    {Traverse arcs from $u$ in $G^{\mathrm{T}}$}
    if ConnectivityDFSTransposed($G$, covered, $u$) $< n$ then
        return false
    end if
    return true
end procedure
```

Algorithm 26 performs two DFS traversals of the input graph G, both of which require $O(n+m)$ time in the worst case. Therefore, algorithm 26 is $O(n+m) = O(n^2)$, which is asymptotically optimal for testing connectivity. Figure 77 shows the DFS trees traversed by algorithm 26 on a sample graph. Back edges are shown in green, cross edges are shown in pink and forward edges are shown in light blue. The root of the DFS tree is shaded blue, and other vertices covered by the traversals are shaded red. Since there are vertices which are not covered by one of the traversals, the graph in figure 77 is not strongly connected.


(a) DFS Tree for G

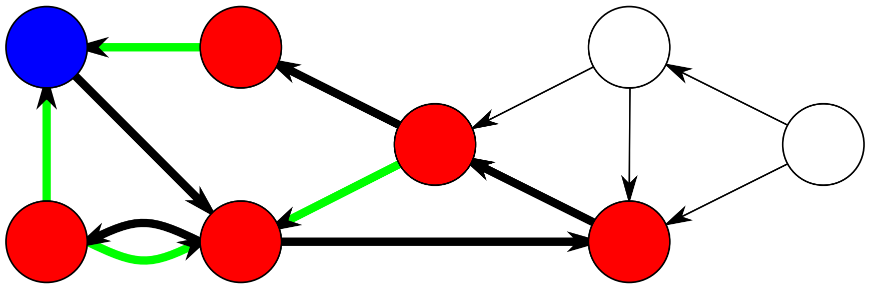

(b) DFS Tree for G^{T}

Figure 77: The DFS trees traversed by algorithm 26 on a sample input graph for a particular starting vertex (blue).

Similar to how algorithm 25 finds undirected connected components by iterating the connectivity test, an algorithm to find the strongly connected components of a directed graph can be created by iterating the strong connectivity test of algorithm 26, identifying a single strongly connected component on each pass. However, unlike the undirected case, the resulting algorithm is not asymptotically optimal, since there is no guarantee that each edge will be visited by only one pass, so the algorithm can take $O(n^3)$ time in the worst case. As in the undirected case, the asymptotic lower bound for computing the strongly connected components is $\Omega(n + m)$, since every vertex and edge must be visited at least once. There are several specialized algorithms which can compute the set of strongly connected components in $O(n+m)$ using directed traversals with auxiliary data structures.

25 Playing Games with Graphs

A game or puzzle with a set of discrete states can be modeled with a graph, and graph algorithms can be used as the basis for artificial intelligence methods. Each possible state of the game can be represented by a vertex, with edges added between pairs of states which are reachable from each other. The paths through the resulting graph encapsulate every possible course of gameplay. Graph traversals can then be used to 'solve' the game by systematically evaluating all possible courses the game can follow from a given position, allowing a computer to play the game 'perfectly' by always choosing moves that guarantee the largest chance of winning. In some cases, a perfect player is guaranteed to win, regardless of their opponent's strategy. In

Checkers and Tic-Tac-Toe, for example, the player who moves first can guarantee themselves a win or draw with the correct selection of moves[39].

25.1 Tic-Tac-Toe

The game of Tic-Tac-Toe is played on a 3×3 grid by two players, $\times$ and $\bigcirc$. With $\times$ moving first, the two players alternately take squares in the grid until one player has a line of 3 (in which case that player wins) or no empty squares remain (in which case the game is a draw). Since there are at most $3^9 = 19683$ possible boards[40], the entire set of states can be stored in memory as a graph, and the graph algorithms studied in this course can be applied to solve it.

The vertices of the game graph are the possible boards, and each board is connected by a directed edge to all boards which can be obtained after one move. There is no need to separately keep track of whose move it is at each step, since that can be derived from the numbers of each symbol on the board (if there are fewer $\bigcirc$s than $\times$s, it is $\bigcirc$'s move. Otherwise, it is $\times$'s move). Since it is not possible to remove symbols from the board, the resulting graph is a directed acyclic graph (see section 26 for definitions and information about directed acyclic graphs). The minimum node of the graph is the initial blank board. Figure 78 shows the initial board and some of its successors.

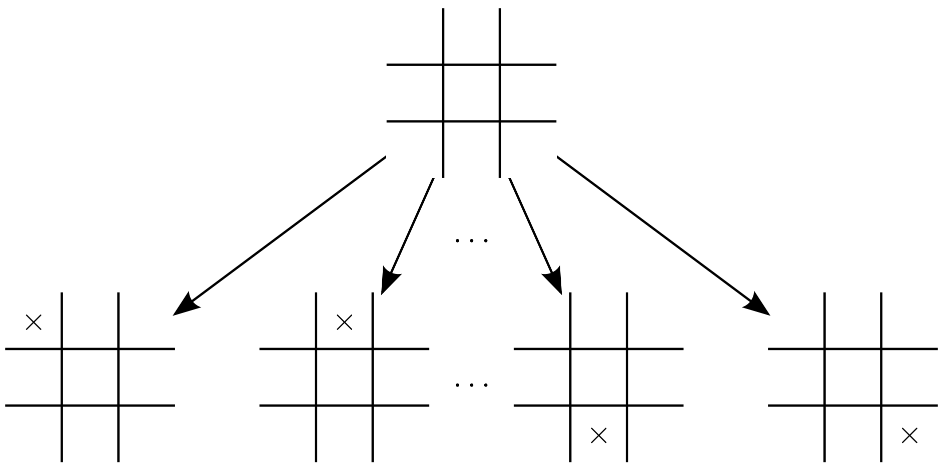


Figure 78: The first two levels of the game graph for Tic-Tac-Toe.

Consider an algorithm which plays the $\times$ player in Tic-Tac-Toe. Given a board representing the current state of the game, the game graph can be traversed from the corresponding vertex to find all possible outcomes of the game, and the next move can be chosen based on which outcome is more likely to result in $\times$ winning (or a draw if winning is impossible). Since the boards corresponding to endgames (win, lose and draw) can be identified without knowing the sequence of moves which produced them (since we can check for lines of one symbol), the vertices for endgame boards can be labeled before the game is played. If all of the endgame boards reachable from a given vertex are wins, then $\times$ is guaranteed to win. Similarly, if all endgame boards are losses, then $\times$ is guaranteed to lose. Figure 79 shows a subset of the game

39. Checkers was solved in 2007 with techniques far beyond the scope of CSC 225 (due to the game having more than 10^{20} states). See Schaeffer et al., *Checkers is Solved*, Science 317.5844 (2007) 1518-1522.

40. Not all boards can be encountered in actual play. For example, boards with more $\bigcirc$s than $\times$s and boards with two or more disjoint lines (which would only be encountered if the game continued after one player had already won) will never appear.

graph for Tic-Tac-Toe. Boards which are wins for $\times$ are framed in green and boards which are wins for $\bigcirc$ are framed in red.

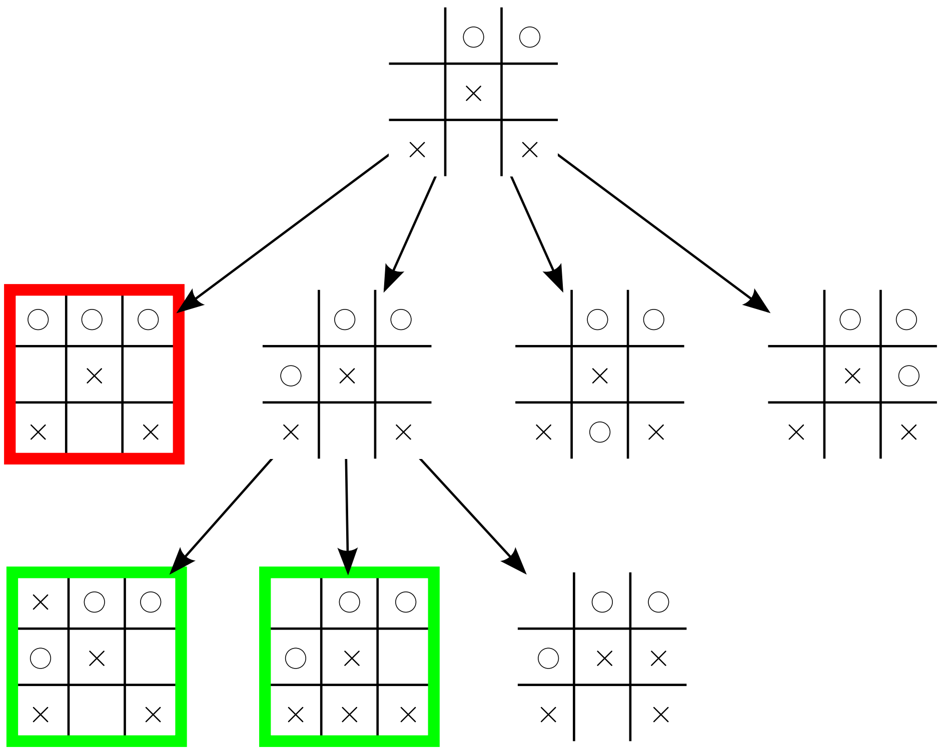


Figure 79: A sample board and some of its successors. Winning boards for $\times$ are framed in green. Winning boards for $\bigcirc$ are framed in red.

25.2 The 16-puzzle

The 16-puzzle consists of 15 tiles containing the numbers $1, 2, \ldots, 15$ in a 4×4 grid, with an empty space left by the missing 16th tile. The goal of the 16-puzzle is to rearrange the tiles into order by sliding tiles to occupy an empty space. Figure 80 shows a sample board along with the goal configuration where the tiles are in order.

2	15	4	8
13	3	14	11
6		1	12
7	9	5	10

(a) A scrambled board

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) A solved board

Figure 80: Examples of the 16-puzzle.

A similar puzzle can be devised for any $n \times n$ board. On a board with N positions (including the empty space), the total number of possible configurations is $N!$, since every arrangement of tiles can be encoded by a permutation of $\{1, \ldots, N\}$ (where the empty space is treated as an invisible tile marked with N), although some configurations cannot be solved. The game graph for the N-puzzle contains vertices for each possible board, and an undirected edge connects every pair of boards which can be transformed into each other by one move. Since every move is reversible (that is, we can always move a tile back after the initial move), there is no need for directed edges as in section 25.1. The game graph for the 4-puzzle contains only $4! = 24$ states, and is shown in figure 81. The goal state is framed in green.

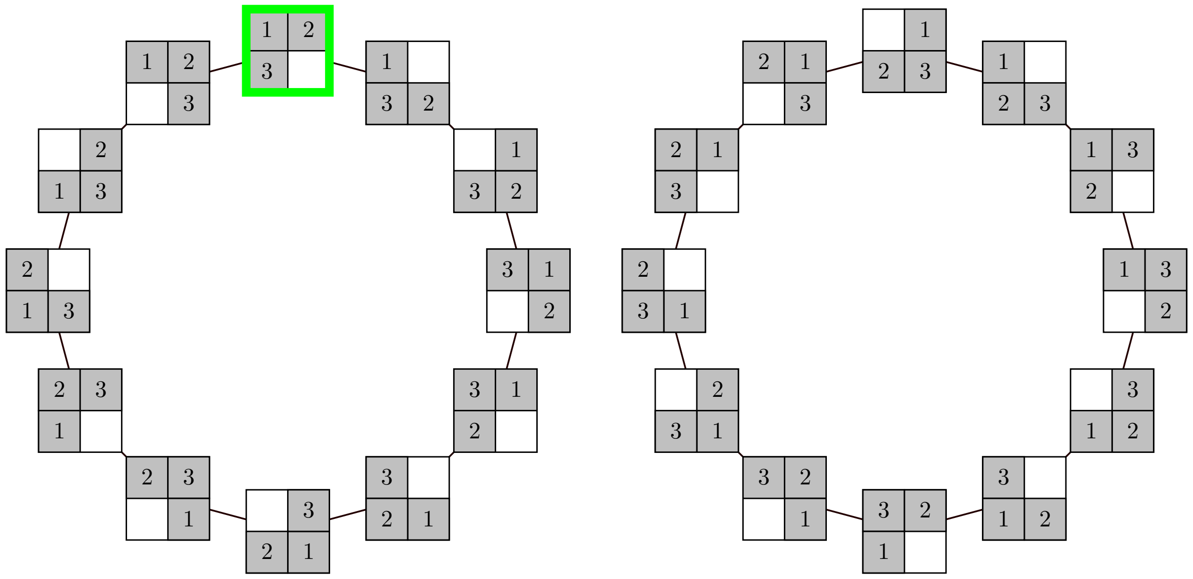


Figure 81: The entire game graph for the 4-puzzle, with the goal state framed in green.

The 9-puzzle has $9! = 362880$ states, so it is possible to compute and store the entire game graph on a current machine. Graph algorithms can then be used to find solutions to each board. For example, a path from a given board b to the goal configuration g (in which all tiles are in order and the empty space is at the lower right) represents a sequence of valid moves which solve b. If g is not reachable from b, then b has no solution. In general, the game graph of a puzzle may have several different connected components, and there may not be a goal state in each component. The game graph for an N-puzzle always has two components, and there is only one goal state. Algorithms for finding connected components (see section 24) can be used to find all solvable configurations of a puzzle. For the N-puzzle, it is also possible to determine whether a given board is solvable without traversing the game graph by using techniques from permutation theory (which is beyond the scope of this course).

Figure 82 shows the neighbourhood of the goal state of the 9-puzzle. Algorithm 27 gives pseudocode to build the game graph of an N puzzle.

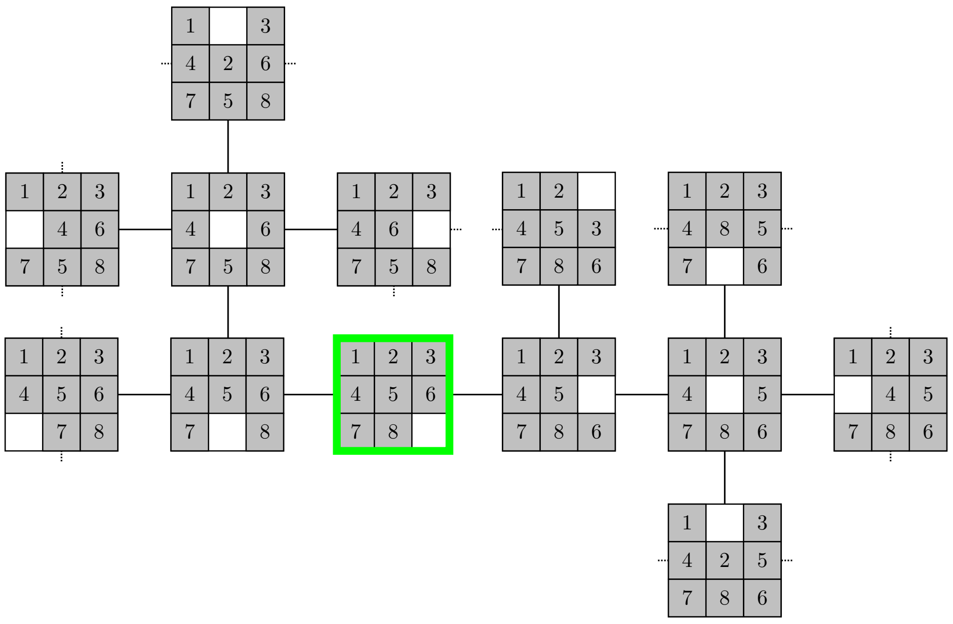


Figure 82: A subset of the game graph for the 9-puzzle, with the goal state framed in green.

Algorithm 27 Build the game graph of an N-puzzle

```
procedure BuildNPuzzleGraph($N$)
    $G \leftarrow$ New graph on $N!$ vertices, each corresponding to a board.
    $n \leftarrow \sqrt{N}$ (number of rows/columns of the board)
    for each vertex $v$ of $G$ do
        $B \leftarrow$ The board corresponding to $v$
        $i, j \leftarrow$ Coordinates of the empty space in $B$
        {Consider all of the neighbouring squares of the empty space}
        {and find all of the boards which can be created by moving them.}
        if $i > 0$ then
            $B' \leftarrow$ The board created by moving tile $[i-1, j]$ down one row.
            $u \leftarrow$ The vertex corresponding to board $B'$
            Add the edge $uv$ to $G$
        end if
        if $i < n-1$ then
            $B' \leftarrow$ The board created by moving tile $[i+1, j]$ up one row.
            $u \leftarrow$ The vertex corresponding to board $B'$
            Add the edge $uv$ to $G$
        end if
        if $j > 0$ then
            $B' \leftarrow$ The board created by moving tile $[i, j-1]$ to the right.
            $u \leftarrow$ The vertex corresponding to board $B'$
            Add the edge $uv$ to $G$
        end if
        if $j < n-1$ then
            $B' \leftarrow$ The board created by moving tile $[i, j+1]$ to the left.
            $u \leftarrow$ The vertex corresponding to board $B'$
            Add the edge $uv$ to $G$
        end if
    end for
end procedure
```

26 Topological Sorting

In a directed graph G, a vertex u is said to be *reachable* from a vertex v if there exists a directed path from v to u. We adopt the convention that a vertex v is reachable from itself (even though, in a simple graph, there will never be an edge from v to itself).

In a directed acyclic graph, the reachability relation is a partial order:

- Every vertex v is reachable from itself, by the convention above (Reachability is reflexive).
- If w is reachable from u and u is reachable from v, then w is reachable from v (Reachability is transitive).
- If u is reachable from v, then v is not reachable from u, since otherwise there would exist a directed cycle (Reachability is anti-symmetric).

As a result, we can think of reachability as being analogous to an inequality operator, so instead of saying 'u is reachable from v', we can write $v \leq u$. In some sources, reachability in directed acyclic graphs is described with terminology borrowed from trees: if $v \leq u$, v is an *ancestor* of u and u is a *descendant* of v. In general, the reachability relation behaves very similarly to the $\leq$

operator. The main difference is that for two (distinct) numbers a and b, either $a \leq b$ or $b \leq a$. In a directed acyclic graph, there may be pairs of vertices u and v where $v \nleq u$ and $u \nleq v$.

Let G be a directed acyclic graph. A vertex v is called *minimum* if it has no ancestors. Similarly, v is *maximum* if it has no descendants. A sequence of vertices $v_1, \dots, v_n$ is *topologically sorted* if every vertex appears before any of its descendants. Topological sorting can be considered a generalization of 'ordinary' sorting, since a sequence of integers $a_1, \dots, a_k$ can be sorted by creating a graph with each a_i as a vertex, inserting arcs $a_i a_j$ whenever $a_i < a_j$, and generating a topologically sorted ordering of the vertices.

Since a minimum vertex has no ancestors, it can be placed at the beginning of a topologically sorted ordering. This observation leads to a simple algorithm to topologically sort the graph: Repeatedly choose a minimum element and delete it from the graph until no elements remain. It can be shown that any non-empty directed acyclic graph has at least one minimum element. Algorithm 28 gives pseudocode for this approach. Note that InDegree(v) is the number of incoming arcs to vertex v. When InDegree(v) = 0, v must be a minimum element.

Algorithm 28 Naive Topological Sorting

```
procedure TopologicalSortNaive($G_{in}$)
    $G \leftarrow$ Copy of $G_{in}$
    Sorted $\leftarrow$ Empty Sequence
    while $G$ is non-empty do
        for each vertex $v$ of $G$ do
            if InDegree($v$) = 0 then
                Append $v$ to Sorted
                Delete $v$ from $G$
            end if
        end for
    end while
end procedure
```

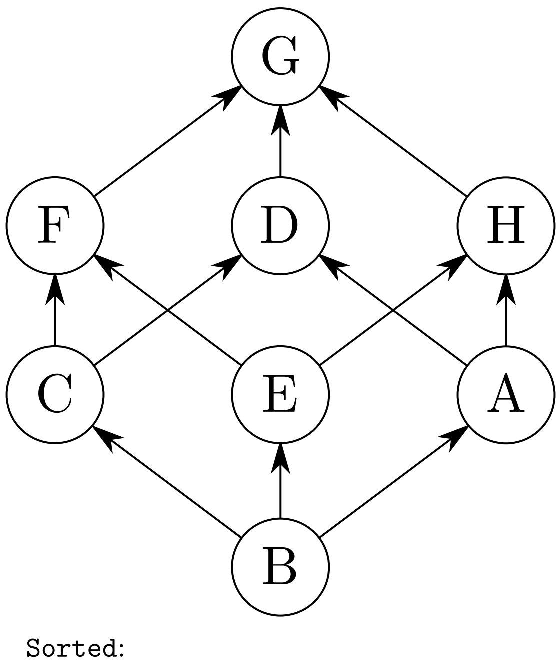


Figure 83: Naive Topological Sorting - Initial Graph

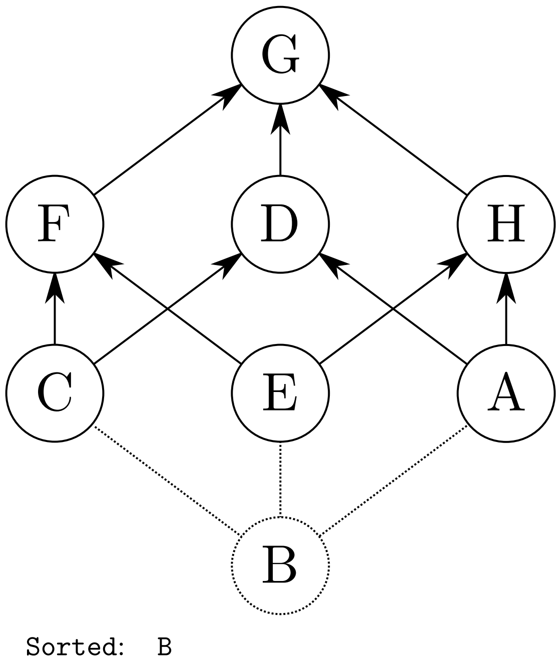
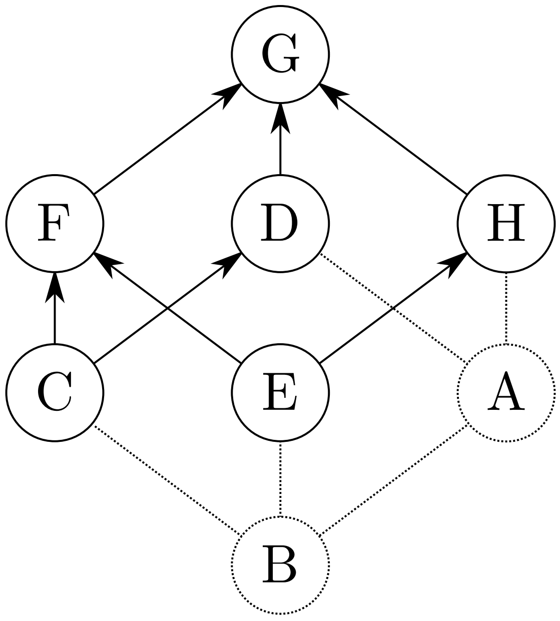
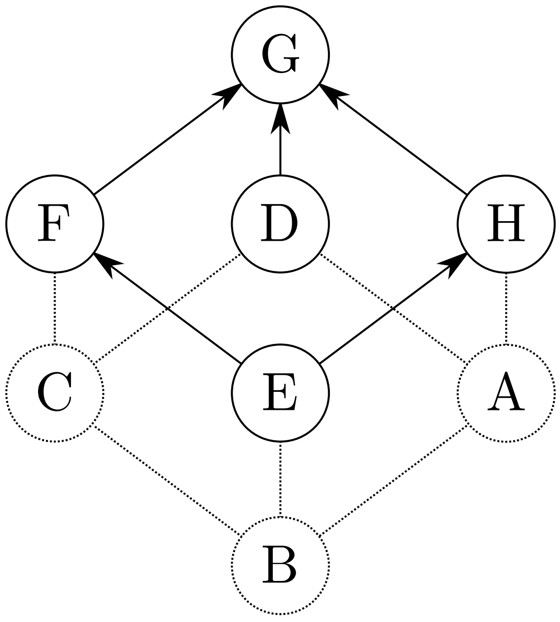


Figure 84: Naive Topological Sorting - Step 1



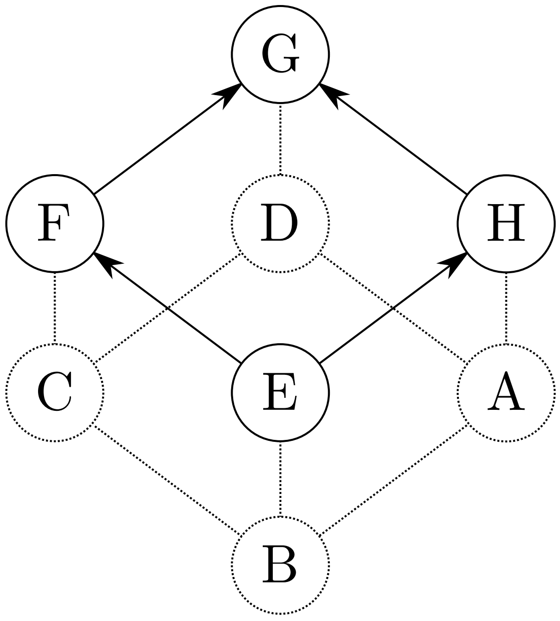
```
Sorted:  B A
```

Figure 85: Naive Topological Sorting - Step 2



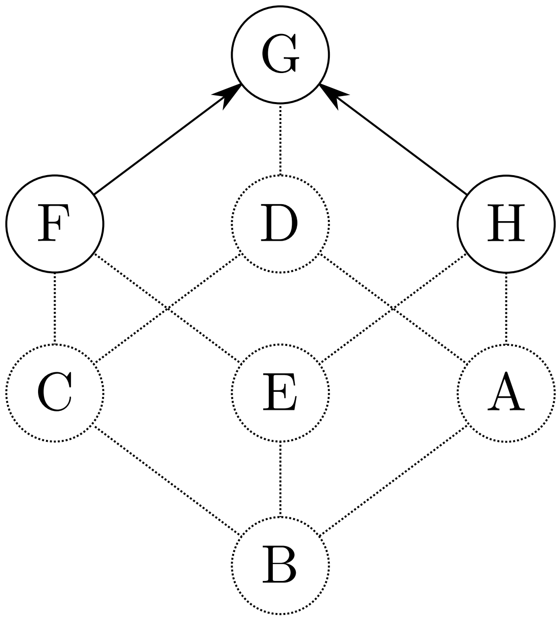
```
Sorted:  B A C
```

Figure 86: Naive Topological Sorting - Step 3



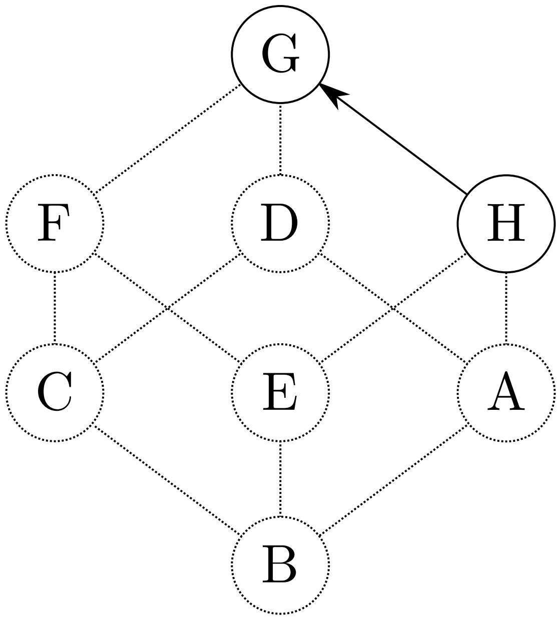
```
Sorted:  B A C D
```

Figure 87: Naive Topological Sorting - Step 4



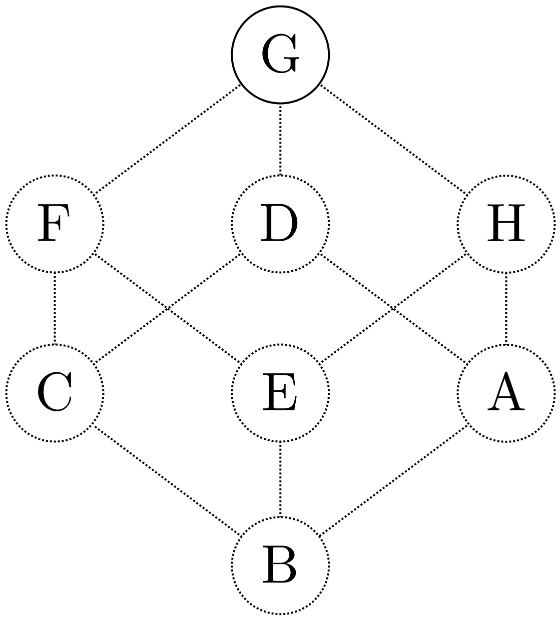
```
Sorted:  B A C D E
```

Figure 88: Naive Topological Sorting - Step 5



Sorted: B A C D E F

Figure 89: Naive Topological Sorting - Step 6



Sorted: B A C D E F H

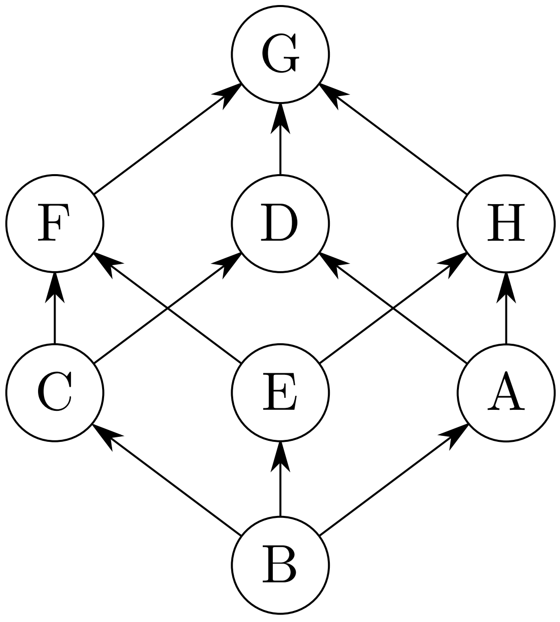
Figure 90: Naive Topological Sorting - Step 7

Figures 83 through 90 illustrate the execution of algorithm 28. Nodes and edges which have been removed from the graph are indicated with a dotted outline. The last step (in which the final vertex is removed) is omitted. The final ordering is `B A C D E F H G`.

Since the elements which become minimum at each step are always neighbours of the last vertex removed, algorithm 28 can be improved by tracking the degree of each vertex and using a queue to store minimum vertices as they are created, rather than explicitly deleting vertices and searching through the graph to find the new minimum vertices. Algorithm 29 implements these improvements.

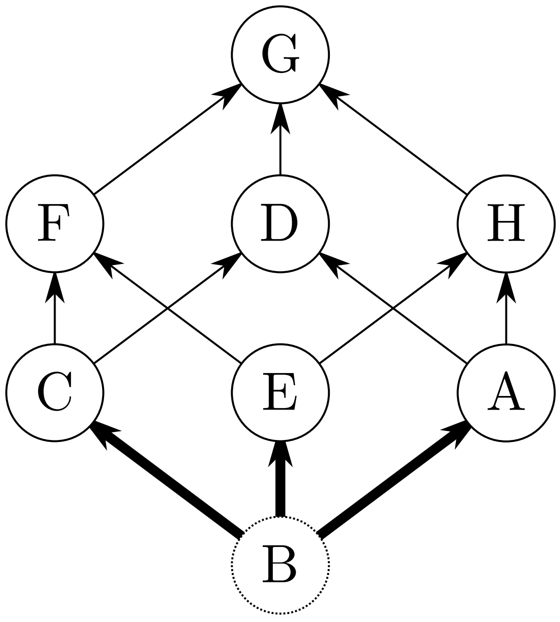
Algorithm 29 Improved Topological Sorting. Although this algorithm is derived from algorithm 28, it is structurally a variant of BFS.

```
procedure TopologicalSortImproved($G$)
    Sorted ← Empty Sequence
    $Q$ ← Empty Queue
    degrees ← Array of size $|V(G)|$
    for each vertex $v$ of $G$ do
        degrees[$v$] ← InDegree($v$)
        if degrees[$v$] = 0 then
            Enqueue $v$ in $Q$
        end if
    end for
    while $Q$ is non-empty do
        $v$ ← Dequeue($Q$)
        Append $v$ to Sorted
        for each neighbour $u$ of $v$ do
            degrees[$u$] = degrees[$u$] − 1
            if degrees[$u$] = 0 then
                Enqueue $u$ in $Q$
            end if
        end for
    end while
end procedure
```



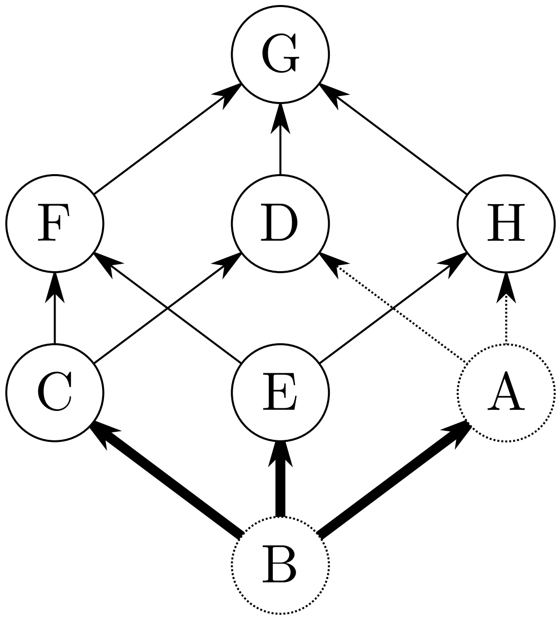
```
Q:        B
Sorted:
```

Figure 91: Improved Topological Sorting - Initial Graph



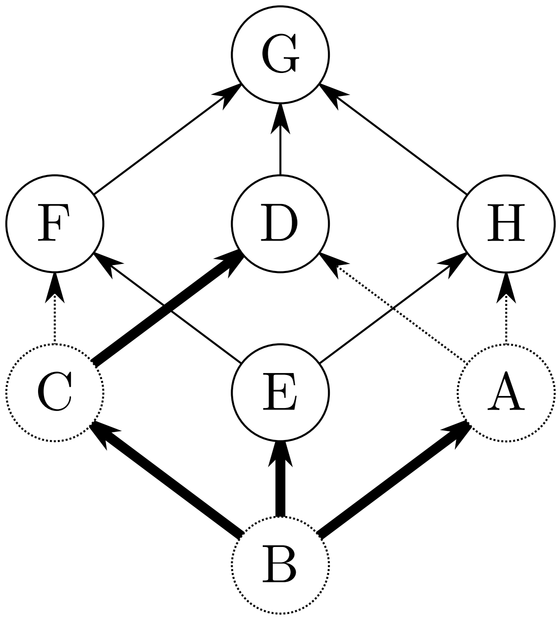
```
Q:        A C E
Sorted:   B
```

Figure 92: Improved Topological Sorting - Step 1



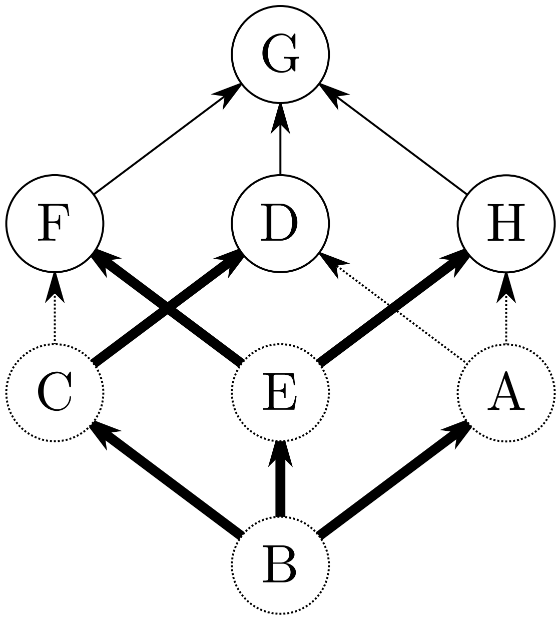
```
Q:       C E
Sorted:  B A
```

Figure 93: Improved Topological Sorting - Step 2



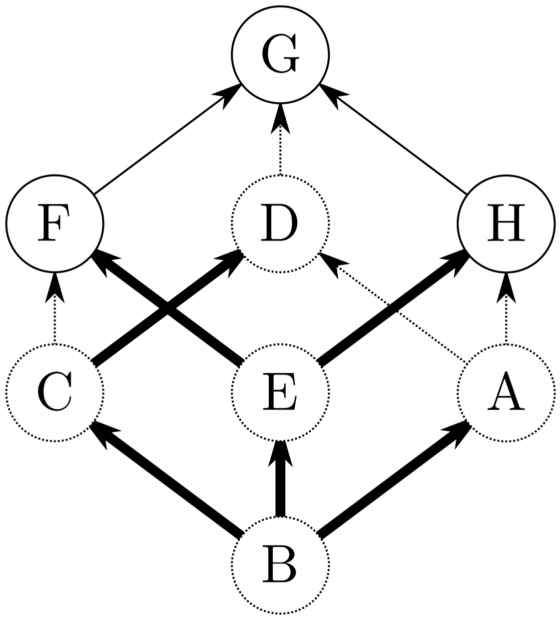
```
Q:       E D
Sorted:  B A C
```

Figure 94: Improved Topological Sorting - Step 3



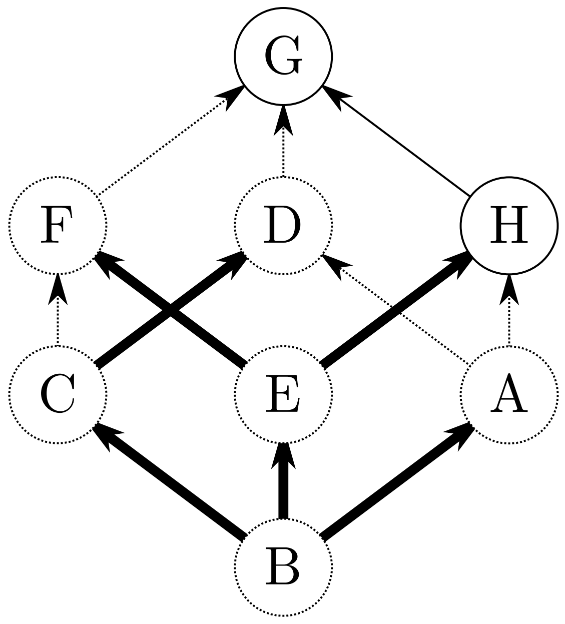
```
Q:       D F H
Sorted:  B A C E
```

Figure 95: Improved Topological Sorting - Step 4



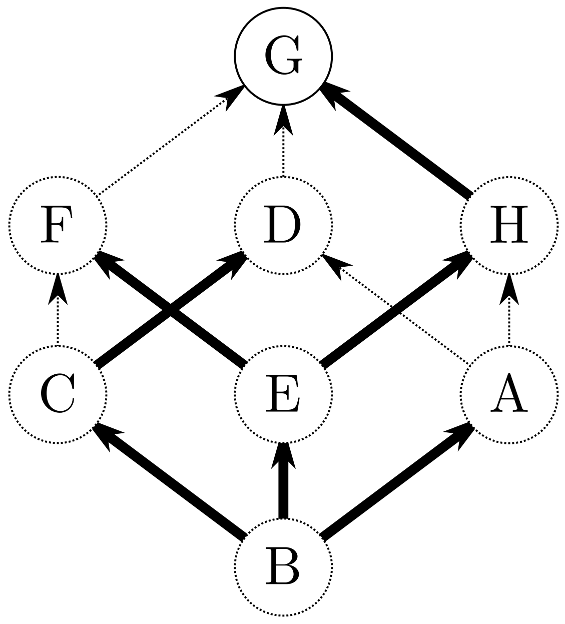
```
Q:       F H
Sorted:  B A C E D
```

Figure 96: Improved Topological Sorting - Step 5



```
Q:        H
Sorted:   B A C E D F
```

Figure 97: Improved Topological Sorting - Step 6



```
Q:        G
Sorted:   B A C E D F H
```

Figure 98: Improved Topological Sorting - Step 7

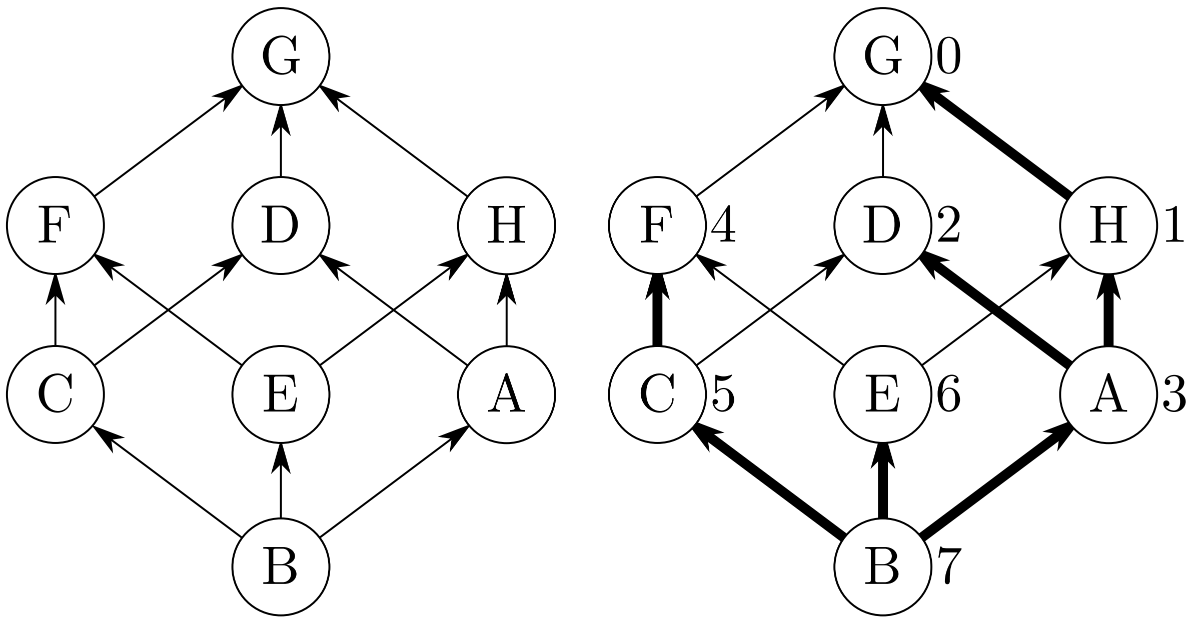
Figures 91 through 98 illustrate the execution of algorithm 29. Edges which are followed to a vertex which is not a minimum (at the current step) are dotted. Edges which are followed to a vertex which is a minimum are darkened. Note that the darkened edges form a BFS tree, but the tree is constructed in the opposite order from the typical BFS algorithm, since vertices are added to the tree after the last visit from an ancestor, rather than after the first. The final ordering is `B A C E D F H G`.

Algorithm 29 is actually a modified breadth-first traversal of the graph. Topological sorting can also be implemented using a recursive depth-first traversal. If $u \leq v$, then the post-order index assigned to u in a depth-first traversal will always be greater than the post-order index assigned to v. In graphs with a unique minimum element M, every vertex v must be reachable from M, so a topologically sorted ordering of vertices can be obtained by reversing the post-order numbering assigned by a depth-first search starting at M. In graphs with multiple minimum vertices, more than one depth-first traversal is required, but each vertex is visited by only one traversal. Algorithm 30 gives pseudocode for DFS-based topological sorting. Figure 99 illustrates the broad steps of algorithm 30 on the graph from the previous examples. The final ordering is `B E C F A D H G`.

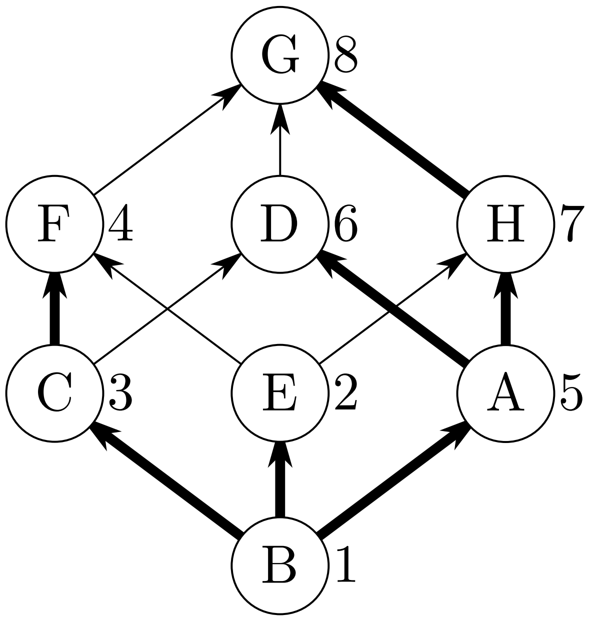
Algorithm 30 Topological Sorting with DFS

```
{Global Variable}
{In an actual implementation, this would likely be a local}
{variable passed by reference to the DFS function.}
nextIndex ← 0
procedure PostOrderDFS($G$, indices, covered, $v$)
    covered[$v$] = true
    for each neighbour $u$ of $v$ do
        if covered[$u$] = true then
            {$u$ has already been visited}
            Continue
        end if
        PostOrderDFS($G$, indices, covered, $u$)
    end for
    indices[$v$] = nextIndex
    nextIndex = nextIndex + 1
end procedure

procedure TopologicalSortDFS($G$)
    covered ← Array of size $|V(G)|$, initialized to false
    indices ← Array of size $|V(G)|$
    nextIndex ← 0
    for each vertex $v$ of $G$ do
        if InDegree($v$) $\neq 0$ then
            {$v$ is not a minimum element}
            Continue
        end if
        PostOrderDFS($G$, indices, covered, $v$)
    end for
    {At this point, all vertices will be covered}
    {Reverse the numbering to obtain a topological ordering}
    {The index of vertex $i$ in the sorted sequence will be $|V(G)| -$ indices[$i$]}
    Sorted ← Empty Sequence
    for $i = 0, \dots, |V(G)| - 1$ do
        Sorted[$|V(G)| -$ indices[$i$]] = $v_i$
    end for
end procedure
```



(a) Initial Graph (b) DFS tree with post-order numbering



(c) Reversed post-order numbering

Figure 99: The steps of algorithm 30.

27 Mathematical Optimization

In the context of an algorithms course, the term 'dynamic programming' is easy to misinterpret as referring to computer programming. Although dynamic programming is indeed relevant to computer programming, the word 'programming' is used to mean 'optimization' in this context.

Before the advent of computers, 'programming' and 'program' were often used to refer to problems involving scheduling and logistics. The study of military logistics during the second world war led to the development of the field of 'Operations Research', in which mathematical optimization plays a large role. Coincidentally, some of the first programmable computers were involved in solving some of these military logistics problems.

To avoid confusion, the term 'mathematical optimization' tends to be used instead of 'mathematical programming' in modern sources. However, certain specific types of optimization continue to be called 'programming' for historical reasons, such as 'Linear programming', 'Convex Programming' and 'Dynamic Programming'. While computers are often used to solve mathematical programs, nothing about mathematical programming is inherently connected to computer programming.

The mathematical optimization problems that are typically discussed in calculus courses fall into the general sub-discipline of 'non-linear programming'. In these problems, we seek to minimize (or maximize) the value of a function (the *objective function*) subject to a set of constraints. A typical non-linear program over the real numbers has a general form similar to

$$\begin{array}{lll} \text{minimize} & f(x_1,\dots,x_k) & \\ \text{subject to} & g_1(x_1,\dots,x_k) & \leq 0 \\ & \vdots & \\ & g_n(x_1,\dots,x_k) & \leq 0 \\ & h_1(x_1,\dots,x_k) & = 0 \\ & \vdots & \\ & h_m(x_1,\dots,x_k) & = 0 \end{array}$$

In two dimensions ($k = 2$), a non-linear program can be illustrated graphically. Figure 100 shows an example of such a problem. There are currently no known algorithms to solve general non-linear programs in the real numbers reliably (although there are high-accuracy numerical algorithms for certain well-behaved sub-classes of problems).

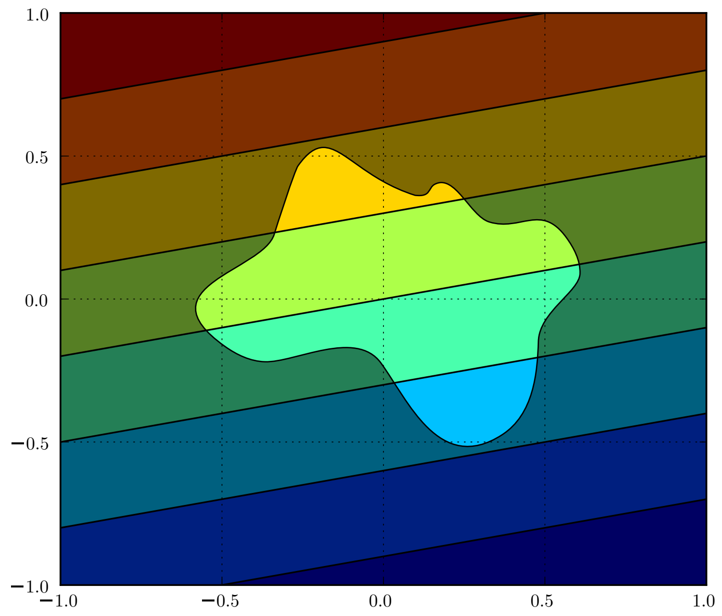


Figure 100: Contour plot of the function $f(x, y) = 5y - x$ with a constraint region highlighted. The minimum of f inside the constraint region lies in the lower-right of the region. Note that even though the objective function is linear, this is not a linear program since the constraint region is not a convex polygon.

A *linear program* is a mathematical program in which all functions are linear (having the form $a_1x_1 + a_2x_2 + \ldots + a_kx_k$, where each a_i is a constant). Although the restriction to linear functions significantly reduces the expressive power of the problem, many practical problems (or approximations to them) can be modeled with linear programs. The added restrictions also enable algorithms to solve linear programs with certainty. The most famous algorithm (and arguably one of the most important algorithms of the 20th century) for linear programming is the Simplex Algorithm, which (when used judiciously) can find the optimal solution to any linear program. The simplex algorithm is exponential in the worst case, but for practical purposes it is very efficient. Figure 101 illustrates a linear program in 2 dimensions.

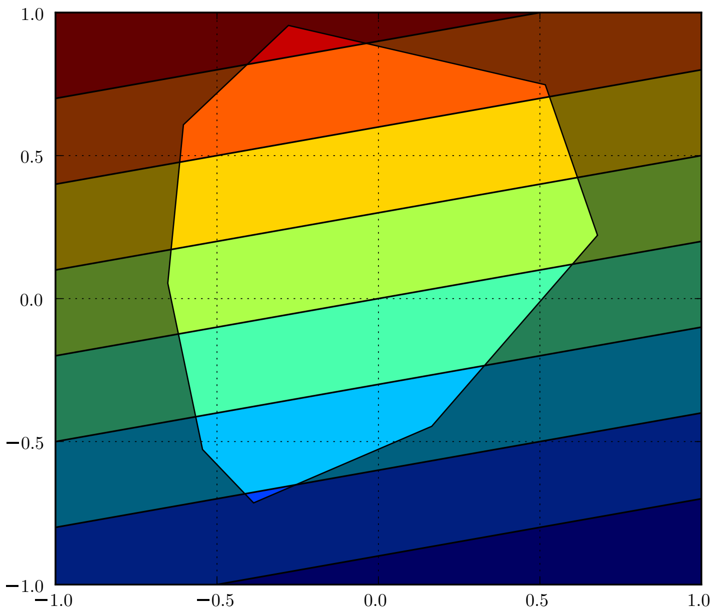


Figure 101: Contour plot of the function $f(x, y) = 5y - x$ with a linear constraint region highlighted. The minimum of f inside the constraint region lies in the lower-left of the region.

An *integer program* is a mathematical program in which all variables, and the result, must take integer values. General integer programs tend to be more difficult to solve than real-valued linear programs. Given an integer program, it is not sufficient to simply round the value of the corresponding real-valued program to an integer. However, a linear integer program can be solved with a sequence of real-valued linear programs using a 'branch-and-bound' algorithm. In some cases, an integer program can be solved directly with a linear program, if it can be proven that the optimal value of the linear program (and the associated optimal values of the input variables) is an integer.

The term 'dynamic programming' was coined to describe optimization problems in which the goal is to optimize a series of decisions, where the path taken by one decision may affect the options available for the next decision. For computational problems that can be decomposed into a set of (possibly overlapping) simpler sub-problems, a 'dynamic programming solution' seeks to evaluate sub-problems in such a way as to minimize the total computation needed. Often, this is done by finding an order in which sub-problems can be evaluated that allows computed results to be recycled. It may also involve finding a decomposition with the smallest number of sub-problems, or a decomposition into particularly easy sub-problems. It is important to note, however, that the term 'dynamic programming' refers to the *design* of the algorithm, not to the mechanism of the algorithm itself (although there are certain characteristics which many dynamic programming solutions share).

28 The Floyd-Warshall Algorithm

The *transitive closure* of a graph G is a graph C in which for every $u, v \in V(G)$, the edge uv is in $E(C)$ if and only if there exists a $u - v$ path in G. The transitive closure of a connected undirected graph on n vertices is the complete graph K_n. Similarly, the transitive closure of a graph with connected components of sizes $n_1, n_2, \ldots, n_k$ is a graph whose components are $K_{n_1}, K_{n_2}, \ldots, K_{n_k}$. It is, however, useful in programming when an application needs to quickly determine whether paths exist between pairs of vertices (without needing to know the specific path).

Transitive closure is also defined for directed graphs, with the arc uv added to $E(C)$ if and only if there exists a directed path from u to v. The strongly connected components of a directed graph become complete directed graphs in the transitive closure. A simple method for computing the transitive closure of a graph G is to run a traversal algorithm starting at each vertex u, and add an edge uv to the transitive closure for every vertex v encountered by the traversal. Algorithm 31 gives pseudocode for a DFS-based transitive closure algorithm for directed graphs. Note that BFS can also be used with only minor modifications.

Algorithm 31 Transitive Closure with DFS

```
procedure TCDFS($G, C$, covered, $u, v$)
    covered[$v$] = true
    for each neighbour $w$ of $v$ do
        if covered[$w$] = true then
            {$w$ has already been visited}
            Continue
        end if
        Add the arc $uw$ to $C$
        TCDFS($G, C$, covered, $u, w$)
    end for
end procedure

procedure TransitiveClosureDFS($G$)
    $C \leftarrow$ Empty graph on the same vertex set as $G$
    covered $\leftarrow$ Array of size $|V(G)|$, initialized to false
    for each vertex $u$ of $G$ do
        TCDFS($G, C$, covered, $u, u$)
    end for
end procedure
```

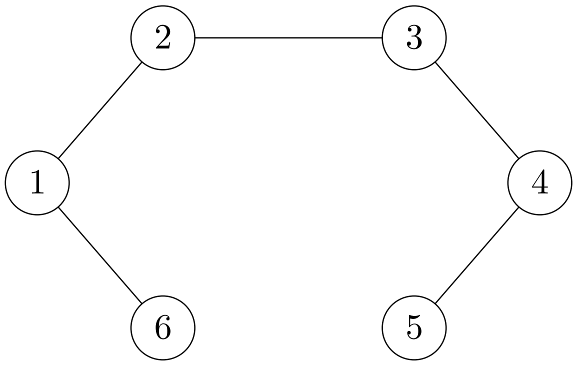
DFS has a worst-case running time of $O(n^2)$ in a graph on n vertices. Algorithm 31 runs n independent DFS traversals, so the worst case running time of algorithm 31 is $O(n^3)$.

Warshall's algorithm for transitive closure forms the structural basis for Floyd's algorithm for computing all-pairs shortest paths. Since the all-pairs shortest paths problem is a generalization of transitive closure, the term 'Floyd-Warshall algorithm' is used to describe both variants. Algorithm 32 gives the transitive closure variant.

Algorithm 32 Transitive Closure with the Floyd-Warshall algorithm

```
procedure TransitiveClosureFW($G$)
    $n \leftarrow |V(G)|$
    $C \leftarrow$ Copy of $G$
    for $k = 1, \dots, n$ do
        for $i = 1, \dots, n$ do
            for $j = 1, \dots, n$ do
                if $v_i v_k \in E(G)$ and $v_k v_j \in E(G)$ then
                    Add an arc $v_i v_j$ to $C$
                end if
            end for
        end for
    end for
end procedure
```

When the graphs G and C are represented with adjacency matrices (allowing constant time edge lookup and insertion operations), algorithm 32 has a worst-case running time of $O(n^3)$. Figures 102 through 106 show the operation of algorithm 32 on an undirected graph. Figures 107 through 111 show the steps for a directed graph. The transitive closure of both graphs is completed after the $k = 4$ step, so the $k = 5$ and $k = 6$ steps are not shown. In both figures, vertices are labeled with their numerical index and the edges created at each step are darkened.

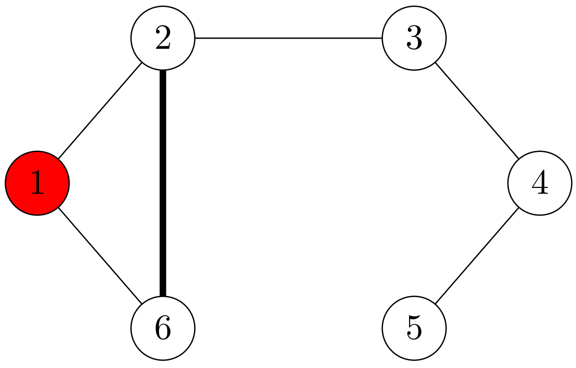


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	0	0	0	0	0

(b) Adjacency Matrix

Figure 102: Algorithm 32 (undirected) - Initial Graph.

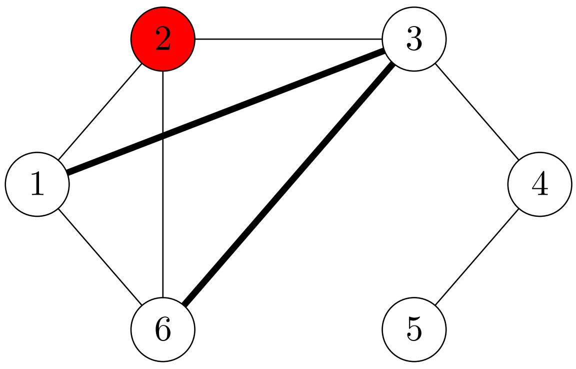


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	**1**
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	**1**	0	0	0	0

(b) Adjacency Matrix

Figure 103: Algorithm 32 (undirected) - $k = 1$

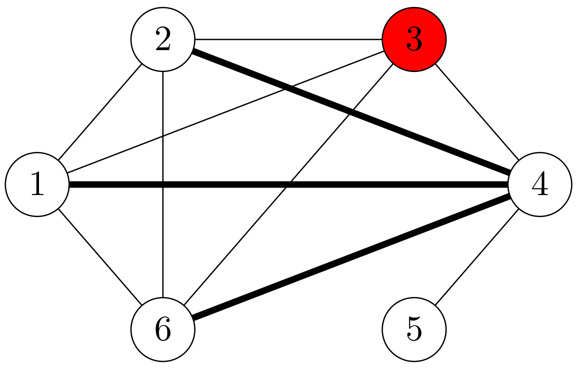


(a) Graph

	1	2	3	4	5	6
1	0	1	**1**	0	0	1
2	1	0	1	0	0	1
3	**1**	1	0	1	0	**1**
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	1	**1**	0	0	0

(b) Adjacency Matrix

Figure 104: Algorithm 32 (undirected) - $k = 2$

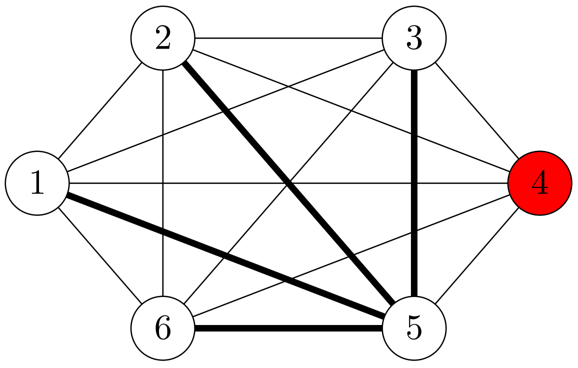


(a) Graph

	1	2	3	4	5	6
1	0	1	1	**1**	0	1
2	1	0	1	**1**	0	1
3	1	1	0	1	0	1
4	**1**	**1**	1	0	1	**1**
5	0	0	0	1	0	0
6	1	1	1	**1**	0	0

(b) Adjacency Matrix

Figure 105: Algorithm 32 (undirected) - $k = 3$

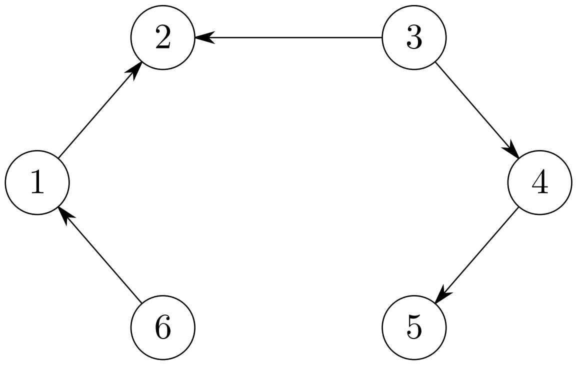


(a) Graph

	1	2	3	4	5	6
1	0	1	1	1	**1**	1
2	1	0	1	1	**1**	1
3	1	1	0	1	**1**	1
4	1	1	1	0	1	1
5	**1**	**1**	**1**	1	0	**1**
6	1	1	1	1	**1**	0

(b) Adjacency Matrix

Figure 106: Algorithm 32 (undirected) - $k = 4$

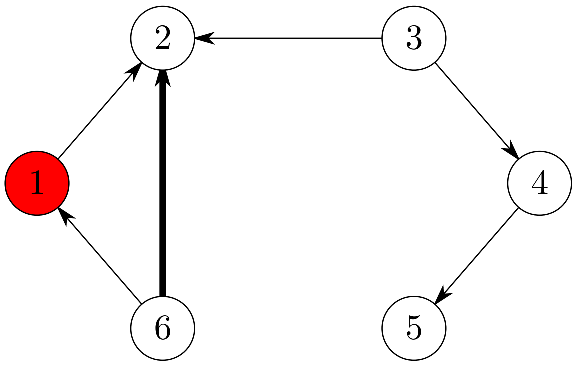


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	0	0	0	0	0

(b) Adjacency Matrix

Figure 107: Algorithm 32 (directed) - Initial Graph.

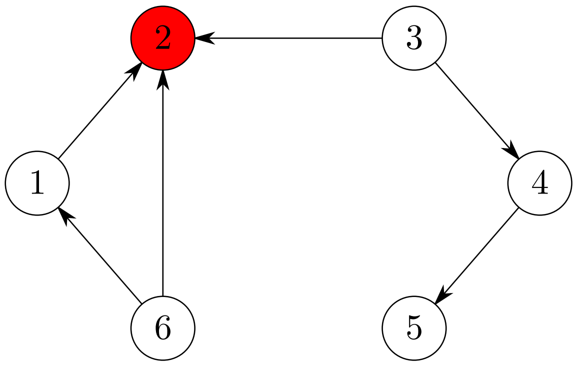


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	**1**	0	0	0	0

(b) Adjacency Matrix

Figure 108: Algorithm 32 (directed) - $k = 1$

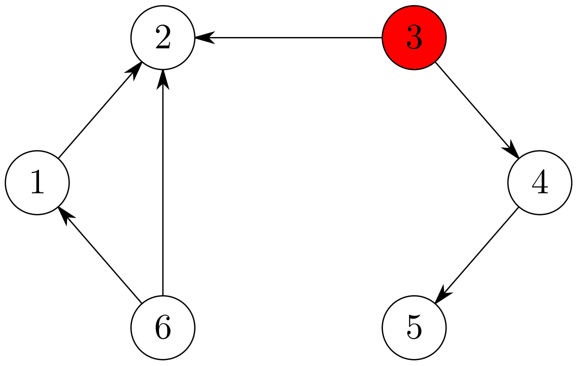


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	1	0	0	0	0

(b) Adjacency Matrix

Figure 109: Algorithm 32 (directed) - $k = 2$

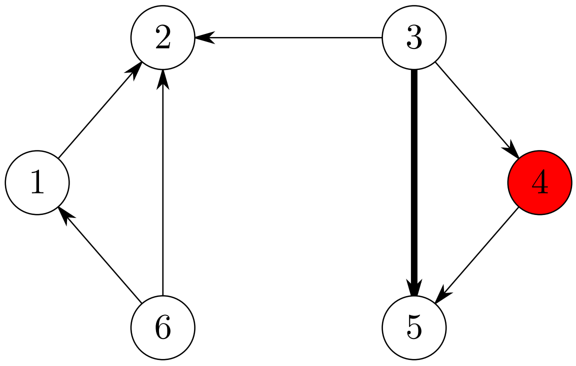


(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	0	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	1	0	0	0	0

(b) Adjacency Matrix

Figure 110: Algorithm 32 (directed) - $k = 3$



(a) Graph

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	0
3	0	1	0	1	**1**	0
4	0	0	1	0	1	0
5	0	0	0	1	0	0
6	1	1	0	0	0	0

(b) Adjacency Matrix

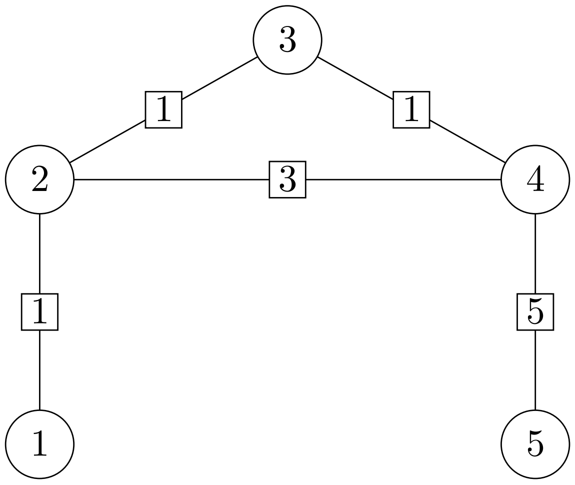
Figure 111: Algorithm 32 (directed) - $k = 4$

Algorithm 33 gives the all-pairs shortest paths variant of the Floyd-Warshall algorithm. The most natural expression of this algorithm operates on a weighted adjacency matrix which stores the input graph and edge weights. In a weighted adjacency matrix A, the entry A_{ij} is set to the weight of the edge between vertex i and j. If there is no edge between vertices i and j, the weight is set to ∞. The value A_{ii} is typically set to zero, although it is not used by the Floyd-Warshall algorithm. When edge weights correspond to distances or lengths, the entry A_{ij} can be interpreted as the distance between vertex i and vertex j. After the Floyd-Warshall algorithm completes, every entry of the resulting matrix will correspond to the minimum possible distance.

Algorithm 33 All-pairs shortest-paths with the Floyd-Warshall Algorithm

```
procedure FloydWarshall($G$)
    $n \leftarrow |V(G)|$
    $A \leftarrow$ Weighted adjacency matrix of $G$
    for $k = 1, \ldots, n$ do
        for $i = 1, \ldots, n$ do
            for $j = 1, \ldots, n$ do
                {Update entry $A_{ij}$ if a shorter path through $k$ exists}
                $A[i][j] \leftarrow \min(A[i][j], A[i][k] + A[k][j])$
            end for
        end for
    end for
end procedure
```

Figures 112 through 117 show the steps of algorithm 33 for a sample (undirected) graph. Vertices are labeled with their index and edges are labeled with their weight. Newly added edges and modified weight values are darkened. Note that the underlying (unweighted) graph at the last step is the transitive closure of the initial graph.

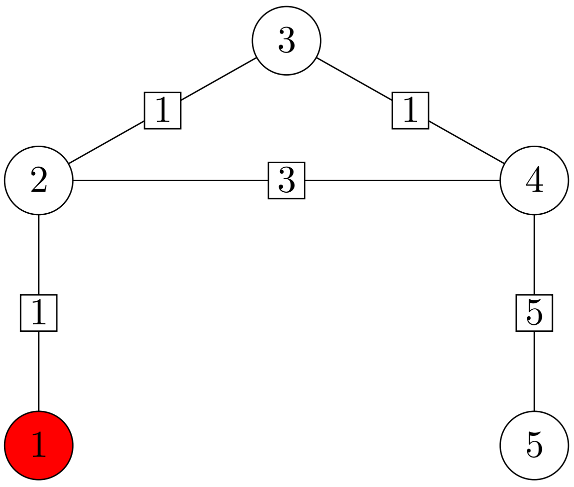


(a) Graph

	1	2	3	4	5
1	0	1	∞	∞	∞
2	1	0	1	3	∞
3	∞	1	0	1	∞
4	∞	3	1	0	5
5	∞	∞	∞	5	0

(b) Weighted Adjacency Matrix

Figure 112: Algorithm 33 (undirected) - Initial Graph.

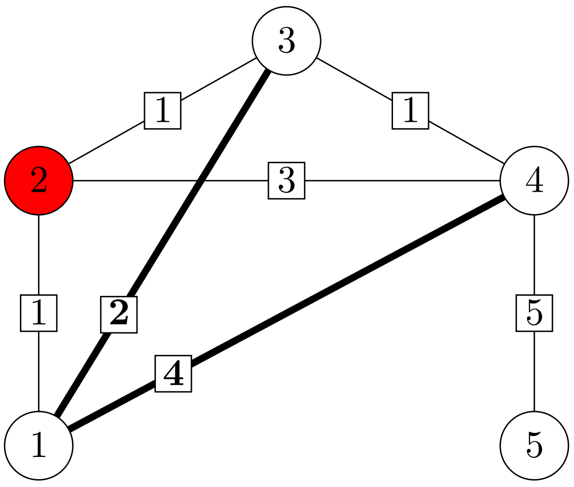


(a) Graph

	1	2	3	4	5
1	0	1	∞	∞	∞
2	1	0	1	3	∞
3	∞	1	0	1	∞
4	∞	3	1	0	5
5	∞	∞	∞	5	0

(b) Weighted Adjacency Matrix

Figure 113: Algorithm 33 (undirected) - $k = 1$

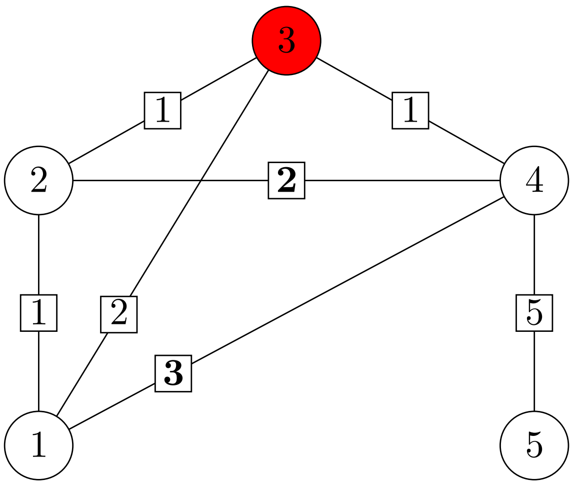


(a) Graph

	1	2	3	4	5
1	0	1	**2**	**4**	∞
2	1	0	1	3	∞
3	**2**	1	0	1	∞
4	**4**	3	1	0	5
5	∞	∞	∞	5	0

(b) Weighted Adjacency Matrix

Figure 114: Algorithm 33 (undirected) - $k = 2$

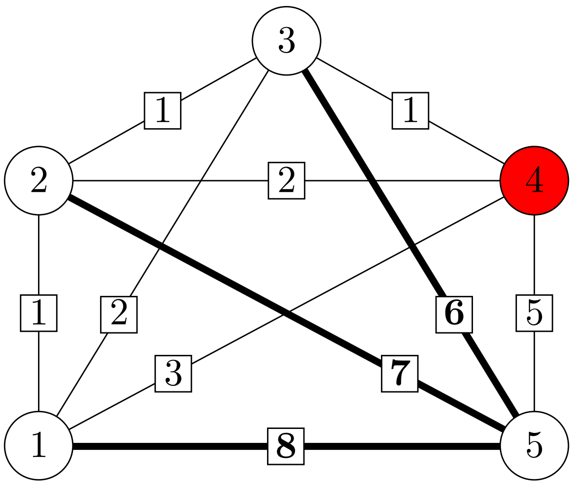


(a) Graph

	1	2	3	4	5
1	0	1	2	**3**	∞
2	1	0	1	**2**	∞
3	2	1	0	1	∞
4	**3**	**2**	1	0	5
5	∞	∞	∞	5	0

(b) Weighted Adjacency Matrix

Figure 115: Algorithm 33 (undirected) - $k = 3$

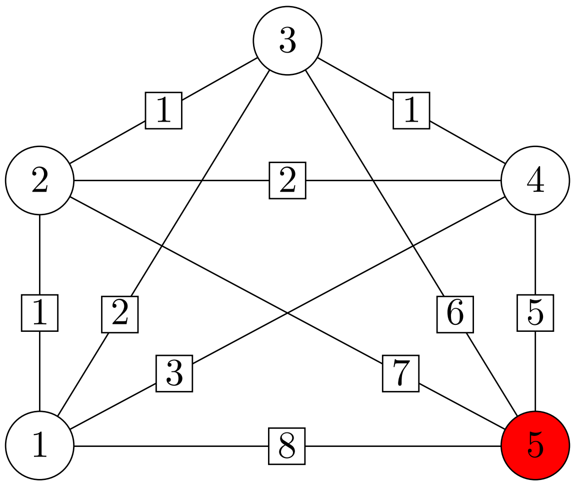


(a) Graph

	1	2	3	4	5
1	0	1	2	3	**8**
2	1	0	1	2	**7**
3	2	1	0	1	**6**
4	3	2	1	0	5
5	**8**	**7**	**6**	5	0

(b) Weighted Adjacency Matrix

Figure 116: Algorithm 33 (undirected) - $k = 4$



(a) Graph

	1	2	3	4	5
1	0	1	2	3	8
2	1	0	1	2	7
3	2	1	0	1	6
4	3	2	1	0	5
5	8	7	6	5	0

(b) Weighted Adjacency Matrix

Figure 117: Algorithm 33 (undirected) - $k = 5$