CSCI203

Week 6 – Lecture B

Looking for Text (In all the right places)

- Consider the problem of *String Searching*:
  - Given a text, *t*, is the subtext, *s*, present in it?
- This problem occurs in many real-life applications:
  - grep;
  - find in a text editor;
  - Genome matching;
  - Google search.
- There are a wide number of techniques to achieve this.
- Let us look at a couple of examples.

Linear Search

The Naïve Approach: Linear Search

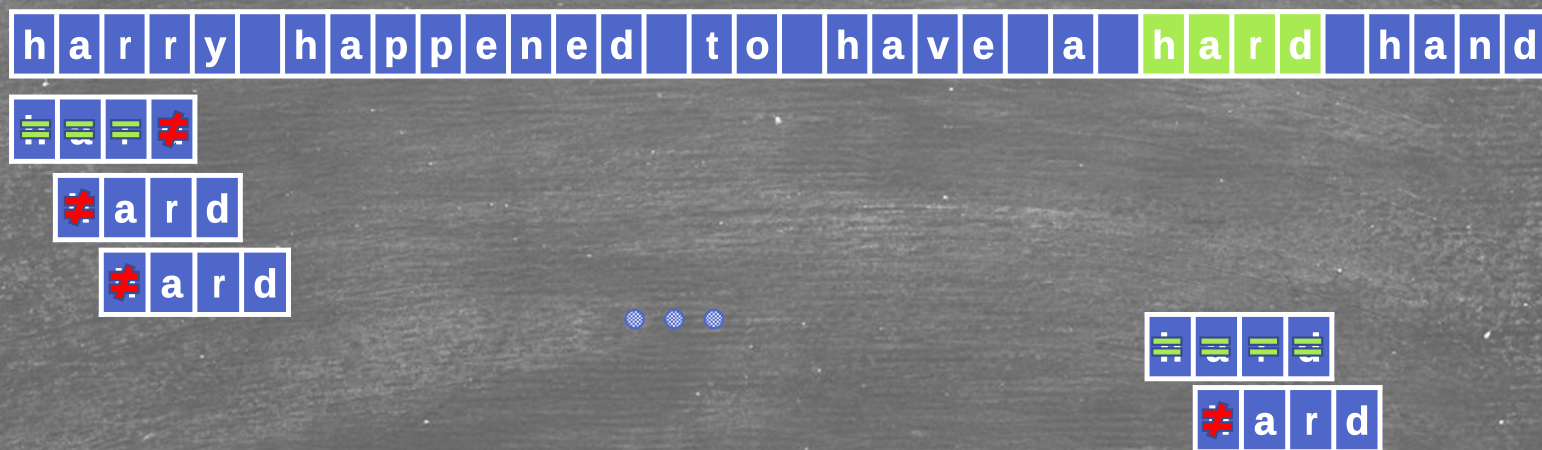
- The simplest possible approach is linear search:
  - Try to match s starting at each location in *t*.
  - for i in 0… length(t) – length(s)

```
for i in 0… length(t) – length(s)
    j=0
    while j < length(s) do
        if (s(j) != t(i+j)) break
        j++
    od
    if j == length(s) print(" string found starting at location " i)
rof
```

- We can see this with an example.

Linear Search: an Example

- Let *t* be the string “harry happened to have a hard hand”.
- Let s be the string “hard”.
- The search proceeds as follows:



Linear Search ≠ Linear Time Search

- The outer loop in our algorithm is repeated $|t|-|s|$ times.
  - Typically the string t is much longer than the string s, so this is $\Theta(t)$.
- The inner loop is repeated up to $|s|$ times for each time round the outer loop.
  - This is $\Theta(s)$
- The total number of comparisons is $\Theta(|s| \times |t|)$.
- Is this the best we can do?
- The best we can possibly do is $\Theta(|s|+|t|)$;
  - we have to at least look at each string!
- Can we actually achieve this goal of a linear time algorithm?

Linear Time Search

Linear Time Search

- To do this we will use hashing.
- We compare the hash of string *s* with the hash of each substring of *t* with the same length:
  - ```
    hash_s = hash(s)
    for i in 0…length(t)-length(s)
       hash_t = hash(t[i..i+length(s)-1])
       if hash_s == hash_t then
             brute-force compare s and the substring
             if they match print(" string found starting at
                                                  location ", i)
       fi
    rof
    ```
    ```

Linear Time Search

- This algorithm takes linear time, provided:
  - The hash function only collides rarely;
  - The hash function takes constant time to compute;
    - Independent of the length of string *s*!
- Surely, the second requirement is impossible.
- To hash a string of length $|s|$ must take $\Theta(|s|)$ operations.
  - Yes?
  - No!
- Not if we are clever.

Clever Hashing

- We note that we need $\Theta(|s|)$ time to compute h(*s*).
- We also need $\Theta(|s|)$ time to compute h(*t*[0..|*s*|-1), the initial substring of *t*.
- The trick is to compute the hash of each successive substring of *t* in constant time.
- If we look closely at these substrings, we see an interesting feature:
  - Successive substrings differ only by two characters.
- The first character of the first substring;

h	a	r	r

- The last character of the next substring.

a	r	r	y

Rolling Hash

- Maybe we can define a hash function which, given h(“harr”) can compute h(“arry”) in constant time.
- Let us define a *rolling hash* function, r(), so that:
  - h(“arry”)=r(h(“harr”),”h”,”y”)
  - We compute the hash of the next substring by removing the first and appending the new last characters;
    - In this case we remove “h” and append “y”.
- If we can compute a rolling hash in constant time than we can do string matching in linear time.
- How?

Karp-Rabin String Search

- The Karp-Rabin algorithm looks like this:
  - ```
    hash_s=hash(s)
    hash_t=hash(t[0..length(s)-1])
    for i in 0…length(t)-length(s)
       if hash_s == hash_t then
             brute-force compare s and the substring
             if they match print(“ string found starting at
                                                     location “ i)
       fi
       hash_t=roll(hash_t,t[i],t[i+length(s)])
    rof
    ```
  - The function roll(*h*,*p*,*s*) computes the rolling hash of the next substring given the hash of the existing substring, *h*, with the prefix *p* removed and the suffix, *s*, appended.
  - We need only find a suitable function roll().
    ```

How We Roll

- One popular way to compute roll() is to use something called the Rabin fingerprint.
- We start by treating each symbol in the alphabet as an integer – use the ASCII code for example.
- We then find a random prime number > the size of the alphabet—let's pick 257.
- We now compute h("harr") as:
  - $257^3.104+257^2.97+257^1.114+257^0.114$
  - = 1,771,793,837
  - Note: "h" = 104, "a" = 97 and "r" = 114.

The Next Hash

- Ok, so given that h(“harr”) = 1,771,793,837 how do we get h(“arry”)?
- It’s easy:
- Simply compute $r(h,p,s)= 257.(h–257^3.p)+s$
- $257.(1,771,793,837–257^3.104)+121$
- In this case the result is 1,654,094,526 which is exactly the same as h(“arry”)
- $257^3.97+257^2.114+257^1.114+257^0.121$
- Note: if these values become too large, we can reduce them modulo *m*, where *m* is a convenient value—say 2^{15} or 2^{31}.

Efficient?

- We can compute our hash values for *s* and the initial substring of *t* using *compact evaluation*.
- $p^{k-1}.c_1+p^{k-2}.c_2+ \dots + p.c_{k-1}+c_k$
- This requires a lot of multiplication!
- It can be re-written as…
- $h= c_k+p(c_{k-1}+p(c_{k-2}+\dots +p(c_3+ p(c_2+pc_1))..))$
- Where $k=|s|$ and c_i is the i^{th} character of *s*.
- This requires $|s|-1$ multiplications and $|s| -1$ additions.

Efficiency!

- If we precompute $q=p^{k-1}$ we can find the next hash value, h' as:
- $h'=p.(h-q.c_i)+c_j$ where we remove character *i* and add character *j*.
- This requires only 1 multiplication and 1 addition.
- Constant time.
- Thus we have $\Theta(|s|)$ operations to perform the initial hashes and $|t|-|s| * \Theta(1)$ operations to do the rehashing.
- Overall: our algorithm operates in $\Theta(|s|+|t|)$ time.
- We win!