Competitive
Programming and
Mathematics
Society

Hashing In The Real World

Kyle and Freddie

Table of contents

Let’s Brainstorm

- Consider the average case time complexities of array operations

Let's Brainstorm

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)

Let’s Brainstorm

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?

Let’s Brainstorm

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?
  - Bob the Builder, Yes We Can!
- Idea: If we give each element a ’fixed’ position in an array, then when we insert that element, we don’t need to move other elements around $\implies O(1)$ insertion.

Let’s Brainstorm

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?
  - Bob the Builder, Yes We Can!
- Idea: If we give each element a ’fixed’ position in an array, then when we insert that element, we don’t need to move other elements around $\implies O(1)$ insertion.
  - Similarly for deletion.

Enter - The Hash Function

- A hashing function give us this ’fixed’ position within an array.

Enter - The Hash Function

- A hashing function give us this ’fixed’ position within an array.

```
int hash(int N /* (size of our array we're hashing into) */, int elem) {
    int hash = elem * elem;
    return hash % N;
}
```

- Let’s insert into an array of size 8 using this hash function. Now we can call this ’array’ a hash table. Consider 10, 12, 13:

Enter - The Hash Function

- A hashing function give us this ’fixed’ position within an array.

```c
int hash(int N /* (size of our array we're hashing into) */, int elem) {
    int hash = elem * elem;
    return hash % N;
}
```

- Let’s insert into an array of size 8 using this hash function. Now we can call this ’array’ a hash table. Consider 10, 12, 13:

12	13			10			
0	1	2	3	4	5	6	7

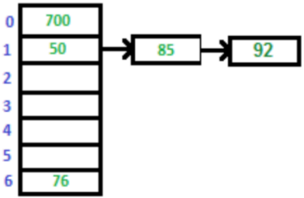
- Now it’s your turn - give us some integers to insert!

Nothing's Perfect...

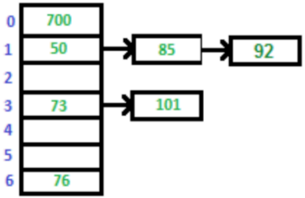
- As we saw from your examples, we run into issues when distinct elements have the same hash! Let's resolve these 'hash collisions':

Nothing’s Perfect...

- As we saw from your examples, we run into issues when distinct elements have the same hash! Let’s resolve these ‘hash collisions’:
    - A natural idea is to chain these elements with identical hashes in a linked list:
- We can chain these together very much like a simple linked list!



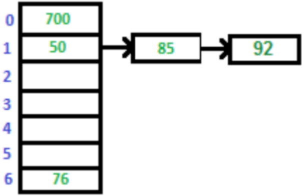
Inser 92 Collision
Occurs, add to chain



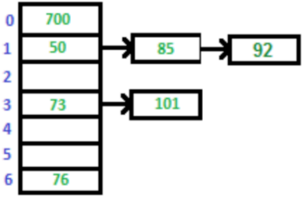
Insert 73 and 101

Nothing’s Perfect...

- As we saw from your examples, we run into issues when distinct elements have the same hash! Let’s resolve these ‘hash collisions’:
  - A natural idea is to chain these elements with identical hashes in a linked list:
- We can chain these together very much like a simple linked list!



Inser 92 Collision
Occurs, add to chain



Insert 73 and 101

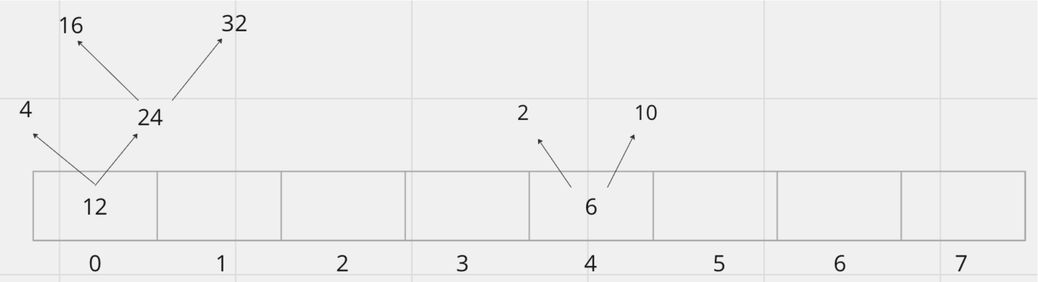
- Can we do **better**?
  - If all elements hash to a single slot (in the worst case), we’ll have a $O(n)$ search, insert *and* delete ... so what was the point?!

A (not so) Subtle Improvement

- A natural improvement on a linked list is a self-balancing tree (such as an AVL tree), which will guarantee us $O(\log n)$ search, insert and delete in the worst case.

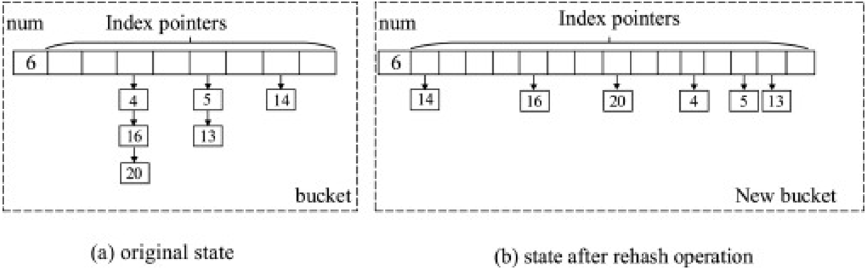
A (not so) Subtle Improvement

- A natural improvement on a linked list is a self-balancing tree (such as an AVL tree), which will guarantee us $O(\log n)$ search, insert and delete in the worst case.
- Going back our earlier example, even in the worst case, we'd have a hashtable that looks visually like:



Implementation of HashMap in C++

- The implementation of HashMap utilises separate chaining to deal with collisions.
- It dynamically resizes the underlying array when the total number of elements inserted is greater than the size of the array.
- This yields an amortized (averaged-out) complexity of $O(1)$ for all operations on a HashMap in C++.



(a) original state

(b) state after rehash operation

Implementation of HashMap in C++

- Let’s declare a HashMap which maps a string key to an integer value.

```
unordered_map<string, int> m;
```

Implementation of HashMap in C++

- Let's declare a HashMap which maps a string key to an integer value.

```
unordered_map<string, int> m;
```

- We can now assign an integer value to our string key.

```
unordered_map<string, int> m;
m["bob"] = 5;
```

Implementation of HashMap in C++

- Let’s declare a HashMap which maps a string key to an integer value.

```
unordered_map<string, int> m;
```

- We can now assign an integer value to our string key.

```
unordered_map<string, int> m;
m["bob"] = 5;
```

- As you may have guessed, this is a powerful tool that let’s us access a value not necessarily based on its index as in array

Simple Example

You are given an integer N, then N names of everyone in the class and their corresponding age. You will then be given an integer Q, and then Q names. You are then tasked with printing the total sum of the age of everyone in the given subset.

- **Sample Input:**
  3
  Bob 12
  Kelly 19
  George 20
  2
  Kelly George
- **Sample Output:** 39

Simple Example

```
unordered_map<string, int> ageMap;
int n; cin >> n;
for (int i = 0; i < n; i++) {
    string s; cin >> s;
    int age; cin >> age;
    ageMap[s] = age;
}

int q; cin >> q;
int sum = 0;
for (int i = 0; i < q; i++) {
    int name; cin >> name;
    sum += ageMap[name];
}
cout << sum;
```

Other Uses of Hashing

- Hashing can be used to verify that our data has not been corrupted.
  - This stems from the fact that when even one single character has been changed, the resulting hashes are entirely different.

Other Uses of Hashing

- Hashing can be used to verify that our data has not been corrupted.
  - This stems from the fact that when even one single character has been changed, the resulting hashes are entirely different.

- Hashing is used to store our passwords.
  - Most websites store the hashed version of our passwords, so when users attempt to log in, the system applies the same hash to the provided password and simply checks if the result of the two hashes is the same.

Other Uses of Hashing

- Hashing can be used to verify that our data has not been corrupted.
  - This stems from the fact that when even one single character has been changed, the resulting hashes are entirely different.

- Hashing is used to store our passwords.
  - Most websites store the hashed version of our passwords, so when users attempt to log in, the system applies the same hash to the provided password and simply checks if the result of the two hashes is the same.
  - This reduces the effect of data breaches.

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).
  3. Insert all integers x into a HashMap in $O(1) \cdot n$, then for each element, search for its complement $t - x$ also in $O(1) \cdot n \implies O(n)$!

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).
  3. Insert all integers x into a HashMap in $O(1) \cdot n$, then for each element, search for its complement $t - x$ also in $O(1) \cdot n \implies O(n)$!
- Hence, design an $O(n^2)$ algorithm to determine if any three unique elements in an n-sized array sum to a given target t.

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).
  3. Insert all integers x into a HashMap in $O(1) \cdot n$, then for each element, search for its complement $t - x$ also in $O(1) \cdot n \implies O(n)$!
- Hence, design an $O(n^2)$ algorithm to determine if any three unique elements in an n-sized array sum to a given target t.
  - Idea: Consider all pairs a, b and search in hashtable for their complement $x - a - b$.

A Classic Application

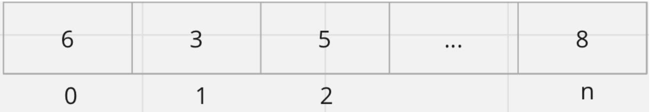
- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).
  3. Insert all integers x into a HashMap in $O(1) \cdot n$, then for each element, search for its complement $t - x$ also in $O(1) \cdot n \implies O(n)$!
- Hence, design an $O(n^2)$ algorithm to determine if any three unique elements in an n-sized array sum to a given target t.
  - Idea: Consider all pairs a, b and search in hashtable for their complement $x - a - b$.
  - *But*, is that diligent enough?

A Classic Application

- The Classic Two-Sum: Given an array of n integers, return two elements that sum to a given target t.
  1. ~~Naive solution of comparing all $\binom{n}{2}$ pairs $\implies O(n^2)$ comparisons.~~
  2. Merge sorting, and then two-pointer approach $\implies O(n \log n)$.
     Alternatively, insert all integers into AVL tree and for each element x, search for its complement $t - x$ (still $O(n \log n)$).
  3. Insert all integers x into a HashMap in $O(1) \cdot n$, then for each element, search for its complement $t - x$ also in $O(1) \cdot n \implies O(n)$!
- Hence, design an $O(n^2)$ algorithm to determine if any three unique elements in an n-sized array sum to a given target t.
  - Idea: Consider all pairs a, b and search in hashtable for their complement $x - a - b$.
  - *But*, is that diligent enough?

Three Sum Review

Let’s test our idea on the following array searching for **15.**



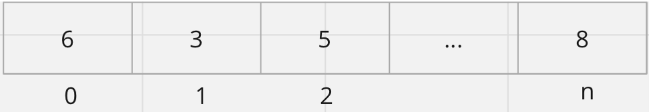
6	3	5	...	8
0	1	2		n

Our first pair is 3and6, so we’ll search in the hashtable for the presence of $15 - 3 - 6 = 6$.

If we aren’t careful, we may find the 6 (from index 0 in the original array) in our hash table and return a false positive!

Three Sum Review

Let's test our idea on the following array searching for **15**.



6	3	5	...	8
0	1	2		n

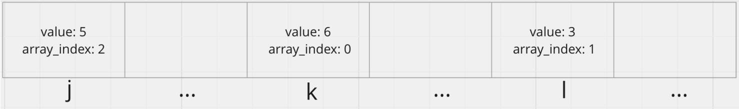
Our first pair is $3 and 6$, so we'll search in the hashtable for the presence of $15 - 3 - 6 = 6$.

If we aren't careful, we may find the 6 (from index 0 in the original array) in our hash table and return a false positive!

A clean way to deal with this is to store the original indices of terms (in the array) within our hashtable so we can check for these duplicates.

Three Sum Review

- A hypothetical hashtable may look like:



value: 5 array_index: 2		value: 6 array_index: 0		value: 3 array_index: 1	
j	...	k	...	l	...

- Whenever we find a complement c of a pair (a, b), we ensure $array\_index(c) \neq array\_index(a)$ and $array\_index(c) \neq array\_index(b)$, so that we're choosing 3 distinct terms.

4-4-4-4 Sum

Your next challenge: What about for a group of 4 integers - can you do this in $O(n^2)$?

4-4-4-4 Sum

Your next challenge: What about for a group of 4 integers - can you do this in $O(n^2)$?

- We can reduce this problem to asking: ‘are there any mutually disjoint’ pairs that sum up to the target t.
    - Mutually disjoint meaning the two pairs don’t share any common elements.

4-4-4-4 Sum

Your next challenge: What about for a group of 4 integers - can you do this in $O(n^2)$?

- We can reduce this problem to asking: ‘are there any mutually disjoint’ pairs that sum up to the target t.
  - Mutually disjoint meaning the two pairs don’t share any common elements.
- We insert the sum of all $\frac{n(n-1)}{2}$ pairs into a hash table along with the indices of each element in the pair. Now for each pair (a, b), we search for its complement $x - a - b$ in our hashtable.
  - If found, we ensure that all four elements across the two pairs are unique by comparing their $array\_indices$.

4-4-4-4 Sum

Your next challenge: What about for a group of 4 integers - can you do this in $O(n^2)$?

- We can reduce this problem to asking: ‘are there any mutually disjoint’ pairs that sum up to the target t.
  - Mutually disjoint meaning the two pairs don’t share any common elements.
- We insert the sum of all $\frac{n(n-1)}{2}$ pairs into a hash table along with the indices of each element in the pair. Now for each pair (a, b), we search for its complement $x - a - b$ in our hashtable.
  - If found, we ensure that all four elements across the two pairs are unique by comparing their $array\_indices$.
- Clearly $O(n^2)$ in the average case since there are $O(n^2)$ pairs and insertion and searching both take constant time.
- Even in worst case, we will have $O(n^2 \log n)$ using our AVL tree collision resolution mechanism.

Sidetrack: Prefix Sum

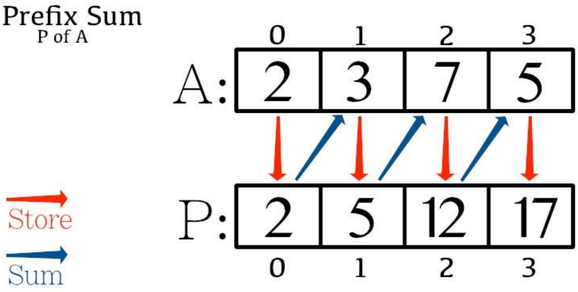
- There is a simple yet powerful technique that allows for the fast calculation of sums of elements in a given slice (contiguous segments of an array).

Sidetrack: Prefix Sum

- There is a simple yet powerful technique that allows for the fast calculation of sums of elements in a given slice (contiguous segments of an array).
- Its main idea uses prefix sums which are defined as the consecutive totals of the first $0, 1, 2, \ldots, n$ elements of an array.

Sidetrack: Prefix Sum

- There is a simple yet powerful technique that allows for the fast calculation of sums of elements in a given slice (contiguous segments of an array).
- Its main idea uses prefix sums which are defined as the consecutive totals of the first $0, 1, 2, \ldots, n$ elements of an array.



Sidetrack: Prefix Sum

- Why is this useful?

Sidetrack: Prefix Sum

- Why is this useful?
- We can now find the sum between two elements in a given array in $O(1)$!.

Sidetrack: Prefix Sum

- Why is this useful?
- We can now find the sum between two elements in a given array in $O(1)$!.

Calculate Prefix Sum Array in $O(N)$

```
for $i = 1$ to $N$ do
    $pref[i] = pref[i-1] + arr[i]$
end for
```

Sidetrack: Prefix Sum

- Why is this useful?
- We can now find the sum between two elements in a given array in $O(1)$!.

Calculate Prefix Sum Array in $O(N)$

```
for $i = 1$ to $N$ do
    $pref[i] = pref[i-1] + arr[i]$
end for
```

Now to find the sum between element L and element R we can simply do

$$pref[R] - pref[L-1]$$

Candy

You are first given two integers N and K, you are then given an N element array $candy$, where $candy[i]$ represents the number of candies you gain/lose by walking over index i (either someone robs you or someone donates to you). Since you can't teleport, you can only walk in a contiguous region and since you are also lazy, you want to travel the shortest distance. Where should you start and end such that you will gain exactly K candies?

Sample Input:

```
5 3
2 -1 5 2 -4
```

Sample Output:

```
3 5
```

Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?

Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?
- Prefix Sum!

Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?
- Prefix Sum!
- This problem can now be reformulated as $prefCandy[r] - prefCandy[l-1] = K$. Now this problem looks much more solvable.

Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?
- Prefix Sum!
- This problem can now be reformulated as $prefCandy[r] - prefCandy[l-1] = K$. Now this problem looks much more solvable.
- Suppose now we are at index i, we want to find an index j such that $j < i$ and that $prefCandy[j] = prefCandy[i] - K$, then we would have found the two endpoints of the region.

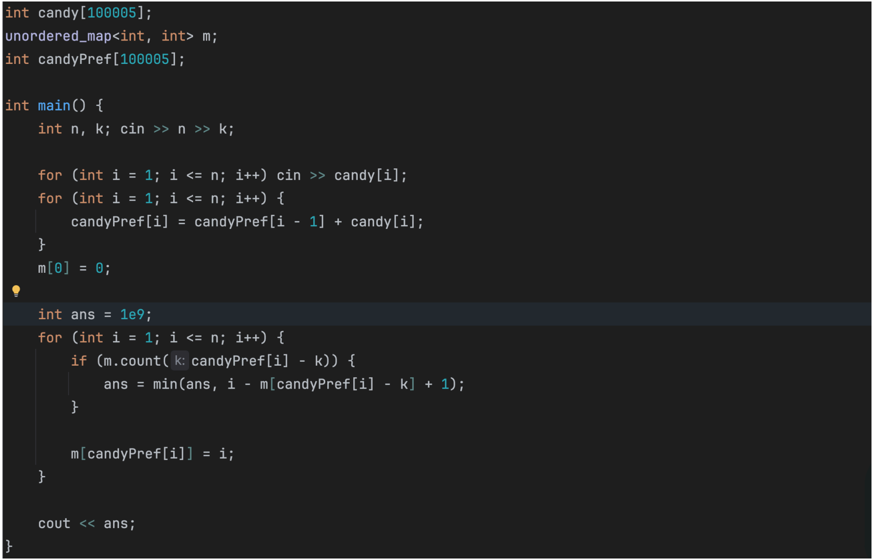
Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?
- Prefix Sum!
- This problem can now be reformulated as $prefCandy[r] - prefCandy[l-1] = K$. Now this problem looks much more solvable.
- Suppose now we are at index i, we want to find an index j such that $j < i$ and that $prefCandy[j] = prefCandy[i] - K$, then we would have found the two endpoints of the region.
- How do we find an index j that satisfies our condition? We use a map!

Candy

- We want to find a contiguous region where the sum is exactly K. More formally, $\sum_{i=l}^{r} candy[i] = K$.
- How might we simplify this expression?
- Prefix Sum!
- This problem can now be reformulated as $prefCandy[r] - prefCandy[l-1] = K$. Now this problem looks much more solvable.
- Suppose now we are at index i, we want to find an index j such that $j < i$ and that $prefCandy[j] = prefCandy[i] - K$, then we would have found the two endpoints of the region.
- How do we find an index j that satisfies our condition? We use a map!
- Our HashMap maps a prefix sum value to the index where this prefix value appears. If m is our HashMap, then $m[prefCandy[i]] = i$.
- Then at an index i, we simply have to query for $m[prefCandy[i] - K]$ to see if a corresponding matching index exists, if it exists, we have found a valid contiguous region that sums to K.

Candy



```
int candy[100005];
unordered_map<int, int> m;
int candyPref[100005];

int main() {
    int n, k; cin >> n >> k;

    for (int i = 1; i <= n; i++) cin >> candy[i];
    for (int i = 1; i <= n; i++) {
        candyPref[i] = candyPref[i - 1] + candy[i];
    }
    m[0] = 0;

    int ans = 1e9;
    for (int i = 1; i <= n; i++) {
        if (m.count( k: candyPref[i] - k)) {
            ans = min(ans, i - m[candyPref[i] - k] + 1);
        }

        m[candyPref[i]] = i;
    }

    cout << ans;
}
```

Attendance and Feedback :D

Further events

Please join us for:

- IMC Coding Competition Wednesday Next Week! @ Mathews Theatre B
- Our Next Programming Workshop in W7 (stay updated on our socials!)

Any Final Questions?

- Thank you for coming!