Discrete Mathematics and Algorithms
ICME Refresher Course

Austin Benson

September 15, 2014

These are the lecture notes for the ICME summer 2014 refresher course on discrete mathematics and algorithms. The material is meant to be preparatory for CME 305, the ICME core course on discrete mathematics and algorithms taught in the winter quarter. Originally, I was planning to teach material on data structures as preparation for programming-based courses. However, I felt that more material on graph algorithms would be more useful. The material on data structures is in the appendix. Please email me (`arbenson@stanford.edu`) with typos, comments, etc. Thanks to Victor Minden for reading these notes.

1 Counting

References: [Ros11]

We are going to start this refresher course with what is hopefully the simplest concept: counting.

Definition 1.1. Let $k \leq n$ be integers. An *ordered arrangement* of k elements from a set of n distinct elements is called a *k-permutation*. An *un-ordered collection* of k elements from a set of n distinct elements is called an *k-combination*.

Theorem 1.2. *The number of k-permutations from a set of n elements is $\frac{n!}{(n-k)!}$, and the number of k-combinations is $\frac{n!}{k!(n-k)!}$.*

We write $\frac{n!}{k!(n-k)!}$ as $\binom{n}{k}$ and say it as "n choose k". As a sanity check, note that if $k = 1$, both terms simplify to n, corresponding to picking one of the elements from the set. And if $k = n$, the number of k-combinations is one—you have to select all elements for the combination. Also, we see that the number of k-permutations is $k!$ multiplied by the number of k-combinations. This corresponds to the $k!$ ways of ordering the elements in the un-ordered combination. Now, an informal proof of Theorem 1.2:

Proof. There are n ways to pick the first element, $n-1$ to pick the second element, and so on. Thus, there are $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$ k-permutations. There are $k!$ ways to order k elements, so there are $\frac{n!}{k!(n-k)!} = \binom{n}{k}$ k-combinations. □

Theorem 1.3. ***The binomial theorem.***

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

Proof. All terms in the product are of the form $x^{n-k}y^k$ for some k. For a fixed k, we count how many ways we get a term $x^{n-k}y^k$. We have a collection of n instances of the term $(x+y)$, and choosing exactly k "y terms" gives us y^k. The number of ways to choose exactly k "y terms" is a k-combination from a set of n elements. □

The $\binom{n}{k}$ term is called the *binomial coefficient*. The binomial theorem is a clever way of proving interesting polynomial and combinatorial equalities. For example:

Example 1.4.

$$\sum_{k=0}^{n} \binom{n}{k} (-1)^k = 0$$

To see this, apply the binomial theorem with $x = 1$ and $y = -1$.

We end this section with an "obvious" principle, but it is worth remembering.

Theorem 1.5. ***The pigeonhole principle.*** *If $n \geq k+1$ objects are put into k containers, then there is a container with at least two objects.*

You can prove the pigeonhole principle by contradiction. An extension of the principle is:

Theorem 1.6. ***The generalized pigeonhole principle.*** *If n objects are put into k containers, then there is a container with at least $\lceil n/k \rceil$ objects.*

2 Computational complexity and big-O notation

References: [Ros11]

The time that algorithms take to solve problems depends on the implementation, the software, the hardware, and a whole host of factors. We use *big-O* notation as a way of simplifying the running time of an algorithm based on the size of its input. The notation allows us to talk about algorithms at a higher level and estimate how implementations of algorithms will behave. We will use big-O notation extensively in this refresher course.

Definition 2.1. Let f, g: $\mathbb{R} \to \mathbb{R}$. We say that f is $O(g)$ if there are positive constants c and N such that for $x > N$, $|f(x)| \leq c|g(x)|$.

Example 2.2. *The number of multiplications and additions needed to multiply two $n \times n$ matrices together is $n^2(2n-1)$, which is $O(n^3)$.*

To see Example 2.2, let $f(n) = n^2(2n-1)$ and $g(n) = n^3$. Choose $N = 1$ and $c = 2$. For $n > N$, $f(n) = 2n^3 - n^2 < 2n^3 = cg(n)$. A general advantage of big-O notation is that we can ignore the "lower order terms":

Example 2.3. *Let $a_0, a_1, \ldots, a_k$ be real numbers. Then $f(x) = a_0 + a_1 x + \ldots a_k x^k$ is $O(x^k)$. To see this, choose $c = |a_0| + |a_1| + \ldots + |a_k|$ and $N = 1$.*

Exercise 2.4. *Show the following:*

- *If f_1 and f_2 are both $O(g)$, then so are $\max(f_1, f_2)$ and $f_1 + f_2$.*
- *If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.*
- $\log n!$ *is $O(n \log n)$.*

Theorem 2.5. *For any k, x^k is $O(2^x)$.*

Proof. The taylor series of 2^x about $x = 0$ is

$$2^x = 1 + (\ln 2)x + \frac{\ln^2 2}{2}x^2 + \ldots + \frac{\ln^k 2}{k!}x^k + \frac{\ln^{k+1} 2}{(k+1)!}x^{k+1} + \ldots$$

Choose $c = \frac{\ln^k 2}{k!}$ and $N = 1$. □

The big-O notation allows us to say that some functions are smaller than other functions in an asymptotic sense. In other words, big-O notation lets us describe *upper bounds*. We may also want to describe whether or not upper bounds are tight, lower bounds, and asymptotic equivalency. There is an alphabet soup that lets us describe these relationships.

Definition 2.6. Let f, g: $\mathbb{R} \to \mathbb{R}$.

- Ω (**big-Omega**): We say that $f(x)$ is $\Omega(g(x))$ if there exists positive constants c and N such that for $x > N$, $|f(x)| \geq c|g(x)|$.
- Θ (**big-Theta**): We say that $f(x)$ is $\Theta(g(x))$ if there exists positive constants c_1, c_2, and N such that for $x > N$ $c_1 g(x) \leq f(x) \leq c_2 g(x)$. Equivalently, $f(x)$ is $O(g(x))$ and $\Omega(g(x))$.

- o (**little-o**): We say that $f(x)$ is $o(g(x))$ if for every positive constant ϵ, there is a positive constant N such that for $x > N$, $|f(x)| \leq \epsilon|g(x)|$.
- ω (**little-omega**): We say that $f(x)$ is $\omega(g(x))$ if for every positive constant C, there is a positive constant N such that for $x > N$, $|f(x)| \geq C|g(x)|$.

Wikipedia has a nice summary of this notation at http://en.wikipedia.org/wiki/Asymptotic_notation#Family_of_Bachmann.E2.80.93Landau_notations.

Example 2.7. *We have that*

- $\cos(x) + 2$ *is* $\Omega(1)$.
- $n^2(2n-1)$ *is* $\Theta(n^3)$.
- $n \log n$ *is* $o(n^2)$.
- *For any* $\epsilon > 0$, $\log n$ *is* $o(n^\epsilon)$.
- n^2 *is* $\omega(n \log n)$.

Theorem 2.8. $\log n!$ *is* $\Theta(n \log n)$

Proof. In Exercise 2.4 we shows that $\log n!$ is $O(n \log n)$ so it is sufficient to show that $\log n!$ is $\Omega(n \log n)$. Note that the first $n/2$ terms in $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ are all greater than $n/2$. Thus, $n! \geq (n/2)^{n/2}$. So $\log n! \geq \log (n/2)^{n/2} = (n/2) \log(n/2) = (n/2)(\log n - \log 2)$, which is $\Omega(n \log n)$. □

Theorem 2.9. *Let* a_0, a_1, …, a_k *be real numbers with* $a_k > 0$. *Then* $f(x) = a_0 + a_1 x + \ldots a_k x^k$ *is* $\Theta(x^k)$.

Proof. From Example 2.3, $f(x)$ is $O(x^k)$. Let $d = \max_{1 \leq i \leq k-1} |a_i|$. Then

$$\begin{aligned} |f(x)| &= \left|a_0 + a_1 x + \ldots + a_k x^k\right| \\ &\geq a_k x^k - |a_{k-1}|x^{k-1} - |a_{k-2}|x^{k-2} - \ldots - |a_0| \\ &\geq a_k x^k - d x^{k-1} \text{ for } x > 1 \end{aligned}$$

Let $c = a_k/2$ Thus, $|f(x)| \geq c|x^k|$ if $(a_k/2)x^k - dx^{k-1} \geq 0$, which holds for $x > 2d/a_k$. Hence, $|f(x)| \geq c|x^k|$ for all $x > \max(1, 2d/a_k)$. □

Theorem 2.10. $\binom{n}{k}$ *is* $\Theta(n^k)$ *for fixed constant* $k \leq n/2$.

Proof. Note that

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

This is just a polynomial with leading term n^k, so $\binom{n}{k}$ is $\Theta(n^k)$ by Theorem 2.9. □

3 Recurrence relations

References: [Ros11, DPV06]

3.1 Definitions and simple examples

Definition 3.1. A recurrence relation for a function $T(n)$ is an equation for $T(n)$ in terms of $T(0)$, $T(1)$, ..., $T(n-1)$.

Example 3.2. *A well-known recurrence relation is the Fibonacci sequence:*

- $f_0 = 0,\ f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$

Recurrence relations are often the easiest way to describe a function, and there are a few reasons why we are interested in them. First, we can solve recurrence relations to get explicit formulae for functions. Second, we can use recurrence relations to analyze the complexity of algorithms.

Theorem 3.3. *Suppose we have an equation $x^2 - ax - b = 0$ with distinct roots r and s. Then the recurrence $T(n) = aT(n-1) + bT(n-2)$ satisfies*

$$T(n) = cr^n + ds^n,$$

where c and d are constants.

Typically, the constants c and d can be derived from the base cases of the recurrence relation. We can now find an explicit formula for the Fibonacci sequence. Set $a = b = 1$, so the equation of interest is

$$x^2 - x - 1 = 0,$$

which has roots $\left(1 \pm \sqrt{5}\right)/2$. By Theorem 3.3,

$$f_n = c\left[\left(1+\sqrt{5}\right)/2\right]^n + d\left[\left(1-\sqrt{5}\right)/2\right]^n$$

Plugging in $f_0 = 0$ and $f_1 = 1$ gives $c = 1/\sqrt{5}$ and $d = -1/\sqrt{5}$. Therefore,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(1+\sqrt{5}\right)/2\right]^n - \frac{1}{\sqrt{5}}\left[\left(1-\sqrt{5}\right)/2\right]^n$$

Theorem 3.3 assumed that the equation had two distinct roots.

Theorem 3.4. *Suppose that the equation $x^2 - ax - b = 0$ has one (repeated) root r. Then the recurrence $T(n) = aT(n-1) + bT(n-2)$ satisfies*

$$T(n) = cr^n + dnr^n,$$

where c and d are constants.

Example 3.5. *Consider the following game. In the first two steps of the game, you are given numbers z_0 and z_1. At each subsequent step of the game, you flip a coin. If the coin is heads, your new score is four times your score from the previous step. If the coin is tails, your new score is the negation of two times your score from the two steps ago. What is your expected score at the n-th step of the game?*

Let X_n be the score at the n-th step of the game.

$$X_n = 4X_{n-1}\mathbb{I}\left(n\text{-th coin toss is heads}\right) - 2X_{n-2}\mathbb{I}\left(n\text{-th coin toss is tails}\right)$$

By linearity of expectation and independence of the n-th coin toss from the previous scores,

$$\mathbb{E}[X_n] = 4\mathbb{E}[X_{n-1}]\Pr\left[\text{heads}\right] - 2X_{n-2}\Pr\left[\text{tails}\right] = 2\mathbb{E}[X_{n-1}] - \mathbb{E}[X_{n-2}]$$

Let $T(n) = \mathbb{E}[X_n]$. By Theorem 3.4, we are interested in the roots of the equation $x^2-2x+1 = 0$, which has a single repeated root $r = 1$. Thus, $T(n) = c1^n + dn1^n = c + dn$. Plugging in $T(0) = z_0$ and $T(1) = z_1$, we get $c = z_0$ and $d = z_1 - z_0$. Therefore, the expected score at the n-th step is

$$z_0 + (z_1 - z_0)n.$$

3.2 The master theorem and examples

Now we will switch gears from getting exact functions for recurrences to analyzing the asymptotic complexity of algorithms. The theory is summarized in Theorem 3.6.

Theorem 3.6. ***The master theorem.*** *Suppose that $T(n) = aT\left(\lceil n/b \rceil\right) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then*

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

3.2.1 Fast matrix multiplication

The first example we are going to work through is fast matrix multiplication. Consider multiplying $C = A \cdot B$, where A and B are both $n \times n$ matrices with n a power of two:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where we have partitioned the matrices into four sub-matrices. Multiplication with the classical algorithm proceeds by combining a set of eight matrix multiplications with four matrix additions:

$$\begin{aligned} M_1 &= A_{11} \cdot B_{11} & M_2 &= A_{12} \cdot B_{21} & M_3 &= A_{11} \cdot B_{12} & M_4 &= A_{12} \cdot B_{22} \\ M_5 &= A_{21} \cdot B_{11} & M_6 &= A_{22} \cdot B_{21} & M_7 &= A_{21} \cdot B_{12} & M_8 &= A_{22} \cdot B_{22} \end{aligned}$$

$$\begin{aligned} C_{11} &= M_1 + M_2 & C_{12} &= M_3 + M_4 \\ C_{21} &= M_5 + M_6 & C_{22} &= M_7 + M_8 \end{aligned}$$

Let $T(n)$ be the number of floating point operations used to multiply two $n \times n$ matrices with this algorithm. There are two costs: the eight multiplications and the four additions. Since adding two $n \times n$ matrices uses $O(n^2)$ floating point operations, the cost of the algorithm is

$$T(n) = 8T(n/2) + O(n^2)$$

Applying the master theorem with $a = 8$, $b = 2$, and $d = 2$, we get that

$$T(n) = O\left(n^{\log_2 8}\right) = O(n^3).$$

Can we do better with a recursive algorithm? Looking at Theorem 3.6, if we could reduce the number multiplications and increase the number of additions, we could get a faster algorithm. This is exactly the idea of Strassen's algorithm:

$$\begin{array}{llll} S_1 = A_{11} + A_{22} & S_2 = A_{21} + A_{22} & S_3 = A_{11} & S_4 = A_{22} \\ S_5 = A_{11} + A_{12} & S_6 = A_{21} - A_{11} & S_7 = A_{12} - A_{22} & \\ T_1 = B_{11} + B_{22} & T_2 = B_{11} & T_3 = B_{12} - B_{22} & T_4 = B_{21} - B_{11} \\ T_5 = B_{22} & T_6 = B_{11} + B_{12} & T_7 = B_{21} + B_{22} & \end{array}$$

$$M_r = S_r T_r, \quad 1 \leq r \leq 7$$

$$\begin{array}{ll} C_{11} = M_1 + M_4 - M_5 + M_7 & C_{12} = M_3 + M_5 \\ C_{21} = M_2 + M_4 & C_{22} = M_1 - M_2 + M_3 + M_6 \end{array}$$

This algorithm uses seven matrix multiplications multiplications (the M_r) and 18 matrix additions / subtractions. The running time is thus

$$T(n) = 7T(n/2) + O(n^2)$$

Applying Theorem 3.6 gives $T(n) = O(n^{\log_2 7}) = o(n^{2.81})$.

3.2.2 k-select

Our second application of the master theorem is the analysis of the k-select algorithm (Algorithm 1). The k-select algorithm finds the k-th smallest number in a list of numbers L. The algorithm can be used to find the median of a set of numbers (how?). This algorithm is similar to Quicksort, which we will see in Section 4.

We are interested in the running time of k-select algorithm. Note that we can form the lists $L_<$, $L_>$, and $L_=$ in $\Theta(n)$ time, where n is the number of elements in the list L. It might happen that the pivot element is always chosen to be the largest element in the array, and if $k = 1$, then the algorithm would take $n + (n-1) + \ldots 1 = \Theta(n^2)$ time. While the *worst-case* running time is $\Theta(n^2)$, we are also interested in the *average-case* running time.

Data: list of numbers L, integer k
Result: the k-th smallest number in L, x_*
pivot $\leftarrow$ uniform random element of L
$L_< \leftarrow \{x \in L | x < pivot\}$
$L_> \leftarrow \{x \in L | x > pivot\}$
$L_= \leftarrow \{x \in L | x = pivot\}$
if $k \leq |L_<|$ **then**
| $x_* \leftarrow$ `select(`$L_<$`, k)`
end
else if $k > |S_<| + |S_>|$ **then**
| $x_* \leftarrow$ `select(`$L_>$`,` $k - |S_<| - |S_>|$`)`
end
else
| $x_* \leftarrow$ pivot
end

Algorithm 1: k-select algorithm, `select(L, k)`

Let's consider how long it takes, on average, for the current list to have $3n/4$ or fewer elements. This occurs if we choose a pivot with at least $1/4$ of the elements of L smaller and at least $1/4$ of the elements are larger. In other words, the pivot has to be in the middle half of the elements of L. Since we choose the pivot uniformly at random, this condition holds with probability $1/2$ in the first call to the k-select algorithm. The expected number of steps until choosing a pivot in the middle half of the data is two (you should be able to solve this after the first probability/statistics refresher lecture).

The *expected* running time of the algorithm, $T(n)$ satisfies

$$T(n) \leq T(3n/4) + O(n)$$

The $O(n)$ term comes from the two steps needed (in expectation) to find a pivot that partitions the data so that the algorithm operates on at most $3/4$ the size of the original list. Using Theorem 3.6, the expected running time is $O(n)$. Since we have to read all n elements of L, the expected running time is $\Theta(n)$.

4 Sorting

References: [CLRS01, DPV06]

The k-select algorithm gave us a taste for ordering of a list of numbers. In general, the task of ordering all elements in some set is called *sorting*.

4.1 Insertion sort

Insertion sort works by inserting an element into an already sorted array. Starting with a sorted list of n numbers, we can insert an additional element and keep the array sorted in $O(n)$ time. The idea of insertion sort is to start with a sorted list consisting of the first element of some un-sorted list L. We then take the second element of L and insert it into the sorted list, which now has size two. Then we take the third element of L and insert it into the sorted list, which now has size three. We carry on this procedure for all elements of L. Algorithm 2 describes the full algorithm.

Data: list of elements L
Result: sorted list S of the elements in L
$n \leftarrow$ number of elements in L
for $i = 0$ *to* $n - 1$ **do**
  $j \leftarrow i$
  while $j > 0$ *and* $L(j-1) > L(j)$ **do**
    swap $L(j-1)$ and $L(j)$
    $j \leftarrow j - 1$
  end
end
$S \leftarrow L$

Algorithm 2: Insertion sort sorting algorithm, `insertionsort(L)`

Note that if the array is already sorted, $L(j-1) > L(j)$ never holds and no swaps are done. Thus, the best case running time is $\Theta(n)$. If the elements are in reverse-sorted order, then all possible swaps are made. Thus, the running time is $1 + 2 + \ldots n - 1 = \Theta(n^2)$. The expected running time is also $\Theta(n^2)$ [CLRS01].

4.2 Merge sort

Merge sort makes use of an auxiliary function, `merge`, that takes two sorted lists, S_1 and S_2 of sizes n_1 and n_2 and creates a sorted list S of size $n_1 + n_2$. The function can be implemented in $O(n_1 + n_2)$ time. Merge sort is presented in Algorithm 3 and visualized in Figure 1.

Let $T(n)$ be the running time of merge sort on a list with n element. Since the merge routine takes $O(n)$ time,

$$T(n) = 2T(n/2) + O(n).$$

By Theorem 3.6, $T(n) = O(n \log n)$. We can see from Figure 1 that we always have $\Theta(\log n)$ steps of recursion. Assuming that the merge function looks at all of the elements (if, for example, the data are copied), then $\Theta(n)$ operations are used at each level of recursion. Thus, the expected, best-case, and worst-case running times are all $\Theta(n \log n)$.

Data: list of elements L
Result: sorted list S of the elements in L
$n \leftarrow$ number of elements in L
if $n = 1$ **then**
  $S \leftarrow L$
end
else
  $L_1 = [L(1), \ldots, L(n/2)]$
  $L_2 = [L(n/2+1), \ldots, L(n)]$
  $S_1 \leftarrow$`mergesort(`L_1`)`
  $S_2 \leftarrow$`mergesort(`L_2`)`
  $S \leftarrow$`merge(`S_1`,` S_2`)`
end

Algorithm 3: Merge sort sorting algorithm, `mergesort(L)`

4.3 Quicksort

Quicksort is similar in flavor to the k-select algorithm (Algorithm 1). We again rely on pivot elements, and the analysis is similar. Algorithm 4 describes quicksort.

In the worst case, the pivot could always be chosen as the smallest element, and $S_>$ would have size $n-1$, $n-2$, $\ldots$, 1 in some path of the recursion tree. This results in a worst-case running time of $\Theta(n^2)$. To get intuition for the average-case running time, we can follow the same analysis as in the analysis of k-select. In expectation, it takes two steps of recursion for the largest list to have size $3n/4$. Thus, the total number of steps in the recursive tree is $O(\log_{4/3} n) = O(\log n)$. Since $O(n)$ work is done in total at each step of the recursion tree, the expected running time is $O(n \log n)$. In the best case, the pivot always splits the data in half. Hence, the best case running time of $\Theta(n \log n)$.

Data: list of elements L
Result: sorted list S of the elements in L
pivot $\leftarrow$ uniform random element of L
$L_< \leftarrow \{x \in L | x < pivot\}$
$L_> \leftarrow \{x \in L | x > pivot\}$
$L_= \leftarrow \{x \in L | x = pivot\}$
$S_< \leftarrow$ `quicksort(`$L_<$`)`
$S_> \leftarrow$ `quicksort(`$L_>$`)`
$S \leftarrow [S_<, S_=, S_>]$

Algorithm 4: Quicksort algorithm

We now introduce the notion of stability in sorting algorithms.

Definition 4.1. Consider a sorting algorithm that sorts a list of elements of L and outputs a sorted list S of the same elements. Let $f(i)$ be the index of the element $L(i)$ in the sorted list S. The sorting algorithm is said to be *stable* if $L(i) = L(j)$ and $i < j$ imply that $f(i) < f(j)$.

Note that insertion sort and merge sort are stable as described. Typical efficient implementations of quick sort are not stable.

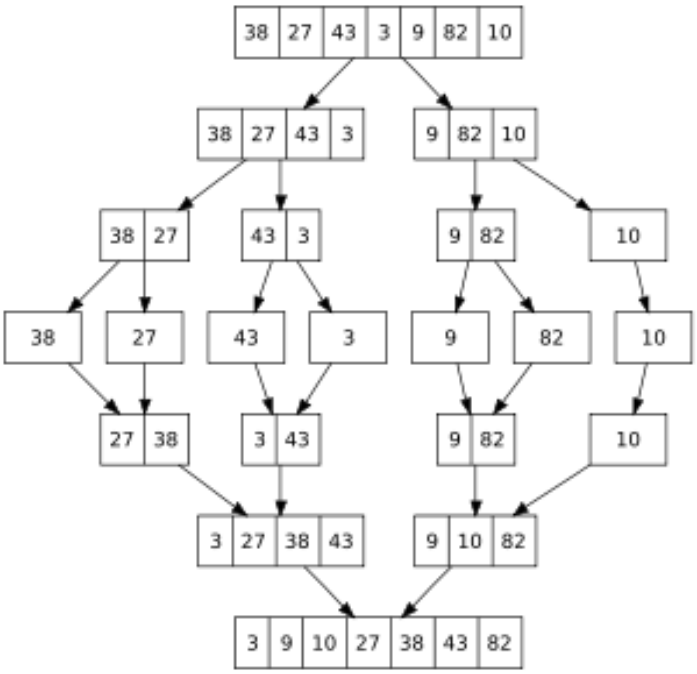

Figure 1: Illustration of the recursion in merge sort. Image from http://i.stack.imgur.com/4tsCo.png.

Table 1 summarizes the sorting algorithms we have covered so far in this section. There are a host of other sorting algorithms, and we have only brushed the surface of the analysis. How do we choose which algorithm to use in practice? Often, we don't need to choose an efficient implementations are already provided, e.g., the C++ standard template library functions `std::sort` and `std::stable_sort`. Quicksort is typically fast in practice and is a common general-purpose sorting algorithm. Also, quicksort has a small memory footprint, which is something we have not considered. However, quicksort is not stable, which might be desirable in practice. Insertion sort is nice for when the array is nearly sorted.

Table 1: Summary of sorting algorithms.

Algorithm	worst-case	best-case	expected	stable?
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

4.4 Bucket sort and radix sort

We have so far been bounded by $\Theta(n \log n)$ complexity. The reason is that we have only considered *comparison-based sorting algorithms*, i.e., algorithms that depend only on a comparison function that determines whether one element is larger than another.

Suppose have a list L of numbers in $\{0, 1, \ldots, 9\}$. We want to sort these numbers in $\Theta(n)$ time.

We can create ten buckets—one for each digit. For each element in the list, we simply add it to the corresponding bucket. The sorted list is then the concatenation of the buckets. This process is described in Algorithm 5. Using, for example, a linked list to implement the buckets, bucket sort is stable. The running time of bucket sort is $\Theta(n)$, but the catch is that the elements all had to be digits. Note that bucket sort makes no comparisons!

Data: list of numbers L with each number in $\{0, 1, \ldots, 9\}$
Result: sorted list S of the numbers in L
$n \leftarrow$ number of elements in L
Create buckets $B_0, \ldots, B_9$. **for** $i = 1$ *to* n **do**
  Append $L(i)$ to bucket $B_{L(i)}$.
end
$S \leftarrow [B_0, \ldots, B_9]$

Algorithm 5: Bucket sort algorithm

We will now leverage bucket sort to implement radix sort. Suppose that we have a list of L numbers in $\{0, 1, \ldots, 10^d - 1\}$. We can use Radix sort (Algorithm 6) to sort this list efficiently. The cost of each iteration of the j index takes $\Theta(n)$ time. Thus, the total running time is $\Theta(dn)$. Note that since the bucket sort is stable, radix sort is also stable.

Data: list of numbers L with each number in $\{0, 1, \ldots, 10^d - 1\}$
Result: sorted list S of the numbers in L
$n \leftarrow$ number of elements in L
for $j = 1$ *to* d **do**
  `// 1 is least significant digit, d is most significant`
  $L \leftarrow$`bucketsort(`L`)` using j-th digit to place in buckets
end
$S \leftarrow L$

Algorithm 6: Radix sort algorithm

Radix sort can be used to handle more general sorting routines. In general, we can consider the elements of L to be length-d tuples such that each element of the tuple has at most k unique values. Then the running time of radix sort is $\Theta(d(n + k))$.

5 Basic graph theory and algorithms

References: [DPV06, Ros11].

5.1 Basic graph definitions

Definition 5.1. A *graph* $G = (V, E)$ is a set V of *vertices* and set E of *edges*. Each edge $e \in E$ is associated with two vertices u and v from V, and we write $e = (u, v)$. We say that u *is adjacent to* v, u *is incident to* v, and u *is a neighbor of* v.

Graphs are a common abstraction to represent data. Some examples include: road networks, where the vertices are cities and there is an edge between any two cities that share a highway; protein interaction networks, where the vertices are proteins and the edges represent interactions between proteins; and social networks, where the nodes are people and the edges represent friends.

Sometimes we want to associate associate an *direction* with the edges to indicate a one-way relationship. For example, consider a predator-prey network where the vertices are animals and an edge represents that one animal hunts the other. We want to express the fox hunts the mouse but not the other way around. This naturally leads to *directed graphs*:

Definition 5.2. A *directed graph* $G = (V, E)$ is a set V of *vertices* and set E of *edges*. Each edge $e \in E$ is an ordered pair of vertices from V. We denote the edge from u to v by (u, v) and say that u *points to* v.

Here are some other common definitions that you should know:

Definition 5.3. A *weighted graph* $G = (V, E, w)$ is a graph (V, E) with an associated weight function $w : E \to \mathbb{R}$. In other words, each edge e has an associated weight $w(e)$.

Definition 5.4. In an undirected graph, the *degree* of a node $u \in V$ is the number of edges $e = (u, v) \in E$.

Definition 5.5. In a directed graph, the *in-degree* of a node $u \in V$ is the number of edges that point to u, i.e., the number of edges $(v, u) \in E$. The *out-degree* of u is the number of edges that u points to, i.e., the number of edges $(u, v) \in E$.

Definition 5.6. An edge (u, u) in a graph is called a *self-edge* or *loop*.

Definition 5.7. A *path* on a graph is a sequence of nodes $v_1, v_2, \ldots v_k$ such that the edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$.

Definition 5.8. We say that node v is reachable from node u if there is a path from u to v.

Definition 5.9. A *cycle* is a path $v_1, v_2, \ldots v_k$ with $k \geq 3$ such that $v_1 = v_k$.

Lemma 5.10. ***The handshaking lemma****. In an undirected graph, there are an even number of nodes with odd degree.*

Proof. Let d_v be the degree of node v.

$$2|E| = \sum_{v \in G} d_v = \sum_{v \in G : d_v \text{ even}} d_v + \sum_{v \in G : d_v \text{ odd}} d_v$$

The terms $2|E|$ and $\sum_{v \in G : d_v \text{ even}} d_v$ are even, so $\sum_{v \in G : d_v \text{ odd}} d_v$ must also be even. □

Definition 5.11. A *directed acyclic graph (DAG)* is a directed graph that has no cycles.

DAGs characterize an important class of graphs. One example of a DAG is a schedule. There is an edge (u, v) if task u must finish before task v. If a cycle exists, then the schedule cannot be completed! We will show how to order the tasks to create a valid sequential schedule in Section 5.3.1.

5.2 Storing a graph

How do we actually store a graph on a computer? There are two common methods: an *adjacency matrix* or an *adjacency list*. In the adjacency matrix, we store a matrix A of size $|V| \times |V|$ with entry $A_{ij} = 1$ if edge (i, j) is in the graph, and $A_{ij} = 0$ otherwise. The advantage of an adjacency list is that we can query the existence of an edge in $\Theta(1)$ time. The disadvantage is that storage is $\Theta(|V|^2)$. In many graphs, the number of edges is far less than $|V|^2$, which makes adjacency lists impractical for many problems. However, adjacency lists are still a useful tools for understanding graphs.

Adjacency lists only store the data that they need. They consist of a length-$|V|$ array of linked lists. The ith linked list contains all nodes adjacent to node i (also called node i's *neighbor list*). We can access the neighbor list of i in $\Theta(1)$ time, but querying for an edge takes $O(d_{\max})$ time, where $d_{\max} = \max_j d_j$.

5.3 Graph exploration

We now cover two ways of exploring a graph: depth-first search (DFS) and breadth-first search (BFS). The goal of these algorithms is to find all nodes reachable from a given node. We will describe the algorithms for undirected graphs, but the generalize to directed graphs.

5.3.1 Depth-first search

This construction of DFS comes from [DPV06]. We will use an auxiliary routine called `Explore(v)`, where v specifies a starting node. We use a vector C to keep track of which nodes have already been explored. The routine is described in Algorithm 7. We have included functions `Pre-visit()` and `Post-visit()` before and after exploring a node. We will reserve these functions for book-keeping to help us understand other algorithms. For example, `Pre-visit()` may record the order in which nodes were explored.

```
Data: Graph G = (V, E), starting node v, vector of covered nodes C
Result: C[u] = true for all nodes reachable from v
C[v] ← true
for (v, w) ∈ E do
    if C[w] is not true then
        Pre-visit(w)
        Explore(w)
        Post-visit(w)
    end
end
```

Algorithm 7: Function `Explore(v)`

Given the explore routine, we have a very simple DFS algorithm. We present it in Algorithm 8. The cost of DFS is $O(|V| + |E|)$. We only call `Explore()` on a given node once. Thus, we only consider the neighbors of each node once. Therefore, we only look at the edges of a given node once.

Data: Graph $G = (V, E)$
Result: depth-first-search exploration of G
for $v \in V$ **do**
  `// Initialized covered nodes`
  $C[v] \leftarrow true$
end
for $v \in V$ **do**
  if $C[v]$ *is not true* **then**
    `Explore(v)`
  end
end

Algorithm 8: Function `DFS(G)`

We now look at an application of DFS: topological ordering of DAGs. A *topological ordering* of the nodes v is an order such that if `order[v]` $<$ `order[w]`, then there is no path from w to v

Example 5.12. *Suppose we define the post-visit function to simply update a count and record the counter, as in Algorithm 9. Ordering the nodes by decreasing `post[v]` after a DFS search gives a topological ordering of a DAG.*

Proof. Suppose not. Then there are nodes v and w such that post$[v]$ > post$[w]$ and there is a path from w to v. If we called the explore function on w first, then post$[v]$ < post$[w]$ as there is a path from w to v. If we called the explore function on v first and post$[w]$ > post$[w]$, then w was reachable from v. Hence, there is a cycle for the graph. □

```
// Post-visit()
post[v] ← count
count ← count + 1
```

Algorithm 9: Post-visit functions to find topological orderings

5.3.2 Breadth-first search

With DFS, we went as far along a path away from the starting node as we could. With breadth-first-search (BFS), we instead look at all neighbors first, then neighbors of neighbors, and so on. For BFS, we use a queue data structure (see Appendix A.1). A *queue* is an object that supports the following two operations:

- `enqueue(x)`: puts the data element x at the back of the queue
- `dequeue()`: returns the oldest element in the queue

Algorithm 10 describes BFS and also keeps track of the distances of nodes from a source node s to all nodes reachable from s.

Data: Graph $G = (V, E)$, source node s
Result: For all nodes t reachable from s, `dist[t]` is set to the length of the smallest path from s to t. `dist[t]` is set to ∞ for nodes not reachable from s

```
for v ∈ V do
    dist[v] ← ∞
end
dist[s] ← 0
Initialize queue Q
Q.enqueue(s)
while Q is not empty do
    u ← Q.dequeue()
    for (u, v) ∈ E do
        if dist[v] is ∞ then
            Q.enqueue(v)
            dist[v] ← dist[u] + 1
        end
    end
end
```

Algorithm 10: Breadth-first search

5.4 Connected components

Definition 5.13. An *induced subgraph* of $G = (V, E)$ specified by $V' \subset V$ is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E | u, v \in V'\}$.

Definition 5.14. In an undirected graph G, a *connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there is a path from u to v in G'.

By "maximal", we mean that there is no V'' such that $V' \subset V''$ and the induced subgraph specified by V'' is also a connected component.

Definition 5.15. In a directed graph G, a *strongly connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there are path from u to v and from v to u in G'.

Definition 5.16. An undirected graph is *connected* if G is a connected component. A directed graph is *strongly connected* if G is a strongly connected component.

We now present a simple algorithm for finding connected components in an undirected graph. First, we update the DFS procedure (Algorithm 8) to increment a counter `count` before calling `Explore()`, with initialization to 0. Then the previsit function assigns connected component numbers.

```
// Pre-visit()
connected_component[v] ← count
```

Algorithm 11: Pre-visit for connected components in undirected graphs

5.5 Trees

Definition 5.17. A *tree* is a connected, undirected graph with no cycles.

Definition 5.18. A *rooted tree* is a tree in which one node is called the root, and edges are directed away from the root.

Note that all trees can be rooted tree with any vertex in the tree since there are no cycles.

Definition 5.19. In a rooted tree, for each directed edge (u, v), u is the *parent* of v and v is the *child* of u.

Note that the parent of a node v must be unique.

Definition 5.20. In a rooted tree, a *leaf* is a node with no children.

Note that all finite trees must have at least one leaf node (we can find them by a topological ordering).

Theorem 5.21. *A tree with n vertices has $n - 1$ edges.*

Proof. We go by induction. With $n = 1$, there are no edges. Consider a graph with k nodes. Let v be a leaf node and consider removing v and (u, v) from the tree, where u is the parent of v. The remaining graph is a tree, with one fewer edge. By induction, it must have $k - 2$ edges. Therefore, the original tree had $k - 1$ edges. □

5.5.1 Binary trees

Definition 5.22. A *binary tree* is a tree where every node has at most two children.

In a binary tree, there are a few natural ways to order the nodes (or *traverse* the tree). We describe them in Algorithm 12.

Exercise 5.23. *Which traversals in Algorithm 12 correspond to DFS and BFS?*

5.5.2 Minimum spanning trees

Definition 5.24. A spanning tree of an undirected graph $G = (V, E)$ is a tree $T = (V, E')$, where $E' \subset E$.

Definition 5.25. The minimum spanning tree $T = (V, E')$ of a weighted, connected undirected graph $G = (V, E)$ is a spanning tree where the sum of the edge weights of E' is minimal.

Two algorithms for finding minimum spanning trees are Prim's algorithm (Algorithm 13) and Kruskal's algorithm (Algorithm 14). Both procedures are straightforward. We won't prove correctness or analyze running time here, but both algorithms can run in time $O(|E| \log |V|)$.

Data: Node v
Result: Traversal of subtree rooted at node v
$v_l \leftarrow$ left child (if it exists) $v_r \leftarrow$ right child (if it exists)
`// Post-order`
Post-order(v_l)
Post-order(v_r)
Visit(v)

`// Pre-order`
Visit(v)
Pre-order(v_l)
Pre-order(v_r)

`// In-order`
In-order(v_l)
Visit(v)
In-order(v_r)

Algorithm 12: Tree traversal algorithms

Data: Connected weighted graph $G = (V, E)$
Result: Minimum spanning tree $T = (V, E')$
$S \leftarrow \{w\}$ for any arbitrary node $w \in V$
$E' \leftarrow \{\}$
while $|S| < |V|$ **do**
  $(u, v) \leftarrow \arg\min_{u \in S, v \notin S} w((u, v))$
  $S \leftarrow S \cup \{v\}$
  $E \leftarrow E' \cup \{(u, v)\}$
end

Algorithm 13: Prim's algorithm for minimum spanning trees

Data: Connected weighted graph $G = (V, E)$
Result: Minimum spanning tree $T = (V, E')$
$E' \leftarrow E$
foreach $e \in E$ *by increasing* $w(e)$ **do**
  $e \leftarrow \arg\min_{(u,v) \in S}$
  if *adding* (u, v) *to* E *does not creates a cycle* **then**
    $E' \leftarrow E' \cup \{e\}$
  end
end

Algorithm 14: Kruskal's algorithm for minimum spanning trees

A Abstract data types and data structures

References: [GT08]

This material is what you would find in a beginner undergraduate computer science course in data structures. This material is not really covered in other courses, but it is often necessary for analyzing the running time of algorithms, *e.g.*, Dijkstra's Algorithm.

A.1 Abstract data types

An *abstract data type* (ADT) is a model for data structures that have similar behavior. They describe only the operations that the data structures provide, but do not describe the implementation. Later, in subsection A.2, we will discuss *data structures* that are used to implement abstract data types. The data structures we use to implement ADTs determine the running time of the operations that the ADTs provide.

A *stack* is an ADT that supports two operations:

- `push(x)`: puts the data element x on top of the stack
- `pop()`: returns the last element pushed onto the stack

We are going to assume that stacks and all other ADTs we discuss also support operations like `size()`, which says how many elements are in the ADT. This also allows us to easily check if data structures are empty. We can assume that this method will always be implemented in practice and will not affect the running time of any algorithms we discuss.

A *queue* is an ADT that supports two operations:

- `enqueue(x)`: puts the data element x at the back of the queue
- `dequeue()`: returns the oldest element in the queue

An *array list* is an ADT that contains a fixed number of contiguously indexed elements and supports the following operations:

- `get(i)`: return the element with index i
- `set(i, x)`: set the element with index i to the value x
- `add(i, x)`: set the element with index i to the value x
- `remove(i)`: remove the element indexed by i

A *tree* is an ADT that consists of "nodes" which contain elements. The nodes satisfy a parent-child relationship in the following sense:

1. There is a special node called the root node.
2. Every node v that is not the root node has a unique parent node w.

The tree ADT supports the following operations:

- `GetNode(v)`: returns the node v

- `Root()`: returns the root node
- `Children(v)`: returns all children of the node v
- `Parent(v)`: returns the parent of node v

For now, we will discuss trees as abstract data types. In Lecture 5, we will see trees as graphs. We are now going to implement *pre-order traversals* and *post-order traversals* of a tree using the stack and queue data types.

Data: tree T
Result: pre-order traversal of the nodes in a tree
Initialize stack S
S.push(T.Root())
while S *is not empty* **do**
  $v \leftarrow S$.pop()
  visit(v)
  for w *in* T.*children(v)* **do**
    S.push(w)
  end
end

Algorithm 15: Pre-order traversal of a tree using a stack

Data: tree T
Result: post-order traversal of the nodes in a tree
Initialize queue Q
Q.enqueue(T.Root())
while Q *is not empty* **do**
  $v \leftarrow Q$.dequeue()
  visit(v)
  for w *in* T.*children(v)* **do**
    Q.enqueue(w)
  end
end

Algorithm 16: Post-order traversal of a tree using a queue

A *priority queue* is an ADT with the following operations:

- `insert(k, x)`: puts the data element x in the priority queue with key k
- `RemoveMin()`: remove the data element with the smallest key and returns the data

Priority queues are often used for sorting, which we will discuss in more detail in Subsection 4. For now, we will describe an abstract sorting algorithm based on a priority queue. The running time of the algorithm will depend on the implementations we discuss in Subsection A.2.

Data: Array list of elements, L
Result: Sorted elements of the array list, L_s
Initialize priority queue Q
for $i = 0$ *to* $L.size()$ *- 1* **do**
  $x \leftarrow L$.get(i)
  Q.insert(x.key(), x)
end
Initialize sorted array list L_s
while Q *is not empty* **do**
  $x \leftarrow Q$.RemoveMin()
  L_s.insert(L_s.size(), x)
end

Algorithm 17: Sorting algorithm that uses a priority queue

A.2 Data structures

Now that we have learned about ADTs, we are going to discuss *data structures*, the tools actually used to implement ADTs.

A.2.1 Linear arrays

Arrays are perhaps the simplest of data structures, and they are used all the time when programming or describing algorithms. A linear array is a collection of elements such that each element can be "accessed" by an "index". We might store a collection of numbers

$$|10|13|12|19|8|101|$$

The above array has six elements, so we might say that the array *is of size six* or *has length six*. If we index by the order, we may say that the element 10 has index 0, the element 13 index 1, the element 12 index 2, and so on.

So what makes an array an array, or how can we formalize this data structure? The only real assumption we make is that we can access any elements in $\Theta(1)$ time given its index. We will write the operation of accessing the element indexed by i as `array[i]`. The idea of an array is intimately tied to memory in a computer, where data is stored as a linear sequence of bytes. In this case the indices, are memory addresses such as `0xab4800`. We assume that, given a memory address, the computer can give us the data stored at that memory address in $\Theta(1)$ time.

The main advantages of arrays are that they are simple to understand and have fast access times. The disadvantages are that it is not clear how to dynamically manage their size and it is

not clear how to add or remove elements. Also, we need a contiguous chunk of memory on the computer to store the array.

Implementing an array list with a linear array

Implementing the array list is quite natural with a linear array. We will assume that there will be at most N elements. We can set the i-th element to a `NULL` value to indicate that an element does not exist. We can implement the linear array ADT as follows:

- `get(i)`: return `array[i]`
- `set(i, x)`: `array[i]` $\leftarrow$ `x`
- `add(i, x)`: `array[i]` $\leftarrow$ `x`
- `remove(i)`: `array[i]` $\leftarrow$ `NULL`

All of these operations take $\Theta(1)$ time, which seems optimal. However, we needed to know ahead of time the length of the list.

Implementing binary trees with a linear array

Suppose we have a binary tree of a fixed depth d. Then there at most $2^d - 1$ nodes (you can prove this by induction), so we can create a length $2^d - 1$ linear array. We will order the nodes so the root node is indexed by 0 We implement the methods for the binary tree ADT as follows:

- `GetNode(v)`: `return array[v]`
- `Root()`: `return array[0]`
- `Children(v)`: returns `array[2*v]` and `array[2 * v + 1]`
- `Parent(v)`: returns `array[` $\lfloor v/2 \rfloor$ `]`

Again, all of these operations are $\Theta(1)$ time, but we needed to know the size of the tree ahead of time.

A.2.2 Linked lists

The central drawback of arrays is that they were not dynamic. *Linked lists* are a data structure that consists of an ordered collection of nodes that store data and can access the data at the subsequent node in $\Theta(1)$ time. There is a "head node" that stores a reference to the first element. The last node in the sequence has an empty pointer, so we can tell that it is the last element in the linked list. Figure 2 illustrates a linked list.

Implementing a queue list with a linked list

Let n be the number of nodes in the linked list at a given time.

- `enqueue(x)`: Adjust the head node to point to x and have x point to the node that the head node points to. This takes $\Theta(1)$ time.
- `dequeue(x)`: Iterate through the linked list to find the last element. This takes $\Theta(n)$ time.

We could add reverse pointers to the structure in Figure 2 and add a pointer to the last node in the sequence (called a "tail node"). This structure is called a *doubly linked list*. With this structure, the `dequeue(x)` operation also takes $\Theta(1)$ time.

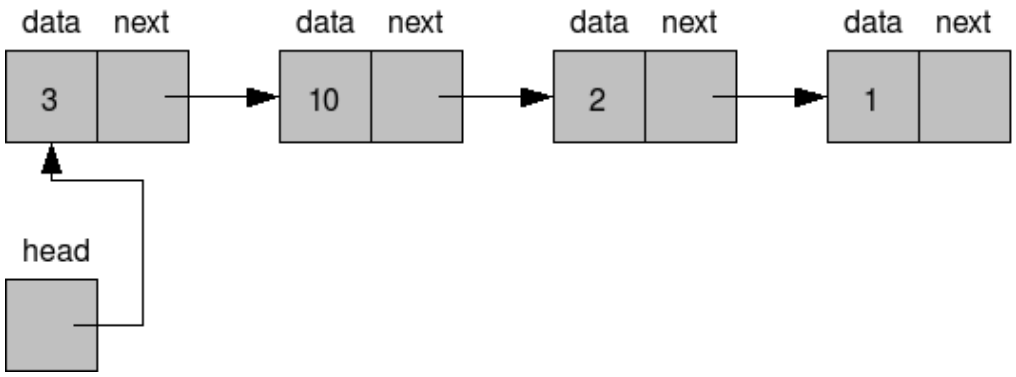

Figure 2: Linked list data structure. Each node stores data and can access the subsequent node in $\Theta(1)$ time. Image from http://www.java2novice.com/images/linked_list.png.

Implementing a priority queue with a linked list

For priority queues, we need to maintain an order to the keys. We will assume that our nodes are ordered from smallest to largest so that the head node points to the minimum element in the priority queue. We can implement the priority queue as follows:

- `insert(k, x)`: Iterate through the linked list until we find the first node with key greater than k. Insert x with key k before this node. In the worst case, this is $\Theta(n)$ time.
- `RemoveMin()`: Remove the node that the to which the head node points. Update the head node to point to the next node.

How long does the abstract sorting algorithm (Algorithm 17) take with a linked list implementation of a priority queue? First, we analyze the time to add all of the elements to the priority queue. In the worst case, the elements are already sorted in the array list so that each `insert(k, x)` call to the priority queue takes the longest time. In this case, the running time is $1+2+\ldots+n$, which is $\Theta(n^2)$.

A.3 Heaps

A *heap* is a tree-based data structure that maintains the *heap property*: the key of a node v is always smaller than the key of all of its children. Figure 3 illustrates this. Let n be the number of nodes in the heap You will have to take my word that heaps have the following properties (see http://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html for more details):

- We can remove the minimum element in the tree and re-arrange the tree to maintain the heap property in $O(\log n)$time.
- We can insert nodes and maintain the heap property in $O(\log n)$ time

Implementing a priority queue list with a linked list

Because heaps already maintain an ordered property, priority queues are easily implemented with heaps:

- `insert(k, x)`: use the heap's insertion method that takes $O(\log n)$ time

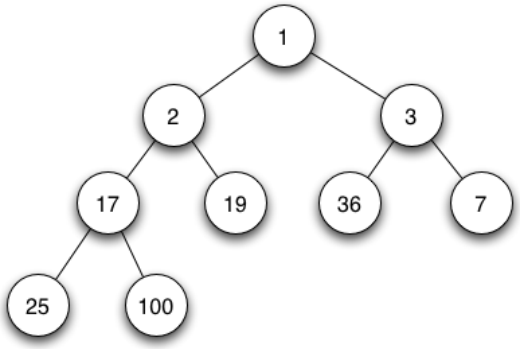

Figure 3: A heap data structure. The tree maintains the *heap property*: the key of a node v is always smaller than the key of all of its children.

- `RemoveMin()`: The root node in the heap tree is the minimum element. We can remove it in $O(\log n)$ time.

How long does Algorithm 17 take with the heap implementation of a priority queue? We insert into the heap n times and remove the minimum node n times. Each of these operations takes $O(\log n)$ time, so in total the sorting time is $O(n \log n)$.

References

[CLRS01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[DPV06] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006.

[GT08] Michael T Goodrich and Roberto Tamassia. *Data structures and algorithms in Java*. John Wiley & Sons, 2008.

[Ros11] Kenneth Rosen. *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill Science, 2011.