# 6.1220 Notes

**Lecturers: Aleksander Madry, Srini Raghuraman, Virginia Williams**

Andrew Liu

Spring 2023

My notes for 6.1220, "Design and Analysis of Algorithms". The instructors for this course were Aleksandr Madry (`https://madry.mit.edu/`), Srinivasan Raghuraman (`https://people.csail.mit.edu/srirag/`), and Virginia Williams (`https://people.csail.mit.edu/virgi/`).

Last updated on Saturday 27th May, 2023.

# Contents

# 1 February 7, 2023

## 1.1 Administration

Problem sets are hard. Start early, don't wait until the last minute (in fact, there will be no office hours on the day that problem sets are due, in order to discourage this practice). The worst two problem sets are dropped. You are allowed to turn in problem sets up to 2 days late, at most 3 times, with no penalty. Collaboration on problem sets is highly encouraged.

Prof. Madry calls this class the "Art and Craft" of Algorithms. In this class, the content covered will generally fall into three different themes:

- Techniques
- Models of Computation
- Intractability

## 1.2 Intro to Complexity

We will learn more about this throughout the semester. Algorithms that can be solved in $O(n^c)$ for some constant $c$ are said to be efficient.

**Definition 1.1** (P)
$P$ is the set of all decision problems that can be solved in $O(n^c)$.

For example, the **Eulerian cycle problem**: given a graph, is it possible to find a cycle which traverses every edge of the graph exactly once?

**Definition 1.2** (NP)
$NP$ is the set of all decision problems that can be verified in $O(n^c)$.

$NP$ does not stand for "not polynomial"; instead, it stands for non-deterministic polynomial time.

As an example, the **Hamiltonian cycle problem** is NP-complete: given a graph, is it possible to find a cycle which traverses every vertex of the graph exactly once? NP-completeness means that it is both in NP (since it is easy to verify that a valid

cycle is in fact valid), and it is NP-hard (approximately, it is as hard as all other problems in NP). We will work with these ideas more formally in later lectures.

## 1.3 Scheduling (greedy)

We have one resource, and $n$ requests for this resource $R = \{r_1, \ldots, r_n\}$. Each request corresponds to some start and finish time, i.e., $r_i = [a_i, b_i]$. For each request $r$, define its set of incompatible requests as

$$\text{Inc}(r) = \{r' | r \cap r' \neq \emptyset\}$$

Given $R$, compute the maximal set of compatible requests.

---

A typical greedy approach is structured like this:

- Use a simple (myopic) rule to pick $r_i$
- Include $r_i$ in our solution, and remove everything in the set $\text{Inc}(r_i)$.
- Repeat until there are no more requests remaining.

It remains to figure out what simple rule we should use. Let's consider a few candidates.

- Remove the request with the shortest length: this does not work. For a counter example, consider $R = \{[1,3],[3,5],[2.5,3.5]\}$. The shortest interval can still kill multiple longer intervals that do not overlap.
- Remove the request with the earliest start time: this does not work. For a counter example, consider $R = \{[1,5],[2,3],[4,5]\}$. The earliest request can kill everything else.
- Remove the request with the smallest number of incompatible requests: this does not work. For a counter example, consider

  $$R = \{[1,2],[2,3],[3,4],[4,5],[1.5,2.5],[1.5,2.5],[2.5,3.5],[3.5,4.5],[3.5,4.5]\}.$$

- Remove the request with the earliest end time: this works.

**Lemma 1.3**

The greedy algorithm while always removing the request with earliest end time produces an optimal solution.

*Proof.* Assume otherwise. Let $S$ be our greedy scheduling, and $S'$ be an optimal scheduling. Let $s_1, \ldots$ be the greedy schedule, in order, and $s'_1, \ldots$ be the optimal schedule, in order. Let $s_i = [a_i, b_i]$ be the first task at which the two schedules differ (if the two schedules are the same, we are done). We must have $b_i < b'_i$ by our greedy algorithm. Therefore, we could replace $s'_i$ with $s_i$ and not decrease the number of scheduled tasks in the optimal solution. This implies $|S| \leq |S'|$, which means that the greedy solution is also optimal. □

## 1.4 Scheduling (DP)

Now, imagine that we have the same problem, but tasks are weighted. Our new goal is to schedule tasks so that the weight is maximal.

---

We can use dynamic programming. Sort the tasks by start time, so that $a_1 \leq a_2 \leq \ldots \leq a_n$. Let $\textsc{Opt}(R)$ be weight of the best schedule given $R$. The recurrence is given by

$$\textsc{Opt}(R) = \max\{\textsc{Opt}(R - r), w_r + \textsc{Opt}(R - \text{Inc}(r))\},$$

with base case $\textsc{Opt}(R) = 0$ when $R = \emptyset$. Computing $\text{Inc}(r)$ naively at each step gives a runtime of $O(n^2)$. This can be sped up using binary search to give a runtime of $O(n \log n)$.

# 2 February 9, 2023

Today we'll be going over the divide and conquer technique, which are problems that can be split into subproblems in such a way that

$$T(n) = aT\left(\frac{m}{b}\right) + (\text{anything else})$$

## 2.1 Median Finding

Given a set $S$ of $n$ numbers, define the rank of $x$ as the number of elements in $S \leq x$. Define the median of $S$ as the element of rank $\lfloor (n+1)/2 \rfloor$.

---

Idea: solve this more general problem first.

**Example 2.1**
Given a set of $n$ distinct numbers $S$, and index $i \in [n]$, find $x \in S$ such that $\text{rank}(x) = i$.

One idea: sort $S$, then return the element at position $i$. The runtime of this algorithm is $O(n \log n)$.

We can do better, due to an algorithm by **Blum, Floyd, Pratt, Rivest, Tarjan** in 1973:

- Pick an element $x \in S$
- Compute $L = \{y \in S : y < x\}$, and $G = \{y \in S : y > x\}$. From this, we know that $\text{rank}(x) = |L| + 1$.
- If $\text{rank}(x) = i$, then we're done. If $\text{rank}(x) > i$, find the element of rank $i$ in $L$. Otherwise, find the element of rank $i - |L| - 1$ in $G$.

The runtime of step 1 is $O(n)$ (generous). The runtime of step 2 is $O(n)$. The runtime of step 3 is $T(\max\{|L|, |G|\})$. Therefore,

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

Note that $\max\{|L|, |G|\} \leq n - 1$, so our worst case performance is $T(n) = O(n^2)$. However, this is only possible if we were to choose $x$ that is optimally bad at each step of the recursion. If we instead pick $x$ "cleverly", we can guarantee a better runtime.

**Definition 2.2**
$x$ is **$c$-balanced** for some $c < 1$ if and only if

$$\max\{\text{rank}(x), n - \text{rank}(x)\} \leq cn.$$

If we choose $c$-balanced $x$ at each step of the way, our new recursion becomes

$$T(n) = T(c \cdot n) + O(n),$$

which gives

$$T(n) = O(n) + O(c \cdot n) + \ldots \in O\left(\frac{n}{1-c}\right) \in O(n),$$

which is what we wanted. Here is one way to run the algorithm in a way that selects $x$ "cleverly":

- Divide $S$ into $n/5$ groups of size 5.
- Sort each group and find the median of each one.
- Compute the median of these $n/5$ medians, call $x$.
- Continue as before.

**Lemma 2.3**
$x$ is 3/4-bounded.

*Proof.* There are at least $n/10$ group medians $\leq x$, which implies that there are at least $3n/10$ elements $\leq x$. Therefore,

$$|L| \geq \frac{3n}{10} \geq \frac{n}{4},$$

which implies that $n - \text{rank}(x) \leq 3n/4$. Similarly, there are at least $3n/10$ elements $\geq x$, so $\text{rank}(x) \leq 3n/4$. □

**Lemma 2.4**
Given that $x$ is 3/4-bounded, our algorithm is linear.

*Proof.* This is equivalent to showing that there exists $c_1$ such that $T(n) \leq c_1 \cdot n$. We will prove this using induction. Our recursion is

$$T(n) = T\left(\frac{3}{4}n\right) + T\left(\frac{n}{5}\right) + O(n).$$

Let $c_2$ be the constant such that the $O(n)$ term contributes less than or equal to $c_2 \cdot n$ work at each step of the recursion. Now, assume that our claim is true for all

$1, \dots, n-1$. Then,

$$T(n) \le \frac{c_1 \cdot n}{5} + \frac{3c_1 n}{4} + c_2 n = c_1 n + \left(c_2 - \frac{c_1}{20}\right) n < c_1 n,$$

since we can set $c_1$ to be as large as we want. □

## 2.2 Integer multiplication

Given two $n$-bit integers $a, b$, compute $a \cdot b$. ($n$ can be as large as we want, so multiplication in the usual sense is not constant time).

---

One approach is to do "old-school" multiplication, i.e., set up the multiplication table and manually compute everything. The complexity of this solution is $O(n^2)$.

A more efficient solution is to use divide and conquer. Let

$$a = 2^{n/2} \cdot x + y,$$
$$b = 2^{n/2} \cdot w + z,$$

where $x, y, z, w$ are all $n/2$-bit integers. Then,

$$a \cdot b = 2^n \cdot xw + yz + 2^{n/2} \cdot (xz + yw),$$

so our recursion is

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) = O(n^2),$$

by Master theorem. Unfortunately, this is just as bad as the normal multiplication algorithm, but this bound can be improved by reducing the number of multiplications required.

**Lemma 2.5** (Anatoli Karatsuba, 1962)

$$XZ + YW = (X + Y)(Z + W) - XW - YZ.$$

So, to complete the recursion, it suffices to only compute three products: $(x +$

$y)(z+w)$, $xw$, and $yz$. Then, our recursion becomes

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.58}).$$

# 3 February 14, 2023

Today we'll be going over randomized algorithms. Properties of randomized algorithms:

- Decisions are in some way based on random numbers $r_1, \ldots, r_k \in_R \{1, \ldots, R\}$
- Given some input $x$, the same algorithm may run different sequences of operations, have different running times, and produce different outputs.

Two examples of general classes of randomized algorithms:

- **Monte Carlo** algorithms are always polynomial time. The probability that these algorithms return a correct answer is "high", but not necessarily 1.
- **Las Vegas** algorithms have polynomial run time in expectation. On the other hand, it guarantees a correct answer.

## 3.1 Verifying matrix products

Given matrices $A$,$B$,$C$, verify whether $A \cdot B = C$. For now, assume we are working modulo 2.

---

Introducing **Frievald's Algorithm**:

- Pick a random binary $v = (v_1, \ldots, v_k)$, such that $\mathbb{P}[v_i = 1] = 1/2$
- Compute $\hat{v} = A(Bv)$ and $\widetilde{v} = Cv$
- If $\hat{v} = \widetilde{v}$, return YES. Otherwise, return NO.

The runtime of this algorithm is $O(n^2)$, which is faster than normal matrix multiplication. This is an example of a Monte Carlo algorithm, since it is always time-efficient, but does not guarantee a correct answer. The main hope is that the randomization makes the probability that an incorrect answer is produced low.

**Lemma 3.1**
If $A \cdot B \neq C$, then $\mathbb{P}[ABv \neq Cv] \geq 1/2$.

*Proof.* Let $D = C - AB \neq 0$. Consider any vector $r$ such that $Dr = 0$, i.e., if our algorithm chooses $r$, then it would return an incorrect result. Since there exists some $D_{ij} = 1$, we have $De_i \neq 0$, so $D(r + e_i) \neq 0$. Note that $r + e_i$ is the same vector as $r$ with the $i$th bit flipped, so there is an injective mapping between vectors that return the incorrect result and vectors that return the correct result, which proves the lemma. □

## 3.2 Quick Select

Recall the median finding (rank finding) algorithm we discussed last time:

- Pick an element $x \in S$ "cleverly"
- Compute $L = \{y \in S : y < x\}$, and $G = \{y \in S : y > x\}$. From this, we know that $\text{rank}(x) = |L| + 1$.
- If $\text{rank}(x) = i$, then we're done. If $\text{rank}(x) > i$, find the element of rank $i$ in $L$. Otherwise, find the element of rank $i - |L| - 1$ in $G$.

The main idea was that we could pick $x$ that was always 3/4-balanced, to guarantee constant reduction for each subtask and consequently linear run time. If we were to remove this step and instead pick $x \in S$ randomly, we would still have a correct algorithm, but the run time would pick up a non-zero probability of not being linear.

The expected run time becomes:

$$T(n) = T\left(\frac{3}{4}n\right) + (\mathbb{E}[\#\text{ iterations}] + 1) \cdot cn,$$

where the expected number of iterations refers to the expected number of iterations it takes before $x$ is 3/4-balanced (with respect to the most recent subtask for which

$x$ was 3/4-balanced). Since $\mathbb{P}[X \text{ is } 3/4-\text{balanced}] \geq 1/2$, $\mathbb{E}[\#\text{ iterations}] \leq 2$, so our expected runtime is still linear.

# 4 February 16, 2023

## 4.1 Union Find

The goal of this data structure is to maintain a dynamic collection of pairwise disjoint sets $S = \{s_1, \ldots, s_r\}$ with a single arbitrary representative per set, $Rep[s_i]$. Our goal is to be able to support these three methods:

- Make-Set($x$): add set $\{x\}$ to the set of sets, with $x$ as a representative. This is an initialization method.
- Find-Set($x$): if $s(x)$ is the set containing element $x$, this method returns $Rep[s(x)]$.
- Union($x, y$): let $s(x)$ and $s(y)$ be the two sets containing $x$ and $y$ (possibly equal). This method replaces both sets with $s(x) \cup s(y)$ with a single representative.

**Example 4.1**
Use doubly-linked lists to implement this data structure.

- Make-Set(x): Runtime: $O(1)$.
- Find-Set(x): Iterate through $s(x)$ until the leader is found. Runtime: $O(L) \in O(n)$, where $L$ is the length of $s(x)$.
- Union(x,y): Link together $s(x)$ and $s(y)$. Runtime: $O(L) \in O(n)$.

Consider the following scenario: we call Make-Set for $n$ elements $a_1, \ldots, a_n$. Then, we call Union$(a_1, a_2)$, Union$(a_1, a_3), \ldots,$ Union$(a_1, a_n)$. Since we are traversing a list of growing size, the total runtime of these $n$ operations is

$$\Theta(1) + \ldots + \Theta(n) = \Theta(n^2).$$

**Example 4.2**
Optimization 1.1: create a pointer from each element to the head of the set. By maintaining these pointers, we can retrieve leaders in constant time.

This changes the runtime of FIND-SET($x$) to constant time. However, UNION($x, y$) remains bad, since we have to replace the "leader" pointers for each element in the merged list.

**Example 4.3**
Optimization 1.2: maintain the size of each set. During the UNION operation, only join lists that are shorter to lists that are longer.

**Lemma 4.4**
Using this optimization, the amortized time of UNION is $O(\log n)$. In other words, any sequence of $m \geq n$ operations that contains $k$ union operations runs in time $O(m + k \log n)$.

*Proof.* Focus on a single element $u$. Each time it is merged with another element $x$, we only update its leader (which comes with cost $O(L)$) when Length($x$) $\geq$ Length($u$), i.e., Length($u$) doubles. If we merge $u$ with a smaller element, it comes with no cost associated with the element $u$. Since we can only double the length of the set containing $u$ at most $\log n$ times, the total cost of all union operations is bounded by $O(n \log n)$. □

Let's change our abstraction. Instead of thinking of sets as doubly-linked lists, let's think of them as trees. For the union operation, update the "leader" pointer of only the leader of the merged set. For the find operation, traverse upwards until the leader is found.

**Example 4.5**
Optimization 2.1: Define the **rank** of a leader $Rep[s(x)]$ to be its number of children. Then, only join trees with smaller ranks to trees with larger ranks.

**Example 4.6**

Optimization 2.2: **path compression**. During each FIND-SET operation, redirect the parent pointer of each visited node to the leader.

The idea behind this optimization is to flatten the tree. Each time we call FIND-SET, we take all traversed nodes and reposition them so that they are adjacent to the leader node.

**ALGORITHM 1:** FIND-SET

```
if x ≠ x.parent then
    x.parent = FIND-SET(x.parent)
return x.parent
```

**ALGORITHM 2:** UNION

```
Ū = FIND-SET(U)
V̄ = FIND-SET(V)
if Ū = V̄ then
    return
if Ū.rank = V̄.rank then
    Ū.rank = Ū.rank + 1
    V̄.parent = Ū
else if Ū.rank > V̄.rank then
    V̄.parent = Ū
else
    Ū.parent = V̄
```

- Claim 1: Optimization 2.1 gives $O(\log n)$ UNION and FIND-SET operations. The proof for this is the same as it was for the linked-list representation.
- Claim 2: Optimization 2.2 gives $O(\log n)$ amortized. We will prove this when we discuss amortized analysis in a bit. (Example 4.10).
- Claim 3: Both optimizations together produces $O(\alpha(n))$, which is the **inverse ackermann** function. This can essentially be thought of as constant time. For example, $\alpha(10^{80}) \leq 4$. We will not be proving this.

## 4.2 Amortized Analysis (Queue using Two Stacks)

A queue can be implemented using two stacks, $s_1$ and $s_2$, in the following way:

- Enqueue(x): push $x$ onto $s_1$
- Dequeue(): If $s_2$ is empty, pop elements in $s_1$ until $s_1$ is empty, then push them onto $s_2$. Then, pop a single element off of $s_2$.

The Enqueue operation is always constant. The Dequeue operation is not always constant, e.g., when $s_1$ is full, but is constant amortized. Here are three ways to show that the amortized cost is constant.

**Definition 4.7**

**Aggregated** amortized time analysis is the most straightforward to understand (but often hard to compute in practice). It computes the total cost of $n$ operations by directly summing the cost of each operation.

Consider the cost of the $i$th call to Dequeue. Suppose $n_i$ is the total number of calls to Enqueue that were made before the current Dequeue. The total cost is at most $O(n_i)$, since it takes $O(n_i)$ to pop everything off of $s_1$, $O(n_i)$ to push everything onto $s_2$, and possibly some extra cost to pop off the element itself. Note that the total number of enqueues is $O(n)$, so the total cost of all dequeues is $O(n_1+\ldots+n_k) \in O(n)$. Therefore, the amortized cost of Dequeue is $O(1)$.

**Definition 4.8**

The **potential** amortized time analysis assigns each operation with a new cost via a potential function. The main insight is that if we can choose a potential function to "balance out" expensive operations, our newly assigned costs can all be bounded by our desired amoritzed cost.

For each of the $n$ operations, let their actual costs $c_1,\ldots,c_n$. Our goal is to compute the amortized cost $\sum c_i/n$. For each operation, defined its amortized cost to be

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}),$$

where $\Phi$ is a potential function mapping each state of the process $D_i$ to a real

number. Note that

$$\sum \hat{c_i} = \sum c_i + (\Phi(D_n) - \Phi(D_0)).$$

Therefore, as long as $\Phi(D_i) \geq \Phi(D_0)$ for all $i$, we can say that $\sum \hat{c_i} \geq \sum c_i$, and compute an upper bound on the actual total cost of all operations.

As some general intuition, our goal is to pick a potential function that makes $\hat{c_i}$ as close as possible to the actual amortized cost that we want to prove. For example, in the queue with two stacks example, we want to show that dequeue is $O(1)$ amortized, so we want to pick a potential function such that all $\hat{c_i}$ are constant time. We know that the most expensive operations for dequeue are related to the number of elements in $s_1$, so we are motivated to choose a potential function that offsets this heavier cost.

**Example 4.9**

Proof that the DEQUEUE method runs in $O(1)$ amortized, using the potential method.

Let $\Phi(D_i)$ be $2|s_1|$ after the $i$th operation has completed. For each call to ENQUEUE, the actual cost is 1 for a simple push operation, and the size of $s_1$ increases by 1 element, so

$$\hat{c_i}^{\text{ENQUEUE}} = 1 + \Delta\Phi = 3.$$

For each call to DEQUEUE, the actual cost is $2|s_1| + 1$, since it costs $|s_1|$ to pop every element in $s_1$, $|s_1|$ to push them all back onto $s_2$, and an additional 1 to pop the last element from $s_2$. Also, the change in potential is $-2|s_1|$. So,

$$\hat{c_i}^{\text{DEQUEUE}} = 2|s_1| + 1 + \Delta\Phi = 1.$$

In both cases, $\hat{c_i}$ is constant. Moreover, since $\Phi(D_0) = 0$, it must be the case that $\Phi(D_i) \geq \Phi(D_0)$ for all $i$. Together, this implies that the sum of the actual costs is $O(n)$, hence we have $O(1)$ amortized as desired.

**Example 4.10**

Proof that path compression results in $O(\log n)$ amortized time for both FIND-SET and UNION in the union-find data structure, using the potential method.

We will show this using the potential function

$$\Phi(D_i) = \sum_{u \in D_i} \log(u.size),$$

where $u.size$ is the number of nodes in the subtree rooted at $u$. Note that this sum only iterates over nodes which are included in at least one set ($u \in D_i$). In particular, $\Phi(D_0) = 0$, so this is a valid potential function to use, since $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all $i$.

First, consider the Find-set operation. If we call the operation on $v_0$, we will follow a path $v_0, v_1, \ldots, v_k = Rep[S(v_0)]$, so the actual cost is $k$. For each vertex in this path, we redirect its leader pointer to $v_k$. Therefore, for each $1 \leq i \leq k-1$, the updated size of $v_i$ is $v_i.size - v_{i-1}.size$, since each node is losing all of the children of its child when it is redirected. So, our amortized cost is given by

$$\begin{aligned}\hat{c_i}^{\text{Find-set}} = k + \Delta\Phi &= k + \sum_{i=1}^{k-1} \log\left(\frac{v_i.size - v_{i-1}.size}{v_i.size}\right) \\ &= 1 + \sum_{i=1}^{k-1}\left[1 + \log\left(\frac{v_i.size - v_{i-1}.size}{v_i.size}\right)\right].\end{aligned}$$

When $v_i.size > 2 \cdot v_{i-1}.size$, the summand is non-negative and bounded above by $1+\log(1) = 1$. Otherwise, the summand is negative. However, along the chain, there can be at most $\log(n)$ doublings, since the total number of nodes is bounded above by $n$. Therefore, there are at most $\log(n)$ non-negative terms in the summand, each $\leq 1$, so $\hat{c_i}^{\text{Find-set}} \in O(\log n)$.

Next, consider the Union operation. This operation consists of two find-set operations, then a constant-time join operation, so the amortized cost is

$$\hat{c_i}^{\text{Union}} = 2\hat{c_i}^{\text{Find-set}} + c_{join} + \Delta\Phi_{join} = 2\hat{c_i}^{\text{Find-set}} + \Delta\Phi_{join}.$$

Since Find-set is $O(\log n)$ amortized, it suffices to compute $\Delta\Phi_{join}$:

$$\Delta\Phi_{join} = \log\left(\frac{x.size + y.size}{x.size}\right) \leq \log\left(\frac{n}{x.size}\right) \in O(\log n).$$

So, the amortized cost for Find-set is also $O(\log n)$ as desired.

# 5 February 23, 2023

## 5.1 Competitive Analysis and Self-organizing Lists

Think of a self-organizing list as a singly-linked list.

The method Access($x$) finds and returns the element with key $x$. The cost of Access($x$) is $\textsc{Rank}_L(x)$.

After accessing, we reorder $L$ via transposing adjacent elements. The cost of reordering is equal to the number of transpositions.

Consider a sequence of operations on elements $S = \{x_1, \ldots, x_N\}$. The cost of operation $i$ is the sum of its axis cost and reordering cost. Say that the reordering cost of operation $i$ is $t_i$. Then, the total access cost and total reordering cost for an arbitrary sequence of $N$ operations is

$$C_A(S) = \sum_{i=1}^{N} \textsc{Rank}(x_i) + \sum_{i=1}^{N} t_i.$$

**Definition 5.1**

An **offline algorithm** is an algorithm that knows what operations it needs to perform in advance. An **online algorithm** does not.

Say we have $L$ with these keys:

$$1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4$$

Say we design the (online) algorithm so that we always move the previously accessed element to the front. Then, we can design a sequence of inputs such that it still costs $n$ per operation: $S = \{4, 3, 2, 1\}$. In general, we won't be able to do better than this for the online version of the algorithm.

**Definition 5.2**

Let OPT be an optimal offline algorithm. Let $A$ be an online algorithm. $A$ is **$\alpha$-competitive** if there exists a constant $c$ such that for any access sequence $S$,

$$c_A(S) \leq \alpha \cdot c_{\text{OPT}}(S) + c.$$

In some cases, even the efficiency of the offline algorithm OPT cannot be better than the online algorithm. For example, when we have $L$:

$$1 \longrightarrow 2 \longrightarrow \ldots \longrightarrow n,$$

accessing $S = \{n, n-1, \ldots\}$ is $\Omega(n^2)$, even for OPT. For example, consider the cost of the first $n/3$ operations. For each of these operations, we can choose to transpose them beforehand to get a lower cost, or access them directly. Both of these options are $\Omega(n)$, so the cost of the first $n/3$ operations is $\Omega(n^2)$.

**Theorem 5.3**
Consider the **MTF** (move to front) strategy: after accessing an element, move it to the front. MTF is 4-competitive.

We will prove this theorem using potential function analysis, like we used during recitation. Recall:

- We want to define a potential function $\Phi$ taking the state of the data structure to some real number. Let $D_i$ be the state of the data structure after update $i$. We require $\Phi(D_i) \geq \Phi(D_0)$.
- We define an imaginary (amortized) cost:

  $$\hat{c}_i = c(i) + \Delta\Phi,$$

  where $c(i)$ is the true cost of operation $i$.
- In sum,

  $$\sum_{i=1}^{|S|} c(i) \leq \sum_{i=1}^{|S|} \hat{c}(i),$$

  completing our amortized analysis.

*Proof.* Both MTF and OPT start with the same list $L_0$, and sequence of operations $S = \{x_1, \ldots, x_N\}$. After $\textsc{Access}(x_i)$, MTF has list $L_i$, and OPT has list $L_i^*$. Now, consider the cost of operation $i+1$.

- For MTF, the total cost is

  $$\textsc{Rank}_{L_i}(x_{i+1}) + \textsc{Rank}_{L_i}(x_{i+1}) - 1 = 2\textsc{Rank}_{L_i}(x_{i+1}) - 1.$$

- For OPT, the total cost is

$$\textsc{Rank}_{L_i}(x_{i+1}) + t_{i+1},$$

where $t_{i+1}$ is the number of transpositions that the algorithm performs.

Define our potential function as follows:

$$\Phi(i) = 2 \cdot |\{(x, y) : x <_{L_i} y, x >_{L_i^*} y\}|.$$

This is a valid potential function, since $\Phi(i) \geq 0 = \Phi(0)$ for all $i$.

Now, after the $i$th operation concludes, let $A_{i+1}$ be the set of all keys before $x_{i+1}$ in both $L_i$ and $L_i^*$; let $B_{i+1}$ be the set of all keys after $x_{i+1}$ in $L_i$, and before $X_{i+1}$ in $L_i^*$; let $C_{i+1}$ be the set of all keys after $x_{i+1}$ in $L_i^*$, and before $x_{i+1}$ in $L_i$; and let $D_{i+1}$ be the set of all keys after $x_{i+1}$ in both $L_i$ and $L_i^*$.

Let's consider $\Delta\Phi$. During the operation changing $L_i$ to $L_{i+1}$, $x_{i+1}$ is moved to the first element in the list. This creates $|A_{i+1}|$ new inversions, and removes $|C_{i+1}|$ inversions. During the operation changing $L_i^*$ to $L_{i+1}^*$, $x_{i+1}$ is transposed $t_{i+1}$ times, which creates at most $t_{i+1}$ new inversions. Therefore,

$$\Delta\Phi \leq 2(|A_{i+1}| - |C_{i+1}| + t_i).$$

We also know that $\textsc{Rank}_{L_i}(x_{i+1}) = |A| + |C| + 1$, and $\textsc{Rank}_{L_i}(x_{i+1}) = |A| + |B| + 1$. So, we may say

$$\begin{aligned}
\hat{c}_{i+1}^{MTF} &= 2\textsc{Rank}_{L_i}(x_{i+1}) - 1 + \Delta\Phi \\
&\leq 2(|A_{i+1}| + |C_{i+1}| + 1) - 1 + 2(|A| - |C| + t_{i+1}) \\
&\leq 4(|A_{i+1} + 1 + t_{i+1}|),
\end{aligned}$$

while

$$c_{i+1}^{OPT} = |A_{i+1}| + |B_{i+1}| + 1 + t_{i+1} \geq |A_{i+1}| + 1 + t_{i+1}.$$

Thus,

$$\sum_i c_i^{MTF} \leq \sum_i \hat{c}_i^{MTF} \leq 4 \sum_i (|A_i| + 1 + t_i) \leq 4 \sum_i c_i^{OPT},$$

which completes the proof. □

**Example 5.4**

Here is some motivation for the potential function that we used to prove the previous theorem. Consider the reordering cost of $L$ into $L^*$:

- The minimum number of transpositions needed to turn $L$ into $L^*$ is at least the number of inversions. If $x$ and $y$ are inverted, there is no transposition that can swap $x$ and $y$ besides the transposition that swaps $x$ and $y$ directly.
- There is a sequence of $v$ transpositions that fixes all $v$ transpositions. Fix each element in reverse order, and it works.

Since the potential function counts the number of inversions, it approximately captures how far apart $L$ and $L^*$ are after a given operation.

# 6 February 28, 2023

Today we will talk about Hashing. We want to support a dictionary $T$ that stores $(k, v)$ key value pairs, where all keys $k \in \mathcal{U}$ for some universe $\mathcal{U}$. Let $n$ be the number of objects in $T$, and $m = |T|$.

We want to support three methods:

- Insert($x$): insert $x = (x.\text{Key}, x.\text{Value})$ into the dictionary
- Delete($x$): delete $x = (x.\text{Key}, x.\text{Value})$ from the dictionary
- Search($k$): search for the key $k$, i.e., return $x.\text{Value}$ if $x.\text{Key} = k$.

## 6.1 Direct Addressing

Make $m = |\mathcal{U}|$. Then, we could insert, delete, and search in constant time. Unfortunately, the space complexity here is horrible, since $|\mathcal{U}| \gg n$ most of the time.

## 6.2 "Solution Zero"

- Use a list (e.g., linked) as the dictionary. In the worst case, Insert and Delete are $O(1)$, while Search is $O(n)$.

- Use some sort of tree as the dictionary. In this case, we can achieve $O(\log m)$ for all three methods.

## 6.3 Hashing

Hashing allows us to shrink $\mathcal{U}$. We start with a hash function

$$h : \mathcal{U} \to M = \{0, \ldots, m-1\}.$$

Assuming that we restrict $m \in O(n)$, this is highly compressed. Inevitably, there will exist two keys $k, k'$ such that $h(k) = h(k')$, which is called a **collision**. There are many different ways to deal with collisions.

**Example 6.1**

One way to deal with collisions is **chaining**.

In chaining, for each available space in the dictionary, instead of storing a single value, store another data structure, like a list (or even another dictionary). So, when two keys collide, insert them into the same data structure.

The space complexity of this solution is $O(m+n)$. Assuming that hashing is constant, and inserting into our list is constant, insertion takes $O(1)$. Deletion can also take $O(1)$. Searching takes $O(|L|)$, where $L$ is the length of the list stored at a particular key value.

**Definition 6.2**

Define **load factor** as $\alpha = n/m$, which is the ratio of the number of items we are trying to store in our dictionary, and its total space.

Note $\mathbb{E}[|L|] = \alpha$. In a **random oracle** process, we make a hash function that assigns objects to lists at random and independently. It is not possible to design an efficient random oracle, but we ignore this for now.

Using the random oracle, we intuitively keep the lengths of all of the lists short. In fact, by the Chernoff bounds, if $n = \Theta(m)$, the maximum load of any bin is $O(\log m/(\log \log m))$ with high probability.

## 6.4 Universal Hashing

**Definition 6.3**

A hash family $\mathcal{H} = \{h : \mathcal{U} \to M\}$ is **universal** if for all $k, k' \in \mathcal{U}$,

$$\mathbb{P}[h(k) = h(k')] \leq \frac{1}{m}.$$

Note that this probability is taken over all hash functions in $\mathcal{H}$. The motivation for this definition is that the bound is tight for the random oracle, i.e., $\mathbb{P}[\mathrm{RO}(k) = \mathrm{RO}(k')] = 1/m$.

Consider the expected length of each list in universal hashing. For $j \neq j'$, let $X_{j,j'} = \mathbb{1}(h(k_j) = h(k_{j'}))$. Assuming that our hash family is universal, $\mathbb{E}[X_{j,j'}] \leq 1/m$. By linearity of expectation, the number of keys colliding with $k_j$ has expected value

$$\sum_{j \neq j'} \mathbb{E}[X_{j,j'}] \leq \frac{n-1}{m},$$

so the expected length of the list with $k_j$ is $1 + (n-1)/m < 1 + \alpha$.

**Example 6.4**

Construct a universal hash function.

Let $r = \log_m |\mathcal{U}|$. Write all keys in base $r$, i.e., for any key $k \in \mathcal{U}$, express it as some $a \in M^r$, $a = (a_0, a_1, \ldots, a_{r-1})$. Now, define our family of hash functions

$$h_a(k) = \langle a, k \rangle = \sum_{\ell=0}^{r-1} a_\ell d_\ell^{(k)} \pmod{m},$$

where $k = (d_0^{(k)}, d_1^{(k)}, \ldots, d_{r-1}^{(k)})$.

**Theorem 6.5**

This hash function is universal.

*Proof.* For any two $k, k' \in \mathcal{U}$, we want to show

$$\mathbb{P}_{a \in_R M^r}[\langle a, k \rangle = \langle a, k' \rangle] \leq \frac{1}{m}.$$

Since $k \neq k'$, let $\ell^*$ be an index such that $d_{\ell^*}^{(k)} \neq d_{\ell^*}^{(k')}$. Now, $\langle a, k\rangle = \langle a, k'\rangle$ if and only if

$$\begin{aligned}
\sum_{\ell=0}^{r-1} a_\ell d_\ell^{(k)} \pmod m &\equiv \sum_{\ell=0}^{r-1} a_\ell d_\ell^{(k')} \pmod m \\
\iff \sum_{\ell=0}^{r-1} a_\ell \left(d_\ell^{(k)} - d_\ell^{(k')}\right) &\equiv 0 \pmod m \\
\iff a_{\ell^*}\left(d_{\ell^*}^{(k)} - d_{\ell^*}^{(k')}\right) &\equiv -\sum_{\ell\neq\ell^*} a_\ell \left(d_\ell^{(k)} - d_\ell^{(k')}\right) \pmod m \\
\iff a_{\ell^*} = -\left(d_{\ell^*}^{(k)} - d_{\ell^*}^{(k')}\right)^{-1} &\sum_{\ell\neq\ell^*} a_\ell \left(d_\ell^{(k)} - d_\ell^{(k')}\right) \pmod m,
\end{aligned}$$

Assuming that we are working with a "nice" $m$, i.e., some prime number, the probability that this is true is exactly $1/m$, since everything is chosen randomly. □

# 7 March 2, 2023

Review from last time: we want to create methods Insert, Delete, Search. The keys $k \in \mathcal{U}$, the number of objects is $n$, the size of the dictionary $T$ is $m$, and the load factor $\alpha = n/m$.

## 7.1 Open Addressing

Open addressing is another way to address collisions. In open addressing, we guarantee a different hash for every different element. In this case, we require $n \leq m \implies \alpha \leq 1$.

For each $k \in \mathcal{U}$, define a **probing sequence** $\sigma(k) = \{i_0^{(k)}, i_1^{(k)}, \ldots, i_{m-1}^{(k)}\}$ that we look through to assign keys to different elements. The probing sequence is a permutation of $m$. Now, model this probing sequence as an output to a hash function $h : \mathcal{U} \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$:

$$h(k, p) = i_p^{(k)},$$

where $p$ is called the **probe number**.

For each method, we still start with a normal hash function $h(k) = i_0^{(k)}$. To insert,

keep incrementing $p$ until $T[h(k,p)]$ until it is empty or marked deleted, and insert the element. To delete, keep incrementing $p$ until $T[h(x.\textsc{Key}, p)] = x.\textsc{Val}$, and mark it deleted.

- Linear probing:
$$h(k,p) = (h(k) + c \cdot p) \pmod{m}.$$
Clustering is a problem with linear probing. For elements that hash to keys that are closer together, the searches start to overlap heavily.
- Double hashing:
$$h(k,p) = (h_1(k) + p \cdot h_2(k)) \pmod{m},$$
for two hash functions $h_1, h_2$. Double hashing gets us closest to the uniform hashing assumption.

**Definition 7.1**
The **uniform hashing assumption** says that for all $k \in \mathcal{U}$, the probe sequence $\sigma(k)$ is a independent and uniformly random permutation.

Let $X$ be the number of steps it takes before a hash is successful. Under the uniform hashing assumption, $\mathbb{P}[X \geq k] = \alpha^{k-1}$, since each new part of the sequence is filled with probability $\alpha$. Therefore, the expected runtime for each hash is

$$O(1 + \alpha + \ldots) \in O(1/(1-\alpha)).$$

## 7.2 Perfect Hashing

**Definition 7.2**
A **static dictionary** is a dictionary that only supports the Search method. All inserts are performed beforehand during a preprocessing phase.

A perfect hashing scheme is a hashing scheme that ensures no collisions, which guarantees $O(1)$ search time. One way to achieve this is to use solution zero (i.e., chaining with a list), but to implement the "list" as another dictionary with hashing. So, there are two levels of hashing:

- $h_1(k)$ is a slot in $T$
- If $h_1(k) = i$, the final location for $k$ is $h_{2,i}(k)$.

**Theorem 7.3**

Let $\mathcal{H} = \{h : \mathcal{U} \to M\}$ universal. If we pick $h \in_R \mathcal{H}$ and hash $n$ keys with $m \geq n^2$, then the probability of a collision is $< 1/2$.

*Proof.* Proof by union bound. The probability $P$ satisfies

$$P \leq \binom{n}{2}\frac{1}{n^2} < \frac{1}{2}.$$

□

In principle, we could use this fact to preprocess hash functions until there are no collisions; in expectation, this would only require two random choices. The problem with doing this is the requirement that $m \geq n^2$, leading to quadratic space complexity. Using chaining with a second hash function allows us to circumvent this.

Let $h_1$ map $N_i$ keys to the $i$th mini dictionary. Our goal is to achieve

$$\sum N_i^2 = O(n),$$

so that our space is still linear. To compute the probability that this is true, we can compute its expected value, and then apply Markov's inequality. By linearity of expectation,

$$\begin{aligned}\mathbb{E}\left[\sum N_i^2\right] &= \sum_{(k_1,k_2)\in\mathcal{U}^2} \mathbb{P}(h_1(k_1) = h_2(k_2)) \\ &= n + \sum_{k_1\neq k_2} \mathbb{P}(h_1(k_1) = h_2(k_2)) \\ &\leq n + \frac{n(n-1)}{m} < 2n,\end{aligned}$$

by the property of universal hashing, assuming that $m \geq n$. This implies that, $\mathbb{P}[\sum N_i^2 > 4n] < 1/2$ by Markov's inequality. So, in expectation, we only need to try to pick $h_1$ twice before it is "good enough".

# 8 March 7, 2023

## 8.1 Spanning Trees

**Definition 8.1**

A **spanning tree** of $G = (V, E)$ is a subgraph $(V, E_T)$ which is a connected tree.

Spanning trees are not necessarily unique! An easy way to find a spanning tree is to use DFS, which has linear runtime.

Observation about any spanning tree $T$ of $G$: for every non-tree edge $(x, y)$, there exists a unique path $P_{(x,y)}$ from $x$ to $y$ in $T$. $P_{(x,y)} \cup \{(x, y)\}$ forms a cycle, which is called the fundamental cycle of $(x, y)$.

**Lemma 8.2** (Cut and Swap Property of Spanning Trees)

For any $G = (V, E)$ and spanning tree $T$, non-tree edge $(x, y)$, and tree edge $e \in P_{(x,y)}$, the graph $T' = T \cup \{(x, y)\} - \{e\}$ is also a spanning tree.

*Proof.* Consider the unique path $P$ with endpoints $u, v$ in $T$. If $e \notin P$, then $P$ is in $T'$, so $P$ also connects $u, v$ in $T'$.

Otherwise, $e \in P$. Let $C_{(x,y)} = P_{(x,y)} \cup \{(x, y)\}$. WLOG, the distance from $u$ to $x$ is smaller than the distance from $u$ to $y$. Then $P_{(u,x)} \cup (C_{(x,y)} - e) \cup P_{(y,v)}$ is a path from $u$ to $v$ in $T'$. So, we are done. □

Note that this operation preserves the number of edges, which makes sense, since trees always have $|V| - 1$ edges.

## 8.2 Minimal Spanning Trees (MST)

**Definition 8.3**

Given graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$, a **minimal spanning tree** of $G$ is a spanning tree of minimum weight.

Like the normal spanning tree, MSTs are not necessarily unique, due to the possibility of different edges having the same weights. But, given that all edges have different edges, it turns out that there is a unique MST.

A general greedy approach to finding a minimum spanning tree might look as follows. First, initialize a set of edges $A = \emptyset$. While $A$ is not a spanning tree, find a "safe" edge to add to $A$.

When $A = \emptyset$, the min weight edge $e^* = (a, b)$ is safe. Let $T$ be a minimal spanning tree. If $e^* \in T$, then we are done. Otherwise, we can use the cut and swap property with another edge on the path $P_{(a,b)}$ to achieve another spanning tree $T'$ with $w(T') \leq w(T) \implies w(T') = w(T)$, and $T'$ is also an MST.

**Definition 8.4**

A **cut** of $G = (V, E)$ is a partition of $V$ into $(S, V - S)$. An edge $(x, y) \in E$ **crosses cut** $(S, V - S)$ if $x \in S$ and $y \in V - S$. A cut $(S, V - S)$ **respects** a set of edges $A \subseteq E$ if no edge of $A$ crosses the cut. An edge $(x, y) \in E$ is a **light edge** for cut $(S, V - S)$ if it crosses the cut and has minimum weight among all edges crossing the cut.

**Theorem 8.5**

Let $A \subseteq E$, where $A$ is a proper subset of the edges of some MST. Let $(S, V - S)$ be a cut that respects $A$. Let $e$ be a light edge for the cut. Then $e$ is safe for $A$.

*Proof.* We will prove this using the cut and swap property. Let $T$ be some MST containing $A$.

Say $e = (a, b) \notin T$. Let $P_{(a,b)}$ be path in $T$ from $a$ to $b$. Note that $T$ must cross the cut, since $a$ and $b$ are in different parts of the cut. Let $e'$ be an edge in $P$ that crosses the cut. By the cut and swap property, we can replace $e'$ with $e$ to obtain another spanning tree $T'$. Since $w(e') \leq w(e)$, $w(T') \leq w(T) \implies w(T') = w(T)$, so $T'$ is also a minimal spanning tree. □

Now, we have a general approach for picking safe edges:

**Algorithm 3:** Meta-Greedy MST

```
A = ∅
while A is not a spanning tree do
    Pick cut (S, V − S) that respects A
    Let e be a light edge for the cut
    A = A ∪ {e}
return A
```

Note that the loop should run exactly $|V|-1$ times, since this is the final size of $A$. There are many ways to pick these cuts efficiently:

- **Prim's Algorithm**: $A$ is a tree of isolated vertices. Pick the cut $(V(T_A), V - V(T_A))$. The implementation for Prim's is very similar to Dijkstra, since we are essentially just adding the min-weight edge adjacent to $A$, which we can keep track using a priority queue.
- **Kruskal's Algorithm**: $A$ is a forest of trees. Pick $e$ to get the min weight edge connecting 2 different trees. This can be implemented using the union find data structure.

## 8.3 Implementation details Kruskal's

In Kruskal's we construct the MST by building up a forest of smaller trees and connecting them until they eventually become a single unified tree. First, we sort the edges by decreasing weight. Then, we add edges greedily as long as they cross cut, i.e., connect two different trees. Correctness comes from the previous facts that we proved about light edges (we always add light weight edges due to the sorting).

**Algorithm 4:** Kruskal's MST

```
A = ∅
For all v ∈ V, Make-Set(v)
Sort E in non-decreasing order
for e = (u, v) ∈ E do
    if Find(u) ≠ Find(v) then
        A = A ∪ (u, v)
        Union(u, v)
return A
```

The runtime of this algorithm is dominated by the sorting, $O(|E|\log|E|)$. The cost of the loop is $O(|E|\alpha(|V|))$, since the number of union and finds is bounded by the number of edges.

## 8.4 Implementation details Prim's

For Prim's, we construct our MST by maintaining a single smaller tree, and at each step adding a new edge until the tree eventually becomes an MST. The idea is that we should always add the lowest weight edge adjacent to our current tree, since this is a light edge (and hence correctness comes from the ideas we proved earlier).

There are a few different ways to implement Prim's. The naive way is to, for each step, iterate through all edges and choose the lowest weight available edge adjacent to our current tree.

**Algorithm 5:** Prim's MST, slow

```
A = ∅
Pick starting vertex s ∈ V, set V(A) = {a}
while A is not a spanning tree do
    Initialize a min-weight edge e*
    for e = (u, v) ∈ E do
        if w(e) < w(e*) and u ∈ V(A), v ∉ V(A) then
            e* = e
    Add e* to A
return A
```

The runtime is $O(|V||E|)$, since the loop "while $A$ is not a spanning tree" runs $O(|V|)$ times (we add exactly $|V|-1$ edges until $A$ becomes a spanning tree). An easy way to speed this up is to iterate over vertices instead of edges, by keeping track of the distances to the tree:

**Algorithm 6:** Prim's MST, dense graphs

```
1  A = ∅
2  Init D, set of distances from the tree
3  Pick starting vertex s ∈ V, set V(A) = {a}
4  while A is not a spanning tree do
5      Initialize a min-weight edge e* for v ∈ V do
6          if D[v].dist < w(e*) then
7              e* = (D[v].parent, v)
8      Add e* to A
9      for v adjacent to the newly added vertex v* in e* do
10         if w((v, v*)) < D[v].dist then
11             D[v].dist = w(v)
12             D[v].parent = v*
13 return A
```

The runtime is $O(|V|^2)$, since we make two linear traversals through the vertices for each added edge. Note that the runtime of Kruskal's is $O(|E|\log|E|)$. **This version of Prim's is more desirable than Kruskal's for dense graphs, i.e., when** $|E| \in O(|V|^2)$. For example, see here (USACO).

The last, most efficient but hardest to implement, version of Prim's is to implement it with a Min-Prio queue, like how we implement Dijkstras. This is similar in idea to the first version (Algorithm 5). Instead of iterating all edges, we keep track of the closest ones with a Min-Prio queue:

**Algorithm 7:** Prim's MST, PQ

```
A = ∅
Pick starting vertex s ∈ V
Initialize Min-Prio Queue PQ
for v ∈ V do
    v.key = ∞
s.key = 0
while A is not a spanning tree do
    v = PQ.top()
    Add v to V(A)
    If not the first iteration, add (v, v.parent) to E(A)
    for u adjacent to v do
        if w((u, v)) < u.key then
            u.key = w((u, v))
            u.parent = u
return A
```

The outer loop runs $O(|V|)$ times. Extracting the minimum with a normal heap takes $O(\log|V|)$, while updating the queue in place can be done in $O(1)$ using fibonacci heap. The inner loop performs $O(|E|)$ of these updates (across the entire algorithm), so the final runtime is $O(|E|+|V|\log|V|)$.

Without fibonacci heap, updates take $O(\log|V|)$, giving a final runtime of $O(|E|\log|V|+|V|\log|V|) \in O(|E|\log|V|)$. This is the the same performance as Kruskal's algorithm.

# 9 March 9, 2023

## 9.1 Max Flow Problem

The setup for this problem:

- directed graph $G = (V, E)$
- a source vertex $s \in V$
- a sink vertex $t \in V$

- a max capacity for each edge, represented by a function $c : E \to \mathbb{R}^{\geq 0}$. we additionally say that $c(u,v) = 0$ for all $(u,v) \notin E$.

The question we want to answer is approximately as follows. Each directed edge represents a one-way road from one place to another. The capacity of each edge represents the number of lanes that this road has. We want to find the maximum rate of traffic that can flow from $s$ to $t$.

**Definition 9.1**
**Gross Flow**.

Define the "gross flow" $g : E \to \mathbb{R}^{\geq 0}$ be such that $g(u,v)$ denotes the amount of flow on the edge $(u,v)$. A valid flow satisfies

- Feasability:
$$0 \leq g(u,v) \leq c(u,v) \forall (u,v) \in E$$
- Flow conservation:
$$\sum_u (g(u,v) - g(v,u)) = 0 \forall v \neq s,t.$$
This quantity represents the net flow (in and out) from the vertex $v$.

**Definition 9.2**
**Net flow**.

This is the notation that we will primarily be using in this class. Define the "net flow" $f : V \times V \to \mathbb{R}$ (note that this can be negative) be such that $f(u,v)$ denotes the flow from $u$ to $v$. A valid flow satisfies

- Feasability:
$$f(u,v) \leq c(u,v) \forall u,v \in V$$
- Flow conservation:
$$\sum_u f(u,v) = 0 \forall v \neq s,t$$
As before, this represents the net flow (in and out) from the vertex $v$.

- Flow symmetry:

$$f(u,v) = -f(v,u) \forall v,u$$

**Definition 9.3**

The **value** of a flow $f$ is given by

$$|f| = \sum_v f(s,v).$$

Now, we can formally define the maximum flow problem.

**Definition 9.4** (Max Flow Problem)

Given a network $G = (V,E,s,t,c)$, we want to find $f^*$ such that $|f^*|$ is maximal.

Observe that any flow can be decomposed into a collection of $s-t$ paths and cycles. We can think of these as the elementary "building blocks" of flows.

**Lemma 9.5** (Flow decomposition lemma)

Let $\text{supp}_f(G)$ be the subgraph of $G$ of edges $(u,v)$ with $f(u,v) > 0$. Then, $\text{supp}_f(G)$ can be decomposed into a collection of flow paths and cycles.

**include proof?**

**Lemma 9.6**

$|f^*| > 0$ if and only if there exists an $s-t$ path $P$ in $G^+$, where $G^+$ is the subgraph of edges with positive capacities.

*Proof.* An $s-t$ path in $G^+$ implies positive flow from $s$ to $t$, hence $|f^*| = 0$. The other direction follows from the flow decomposition lemma; all cycles contributes 0 flow, so positive net flow implies that there exists a positive path from $s$ to $t$. □

Let

$$\hat{S} = \{v \in V : \exists s-v \text{ path in } G^+\}.$$

Note that $s \in \hat{S}$. If $|f^*| = 0$, meaning that there does not exist an $s-t$ path in $G^+$, then $t \notin \hat{S}$, i.e., $t \in V - \hat{S}$. This implies that $\hat{S}$ is an $s-t$ **cut** $(\hat{S}, V-\hat{S})$ which separates $s$ and $t$.

**Definition 9.7**

The **capacity** of a cut $(S, V-S)$ is

$$c(S) = \sum_{u \in S} \sum_{v \in V-S} c(u,v).$$

In other words, $c(S)$ is the total capacity of all edges that leaves $S$.

Given that $c(S) = 0$, there is no positive flow from $s$ to $t$, meaning that $(S, V-S)$ is an $s-t$ cut. In particular, $\hat{S}$ from above satisfies $c(\hat{S}) = 0$.

**Definition 9.8** (Minimum $s-t$ cut problem)

Given $G = (V, E, s, t, c)$, find an $s-t$ cut of minimum capacity.

Let

$$f(S) = f(S, S-V) = \sum_{u \in S} \sum_{v \in V-S} f(u,v).$$

In other words, $f(S)$ represents the net flow across the cut. Note that it must be true that

$$f(S) \leq c(S),$$

by feasibility.

**Claim 9.9**

$f(S) = f(S')$ for any $S, S'$ which are $s-t$ cuts.

*Proof.* Using the flow decomposition lemma, $f$ is a collection of flow cycles and $s-t$ flow paths. The main idea is that each flow cycle contributes 0 to $f(S)$ and $f(S')$, while each $s-t$ path contributes the same flow to both $f(S)$ and $f(S')$. □

By this claim, $|f| = f(\{s\}) = f(S)$ for any $s-t$ cut $(S, V-S)$. Thus:

**Lemma 9.10** (Weak Duality Principle)

The max flow is smaller than the minimum $s-t$ cut.

$$|f^*| = f^*(S^*) \leq c(S^*).$$

So, the max-flow algorithm:

- Start with $|f| = 0$
- As long as there exists an $s-t$ path in $G^+$, keep increasing $|f|$ by the flow along this path.

**Definition 9.11** (Residual Network)
A **residual network** $G_f = (V, E_f, s, t, c_f)$ of a flow $f$ in the network $G$ with residual capacities $c_f(u,v) = c(u,v) - f(u,v)$ and set of edges

$$E_f = \{(u,v) : c_f(u,v) > 0.\}$$

We need the extra edge definition in the cases when $f$ completely saturates edges, i.e., they can no longer be used in the residual network $G_f$.

Given that $f$ is a flow in $G$ and $f'$ is a flow in $G_f$, $f + f'$ is a flow in $G$. So, in order to improve a flow $f$ in $G$, it suffices to find a non-zero flow in $G_f$. If it is impossible to do so, this implies that there exists an $s-t$ cut $\hat{S}$ with $c_f(\hat{S}) = 0$.

**Proposition 9.12**
For any $s-t$ cut $S$, $c_f(S) = c(S) - f(S)$.

*Proof.*

$$c_f(S) = \sum_{u \in S} \sum_{v \in V-S} c_f(u,v) = \sum_{u \in S} \sum_{v \in V-S} c(u,v) - f(u,v) = c(S) - f(S).$$

□

Thus, if $c_f(\hat{S}) = 0$, we have $c(\hat{S}) = |f|$. But, by weak duality,

$$c(\hat{S}) = |f| \leq |f^*| \leq c(S^*) \leq c(\hat{S}),$$

implying that $f = f^*$, i.e., $f$ **is a max flow, and** $\hat{S}$ **is the min** $s-t$ **cut**.

**Theorem 9.13** (Max Flow - Min Cut Theorem)

$$|f^*| = c(S^*).$$

This is also known as the **strong duality** of flows and $s-t$ cuts.

# 10 March 14, 2023

## 10.1 Ford-Fulkerson

First, recap on some definitions from last time. A net flow $f : V \times V \to \mathbb{R}$ is a function satisfying:

- feasability: $f(u,v) \leq c(u,v) \forall u,v$
- flow conservation: $\sum_u f(u,v) = 0 \forall v \neq s,t$
- skew symmetry: $f(u,v) = -f(v,u)$

The residual network $G_f = (V, E_f, s, t, c_f)$ of a flow $f$ in graph $G$ has a different set of **residual capacities** $c_f(u,v) = c(u,v) - f(u,v)$, and **edges** $(u,v) \in E_f$ whenever $c_f(u,v) > 0$, so that saturated edges are disguarded.

**Definition 10.1**
An **augmenting** path in $G_f$ is a directed $s-t$ path in $G_f$. For any augmenting path $P$, define the **bottleneck capacity** $c_f(P) = \min_{(u,v)\in P}(c_f(u,v))$.

We saw last time that pushing $c_f(P)$ along any augmenting path increases $|f|$ by $c_f(P)$; and, if no such path exists, then we have found a max flow. So now, the **Ford Fulkerson algorithm**:

- start with $f \equiv 0$
- while an augmenting path $P$ exists in $G_f$, augment $f$ by $c_f(P)$
- return $f$

The correctness of this algorithm comes from this important theorem:

**Theorem 10.2** (Max Flow-Min Cut Theorem)
The following statements are equivalent:

(1) $|f| = c(S)$ for some $s-t$ cut $S$

(2) $f$ is a max flow

(3) $f$ admits no augmenting path, i.e., there is no $s-t$ path in $G_f$

(1) $\iff$ (2) implies that, if $S^*$ is the minimal $s-t$ cut, then $|f^*| = c(S^*)$. This is called the **strong duality** of flows and $s-t$ cuts. This implies the weak duality principle, i.e., that the max flow is smaller than the minimum $s-t$ cut.

(3) $\implies$ (2) implies that Ford-Fulkerson always works, since the algorithm keeps removing augmenting paths until no more exist.

Runtime:

- If capacities are integers bounded above by $C$, $c_f(P) \geq 1$ always. This implies that the number of augmentations $X$ satisfies:

$$X \leq |f^*| \leq c(\{s\}) \leq n \cdot C.$$

  With some dfs strategy, it takes $O(m)$ time to find an augmented path and perform the necessary augmentations. Therefore, an upper bound on the runtime is $O(mnC)$, which is **pseudo-polynomial**.

- If capacities are rational, the number of augmentations is bounded above by something least common multiples of the capacity of each edge adjacent to $s$. This still gives finite, pseudo-polynomial runtime.

- If capacities are real, it is possible to construct a graph that gives infinite runtime, since there is no longer the guarantee that some linear combination of edge weights can be an integer.

## 10.2 Optimizing Ford-Fulkerson (weakly polynomial)

Instead of picking any augmenting path, we can try choosing the augmenting path at each step "smartly".

**Definition 10.3**

The **maximum bottleneck path** is the augmenting path $P$ that maximizes the bottleneck capacity $c_f(P)$.

**Claim 10.4**

It is possible to choose the maximum bottleneck path in $O(m \log n)$.

*Proof.* Sort all edges by their residual capacity in $O(m \log m) \in O(m \log n)$. Then, perform a binary search on the bottleneck capacity through this list of edges; for each iteration, run a DFS to check whether or not it is possible to find an augmenting path with a given bottleneck capacity. Since each DFS is $O(m)$, and we run $O(\log m) \in O(\log n)$ iterations, the total runtime is $O(m \log n)$. □

**Claim 10.5**

In any graph $G_f$, there exists an $s-t$ path $P$ in $G_f$ with

$$c_f(P) = \min_{e \in P} c(e) \geq |f^*|/m,$$

where $|f^*|$ is the value of the max flow in $G_f$ (not the original graph $G$).

*Proof.* By the flow decomposition lemma, the graph with positive flow edges can be decomposed into paths and cycles. If we choose these paths and cycles such that each completely saturates at least 1 edge, the number of such paths/cycles $\leq m$.

Thus, we have $\leq m$ paths from $s$ to $t$ that partition the flow in $G_f$. Since the max flow is $|f^*|$, this implies that at least one of these paths has flow $|f^*|/m$, as desired. □

The improved runtime is now as follows:

- The implication of the first claim is that we can perform each augmentation in $O(m \log n)$.
- The implication of the second claim is that (with some proof) the total number of augmentations is bounded above by $O(m \log nC)$. Each time we augment a path, we multiply the amount of flow we have remaining by a factor of $(1-1/m)$. Since $|f^*| \leq nC$, this reduces to bounding $nC(1-1/m)^x < 1$, where $x$ is the number of augmentations. Now,

$$nC\left(1-\frac{1}{m}\right)^x < 1 \implies -\log(nC) < x \log\left(1-\frac{1}{m}\right).$$

  Since $x\log(1-1/m) > x \cdot (-1/m)$, this holds for some $x < m\log(nC)$, as desired.

Thus, the total runtime of this version of the algorithm is $O(m^2 \log n \log nC)$, which is **weakly polynomial**.

## 10.3 Edmonds-Karp (polynomial Ford Fulkerson)

To achieve polynomial runtime, choose augmenting paths in another different way. In particular, choose them with BFS, so that the augmented path is always the shortest in terms of path length. It can be shown that the runtime in this case is $O(nm^2)$.

# 11 March 21, 2023

## 11.1 Linear Programming (LP)

**Example 11.1**
Motivating example.

We are a politician with a limited budget. We can campaign on four issues: building roads (1), gun control (2), forming subsidies (3), and gasoline tax (4). We can campaign each of these issues to urban, suburban, or rural populations, at some cost for for each separate issue campaigned.

For each additional dollar we spend campaigning a topic, this is the number of voters we gain from each region:

| | u | s | r |
|---|---|---|---|
| roads | -2 | 5 | 3 |
| guns | 8 | 2 | -5 |
| subsidies | 0 | 0 | 10 |
| gas | 10 | 0 | -2 |

Say $y_i$ is the amount of money that we spend on each topic. Say also that our political strategy insists that we get at least $50,000$ urban votes, $100,000$ suburban votes, and $25,000$ rural votes.

Now, our problem is to find

$$\min(y_1 + y_2 + y_3 + y_4),$$

subject to the constraints

$$\begin{cases} -2y_1 + 8y_2 + 0y_3 + 10y_4 \geq 50,000, \\ 5y_1 + 2y_2 + 0y_3 + 0y_4 \geq 100,000, \\ 3y_1 - 5y_2 + 10y_3 - 2y_4 \geq 25,000, \\ y_1, y_2, y_3, y_4 \geq 0. \end{cases}$$

The optimal solution $(y_1^*, y_2^*, y_3^*, y_4^*) = (2050000/111, 425000/111, 0, 625000/111)$, giving an optimal cost 3100000/111.

**Definition 11.2**

**Linear Programming** problems seek to maximize a **linear** objective subject to a **linear** constraint.

- the variable vector is given by

$$\vec{x} = [x_1, \ldots, x_n]^T \in \mathbb{R}^n.$$

- the objective function is given by

$$c_1 x_1 + \ldots + c_n x_n = \vec{c} \cdot \vec{x},$$

  where the vector of constants

$$\vec{c} = [c_1, \ldots, c_n]^T \in \mathbb{R}^n.$$

  technically, the objective function is given by $\vec{c}^T \cdot \vec{x}$. for clarity, we won't write these transposes.

- the constraints are given by

$$\sum_{i \in [n]} A_{ij} x_j \leq b_j \quad \forall j \in [m].$$

  more compactly, we will use the notation

$$A \cdot \vec{x} \leq \vec{b},$$

with $A \in \mathbb{R}^{m \times n}$, where $m$ is the number of constraints.

**Definition 11.3**
Standard form LP.

The "standard form" of linear programming problems is to maximize $\vec{c} \cdot \vec{x}$, subject to $A\vec{x} \leq \vec{b}$ and $\vec{x} \geq 0$. As before, the "$\leq$" and "$\geq$" is row-wise, i.e., $\vec{x} \geq 0 \iff x_i \geq 0 \forall i \in [n]$.

**Claim 11.4**
Any LP can be reduced to standard form LP.

*Proof.* If the objective is going the wrong direction, $\min(\vec{c} \cdot \vec{x}) \iff \max(-\vec{c} \cdot \vec{x})$. Similar logic can be applied to any inequality that faces the wrong direction. If we insist that any constraint requires equality, we can turn the constraint into two separate constraints with both $\geq$ and $\leq$. For any $x_i \in \mathbb{R}$, we can write $x_i = x_i^+ - x_i^-$, where $x_i^+, x_i^- \geq 0$. □

**Example 11.5**
Maximize $x_1 + x_2$ subject to $x_1 + 2x_2 \leq 4$ and $x_1, x_2 \geq 0$.

$\vec{c} = (1, 1)$. One way to view this problem is to find the point in the triangular region bounded by the constraints that is "as far as possible" in the direction of $\vec{c}$.

We can reason that the optimal point must lie at one of the vertices as follows. The optimal solution must lie on a border, because for any point strictly inside of the triangle, we can move further in the direction of $\vec{c}$ by moving to a boundary. Once we are on the boundary, by monotonicity, moving along at least one of the directions (right/left) on the boundary *cannot decrease* the value of the objective function. So, it suffices to check the vertices of the triangle, giving is $(4, 0)$ as the optimal solution.

## 11.2 LP Algorithms

**Example 11.6**
Simplex (Dantzig 1947).

Like the previous example, walk from vertex to vertex along the feasible polytope in the direction of $\vec{c}$. This is practical to implement, but has worst case exponential run time.

**Example 11.7**
Ellipsoid (Khachiyan 1979).

Maintains an ellipsoid that is guaranteed to contain the optimal solution. At each step, the ellipsoid is shrinked. This achieves worst case polynomial runtime, but is impractical to implement.

**Example 11.8**
Interior-point method (Karmarkar 1984).

Moves the current solution strategically so that it travels close to to the optimal solution. Instead of moving only on the boundaries (like the Simplex method), this moves the current point through the polytope itself. This is polynomial and practical to implement.

**Warning** ⚠
If we force $x_1, \ldots, x_n \in \mathbb{Z}$, the question becomes NP-hard. Even though it seems related, it is much more difficult to algorithmically solve this variant of the problem.

## 11.3 LP Duality

Let's return to the motivating example. It turns out that summing $25/222 \cdot (1) + 46/222 \cdot (2) + 14/222 \cdot (3)$ gives

$$y_1 + y_2 + \frac{140}{222} y_3 + y_4 \geq \frac{31000000}{111}.$$

This immediately proves optimality, since the left hand side bounds our objective function from below.

**Definition 11.10**
Dual LP

We can take any LP in standard form (the "primal program") and transform it into an equivalent, dual LP (the "dual program"). Take the primal program as it was written before. Then, the dual program seeks to minimize $\vec{b} \cdot \vec{y}$, subject to $A^T \vec{y} \geq \vec{c}$ and $\vec{y} \geq 0$.

This can be thought of as a generalization of the way that we solved the motivating example:

**Example 11.11**

**Theorem 11.12** (Weak LP Duality)
For any feasible solution to the primal program $\vec{x}$ and feasible solution to the dual program $\vec{y}$,

$$\vec{b} \cdot \vec{y} \geq \vec{c} \cdot \vec{x}.$$

*Proof.* Any solution to the primal program satisfies

$$\sum_j y_j \left( \sum_i A_{ij} x_i \right) \leq \sum_j y_j b_j = \vec{b} \cdot \vec{y},$$

since this is the primal constraints summed over all $y_i$. Similarly, any solution to the primal program satisfies

$$\sum_i x_i \left( \sum_j A_{ij} y_j \right) \geq \sum_i x_i c_i = \vec{c} \cdot \vec{x},$$

since this is the dual constraints summed over all $x_i$. On the other hand, the left hand side of both equations is the same: the first equation computes $y^T A x$, while the second computes $x^T A^T y$. Therefore, $\vec{b} \cdot \vec{y} \geq \vec{c} \cdot \vec{x}$, as desired. □

**Theorem 11.13** (Strong LP duality)
If $x^*$ is optimal solution to the primal program and $y^*$ is the optimal solution to the dual program, then equality holds;

$$\vec{c} \cdot \vec{x^*} = \vec{b} \cdot \vec{y^*}.$$

# 12 March 23, 2023

Game Theory!

## 12.1 Motivation: Prisoner's Dilemma

Two prisoners, prisoners $A$ and $B$. Their sentenced vary based on whether or not they choose to testify against the other:

| A/B | silent | testify |
|---|---|---|
| silent | 1/1 | 3/0 |
| testify | 0/3 | 2/2 |

For example, if $B$ testifies against $A$, but $A$ chooses to remain silent, then $B$ will walk free, while $A$ will serve 3 years in prison.

Intuitively, the best outcome is to both stay silent. However, from either individual perspective, it is always better to testify; if $B$ knows that $A$ will testify, it is better for $B$ to testify, and if $B$ knows that $A$ will stay silent, it is still better for $B$ to testify. This leads to the only "stable outcome", which is for both prisoners to testify. In this scenario, neither prisoner has motivation to deviate.

## 12.2 Games

More generally, game theory seeks to predict and reason about the actions of rational agents in situations of conflict.

**Definition 12.1** (Game)
In this class, we will only deal with two-player games between player $A$ and $B$. Let $A$ be the utility matrix of player $A$, and $B$ be the utility matrix of player $B$. Then, we say that $A_{ij}$ (resp. $B_{ij}$) is the utility of player $A$ (resp. player $B$) given the action $i$ of player $A$ and action $j$ of player $B$.

For example, in the prisoner's dilemma:

A

| | |
|---|---|
| -1 | -3 |
| 0 | -2 |

B

| | |
|---|---|
| -1 | 0 |
| -3 | -2 |

$A$ is the "row player" and $B$ is the "columm player". The first row/column represents staying silent, and the second row/column represents testifying.

## 12.3 Two-player zero-sum

In a **zero-sum** game, $A_{ij} = -B_{ij}$ for all $i, j$. In other words, "my gain is your loss". These games can be fully described by matrix $A$ or $B$.

**Example 12.2**
Rock-paper-scissors is a zero-sum game.

| | $R$ | $P$ | $S$ |
|---|---|---|---|
| $R$ | 0 | -1 | 1 |
| $P$ | 1 | 0 | -1 |
| $S$ | -1 | 1 | 0 |

There is no "stable outcome" for this game, if we insist that each player must stick to one action only:

- If I know the other player will pick $R$, I switch to $P$
- If they know I will switch to $P$, they will switch to $S$
- If I know they will switch to $S$, I switch to $R$
- ...

If we allow randomized strategies, then the pair of strategies $((1/3, 1/3, 1/3), (1/3, 1/3, 1/3))$ is stable, because there is no motivation for either player to deviate from this truly random strategy.

**Definition 12.3** (Nash Equilibrium)
A **nash equilibrium** is an outcome such that no player has an incentive to unilaterally deviate.

**Theorem 12.4** (Nash, 1950)
Every finite game has a Nash equilibrium.

**Theorem 12.5** (Min-Max Theorem – Von Neumann, 1928)
Let $P = \{x | x \geq 0, \sum x_i = 1\}$, $Q = \{y | y \geq 0, \sum y_i = 1\}$. For any $A$, let

$$V_R = \max_{x \in P} \min_{y \in Q} x^T A y$$

and

$$V_C = \min_{y \in Q} \max_{x \in P} x^T A y.$$

Then, $V_R = V_C = V$.

Think of $P$ as the set of strategies that the row player can choose from, and $Q$ the set of strategies that the column player can choose from. $x^T A y$ can be thought of as the utility of the row player given that he chooses strategy $x$, and the column player chooses strategy $y$. Then, $V_R$ represents the utility of the row player if he plays first, while $V_C$ represents the utility of the row player if his adversary plays first.

**Proof follows from LP duality**.

## 12.4 Stock Market

Let $x_t$ be the index of a stock on day $t$. Initially, $x_0 = 0$. On day $t$, you have to predict if $x_t = x_{t-1} + 1$ or $x_t = x_{t-1} - 1$. After you have made your prediction, $x_t$ is revealed. If your prediction is correct, you make money; otherwise, you lose money.

Additionally, on day $t$, you get advice from $n$ experts telling you that the index will go up or down the next day.

**Definition 12.6** (Regret)
Define the **regret** to be the number of mistakes that I make, $m$, minus the number of mistakes that the best expert would make, $m^*$.

Our goal is to minimize regret.

**Example 12.7**
Assume $m^* = 0$.

If $m^* = 0$, at least one of the experts plays the market perfectly. In this case, we

can use the **Halving algorithm**:

- Maintain a list of "credible" experts that have not made any mistakes, $S$
- At each step, predict the majority pick of experts in $S$
- Remove from $S$ anyone who made a mistake

The regret of the Halving algorithm is $O(\log n)$. A mistake is made if and only if at least half of the experts in $S$ predict wrong, in which the set $S$ is reduced by at least half. Since $|S| = n$ in the beginning, we make at most $\log n$ mistakes.

**Example 12.8**

Now consider the general case, i.e., $m^* \geq 0$.

One idea is to use an iterated halving algorithm. Run the halving algorithm until all experts have made a mistake; then, reset $S$ to all experts, and repeat. The regret of this algorithm is $O(\log n \cdot (m^* + 1))$, since we need to run the algorithm at least $m^* + 1$ times.

Intuitively, this is inefficient, since we "reset" our knowledge each time we reset the algorithm. An algorithm that better captures the intuition that we want to retain information about each expert is the **weighted majority algorithm**:

- maintain weight $w_i$ for each expert $i$
- initially, all weights are 1
- at day $t$, predict according to the weighted majority
- set $w_i = w_i/2$ for each expert $i$ that was wrong.

**Claim 12.9**

The weighted majority algorithm achieves regret of $\leq 1.4m^* + 2.4 \log m$.

*Proof.* Let $W^t$ denote the total weight of all experts on day $t$. Each time a mistake in the algorithm is made, at least half of the total weight of all experts is halved, i.e., $W^t \leq 3W^{t-1}/4$ in these rounds, which implies

$$W^t \leq \left(\frac{3}{4}\right)^m W^0 = \left(\frac{3}{4}\right)^m n.$$

On the other hand, given that the best expert makes $m^*$ mistakes, his weight $w_{i^*}^t = (1/2)^{m^*}$. So, we have

$$\left(\frac{1}{2}\right)^{m^*} = w_{i^*}^t \leq W^t \leq \left(\frac{3}{4}\right)^m n,$$

which implies

$$m^* \log 1/2 \leq m \log 3/4 + \log n,$$

which reduces to

$$m - m^* \leq 1.4m^* + 2.4 \log n.$$

□

# 13 April 4, 2023

Intractability I.

## 13.1 P, NP, NP-Completeness

**add motivating examples**

- Optimization problem: on input $I$, find an object satisfying some property and having min/max weight
- Search problem: on input $I$ and value $K$, find an object satisfying some property and having weight $\leq K$ or $\geq K$
- Decision problem: on input $I$ and value $K$, decide if there exists an object satisfying some property and having weight $\leq K$ or $\geq K$

**Example 13.1**
Search for an $s-t$ shortest path.

- Search: given $K$, output a path from $s$ to $t$ of length $\leq K$, or output that one does not exist.
- Decision: output yes if there exists a path $\leq K$, and no otherwise.

In general, decision $\leq$ search $\leq$ optimization. For example, we can deduce a solution to the decision problem above by using the search algorithm. Therefore,

if the decision problem is "hard", it must be the case that all other instances of the problem are hard.

**Definition 13.2** (P Class)
A decision problem $\pi$ is solvable in polynomial time $\pi \in P$ if there is an algorithm $A$ and constant $c$ s.t. for every input $x \in \pi$ of length $n$, $A(x)$ runs in $O(n^c)$ time and $A(x)$ returns the correct answer, i.e., $A(x)$ returns yes if and only if $\pi(x)$ returns yes.

NP does not mean "non-polynomial"; instead, it means "nondeterministic polynomial time". A good wrong name to remember is "nifty proofs".

**Definition 13.3** (NP Class)
A decision problem $\pi$ is in nondeterministic polynomial time $\pi \in NP$ if there is an algorithm $V_\pi$ (called the "verifier") and constants $c, c'$ s.t.

- $V_\pi$ takes two inputs, $x$ an instance of $\pi$, and $y$ a "certificate" or "proof"
- $V_\pi$ runs in $O((|x|+|y|)^c)$
- for every instance $x$ of $\pi$ of size $n$, $\pi(x)$ is yes if and only if there exists $y$ of size $|y| \leq n^{c'}$ such that $V_\pi(x,y)$ is yes. conversely, if $\pi(x)$ is no, then $V_\pi(x,y)$ is no for all possible $y$.

In other words, the class of NP problems is the class of problems for which there exists a "nifty proof", i.e., a polynomial certificate $y$ with length $|y| \leq n^{c'}$.

**Example 13.4**
The $s-t$ shortest path problem is in NP.

Let $x = (G, s, t, k)$ be an instance of the shortest path problem. Let $y$ be any input (it does not necessarily have to mean anything). Then, let the verifier algorithm $V(x,y)$ return YES if and only if $y$ is a path in $G$ from $s$ to $t$ of length $\leq k$.

- This verifier is linear in the inputs, since it just needs to check that $y$ is a valid path
- If the answer to the problem is YES, then a path with length $|y| \leq k$ is polyno-

mial. Otherwise, the algorithm will return NO for all possible $y$.

**Example 13.5**
The 3$D$ matching problem is in NP.

Let $x = (H, k)$ be an instance of the 3$D$ matching problem, where $H = (V, E)$ is a hypergraph with $E \subseteq V^3$. The 3$D$ matching problem seeks to find the maximum set of edges that do not share end points, i.e., that forms a **matching**.

Let $y$ be any input. The verifier $V(x, y)$ returns YES if and only if $y$ is a set of hyperedges in $H$ of size $\geq k$ that do not share endpoints. This verifier is polynomial. On the other hand, the problem itself (i.e., finding the "nifty proof" $y$) currently does not have a known polynomial time algorithm, so it is not in $P$.

**Example 13.6**
SSSP. Given $G = (V, E)$ and $s \in V$, output $d(s, v)$ for all $v \in V$.

As written, this problem is not in NP (or P), since P/NP are classes of decision problems, and this is not a decision problem.

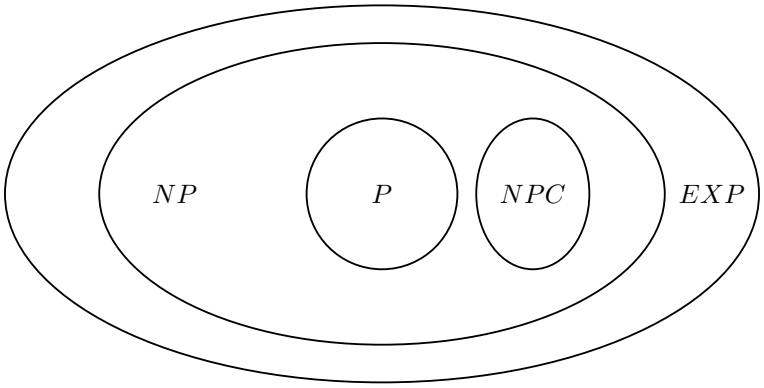
**Example 13.7**
The no-shortest path problem: given $G = (V, E)$, $s, t \in v$, and $k$, output YES if no path from $s$ to $t$ has length $\leq k$ and NO otherwise.

This problem is in NP because it is in P (for example, we can solve the $s - t$ shortest path in polynomial time, and check if the shortest path has length $\leq k$). Since we can solve the problem itself in polynomial time, the verifier can ignore any certificate and just solve the problem itself. This demonstrates the following theorem.

**Theorem 13.8**
$P \subseteq NP$.

*Proof.* Let $\pi \in P$ be a decision problem with polynomial time algorithm $A$. Then, the verifier $V_\pi(x, y) = A(x)$ works. □

## 13.2 Current State of the World




- It is unknown whether $P = NP$
- Equivalently, it is unkonwn whether $P$ and $NPC$ overlap. If the overlap is non-empty, then all problems in $NP$ can be reduced to $P$, implying that $P = NP$. Otherwise, there are $NPC \subseteq NP$ problems not in $P$, implying that $P \neq NP$.
- It is known that $P \neq EXP$.
- It is known that $NP \subseteq EXP$. Problems in $NP$ have polynomial time verifiers, so an $EXP$ algorithm for every problem in $NP$ is to bash through every possible certificate, which takes $O(2^{O(n^c)})$, and see if the verifier ever returns YES. This is $EXP$ because the length of each certificate is assumed to be polynomial.

**Definition 13.9** (Many-one polynomial time reduction)
Let $Q$ and $\pi$ be decision problems. A many-one polynomial time reduction from $Q$ to $\pi$ is an algorithm $R$ that takes in $x$, an instance of $Q$, and outputs $y$, an instance of $\pi$, such that

- $R$ runs in polynomial time
- if $Q(x)$ is yes, then $\pi(y)$ is yes
- if $Q(x)$ is no, then $\pi(y)$ is no

We say $Q \leq_p \pi$ to denote that $Q$ can be reduced to $\pi$. Note that $\pi$ is "harder", since if we can solve $\pi$ in polynomial time, this implies that we can solve $Q$ in polynomial time by the reduction.

**Definition 13.10** (NP-Complete)
$\pi \in NP$-complete if and only if $\pi \in NP$ and $\pi$ is **NP-hard**. Formally, $Q \leq_p \pi$ for all $Q \in NP$. Informally, $\pi$ is at least "as hard" as every other problem in NP.

# 14 April 6, 2023

Intractability II.

## 14.1 Circuit-SAT

The input to Circuit-SAT:

- Boolean circuit $C$ on variables $x_1, \ldots, x_n$

Output:

- YES if $C$ is satisfiable, and NO otherwise.

**Definition 14.1**
A **boolean circuit** is a directed acyclic graph. Each node represents a boolean gate, e.g., AND, OR, NOT, that takes in the appropriate number of inputs and outputs any number of outputs which all have the value of the operation on the inputs. One of the nodes is designated as the output node. The input nodes $x_1, \ldots, x_n$ are fed through the graph, and the output of the graph is said to be the output value of the output node.

Boolean circuits can be evaluated in linear time. In particular, because they are DAGs, there is a topological ordering of the vertices. Since the input nodes $x_1, \ldots, x_n$ are sources, and the output node is a sink, we can use DP to compute the value of each node in linear time.

Circuit-SAT asks whether $C$ is satisfiable. $C$ is said to be **satisfiable** if there exists an initial assignment to $x_1, \ldots, x_n$ which produces an output value of 1. The fastest known algorithm is to brute force all inputs, $2^n$.

**Theorem 14.2** (Cook-Levin)
Circuit-SAT is NP-complete.

For any problem $\pi$, there are two steps to show that $\pi$ is NP-complete:

- prove that $\pi \in NP$.
- prove that there exists a reduction $Q \leq_p \pi$ from any problem $Q \in NP$.

**Claim 14.3**
Circuit-SAT $\in$ NP.

*Proof.* Verifying Circuit-SAT is equivalent to evaluating a boolean circuit in linear time, which we discussed above is possible with dynamic programming. □

To prove that Circuit-SAT is NP-hard, we will make use of the following theorem:

**Theorem 14.4**
Let $Q$ be some decision algorithm, and $A$ an algorithm solving $Q$ in $p(n)$ time on inputs of size $n$. Then, for every fixed $n$, there is a Boolean circuit $C_n$ of size poly$(p(n))$ s.t. for every $n$-length input $x$, $Q(x) = C_n(x)$. And, $C_n$ can be constructed in poly$(p(n))$ time.

Assumption of the model:

- Let $L_i$ be the state in memory at step $i$.
- Between states $L_i$ and $L_{i+1}$, a single operation is performed transforming $L_i$ into $L_{i+1}$.
- Then, there is a fixed boolean circuit $M$ that takes $L_i$ to $L_{i+1}$, which has size poly$(p(n))$, and can be written down in poly$(p(n))$. This is a reasonable assumption because this is how processors work.

*Proof.* Look at a run of $A$ on size $n$ inputs. Each operation can be modelled by a change in state from $L_i$ to $L_{i+1}$ on whatever machine is running our algorithm. Therefore, given the sequence of states $L_0, \ldots, L_{p(n)}$ which runs $A$, there is a sequential circuit taking $L_0$ to $L_{p(n)}$ with size poly$(p(n))$, so we are done. □

**Theorem 14.5**
Circuit-SAT is NP hard.

*Proof.* Let $V_Q$ be the verifier algorithm for $Q$. Let $x$ be an instance of $Q$ of size $n$. $V_Q(x,y)$ where $|x| = n$ and $|y| = p(n) \in \text{poly}(n)$. Let $N = |x| + |y| \in \text{poly}(n)$. By the previous theorem, there exists a boolean circuit $C_N$ such that $C_N(x,y) = V_Q(x,y)$ for all $x, y$ which is constructable in $\text{poly}(n)$ time.

Now we have a circuit $C_N$ which takes $x, y$ as input and produces $V_Q(x,y)$ in $\text{poly}(n)$ time. We want to reduce $x$ itself to a boolean circuit problem. Since $x$ has a binary representation, hard wire its input in $C_N$, which produces a circuit $C_{N,x}$ whose only input is a certificate $y$.

Finally, $Q(x)$ has a solution if and only if there exists some certificate $y$ such that $V_Q(x,y) = 1$. Since the circuit $C_{N,x}$ takes in any certificate and produces $V_Q(x,y)$ in polynomial time, finding $Q(x)$ is equivalent to seeing whether there exists an input to $C_{N,x}$ which produces a positive output value, which is Circuit-SAT, and we are done. □

## 14.2 CNF-SAT, 3-SAT

CNF: Conjunctive Normal Form.

**Definition 14.6**
$F$ is a CNF formula on $x_1, \ldots, x_n$ if it is of the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where each $C_i$ is a **clause** of the form $(l_1 \vee \ldots \vee l_{k_i})$, and each **literal** $l_i$ is either $x_r$ or $\neg x_r$ for some $r$.

$F$ is a 3-CNF formula if every clause has three literals. The 3SAT problem takes in some 3-CNF formula $F$ on $n$ variables and outputs whether there is a boolean assignment on which $F$ evaluates to true.

**Theorem 14.7**
3SAT is NP-complete.

*Proof.* 3SAT is in NP because it is trivial to check whether a boolean assignment $y$ satisfies $F$.

Now, we show that 3SAT is NP-hard by showing that there exists a reduction from Circuit-SAT to 3-SAT. Take any instance of Circuit-SAT $C$ on $n$ variables $x_1, \ldots, x_n$, with $t$ gates and $m$ wires. Since we may assume everything is connected, all sizes are $O(m)$. At each gate, define a variable $g_i$. Now, construct the following "clauses" on $g_1, \ldots, g_t$ (these are not necessarily proper CNF clauses):

- The value of the output gate is 1
- For every $i$, $g_i$ computes the value of the gate. For example, the clause corresponding to $g_2$ would be $(g_2 \iff \neg x_3)$ for a NOT gate which takes $x_3$ as input.

It is possible to construct all of these "clauses" in linear time $O(m)$ by iterating through the circuit in topological order.

To finish, rewrite each "clause" with proper CNF clauses. For example, $(g_i \iff \neg a)$ is equivalent to $(\neg g_i \vee \neg a) \wedge (g_i \vee a)$. By writing out the full normal form of each possible gate, it is possible to show that we can replace each "clause" with at most 3 CNF clauses, hence our final expression is a 3-CNF formula and we are done. □

# 15 April 11, 2023

Intractability III.

## 15.1 Vertex Cover

**Definition 15.1**

A **vertex cover** of a graph $G = (V, E)$ is a vertex subset $S \subseteq V$ such that for every edge $(u, v)$ either $u \in S$ or $v \in S$ or both.

**Example 15.2**

Vertex cover of $K_4$.

All four vertices forms a vertex cover of size 4. Any subset of three vertices forms a vertex cover of size 3. There are no vertex covers of size $\leq 2$, because then

there exists an edge between the other two vertices which is not covered by the two selected vertices.

The vertex cover decision problem has graph $G = (V, E)$ and integer $K$. The output is YES if and only if there is a subset $S \subseteq V$ of size $\leq K$ s.t. $S$ is a vertex cover.

**Theorem 15.3**

Vertex Cover is NP-complete via $3\text{SAT} \leq_p \text{VC}$.

As a reminder, to prove this, we need to show:

- Vertex Cover is in NP. This is easy to show, since it is easy to check whether a given vertex cover is valid.
- there exists a graph $G = (V, E)$ and integer $K$ such that $G, K$ is created in polynomial time in $n$ and $m$, given a 3CNF formula $F$
- $F$ has a satisfying assignment if and only if $G$ has a vertex cover of size $\leq K$.

*Proof.* First, we construct our reduction:

- Let $m$ be the number of clauses and $n$ the number of literals. Set $K = n + 2m$.
- For each variable $x_i$, create two vertices connected by an edge $x_i^T$, $x_i^F$. In a vertex cover, this means that either $x_i^T$ or $x_i^F$ needs to be selected, which can be thought of as the variable $x_i$ being true or false.
- For every clause $(l_{j1}, l_{j2}, l_{j3})$, create a triangle with vertices $l_{j1}, l_{j2}, l_{j3}$.
- For each $l_{jk}$ in clause $j$, connect $l_{jk}$ to $x_p^T$ if $l_{jk} = x_p$ and connect $l_{jk}$ to $x_p^F$ if $l_{jk} = \neg x_p$.

This reduction can be constructed in linear time. Now we show that it works.

---

(FORWARD DIRECTION) Let $x_1 = b_1, \ldots, x_n = b_n$ be a satisfying assignment to $F$. First, add $x_i^T$ to $S$ for all $x_i$ with $b_i = 1$, and $x_i^F$ to $S$ for all $x_i$ with $b_i = 0$. Then, each clause has at least one of the literals set to 1, since we have a satisfying assignment. Pick one of the literals set to 1 and put the vertices corresponding to the other two literals in $S$.

Since we are adding two vertices per clause, and one additional vertex per variable ($x_i^T$ or $x_i^F$), $|S| = n + 2m$. Now we show that $S$ is a vertex cover.

- Edges of the form $(x_i^T, x_i^F)$ are covered, since we always select either $x_i^T$ or $x_i^F$.
- Edges that are clause edges, i.e., $(l_{jk}, l_{jh})$, are covered, since we selected two vertices per clause triangle.
- All other edges are of the form $(l_{jk}, x_p^B)$, where if $l_{jk} = x_p$, then $B = T$, and if $l_{jk} = \neg x_p$, then $B = F$. If $l_{jk} \in S$, then this edge is covered. Otherwise, $l_{jk}$ was set to true. If $l_{jk} = x_p$, this means that $x_p$ was set to true, so $x_p^T$ was selected and the edge was covered. If $l_{jk} = \neg x_p$, this means that $x_p$ was set to false, so $x_p^F$ was selected and the edge was covered.

---

(BACKWARDS DIRECTION) Let $S$ be a vertex cover of $G$ of size $\leq K = 2m + n$. Every one of the $m$ clause triangles must have at least 2 vertices in $S$. Each of the $n$ edges must have at least one vertex in $S$. Thus, $|S| \geq 2m + n = K \geq |S|$, so $|S| = 2m + n$. Now, for our assignment, if $x_i^T \in S$, set $x_i = 1$, otherwise set $x_i = 0$.

To show that this works, we need to show that every clause triangle contains a literal that was set to true. Consider any clause $(l_1, l_2, l_3)$. Exactly two vertices are in $S$. Suppose WLOG that $l_3$ was not selected to be in $S$, and consider the edge $(l_3, x_p^B)$. If $l_3 = x_p$, $x_p^T$ has to be in $S$, which implies that $x_p$ is true, hence $l_3$ is true. Otherwise, $l_3 = \neg x_p$, so $x_p^F$ has to be in $S$, implying that $x_p$ is false, lence $l_3$ is still true. Therefore, the literal not selected to $S$ in each clause triangle evaluates to true, so we are done. □

## 15.2 Subset Sum

- Input: $n$ integers $S = \{a_1, \ldots, a_n\}$ and a target integer $t$ encoded in binary
- Output: YES if there are $a_{i_1}, \ldots, a_{i_k} \in S$ such that $\sum_{j=1}^{k} a_{i_j} = t$. Otherwise, NO.

This can be solved in $O(nt)$ with knapsack DP. Note that this runtime is pseudopolynomial, since it is not polynomial in $\log t$, which is the size of $t$ in binary.

**Theorem 15.4**

Subset Sum is NP-complete via VC$\leq_p$Subset Sum.

*Proof.* First, our reduction. Let $G = (V, E)$ and $K$ be an instance of the vertex cover problem. Let $E = \{0, \ldots, m-1\}$. Create two types of integers in $S$:

- for every $e \in E$, $b_e = 4^e$.
- for every $v \in V$, $b_v = 4^m + \sum_{e \in E, e=(u,v)} 4^e$.

Then, let the target $t = K \cdot 4^m + 2\sum_{e \in E} 4^e$. This is a polynomial-time reduction. Now, we show that it works.

---

(FORWARD DIRECTION) Let $C$ be a vertex cover of $G$ of size $K$. Each edge is covered by $C$ once or twice. Let $E' \subseteq E$ be the edges in $G$ that are covered by $C$ exactly once. Now, let

$$A = \{b_v : v \in C\} \cup \{b_e : e \in E'\}.$$

Note that

$$\sum_{v \in C} b_v = |C|4^m + \sum_{e \in E - E'} 2 \cdot 4^e + \sum_{e \in E'} 4^e.$$

Therefore,

$$\sum_{b \in A} b = \sum_{v \in C} b_v + \sum_{e \in E'} b_e = |C|4^m + 2\sum_{e \in E} 4^e = t,$$

as desired.

---

(BACKWARDS DIRECTION) Suppose $A \subseteq S$ sums to $t$. $A$ contains vertex numbers $b_v$ for vertices from some set $C$, and edge numbers $b_e$ for edges in some set $E'$. As before, $E'$ represents the set of edges that we can cover only once. We know that

$$\sum_{v \in C} b_v + \sum_{e \in E'} b_e = t.$$

The key is to look at each of the numbers in $A$, in base 4. The number of times that edge $e$ is counted corresponds to the number of times that $e$ is counted in the sum. In the sum over the vertices $\sum_{v \in C} b_v$, each edge is counted at most twice, since it is covered at most twice. In the sum over the edges in $E'$, each edge is covered only once. Therefore, in each coordinate $e \in \{0, \ldots, m-1\}$, the value is at most 3 (in particular, there are no carries).

Now, the target has a value of 2 in each edge coordinate. Since the contribution to each edge $e$ is (number of times edge covered by $C$) + (1 if $e \in E'$ and 0 otherwise),

and we know this has a value of 2 for each edge, this means that the number of times each edge is covered by $C$ is at least 1, so $C$ covers every edge. Therefore, $C$ is a vertex cover, and we are done. □

# 16 April 13, 2023

Random Walks I.

## 16.1 Scotland Yard

Consider a game on $K_5$. A runner is allowed to move around the vertices, jumping to an adjacent vertex once each move. A detective, which is visible to the runner, is also able to move around the vertices. After some number of moves, the location of the runner is visible to the detective. If they happen to be in the same place, the detective wins. Otherwise, the game continues.

### 16.1.1 Version 0.5

In this version of the game, the runner cannot stay put. Let $X_i = (p_A, \ldots, p_E)$ be a tuple denoting the probabilities that the runner ends up at each vertex after $i$ moves, assuming that he moves randomly. For example, $X_1 = (0, 1/4, 1/4, 1/4, 1/4)$, and $X_2 = (1/4, 3/16, 3/16, 3/16, 3/16)$.

**Claim 16.1**
Let $A$ be the adjacency matrix of a graph $G$. Then $(A^\ell)_{ij}$ counts the number of paths from $i$ to $j$ in $G$ with length exactly $\ell$.

For example,

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 4 & 3 & 3 & 3 & 3 \\ 3 & 4 & 3 & 3 & 3 \\ 3 & 3 & 4 & 3 & 3 \\ 3 & 3 & 3 & 4 & 3 \\ 3 & 3 & 3 & 3 & 4 \end{pmatrix}.$$

By repeatedly squaring the adjacency matrix $K_5$, we have a way to compute $X_\ell$ for any $\ell$. When we normalize the rows, we get a **transition matrix**. Multiplying to infinity, it turns out that $X_i$ converges to $(1/5, 1/5, \ldots, 1/5)$.

In general, it can be shown that for any graph $G$ that $X_i$ converges to $(p_1, \ldots, p_n)$, where $p_j$ is the ratio of the degree of $j$ to the total number of edges.

## 16.2 Gambling

I start with \$20. I choose to play a game until I either go broke, or double my money and reach \$40. The game that I play has a 50/50 chance of winning and losing. Each win gives me \$5, while each loss causes me to lose \$5.

**Definition 16.2**
A stochastic process is a sequence of random variables $X_0, X_1, \ldots$.

**Definition 16.3**
A markov process is a memoryless stochastic process, i.e., $X_{t+1}|X_t = X_{t+1}|X_t, X_{t-1}, \ldots, X_0$.

**Definition 16.4**
A markov process is **time-homogenous** when $\mathbb{P}[X_t = u|X_{t-1} = v]$ is independent of $t$.

**Definition 16.5**
The **period** of a directed (resp. undirected) graph is defined as the greatest common divisor of the lengths of all directed (resp. undirected) cycles in the graph. If the period is 1, the graph is said to be **aperiodic**. Otherwise, the graph is **periodic**.

Recall that we can represent the transition state for any (time-homogenous) Markov process with a unique walk matrix $W$.

**Definition 16.6**
A stationary distribution $\vec{x}$ is any distribution satisfying $\vec{x}W = \vec{x}$.

For example, in Scotland yard, the distribution $(1/5, 1/5, 1/5, 1/5, 1/5)$ is a stationary distribution.

**Theorem 16.7** (Fundamental Theorem of Markov Chains)
Let $G_\chi$ be a Markov Chain for Markov Process $\chi = \{X_t\}_{t\geq 0}$. If $G_\chi$ is strongly connected and aperiodic, then every random walk on $G_\chi$ converges to a unique stationary distribution.

# 17 April 20, 2023

## 17.1 More on Fundamental Markov Chain Theorems

Some review from last lecture:

- A **stochastic process** is a sequence of random variables $\{X_t\}$
- A **Markov process** is a *memoryless* stochastic process, meaning that $X_{t+1}|X_t = X_{t+1}|X_t, X_{t-1}, \ldots, X_0$. In other words, "the weather today depends only on the weather yesterday".
- A **Markov chain** is is a graphical depiction of a markov process. Let $\chi$ be a markov process. Under the assumption of time homogeneity, the corresponding markov chain is the graph $G_\chi = (\mathcal{D}, E, w)$, where $\mathcal{D}$ is the set of states ($X_t \in \mathcal{D}$), $E$ is a set of edges, and $w$ an edge-weight function.
  - directed edges take states to other states. edges are included only if their corresponding probabilities are non-zero
  - the edge-weight function maps edges to transition probabilities
  - the matrix with entries $w_{uv} = w(u, v)$ for all $u, v \in \mathcal{D}$ makes up the **transition matrix**.
- **Time homogeneity** is the property that $\mathbb{P}[X_{t+1} = u|X_t = v]$ is independent of $t$.
- A **random walk** is an emulation of a Markov Process on a Markov Chain. A **trajectory** is a specific walk. Random walks are described by their probability distributions, i.e., $\mathbb{P}[X_t] = \vec{x}^{(0)}W^t$ for some initial distribution and walk matrix.
- A **stationary distribution** is some distribution satisfying $\vec{x}W = \vec{x}$.

Now,

**Theorem 17.1** (Fundamental Theorem of Markov Chains)
If a markov chain is strongly connected and aperiodic, then all walks converge to a unique stationary distribution $\pi$.

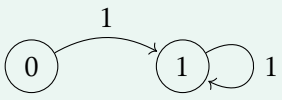
The following two subtheorems are true (and together imply the bigger theorem):

**Theorem 17.2**
If a markov chain is strongly connected, then there exists a unique stationary distribution $\pi$.

**Theorem 17.3**
If a markov chain is aperiodic, then all walks converge to some stationary distribution.

## 17.2 Examples of Markov Chain properties

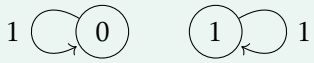
**Example 17.4**




Properties:

- not strongly connected
- aperiodic, since it has a self-loop
- $W = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$
- This graph has a unique stationary distribution $\pi = (0,1)$. Also, every random walk converges to this distribution in one step. This shows that the fundamental theorem of markov chains is not bidirectional.

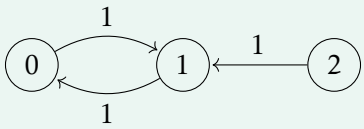
**Example 17.5**




Properties:

- not strongly connected
- aperiodic
- All distributions are stationary, and therefore all walks converge to their initial distribution.

**Example 17.6**




Properties:

- not strongly connected
- periodic, since all cycle lengths are even
- there is one stationary distribution $\pi = (1/2, 1/2, 0)$
- there are many walks that don't converge to $\pi$. for example, any $(x, y, 0)$ with $x, y \neq 1/2$ flips the mass of the first two nodes at each timestep.

## 17.3 Monto-Carlo Markov Chain

Idea: want to produce samples from a probability distribution. To do this, design a markov chain whose stationary distribution is the target distribution, and then perform samples by performing a random walk.

**not finished**

# 18 April 25, 2023

## 18.1 Approximation Algorithms

Notation:

- Let $\mathcal{P}$ denote a class of problems
- Let $P \in \mathcal{P}$ denote a specific problem
- Let $\mathcal{A}$ denote an algorithm for the class of problems
- Let $x$ be a proposed specific solution
- Let $\mathcal{A}(P)$ be the quality of the solution obtained by $\mathcal{A}$
- Let $\text{OPT}(P)$ be the quality of the optimal solution

**$\alpha$-approximation algorithm**: for $\mathcal{P}$, we are given some approximation ratio $\alpha \geq 1$ In minimization, $\mathcal{A}(P) \leq \alpha \cdot \text{OPT}(P)$. In maximization, $\text{OPT}(P) \leq \alpha \cdot \mathcal{A}(P)$.

## 18.2 Approximation Schemes

- minimization: $\forall \varepsilon > 0$, $\forall P \in \mathcal{P}$,

$$S(P, \varepsilon) \leq (1 + \varepsilon)\text{OPT}(P).$$

- maximization: $\forall \varepsilon > 0$, $\forall P \in \mathcal{P}$,

$$\text{OPT}(P) \leq (1 + \varepsilon)S(P, \varepsilon).$$

Efficient – polynomial time approximation scheme. Fully efficient – fully polynomial time approximation scheme, i.e., polynomial in both the size of the problem and $1/\varepsilon$.

**Example 18.1**
Maximum matching.

Proposed algorithm: go over edges greedily, and add an edge to the matching if possible. Let $|M|$ be the size of the matching obtained using this algorithm. Let $|M^*|$ be the size of the true maximal matching.

Note that solving this problem exactly can be done with max flow. On the other hand, this approximation algorithm is a linear scan through the vertices, so the runtime is much more efficient.

**Claim 18.2**

This algorithm is a 2-approximation, i.e., $|M^*| \leq 2|M|$.

*Proof.* $e \in M$, $e^* \in M^*$, label $e$ with $e^*$ if they share an endpoint. Each $e \in M$ has $\leq 2$ labels, since $M^*$ is a matching, so the total number of labels is $\leq 2|M|$. On the other hand, every edge in $M^*$ is included as a label. If not, then any edge that is not included as a label is not adjacent to any of the edges in $M$, meaning that the greedy algorithm would have added it. So the total number of labels is $\geq |M^*|$, which completes the proof. □

**Example 18.3**

Vertex Cover.

Recall that the vertex covering problem seeks to find the smallest subset $S$ of vertices such that every edge in the graph is incident to at least one vertex in $S$.

Consider some maximal matching found by the previous greedy algorithm, $M$. Let $V(M)$ denote the set of vertices adjacent to edges in $M$. Then, $|V(M)|$ is a 2-approximation to the size of best vertex cover $|V^*|$.

**Claim 18.4**

$V(M)$ is a vertex cover, and $|V(M)| \leq 2|V^*|$.

*Proof.* If $V(M)$ is not a vertex cover, there exists some edge that is not adjacent to any other vertices in $V(M)$, so it can be added to $M$ by the greedy algorithm. Further, $|M| \leq |V^*|$; since $M$ is a matching, $V^*$ must include at least one vertex in each edge. Therefore, $|V(M)| = 2|M| \leq 2|V^*|$. □

**Example 18.5**

LP.

Consider the following integer LP: take variables $x_v \in \{0,1\}$, and compute $\min \sum x_v$ subject to the constraint $x_u + x_v \geq 1$ for all $(u,v) \in E$.

This formulation is equivalent to the min-vertex cover problem. Since we are forcing integer values, this problem is NP-hard. But, we can relax it so that $0 \leq x_v \leq 1$ and obtain a polynomial time solution.

For this fractional LP, take $x_v$, round it to the nearest integer, and include it in the VC if it becomes 1. This works, because $x_u + x_v \geq 1$ implies at least one of $\{x_u, x_v\}$ is $\geq 1/2$, so every edge is covered.

**Claim 18.6**

The rounding LP achieves a 2-approximation for VC.

*Proof.* $\sum_{v \in V} \textsc{round}(x_v) \leq 2\text{OPT(fractional)} \leq 2\text{OPT(LP)}$. □

# 19 April 27, 2023

## 19.1 Exponential Time Algorithms

**Definition 19.1**

$O^*(f(n)) = f(n) \cdot \text{poly}(n)$.

Analogously to the way that normal big-$O$ notation ignores constant time factors, $O^*$ notation ignores polynomial time factors, since these are considered "cheap" when doing exponential time analysis.

## 19.2 Subset Sum

- input: $A = \{a_1, \ldots, a_n\}$ and target $t$
- output: whether there is some $S \subseteq A$ such that $\sum_{s \in S} s = t$.

The brute force algorithm is to try every possible subset. This is $O^*(2^n)$.

To improve this runtime, we can also "meet in the middle". Consider splitting $A$ into two lists of size $n/2$. Now construct $L_1$ which has the sum of all possible subsets of the first list, which has size $2^{n/2}$. Let $L_2$ be the set of all $t - w$ for all possible sums $w$ of the second list, which has size $2^{n/2}$. Now, the problem reduces to finding

a common element between $L_1$ and $L_2$. Using hashing, sorting, or otherwise, this can be done in $O(n2^{n/2}) \in O^*(2^{n/2})$.

- Brute force algorithm: $O^*(2^n)$ time, $O^*(1)$ space
- Meet in the middle: $O^*(2^{n/2})$ time, $O^*(2^{n/2})$ space.
- Schroppel-Shamir: $O^*(2^{n/2})$ time, $O^*(2^{n/4})$ space.
- Nederlof-Wegrzycki: $O^*(2^{n/2})$ time, $O^*(2^{0.0249999n})$ space.
- Bansal, Garg, Nederlof: $O^*(2^{0.86n})$ time, poly space.

## 19.3 3SAT

- input: 3CNF formula $F$ on $n$ variables and $m$ clauses
- output: is there a boolean assignment to the $n$ variables satisfying $F$?

The brute force algorithm is to try every possible assignment of variables, which is $O^*(2^n)$.

Branch on variables:

- Plug in $x_1 = 1$ and recurse. If the result was satisfiable, return this result.
- Otherwise, the result was not satisfiable, then try $x_1 = 0$ and recurse. Return the result.

Each "branch" reduces the number of variables by 1, so the total runtime satisfies $T(n) \leq 2T(n-1) + O^*(1) \in O^*(2^n)$.

**Claim 19.2**

2-SAT has a polynomial time algorithm.

Intuition: each branch resolves all clauses in the same connected component.

**Example 19.3**

Improved algorithm.

Using this fact, we can create a new branching algorithm:

- Simplify $F$ by removing redundant variables in clauses. If $F$ is a 2-CNF, then solve $F$ in $O^*(1)$
- Otherwise, pick a clause with three distinct variables $(\ell_1, \ell_2, \ell_3)$. There are exactly $7 < 2^3$ assignments which satisfies this clause; branch on these assignments, and recurse.
- Now, the total depth of the recursion tree is at most $n/3$, since we guarantee the assignment of three distinct variables at each branch. This implies that the total number of nodes is $O(7^{n/3})$. In each node, we perform $O^*(1)$ work to reduce the formula, so our runtime is $O^*(7^{n/3}) \in O^*(1.91^n)$.

**Example 19.4**

Even more improved algorithm.

Instead of directly branching on all 7 possible valid triples $(\ell_1, \ell_2, \ell_3)$, we can branch one at a time:

- try $\ell_1 = 1$ and recurse. If this works, return that it works.
- otherwise, try $\ell_1 = 0$, $\ell_2 = 1$ and recurse. If this works, return that it works.
- otherwise, try $\ell_1 = 0$, $\ell_2 = 0$, $\ell_3 = 1$ and recurse. return the result.

Like the last algorithm, this accounts for all possible triples $(\ell_1, \ell_2, \ell_3)$, but does so by branching one at a time, which is slightly more efficient. The runtime is

$$T(n) \leq T(n-1) + T(n-2) + T(n-3) + O^*(1).$$

If we guess $T(n) \in O^*(a^n)$, this reduces to $a^3 = a^2 + a + 1$, which gives $a \leq 1.84$, so the total runtime is $T(n) \in O^*(1.84^n)$. This is OK. The best known algorithm is $O^*(1.308^n)$.

# 20 May 2, 2023

## 20.1 Online Learning

**Definition 20.1**

In online learning, the length of the input is not known in advance. The problem instance can grow forever, and the goal of an algorithm is to, with a fixed amount of memory, produce reasonable outputs for each new part of the input that is read.

Defining competitiveness for deterministic and random algorithms:

**Definition 20.2**

A *deterministic* algorithm $\mathcal{A}$ is $\alpha$-competitive if it satisfies

$$C_{\mathcal{A}}(R) \leq \alpha \cdot C_{\text{OPT}}(R) + c,$$

for constants $\alpha, c$. A *randomized* algorithm $\mathcal{A}$ is $\alpha$-competitive if it satisfies

$$\mathbb{E}[C_{\mathcal{A}}(R)] \leq \alpha \cdot C_{\text{OPT}}(R) + c.$$

Adversarial inputs in the randomized case are assumed to have no knowledge of the outcome of each random decision. This is called an **oblivious adversary**.

## 20.2 BIT for self-organizing lists

BIT is a randomized algorithm for the problem of self-organizing lists. Consider the variant of this problem where we can move elements to the front of the list for free. We showed previously that the deterministic MTF algorithm achieves 2-competitiveness on this problem.

To start, randomly initialize a binary array $b$ for each entry in the list. Then, for each access:

- flip $b(x) = 1 - b(x)$
- if $b(x) = 1$, move $x$ to the front of $L$.

This means that every element is moved to the front of $L$ on every other access.

**Theorem 20.3**
BIT is 7/4-competitive.

*Proof.* As before, define:

- $A_i$ the number of elements before $x_i$ in $L_i$ and $L_i^*$
- $B_i$ the number of elements before $x_i$ in $L_i$ and after in $L_i^*$
- $C_i$ the number of elements after $x_i$ in $L_i$ and before in $L_i^*$
- $D_i$ the number of elements after $x_i$ in $L_i$ and $L_i^*$

The potential function we will use is

$$\Phi_i = (\text{inversions with } b(y) = 0) + 2 \cdot (\text{inversions with } b(y) = 1).$$

This is a valid potential function, since it starts at $\Phi_0 = 0$ and is always nonnegative. Now there are two cases to analyze, each of which occurs with probability 1/2:

- If $b(x_i) = 0$, BIT moves $x_i$ to the front of $L_i$. This fixes $B_i$ inversions and creates $A_i$ inversions.

  Let $A_i'$ be the number of elements in the $A_i$ new inversions created that OPT fixes. Let $t_i$ be the number of transpositions made afterwards. OPT fixes $A_i'$ inversions and creates at most $t_i$ new inversions.

  Finally, each inversion fixed/destroyed contributes e.v. 3/2 to the potential function. So,
  $$\mathbb{E}[\Delta\Phi] \le \frac{3}{2}(A_i - A_i' - B_i + t_i).$$

- If $b(x_i) = 1$, BIT does not move $x_i$ to the front, so no inversions are fixed by BIT. However, all of the inversions corresponding to $B_i$ now contribute 1 to the potential instead of 2, since $b(x_i)$ flipped, hence $B_i$ are "fixed".

  As before, if OPT moves forwards by $A_i'$, then $A_i'$ new inversions are created. If we let $t_i$ be the number of transpositions made afterwards, at most $t_i$ new

inversions are created. So,

$$\mathbb{E}[\Delta\Phi] \le \frac{3}{2}(A_i' - B_i + t_i).$$

The true cost to access $x_i$ in BIT is $A_i + B_i + 1$, while the true cost to access $x_i$ in OPT is $A_i + C_i + 1 + t_i$. The amortized cost is

$$\begin{aligned} A_i + B_i + 1 + \Delta\Phi &\le A_i + B_i + 1 + \frac{1}{2}\left(\frac{3}{2}(A_i - A_i' - B_i + t_i) + \frac{3}{2}(A_i' - B_i + t_i)\right) \\ &= \frac{7}{4}A_i - \frac{1}{2}B_i + \frac{3}{2}t_i + 1 \\ &\le \frac{7}{4}(A_i + C_i + t_i + 1), \end{aligned}$$

which completes the proof.

□

## 20.3 Regret

Here, we say that our adversarial sequence of inputs is **adaptive**, i.e., it has access to our algorithm and what it has predicted so far. To determine the performance of a learning algorithm, we incur some cost for each action that we take.

Then, we take a benchmark loss, and compare it with our true loss, to determine the **regret** of the algorithm. For example, $B = \sum_{i=1}^{T} \min_{a \in A} c_t(a)$ is a benchmark where the best actions in hindsight are taken at every step, while $B = \min_{a \in A} \sum_{i=1}^{T} c_t(a)$ is the cost incurred by picking the best fixed action in hindsight.

Let $C$ be the true cost incurred by the algorithm. If

$$\frac{1}{T}(\mathbb{E}[C] - \mathbb{E}[B]) = 0,$$

the algorithm has *no regret* with respect to the benchmark. **External regret** is the regret with respect to the benchmark of the best fixed action.

## 20.4 Weighted Majority

**not finished**

# 21 May 4, 2023

## 21.1 Unconstrained Optimization

Given an objective function $f : \mathbb{R}^n \to \mathbb{R}$, our goal is to compute

$$x^* = \arg\min_{x \in \mathbb{R}^n} f(x).$$

We assume that $f$ is continuous and smooth. One idea is **gradient descent**:

- generate a sequence of values $x^0 \to x^1 \to \ldots \to x^t$, which satisfies $f(x^{t+1}) \leq f(x^t)$ for all $t$.
- $x^{t+1} = x^t - \eta \nabla f(x^t)$.

**Definition 21.1**

$f$ is **$\beta$-smooth** for $\beta \geq 0$ if and only if, for all $x, \delta$,

$$\delta^T \nabla^2 f(x) \delta \leq \beta |\delta|^2.$$

Recall that $\nabla^2 f(x)$ is the Hessian matrix, with $\nabla^2 f(x)_{i,j} = \partial^2 f(x)/(\partial x_i \partial x_j)$. The statement of $\beta$-smoothness is equivalent to saying that the largest eigenvalue of the Hessian has magnitude less than $\beta$.

We can show that if $f$ is $\beta$-smooth, then

$$f(x+\delta) \leq f(x) + \nabla f(x)^T \delta + \frac{1}{2}\beta |\delta|^2.$$

Plugging in $\delta = -\eta \nabla f(x)$,

$$f(x+\delta) = f(x) - \eta |\nabla f(x)|^2 + \frac{\beta \eta^2}{2} |\nabla f(x)|^2.$$

Since we want to make progress at each step,

$$\eta |\nabla f(x)|^2 \geq \frac{\beta \eta^2}{2} |\nabla f(x)|^2 \implies \eta \leq \frac{2}{\beta}.$$

If we choose $\eta = 1/\beta$, then we guarantee

$$f(x^{t+1}) - f(x^t) \leq -\frac{1}{2\beta}|\nabla f(x)|^2.$$

**Definition 21.2**

$f$ is convex if and only if

$$f(x+\delta) \geq f(x) + \nabla f(x)^T \delta$$

for all $x, \delta$.

In other words, convexity means that "the function always lies above the tangent plane".

**Definition 21.3**

$x$ is an $\varepsilon$-**optimal** solution if and only if $f(x) - f(x^*) \leq \varepsilon$.

Assuming that $f$ is convex, we can get arbitrarily close to $x^*$. We would like to find how many steps of GD it would take to achieve $x^t$ $\varepsilon$-optimal.

Let $x = x^t$ and $\delta = x^* - x^t$ in the convexity condition. Then,

$$f(x^*) \geq f(x^t) + \nabla f(x^t)^T (x^* - x^t).$$

This implies by Cauchy-Schwarz:

$$f(x^t) - f(x^*) \leq -\nabla f(x^t)^T (x^* - x^t) \leq |\nabla f(x^t)||x^* - x^t| \leq \varepsilon.$$

Therefore, the gradient must satisfy

$$|\nabla f(x^t)| \leq \frac{\varepsilon}{|x^* - x^t|}.$$

**Definition 21.4**

$f$ is **$\alpha$-strongly convex** for $\alpha > 0$ if and only if

$$\delta^T \nabla f(x) \delta \geq \alpha |\delta|^2,$$

for all $x, \alpha$.

We can show that if $f$ is $\alpha$-strongly convex,

$$f(x+\delta) \geq f(x) + \nabla f(x)^T \delta + \frac{\alpha}{2}|\delta|^2.$$

Let $x = x^*$ and $\delta = x^t - x^*$. Then,

$$f(x^t) \geq f(x^*) + \nabla f(x^*)^T \delta + \frac{\alpha}{2}|\delta|^2 = f(x^*) + \frac{\alpha}{2}|\delta|^2 \implies f(x^t) - f(x^*) \geq \frac{\alpha}{2}|x^t - x^*|^2.$$

This shows that

$$|\nabla f(x^t)|^2 |x^* - x^t|^2 \geq (f(x^t) - f(x^*))^2 \geq \left(\frac{\alpha}{2}|x^t - x^*|^2\right)^2 \implies |\nabla f(x^t)|^2 \geq \frac{\alpha(f(x^t) - f(x^*))}{2}.$$

Also, we previously showed $f(x^{t+1}) - f(x^t) \leq -(1/2\beta)|\nabla f(x^t)|^2$, so

$$f(x^{t+1}) - f(x^*) \leq f(x^t) - f(x^*) - \frac{1}{2\beta}\frac{\alpha}{2}(f(x^t) - f(x^*)) \leq (f(x^t) - f(x^*))\left(1 - \frac{1}{4\kappa}\right),$$

where $\kappa = \beta/\alpha > 1$ is the **condition number** of $f$. This implies that each step of our GD decays the distance between $x^t$ and the global minimum by a factor of $1/\kappa$, so the number of steps required is

$$O\left(\kappa \log \frac{f(x^0) - f(x^*)}{\varepsilon}\right).$$

**Theorem 21.5**

If $f$ is $\beta$-smooth and $\alpha$-strongly convex, then for any $\varepsilon > 0$, there exists constant $c$ such that $x^t$ is $\varepsilon$-optimal for any

$$t \geq c \cdot \kappa \log \frac{f(x^*) - f(x^0)}{\varepsilon}.$$

Note that $\kappa$ roughly tells us how convex the function is; $\beta$ bounds the Hessian eigenvalues from above, and $\alpha$ bounds them from below.

# 22 May 9, 2023

## 22.1 Sublinear Algorithms

**Definition 22.1**

Sublinear algorithms don't need to read the entire input to produce an answer.

Exact sublinear algorithms are usually bad. For example, consider the triangle detection problem:

- The input is a graph $G$ in matrix format. Output YES if there is a triangle and NO otherwise.

If $G$ is completely bipartite, there are no triangles. If $G$ is completely bipartite + one edge, then there are triangles. This shows that it is impossible to consistently produce a correct answer to this problem without reading $\Omega(n^2)$ of the input.

A more reliable approach is to try to approximate the answer:

- **Classical approximations** are used for optimization problems. An $\alpha$-approximation factor indicates an answer within $\alpha$ of OPT.
- **Property testing** is an approximation for decision problems. This should always return YES on a YES-instance. This should return NO on NO-instances that are "very far" from YES-instances, with some high probability. It can return anything on other instances.

## 22.2 Diameter of a point set

This problem demonstrates a classical approximation.

- Input: $n \times n$ matrix $D$ where $D_{i,j}$ denotes a distance between $i$ and $j$. This means that $D$ is symmetric and satisfies the triangle inequality, i.e.,

$$D_{i,j} \leq D_{i,k} + D_{k,j}.$$

- Output: $D^*$ which is the largest distance between two points, i.e., the maximum entry of the matrix.

The input has size $N = n^2$. Any sublinear algorithm is $o(N) \in o(n^2)$. An algorithm with approximation factor $c$ should satisfy

$$\frac{D^*}{c} \leq \tilde{D} \leq D^*.$$

---

Algorithm:

- pick an index $i^* \in [n]$.
- read the $i^*$th row, output $\tilde{D} = \max_j D_{i^*,j}$.

The runtime is $O(n)$, so it is sublinear. We claim $\tilde{D} \geq D^*/2$. If the true diameter has a point in the $i^*$th row, then $\tilde{D} = D^*$. Otherwise, $D_{i^*,a} + D_{i^*,b} \geq D_{a,b} = D^*$, so $\max\{D_{i^*,a}, D_{i^*,b}\} \geq D^*/2$. Therefore, this algorithm is a 2-approximation.

## 22.3 Testing for Connectedness

This problem demonstrates a property testing approximation.

- Input: $G = (V, E)$ on $n$ vertices with maximum degree $d$ in an adjacency list format.
- Output: whether or not $G$ is connected.

Our approximation will return:

- YES if $G$ is connected
- NO if $G$ is not $\varepsilon$-close to being connected
- anything otherwise

**Definition 22.2**

$G$ is $\varepsilon$-close to being connected if it is possible to add $\varepsilon nd$ edges to make $G$ connected.

**Claim 22.3**
if $G$ is not $\varepsilon$-close to connected, then $G$ has $> \varepsilon dn$ connected components.

*Proof.* If not, then we can add edges between them. □

**Claim 22.4**
if $G$ is not $\varepsilon$-close to connected, then $G$ has $\geq \varepsilon dn/2$ connected components of size at most $2/(\varepsilon d)$.

*Proof.* The number of connected components is at least $\varepsilon dn$. So if we suppose otherwise, then the number of connected components with size $\geq 2/(\varepsilon d)$ is $> \varepsilon dn/2$. On the other hand, the total number of vertices is $\leq n$, so this is a contradiction. □

**Claim 22.5**
If $G$ is not $\varepsilon$-close to connected, at least $\varepsilon nd/2$ vertices are in small connected components.

*Proof.* Follows from the previous claim. □

---

Algorithm: let $c \geq 2$ be some constant. Repeat $c/(\varepsilon d)$ times:

- pick a node $s$ uniformly at random
- run BFS from $s$ until either $2/(\varepsilon d)$ have been seen, or a CC of size $< 2/(\varepsilon d)$ has been found. In the first case, continue running the algorithm. In the second case, return NO.
- If the algorithm reaches the end of its iterations, return YES.

The runtime is $O(1/(\varepsilon d) \cdot (2/(\varepsilon d) + 2/\varepsilon)) \in O(1/(d\varepsilon^2))$, which is constant time, so this algorithm is sublinear.

**Claim 22.6**
If $G$ is connected, the algorithm will correctly always return YES. Otherwise, it returns NO with probability at least 3/4.

*Proof.* The YES direction is true. Now suppose $G$ is not $\varepsilon$-close to connected. By the third claim, there are $> \varepsilon dn/2$ nodes in a small component, so the probability that $s$ is in a small CC in one iteration is $> \varepsilon d/2$. Therefore, the probability that none of the iterations sample a vertex in a small component is $(1 - \varepsilon d/2)^{2c/(\varepsilon d)} \leq (1/e)^c \leq 1/4$. □

## 22.4 Sortedness of a List

- Input: List $L = \{x_0, \ldots, x_{n-1}\}$.
- Output: YES if $L$ is sorted, NO if the list is not $\varepsilon$-close from sorted with high probability.

**Definition 22.7**

A list of length $n$ is $\varepsilon$-close to sorted if it is possible to delete $n\varepsilon$ elements and get a sorted list.

---

Algorithm: think of the list as a binary search tree. Pick a random index $i$ and search for $x_i$ by binary searching the list. Return YES if we end up at index $i$ with no inconsistencies found during the search, and NO otherwise.

# 23 May 11, 2023

## 23.1 Streaming Algorithms

Basic idea: have a very large stream of inputs of length $n$, but $o(n)$ memory.

In **sketching**, given stream $X$, we want to compute sketch $C(X)$.