Genome 540 Discussion

Conor Camplisson

January 5th, 2023

Genome Sciences
UNIVERSITY OF WASHINGTON

Outline

- Homework #1 Overview
- C++ Programming Tips

Outline

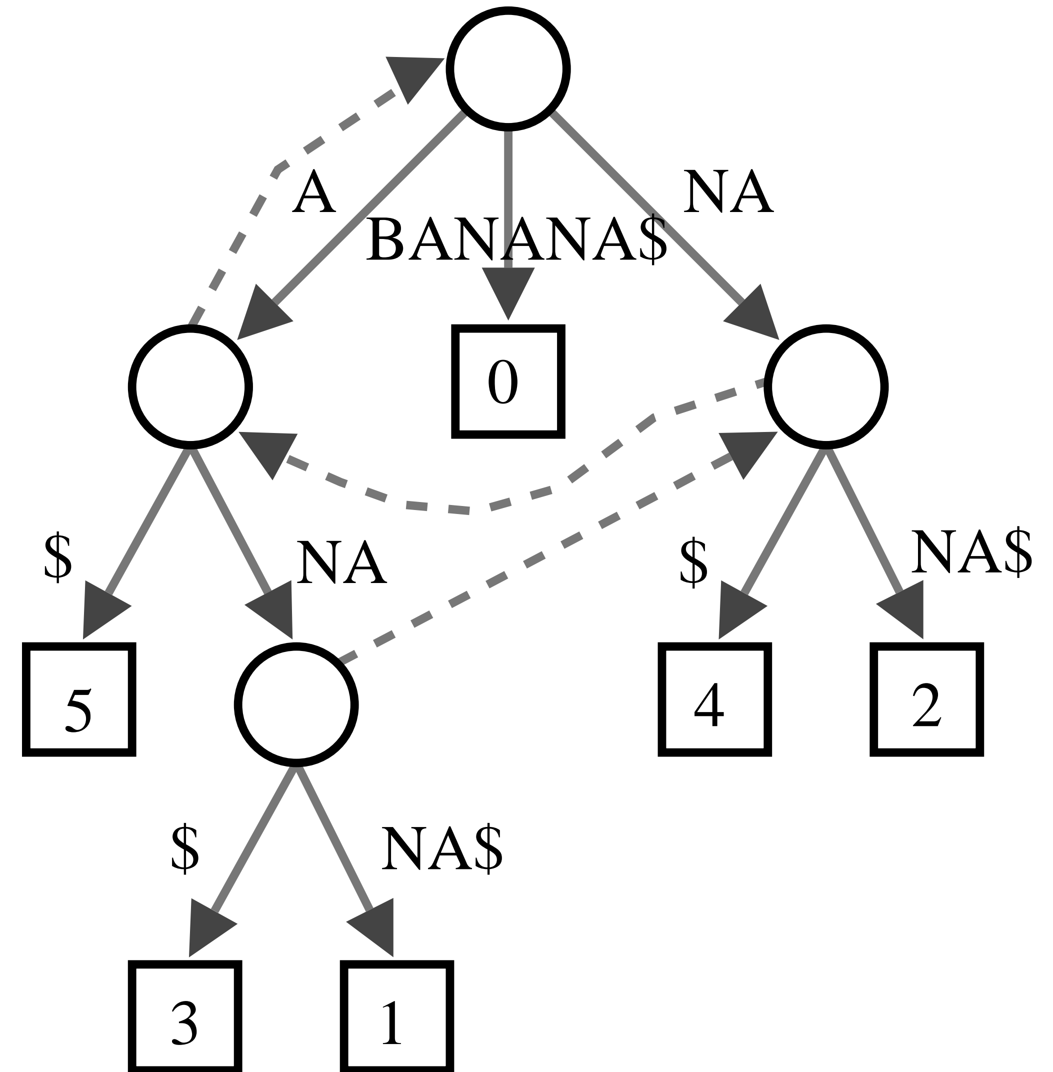
- Homework #1 Overview
- C++ Programming Tips

Homework #1

- Program a suffix array
- Test it on the files provided and compare results to the test output
- Run your program on the orthologous 10-megabase regions in the human and mouse genomes.
- Use the UCSC genome browser (hg38 and mm10) to figure out what biological feature they (the longest match) correspond to
- Submit your program output and your code

Suffix trees

- Applications:
  - Finding longest-repeated substring
  - Finding all repetitions in a string
  - Computing substring statistics
  - Approximate string matching
  - Compression
  - Genetic sequence analysis



- https://www.youtube.com/watch?v=hLsrPsFHPcQ&t=1s

Suffix arrays

- Substantially more space-efficient
- Search times comparable
- Greater upfront build time

- “A primary motivation for this paper was to be able to efficiently answer on-line string queries for very long genetic sequences (on the order of one million or more symbols long). In practice, it is the space overhead of the query data structure that limits the largest text that may be handled.”

- Fancier implementations than yours take advantage of the fact that you’re not sorting arbitrary strings, and that suffix trees can be built more quickly

SIAM J. COMPUT.
Vol. 22, No. 5, pp. 935–948, October 1993

© 1993 Society for Industrial and Applied Mathematics
003

SUFFIX ARRAYS: A NEW METHOD FOR ON-LINE STRING SEARCHES*

UDI MANBER†‡ AND GENE MYERS†§

Abstract. A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, "Is W a substring of A?" to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, an augmented algorithm is given that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ *expected* time, albeit with lesser space efficiency. It is believed that suffix arrays will prove to be better in practice than suffix trees for many applications.

SIAM J. COMPUT.
Vol. 22, No. 5, pp. 935–948, October 1993

© 1993 Society for Industrial and Applied Mathematics
003

SUFFIX ARRAYS: A NEW METHOD FOR ON-LINE STRING SEARCHES*

Yahoo!, Amazon, Google, YouTube, **NIH**

UDI MANBER†‡ AND GENE MYERS†§

Abstract. A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, "Is W a substring of A?" to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, an augmented algorithm is given that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ *expected* time, albeit with lesser space efficiency. It is believed that suffix arrays will prove to be better in practice than suffix trees for many applications.

SIAM J. COMPUT.
Vol. 22, No. 5, pp. 935–948, October 1993

© 1993 Society for Industrial and Applied Mathematics
003

SUFFIX ARRAYS: A NEW METHOD FOR ON-LINE STRING SEARCHES*

UDI MANBER[†‡] AND GENE MYERS[†§]

Yahoo!, Amazon, Google, YouTube, **NIH**

One of the creators of BLAST (1990),
Shotgun sequence assembly

Abstract. A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, "Is W a substring of A?" to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, an augmented algorithm is given that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ *expected* time, albeit with lesser space efficiency. It is believed that suffix arrays will prove to be better in practice than suffix trees for many applications.

Suffix array step 3: Sort the Pointers Lexicographically

ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC

p_{10}	AAACCGTACACTGGGTTCAAGAGATTTCCC
p_{11}	AACCGTACACTGGGTTCAAGAGATTTCCC
p_{28}	AAGAGATTTCCC
p_{17}	ACACTGGGTTCAAGAGATTTCCC
p_{12}	ACCGTACACTGGGTTCAAGAGATTTCCC
p_{1}	ACCTGCACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{7}	ACTAAACCGTACACTGGGTTCAAGAGATTTCCC
p_{19}	ACTGGGTTCAAGAGATTTCCC
p_{29}	AGAGATTTCCC
p_{31}	AGATTTCCC
p_{33}	ATTTCCC
p_{27}	CAAGAGATTTCCC
	⋮

Some HW1 issues

- Efficiency
  - < 1 min, < 200 MB RAM
  - C++ ops to avoid:
    - += (for long strings – e.g. reading files)
    - push_back
    - strlen on suffices
- FASTA file sequence lines may have:
  - White space, digits
  - Mixed case
  - N’s / other ambiguity codes
    - long N tracts are problem for suffix arrays!

HW1 tips

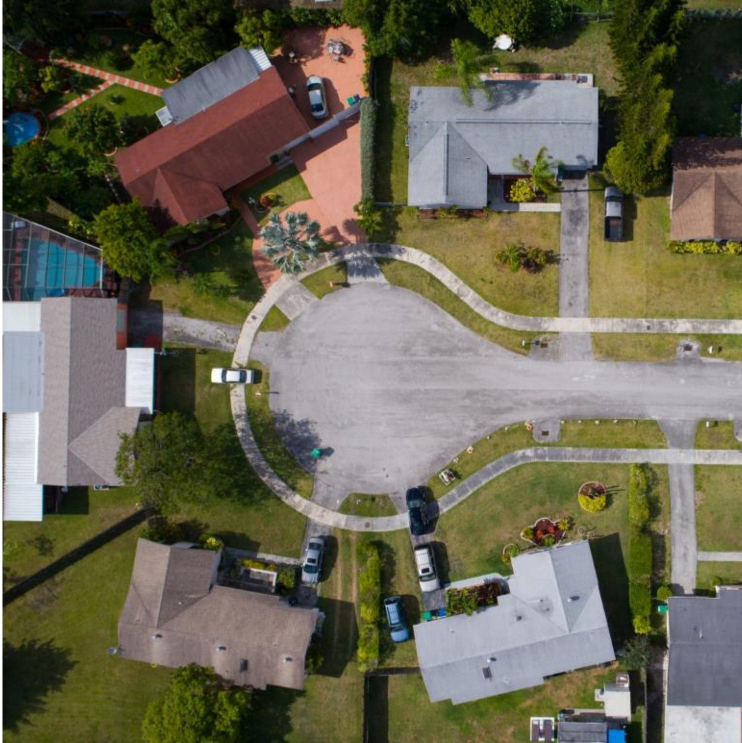
- Start with pseudocode/python
- Get comfortable with pointers
- Think about how to store inputs
- Think about how to store results
- Output to template directly

Outline

- Homework #1 Overview
- C++ Programming Tips

Pointers: conceptual introduction

5 houses



5 pointers to houses

1. 777 Brockton Avenue, Abington MA 2351
2. 30 Memorial Drive, Avon MA 2322
3. 250 Hartford Avenue, Bellingham MA 2019
4. 700 Oak Street, Brockton MA 2301
5. 66-4 Parkhurst Rd, Chelmsford MA 1824

Pointers in C++

- Pointers are memory addresses, which point to variables
- There are two operators essential for handling pointers and memory addresses in C/C++: **`*`** and **`&`**
- Unfortunately, both operators can be used in two ways which often leads to confusion.

* **as a suffix** to a type is a “pointer”

char * p; // p is a memory location that stores a “char”

& as a unary prefix is the address-of operator and obtains the memory address of a variable

```
char x = 'h';

char * p = &x;
```

*** as a unary prefix** means content of that memory location

char y = ‘a’;

char* p = &y; // p points to the memory location of y

char x = *p ; // *p is the object that p points to (i.e. ‘a’)

& as a suffix to a type means “pass by reference”

```
// pass by reference
void my_func(int &a){
        a = 5;
}

int main(int argc, char* argv[] ){
        int a = 0;
        my_func(a);
        cout << a << endl;
}
```

Arrays point to blocks of memory

- Arrays are just pointers to continuous blocks of memory

```
char word[6];  // an array of 6 characters called word
word[0] = 'a';
```

- Array indices are just pointer arithmetic and dereferencing combined
  - a[12] is the same as *(a + 12)
  - &a[3] is the same as a + 3

```
const char *word = "hi";
```

word[0]	word[1]	word[2]
'h'	'i'	'\0'
0x80	0x88	0x90
word	word+1	word+2

- Large arrays should be dynamically allocated (on the heap)
  - Make sure you delete them

```
int n = some_large_number;
double * d = new double[n];
```

Dynamic arrays in C++ (Vectors)

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector> // must have this in order to use vector
#include <algorithm>
using namespace std;

int main(int argc, char* argv[] ){
	vector<string> vec; // make an empty vector that will be made up of strings
	vec.push_back("ZABC"); // adds “ZABC” to the vector, can be accesses with strings[0]
	vec.push_back("DEF"); // adds ”DEF”
	for(auto x : vec){
		cout << x << endl;
	}
	vec.clear(); // empty the vec of all elements
	// documentation http://www.cplusplus.com/reference/vector/vector/
}
```

Custom data types in C++ (Structs)

```
struct complex{
        int real;
        int img;
};

int main(int argc, char* argv[] ){
        complex a = { 10 , 1} ;
        cout << “Real: ” << a.real << “    Img: ” << a.img << endl;

        complex * p = &a;
        cout << “Real: ” << p->real << “    Img: ” << p->img << endl;
}
```

Sorting in C++ with sort

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <algorithm> // must have this in order to use sort
using namespace std;

bool my_cmp(const string & a, const string & b ){
	return( a < b );
}
int main(int argc, char* argv[] ){
	vector<string> vec;
	vec.push_back("ZABC");
	vec.push_back("DEF");
	sort(vec.begin(), vec.end(), my_cmp );
	for(auto x : vec){
		cout << x << endl;
	}
}
```

Other sorting options in C++

function

qsort

<cstdlib>

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

Sort elements of array

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order.

The sorting algorithm used by this function compares pairs of elements by calling the specified *compar* function with pointers to them as argument.

The function does not return any value, but modifies the content of the array pointed to by *base* reordering its elements as defined by *compar*.

The order of equivalent elements is undefined.

Other sorting options in C++

function

qsort

<cstdlib>

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

Sort elements of array

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order.

The sorting algorithm used by this function compares pairs of elements by ca pointers to them as argument.

The function does not return any value, but modifies the content of the array elements as defined by *compar*.

The order of equivalent elements is undefined.

```
1  /* qsort example */
2  #include <stdio.h>      /* printf */
3  #include <stdlib.h>     /* qsort */
4
5  int values[] = { 40, 10, 100, 90, 20, 25 };
6
7  int compare (const void * a, const void * b)
8  {
9    return ( *(int*)a - *(int*)b );
10 }
11
12 int main ()
13 {
14   int n;
15   qsort (values, 6, sizeof(int), compare);
16   for (n=0; n<6; n++)
17      printf ("%d ",values[n]);
18   return 0;
19 }
```

http://www.cplusplus.com/reference/

Loading a file in C++

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <string>
#include <cassert>

using namespace std;

void read_file(string filename, string& contents, int& num_lines)
{
    ifstream f;
    f.open(filename.c_str());
    string line;

    contents = "";
    num_lines = 0;
    while(getline(f, line)) {
      contents.append(line.substr(0, line.length()));
      num_lines++;
    }

    f.close();
}

int main(int argc, const char* argv[])
{
    string fn = argv[1];
    string contents;
    int num_lines;

    read_file(fn, contents, num_lines);

    cout << "Read: " << fn << "\n";;
    cout << "  * " << num_lines << " lines\n";
    cout << "  * " << contents.length() << " characters (excluding newlines)\n";

    char * contents_cstring = (char*)contents.c_str();
    for (int i = 0; i < contents.length(); i++) {
      assert(contents_cstring[i] == *(contents_cstring + i));
      assert(contents_cstring[i] == contents.at(i));
    }
    assert(contents_cstring[contents.length()] == '\0');
}
```

Loading a file in C++

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <string>
#include <cassert>

using namespace std;

void read_file(string filename, string& contents, int& num_lines)
{
    ifstream f;
    f.open(filename.c_str());
    string line;

    contents = "";
    num_lines = 0;
    while(getline(f, line)) {
      contents.append(line.substr(0, line.length()));
      num_lines++;
    }

    f.close();
}

int main(int argc, const char* argv[])
{
    string fn = argv[1];
    string contents;
    int num_lines;

    read_file(fn, contents, num_lines);

    cout << "Read: " << fn << "\n";;
    cout << "  * " << num_lines << " lines\n";
    cout << "  * " << contents.length() << " characters (excluding newlines)\n";

    char * contents_cstring = (char*)contents.c_str();
    for (int i = 0; i < contents.length(); i++) {
      assert(contents_cstring[i] == *(contents_cstring + i));
      assert(contents_cstring[i] == contents.at(i));
    }
    assert(contents_cstring[contents.length()] == '\0');
}
```

```
Annes-MacBook-Pro-2:sandbox anne$ g++ example.cpp -o example
Annes-MacBook-Pro-2:sandbox anne$ ./example example.cpp
Read: example.cpp
  * 45 lines
  * 971 characters (excluding newlines)
```

If you are using Mac
(under macOS12.1, my experience):

potentially you need to install *Xcode*,
and use clang++ instead of g++