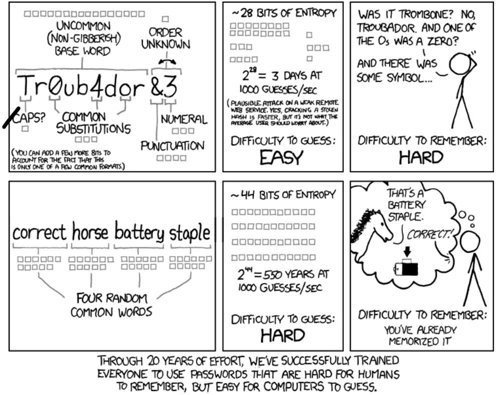
CS319: Scientific Computing (with C++)

Week 6: ~~The Password Problem~~

9am, 16 March, 2021

Title change to:

"Comparing Algorithms".



http://xkcd.com/936

Usual reminders...

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		* LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			~~LECTURE~~		

1. This week, we have just one recorded class: **Tuesdays** at 9.00.
2. **Lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.
3. A short introduction to the lab will be recorded.

Usual reminders...

1. Part 1: A note on complexity
2. Part 2: Merge Sort
   - Why is Merge Sort is fast
   - Implementation
3. Part 3: Comparing in practice
4. Part 4: The Password problem
   - Algorithm (high-level)
   - Implementation

Part 1: A note on complexity

CS319 – Week 6
Week 6: The Password Problem

Start of ...

PART 1: **A note on complexity**

Part 1: A note on complexity

Before we introduce an algorithm that is “better” than Bubble Sort, we need to explain what “better” means.

There are many ways that one algorithm could be considered superior to another, for example:

- takes less time to run; ✓
- takes less memory to run; ✓
- takes less time to program; ←
- is more accurate;
- is more reliable;
- ...? Do you have any idea?

Eg: more porrellizable.

Part 1: A note on complexity

O = "oh" = "Order".

Focusing on efficiency, we now need a way of discussing how the time taken by an algorithm depends on the problem size.

The usual way to discuss this in in terms of the "Big $\mathcal{O}$" notation, which is use to classify how their run-times (for example) grow as the input size grows.

For example, if we say an algorithm for a problem of size n has complexity $\mathcal{O}(n^2)$, then we mean there is some constant, C such that the run-time is at most Cn^2. We don't really care too much about what C is. For example, if Algorithm 1 had complexity $0.1n^2$, and Algorithm 2 had complexity $100n$, then...

Problem size, when sorting, we mean the number of items to be sorted.

Part 1: A note on complexity

Focusing on efficiency, we now need a way of discussing how the time taken by an algorithm depends on the problem size.

The usual way to discuss this in in terms of the "Big $\mathcal{O}$" notation, which is use to classify how their run-times (for example) grow as the input size grows.

For example, if we say an algorithm for a problem of size n has complexity $\mathcal{O}(n^2)$, then we mean there is some constant, C such that the run-time is at most Cn^2. We don't really care too much about what C is. For example, if Algorithm 1 had complexity $0.1n^2$, and Algorithm 2 had complexity $100n$, then...

n	$0.1\,n^2$	$100\,n$	
1	0.1s	100 s	
10	$(0.1)(100) = 10$	1,000	
100	$(0.1)(10{,}000)$ $= 1{,}000$	10,000	
1000	100,000	100,000	← break-even

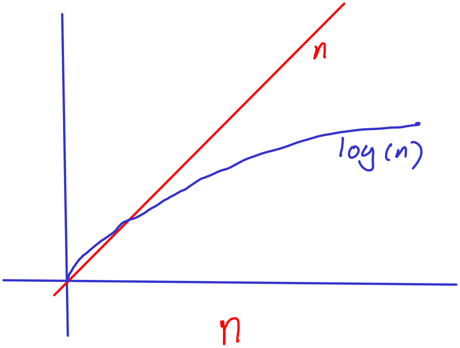
Generally, we work out the "n" part mathematically and C by computing.

Part 1: A note on complexity

The best to worst, some common complexities are

- $\mathcal{O}(1)$ → "Time is independent of Problem size".
- $\mathcal{O}(\log n)$ ⟶
- $\mathcal{O}(n)$
- $\mathcal{O}(n \log n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(2^n)$
- $\boxed{\mathcal{O}(n!)}$

Common, but terrible.



Part 1: A note on complexity

CS319 – Week 5
Week 6: The Password Problem

END OF PART 1

Part 2: Merge Sort

CS319 – Week 6
Week 6: The Password Problem

Start of ...

PART 2: ~~Marge~~ Sort

Merge

Part 2: Merge Sort

The **Bubble Sort algorithm** from last week is much too slow for the project we have in mind: its worse-case complexity is $\mathcal{O}(N^2)$ for a list of length N.

Instead we'll implement the **Merge Sort** algorithm. It has complexity $\mathcal{O}(N \log N)$.

Merge Sort

- Split the list into two smaller lists,
- Split each of those into 2 smaller lists.
- Keep doing this until each list is of length 1.
- A list of length 1 is already sorted, so...
- Reassemble each of your sub-lists by merging these sorted list.

Part 2: Merge Sort

It is useful to write this as a **recursive algorithm**:

Recursive Merge Sort Algorithm

```
procedure mergesort($L = a_1, a_2, \ldots, a_n$)
if $n \geq 1$ then          [if n=0 or 1, we are done!]
   $m :=$ floor($n/2$)      → (if n is even, m=n/2; if n is odd m=(n-1)/2)
   $L_1 := (a_1, a_2, \ldots, a_m)$
   $L_2 := (a_{m+1}, a_{m+\overset{2}{1}}, \ldots, a_n)$
   $L :=$ merge(mergesort($L_1$), mergesort($L_2$)).
end if
```

So we need two functions:

(i) A `Merge()` function to merge two sorted list

(ii) A `MergeSort()` function that

- splits the list in two,
- calls `MergeSort()` for each half
- calls the `Merge()` function

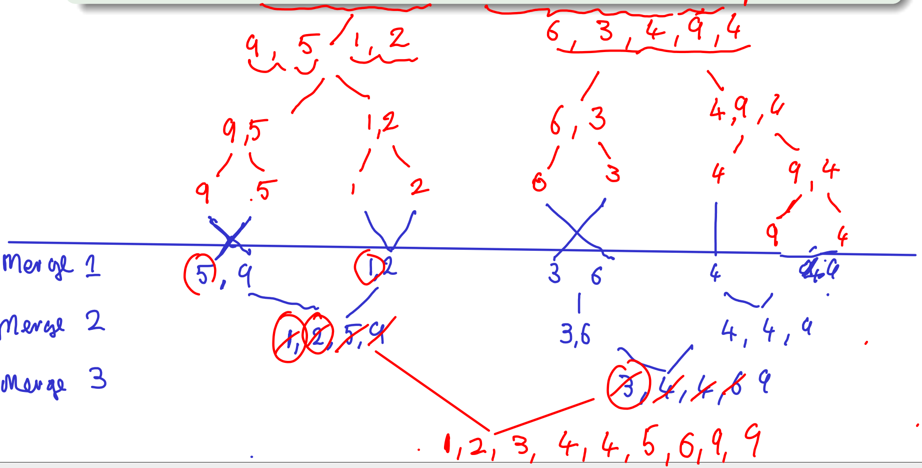
Part 2: Merge Sort

Example (Merge Sort)

Show how **Merge Sort** would sort the list

9 5 1 2 6 3 4 9 4

n = 9
m = 4



Part 2: Merge Sort

Why is Merge Sort is fast

First, why is Bubble Slow?

For this algorithm, we find the largest element in a list of length n. That takes $n-1$ steps.

Next apply bubble to a list of length $n-1$, which takes $n-2$ steps. Repeating, this needs

$$\left.\begin{array}{l} \quad n-1 \\ +\, n-2 \\ +\, n-3 \\ +\, \vdots \\ +\, 2 \\ +\, 1 \end{array}\right\}$$

$$(n-1)+(n-2)+(n-3)+\cdots+(2)+1$$

$$=(n-1)\left(\frac{n}{2}\right)$$

[Check this!]

$$=\frac{1}{2}n^2-\frac{n}{2}$$

which is $O(n^2)$.

Why is Merge Sort is fast

But Merge Sort: Splits the list into lists of length 1 in $\log_2(n)$ steps. (Eg if $n = 8$, 3 steps are needed).

Then merging requires $O(n)$ steps

Total is $n \log_2(n)$

ie $O(n \log(n))$.

!

Our first function will take two sorted lists, and combine them.

00MergeSort.cpp

This function sorts list 1 and list 2 into the "Merged" arrays

```
template <typename MyType>
void Merge(MyType *list1, unsigned int length1,
           MyType *list2, unsigned int length2,
           MyType *Merged)
{
  unsigned int i=0, j=0;
  for (unsigned int k=0; k<length1+length2; k++)
    if ( (i != length1) && ((j==length2)
              || (list1[i] <= list2[j])) ) {
      Merged[k] = list1[i];
      i++;
    }
    else {
      Merged[k] = list2[j];
      j++;
    }
}
```

Part 2: Merge Sort

00MergeSort.cpp

```
template <typename MyType>
void MergeSort(MyType *list, unsigned int length)
{
  if (length <=1) // A list of length 0 or 1 is sorted.
    return;
  else {
    unsigned int m = (unsigned int)floor((double)length/2.0)
    MyType *list1 = new MyType [m];
    MyType *list2 = new MyType [length-m];
    for (unsigned int i=0; i<m; i++)
      list1[i]=list[i];
    for (unsigned int i=0; i<length-m; i++)
      list2[i]=list[m+i];
    MergeSort(list1, m);
    MergeSort(list2, length-m);
    Merge(list1, m, list2, length-m, list);
    delete [] list1;    delete [] list2;
  }
}
```

n

$m = \text{floor}(n/2)$

making sublists

CS319 – Week 5
Week 6: The Password Problem

END OF PART 2

Part 3: Comparing in practice

CS319 – Week 6
Week 6: The Password Problem

Start of ...

PART 3: **Comparing in practice**

(of 3)

Part 3: Comparing in practice

Today, we considered two sorting algorithms

- **Bubble Sort** which is conceptually simple, and has a worst-case complexity of $\mathcal{O}(N^2)$ for a list of length N.
- A recursive **Merge Sort**, which has a worst-case complexity of $\mathcal{O}(N \log N)$ for a list of length N.

This means that if we have a list of length N, then the expected time taken for the methods are $C_B N^2$ and $C_M N \log N$, for some constants C_B and C_M.

C_B = "constant for Bubble"

C_M = "Constant for Merge"

Part 3: Comparing in practice

We want to estimate these constants so that we can predict how long the algorithm will take for some given N.

Before class, I ran both algorithms. Here is a snippet of the code I used, and the output. **Can we estimate how long each algorithm would take for a list of length 32 million?**

Part 3: Comparing in practice

See `01CompareSorts.cpp` for more details

```
for (int n=1000; n<=32000; n*=2) {
  for (int i=0; i<n; i++)
    Copy[i] = Numbers[i];
  start=clock();
  BubbleSort(Copy,n);
  diff = (double)(clock()-start);
  diff_seconds = diff/CLOCKS_PER_SEC;
  C_bubble = diff_seconds/( (double)(n*n));
  std::cout << "Bubble:␣n␣=␣␣" << std::setw(5) << n <<
    "␣took␣" << std::setw(8) << diff_seconds <<
    "␣seconds.␣C=" << C_bubble << std::endl;
}
```

Output (Bubble Sort):

```
Bubble: n =  1000 took  0.01468 seconds. C=1.468e-08
Bubble: n =  2000 took  0.0265  seconds. C=6.625e-09
Bubble: n =  4000 took  0.06219 seconds. C=3.887e-09
Bubble: n =  8000 took  0.2737  seconds. C=4.276e-09
Bubble: n = 16000 took  1.134   seconds. C=4.428e-09
Bubble: n = 32000 took  4.623   seconds. C=4.514e-09
```

Part 3: Comparing in practice

See `01CompareSorts.cpp` for more details

```
for (int n=1000; n<=32000; n*=2) {
  for (int i=0; i<n; i++)
    Copy[i] = Numbers[i];
  start=clock();
  MergeSort(Copy,n);
  diff = (double)(clock()-start);
  diff_seconds = diff/CLOCKS_PER_SEC;
  C_merge= diff_seconds/( (double)(n*log2(n)));
  std::cout << "Merge:␣n␣=␣␣" << std::setw(5) << n <<
    "␣took␣" << std::setw(5) << diff_seconds <<
            << "␣seconds.␣C=" << C_merge << std::endl;
}
```

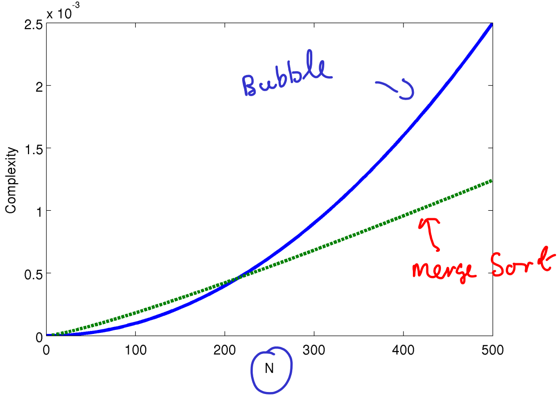
Output (Merge Sort):

```
Merge: n =  1000 took 0.000245 seconds. C=2.458e-08
Merge: n =  2000 took 0.000425 seconds. C=1.938e-08
Merge: n =  4000 took 0.000873 seconds. C=1.824e-08
Merge: n =  8000 took 0.001754 seconds. C=1.691e-08
Merge: n = 16000 took 0.003692 seconds. C=1.652e-08
Merge: n = 32000 took 0.008756 seconds. C=1.828e-08
```

Part 3: Comparing in practice

Question?

How long would it take to sort a list of length 32,000,000?



Part 3: Comparing in practice

CS319 – Week 5
Week 6: The Password Problem

END OF PART 3

Finished Here