CMSC 351 - Introduction to Algorithms
Fall 2014*

Instructor: Hamid Mahini

1 Introduction

In this class you learn what an **algorithm** is, you will learn how to design and analyze algorithms, you will learn fundamental algorithms, and you will also learn about the running times of algorithms.

The word "algorithm" comes from the name of Al-Khwarizmi, Muhammad (a Persian scientist, mathematician and author) who developed the concept of an algorithm in mathematics. An algorithm is an effective method for solving a problem expressed as a finite sequence of steps. Each algorithm is a list of well-defined tasks instructions for computing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state.

2 Mathematical Induction

As you have learned in CMSC 250, mathematical induction is a very powerful proof technique that plays a major role in algorithm design. In this session we briefly review induction through some examples.

It usually works as follows. Let T be a theorem (statement that we want to prove). Suppose T includes a parameter n whose value can be any natural number (a positive integer). Instead of proving directly that T holds for all values of n, we prove the following two conditions:

1. **Basis of the induction**: T holds for $n = 1$
2. **Inductive Step**: For every $n > 1$, if T holds for $n - 1$, then T holds for n.

The hypothesis in the inductive step that the statement holds for n called the **induction hypothesis**. The reason that these two conditions are sufficient is clear. Conditions 1 and 2 imply directly that T holds for $n = 2$. If T holds for $n = 2$, then condition 2 implies that T holds for $n = 3$ and so on. The induction

*Thanks Prof. Hajiaghayi for sharing his notes for CMSC 351.

principle itself is so basic that we consider it as an axiom and it is usually not proved. Proving induction hypothesis is easier in many cases than proving the theorem directly since it comes for free. Thus the induction principle is defined as follows:

> **Induction Principle**
> If a statement T, with a parameter n, is true for $n = 1$, and if for every $n > 1$, the truth of T for $n - 1$ implies its truth for n, then T is true for all natural numbers.

Strong Induction

Another variant, called **strong induction**, says that in the second step we may assume the statement holds for all $m < n$, and then prove the statement is true for n.

> **Strong Induction Principle**
> If a statement T, with a parameter n, is true for $n = 1$, and if for every $n > 1$, the truth of T for all $1 \geq m < n$ implies its truth for n, then T is true for all natural numbers.

2.1 Three simple examples of induction

More examples and exercises on induction are available in Chapter 2 of the book [2] for this course.

Theorem 2.1 *For all natural numbers x and n, $x^n - 1$ is divisible by $x - 1$.*

Proof: The proof is by induction on n. The theorem is trivially true for $n = 1$. We assume $x^{n-1} - 1$ is divisible by $x - 1$ for all natural numbers x. We now prove that $x^n - 1$ is divisible by $x - 1$. The idea is to try to write the expression $x^n - 1$ using $x^{n-1} - 1$, i.e., $x^n - 1 = x(x^{n-1} - 1) + (x - 1)$. The second term is clearly divisible by $x - 1$ while the first term is divisible by $x - 1$ due to the induction hypothesis. ■

Theorem 2.2 *The sum of the first n natural number is $\frac{n(n+1)}{2}$.*

Proof: The proof is by induction on n. For $n = 1$ we have $1 = \frac{1\times 2)}{2} = 1$. We assume the sum of the first n natural numbers is $S(n) = \frac{n(n+1)}{2}$. Then $S(n+1) = S(n)+(n+1)$. By induction hypothesis $S(n) = \frac{n(n+1)}{2}$. So $S(n+1) = \frac{n(n+1)}{2} + (n + 1) = \frac{(n+1)(n+2)}{2}$, which is exactly what we wanted to prove. ■

Theorem 2.3 *If n is a natural number and $1 + x \geq 0$ then $(1 + x)^n \geq 1 + nx$.*

Proof: The proof is by induction on n. For $n = 1$ both sides are equal to $1+x$. By induction hypothesis $(1+x)^n \geq 1+nx$. Then $(1+x)^{n+1} = (1+x)^n(1+x)$. Now induction hypothesis implies $(1+x)^n \geq 1+nx$. Also $1+x \geq 0$. Hence $(1+x)^{n+1} = (1+x)^n(1+x) \geq (1+nx)(1+x) = 1+(n+1)x+nx^2 \geq 1+(n+1)x$ as $nx^2 \geq 0$. ■

2.2 More deeper examples

Now we look at more involved examples of induction.

Coloring

Definition 2.1 *Consider n distinct lines in the plane not necessarily in general position. Two regions are called as **neighboring** if and only if they have an edge in common. An assignment of colors to the regions formed by these lines is called as a **valid coloring** if neighboring regions have different colors.*

Theorem 2.4 *It is possible to have a valid coloring of any number of lines in the plane with only two colors.*

Proof: For $n = 1$ it is trivial. Assume it is correct for $n-1$. Consider the n^{th} line. The only question is how to modify the coloring when the n^{th} line is added. Consider the following algorithm:

1. Divide the regions into two groups according to which side of the n^{th} line they lie.
2. Leave all regions on one side colored the same as before, and reverse the colors of all regions on the other side.

To prove that the algorithm works consider two neighboring regions R_1 and R_2 and the edge common between them. If both are on the same side of the n^{th} line, then by the induction hypothesis they were colored differently before the n^{th} line was added and so even if both colors get reversed still they will remain different. If the edge between them is part of the n^{th} line, then they belong to the same region before the n^{th} line was added. Since the color of one region was reversed, they are now colored differently. ■

Converting a number to its binary representation

Induction is a very good tool for proving the correctness of algorithms. Assume a program has a loop that is supposed to compute a certain value. We can use induction on the number of times this loop is executed to prove that the result is correct. An induction hypothesis which reflects the relationships between the variables during the loop execution is called a **loop invariant**.

Theorem 2.5 *When Algorithm 2.1 terminates it stores the binary representation of n in the array b.*

Algorithm 2.1 Convert-To-Binary(n)

Input: A positive integer n.
Output: An array of bits b corresponding to the binary representation of n.

```
1: t=n (a new variable to preserve n)
2: k=0
3: while t > 0 do
4:    b[k] = t mod 2
5:    k = k + 1
6:    t = ⌊t/2⌋
7: end while
```

Proof: The proof is by induction on k, the number of time the loop is executed. In this case the induction hypothesis is the loop invariant. The Induction Hypothesis is "If m is the integer represented by the binary array $b[0, 2, \ldots, k]$, then $n = 2^k t + m$". Intuitively it says that at step k of the loop, the binary array represents the k least significant bits of n, and that the value of t, when shifted by k corresponds to the rest of the bits.

To prove the correctness of the algorithm we need to show

1. The hypothesis is true at the beginning (basis).
2. The truth of the hypothesis at step k implies the truth of the hypothesis at step $k + 1$.
3. When the loop terminates the induction hypothesis implies the correctness of the algorithm.

The first point is easy to show: $k = 0 = m$ (since the array is empty by definition) and $t = n$. Thus $n = 2^0 t + 0$. For the third point we note that $t = 0$ and thus $n = 2^k 0 + m = m$. Finally for the second point we consider two cases for the start of the k^{th} loop:

- If t is even, then $t \bmod 2 = 0$ and thus there is no change to the array, t gets divided by 2 and k is incremented, i.e., $n = \frac{t}{2} 2^{k+1} + m = 2^k t + m$.
- If t is odd, then $b[k+1]$ is set to 1, which contributes 2^k to m, t is changed to $\frac{t-1}{2}$ and k is incremented. So the expression is $\frac{t-1}{2} 2^{k+1} + m + 2^k = (t-1)2^k + m + 2^k = 2^k t + m = n$ as desired.

■

Read about the **common errors** is Section 2.13 of the book [2] to avoid them.

Finding one-to-one mappings

So far we have seen the use of induction in the proof of theorems and showing correctness of algorithms. In a sense, the induction idea gives us **recursive** algorithms, and we can leverage this idea in order to design efficient algorithms.

Let f be a function that maps a finite set $A = \{1, 2, \ldots, n\}$ to itself. Assume f is represented by an array $f[12 \ldots n]$ such that $f[i]$ holds the value of $f(i)$ which is an integer between 1 and n. We call f a **one-to-one** function if for every element j, there is at most one element i that is mapped to j. Consider an example of a function in Figure 1.

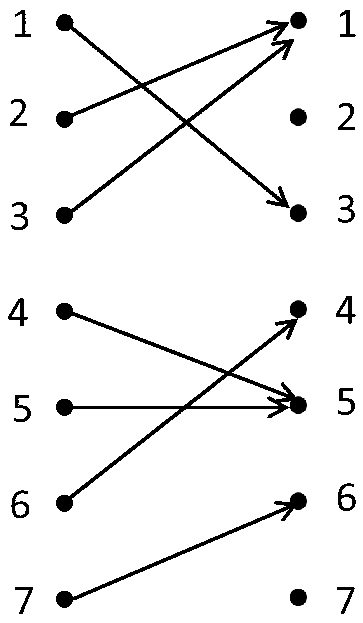

Figure 1:

The Problem: Given a finite set A and a mapping $f : A \to A$ find a subset $S \subseteq A$ with maximum number of elements such that

1. The function f maps every element of S to some other element within S itself.
2. No two elements of S are mapped to the same element, i.e., f is one-to-one when restricted to S.

Algorithm: If f is originally one-to-one then taking $S = A$ we are done. If on the other hand $f(i) = f(j)$ for some $i \neq j$ then S cannot contain both i and j. We can try all subsets $S \subseteq A$, but the running time is $2^n n$ which is exponential. We want a more efficient algorithm. For example, in Figure 1 we have $f(2) = 1 = f(3)$. So S cannot contain both 2 and 3. The choice between 2 and 3 is important: if we eliminate 3, then 1 is eliminated and then 2 is eliminated. However if we instead eliminate 2 only then we obtain $\{1, 3\}$ as a good set.

So reducing the problem to a smaller one is important and non-trivial. We can use induction hypothesis: We know how to solve the problem for sets with $n-1$ elements. The base case is trivial: if there is only one element in the set then it must be mapped to itself. But the induction hypothesis is non-trivial when we have a choice (e.g. 2 and 3 in Figure 1). The key idea is that the element i that has no element mapped to it cannot belong to S and thus we must remove it. This is because $i \in S$ and $|S| = k$ implies that the k elements of S must be mapped to at most $k-1$ elements of S and hence f cannot be one-to-one when restricted to S. So we remove $\{i\}$ from A and iterate. Note that the reduced problem is exactly the same (except the size) as the original problem. We have to be careful though: the only condition we had previously was that f maps A to itself. This condition is still maintained for the set $A \setminus \{i\}$. By this algorithm we need to consider only n elements instead of 2^n choices for S. Thus this algorithm is much more efficient.

3 Analyzing Algorithms

We care about the efficiency of algorithms. A good measure is running time which can predict the behavior of an algorithm without implementing it on a specific computer. Again we usually consider running times of the small parts of programs which are the **heart** of the programs and not everything (like number of multiplications, additions, etc). We define certain parameters and certain measures that are the most important for the analysis (an approximation of the real world).

The main feature (as you have seen in previous classes) is to ignore the constant factors and concentrate on the behavior of the algorithm when the **size of the input** goes to infinity. The size is usually defined as a measure of amount of space required to store the input and usually will be denoted by n.

Given a problem and a definition of size, we want to find an expression that gives the running time of the algorithm relative to the size usually in the **worst case**. The best case is usually ruled out since it is not representative. The average case might be a good choice but is usually very hard to measure (even the definition of average is not clear when we have different parameters, also mathematically difficult to analyze). Also it is hard to predict what will happen in practice, and thus considering worst-case scenario is safer. We will analyze some algorithms in the remaining part of this section.

Insertion Sort

Insertion Sort is a sorting algorithm using induction. Suppose we know how to sort $n-1$ elements and we are given n numbers to sort. We can start with $n-1$ numbers and then put the n^{th} number in its correct place by scanning the $n-1$ sorted numbers until the correct place to insert is found (see Algorithm 3.1).

The total number of comparisons for sorting n numbers may be as high as $1+2+\ldots+(n-1) = \frac{(n-1)n}{2} = O(n^2)$. Also for inserting, and thus moving, in

the worst case we need $n-1$ elements to be moved and hence the total number of movements is also $O(n^2)$. For more details about the insertion sort, look at `http://en.wikipedia.org/wiki/Insertion_sort`

Algorithm 3.1 Insertion Sort

Input: Array $A[1, 2, \cdots, n]$.
Output: Array A in an increasing order.

```
1: for i = 2 to n do
2:    key = A[i]
3:    j = i
4:    while (j > 1 and A[j − 1] > key) do
5:       A[j] = A[j − 1]
6:       j = j − 1
7:    end while
8:    A[j] = key
9: end for
```

Selection Sort

We can select the minimum number and put it at the first of the array by swapping it with the other elements (See Algorithm 3.2). We recursively sort the rest of the elements. The advantage over insertion sort is that only $n-1$ data movements (swaps) are required versus $O(n^2)$ needed for insertion sort. However it takes $n-1$ comparisons to find the maximum element and so selection sort needs $O(n^2)$ comparisons. For more details about the selection sort, look at `http://en.wikipedia.org/wiki/Selection_sort`

Algorithm 3.2 Selection Sort

Input: Array $A[1, 2, \cdots, n]$.
Output: Array A in an increasing order.

```
1: for i = 1 to n − 1 do
2:    minIndex = i
3:    for j = i + 1 to n do
4:       if A[j] < A[minIndex] then
5:          minIndex = j
6:       end if
7:    end for
8:    swap(A[i], A[minIndex])
9: end for
```

Binary Search

The Problem: Let $A[1], A[2], \ldots, A[n]$ be an array of real numbers such that $A[1] \leq A[2] \leq \ldots A[n]$. Given a real number x, we want to find whether x

appears in the array and if it does, then find an index i such that $A[i] = x$.

Approach: We compare x with $A[\frac{n}{2}]$. If $x < A[\frac{n}{2}]$ then x is clearly in the first half of the array, and if $x > A[\frac{n}{2}]$ then x is in the second half of the array. Otherwise $x = A[\frac{n}{2}]$ and we are done. Finding x in either half is a problem of size $\frac{n}{2}$, which can be solved by induction. We handle the base case of $n = 1$ by directly comparing x to the element (see Algorithm 3.3).

Time Complexity: Time Complexity: Let $T(n)$ denote the time complexity for sorting n numbers. Then the recurrence is

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

Therefore, $T(n) = \Theta(\log n)$. For more details about the binary search algorithm, look at `http://en.wikipedia.org/wiki/Binary_search_algorithmt`

Algorithm 3.3 Binary Search

Input: Array $A[1, 2, \cdots, n]$ in a non-decreasing order, and a number x.
Output: Find an index i such that $A[i] = x$, and return NOT FOUND if there is no such an index.

```
1:  first = 1
2:  last = n
3:  while first ≤ last do
4:     mid = ⌊(first+last)/2⌋
5:     if A[mid] < x then
6:        first = mid + 1
7:     else if A[mid] > x then
8:        last = mid − 1
9:     else
10:       return mid
11:    end if
12: end while
13: return NOT FOUND
```

Merge Sort

Merging Operation: Denote the first set by $a_1, a_2, \ldots, a_n$ and the second set by $b_1, b_2, \ldots, b_m$. We assume both are sorted in increasing order. Scan the first set until the right place to insert b_1 is found and insert it. Then continue the scan from that place until the right place to insert b_2 is found, and so on. Since the $b_i's$ are sorted we never need to go back. The total number of movements is $O(n + m)$.

Data movement is inefficient if we insert the $b_i's$, however if we use a temporary array where each element is copied exactly once then the overall time is $O(n + m)$.

Merge Sort is a divide-and-conquer (recursive) algorithm as follows:

1. Divide the sequence into parts of close-to-equal size.
2. Sort each part separately recursively.
3. Merge the two parts into one sorted array.

Time Complexity: Let $T(n)$ denote the time complexity for sorting n numbers. Then the recurrence is

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \; ; \; T(1) = 1$$

It can be prove that $T(n) = \Theta(n \log n)$ which is much better than insertion or selection sorts. However it requires additional storage to copy the merged sets and is not an in-place sorting algorithm. For more details, look at `http://en.wikipedia.org/wiki/Merge_sort`

4 The Big-O Notation

Definition 4.1 *A function $g(n)$ is $O(f(n))$ for another function $f(n)$, if there exist constants c and N such that for all $n \geq N$, we have $g(n) \leq c.f(n)$*

In other words, for large enough n, the function $g(n)$ is no more than a constant times the function $f(n)$. The O-notation is like $\leq$ only from the above, though we usually try to find the tightest bound.

Example: $5n^2 + 15 = O(n^2)$ since $5n^2 + 15 \leq 16n^2$ for $n \geq 4$. Also it is $O(n^3)$ since $5n^2 + 15 \leq n^3$ for all $n \geq 6$.

We always write $O(n)$ instead of $O(5n + 4)$. We also write $O(\log n)$ without specifying the base of the logarithm. So $O(1)$ means constant, $O(n)$ is linear, $O(n^2)$ is quadratic and so on. Though in general showing $g(n) = O(f(n))$ is not easy, it will be easy for almost all algorithms in this class.

4.1 Polynomial vs Exponential

Theorem 4.1 *$n^c = O(a^n)$ for all constants $c > 0$ and $a > 1$.*

Note that $(\log_a m)^c = O(a^{\log_a m}) = O(m)$. We call $(\log_a m)^c$ as "polylog" in m. So we prefer polynomials over exponentials (like 2^n) and polylog over polynomials.

4.2 Rules

We can add and multiply using the following rules (but we cannot subtract or divide). Proofs follow from the definition and are given in the book [2].

1. If $f(n) = O(s(n))$ then $c.f(n) = O(s(n))$ for any constant $c > 0$.
2. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n) + g(n) = O(s(n) + r(n))$.
3. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n).g(n) = O(s(n).r(n))$.

4.3 Other Notation

We often try to use (tight) upper bounds on running times of algorithms which means there is an algorithm with at most this running time. However often we need to say that there is no algorithm that can achieve a better running time. Of course this is much harder since we should model **every** algorithm. The notation for lower bounds is Ω.

Definition 4.2 *A function $T(n)$ is $\Omega(g(n))$ for another function $g(n)$, if there exist constants c and N such that for all $n \geq N$, we have $T(n) \geq c.g(n)$.*

Examples are $n^2 = \Omega(n^2 - 100)$ and $n = \Omega(n^{0.9})$. The O-notation corresponds to $\leq$ and the Ω-notation corresponds to $\geq$. What about $=$? We have the following notation:

Definition 4.3 *A function $f(n)$ is $\theta(g(n))$ for another function $f(n)$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Example is $5n \log_2 n - 10 = \theta(n \log_2 n)$. The constants in O and Ω in the above definition need not be the same. Also note that if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$.

What about strict inequalities, i.e., $<$ and $>$ instead of $\leq$ and $\geq$?

Definition 4.4 *We say $f(n) = o(g(n))$, i.e., $f(n)$ is little-oh of $g(n)$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Similarly we say that $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.*

For example, $\frac{n}{\log n} = o(n)$ but $\frac{n}{10} \neq o(n)$.

Theorem 4.2 *$n^c = o(a^n)$ for all constants $c > 0$ and $a > 1$.*

Corollary 4.1 $(\log_a n)^c = o(n)$

Note that we usually ignore the constants in O-notation but sometimes in practice even the constant matters, e.g., in Google.

4.4 Time and Space Complexity

We analyze an algorithm by counting the number of major steps the algorithm performs. For example, in sorting we count the number of comparisons. In the celebrity problem, we counted the number of questions asked. In a sense we say since we ignore the constant factors, all other operations are only a constant factor of the number of major steps. In this case we say **time complexity** or the **running time** of the algorithm is in $O(f(n))$.

The **space complexity** of an algorithms indicates the amount of temporary storage (not including the input and output) for running the algorithm. An $O(n)$ space algorithm requires a constant amount of memory per input primitive. An $O(1)$ space algorithm needs a constant amount of memory irrespective of the

size of the input. Given the current amount of storage in disks that we have, we focus mainly on running time and not space complexity (although it is important for Google).

4.5 Some Mathematical techniques for Computing Running Times

Now we consider some mathematical techniques such as recursion, summation, etc. for computing the running time of algorithms. The proofs are often by induction.

Theorem 4.3 $S_1(n) = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$.

Proof: We have seen the proof of this theorem in Lecture 1. ■

Theorem 4.4 $S_2(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

Proof: Proof is by induction and is given in the book [2]. There is also another way to prove this theorem without using induction. ■

Theorem 4.5 $F(n) = \sum_{i=0}^{n} 2^i = 1 + 2 + \ldots + 2^n = 2^{n+1} - 1$.

Proof: There is a simple proof by induction. Alternatively we can try to compare $F(n)$ with another expression involving $F(n)$. Note that $2F(n) = 2 + 4 + 8 + \ldots + 2^{n+1}$. Take the difference between $2F(n)$ and $F(n)$ to obtain $F(n) = 2^{n+1} - 1$. ■

Theorem 4.6 $G(n) = \sum_{i=1}^{n}(i \times 2^i) = (1 \times 2^1) + (2 \times 2^2) + \ldots + (n \times 2^n) = (n-1)2^{n+1} + 2$.

Proof: There is a simple proof by induction. Alternatively we can try to compare $G(n)$ with another expression involving $G(n)$. Note that $2G(n) = (1\times 2^2)+(2\times 2^3)+(3\times 2^4)+\ldots+(n\times 2^{n+1})$. Take the difference between $2G(n)$ and $G(n)$ to obtain $G(n) = n2^{n+1} - \left(2^1+2^2+\ldots+2^n\right) = n2^{n+1} - (F(n)-1) = n2^{n+1} - (2^{n+1} - 2) = (n-1)2^{n+1} + 2$. ■

4.6 Further Reading

It is also good to know the following facts:

1. Arithmetic Series:
   - $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$
   - If $a_n = a_{n-1} + c$ for some constant c, then $a_1 + a_2 + \ldots + a_n = \frac{n(a_1+a_n)}{2}$
2. Geometric Series:
   - $1 + 2 + \ldots + 2^n = 2^{n+1} - 1$

- If $a_n = c.a_{n-1}$ for some constant $c \neq 1$, then $a_1 + a_2 + \ldots + a_n = a_1 \frac{c^n - 1}{c-1}$
- If $a_n = c.a_{n-1}$ for some constant $1 > c > 0$, then $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c}$

3. Sum of Squares: $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$
4. Harmonic Series: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \ln(n) + \gamma + O(\frac{1}{n})$ where $\gamma = 0.577\ldots$ is the Euler's constant.
5. Logarithm Formulas:
   - $\log_b a = \frac{1}{\log_a b}$
   - $\log_a x = \frac{\log_b x}{\log_b a}$
   - $b^{\log_b x} = x$
   - $b^{\log_a x} = x^{\log_a b}$
6. Sum of Logarithms: $\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 = \theta(n \log n)$
7. Summation by Integral: For a monotone increasing continuous function f, we have $\sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)\,\mathrm{d}x$
8. Stirling's Approximation: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O(\frac{1}{n})\right)$. Thus $\log_2 n! = \theta(n \log n)$

5 Recurrence Relations

A **recurrence relation** is a way to define a function by an expression involving the same function, i.e., defining the function inductively.

The most famous example of a recurrence relation are the Fibonacci numbers given by $F(n) = F(n-1)+F(n-2), F(0) = 1, F(1) = 1$. We should have enough information to obtain $F(n)$ inductively, i.e., the base case and the rest of the inductive case.

Note that in the above example, to obtain $F(n)$ we need to compute all the values $F(1), F(2), \ldots, F(n-1)$ which takes $O(n)$ time. It would be much easier if we have an explicit (or closed-form) expression for $F(n)$. This is called as solving the recurrence relation, e.g., for the n^{th} Fibonacci we can get the following explicit formula:

$$F(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}} \tag{1}$$

where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180$ (golden ratio) and $\psi = \frac{1-\sqrt{5}}{2} = -0.618033$.

5.1 Solving Recurrence Relations

Recurrence relations are used quite often in the analysis of algorithms and hence we learn how to solve them.

Guessing and playing with the recurrence relation works well for wide class of recurrence relations. Of course this needs some experience and practice of working with recurrence relations. Expansion and tree-recursion methods are another techniques which are used quite often. Let us now see some examples:

Expansion Method:

1. $T(n) = rT(n-1)$ and $T(0) = k$
   Then we have $T(1) = rk, T(2) = r^2k$ and in general $T(n) = r^nk$.
2. $T(n) = T(n-1) + n$ and $T(0) = k$
   Then

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= \ldots \\ &= T(0) + 1 + 2 + \ldots + n \\ &= k + \frac{n(n+1)}{2} \end{aligned}$$

   This is an example of proof by **expansion**.

Substitution Method:

1. $T(n) = AT(n-1) + BT(n-2)$; $T(0) = k_0$ and $T(1) = k_1$
   Let us guess $T(n) = r^n$ and substitute it in the formula. This gives $r^n = Ar^{n-1} + Br^{n-2} \Rightarrow r^2 = Ar + B$. We solve this to get two roots r_1 and r_2. Now it is easy to see that Cr_1^n and Dr_2^n and more generally $Cr_1^n + Dr_2^n$ (linear combination) if $r_1 \neq r_2$, and $Cr_1^n + nDr_2^n$ if $r_1 = r_2$ are also the solutions and indeed are the most general solutions. The constants C and D are chosen based on the two given initial values $T(0) = k_0$ and $T(1) = k_1$, i.e., $C + D = k_0$ and $Cr_1 + Dr_2 = k_1$ (solve the exact equations to obtain C and D). For the Fibonacci numbers, if we solve $C + D = 1$ and $Cr_1 + Dr_2 = 1$ then we obtain the explicit formula given in Equation 1. The equation $r^2 = Ar + B$ is called as the **characteristic** equation and this same technique can also be used for $T(n) = A_1T(n-1) + A_2T(n-2) + \ldots + A_kT(n-k)$. Again we note that proof would be by induction. See Wikipedia for more formulae.
2. $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$ and $T(0) = 0$
   Let us guess $T(n) = O(n \log n)$ and substitute it in the formula. Now we can come back and prove inductively that $T(n) \leq cn \log n$ for some constant $c > 0$. The base case is $0 = T(1) \leq c \times 1 = c$ which holds for all $c \geq 0$. Induction Hypothesis is $T(\lfloor \frac{n}{2} \rfloor) \leq c\lfloor \frac{n}{2} \rfloor \log \lfloor \frac{n}{2} \rfloor$, then

we have $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n \leq 2c\lfloor \frac{n}{2} \rfloor \log \lfloor \frac{n}{2} \rfloor + n \leq 2c\frac{n}{2} \log \frac{n}{2} + n \leq cn(\log n - 1) + n = cn \log n - cn + n \leq cn \log n$ for all $c \geq 1$. So the induction works for any $c \geq 1$.

Indeed the formula above is the number of comparisons in a procedure for sorting n numbers, called the Mergesort: We divide the sequence into two parts of (almost) equal size $(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil)$. We sort them and then merge them with n more comparisons. Thus we have the recurrence given above. Look at Chapter 4.3 of [1] for more details.

Recursion-tree Method:

1. $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ and $T(1) = 0$. Look at Section 4.4 of [1] for more details.
2. $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n)$ and $T(1) = 0$. Look at Section 4.4 of [1] for more details.

5.2 Master Theorem

In divide-and-conquer algorithms (like Mergesort), the problem is divided into smaller subproblems, each subproblem is solved recursively and a combined algorithm is used to obtain the final solution from the solutions of the subproblems. Assume there are a subproblems, each of size $\frac{1}{b}$ of the original problem and the algorithm to combine the subproblems takes polynomial time given by cn^k for some constants a, b, c & k. Then we have

$$T(n) = aT(\frac{n}{b}) + cn^k \tag{2}$$

By an expanding method very similar to the one in the last lecture for $a_n = 2a_{\lfloor \frac{n}{2} \rfloor} + n$, we can prove the following theorem: (it is a good exercise to prove it for yourself or see the proof in the book [1]).

Theorem 5.1 *If $T(n) = aT(\frac{n}{b}) + cn^k$ for integers $a \geq 1, b \geq 2$ and positive constants c and k, then*

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

This formula is very useful in many divide-and-conquer algorithms and you should **memorize** it in this course.

A more general theorem which consider a general function $f(n)$ instead of cn^k in Equation 2 and obtains θ instead of O, is called the **Master Theorem**. See the Wikipedia article on the Master Theorem. Again there are three cases to consider conditioned on the function $f(n)$.

An even more general theorem which gives one formula instead of the three cases above is called **Akra-Bazzi** theorem. In our class we always approximate

$f(n)$ with appropriate cn^k and use Theorem 5.1 instead and we obtain θ usually (and not just O).

6 Elementary Data Structures

Data structures are the building blocks of computer algorithms. Here we consider abstract data type (ADT), i.e., we do not specify the data types like integers, reals, characters, etc. It is more convenient and more general to design algorithms for these operations without specifying the data type of the items.

Modern object-oriented languages such as C++ or Java support a form of abstract data. When a class is used as a type, it is an abstract type that refers to hidden representation. An ADT is typically implemented as a class and each instance of the ADT is an object of that class.

6.1 Arrays

An array is a row of elements of the same type. The size of an array is the number of elements in that array. The size of the array should be fixed and allocated in memory in advance. The elements can be bytes, integers, strings or more generally any elements. Arrays are very efficient and very common data structures and each element of it can be accessed in constant time. Arrays are good except they store elements of the same type and their size is fixed. We represent them as $a[12 \ldots n]$ and access element i as $a[i]$ for $1 \leq i \leq n$.

6.2 Linked Lists

There are many applications in which the number of elements is changing dynamically as the algorithm progresses (we can have large arrays to overcome this, but it is often not a good solution. Also due to consecutive representation of arrays doing some operations like insertion and deletion in the middle of an array is very inefficient). Linked lists are the simplest form of **dynamic** data structures. Here each element is represented separately and all elements are connected through the use of **pointers**. A pointer is simply a variable that holds as its value the address of another element. It exists in C, C++ but not explicitly in Java (although it can be simulated).

A linked list is a list of pairs, say in a record, each containing an element and a pointer, such that each pointer contains the address of the next pair. It is not possible to access each element directly in a linked list, instead one needs to scan it by following the addresses in the pointer. This is called as a linear scan. The end of the list will be denoted by null, a pointer which points to nowhere. An example is given in Figure 2.

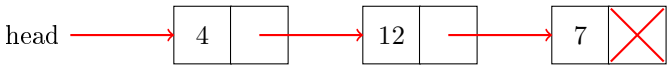

Figure 2: Example of a linked list

DRAWBACKS OF LINKED LISTS

1. Requires more space: one additional pointer per element (not a huge problem though).
2. Need a linear scan to find say the 30th element.

ADVANTAGES OF LINKED LISTS

1. With only 2 operations we can insert an element in the middle.
2. With only 1 change we can delete an element.

Doubly Linked List

In a doubly linked list, each record has a pointer to the previous record as well as a pointer to the next record (see Figure 3). Look at Section 10.2 of [1] for more details about doubly linked list.

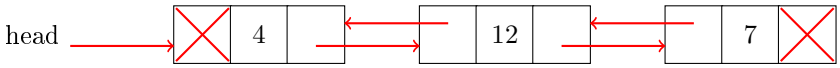

Figure 3: Example of a doubly linked list

6.3 Stacks

Stacks are dynamic data structures in which we delete (**pop**) an element which was most recently inserted (**push**). A stack implements a last-in-first-out (LIFO) policy. We have also an operation **top** which gives the element at the top of the stack. We can implement it with both arrays (if we know the maximum size) or linked lists. We now show how to implement the Push, Top and Pop functions using arrays. As an exercise, you can implement them with linked lists. Each of Push, Pop and Top operations takes $O(1)$ time.

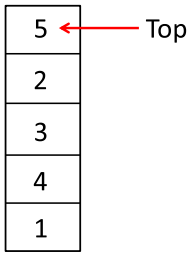

Figure 4: Example of a stack

Algorithm 6.1 POP(S)

```
1: if top==0 then
2:    return error "stack underflow"
3: else
4:    top ← top − 1
5:    return S[top + 1]
6: end if
```

Algorithm 6.2 PUSH(S, x)

```
1: top ← top + 1
2: S[top] ← x
```

Algorithm 6.3 TOP(S)

```
1: if top==0 then
2:    return error "stack underflow"
3: else
4:    return S[top]
5: end if
```

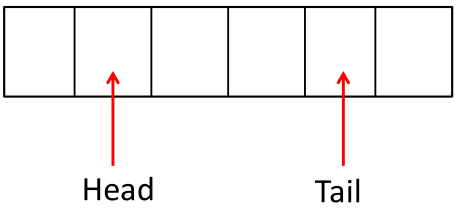

Figure 5: Example of a queue

6.4 Queues

A queue is a dynamic data structure in which the element deleted (**dequeue**) is always the one that has been inserted (**enqueue**) for the longest time. A queue implements a first-in-first-out (FIFO) policy like in the registrar's office. The queue has a head and a tail. We insert at the tail and delete from the head. It can be implemented by array or linked lists (like stacks). We now show how to implement the Enqueue and Dequeue functions using arrays as if the queue is circular. As an exercise, you can implement them with linked lists. Each of Enqueue and Dequeue operations takes $O(1)$ time.

Algorithm 6.4 ENQUEUE(Q, x)

```
1: Q[tail] ← x
2: if tail == Q.length then
3:    tail ← 1
4: else
5:    tail ← tail +1
6: end if
```

Algorithm 6.5 DEQUEUE(Q, x)

```
1: x ← Q[head]
2: if head = Q.length then
3:    head ← 1
4: else
5:    head ← head +1
6: end if
7: return x
```

6.5 Rooted Trees

Each node of a tree can be represented by an object with a key attribute and pointers to other nodes. In a **binary tree** each node has at most two children, and thus we can implement a binary tree by the following pointers:

- *left:* a pointer to the left child.
- *right:* a pointer to the right child.
- *parent:* a pointer to the parent. This pointer is optional and one may not use this pointer in his/her implementation.

The above data structure is useful for implementing a tree with a bounded number of children. If we want to design a data structure for storing a tree with unbounded number of children, then one solution is to store the following pointers:

- *left-child:* a pointer to the leftmost child.
- *right-sibling:* a pointer to the sibling immediately to its right..
- *parent:* a pointer to the parent. This pointer is optional and one may not use this pointer in his/her implementation.

Look at Section 10.4 of [1] for more details.

6.5.1 Tree Traversal

Tree traversal is the process of visiting each node in a tree exactly once. We describe three tree traversals for binary tree as follows:

- **Preorder:** Visit root, traverse the left subtree, and traverse the right subtree (see Algorithm 6.6).
- **Inorder:** Traverse the left subtree, visit root, and traverse the right subtree (see Algorithm 6.7).
- **Postorder:** Traverse the left subtree, traverse the right subtree, and visit root (see Algorithm 6.8).

Algorithm 6.6 PREORDER-TRAVERSAL(x)

```
1: if x ≠ null then
2:    print x.value
3:    PREORDER-TRAVERSAL(x.left)
4:    PREORDER-TRAVERSAL(x.right)
5: end if
```

Algorithm 6.7 INORDER-TRAVERSAL(x)

```
1: if x ≠ null then
2:    INORDER-TRAVERSAL(x.left)
3:    print x.value
4:    INORDER-TRAVERSAL(x.right)
5: end if
```

7 Binary Search Trees

We see **implicit** representations of trees for heaps. For explicit representation all nodes have $m+1$ pointers, one to the parent and m to the children, where m is the maximal number of children in a tree. Alternatively, if m is unknown, we have two pointers: one to the first child and second to the next sibling (a linked list or doubly-linked list). Binary search trees is another way to implement **dictionary operations** without randomization and it is comparison-based (the bounds are the worst-case bounds). The dictionary operations are as follows:

1. Search(x): Find x or determine x is not there (assume all elements are distinct for now).
2. Insert(x): Insert x unless it is already there.
3. Delete(x): Delete x if it is there.

Binary search trees also implement more complicated operations efficiently. We use explicit representation for binary trees by keeping a record containing at least three fields: key, left and right (child). Due to explicit representation, any node may be added or removed unlike heaps where we only remove or insert leaves. Each key in the binary search tree serves to divide the range of the keys below. The keys in the left subtree are all smaller than it and the keys in the right subtree are all greater than it. We say a tree is **consistent** if all keys satisfy this condition.

Search Operation: We first compare x against the root of the tree, whose value is say r. If $x = r$, then we are done. If $x < r$, we continue from the left child. Otherwise we continue from the right child.

Insertion Operation: First we search for x and abort insert if x is already in the tree. Otherwise, the search ends at a leaf and the new key can then be inserted below that leaf depending on the value of x (at left or right). Note that the tree remains consistent.

Algorithm 6.8 POSTORDER-TRAVERSAL(x)

```
1: if x ≠ null then
2:    POSTORDER-TRAVERSAL(x.left)
3:    POSTORDER-TRAVERSAL(x.right)
4:    print x.value
5: end if
```

Deletion Operation: Deletion is more complicated. It is easy if it is a leaf (by making it NULL). Even if the node has one child, it is easy by changing the child pointer of the parent to the child of x (instead of x). What about if the node x has two children? First we find the predecessor of x in the tree which is a node at least as large as all the keys in the left subtree of x and smaller than all the keys in the right subtree of x. To find it first we go to the left child of x and then follow the right child as long as we can. When we end up by finding a vertex say p, since it has only at most left child, we will delete it easily and replace x with p. The consistency is preserved since p is the largest element of the left subtree of x, i.e., the tree rooted at the left child of x.

Time Complexity: All running times depend on the shape of the tree and the location of the relevant node; in the worst case it is $O(\text{height}(T))$ where the height (or depth) of a tree is defined as the maximum length of a path from its root to any leaf. Thus a rooted tree with only one vertex (the root) has height 0 and the height of an empty tree is defined to be -1. If the tree is balanced, i.e., $\text{height}(T) = O(\log n)$, where n is the number of elements, all operations are in $O(\log n)$ and thus efficient. If the keys are inserted in a random order, then one can probe that the expected height of the tree is $O(\log n)$ (like the depth of quicksort) and thus dictionary operations are efficient in the average case. However if the insertions are in a sorted, or closed to sorted, order, then the tree would be a linked list (a long path) and the operations can take $\Omega(n)$ time. To prevent this worst case we can use **red-black** trees or **AVL** trees. Look at Chapter 13 of [1] and Section 4.3.4 of [2] for more details about red-black trees and AVL trees.

8 Hashing

A hash table is a generalization of the simpler notion of an ordinary array. Many applications require a dynamic set that supports only the **dictionary operations**: Insert, Search and Delete. A hash table is an effective data structure for implementing dictionaries. If an application needs dictionary operations with keys drawn from the universe $\mathbb{U} = \{0, 1, \ldots, u-1\}$ where u is not too large, then we can have direct-address table method by using an array $T[0, 1, \ldots, u-1]$. What about when the universe $\mathbb{U}$ is so large that storing a table T of size $|\mathbb{U}|$ may be impractical and in addition the set K of keys actually stored may be so small relative to $\mathbb{U}$ (e.g. keys in a dictionary). With hashing, we can reduce space to $\theta(|K|)$ with search time only $O(1)$ (though in average). With direct

addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, a **hash function** is used to compute the slot from the key k. Here h maps the universe $\mathbb{U}$ of keys into the slots of a hash table $T[0, 1, \ldots, m-1]$, i.e., $h : \mathbb{U} \to \{0, 1, \ldots, m-1\}$. We say that an element with key k hashes to slot $h(k)$ or $h(k)$ is the hash value of key k.

The main issue is that two keys may hash to the same slot. This is called as a collision for which we see effective techniques in this lecture. One idea is to make h appear to be "random" to avoid collisions.

8.1 Hashing Functions

The average performance of hashing depends on how well it distributes the keys in the m slots, i.e., how "random" it is to avoid collisions. We consider integers or characters, fixed length arrays (of integers) and variable length arrays (strings):

1. **Hashing integers:** The original proposal of Cater and Wegman was to pick a prime $p \geq u$ and define

$$h_{a,b}(x) = \Big((ax + b) \mod p\Big) \mod m$$

where a, b are randomly chosen integers mod p and $a \neq 0$.

2. **Hashing fixed-length array:** The input is a vector $\overline{x} = \{x_0, x_1, \ldots, x_{h-1}\}$ of h integers (or bytes/characters) and we define

$$h(\overline{x}) = \sum_{i=0}^{h-1} h^i_{a,b}(x_i)$$

where each $h^i_{a,b}$ is constructed by choosing a, b randomly mod p with $a \neq 0$. Here again we choose p to be a prime greater than u.

3. **Hashing strings:** If the length of a string can be bounded by a small number it is best to use a fixed-length array solution. Now we assume that we want to hash $\overline{x} = \{x_0, x_1, \ldots, x_L\}$ where a bound on L is not known a priori. Again if $x_i \in \{0, 1, \ldots, u-1\}$ then let $p \geq \max\{u, m\}$ be a prime and define

$$h_c(\overline{x}) = h_{a,b}\Big((\sum_{i=0}^{L} x_i c^i) \mod p\Big)$$

where $a, b\&c$ are randomly chosen integers mod p with $a, c, \neq 0$.

Definition 8.1 *A hash function is called* ***universal*** *if* $\forall\ x, y \in \mathbb{U}, x \neq y$ *we have* $Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Note that this is the ideal case in which any two keys of the universe collide with probability at most $\frac{1}{m}$. All functions above provide such a guarantee. Note that without randomness we cannot guarantee such functions. Universal hashing guarantees any given element is equally likely to hash into any slot.

8.2 Collision Resolution

However if we use any approach then there is a chance of collisions. In this part we describe two methods for handling collisions.

8.2.1 Collision Resolution by Chaining

In chaining we put all elements that hash to the same slot in a linked list. The slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements then slot j contains NIL. The dictionary operations are as follows:

- Hash-Insert(T, X)
  Insert X at the head of the list $T[h(\text{key}[X])]$
- Hash-Search(T, k)
  Search for an element with key k in the list $T[h(k)]$
- Hash-Delete(T, X)
  Delete X from the list $T[h(k)]$

The worst case running time for insertion is $O(1)$ though for delete and search it is $O(|T[h(X)]|)$. Because of the uniformity property, the average length of each list is $\frac{n}{m}$ where n is the number of keys and m is the size of T. Thus running time for delete and search is in $O(1 + \frac{n}{m})$. Thus if m is at least proportional to n, then $n = O(m)$ and the running time is $O(1)$ for all operations.

8.2.2 Open Addressing: Linear Probing

In open addressing, all elements are stored in the hash table itself and it avoids pointers (and waste of memory) altogether. Thus in open addressing, the hash table can fill up so that no further insertion can be made. Instead in the case of collision we need to examine (probe) a sequence of slots.

In the simplest form of open addressing, we are using the method of linear probing as follows: given a key k, the first slot probed is $T[h(k)]$. We next probe $T[h(k)+1]$ and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T[1], \ldots$ until we finally probe slot $T[h(k)]$. Note here that the initial probe position determines the entire probe. In insertion we probe according to this order until we find an empty space or give an error of "hash table overflow". For search, we probe accordingly until we find the element or an empty slot.

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we need to mark it as "deleted" instead of NIL such that search skips over it, but Insert can insert it. Again if load factor $\alpha = \frac{n}{m}$ is constant then Insert and Search can take $O(1)$ in average. However Delete can cause problems and make operations no longer dependent in α. For this reason Chaining is more commonly selected as a collision resolution technique when keys must be deleted often. Sometimes if the number of deletions is a lot we can do re-hashing, i.e., hash in a new array.

9 Heaps

A heap is a binary tree whose keys satisfy the following property:

Heap Property
The key of every node is greater than or equal to the key of any of its children.

Heaps are useful to implement priority queues, an abstract data type defined by the following two operations:

1. Insert(x): insert a key x into the data structure.
2. Max-Remove(): remove the largest key from the data structure and return it.

Priority queues are like queues but when we dequeue the highest priority element is retrieved first.

9.1 Heaps Revisited

Heaps can be represented using Explicit or Implicit tree representation, however since we can ensure that heaps will be balanced we can assume the array is $A[1, 2, \ldots, k]$ where k is an upper bound in the max number of elements the heap will ever contain.

Heap Representation: We use an array. Consider a binary tree. The root is stored in $A[1]$, the left child of $A[1]$ is stored in $A[2]$ and the right child of $A[1]$ is stored in $A[3]$. In general inductively the left child of a node v stored in $A[i]$ is stored in $A[2i]$ and the right child is stored in $A[2i + 1]$. The advantage is that no pointers are needed which saves storage.

Remove Operation: By the heap property, the node with the largest key in a heap is the root $A[1]$. We remove it, take the leaf $x = A[n]$, delete it and put it in the place of the root, i.e., $A[1] := A[n]$ and $n := n - 1$. We have two separate heaps and x at the root. We now propagate x down the tree until it reaches a subtree for which it is a maximum. First we find the max of children and if it is larger than x, then swap it with x. Inductively continue until either x becomes max for a subtree or when it reaches a leaf. The maximum number of comparisons is thus $2\lceil \log_2 n \rceil$.

Insert Operation: It is bottom-up as compared to Remove which is top-down. We increment n by one and insert x as the new leaf $A[n]$. We then compare the new leaf with its parent, and swap if the new leaf is larger than its parent. We continue inductively (for correctness) this process, promoting the new key up the tree until the new key is not larger than its parent or it reaches the root, The maximum number of comparisons is thus $\lceil \log_2 n \rceil$.

Overall we can insert and remove in time $O(\log n)$ per operation for heaps. However heaps are not useful for some operations like search.

9.2 Heapsort

Heapsort is another fast sorting algorithm which runs in $O(n \log n)$.

First building a heap: Given an array $A[1, 2, \ldots, n]$ of elements in an arbitrary order, rearrange the elements so that the array satisfies the heap property. There are two ways: top-down and bottom-up, but we say only top-down which is simpler and more efficient. Consider scanning the array from left to right. The induction hypothesis is that the array $A[1, 2, \ldots, i]$ is a heap. The base case is trivial as $A[1]$ is a heap. The main part is to incorporate $A[i + 1]$ into the heap $A[1, 2, \ldots, i]$ which is exactly the same as inserting $A[i + 1]$ into the heap. The number of comparisons in the worst case is $\lfloor \log_2(i + 1) \rfloor$. The total number of comparisons is thus $\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = \theta(n \log n)$.

The rest of the sort: Since A is a heap, we know $A[1]$ is the max element. Thus we remove from A and put it at $A[n]$ and continue with $A[1, 2, \ldots, n-1]$. The overall running time is $\sum_{i=1}^{n} 2\lceil \log_2(n - i + 1) \rceil = \sum_{i=1}^{n} \lceil \log i \rceil = \theta(n \log n)$. So the overall running time is $O(n \log n)$.

10 Quicksort

As we have seen in the mergesort, by divide-and-conquer, we could divide the problem into two equal-sized subproblems and solve each problem separately and combine the solutions to get a $O(n \log n)$ algorithm. But we needed to have extra space as it was not an in-place sort.

Quicksort is a divide-and-conquer algorithm which spends most of the effort in the divide step and very little in the conquer step. Suppose we know a number X (called **pivot**) such that one-half of the elements are greater than or equal to X and one-half of the elements are smaller than X. We can compare all elements by n comparisons and partition the sequence into equal parts (we can do this without additional space). This is the divide step. We sort each subsequence recursively and the combine step is trivial since the two parts already occupy correct positions in the array and we do not need additional space.

Unfortunately we do not know X, but it works no matter which number is selected. However to do it in-place we use two pointers L and R, where L points to the left side of the array and R points to the right side of the array. The pointers move towards each other such that:

Induction Hypothesis:
At step k of the algorithm, $pivot \geq A[i]$ for all $i < L$ and $pivot < A[j]$ for all $j > R$.

The base case is trivial and when $L > R$ we are done. Two cases to consider: if either $A[L] \leq pivot$ or $A[R] > pivot$ we can move one of them and induction hypothesis is preserved. Otherwise $A[L] > pivot$ and $A[R] \leq pivot$. In this case we do a swap and move both of them. The whole partition needs $O(n)$ comparisons.

Divide-and-conquer algorithms work best when the parts have equal sizes, i.e., the pivot should be the median. We can find it also but not in this class. However choosing a random element from the sequence is also a good choice as we see in the analysis. At the end of the partition we put the pivot in its correct place. You can find another procedure for partitioning in Section 7.1 of [1]. Once we have the procedure PARTITION, the Quicksort algorithm is as follows:

Algorithm 10.1 QUICK-SORT(A, L, R)

```
1: if L < R then
2:    pIndex = PARTITION(A, L, R);
3:    QUICK-SORT(A, L,pIndex - 1);
4:    QUICK-SORT(pIndex + 1, R);
5: end if
```

Running Time: The running time of quicksort depends on the particular input. If the pivot always partitions into two equal parts then the running time is $T(n) = 2T(\frac{n}{2}) + O(n)$; $T(2) = 1$ which gives $T(n) = O(n \log n)$. This is similar to merge sort. However if the pivot is very close to one side of the sequence then the running time can be as bad as $O(n^2)$: say each time the pivot is the smallest element in the sequence, we do $n-1$ comparisons and place only the pivot in the right place, then $n-2$ comparisons, etc. In the algorithm we are choosing the pivot at random, but still there is a chance that we pick the smallest element at thus end up making $O(n^2)$ comparisons. However the chance is very small as we now show.

First each element has the same probability $\frac{1}{n}$ of being picked as the pivot. if the i^{th} element is selected as the pivot, then the running time is $T(n) = (n-1) + T(i-1) + T(n-i)$. Thus the average running time (versus the worst case running time that we have usually considered so far in this class) is:

$$\begin{aligned} T(n) &= \frac{1}{n}\Big(\sum_{i=1}^{n}(n-1) + T(i-1) + T(n-i)\Big) \\ &= (n-1) + \frac{1}{n}\Big(\sum_{i=1}^{n} T(i-1) + T(n-i)\Big) \\ &= (n-1) + \frac{1}{n}\sum_{i=1}^{n} T(i-1) + \frac{1}{n}\sum_{i=1}^{n} T(n-i) \\ &= (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \end{aligned}$$

One can prove that $T(n) = O(n \log n)$. Look at Chapter 7 of [1] and Section 6.4 of [2] for more details.

11 The Order Statistic or Selection Problem

The Problem: Given a sequence $S = A[1], A[2], \ldots, A[n]$ of elements, and integer k such that $1 \leq k \leq n$, find the k^{th}-smallest element in S.

If k is very close to 1 or n, then we can run the algorithm for max or min k times. So the running time is $O(kn)$. We can also sort the sequence in $O(n \log n)$ time and find the k^{th} element in $O(n)$ time. So if $O(kn) > O(n \log n)$, i.e., $k = \Omega(\log n)$ then sorting is faster.

But can we do better? YES. We can divide-and-conquer the same way as quicksort. In quicksort, the sequence is partitioned by a pivot into two subsequences. Here we need only to determine which subsequence contains the k^{th} element and then continue recursively only for that subsequence. The rest of the elements can be ignored.

Running Time: As in Quicksort, the worst case is $O(n^2)$ but we can show the average running time is $O(n)$. In practice it is very fast even though there are some algorithms with running time $O(n)$ in worst case. Indeed this algorithm is the best for finding only the median, i.e., $k = \frac{n}{2}$.

12 Sorting in Linear Time

12.1 Lower Bound for Sorting

We could improve the running time from $O(n^2)$ (insertion or selection sort) to $O(n \log n)$ (merge sort or quicksort). Can we do still better for comparison based (and not radix sort, for example). Unfortunately the answer is NO. We can use information theory to show that indeed any algorithm needs at least $\Omega(n \log n)$ comparisons. See Section 8.1 of [1] if you are interested to know more details.

12.2 Counting Sort

Counting sort assumes that the input elements are integer between 0 and k where $k = O(n)$. Counting sort stores, for each $0 \leq x \leq k$ the number of elements equal to x and uses this information to find the exact location of each element.

12.3 Least Significant Digit Radix Sort

The Least Significant Digit Radix Sort algorithm works in the following steps:

1. Take the least significant digit of each key (say the number of records to be sorted).
2. Group the keys based on that digit, but otherwise keep the original order of the keys.
3. Repeat the grouping process with each more significant digit

Radix sort is stable sort, i.e., it maintains the relative order of records with equal keys. Step 2 is usually done using bucket sort, which is efficient in this case since there are usually only a small number of digits.

The running time of the algorithm is $O(nk)$ where k is the maximum key length, since we need $O(k)$ repetitions of the loop (each step of the loop requires just a single pass over data). We can implement this by a linked list. However by first counting the number of keys that belong to each bucket before moving the keys into those buckets, and storing them in an array we can indeed implement this by an array as well (using pointers). The pointers here are integers (indices) and we move them after each insert. We can prove the correctness using induction: the hypothesis is that we know how to sort elements with $< k$ digits.

Example: Suppose we want to sort the numbers 170, 45, 75, 90, 802, 24, 2, 66. We have the following steps:

- Sort by least significant digit: 17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 7<u>5</u>, 4<u>5</u>, 6<u>6</u>
- Next: 8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0
- Sort by last digit: 2, 24, 45, 66, 75, 90, <u>1</u>70, <u>8</u>02

12.4 Lexicographic Sort (Most Significant Digit Sort)

It can be used to sort keys (especially strings) in lexicographic (dictionary) order. The Most Significant Digit Radix Sort algorithm works in the following steps:

1. Take the most significant digit of each key.
2. Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate (append) the buckets together in order.

Implementation: Again a two-pass method can be used to first find out how big each bucket needs to be and then place each key into the appropriate bucket using pointers. A single pass can also be used by linked lists. We could also use this algorithm to sort the integers if we pad them with zeros. Again the running time is $O(nk)$. The current implementation of lexicographic sort is not in-place but there are some simple in-place implementations for lexicographic sort.

Example: Suppose we want to sort the strings cab, ab, bb, cd, a, c, cda. We have the following steps:

- <u>a</u>b, <u>a</u>, <u>b</u>b, <u>c</u>ab, <u>c</u>d, <u>c</u>, <u>c</u>da
- a, a<u>b</u>, b<u>b</u>, c, c<u>a</u>b, c<u>d</u>, c<u>d</u>a
- a, ab, bb, c, ca<u>b</u>, cd, cd<u>a</u>

12.5 Bucket Sort

The simple mailroom sort is to allocate a sufficient number of "boxes", called buckets, put each element in the corresponding bucket, sort each bucket, and list the elements in each. This is called as bucket sort. Bucket sort assumes that the input is drawn from a uniform distribution, and thus the number of elements in buckets are almost the same.

For example, If the elements are letters and need to be sorted according to states then we need 50 states, however if it is according to zipcodes (5 digits) then we need 100,000 buckets.

Assume we use the insertion sort for sorting each bucket. Then, the running time of the bucket sort can be written as follows:

$$T(n) = \Theta(n) + \sum_{i=1}^{k} O(n_i^2)$$

where n_i is the number of elements in bucket i. Note that if the input is drawn from a uniform distribution, then $E[T(n)] = \Theta(n)$ (see Section 8.4 of [1]).

13 Design Techniques

So far we have seen divide-and-conquer algorithms which recursively break down the problem into two or more subproblems of the same (or almost the same) type until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Quicksort and Mergesort are these types of algorithms. We prove the correctness of such algorithms often by induction and obtain their running times by the Master Theorem.

Backtracking or Branch-and-Bound technique: It is another approach for finding all (or some) solutions to a computational problem. They often incrementally build candidates to the solution recursively in each step and abandons each partial candidate (backwards) as soon as it determines that it cannot be completed to a valid or optimum solution. The running times of these algorithms is often exponential (e.g. 2^n) and there are several techniques especially in AI to improve their running times.

Dynamic Programming: It is a method for solving problems by breaking them down into simpler subproblems. The main idea is as follows: In general to solve a problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. This is the place that dynamic programming saves time compared to backtracking algorithms, since unlike backtracking algorithms, it seeks to solve each subproblem only once, thus reducing the number of computations. The proof of correctness for dynamic programming is often by induction.

Greedy Algorithms: Unlike backtracking algorithms that try every possible choice (solutions), a greedy algorithm tries to make a locally optimal choice

at each step with the hope of finding a global optimum. In other words, a greedy algorithm never reconsiders its choices. That is the reason for many problems greedy algorithms fail to produce the solution or the optimal solution. Say you have 25-cent,10-cent and 4-cent coins and we want to make change of 41 cents: Greedy produces 25,10 and 4 and fails while a backtracking algorithm gives 25 and four 4 cent coins. However greedy algorithms are often very fast unlike backtracking algorithms. Djikstra's algorithm for shortest paths (that we see later) is a greedy algorithm. Greedy coloring is another application.

14 Dynamic Programming

14.1 Computing Fibonacci Series

If we compute the Fibonacci series by a recursive algorithm, the running time will be an exponential function. The recursive algorithm solves a subproblem several times and results in an exponential running time. If we compute the Fibonacci series by a dynamic programming algorithm, the running time will be linear. The dynamic programming algorithm solves each subproblem once and saves the solution for the further usage.

14.2 Subset Sum Problem

Suppose we are given a knapsack and we want to pack it fully, if it is possible. More precisely, we have the following problem: Given an integer k and n items of different sizes such that the i^{th} item has an integer size s_i, find a subset of the items whose sizes sum to exactly k, or determine that no such subset exists.

14.2.1 Greedy Algorithm:

Always use the first (the largest) item that you can pack. This algorithm fails. Example is $k = 13, n = 4$ and the sizes as $6, 5, 4, 3$. Then greedy packs only 6 and 5, but we can pack 6, 4, and 3.

14.2.2 Backtracking Algorithm:

We do brute-force or exhaustive search in this case.

We call at the beginning with BF$(n, k, \emptyset)$ to get the answer. Since we try both cases at each stage, the running time in the worst case in $\Omega(2^n)$.

14.2.3 Dynamic Programming

Similar to backtracking assume DP(n, k) is true if and only if we can construct k with numbers from $s_1, s_2, \ldots, s_n$. Then the recursion for DP is exactly the same as BF. However we can improve the running time a lot by this observation that the total number of problems may not be too high. There are n possibilities for the first parameter and k possibilities for the second parameter. Thus overall

Algorithm 14.1 BF(n, k,sol)

```
1: if n = 0 and k = 0 then
2:    return true;
3: end if
4: if n = 0 and k > 0 then
5:    return false;
6: end if
7: if k < 0 then
8:    return false;
9: end if
10: return ( BF(n − 1, k,sol) OR BF(n − 1, k − s_n,sol∪{s_n}) )
```

we only have nk different subproblems. Thus if we store all known results in a $n \times k$ array then we compute each subproblem only once. If we are interested in finding the actual subset, then we can add to each entry a flag (sol) that indicates whether the corresponding item was selected in that step or not. This flag (sol) can be traced back from the (n, k)-th entry and the subset can be recovered.

Algorithm 14.2 DP(n, k)

```
flag[0, 0] = 1
for j = 1 to k do
   flag[0, j] = 0
end for
for i = 1 to n do
   for j = 0 to k do
      flag[i, j] = flag[i − 1, j];
      sol[i, j] = 0;
      if s_i ≤ j flag[i − 1, j − s_i] = 1 then
         flag[i, j] = 1;
         sol[i, j] = 1;
      end if
   end for
end for
```

14.3 Knapsack Problem

We have n items where item i has a value v_i and an integer weight w_i. We also have a knapsack. The maximum weight that we can carry in the knapsack is k. The goal is to pick a subset of items to maximize the sum of the values of the items in the knapsack.

14.3.1 Dynamic Programming

Similar to the subset sum problem let $flag[n, k]$ be the optimum solution, if we have a knapsack with capacity k and we can choose between items 1 to n. Then the recursion for DP is the same as the subset sum problem. There are n possibilities for the first parameter and k possibilities for the second parameter. Thus overall we only have nk different subproblems. Thus if we store all known results in a $n \times k$ array then we compute each subproblem only once.

Algorithm 14.3 KNAPSAK(n, k)

```
for j = 0 to k do
   flag[0, j] = 0
end for
for i = 1 to n do
   for j = 0 to k do
      flag[i, j] = flag[i − 1, j];
      sol[i, j] = 0;
      if w_i ≤ j  flag[i − 1, j − w_i] + v_i > flag[i, j] then
         flag[i, j] = flag[i − 1, j − w_i] + v_i;
         sol[i, j] = 1;
      end if
   end for
end for
```

14.4 Longest Common Subsequence (LCS)

A subsequence of a sequence is a sequence obtained by deleting some elements without changing the order of the remaining elements. For example, ADF is a subsequence of ABCDEF.

The problem is to find the LCS of two sequences (strings) given by $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$. Again let $LCS[i, j]$ be the length of LCS of $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_j$. Then

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1, j-1] + 1 & \text{if } a_i = b_j \\ \max\Big(LCS[i, j-1], LCS[i-1, j]\Big) & \text{if } a_i \neq b_j \end{cases}$$

Again if we are not careful then we have a brute-force backtracking algorithm with running time $O(2^{\min\{n,m\}})$. But if we use dynamic programming then the running time is $O(nm)$.

14.5 Matrix Chain Multiplication

We are given a sequence of matrices $A_1, A_2, \ldots, A_n$ where the size of matrix A_i is $p_{i-1} \times p_i$. The goal is to find the most efficient way to multiply these matrices.

Assume the cost to multiply two matrices B and C with sizes $p \times q$ and $q \times r$ is $p \times q \times r$.

Solution: Let $M[i,j]$ be the cost of the most efficient way to multiply $A_i, A_{i+1}, \ldots, A_j$. Then we have:

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \left(M[i,k] + M[k+1,j] + p_{i-1} \times p_k \times p_j \right) & \text{if } i < j \end{cases}$$

Using dynamic programming, we have a running time of $O(n^3)$.

14.6 Independent Set in Trees

An independent set in a graph is a set of vertices such that no two of the vertices are adjacent. Given a tree we want to find a largest independent set in it. Without loss of generality we assume the tree is rooted at say r and T_i is the subtree rooted at node i. For every vertex v let $C(v)$ denote the set of children of v. Then we have

$$IS(i) = \begin{cases} 1 & \text{if } i \text{ is a leaf} \\ \max \left(\sum_{j \in C(i)} IS(j), 1 + \sum_{k \in C(j), j \in C(i)} IS(k) \right) \end{cases}$$

Using dynamic programming, we have a running time of $O(n)$.

15 Greedy Algorithm

15.1 Interval Scheduling

We have a set of jobs $\{1, 2, \cdots, n\}$; the i^{th} job corresponds to an interval of time starting at s_i and finishing at f_i. We say a subset set of jobs is *compatible* if no two of them overlap in time. The goal is to find a compatible subset of jobs with the maximum size.

Greedy choice: We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Greedy choice for this problem: Select the job with the smallest f_i.

Proof Idea:

- We should prove there exists an optimum solution which contains the first greedy choice.

Algorithm 15.1 Greedy Algorithm

1: $R = \{1, 2, ..., n\}$
2: $SOL = \emptyset$
3: **while** $R \neq \emptyset$ **do**
4: Select a job $i \in R$ with the smallest finishing time.
5: $SOL = SOL \cup \{i\}$
6: Delete all jobs from R that are not compatible with job i
7: **end while**
8: Return SOL as the optimum solution.

- We should prove after the greedy choice, we face to the same problem with a smaller size.

15.2 Scheduling to Minimize Lateness

We have a set of jobs $\{1, 2, \cdots, n\}$; the job i has a deadline d_i, and it requires a contiguous time interval of length t_i. We will design an algorithm to assign each job i an interval of time of length t_i. Let $[s_i, f_i]$ be this interval, with $f_i = s_i + t_i$. The lateness of job i is defined to be $\max\{0, f_i - d_i\}$. The goal is to find a schedule which minimizes the maximum lateness, $L = \max_i l_i$.

Greedy choice for this problem: First of all, schedule the job with the smallest d_i.

Algorithm 15.2 Greedy Algorithm

1: Sort the jobs in order of their deadlines
2: Assume for simplicity of notation that $d_1 \leq d_2 \leq \cdots \leq d_n$
3: $time = 0$
4: **for** $i = 1$ to n **do**
5: Schedule job i from $time$ to $time + t_i$.
6: $s_i = time$
7: $f_i = time + t_i$
8: $time = time + t_i$
9: **end for**
10: Return the set of scheduled intervals $[s_i, f_i]$ for $i = 1, \cdots, n$

16 Graph

A graph G with n vertices (nodes) and m edges (arcs) consists of a vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$ and an edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$, where each edge is an unordered pair of vertices. We write uv or $\{u, v\}$ for an edge between u and v. If $uv \in E(G)$, then u and v are adjacent. The vertices contained in an edge are called its endpoints.

We can visualize a graph on paper by assigning a point to each vertex and drawing a curve for each edge between the points representing its endpoint. The vertices can be any objects like cities in a country, points in the plane, etc. and edges are showing relation between the two objects. The graphs can model lots of problems. Graphs can also model people and their friendship relations. In this course we will denote a graph by $G = (V, E)$.

A directed graph or digraph is a graph whose edges are ordered pairs. The edges are represented as (u, v) which means that there is an edge from u to v. Sometimes we consider more general models that allow repeated edges or edges with both endpoints same (loops). Such graphs are called multi-graphs. In this course we usually consider only simple graphs. Note that simple graphs have at most $\binom{n}{2} = \theta(n^2)$ edges.

Let $G = (V, E)$ be an undirected graph. An induced subgraph of G is a graph $H = (V', E')$ such that $V' \subseteq V$ and E' includes all edges in E both of whose incident vertices are in V'. If we just have $E' \subseteq E$ then it is called as subgraph of G.

A degree of a vertex v is represented by $d_v(G)$ or $deg_v(G)$ and is the number of vertices adjacent to that vertex.

Theorem 16.1 *Let $G = (V, E)$ be an undirected graph. Then $\sum_{v \in V} deg_v(G) = 2|E|$*

Proof: The equation follows by double counting. ■

A **walk** of length k is a sequence $v_0, v_1, \ldots, v_k$ of vertices such that $e_i = \{v_{i-1}, v_i\} \in E$. A **path** is a walk with no repeated vertex. A **cycle** is a walk such that the first and the last vertex are the same, i.e., $v_0 = v_k$. A cycle is **simple** if there is no repeated vertex.

16.1 Representation of Graphs for Algorithms

There are two ways to represent graphs: as a collection of adjacency lists or as an adjacency matrix. Adjacency list is preferred for sparse graphs, i.e., graphs having small number of edges like $O(n)$. Adjacency matrix is preferred for dense graphs, i.e., graphs having large number of edges like $\Omega(n^2)$.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex of V. For each $u \in V$, the adjacency list $Adj[u]$ contains pointers to all vertices v such that there is an edge $uv \in E$ (the elements of the list can be in any arbitrary order or in a sorted order). Irrespective of whether the graph is directed or undirected, the adjacency-list representation has the memory amount $O(|V| + |E|)$. The disadvantage is that determining if an edge uv is present in the graph or not can take $O(\deg_v(G))$ time.

For the adjacency matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. The

adjacency matrix is a $|V| \times |V|$ array $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{otherwise} \end{cases}$$

The adjacency matrix of a graph requires $\theta(|V|^2)$ space independent of the number of edges in the graph. However checking if $uv \in E$ is in $\theta(1)$.

Our graphs can be weighted also (for which each edge has an associated weight) and can be easily represented by both methods described above. A **complete graph** or **clique** is a simple graph in which every pair of vertices forms an edge and thus $|E| = \binom{n}{2}$. An independent set in a graph is a vertex subset $S \subseteq V(G)$ such that the induced subgraph $G[S]$ has no edges. The **complement** of a simple graph, written as $\overline{G}$, is a graph with same vertex set as G such that u, v are adjacent in G if and only if u, v are not adjacent in $\overline{G}$.

16.2 Bipartite Graph

A graph is **bipartite** if its vertex set can be partitioned into at most two independent sets V_1, V_2 (like the person-job graph above). Then we represent it as $G(V_1, V_2, E)$. A **complete bipartite** graph is a bipartite graph in which the edge set consists of all pairs having a vertex from V_1 and V_2. For bipartite graphs, we can have a simple adjacency matrix as follows: We have a matrix (array) of size $|V_1| \times |V_2|$, namely $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that this array is not symmetric anymore.

Example: Job Assignments and Bipartite Graphs:
Suppose we have m jobs and n people and each person can do some of the jobs. Can we make assignments to fill the jobs? We model the available assignments by a graph having a vertex for each job and each person, putting job j adjacent to person p if p can perform job j. If each person can do only one job, then the assignment problem is the matching problem.

16.3 Tree

A **tree** is a connected undirected graph with no cycles. A **DAG** (directed acyclic graph) is a directed graph with no cycles. The following three statements are equivalent:

1. G is a tree.
2. G is connected and $|E| = |V| - 1$.
3. G has no cycle and $|E| = |V| - 1$.

A **rooted tree** is a tree in which one of the vertices is distinguished from the others, called the root. We often call vertices of a rooted tree (in general for directed graphs) as **nodes**. If the last edge on the path from the root r of a tree T to a node x is (y, x), then y is the **parent** of x and x is the **child** of y. A node with no children is called a **leaf**. A node which is not a leaf is called as an **internal** node. Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called as an **ancestor** of x. If y is an ancestor of x, then x is a **descendant** of y.

17 Graph Traversals

There are two famous algorithms for graph traversal: Depth-first search (DFS) and Breadth-first search (BFS). Both algorithms work for both undirected and directed graphs, though we focus more on undirected graphs.

17.1 DFS

The traversal is started from an arbitrary vertex v, which is called as the root of the DFS. The root is marked as visited. An arbitrary (unmarked) vertex v_1 connected to v is then picked and a DFS starting from v_1 is performed recursively. The recursion stops when it reaches a vertex v such that all the vertices connected to v are already marked. If after the DFS from v_1 terminates all the vertices connected to v are also marked then the DFS for v also terminates. Otherwise another arbitrary unmarked vertex v_2 connected to v is picked and a DFS starting from v_2 is performed, and so on.

Algorithm 17.1 DFS(G, v)

```
1: Mark v
2: DFS-Number[v]=DFS-N, DFS-N++
3: Perform pre-work on v;
4: for all edges v, w do
5:    if w is unmarked then
6:       DFS(G, w); add edge (v, w) to DFS-Tree;
7:       Perform post-work for (v, w);
8:    end if
9: end for
```

To incorporate different applications, we do pre-work at the time the vertex is marked and post-work after we backtrack from an edge and find that the edge leads to a marked vertex.

Time Complexity: Each edge is looked exactly twice (once from each end). So the running time is $O(|E|)$ if the graph is connected and $O(|V| + |E|)$ in general due to vertices that are not connected to anything.

Theorem 17.1 *If G is connected, then all vertices are marked and all edges are visited by the DFS.*

Proof: The proof is by contradiction. Let U be the set of unmarked vertices at the end. Since G is connected at least one vertex $u \in U$ is connected to a marked vertex say v and thus we should have had visited u when we visited v by definition of the algorithm. Now since all vertices are visited and since whenever a vertex is visited all its incident edges are considered, all the edges in the graph are considered too. ■

Thus above theorem gives an application of DFS for checking the connectivity of the graph, i.e., the number of marked vertices that we count should be n for the graph to be connected. DFS has various other applications as well.

17.2 BFS

It traverses the graph level-by-level. If we start from a vertex v, then all "children" of v are visited first. The second level includes a visit to all the "grandchildren" of v and so on. The procedure is non-recursive.

Algorithm 17.2 BFS(G, v)

```
1: Mark v
2: BFS-Number[v]=BFS-N=1, level[v]=0;
3: Put v in a queue (First In First Out);
4: while The queue is not empty do
5:    Remove the first vertex w from the queue;
6:    Perform pre-work on w;
7:    for All edges (w,x) such that x is unmarked do
8:       Mark x; BFS-Number[x]=BFS-N, BFS-N++; level[x]=level[w]+1;
9:       Add (w,x) to the BFS-Tree
10:      Put x in the queue.
11:   end for
12: end while
```

Note that in both DFS and BFS, we assign a DFS-Number or BFS-Number to each vertex and we create a DFS-Tree or a BFS-Tree.

Time Complexity: Again we enqueue and dequeue each vertex once and we visit each edge only twice. So the total running time is $O(|V| + |E|)$.

Theorem 17.2 *For each vertex w, the path from the root to w in the BFS-Tree is a shortest path from the root to w in G.*

Proof: Proof follows by induction of the level of a vertex. ■

17.3 Main Properties of DFS and BFS Trees

Theorem 17.3 ***(Main Property of DFS-Trees)*** *Every edge in G not belonging to T, connects two vertices of G, one of which is the ancestor of the*

other in T, i.e., there are no cross-edges.

Proof: Let (v, u) be an edge of G, and suppose that v is visited by DFS before u. After v is marked, we perform DFS starting from all its neighbors. Since u is a neighbor of v, DFS either starts from u in which case (v, u) belongs to T or the DFS will visit u before it backtracks from v, in which case u is a descendant of v in T. ■

Theorem 17.4 ***(Main Property of BFS-Trees)*** *If (u, v) is an edge of G not belonging to T, then it connects two vertices whose level numbers differ by at most 1.*

Proof: The idea is to consider the first of v and u which is visited and then the other vertex has difference at most one (or potentially zero). You can see detailed proofs in the book [1]. ■

See the properties of DFS in directed graphs in the book [1].

18 Topological Sorting for DAGs

DAGs (directed acyclic graphs) are directed graphs with no directed cycles. Suppose there is a set of tasks that need to be performed one at a time. Some tasks depends on other tasks and they cannot be started until the other tasks are completed. All the dependencies are known and we want to arrange a schedule for performing the tasks which is consistent with the dependencies. We can associate a directed graph whose vertices are the tasks and whose edges are the dependencies. The graph must be acyclic otherwise the tasks can never be completed. This problem is known as topological sorting. More formally, the problem is as follows: Given a DAG with n vertices, label the vertices from 1 to n such that if v is labeled k, then all vertices that can be reached from v by a directed path are labeled with labels $> k$. First we have a small observation: A DAG always contains a vertex of indegree 0. The proof is similar to the one which shows that every tree has a leaf. Otherwise we can traverse the graph forever which contradicts the fact that we do not have a cycle. Note that the same argument gives the existence of a vertex of outdegree 0 as well. Thus the algorithm is as follows: First we find a vertex of indegree 0. Once we find it, we label it with 1 and remove its adjacent edges and label the rest of the graph (which is still acyclic) by induction.

Time Complexity: Initializing indegrees with DFS takes $O(|V|+|E|)$ time. Finding a vertex of indegree 0 takes constant time using a queue. We consider each edge (v, w) exactly once we bring v out of the queue. Thus we update the Indegree variable at most $|E|$ times and hence the total running time is $O(|V| + |E|)$.

Algorithm 18.1 TOPOLOGICAL-SORTING

Input: A DAG G.
Output: A topological sort of G.

```
1:  Initialize v.Indegree for all vertices by DFS;
2:  G-label:=0;
3:  for i = 1 to n do
4:     if v_i.Indegree=0 then
5:        Put v_i in the queue;
6:     end if
7:  end for
8:  while Queue is not empty do
9:     Remove a vertex v from the queue;
10:    G-label:=G-label+1; v.label:=G-label;
11:    for all edges (v,w) do
12:       w.Indegree:=w.Indegree-1;
13:       if w.Indegree =0 then
14:          Put w in the queue.
15:       end if
16:    end for
17: end while
```

19 Single Source Shortest Path

Given a directed graph $G = (V, E)$ with weights (length) associated with the edges, find the shortest paths from v to all the other vertices of G. Here we talk about lengths instead of weights, since the problem is traditionally called as the shortest path (rather than the lightest path) problem. The length of a path is the sum of lengths of its edges.

19.1 Solution for DAGs:

We know how to compute a topological sort for a DAG in $O(|V| + |E|)$ time. If z is a vertex with label k then

1. There are no paths from z to vertices with labels $< k$.
2. There are no paths from vertices with labels $> k$ to z.

Thus without loss of generality, we assume that v is the vertex with label 1. Now we use induction as follows:

Induction Hypothesis
Given a topological ordering, we know how to find the lengths of the shortest paths from v to the first $n - 1$ vertices.

We remove the n-th vertex and solve the reduced problem by induction, then take the min of values SP[w]+ length(w,z) over all edges $(w, z) \in E$ as the value of $SP[z]$. The algorithm can be written using only "for" loops.

Algorithm 19.1 Shortest-Paths for DAGs

```
1: SP[v] = 0;
2: SP[w] = ∞ for w ≠ v;
3: for i = 2 to n do
4:    Let z be the vertex with label i;
5:    for all w such that (w, z) ∈ E do
6:       if SP[w] + length(w, z) < SP[z] then
7:          SP[z] := SP[w] + length(w, z);
8:       end if
9:    end for
10: end for
```

Time Complexity: Topological sort needs $O(|V| + |E|)$ time. We check every vertex and every edge only once. Thus the total running time is $O(|V| + |E|)$.

19.2 Solution for General Directed Graphs

Note that we can replace each edge uv with bidirected edges. This directed graphs are more general than undirected graphs. This fact is usually true. Since general graphs may not have a topological order, the same induction hypothesis as before does not work. A similar idea works however.

Induction Hypothesis
Given a graph and a vertex v, we know the k vertices that are closest to v and the lengths of the shortest paths to them.

Denote the set containing v and the k closest vertices to v by v_k. The problem is to find a vertex w that is closest to v from among the vertices not in v_k, and to find the shortest path from v to w that can go through only the vertices in v_k. It cannot included vertices outside of v_k since they would be closer to v than w. Therefore to find w, it is sufficient to consider only edges connecting vertices from v_k to vertices not in v_k; all other edges can be ignored for now. Let (u, z) be an edge such that $u \in v_k$ and $z \notin v_k$. Such an edge corresponding to a path from v to z, which consists of the shortest path from v to u (already known by induction) and the edge (u, z). We only need to compare all such paths and take the shortest among them.

The algorithm implied by the induction hypothesis is the following: At each iteration, a new vertex w is added such that $\min_{u \in v_k} \Big(SP[u] + \text{length}(v, w) \Big)$ is the minimal over all $w \notin v_k$. By the argument above, w is indeed the $(k+1)$-th closest vertex to v; thus adding it extends the induction hypothesis. This greedy algorithm is known as Djikstra's Algorithm.

Implementation and Complexity: A heap is a good data structure for finding minimum elements and updating lengths of elements. We always keep all vertices not yet in v_k in the heap. Initially all vertices are there with v on the

Algorithm 19.2 Djikstra(G,v)

```
1: SP[v] = 0;
2: (SP[w] = ∞ and mark[w]=false) for w ≠ v;
3: while There exists an unmarked vertex do
4:    Let w be an unmarked vertex such that SP[w] is minimal;
5:    mark[w]=true;
6:    for All edges (w, z) such that z is unmarked do
7:       if SP[w] + length(w, z) < SP[z] then
8:          SP[z] := SP[w] + length(w, z);
9:       end if
10:   end for
11: end while
```

top (all other vertices have shortest paths ∞). We find the minimum easily and delete it in $O(\log |V|)$ with total $O(|V| \log |V|)$. Say w is the minimum. Now we update all $(w, z) \in E$. If SP[z] is updated and changed, then the position of z may change in the heap. First we need to locate z in the heap by another data structure, say an array with pointers to their location in the heap. We can access z in the heap in $O(1)$ and update its position in the heap by exchanging and moving up until its appropriate position is found (note that the path lengths only decrease). This is essentially the same as heap insert and can be done in $O(\log |V|)$. Since there are at most $|E|$ updates and each takes $O(\log |V|)$ leading to $|E| \log |V|$ comparisons in the heap. Hence the total running time is $O\Big((|V| + |E|) \log |V|\Big)$ instead of $O(|V| + |E|)$ that we had for DAGs. All edges selected in the process (from z to its parent w) form the Shortest-Path-Tree (similar to BFS-Tree and DFS-Tree).

20 All-Pairs Shortest Paths

Given a directed graph $G = (V, E)$ with weights (length) associated with the edges, find the shortest paths between every pair of nodes.

20.1 Bellman-Ford algorithm for Single-Source Shortest Paths

Djikstra's algorithm is good if there is no edge of negative length (check this as an exercise!), but the Bellman-Ford works as long as there is no "negative cycle", i.e., a cycle whose edges sum to a negative value and it can even detect such cycles.

Time Complexity: Clearly it is $O(|V| \cdot |E|)$ time which is worse than Djikstra's $O\Big((|V| + |E|) \log |V|\Big)$.

Proof of Correctness: This can be shown using induction.

Algorithm 20.1 Bellman-Ford(G,v)

```
1: SP[v] = 0;
2: SP[w]=∞ for w ≠ v;
3: for i = 1 to |V| − 1 do
4:    for each (u, w) ∈ E do
5:       if SP[u] + length(u, w) < SP[w] then
6:          SP[w] := SP[u] + length(u, w);
7:       end if
8:    end for
9: end for
```

Induction Hypothesis
If there is a path from v to u with at most i edges, then SP[u] is at most the length of the shortest path from v to u with at most i edges, where i is the number of repetitions.

Proof: Consider the shortest path P from v to u with at most i edges. Let w be the last vertex before u on this path. Then the part of P from v to w is a shortest path from v to w with at most i-1 edges and by induction SP[w] after i iterations is at most the length of this path. This SP[w]+ length(w,u) is at most the length of P and we find it in the i-th iteration. ■

Now if there is no negative cycle, the length of any shortest path in terms of its edges is at most $|V| - 1$ and thus we find it by Induction Hypothesis for $i = |V|-1$. The Induction Hypothesis itself is a good property of this algorithm, which has applications in routing protocols as well.

20.2 Floyd-Warshall Algorithm for All-Pairs Shortest Paths

The Problem: Given a weighted graph $G = (V, E)$ with non-negative edge lengths, find the minimum length paths between all pairs of vertices. Of course we can run Djikstra's algorithm for $|V|$ times to get a total time of $O\Big(|V|(|V| + |E|) \log |V|\Big) = O(|V| \cdot |E| \log |V|)$ which is good for sparse graphs but not the best for dense graphs.

Time Complexity: Very simple algorithm to implement in $O(|V|^3)$ time.

Proof of Correctness: This can be shown using induction.

Induction Hypothesis
We know the lengths of the shortest paths between all pairs of vertices such that only k-paths, i..e, except endpoints, the highest-labeled vertex on the path is labeled k, for some $k \leq m$, are considered. In i-ith iteration of the loop we computed all these.

Proof: For m=1, the basis is correct since we have only direct edges as paths. For general m, the shortest path between any pair can have v_m at most one and

Algorithm 20.2 Floyd-Warshall(G)

```
1: for m = 1 to n do
2:   for x = 1 to n do
3:     for y = 1 to n do
4:       if weight[x, m] + weight[m, y] < weight[x, y] then
5:         weight[x, y] := weight[x, m] + weight[m, y];
6:       end if
7:     end for
8:   end for
9: end for
```

then the paths to and from v_m are k-paths, for $k \leq m-1$. Since we sum the paths to and from v_m and compare it with the best path found so far in the algorithm, we are done.

■

As you saw in Bellman-Ford and Floyd-Warshall, the whole idea is to get the Induction Hypothesis correct and the rest is then trivial.

21 Minimum Spanning Tree (MST)

Say there is a set of computers in a office or a set of sites for VPN or a set of cities that we want to connect to each other. We can model all these with an undirected graph whose edges represent connections and the weights are the lengths. We want to find a connected subgraph with min sum of edge lengths. Note that the subgraph should be a tree. More formally, the problem is as follows: Given an undirected connected weighted graph $G = (V, E)$, find a spanning tree that connects every vertex with minimum cost.

21.1 Prim's algorithm

We now give an algorithm due to Prim for thus problem. Note that this is a greedy algorithm. The implementation is very similar (almost identical) to Djikstra (using heap). At each stage we can keep SE[v] (instead of SP[v] in Djikstra) which keeps the min edge connectivity v_{new} to this vertex v and we update it each time that we add a new vertex to v_{new}. Identical to Djikstra, the running time is $O\Big((|V| + |E|) \log |V|\Big)$.

Proof of Correctness: This can be shown using induction.

Induction Hypothesis
There is always a best solution (optimum) that has the first k edges that we add to E_{new}

Proof: The base is trivial since there is no edge. Note that v_{new} plays the same role of v_k in Djikstra. Consider the tree OPT which only the first k edges

Algorithm 21.1 Prim(G,v)

1: $v_{new} = \{w\}$ and $E_{new} = \{\}$ where w is any arbitrary vertex;
2: **while** $v_{new} \neq v$ **do**
3: Find an edge (u,v) where $u \in v_{new}$ and $v \in V \setminus v_{new}$ with min weight (break ties arbitrarily).
4: $v_{new} = v_{new} + \{v\}$;
5: $E_{new} = E_{new} + \{(u, v)\}$;
6: **end while**

of E_{new}. We prove there is another OPT' with same cost which has the first k+1 edges of E_{new}. Now let (u, v) be the (k+1)-th edge that we add to E_{new} (and thus to the tree) where $u \in v_{new}$ and $v \notin v_{new}$. Now let us add (u,v) to the tree OPT. There should be a cycle and there is an edge (u',v') where $u' \in v_{new}$ and $v' \notin v_{new}$. Since we checked all edges going out of v_{new} we have $w(u', v') \geq w(u, v)$. So we add (u,v) instead of (u', v'), preserve connectivity, maybe lower the total weight and still have k+1 edges in common with new OPT'. ■

21.2 Kruskal's algorithm

Kruskal's algorithm is another greedy algorithm for finding the minimum spanning tree. Here is the algorithm:

- Put each vertex in a separate set.
- Put all edges in the graph in set S
- While S is non-empty do
  - Let $e = (u, v)$ be the edge with the minimum cost in S.
  - Remove e from S
  - If u and v are in different sets, joint two sets and add e to the final solution.

Time Complexity: We can sort all edges (or use a heap) for removing the edge with the smallest cost in each step. The overall cost would be $O(|E|log|E|)$ which is the same as $O(|E|log|V|)$. Next, we may use a disjoint-set data structure to store the component of each vertex. Note that the running time for both operations find and union would be logarithmic in a disjoint-set data structure. We need to preform at most $O(|E|)$ find operations and $O(|V|)$ union operations, and thus the overall running time is $O(|E|log|V|)$.

Proof of Correctness: This can be shown using induction.

Induction Hypothesis
There is always a best solution (optimum) that has the first k edges that we add to F

Algorithm 21.2 Kruskal(G)

```
1: F = ∅
2: S = E
3: while S ≠ ∅ do
4:    (u, v) = edge with the minimum cost in S.
5:    remove (u, v) from S.
6:    if SET(u) ≠ SET(v) then
7:       Add (u, v) to F
8:       UNION(u, v)
9:    end if
10: end while
11: return F
```

Proof: The base is trivial since there is no edge. Consider the tree OPT which only the first k edges of F. We prove there is another OPT' with same cost which has the first $k+1$ edges of F. Now let (u, v) be the $(k+1)$-th edge that we add to F (and thus to the tree). Now let us add (u, v) to the tree OPT. There should be a cycle. Since we know (u, v) does not make a cycle with the first k edges of F, we can conclude that there is an edge such as (u', v') in the cycle with $w(u', v') \geq w(u, v)$. So we add (u, v) instead of (u', v'), preserve connectivity, maybe lower the total weight and still have $k+1$ edges in common with F. ■

22 NP-completeness

22.1 Easy and Hard Problems

So far we have seen many algorithms. All of them had polynomial running times, i.e., $O(n^k)$ for some constant k. Lots of them were indeed linear $O(n)$ or quadratic $O(n^2)$. These problems are **easy** problems. But are there problems which are **hard**, i.e., they need exponential time like $O(2^n)$ or $O(n!)$. The answer is YES. There are some weird problems for which we know we cannot solve them in polynomial time and we need exponential time, but for lots of normal problems, still we do not know the answer. These problems lie in the class of **NP-complete** problem. Note that NP stands for "Non-deterministic polynomial" and not "Not in polynomial" for now, but the conjecture is that indeed it is true. If you can solve this problem, you will get Millennium Prize of US $ 1,000,000 by Clay Mathematics Institute. Let us be a bit more formal now.

22.2 Decision Problems

Decision problems are problems for which the answer is either YES or NO, e.g,, can we find a shortest path or an MST of cost w. Note that if we can solve such

problems then we can solve lots of optimization problems as well by a binary search.

P is the class of all decision problems that can be solved in polynomial time, i.e., $O(n^k)$ for some constant k.

EXP is the class of all decision problems that can be solved in exponential time, i.e., $O(2^{poly(n)})$ where poly(n) is some polynomial in n.

22.3 Polynomial-time Verification

For many problems that may be very hard to solve, we might have the property that it is easy to **verify** whether its answer is correct. For example, consider the coloring problem: Given an undirected graph G, find the min number of colors needed so that we can color each vertex with one of these colors and have a **valid coloring**, i.e., an assignment of colors to vertices such that each vertex is assigned one color and no two adjacent vertices have the same color. The 3-Coloring problem asks if we can have a valid coloring with 3 colors. Note that though finding a 3-coloring of a given graph is not easy, however if you somehow obtain a solution with 3-colors it is easy for someone to **convince** that you did: just check that you did not use more than 3 colors in $O(|V|)$ time and check validity of coloring in $|E|$ time. Thus though we do not know any poly time algorithm to decide if a given graph has a valid 3-coloring, there is a very efficient way to verify if a given graph has your answer as a valid 3-coloring. The solution that you provide is called as a **certificate**. This is some piece of information which allows us to check whether the graph has a valid 3-coloring. If it is possible to verify the accuracy of a certificate in poly time, we say that the problem is **polynomial-time verifiable**.

NP is the set of all decision problems that can be verified by a polynomial time algorithm.

Note that poly-time verifiable and solving in poly time are two very different things, e.g., 3-coloring problem is NP-complete (as we will see later) but is verifiable in poly time. Also note that polynomial verification is not always easy. For example, consider the problem of deciding whether the graph has exactly one valid 3-coloring. It is easy to check that there is one but not clear to show that this is the only one.

Then why do we say NP (non-deterministic poly time) instead of VP (verifiable in poly time)? Due to history, here we are referring to a non-deterministic computer which can make guesses for the certificates and thus we only need to verify the certificate in poly time. You can learn more on this topic in other courses such as Complexity Theory and Formal Language Theory.

Note that it is clear that $P \subseteq NP$, since for any problem that we can solve in poly time, we can surely verify it in poly time. But we still do not know the reverse, i.e., whether $NP \subseteq P$, and this is the big open problem mentioned before in the lecture. Many experts believe that $NP \nsubseteq P$ and thus $P \neq NP$.

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Udi Manber, *Introduction to Algorithms - A Creative Approach*, Addison-Wesley, 1989.