In this lecture, we will first discuss another dynamic programming algorithm. Next we will introduce a mathematical framework useful for the analysis of randomized algorithms ranging from data structures to load balancing. We will then show how we can apply this framework to analyze chain hashing. Finally, we will discuss two applications of hashing where using hashing results in reduced space requirements.

Before we start, let us recall certain structural properties that, when present in a problem, hint to the fact that dynamic programming is a suitable technique.

1. The optimal solution to the problem can be expressed in terms of optimal solutions to subproblems (optimal substructure).
2. There is a natural ordering on subproblems from "smallest" to "largest", together with an easily computed recurrence that allows one to determine the solution to a subproblem from the solutions to smaller subproblems.
3. There is a small (polynomial) number of subproblems.
4. Combining the subproblems involves only polynomial work.

1 Longest common subsequence (LCS)

Recall our chromosome map from Lecture 4: it is a very long string of nucleotide bases which we represent by the symbols $\{A, C, G, T\}$. We want to compare the chromosome maps in two different organisms to see how similar they are.

There are various ways to measure "similarity". Here we will use the *length of the longest common subsequence* of the two strings of nucleotides. We will define this formally shortly, but intuitively, the longer the common subsequence of the two strings, the more "similar" they are.

Definition 1 *Let $X = x_1 x_2 \cdots x_m$ be a sequence of nucleotide bases from A, C, G, T. Then $Z = z_1 z_2 \cdots z_k$ is a subsequence of X if there are indices $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ such that $z_\ell = x_{i_\ell}$ for all $1 \leq \ell \leq k$.*

Note that in this definition the indices in X and Z do **not** have to be contiguous.

Example 1: Let $m = 9$, $X = ACBDABADA$ and $Z = ABDAD$. Then Z is a subsequence of length $k = 5$ of X, with indices $1, 3, 4, 5, 8$.

Example 2: Let X as above, $n = 7$ and $Y = BDCCADA$. Then (by inspection) $Z = BDADA$ is a longest common subsequence of X and Y and its length is $k = 5$.

We will show how to apply dynamic programming to solve this problem. To this end, we have to find optimal substructure and come up with a useful recurrence over the appropriate subproblems.

1.1 Optimal substructure: subproblems and recurrence

We first introduce some notation.

- Let $X_i = x_1 \cdots x_i$ denote the prefix of X that contains the i first symbols of X.
- Let Z denote an LCS of X, Y and let k denote its length.

One thing we might consider in order to come up with subproblems is the pair of final indices from X and Y, that is (x_m, y_n). We now attempt to express Z in terms of subproblems that involve this pair.

- If $x_m = y_n$, then $z_k = x_m$ *(we need to show this)*; then Z_{k-1} is an LCS of X_{m-1}, Y_{n-1} *(we need to show this)*.

  $\Rightarrow$ Thus the length of an LCS of X, Y is simply the length of an LCS of X_{m-1}, Y_{n-1} incremented by 1.

- If $x_m \neq y_n$, then clearly one of the two (or both) will not equal z_k. Thus there are two possibilities:

  – If $x_m \notin Z$, thus $z_k \neq x_m$: in this case, Z is an LCS of X_{m-1} and Y *(we need to show this)*.

  – If $y_n \notin Z$, thus $z_k \neq y_n$: in this case, Z is an LCS of X, Y_{n-1} *(we need to show this)*.

  $\Rightarrow$ Hence the length of the longest common subsequence of X, Y is given by the maximum between the length of the longest common subsequence of X_{m-1}, Y and the length of the longest common subsequence of X, Y_{n-1}.

If we prove that the statements above are true, then we have optimal substructure for our problem.

We remind that Z is an LCS of X, Y of length k. Then

- If $x_m = y_n$: assume for the sake of contradiction that $z_k \neq x_m$. Then if we append $x_m = y_n$ to Z we obtain a longer common subsequence of X, Y contradicting optimality of Z. Hence $z_k = x_m = y_n$.

  So now we need to show that Z_{k-1} is an LCS of X_{m-1}, Y_{n-1}. Assume for the sake of contradiction that there is an LCS Z_1 of X_{m-1}, Y_{n-1} that is longer than Z_{k-1}, that is, has length at least k. Appending x_m to Z_1 yields a common subsequence of X, Y of length at least $k+1$, hence longer than Z.

- If $x_m \neq y_n$, then

  – If $x_m \notin Z$, that is $x_m \neq z_k$, then Z is an LCS of X_{m-1}, Y. For the sake of contradiction suppose that there is a Z' that is an LCS of X_{m-1}, Y and is longer than Z; hence its length is at least $k+1$. Since X_{m-1} is a prefix of X, Z' is also a common subsequence of X, Y, which contradicts optimality of Z.

  – If $y_n \notin Z$, that is $y_n \neq z_k$, then Z must be an LCS of X, Y_{n-1}. This case is entirely similar to the previous one.

So we showed optimal substructure, that is, the optimal solution for the overall problem implies optimal solutions to the subproblems. We can now form the recurrence for the length of an LCS of X, Y using our observations from above.

Let $c(i, j)$ be the length of an LCS of X_i, Y_j, for $0 \leq i \leq m$, $0 \leq j \leq n$. We have

$$c(i,j) = \begin{cases} 0 & \text{, if } i = 0 \text{ or } j = 0 \\ 1 + c(i-1, j-1) & \text{, if } x_i = y_j \\ \max\left\{ c(i-1,j),\ c(i,j-1) \right\} & \text{, if } x_i \neq y_j \end{cases}$$

We want to compute $c(m, n)$.

1.2 Analysis of the algorithm for LCS

We will only fill in a dynamic programming table c of dimensions $(m+1) \times (n+1)$ (you will find pseudocode in your textbook). A few observations are appropriate.

- There is a polynomial number of subproblems: $\Theta(mn)$.

- We can compute the subproblems iteratively row by row, from left to right.

- Total time to fill in c is $\Theta(mn)$ (combining subproblems takes time $\Theta(1)$).
- Total space: $\Theta(mn)$
  - We can reconstruct the optimal solution in $O(m+n)$ time (why?)
  - However, this is not space efficient; recall the size of the chromosome maps... If we only need the length of the LCS (and not the actual LCS), we can improve space requirements to $2n$. Look at the recurrence—you just need two rows of c at a time. (Actually, you need even less, see Exercise 15.4-4.)

2 Balls and Bins problems

We will now introduce a framework that is essential in the analysis of numerous randomized algorithms, ranging from data structures to packet routing and load balancing. In particular, we will use this framework to analyze a hash table. Before we give the general framework, we start with a motivating example.

2.1 The birthday paradox

Suppose we want to find how many people there must be in a lecture room so that it is more likely than not that two people have the same birthday, assuming that people's birthdays are independent and uniformly distributed throughout the year.

So we need to find the probability that two people share the same birthday. It is actually easier to think about the probability that two people do **not** share the same birthday. One way to do this is by considering one person at a time:

- the first person has a birthday;
- the second person has a different birthday with probability $1 - \frac{1}{365}$;
- the third person has a different birthday from the first two (given that the first two have distinct birthdays) with probability $1 - \frac{2}{365}$;
- the m-th person has a different birthday from the first $m-1$ (given that the first $m-1$ have distinct birthdays) with probability $1 - \frac{m-1}{365}$;

Because of independence the probability that all these events happen together is the product

$$\prod_{k=1}^{m-1} \left(1 - \frac{k}{365}\right).$$

Using the inequality $1 + x \leq e^x$ valid for all $x \geq 0$, this is at most

$$\prod_{k=1}^{m-1} e^{-k/365} = e^{-\sum_{k=1}^{m-1} k/365} = e^{-m(m-1)/(2\cdot 365)} \leq e^{-m^2/(2\cdot 365)}$$

Requiring $e^{-m^2/(2\cdot 365)} < 1/2$ yields $m > \sqrt{365 \cdot 2 \ln 2} \approx 22.5$. Thus for $m = 23$ it is more likely than not that two people have the same birthday.

2.2 Balls and bins models and questions

The birthday paradox is an example of a more general mathematical framework that is often formulated in terms of **balls and bins**: We have m balls that are thrown independently into n bins, with the location of each ball chosen *independently and uniformly at random* from all n possibilities. What is the distribution of balls into the bins? For example, in the birthday paradox, bins are days and balls are people and we are asking how big does m have to be so that there is a bin with at least two balls. In Section 2.1, we showed that if m balls are randomly placed into n bins (with $n = 365$ in the birthday example), then, for some $m = \Omega(\sqrt{n})$, it is more likely than not that there is a bin with more than one ball in it.

There are other interesting questions, like

- What is the expected number of balls in a bin?
- What is the expected number of empty bins?
- How many balls are in the fullest bin?

These questions, referred to as occupancy problems, have applications to the analysis of randomized algorithms.

Assumption: We now assume that $m = n$.

Expected number of balls in a bin: For $1 \leq i, j \leq n$, let X_{ij} be the indicator random variable that takes on the value 1 ball i falls into bin j and 0 otherwise. Then for all $1 \leq j \leq n$, $X_j = \sum_{1 \leq i \leq n} X_{ij}$ is the number of balls in bin j and therefore

$$E[X_j] = \sum_{1 \leq i \leq n} \Pr[X_{ij} = 1] = n \cdot \frac{1}{n} = 1.$$

2.3 Maximum load in a bin

For a random variable X, even though $E[X]$ might be small, X may still frequently assume random values that are far higher than its expectation. We often like to know that the probability that this happens is small.

Consider X_j from the previous subsection, in the case where $n = m$. We are now interested in the maximum load in a bin. We showed that $E[X_j] = 1$ but a trivial upper bound for the maximum load is n: all balls fall into the same bin. However, given our experiment, in practice we expect the loads to be more distributed among the n bins. How far is the load of the fullest bin from its expected load?

We will form a probabilistic statement about the maximum load in a bin. Specifically, we will upper bound the maximum load in any bin with probability that goes to 1 as n grows large.

Lemma 1 *When n balls are thrown independently and uniformly at random in n bins, the probability that the maximum load is more than $k^* = 3 \ln n / \ln \ln n$ is at most $1/n$ for sufficiently large n.*

Proof. The idea behind the proof in two lines is: first we upper bound the probability that any bin contains more than k balls. Then we find k^* such that this probability is at most $1/n$ (obviously we want k^* to be as small as possible).

The probability that any fixed bin receives k balls is $\binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}$. Then the probability that a fixed bin receives at least k balls is

$$\sum_{\ell=k}^{n} \binom{n}{\ell} \left(\frac{1}{n}\right)^{\ell} \left(1 - \frac{1}{n}\right)^{n-\ell} \leq \sum_{\ell=k}^{n} \binom{n}{\ell} \left(\frac{1}{n}\right)^{\ell} \leq \sum_{\ell=k}^{n} \left(\frac{n \cdot e}{\ell}\right)^{\ell} \left(\frac{1}{n}\right)^{\ell} = \sum_{\ell=k}^{n} \left(\frac{e}{\ell}\right)^{\ell}$$

For the second upper bound, we used that $\binom{n}{\ell} \leq \left(\frac{n \cdot e}{\ell}\right)^{\ell}$, which follows from $\binom{n}{\ell} \leq \frac{n^{\ell}}{\ell!}$ and Stirling's approximation formula for the factorial (see Lecture 1). We now further upper bound this probability as

$$\sum_{\ell=k}^{n} \left(\frac{e}{\ell}\right)^{\ell} \leq \sum_{\ell=k}^{\infty} \left(\frac{e}{\ell}\right)^{\ell} \leq \sum_{\ell=k}^{\infty} \left(\frac{e}{k}\right)^{\ell} = \left(\frac{e}{k}\right)^{k} \sum_{\ell=0}^{\infty} \left(\frac{e}{k}\right)^{\ell} = \left(\frac{e}{k}\right)^{k} \cdot \frac{1}{1 - e/k}$$

We now have a closed form upper bound for the probability that a fixed bin contains at least k balls. The probability that *any* fixed bin contains at least k balls is given by a union bound over all bins. Hence

$$\Pr[\text{any bin contains at least } k \text{ balls}] \leq n \left(\frac{e}{k}\right)^{k} \cdot \frac{1}{1 - e/k}.$$

So now we want to find k^* as small as possible such that for large enough n

$$\Pr[\text{any bin contains at least } k^* \text{ balls}] \leq \frac{1}{n}.$$

After some algebra and, for example, by experimenting with different values of k, we obtain $k^* = \frac{3 \ln n}{\ln \ln n}$. □

2.4 Hashing

Consider the following setting:

- There is a universe U of all possible elements (e.g., all people), with $|U| = N$ and N is huge.
- We are trying to keep track of a set $S \subseteq U$ (e.g., all the people in a lecture hall or all students in a university) whose size is typically a negligible fraction of U; that is, $|S| = m$ and $m \ll N$.
- We want to maintain a data structure so that we can **Insert**, **Delete** and **Lookup** elements from S efficiently.

A data structure that accomplishes the above is called a **dictionary**. Note that an important aspect of our problem is that U is enormous. Thus we cannot simply maintain an array of size $|U|$ where entry i is 1 if element i of U is in S and 0 otherwise. In fact, we won't be able to maintain an array whose size is anywhere close to U.

Question: can we still implement a dictionary that supports these three basic operations almost as quickly as when U is relatively small?

It turns out that we can by using **hashing**. The basic idea of hashing, as explained in the following section, is to work with an array of size $|S|$ rather than one comparable to $|U|$, which is huge.

2.5 Hash functions, hash tables and chain hashing

- **Hash function:** A hash function is a deterministic mapping from one set into another that appears random. For example, mapping people into their birthdays can be thought of as a hash function. In general, a hash function is a mapping

  $$h : \{0 \ldots N-1\} \to \{0, \ldots, n-1\},$$

  where, typically, $N \gg n$.

- **Hash table:** A hash table is an array of n entries. If we want to add an element x from U to the set S, we simply place x in position $h(x)$ in the hash table H. This would work very well if, for all $x, y \in S$ such that $x \neq y$, $h(x) \neq h(y)$. Then Lookup(x) takes constant time: just check $H[h(x)]$; if it is x, then we know that $x \in S$, otherwise $H[h(x)]$ would be empty.

- **Collision:** However, in general, there can be two distinct elements $x, y \in S$ such that $h(x) = h(y) = i$ (for example, two of the 60 people in the lecture hall have the same birthday). In this case, we have a *collision*, since these elements are mapped to the same entry in the hash table, that is, entry $H[i]$.
- **Chain hashing:** The easiest way to deal with collisions is to maintain a linked list at position i of H of all elements $s \in S$ with $h(s) = i$. This approach is called chain hashing because items that collide are chained together in a linked list. Then Lookup(x) would work as follows
  - compute $h(x)$
  - scan the linked list at position $H[h(x)]$ to see if x is present

Time required for Lookup(x): proportional to the time to compute $h(x)$ **plus** the length of the linked list at $H[h(x)]$.

Goal: find a good hash function, that is, one that "spreads out" the elements being added, so that collisions are minimized and no one entry of the hash table contains too many elements.

Note that no fixed deterministic hash function can guarantee the above: an adversary can always devise a set of elements that all hash to the same location. Hence randomization will be critical for designing such good hash functions. These hash functions rely on number theoretic facts and there is a great deal of theory behind their design which we will not discuss here. Instead, we will henceforth *assume* that a hash function maps elements into hash table locations in a fashion that appears random, so that the location of each element is independently and uniformly distributed. While such completely random hash functions are unrealistic, they do provide a good rough idea of how hashing schemes perform.

2.6 Balls-in-bins analysis for Lookup(x)

We can naturally analyze hash tables by using a balls and bins model:

- balls correspond to elements from S
- bins correspond to the entries in our hash table
- ball i falls in bin j iff $h(i) = j$.

Hence the number of balls in a bin represents the number of collisions for the particular value of the hash function. Based on our previous discussion, consider a hash function h such that

- for each $i \in U$, $\Pr[h(i) = j] = 1/n$ for $0 \leq j \leq n-1$;
- the values of $h(i)$ are independent of each other.

Note that $h(i)$ is fixed for every i: it takes on *one* of the n possible values independently and with equal probability. Given this hash function, the m balls fall into the n bins independently and uniformly at random, just like in the balls and bins model discussed in Section 2.3.

Let's revisit the search time for an element $x \in U$ when there are n bins and m balls (elements in S). As mentioned before, Lookup(x) first computes $h(x)$, and then scans the linked list at entry $H[h(x)]$ to see if x is present in this list. Therefore, the expected time required for operation Lookup(x) is determined by the expected number of balls in bin $j = h(x)$. Note that $h(x)$ takes on any of the n possible values with equal probability. So consider a fixed bin j for $1 \leq j \leq n$. We define the indicator random variable $X_{ij} = 1$ iff ball i falls into bin j; then $\Pr[X_{ij}] = 1/n$ and the random variable $X_j = \sum_{i=1}^{m} X_{ij}$ gives the number of balls in bin j. If we search for an element x such that

- x is **not** in S: the expected number of balls (other elements) in bin $j = h(x)$ is

$$E[X_j] = \sum_{i=1}^{m} \Pr[X_{ij} = 1] = m/n.$$

- x is **in** S: then the expected number of **other** balls (elements) in bin j is

$$E[X_j] = \sum_{i=1}^{m-1} \Pr[X_{ij} = 1] = (m-1)/n,$$

  thus the expected number of balls in the bin where x falls into is $1 + (m-1)/n$.

Case $n = m$

Now suppose we choose $n = m$ entries for our hash table. Thus there are $n = m$ bins in our balls and bins model.

- **Expected time** for Lookup: from our analysis above, the expected number of elements we must search through is $O(1)$. Thus, if computing a hash value takes constant time, then the total expected time for Lookup is constant.
- **Maximum time** for Lookup: by our balls and bins analysis, the maximum time is proportional to the maximum load of any bin. As shown in Section 2.3, this is $\Theta(\ln n / \ln \ln n)$ with probability close to 1 for large enough n. So with high probability, this is the maximum time for a Lookup operation in a hash table with $m = n$ entries.

Note that, **without hashing**, if we wanted to implement Lookup(x), we could keep an array with m entries where the elements in S would appear sorted. Then we could perform binary search for x and in $O(\log m)$ time we could check if x appears in S. So the time required by Lookup improves by using hashing.

The main drawbacks of chain hashing are that Lookup is still much slower in the worst case than it is in the average case, and if we use n bins for n items, several bins will be empty, potentially leading to wasted space.

2.7 Applications of Hashing I: Fingerprints

We will now discuss an application where we use hashing to save space. Specifically, consider a password checker which prevents people from using common, easily cracked passwords by keeping a dictionary of such "bad" passwords. When a user tries to set up a password, the application would like to check if the requested password belongs to the set of "bad" passwords.

We will use hashing differently in this case. Suppose that a password is eight ASCII characters which requires 64 bits (8 bytes) to represent. We will use a hash function to map each password into a 32 bit string. This string will serve as a short **fingerprint** of the password. We will keep the fingerprints in a sorted list. To check if a proposed password is a "bad" one, we do the following:

1. calculate its fingerprint
2. use binary search to find it in the list of fingerprints; if found, we declare the password "bad" and ask the user to enter a new one.

Note the possibility of a **false positive**: a "good" password hashes to the same fingerprint as a "bad" password and thus gets rejected!

- *Question: Is x a bad password?*

- Answer "yes" when we should answer "no" yields a **false positive**: we reject a good password because it hashes to the same fingerprint as a bad password. In this case, our algorithm is too conservative but this is probably acceptable so long as false positives do not occur too often.
- Note that this is the **only** type of mistake that can happen in our setting: we will not answer "no" when we should answer "yes" (no **false negatives**). That is, we will never accept a bad password (why?).

Approximate set membership

How did we come up with the value 32 for the length of the fingerprint in the previous section? To answer this question, it is useful to first note that our password checker belongs to a broad class of problems, called *approximate set membership* problems, with the following general description.

- Given: A set $S = \{s_1, \ldots, s_m\}$ of elements from a large universe U.
- Goal: represent elements so that
  - we can answer quickly queries of the form: "Is $x \in U$ an element of S?";
  - the representation occupies as little space as possible;
  - *occasional* false positives are allowed to save space.

In our example, S is the set of "bad" passwords.

Question: How large should the range of the hash function for the fingerprints be? Equivalently, how many bits should we use to create a fingerprint while maintaining an acceptable false positive probability?

Answer: Let b be the number of bits used by our hash function to map the m passwords into fingerprints, thus $h : \{0, 1, \ldots, m-1\} \to \{0, \ldots, 2^b - 1\}$. We want to choose b so that the probability of a false positive is acceptable, e.g., at most $1/m$. To determine b, we need determine the probability of a false positive.

- There are 2^b possible strings of length b. Fix a password from S;
  - the probability that a "good" password has the same fingerprint as the fixed "bad" password is $1/2^b$.
  - Hence the probability that a "good" password does **not** have the same fingerprint as a fixed password from S is $1 - 1/2^b = p$.
- the probability that a "good" password does **not** have the same fingerprint as *any* of the m "bad" passwords in S is p^m [1] .

$\Rightarrow$ the probability that a "good" password has the same fingerprint as (at least) one of the m "bad" passwords from S is $1 - p^m$.

$\Rightarrow$ false positive probability is $1 - p^m = 1 - (1 - 1/2^b)^m \geq 1 - e^{-m/2^b}$.

So if we want to make the probability of a false positive less than, say, a constant c, we require

$$1 - e^{-m/2^b} \leq c \Rightarrow b \geq \log_2 \frac{m}{\ln(1/(1-c))}.$$

So $b = \Omega(\log_2 \frac{m}{\ln(1/(1-c))})$ bits.

[1] To simplify the analysis, we assume here that all passwords in S have *distinct* fingerprints. This assumption is not necessary: we can prove that it does not affect our subsequent analysis in a significant way.

On the other hand, suppose we use $b = 2\log_2 m$. Then plugging back into the original formula for the probability of false positive, which is $1 - (1 - 1/2^b)^m$, we get

$$1 - (1 - \frac{1}{m^2})^m \leq 1 - (1 - \frac{1}{m}) = \frac{1}{m},$$

where we used that $(1-x)^m \geq 1 - mx$, for $0 \leq x \leq 1$. Thus if our dictionary has $m = 2^{16}$ words (that is, 2^{16} bad passwords), we can use a hash function that maps each of the m passwords to 32 bits and have a false probability of about $1/2^{16}$.

2.8 Applications of Hashing II: Bloom filter

A Bloom filter is the following data structure

- It consists of an array of n **bits**, $A[0]$ to $A[n-1]$, all set to 0 initially.
- It uses k independent hash functions $h_1, \ldots, h_k$ with range $[0, n-1]$. For $1 \leq i \leq k$, h_i maps every element in the universe to a number in $[0, n-1]$, independently and uniformly at random.

Suppose we want to use a Bloom filter to represent a set $S = \{s_1, \ldots, s_m\}$ from a large universe U. For each element $s \in S$ we do the following:

- For $1 \leq i \leq k$, compute $h_i(s)$ and set $A[h_i(s)] = 1$.

Note that a bit location may be set multiple times but only the first change has an effect. To check membership of an element x in S we check whether all array locations $A[h_1(x)]$, ..., $A[h_k(x)]$ are set to 1.

- If no, then clearly $x \notin S$.
- If yes, we say $x \in S$ even though we might be wrong: the bits might have been set by other elements $s \in S$ while $x \notin S$.

So again we may have a **false positive** to the question "does x belong to S". We will not analyze the false positive probability f here. However we note that by trading off the false positive probability and the space requirements, we can use a Bloom filter to achieve the following:

- Space $\Theta(m)$ to represent a set S of m elements. That is, we only use a constant number of bits per element in S. This is in contrast with the fingerprints approach of the previous section, where we needed $\Omega(\log m)$ bits per element in S.
- Constant false positive probability f.

For example, for $n = 8m$, we get $k = 6$ and $f \approx 0.02$: if saving space is critical then we may be willing to accept false positive rates of 1% or 2%.