Clever Linear Time Algorithms

Maximum Subset

String Searching

Maximum Subrange

Given an array of numbers values[1..n] where some are negative and some are positive, find the subarray values[start..end] that has the maximum sum.

Imagine the following approach:

```
start=0; end=-1; maxSum=0;
for possibleStart = 1 to n {
    localSum=0;
    for possibleEnd = possibleStart to n {
        localSum += values[possibleEnd]
        if (localSum > maxSum) {
            start = possibleStart;
            end = possibleEnd;
            maxSum = localSum;
        }
    }
}
```

Does the algorithm work? What is the runtime?

Maximum Subrange Run Time

The approach presented on the previous slides has two nested loops. The outer loop always goes from 1 to n. The inner loop starts based on the current value of the outer loop, but we have seen things like this before…

This algorithm would take $O(n^2)$ time.

We can do better!

The problem can be solved in O(n) time.

Maximum Subrange in Linear Time

To approach solving this problem in linear time, consider that as you pass through the information you can make some local decisions.

Consider minimum finding. You can call the first element in the ***smallest so far*** and store it. Then, as you traverse the rest of the list, if you see something smaller, then it is the smallest so far so you should store that instead. Transitivity means you never have to "look back" with this. At the end, the ***smallest so far*** is the **true** smallest.

Something to think about for this problem is the fact that if you have a ***running sum*** from a starting point, then if (for example) it ever goes negative then you've passed the best ending point for that starting point.

Searching within a string of text

If you had a **pattern** of length *m* and a **large block of text** of length *n*, what could the worst-case search time be?

Consider

```
P: AAAAAAB
T: AAAAAAAAAAAAB
```

Coding a straight-forward (brute force) solution, you might have something O($n \cdot m$) as your algorithm. If the pattern was large, this would be poor runtime in practice.

{yes, since once you shift the pattern past the end of the block of text you can stop so it is really something like O((n-m+1)·m) but we can remove lower-order terms to get a general sense}

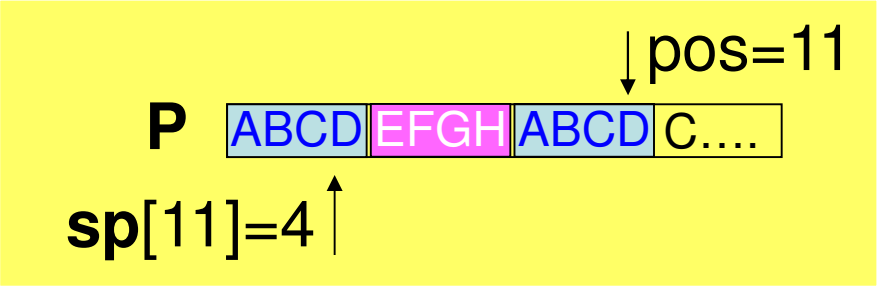
Searching within a string of text in pure linear time

If you had a **pattern** of length *m* and a **large block of text** of length *n*, could we perform the search in the worst-case scenario in linear time of just *n*?

What lessons could be learned from minimum finding and maximum sub-range in terms of think about what you've seen so far and avoiding "looking back" in the list?

Knuth-Morris-Pratt (KMP) Algorithm

Given a pattern (**P**) and a text block (**T**) you preprocess **P** to compute a zero-indexed array **sp**[] where **sp**[pos] contains the length of longest Proper ***prefix*** of **P** that matches a ***suffix*** of **P**[0..pos]



The idea here is essentially “If I hit a mismatch after a partial match, could the past several characters have actually represented the start of the pattern I’m looking for?”

During the execution, this **sp** array will be used to help us figure out how “far we can jump ahead” when we reach a mismatch.

Tracing KMP

We compare **P** with **T** until finding a mismatch.
We'll call that position *i+1* in **P** and *j+1* in **T**.

Mismatch at 24 gives j=23

T ABCD EFGH ABCD E

P ABCD EFGH ABCD C

Mismatch at 12 gives i=11

We then logically shift **P** using the **sp** value.
This allows the first **sp**[*i*] characters to match **T**[(*j*-sp[*i*]+1) .. *j*].
We then continue comparing from **P**[sp[*i*]] and **T**[*j+1*]

new j=23

T ABCD EFGH ABCD E

Shifted **P** ABCD EFGH ABCD C

new i=4

Make the **sp** array for… (answers on next slide)

`pattern: AAAAAAA`

`pattern: AAAAAAB`

`pattern: ABACABC`

Answers…

```
index:   0123456
pattern: AAAAAAA
sp:      0123456
```

```
index:   0123456
pattern: AAAAAAB
sp:      0123450
```

```
index:   0123456
pattern: ABACABC
sp:      0010120
```

Example to Trace

Pattern: **GACGGACA**

SP[ ] array: **00011230**

Let's keep track of comparisons as we trace…

G	A	C	G	G	A	G	A	C	G	G	A	C	A	G	A	C	G

More examples to try to pattern match…

```
index:   0123456
pattern: AAAAAAA
sp:      0123456

index:   0123456
pattern: AAAAAAB
sp:      0123450
```

AAAAABAAAAAABAAAAAAA

Another example to try…

```
index:   0123456
pattern: ABACABC
sp:      0010120
```

ABABBABAABABACABC

Pragmatic note…

The approach presented here (and many other places) has two pragmatic flaws.

(1) You have to special-case a failure on the first character in the pattern.

(2) You store a value for "what if I mismatch after I've already found the pattern" which is extraneous.

For this reason it seems that most implementations typically shift everything over one position in the **sp** array, and then puts a -1 into the 0 index.

I've also seen others that might be a mixture of different generations of KMP and/or might incorporate other mods.

Thought Question

Could you get something with better than linear time as its worst case? expected case?

If this topic really interests you, you might be interested in:

- Rabin-Karp Algorithm
- Boyer-Moore Algorithm(s)
- Applications of Finite State Machines to this…
- Bioinformatics…

Some Interesting Links

http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm

(algorithm)

http://whocouldthat.be/visualizing-string-matching/

(interactive trace, slightly different number use but “animation” is still the same)

http://www.ics.uci.edu/~goodrich/dsa/11strings/demos/pattern/

(interactive trace, not currently working)