String Hash algorithms

Louis vd Walt

Applications of string hashing

- Cryptography and passwords
- Hashmaps
- Filesystems
- Databases
- Checksums
- Pattern matching

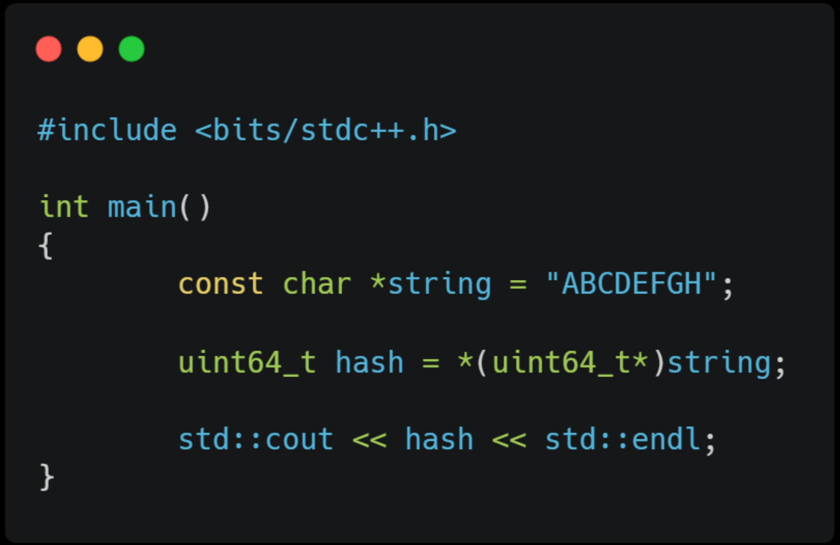
We’ll be taking a look at algorithms for Hashmaps and pattern matching

Why something other than std::hash()?

- 9/10 times you’ll be better off using std::hash.
    - std::hash most likely uses intrinsic instructions
    - std::hash is highly optimised over many years by experienced developers
    - Chosen for best balance between speed and collision frequency
- For the other 10%
    - You need something faster
    - You need something specialised to a certain use case
    - std::hash’s implementation is opaque and you need specific results

Algorithm 1: Basic string cast

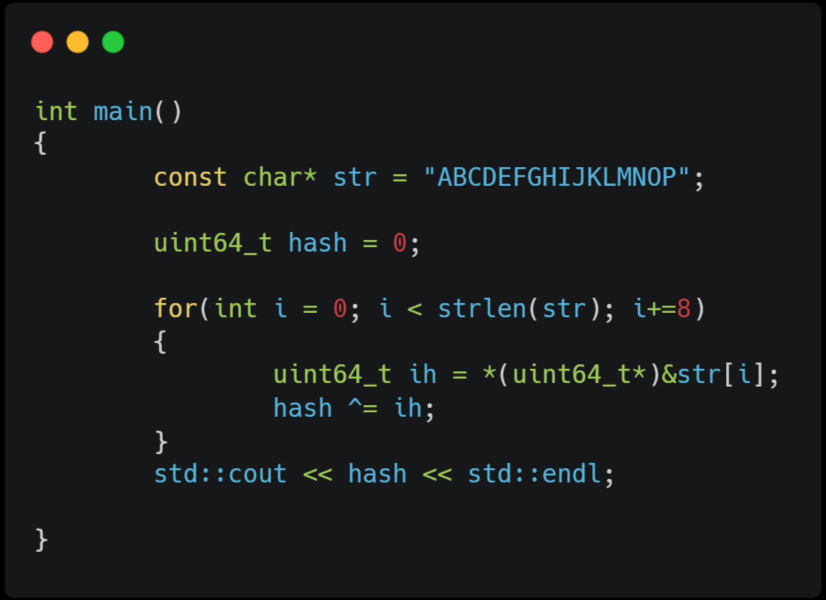
- If strings <= 8 chars
- Might need padding up to 8 chars
- Time complexity: O(1)
- Ease of remembering: 5/5
- Speed compared to std::hash(): Super fast
- Collisions: none



```cpp
#include <bits/stdc++.h>

int main()
{
        const char *string = "ABCDEFGH";

        uint64_t hash = *(uint64_t*)string;

        std::cout << hash << std::endl;
}
```

Algorithm 2: Basic Cast+XOR

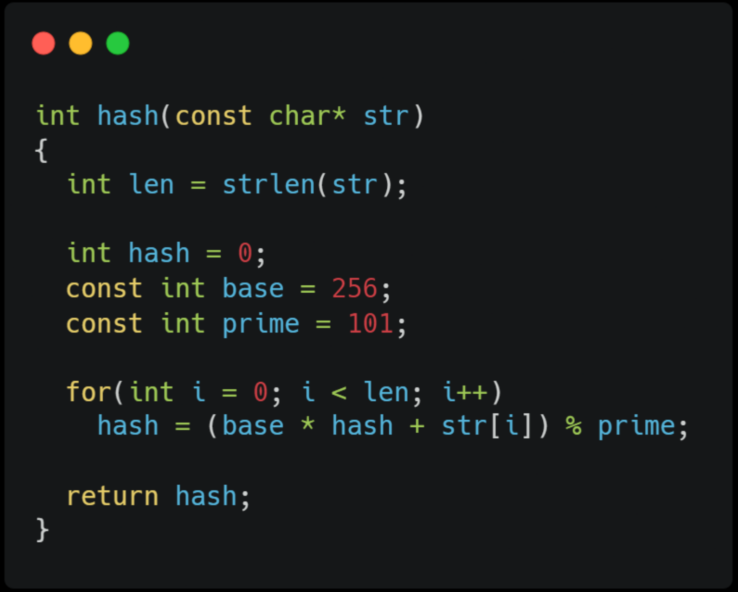
- Modification of previous function
- Needs padded strings up to nearest multiple of 8
- Time complexity: O(n)
- Ease of remembering: 5/5
- Speed compared to std::hash(): Common implementation of std::hash
- Collisions: 3/5



```
int main()
{
        const char* str = "ABCDEFGHIJKLMNOP";

        uint64_t hash = 0;

        for(int i = 0; i < strlen(str); i+=8)
        {
                uint64_t ih = *(uint64_t*)&str[i];
                hash ^= ih;
        }
        std::cout << hash << std::endl;

}
```

Algorithm 3: Common Rabin-Karp hash

- Type of rolling hash
  - Next value can be computed from previous
- Time complexity: O(n) (Hash only)
- Ease of remembering: 5/5
- Speed compared to std::hash: More or less the same
- Collisions: 1/5



```
int hash(const char* str)
{
  int len = strlen(str);

  int hash = 0;
  const int base = 256;
  const int prime = 101;

  for(int i = 0; i < len; i++)
    hash = (base * hash + str[i]) % prime;

  return hash;
}
```

Rabin-Karp Hash Explained

- Let p = 101(or other prime) and b = 255
- Let num(x) = integer value of char
- $hash('str') = num('s') * b^2 + num('t') * b^1 + num('s') * b^1$(mod p)

- $num('s') * b^1 + num('t') * b^0 * \mathrm{b} \; - num('s') * b^1$

Example: Rabin-Karp pattern searching

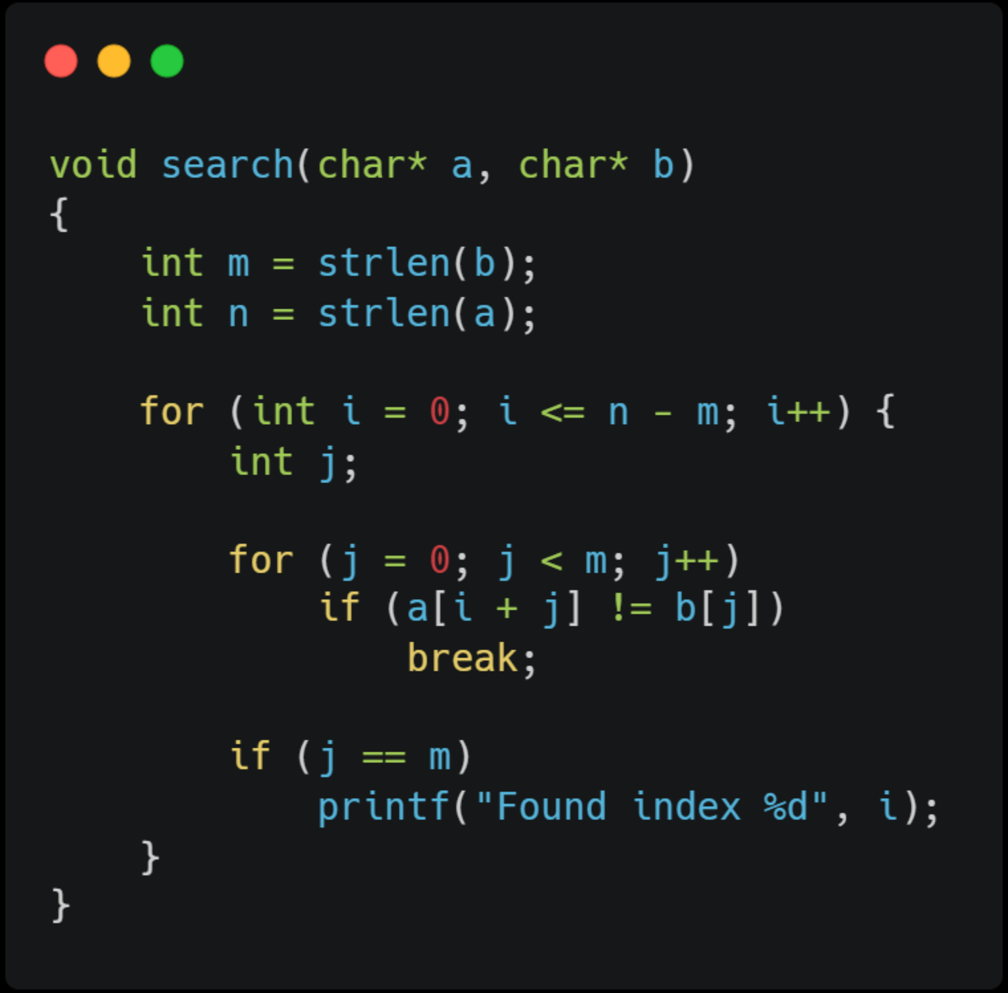
Example question

Given a string a of length n, and a string b of length m, determine the number of occurrences of b as a substring in a where $m < n-1$.

Basic solution in O(nm)

- Loop over string a
- At current position check for a match with b
- If found print position
- Continue to find all occurrences

Basic solution in O(nm)

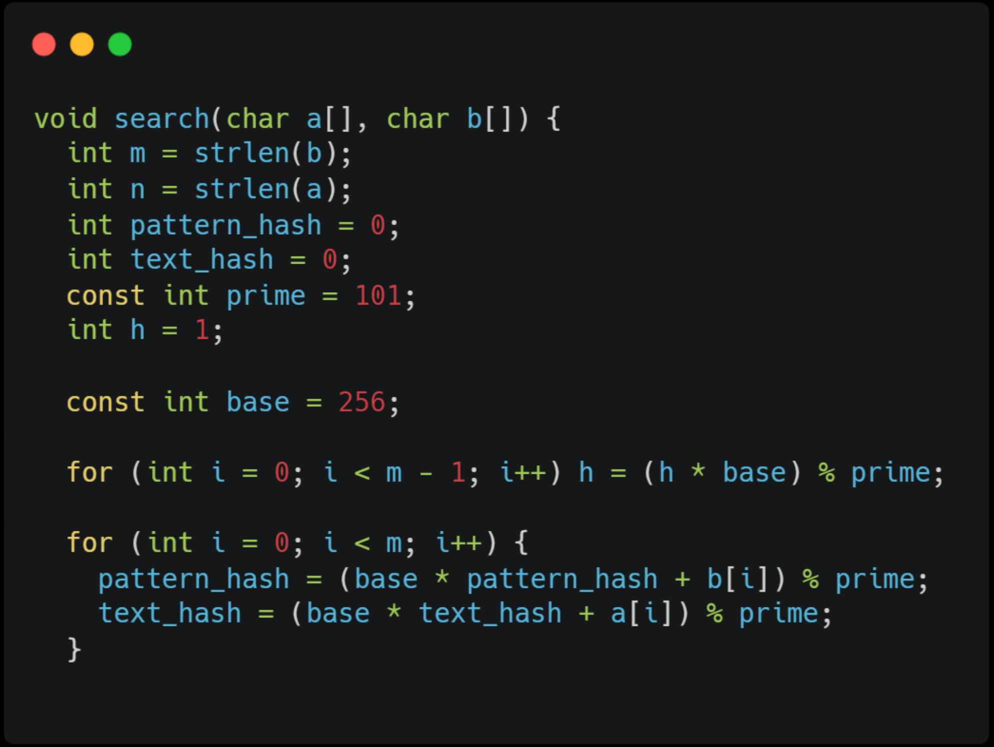


```
void search(char* a, char* b)
{
    int m = strlen(b);
    int n = strlen(a);

    for (int i = 0; i <= n - m; i++) {
        int j;

        for (j = 0; j < m; j++)
            if (a[i + j] != b[j])
                break;

        if (j == m)
            printf("Found index %d", i);
    }
}
```

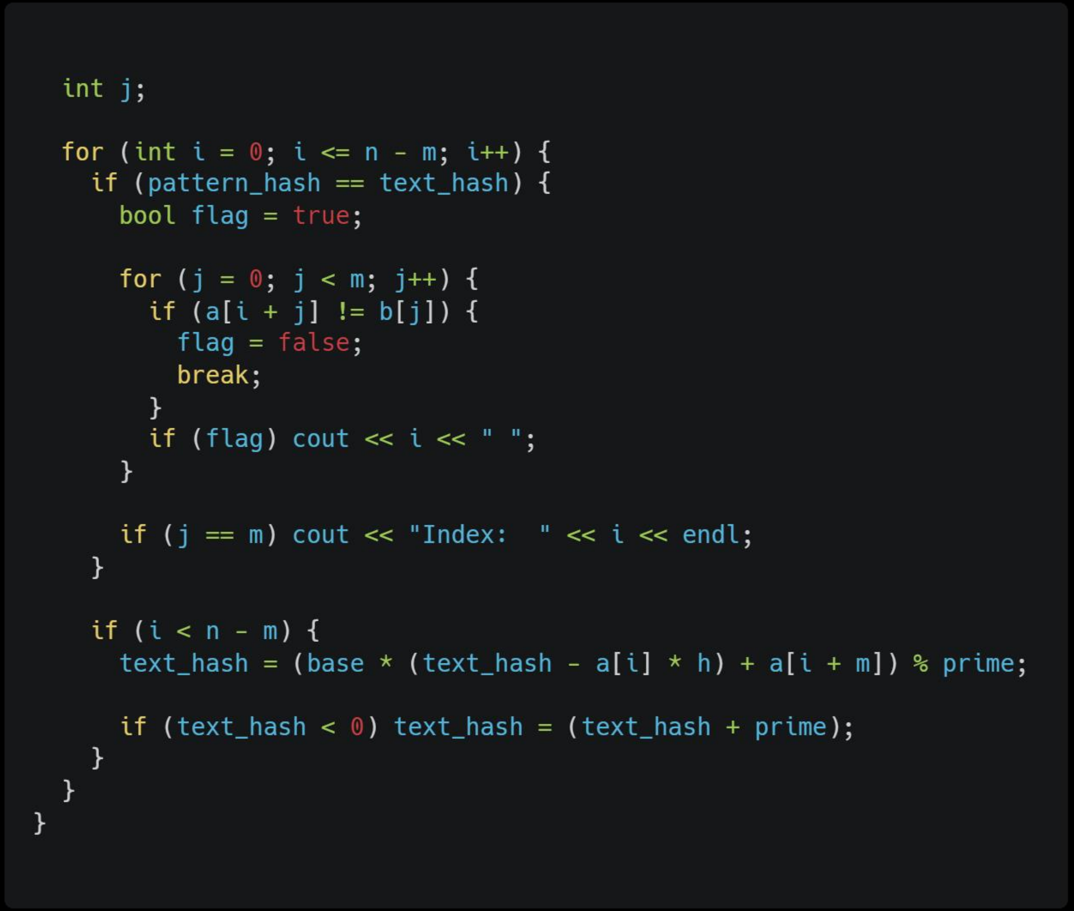
Rabin-Karp in O(n+m) (best) O(nm) (worst)

- Compute the hash of string b and string a up to len m.
- Go through string a char by char
    - Check if hash matches
        - Check match char by char
        - Print match
    - Recalculate hash
        - Remove first letter
        - Times base
        - Add next letter

Rabin-Karp in O(n+m) (best) O(nm) (worst)



```
void search(char a[], char b[]) {
  int m = strlen(b);
  int n = strlen(a);
  int pattern_hash = 0;
  int text_hash = 0;
  const int prime = 101;
  int h = 1;

  const int base = 256;

  for (int i = 0; i < m - 1; i++) h = (h * base) % prime;

  for (int i = 0; i < m; i++) {
    pattern_hash = (base * pattern_hash + b[i]) % prime;
    text_hash = (base * text_hash + a[i]) % prime;
  }
```



```
  int j;

  for (int i = 0; i <= n - m; i++) {
    if (pattern_hash == text_hash) {
      bool flag = true;

      for (j = 0; j < m; j++) {
        if (a[i + j] != b[j]) {
          flag = false;
          break;
        }
        if (flag) cout << i << " ";
      }

      if (j == m) cout << "Index:  " << i << endl;
    }

    if (i < n - m) {
      text_hash = (base * (text_hash - a[i] * h) + a[i + m]) % prime;

      if (text_hash < 0) text_hash = (text_hash + prime);
    }
  }
}
```

Questions?