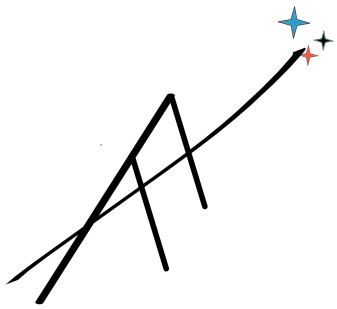
CPE593 Applied Data Structures and Algorithms



Ad Astra Education

Dov Kruger

January 30, 2023

Author

Dov Kruger 

Acknowledgements

I would like to thank the giants who taught me. Danielle Nyman who modelled teaching excellence and taught me many things herself, Simcha Kruger who taught me how to research and provided an amazing personal example, Henry Mullish my first coauthor, Roger Pinkham a towering intellect and storehouse of mathematical knowledge, Stephen Bloom who introduced me to automata and rescued my Masters thesis, Klaus Sutner who aside from teaching me computer science gave a memorable lesson on the pronunciation of LaTeX, Alan Blumberg the messiest yet best programmer I know who taught me computational fluid dynamics and demonstrated how to debug anything, Yu-Dong Yao for giving me my first position at Stevens, and who believed in me sufficiently to give me the data structures course, Michael Bruno for an outstanding example of teaching, as well as teaching the dynamics of waves that so terrified me as an undergraduate, and Min Song who supports my research and scholarship today. Thanks also to Moshe Kruger for creating the new Ad Astra Logo and for enriching my life in many ways. And thanks to Ellen who took care of my in this trying time when I had so many physical challenges and helped me get this done.

I would also like to thank some top students, TAs and graders who have contributed valuable corrections and insights over the years including Andres Contreras, Zachary Binger, Peter Ho, Yujie Liu, Pridhvi Myneni, Sarah Maven Oro, Fan Yang, Daniel Zheng, and Alice Huston.

Contributors

Jennifer Lloyd Pridhvi Myneni John Onwugbonu

Contents

1. Prerequisites

This course covers theory of computation in pseudocode but homeworks are in either C++ or Java, and we often delve into practical details of memory performance. Accordingly, you must know C++ or Java well enough to create a data structure involving pointers and dynamic memory in order to be able to handle the material in this class, and write and debug a complex algorithm. If you cannot do this, take C++ or Java first, before attempting this course.

2. Introduction

The full name of this course is Applied Data Structures and Algorithms, and for the last seven years, I have been curating the best algorithms from many different areas of computer science and giving students a broad overview of the field at the masters level. We do not cover a huge amount of graph theory, but as a result the course gets to touch on many of the fascinating topics in algorithms including number-theoretic, numerical methods, and depending on the semester, sometimes localization (navigation algorithms for robots) and other special topics.

The course has always been in flux, so the notes have always been one file per chapter, not particularly good looking. But as COVID has raged on, I have collected a lot of the material into a better-formatted LaTeX document and tried to create a single, coherent set of notes. I won't teach from these notes, we will still type everything in from scratch each period, because looking at already-constructed answers is not good for active learning. But now, when students want to check back, there is a document to look at. It already has significant advantages over the notes in separate google doc files. There are better diagrams, and code is uniformly shown in LaTeX style for greater readability. The course notes will continue to be a work in progress. If you have any comments or suggestions, feel free to email them to me. Volunteers to draw better diagrams and write pseudocode or code are always welcome.

3. Analysis of Algorithms

3.1 Assumptions

Data Structures and Algorithms is a theoretical field where we are interested in the asymptotic behavior of algorithms as the scale of the problem grows. Since computers get faster over time, and there are faster and slower machines, we are not interested in a constant factor, but rather are concerned with how the computation scales with the size of the problem.

We consider memory to be uniform, without considering faster access for sequential memory, no cache that makes retrieving recent memory faster. These features are good, they can speed up an algorithm but generally only by a constant factor.

All operations are considered to be the same speed. While addition is faster than multiplication on real CPUs, again it is only a constant factor and therefore irrelevant to complexity.

A very important part of this course is to develop skills in examining an algorithm or code in a programming language and determining its computational complexity. In every problem we look at, we will always ask "What is the complexity of this algorithm?" That is the first topic of the course, followed in turn by all the areas of classical problems.

3.2 Formal notation O(), $\Omega()$, $\Theta()$

Complexity analysis is based on determining a number of performance parameters on an algorithm. The foremost is the worst-case complexity, called big-O. An algorithm that is $O(n)$ scales linearly as the problem size n increases. The second criteria is best case using the greek letter omega; $\Omega(n)$ means that there is no faster way to complete the problem than to use n units of work. If both big-O and Ω are the same expression, this can be abbreviated to the greek letter theta (θ).

There is a formal definition of big-O and Ω.

$f(n) = O(g(n))$ means there exists some c for which $cO(g(n)) > f(n)$ for some n.

Example:

1. $f(n) = 600n + 20000000000, c = 601, O(n)$
2. $O(n^2 + 10000n + .000001n^3 + 10^6) = c = 1, O(n^3)$

A function that is $f(n) = O(n)$ and $\Omega(n)$ is $\Theta(n)$

Most people can intuitively understand big-O, the worst case. For example, in order to find a number in a list, the complexity is $O(n)$. This means that if the list gets twice as long, the worst case search is twice as long because worst case is searching the entire list. Determining if a number is not in the list is the worst case as well, because the only way to prove the number is not in the list is to search the entire list.

In this case, people also intuitively understand the best case. Even in a list of $n = 10^6$ elements, we may get lucky and happen to find the number we are looking for in the first element. So the best case is $\Omega(1)$.

However, a very common mistake is to misunderstand omega in other contexts. Suppose the problem is to sum the numbers in a list. For example:

$sum([1,3,5]) = 1 + 3 + 5 = 9$

This problem is $O(n)$, clearly if the list increases, so does the computational difficulty. There is no way to compute the answer without visiting every element so the lower bound is also $\Omega(n)$

However, many people will say, even after learning the above definitions, that in this case $\Omega(1)$. Their explanation is that summing a very short list (a single element) takes only a single unit of time, for example:

$sum([1]) = 1$

This mistake is extremely common, but of course anything is fast if the problem is small. That is not the purpose of the Ω notation at all. Ω means that the solution may sometimes be found quickly even if the problem is large.

3.3 Asymptotic Growth of Functions

In order to compare algorithms of different complexities, it is important to be able to compare common functions. The best kind of algorithm is O(1). This means it never takes longer no matter how the problem size grows. This is rather rare, most problems are not O(1). The function $\log(n)$ grows extremely slowly with n, so that is a common next step. Again, many people seem to confuse $log(n)$ and $\sqrt{n}$ so think carefully about these two functions. At $n = 1$, both are about the same: $log(n) = 0$ and $\sqrt{n} = 1$. By $n = 100$ $log_2(n) < 6$ while the square root is 10, so square root is pulling ahead. By n=1 million, $log(n)$ 20 and $\sqrt{n} = 1000$. Clearly log grows much, much smaller than square root. We examine this relationship in number-theoretic algorithms.

n	1	log_n	$\sqrt{n}$	n	n^2
1	1	0	1	1	1
10	1	3	3	10	10^2
10^2	1	7	10	10^2	10^4
10^3	1	10	33	10^3	10^6
10^6	1	20	1000	10^6	10^{12}
10^9	1	30	33000	10^9	10^{18}
10^{12}	1	40	10^6	10^{12}	10^{24}

3.4 Analyzing Code

A key skill in algorithms is to inspect code and determine the complexity. The fundamental building blocks are:

1. A loop that executes n times is $O(n)$
2. Two sequential loops add
3. Nested loops multiply (for each iteration of the outer loop, the inner executes completely)
4. Tail recursion n levels deep is $O(n)$
5. Recursion n levels deep where a function calls itself k times is k^n

Examples of code and complexity.

Code	Complexity	Comment
`for (int i = 0; i < n; i++) {` `}`	$O(n)$	counting n times
`for (int i = 1; i <= n; i++) {` `}`	$O(n)$	same
`for (int i = 5; i <= n; i += 3) {` `}`	$O(n)$	constants don't matter

Continued on next page

– continued from previous page

Code	Complexity	Comment
`for (int i = 5; i < 2*n; i += 7) {` `}`	$O(n)$	constant factors don't matter
`for (int i = 0; i < n; i += n/2) {` `}`	$O(1)$	count by $n/2$ means 2 times
`for (int i = 0; i < p; i += 2) {` `}`	$O(p)$	answer is not always $f(n)$ use the variable you are given
`for (int i = 1; i < n; i *= 2) {` `}`	$O(\log n)$	$\log_2(n) = O(\log n)$ all logs are similar
`for (int i = 1; i < n; i += i) {` `}`	$O(\log n)$	$\log_2(n) = O(\log n)$ all logs are similar
`for (int i = 1; i < n; i *= 3) {` `}`	$O(\log n)$	$\log_3(n) = c\log_2(n)$ all logs are similar
`for (int i = n; i > 0; i /= 3) {` `}`	$O(\log n)$	
`for (int i = 0; i < sqrt(n); i++) {` `}`	$O(\sqrt{n})$	

Continued on next page

– continued from previous page

Code	Complexity	Comment
`for (int i = sqrt(n); i <= n; i++) {` `}`	$O(n)$	$n - \sqrt{n} = O(n)$

`for (int i = 0; i < n; i++) {`
`}`
`for (int i = 0; i < n; i++) {`
`}`
	$O(n)$	sequential loops add. However $n + n = 2n$ and constants don't matter.

`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < n; j++) {`
`  }`
`}`
	$O(n^2)$	nested loops multiply

`for (int i = 0; i < a; i++) {`
`  for (int j = 0; j < b; j++) {`
`  }`
`}`
	$O(ab)$	nested loops multiply

`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < i; j++) {`
`  }`
`}`
	$O(n^2)$	$\frac{n(n+1)}{2} = O(n^2)$

`for (int i = 0; i < n; i++) {`
`  for (int j = i; j < n; j++) {`
`  }`
`}`
	$O(n^2)$	$\frac{n(n+1)}{2} = O(n^2)$

Continued on next page

– continued from previous page

Code	Complexity	Comment

`for (int i = 1; i <= n; i *= 2) {`
`  for (int j = 1; j < i; j++) {`
`  }`
`}`
	$O(n)$	$1 + 2 + 4 + 8 + ... \frac{n}{2} + n = 2n = O(n)$

`for (int i = 1; i < n; i++) {`
`  for (int j = 1; j < n; j *= 2) {`
`  }`
`}`
	$O(n \log n)$	

3.5 Test Yourself

See if you can figure out the complexity of each code snippet below:

```
for (int i = n; i < 2*n; i++){}
```

```
for (int i = n; i < 2*n; i+=n/2){}
```

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

3.6 Recursion

A recursive function is one that calls itself. All recursive functions must have a termination condition or they will never end (and be equivalent to an infinite loop, albeit one that will crash because on an actual computer calling a function requires memory on the stack.

The following example shows the classic recursive function factorial:

```
int fact(int n)
  if (n <= 1)
```

```
    return 1;
  return n * fact(n-1);
}
```

Because this function calls itself once, it is $O(n)$. For example, fact(3) calls fact(2) which calls fact(1) and then returns to compute $1 * 2 * 3 = 6$. There is no early way out of this algorithm, so it is also $\Omega(n)$. In this case, the if statement represents the end of the recursion, not an early exit condition. Compare this to the following:

```
int f(int n)
  if (n <= 1)
    return 1;
  if n % 13 == 0
    return 92;
  return n * f(n-1);
}
```

The above function looks a lot like factorial, but it has a weird special case. If the number is a multiple of 13 then it will terminate early. Thus $f(10) = fact(10)$ but $f(13) = 92$. Since 26, 39, are all multiples of 13, there is no number that will go more than 13, and the function f is therefore $O(1)$ and $\Omega(1)$ (since 13 is just a constant independent of n).

Functions that call themselves more than once are exponentially explosive. The following function fibo calls $fibo(n-1)$ and $fibo(n-2)$. Therefore, $fibo(n)$ is twice as expensive as $fibo(n-1)$. Calling fibo on a number 10 times bigger is not just $10n$, it is $2^{10}n = 1024n$. With a modern computer capable of on the order of billions of operations per second, we can achieve $n = 30$ which is 2^{30} (approximately 1 billion). Attempting $n = 40$ will be 1000 times harder, and $n = 50$ will be 1 million times more work! Even if we double the speed of our computer, that will only allow increasing n by 1.

Algorithm Fibonacci (recursive) Complexity $O(2^n)$

```
int fibo2(int n) {
   if ( n <= 2)
      return 1;
   return fibo2(n-1) + fibo2(n-2);
}
```

Similarly, for a function that calls itself 3 times, the complexity is $O(3^n)$.

Algorithm call3 Complexity $O(3^n)$

```
int call3(int n) {
   if (n <= 100)
      return 3;
   return call3(n-2) + call3(n-4) + call3(n-6);
}
```

The above algorithm shows the key feature, that it calls itself 3 times. Do not be fooled by the fact that any n less than 100 ends instantly. That is just a constant. Calling call3(101) will have to add $call3(99) + call3(97) + call3(95)$

call3(200) will call approximately $3^{(200-100)/6}$ times. Yes, because of the fact that we call $n-2, n-4, n-6$ effectively the exponent is divided by 6, but it is exponential, and it will grow very, very fast.

3.7 Dynamic Programming

Dynamic programming is a technique that will reduce the cost of an exponential recursive function from $O(2^n)$ or $O(3^n)$ to $O(n)$ for a one-dimensional case and order $O(mn)$ in a two-dimensional case. The problem with fibonacci above is that it is recomputing every answer multiple times. For example, $fibo(10) = fibo(9) + fibo(8)$ but $fibo(9) = fibo(8) + fibo(7)$. Even with only those two, $fibo(8)$ is computed twice. Keep trying that, and you can see how many times a smaller number would be computed, like $fibo(3)$.

The trick with dynamic programming is to store answers so they are only computed once. If we are calculating fibonacci numbers, and compute $fibo(n-1)$ there is no reason to ever compute it again. We need an array to remember the values, and this technique is also called memoization.

First, clearly the following non-recursive form can be computed in $O(n)$ time:

Algorithm Fibonacci (iterative) Complexity $O(n)$

```
int fibo2(int n) {
  int a = 1, b = 1, c;
  for (int i = 0; i < n; i++) {
    c = a + b;
    a = b;
    b = c;
  }
  return c;
}
```

Next, to duplicate this in a recursive routine, define a static variable which does not lose its values between invocations (in both C and Java). The variable is initialized with zeros, and whenever a value is computed it is stored in the memo array. Whenever checking for a particular value, we first test whether we have already computed it, thus only computing each value once. Since there are only n values, this means the computation can only be $O(n)$ although there is a significantly higher constant than simply computing in the above loop.

Algorithm Memoized Fibonacci (recursive) Complexity $O(n)$

```
int fibo3(int n) {
  static int memo[30] = {0}; // all memoized values start at zero
  // static variables are set only once,
  //not every time the function is entered

  if (n <= 2)
    return 1; // base case
  if (memo[n] != 0)
    return memo[n]; // if seen before, exit with the answer
  return memo[n] = fibo3(n-1) + fibo3(n-2); // store the answer
     and return
}
```

3.8 Exercises

Determine the complexity of each function (big O and Ω). These now are no longer exactly the same as the ones you were shown.

$O()\Omega()$

```
int a(int x) {
  int sum = 0;
  for (int i = 0; i < x; i++) {
    sum = sum + i;
    if (i == 23)
      return sum;
  }
  return sum;
}
```

O( ) Ω( )

```
int b(int x) {
  int sum = 0;
  for (int i = 1; i < x; i++) {
    for (int j = 1; j < x; j++) {
      if (i != j)
        sum = sum + i;
    }
  }
  return sum;
}
```

O( ) Ω( )

```
int c(int y) {
  int sum = 0;
  for (int i = 1; i < y; i++) {
    for (int j = 1; j < y; j++) {
      if (i == j)
        return sum;
      sum = sum + i;
    }
  }
  return sum;
}
```

O( ) Ω( )

```
int d(int a) {
   if (a <= 3)
     return 2;
   if (a % 20 == 0)
     return 5;
   return d(a-2) + d(a-4) + d(a-6);
}
```

```
input: int N, int D
output: int
begin
  res ← 0
  while N ≥ D
    N ← N − D
    res ← res + 1
  end
  return res
end
```

Algorithm 3.1: Integer division.

```
gcd(a, b)
  if b = 0
   return a
 endif
 return gcd(b, a−b)
end
```

Algorithm 3.2: Greatest Common Denominator.

```
gcd(a, b)
  if b = 0
   return a
```

```
  endif
  return gcd(b, a mod b)
end
```

Algorithm 3.3: Greatest Common Denominator.

Algorithm Fibonacci (iterative) Complexity $O()$

```
int fibo(int n) {
  int a = 1, b = 1, c;
  for (int i = 0; i < n; i++) {
   c = a + b;
   a = b;
   b = c;
  }
  return c;
}
```

Algorithm Fibonacci (recursive) Complexity $O()$

```
int fibo2(int n) {
   if ( n <= 2)
      return 1;
   return fibo2(n-1) + fibo2(n-2);
}
```

Algorithm Fibonacci (recursive) with dynamic programming Complexity $O()$

```
int fibo2(int n) {
   if ( n <= 2)
      return 1;
   static int memo[200] = {0};
   if (memo[n] != 0)
     return memo[n];
   return memo[n] = fibo2(n-1) + fibo2(n-2);
}
```

4. Dynamic Arrays

4.1 Introduction

4.2 Bad Dynamic Arrays

A list that is supposed to grow must allocate new memory whenever it runs out of space. The obvious implementation is an object that contains a pointer to memory and a size. Each time we add to the list, the size must grow. Each add operation is $O(n)$ and doing this n times yields a total of $O(n^2)$. For a list of 1 million elements, that would be 10^{12} operations, and on a computer capable of on the order of 1 billion operations per second, that is 1000 seconds, or about 20 minutes minimum. In practice it is much more because allocating all that memory requires a huge amount of time. In any case, even 10 million elements becomes completely intractable.

Here is a class definition for BadGrowArray

```
public class BadGrowArray {
  private int[] data;


  public BadGrowArray() {
    data = new int[0]; // special case needed for Java because
       length is part of the array
  }
  public void addEnd(int v) { // O(n)
    int[] old = data;
    data = new int[old.length+1]; // grow by 1
    System.arrayCopy(old, 0, data, 0, old.length);
    data[old.length] = v;
  }
}
```

In the next section, we show that adding a size and a capacity is the crucial feature that allows the list to grow for a time until it reaches full, and most important, it must double in size every time. In this way, the grow only happens $\log n$ times and the result is $O(2n)$

4.3 Operations

When building lists we must support a variety of operations including:

- Add an element to the end
- Add an element to the start
- Insert an element at position i
- Remove an element from the end
- Remove an element from the start
- Remote an element at position i
- Get the current size of the list
- Get ith element
- Set ith element

There are many higher level operations that can be built on top of these functions. Operations such as find, replace, are all built on the above primitives and are not covered in this course.

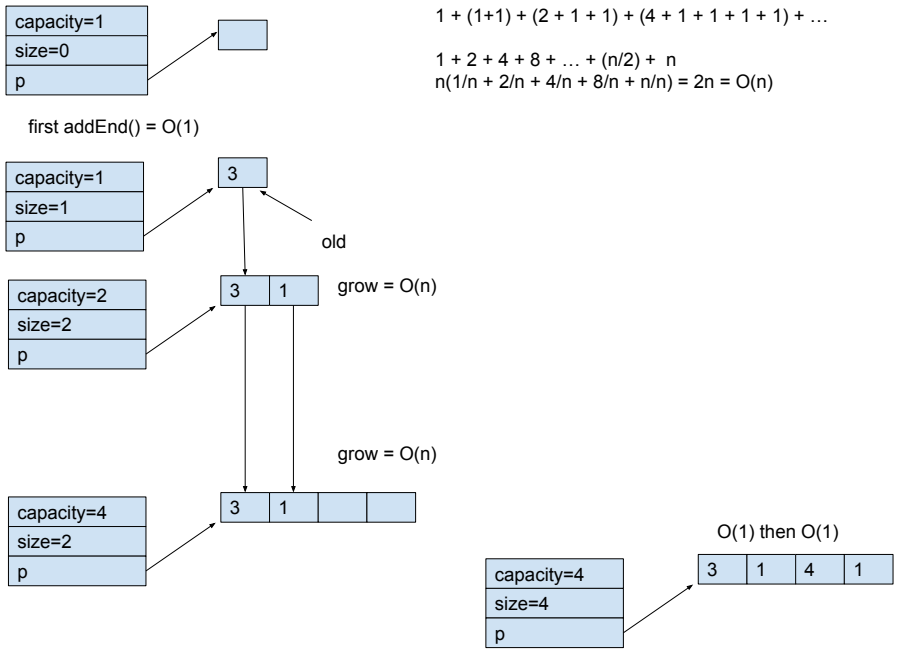
[TODO: insert description of bad dynamic arrays]

4.4 Dynamic Arrays

As we have seen, a dynamic array that only has a current size is $O(n)$ for all add operations, which means that adding n elements is a prohibitive $O(n^2)$. In order to do better, the dynamic array must not grow every time. By defining a dynamic array that has a capacity (how large the array is) and a size (how many elements are currently used) it is possible to reduce the number of times that the grow method is called. If grow is by 2, then the grow function will be called every other time, and the complexity is still $O(n)$. If it is called every 5th time, the same is true asymptotically. The standard solution used by vector in C++ and ArrayList in Java is to double in size each time. Consider growing a list to 1 million elements; the list starts with 1 element, doubles to 2 (calling grow) doubles again to 4, then 8, and so on up to n. There are $logn$ times that the list will grow.

The complexity is: $1 + 2 + 4 + ... + (n/2) + n = n(\frac{1}{n} + \frac{2}{n} + \frac{4}{n} + ... + 1) = n(2) = O(n)$

In other words, using the doubling strategy, growing a list only takes twice as long as pre-allocating the right size, it is still $O(n)$. Having said this, if the list size is known in advance, pre-allocating the right size is an easy way to double the speed of that part of the code.




```
class GrowArray {
private:
  uint32_t capacity;
  uint32_t size;
  int* data;
  void grow(); // should double the size of the array for
     efficiency
public:
  GrowArray(uint32_t initialCapacity);
  void addEnd(int v);
  void addStart(int v);
  void insert(int pos, int v);
  int removeEnd();
  int removeStart();
  int remove(int pos);
  uint32_t getSize();
  uint32_t getCapacity();
}
```

4.5 Iterators

An iterator is an object that hides the details of how to traverse through an object. In the case of a dynamic array, the iterator starts at the first element and advances each time to the end.

As will be seen later in the chapter on linked lists, it is very inefficient $O(n^2)$ to traverse a

linked list using an integer position. While it is perfectly fine to do so with a dynamic array, we must generate an equivalent iterator so that the two can be interchangable.

There are four operations needed for traversal. The iterator must be able to:

- Initialize to the start of the list
- Check when the end is reached
- Advance to the next item
- Access the current element

In C++, the dynamic array is called vector. The following shows how to traverse a vector for read-only use, and for modification.

```
vector<int> mylist = {5, 4, 3};
for (vector<int>::const_iterator i = mylist.begin(); i != mylist.
   end(); ++i)
  cout << *i << ' ';
for (vector<int>::iterator i = mylist.begin(); i != mylist.end();
    ++i)
  *i ++; // add 1 to every element in the list
```

Note that in the case of a dynamic array, both the above traversals are equivalent to using an integer index:

```
for (int i = 0; i < mylist.size(); i++)
  cout << mylist[i];
```

However, using the integer approach will become really inefficient if the implementation switches to linked list, so using iterators is always recommended. In fact, the new C++11 syntax is simpler and preferable from a maintenance point of view:

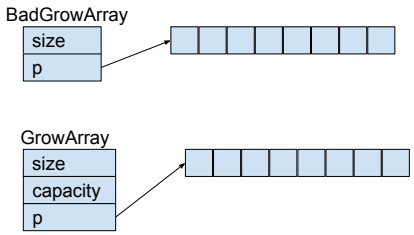
```
for (auto e : mylist)
  cout << e << ' ';
for (auto& e : mylist)
  e++;
```

In Java, the convention is slightly different. There are only three methods because next() both advances and returns the value of the current element, and must be executed before the first one. The equivalent is shown in Java for ArrayList:

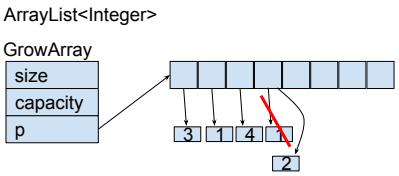
```
ArrayList<Integer> a = new ArrayList<>();
for (Iterator<Integer> i = a.iterator(); i.hasNext(); ) {
  System.out.print(i.next() + " ");
}
```

Note that in Java, the object Integer is immutable, so the only way of ”changing” an element of the list in this case involves replacing the Integer with a new one. This does work, but it is quite slow as object creation is quite expensive. Writing a list class in Java for builtin types like int can be from 8 to 10 times faster as a result.

The following diagram shows the memory layout of a dynamic array similar to C++ vector<**int**>:



The following diagram shows the memory layout of ArrayList<Integer> for comparison:



5. LinkedLists

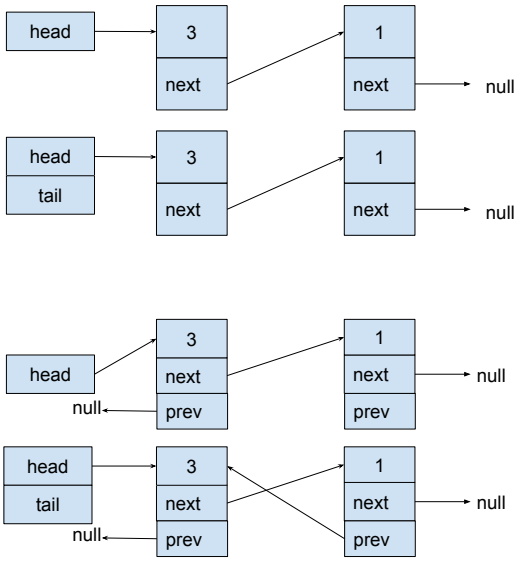
5.1 Introduction

Linked Lists are data structures that start with a pointer and contain a sequence of objects (nodes) each containing some data and pointers to the next and possibly the previous node. Because each element is not contiguous in memory, LinkedLists are efficient for data that is inserted in any order. It is no more expensive to insert at the beginning or the middle of a list than at the end. It is, however, inefficient to reach a particular element or location because it requires starting from the beginning (or for some variants, from the end) and scanning forward or backward.

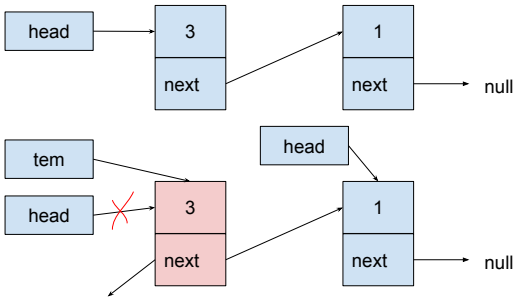
LinkedLists then are not better or worse than dynamic arrays. Each is better at one specific purpose. Dynamic arrays are more efficient for building a list sequentially adding at the end and for random access of any item in the list. LinkedLists are better for insertion from the beginning or in the middle. LinkedLists provide excellen practice for data structures with pointers, so this chapter prepares you to work with trees which are similar but more complex.

5.2 The Four Varieties of LinkedList

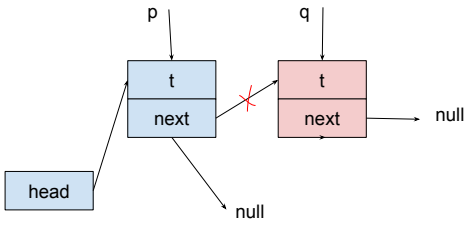
There are four kinds of LinkedList. The object can contain a pointer to head, or a pointer to head and tail. And each node can contain a pointer to the next node (a singly linked list), or a pointer to next and previous (a doubly linked List). The following diagram shows the four different kinds of LinkedList.



For the type of LinkedList with a pointer to head, it is easy to access the first element, but getting access to the end requires traversing the entire list which is $O(n)$. The following diagram shows deleting the head and tail of this kind of list, and clearly deleting the head is much faster.

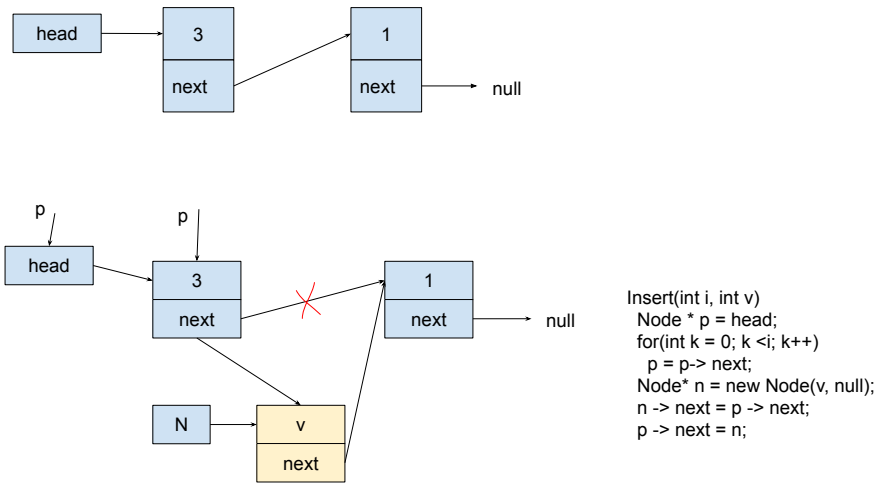



```
Delete_Head()
  Node * tem = head;
  head = head -> next;
  delete tem;
end
```




```
Delete_Tail()
  Node * p = head;
  Node * q = head -> next;
  for (; q->next !=null; p = q, q = q -> next)
    ;
  delete q;
  p -> next = null
end
```

Inserting in the middle (random access) via a position number is always painful in a linked list. The only way of doing so is to iterate through all the elements up to that position, and if it is beyond the end, the code will crash. It is, of course, possible to test for traversing beyond the end of the list but the list must throw some kind of condition.



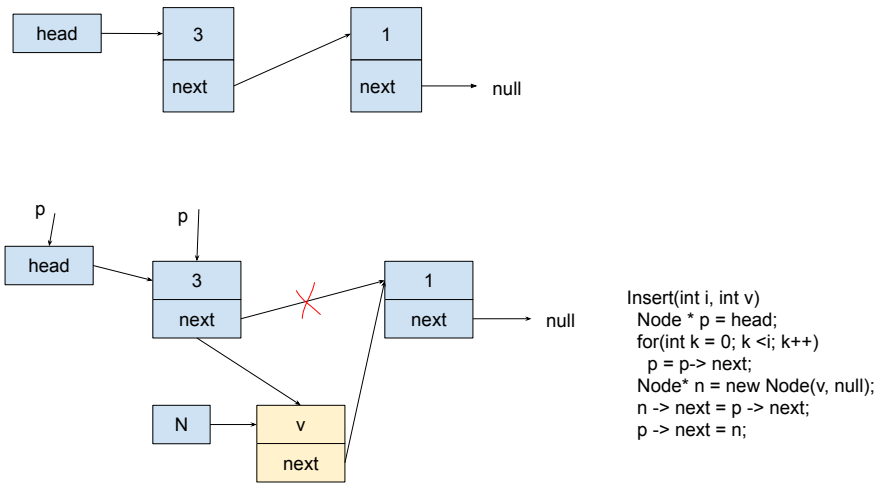
A second variety of linked list has a head and tail. This results in fast access to the beginning and end of the list, though a special if statement is needed for the special case when the list is empty.

```
if head = null
  head ← new Node(v, null)
  tail ← head
else
  head ← new Node(v, head)
end
```

Algorithm 5.1: LinkedList2.addStart(v)

```
if tail = null
  tail ← new Node(v, null)
  head ← tail
else
  tail.next ← new Node(v, null)
  tail ← tail.next
end
```

Algorithm 5.2: LinkedList2.addEnd(v)



Traversing to the ith element of a linked list is still slow $O(n)$, the extra pointer to the tail does not help.

```
Node p ← head                                                    1
for j ← 1 to i                                                   2
  p = p.next                    // if this crashes, then you shouldn' 3
    t have asked
end                                                              4
Node temp ← new Node(v, p.next)                                  5
p.next ← temp                                                    6
```

Algorithm 5.3: LinkedList2.insert(i, v)

5.2.1 Double Linked lists

For double linked lists, each node has a pointer to the next node, but also a back pointer to the previous node. This allows efficient reverse traversal of the list, as well as the ability to efficiently insert a value before a node. There is no difference in performance for adding or removing values from the beginning, end or middle of the list. There is a somewhat higher constant because the extra pointers must be assigned values.

5.3 Iterators

Because asking for a position by number is so inefficient in a linked list, it is vital to define a way to store the position within the list for fast retrieval. Yet from an

object-oriented design point of view, the programmer should not be allowed access to the list internals. With an iterator object, the implementation can be hidden from the programmer using the list, yet the result is as efficient as it can possibly be.

If we consider the list to be the single most-used design pattern, then the concept of traversing a list is second. The position should not be part of the list itself. If iteration is built into a list, then it is only possible to traverse a list one at a time. In actual use, multiple algorithms may need to track a position within the same list simultaneously. An iterator is therefore a separate object that encapsulates moving through a data structures.

Iteration consists of four operations: assignment of a start point (such as the start of the list), determining whether the end has been reached, traversing to the next element, and accessing the current elements.

The following code shows traversal of a list in C++. Note that the implementation can be switched from vector to linked list just by substituting the name of the class.

```
int main() {
  vector<int> a;
  for (int i = 0; i < 10; i++)
    a.push_back(i); // add the elements to the list

  for (vector<int>::const_iterator i = a.begin(); i != a.end();
     ++i)
    cout << *i;
  }

  list<int> b; // linked list implementation
  for (int i = 0; i < 10; i++)
    b.push_back(i); // add the elements to the list

  for (list<int>::const_iterator i = a.begin(); i != a.end(); ++i
     )
    cout << *i << ' ';
  }

  // an iterator to change the list, doubling each element is
     shown below:

  for (list<int>::iterator i = a.begin(); i != a.end(); ++i)
    *i *= 2;
  }
```

The only annoyance is that the code that refers to the iterator must mention the type of the object. C++-11 fixes that with the auto type inference mechanism. If the parameter in the following function were changed, the code would still work.

```
void display(const list<int>& a) {
  for (auto e : a) {
```

```
    cout << e << ' ';
  }
}
```

In Java, the convention is to combine the four separate operations into three. The operations to get the current value and move to the next one are combined.

```
public static void display(LinkedList list) {
  ListIterator<String> i = list.listIterator();
  while (i.hasNext()) {
    System.out.println(i.next());
  }
}
```

Note that this means if the loop requires getting the value twice, it must be stored in a variable, since each time the next() method is called, the iterator advances.

```
  while (i.hasNext()) {
    Object obj = i.hasNext(); // use the variable obj
    ...
  }
```

6. Stacks and Queues

6.1 Stacks

A stack is an abstract data type that supports three main operations:

- push (add an element to the top of the stack)
- pop (remove an elements from the top of the stack)
- isEmpty (tell whether the stack has any elements)

In many implementations, popping an element off the stack also returns the value to the caller. If not, then an extra element is needed. In some implementations, we can ask for the size of the stack instead of using isEmpty, but if the stack is implemented using a linked list this is not efficient.

Consider whether the stack can be implemented using some of the data structures already covered. Can it be implemented using a dynamic array? Can it be implemented using a linked list? The answer is yes, of course, but how efficient is the resulting implementation?

Consider the following implemenation using a Dynamic array with a capacity and size. Such an array can be efficiently grown.

```
class Stack {
private:
  DynArray impl;
public:
  Stack() : impl() {}
  void push(int v) { impl.addFirst(v); }
  int pop() {
    return impl.removeFirst();
  }
  bool isEmpty() const {
    return impl.size() == 0;
  }
};
```

In the above case, inserting and removing elements from the beginning of the dynamic array is $O(n)$, so the implementation is $O(n)$ which is not good. The isEmpty() method uses the underlying size which is $O(1)$.

Instead, consider an implementation which adds and removes from the end of the list:

```
class Stack {
private:
  DynArray impl;
public:
  Stack() : impl() {}
  void push(int v) { impl.addEnd(v); }
  int pop() {
    return impl.removeEnd();
  }
  bool isEmpty() const {
    return impl.size() == 0;
  }
};
```

This time, all methods are $O(1)$. It is for this reason that in C++, the methods to add and remove elements from the list are called push_back and pop_back.

Next, consider whether the stack can be implemented using a linked list. For a linked list containing only a head pointer, it is $O(n)$ to access the last element, but we can just use addFirst and removeFirst.

```
class Stack {
private:
  LinkedList impl;
public:
  Stack() : impl() {}
  void push(int v) { impl.addFirst(v); }
  int pop() {
    return impl.removeFirst();
  }
  bool isEmpty() const {
    return impl.isEmpty();
  }
};
```

However, asking a linked list for its size is $O(n)$ so instead, we assume that the underlying list implementation has a method isEmpty() which just checks whether the head pointer is null.

6.2 Queues

A queue is a data structure where elements are added on one end and removed on the other. For many applications first come, first served is important. For fairness, the first person to come to a store or restaurant should be served first.

The operations for a queue are called enqueue (adding an element) and dequeue (removing an element). The isEmpty() method is still required because it is an error to attempt to dequeue if the queue is empty. There is no good way of implementing a queue using a dynamic array because adding or removing from the front is inefficient. It might appear that implementing using a singly linked list is not efficient, but it can be done. A head and tail pointer are required to get to both ends of the list, but even so it is possible to get it wrong. Here is an inefficient implementation:

```
class Queue {
private:
  LinkedList impl;
public:
  Queue() : impl() {}
  void enqueue(int v) {  // O(1)
    impl.addFirst(v);
  }
  int dequeue() {              // O(n)
    return impl.removeLast();
  }
  bool isEmpty() const { // O(1)
    return impl.isEmpty();
  }
};
```

The problem is that removing from the end is inefficient because removing from the end requires a loop. The tail pointer makes it easy to find the last element, but since there is no back pointer, the only way to remove the last element is to start from the beginning and find the second to last element, decoupling the tail.

This does not mean the job is impossible though. Simply reverse the ends and enqueue on the tail, and dequeue at the head as follows and all operations become $O(1)$:

```
class Queue {
private:
  LinkedList impl;
public:
  Queue() : impl() {}
  void enqueue(int v) {  // O(1)
    impl.addLast(v);
  }
  int dequeue() {              // O(1)
    return impl.removeFirst();
```

```
  }
  bool isEmpty() const { // O(1)
    return impl.isEmpty();
  }
};
```

Last, it is possible to create an array-based queue, but this requires a custom data structure with two indices head and tail. This is also called a circular buffer. The following code shows a minimal C++ queue without implementing grow:

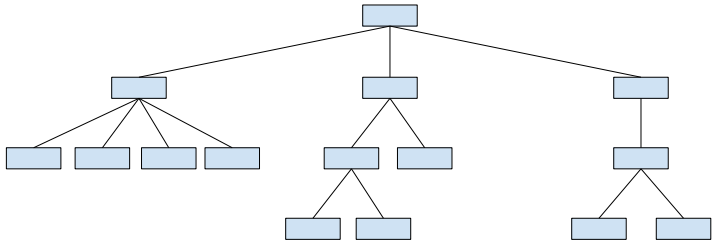
```
class Queue {
private:
  int* buffer;
  uint32_t size;
  uint32_t head, tail;
public:
  Queue(int size) : buffer(new int[size]), size(size), head(0),
     tail(0) {}
  ~Queue() { delete [] buffer; }
  Queue(const Queue& orig) = delete;
  Queue operator =(const Queue& orig) = delete;
  void enqueue(int v) {
    buffer[tail++] = v;
    if (tail >= size)
      tail = 0;
    if (tail == head)
      grow(); // buffer is full
  }
  int dequeue() {
    int temp = buffer[head++];
    if (head >= size)
      head = 0;
    return temp;
  }
  bool isEmpty() const { // O(1)
    return head == trail;
  }
};
```

7. Trees, Balanced Trees, and Tries

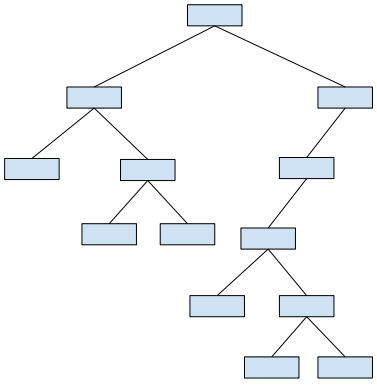
7.1 Trees

A tree is a data structure that contains nodes. One special node is designated the root. Each node can have a number of children, but no node may share children with another node. Thus, unlike the more generalized graph, trees cannot contain loops (cycles). A binary tree has nodes that may at most two children (and may have zero). Nodes without children are called leaf nodes.

The following diagram shows a general tree:



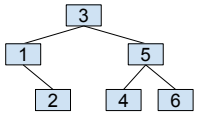
The following diagram shows a binary tree where each nod has at most two children.



Consider an ordered binary tree, for which the value in every internal node is greater than every node in the left subtree, and is less than every element in the right subtree.

Suppose we insert the following values into an ordered tree: 3, 5, 1, 2, 4, 6. The resulting

tree would look like:

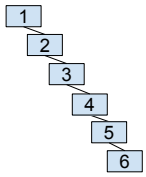


The following algorithm inserts into an ordered tree. There is a special case if the tree is empty to create the root node.

```
class Node {
  int val            // each Node has a value, in this case an
     integer
  Node left, right   // each Node has two children, each can be
     null
}
OrderedTree.insert(v)
  if root = null
    root ← new Node(v)
    return
  end
  p ← root
  while true
    if v > p.val
      if p.right = null
        p.right ← new Node(v)
        return
      end
      p ← p.right
    else if v < p.val
      if p.left = null
        p.left ← new Node(v)
        return
      end
      p ← p.left
    else
      return // this case exits if the value is already in the
         tree
    end
  end
```

Algorithm 7.1: OrderedTree.insert()

The problem with ordered trees is that their shape depends on the order in which elements are inserted. The tree above was fairly well balanced. No one branch was too long. For each level k of a binary tree has 2^k elements, the total has $2^{k+1} - 1$, and therefore for n elements, if the tree is full search to find any particular element is $O(logn)$. However, if we insert n elements in order: 1, 2, 3, 4, 5, 6 the results are quite different:+



In the above case the tree has depth n, and the binary tree is no faster than a linked list, with all the left children being nonexistent and the pointers unused. The solution to this problem is to define an algorithm to automatically balance the tree as elements are inserted. This must be done carefully – too often, and insertion can be more than $O(logn)$. Too infrequently, and searching can become slow.

The goal is to create a tree that is wide not deep. As can be seen by the above problem, this will require a balanced tree algorithm covered later in this chapter.

In n layers a complete tree can store at most $2^n - 1$ elements.

7.2 Tree Traversal Algorithms

There are three simple traversal algorithms that everyone should know: preorder, postorder and inorder.

Preorder is defined as first processing the current node, then recursively processing the left child and right child if they exist. Postorder is the opposite, first processing the left and right children recursively if they exist, then executing the operation, and inorder is defined as first recursively calling the left child, then processing the current node, then recursively calling the right child.

The algorithms are shown below. In each case, the operation is in this case printing out the current node.

```
preorder(node n)
  if n == null
     return
  print (val)
  preorder(n.left)
  preorder(n.right)
end
```

Algorithm 7.2: Preorder(n)

```
postorder(node n)
  if n == null
     return
  end
  postorder(n.left)
  postorder(n.right)
  print (val)
```

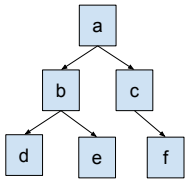
```
end
```

Algorithm 7.3: Postorder(n)

```
inorder(node n)
  if n == null
     return
  inorder(n.left)
  print (val)
  order(n.right)
end
```

Algorithm 7.4: Inorder(n)

The following diagram shows a tree and the order of traversal for each of the three algorithms.

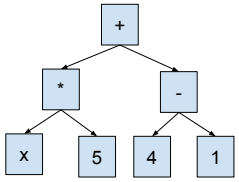


PreOrder traversal: a, b, d, e, c, f

InOrder traversal: d, b, e, a, c, f

PostOrder traversal: d e b f c a

Trees do not have to have only values. The following tree is a standard way compilers represent expressions internally after parsing.



PreOrder traversal: + * x 5 - 4 1

InOrder traversal: (x * 5) + (4 - 1)

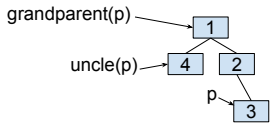
PostOrder traversal: x 5 * 4 1 - +

7.3 Balanced Trees

There are a number of different algorithms for balanced trees including Red/Black (RB-Tree), AVL Trees, Fibonacci tree, and BTrees. All have a similar concept, to avoid the problem of unbalanced trees by periodically rebalancing, doing as little work as possible each time (not constantly rewriting the entire tree) and not doing so too often so that operations are still on average, $O(\log n)$. We will only cover RB-tree, which is used by C++ map among others.

The algorithm for RB-Tree has many cases, and requires defining a few terms for clarity. First, the grandparent of a node is the parent of a parent. The uncle of a node is the parent's sibling, and since this is a binary tree, if the parent is the right child of their

parent, then the uncle must be the left child, and if the parent is the left child, then the uncle must be the right.

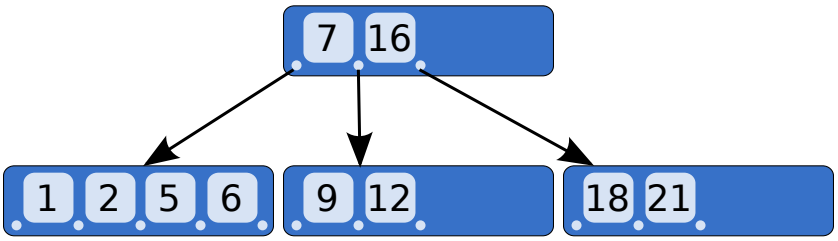


[TBD]

7.4 B-Trees

B-Trees are generalized RB-Trees with higher degree than 2, developed by Bayer and McCreight at Boeing research in 1970. The higher the degree of the tree, the fewer nodes are required. This is not an advantage in terms of computational complexity since all logs differ only by a constant factor. However, when the time to access a node is high, then the constant factor may be important. This is the case in databases, which have historically been stored on hard drives. A hard drive is a block-oriented device, so it is just as fast to access 1k bytes as it is to access 8 bytes. Given a particular block size such as 1k or 4k, the question is how to minimize access time for the database.

The answer has for the last 50 years been B-Trees. Databases typically cache the root (and perhaps the second layer as well) in RAM for speed, though to make sure the database is consistent in the event of power outages all changes must be written through to the disk as well. The illustration shown below is a B-Tree of degree 5 taken from Wikipedia, but in practice the degree is the largest that can fit in a single disk block.



For block-oriented devices like hard drives, with a block size of 4k bytes, it may be possible to store 64 values in a single block. This means that each layer of the tree has 2^6

7.5 Expression Trees

Expressions in mathematics are represented in multiple different ways. There is inline:

$(3 + 4) * 5$

preorder:

$* + 345$

and postorder:

$34 + 5*$

All three representations are actually different views of an expression tree:

7.6 Tries

A trie is a high-degree tree designed to store words. Each node typically has an out-degree of the number of letters in the alphabet. The following diagram shows a trie storing the word cat, dog, and dogged:

In order to insert into a trie, start at the root. For each letter in the string, look up the pointer. If it is null, create a new node. Regardless, traverse to the pointer.

```
insert(word)
  p ← root
  foreach letter in word
    if next[letter] == null
      next[letter] = new Node()
    end
    p = p.next[letter]
  end
end
```

Algorithm 7.5: insert(word)

To find out if a word is contained within a trie is similar except that if a pointer is null the answer is immediately false.

```
contains(word)
  p ← root
  foreach letter in word
    p = p.next[letter]
    if p == null
      return false
    end
  end
  return p.isWord
end
```

Algorithm 7.6: contains(word)

Testing for containment of a prefix is almost the same as containment of a word. The only difference is that if the node exists then a prefix must exist, there is no need to check whether the node has a word or not.

```
containsPrefix(word)
  p ← root
  foreach letter in word
    p = p.next[letter]
    if p == null
      return false
    end
  end
  return true
end
```

Algorithm 7.7: containsPrefix(word)

7.7 Rope

This section is incomplete. The original paper on rope is in the reference directory.

Rope is a tree structure that replaces simple strings so operations can be performed on large text in O(log n). The concept of rope is a tree of fixed strings, so at the lowest level, the strings are immutable (do not change).

Let's compare operations on a large file, 100 million bytes, using a string and rope. The cases to consider would be operations that are constantly performed on text files: appending new text at the end, inserting text in the middle (typical editing), deleting text, and pasting in a section of text.

7.8 Efficient Implementation of Tree Operations for a Text Editor

One potential project is to implement a data structure suitable for use in a text editor, something that wold be able to edit huge files instantly. It must support editing in a particular location because that is what typically happens. A human will type, then delete, then type more in a location. For a large document, the text editor also needs to be able to display what is on the screen efficiently. For a multi-user system like google docs, we can conceive of multiple users across the internet all editing the same document simultaneously.

In order to improve on rope, and add functions to make the above more efficient, we have to develop the following concepts:

Node

A node of the tree can be more complicated than rope. We should consider higher degree such as 4 or 16 so there can be fewer nodes and less overhead. The tradeoff will be complexity, and somewhat more work when inserting and deleting. Operations will still be $O(\log n)$ but with a tree of degree 4 an insertion might be $O(\log_4 n)$ and while that is just a constant factor, it might be a factor of 2 which would be significant for text processing.

Line

Rather than having a tree structure down to the individual letter, storing a string for each line is probably more efficient. At this scale, inserting text would mean replacing one string with another. So leaves in this system are strings, but they can be non-growable (immutable) strings, every time one is edited just swap it out with a bigger or smaller one.

Cursor

A cursor location is like an iterator into the rope structure. It must represent a location within a document. It is vital that if multiple cursors exist in a document (such as for different users editing the same document simultaneously) that inserting in one does not invalidate the other cursors.

Range

A range within the document is a region defined by a start and end cursor, or a start cursor and length. Again it is vital that multiple people editing a document do not invalidate each others' cursors or ranges.

Window

In order to edit a large buffer we must be able to view a rectangular region. The document is vastly bigger than an individual window in the vertical, but it is also possible that the document is wider than the window.

7.9 References

8. Hashing

8.1 Introduction

Hashing is a technique of rapid search that turns the target data (the key) into the location where it may be found in O(1) time. This means that in theory it does not take longer to search for the data regardless of how many items there are, the fastest possible search. However it is not quite that simple.

The critical part of any hash algorithm is a hash function which turns the bits of the input into where to locate them. A poorly chosen hash function may result in many values located in the same place (collisions) which will then require a secondary means to find the right one. In this chapter, we cover how to organize a hashmap to efficiently find the data, how to write a good hash functions

8.2 Hash Functions: Key to Hashing

8.2.1 Hash function for integers

The goal of a hash function is to uniformly spread the values across all the bins.

8.2.2 Hash function for strings

Hashing strings must not collide for anagrams because many words contain the same letters (tea, eat, ate, eta).

Hashing strings must uniformly use the entire hashmap, it's not great if short words all bunch up low in the hash map.

Ideally, if we could hash 4 or 8 bytes at a time, we could do fewer operations, but in some circumstances, bytes are not aligned (meaning we have to read twice to read a single 4 or 8byte chunk). Worse yet, if we don't know the length of the string it is hard to do this. Theoretically of course, it does not matter if we do strings 1 byte at a time or 8, but in practice that is a huge constant factor.

The following hash is representative but not great or tested. Note the tricks. Because each pair of shifts is ored, the compiler recognizes that is a rotate operation so it is optimized into a single instruction, and we don't throw out any bits. We use xor to combine the bits because that is bit-neutral. If we used or, there would be more and more ones. If we used and, there would be zeros unless two coincided. A function like this must be heavily tested, but something of this type should work well. Given two different rotates mixes bits across different parts of the hash, the bits of any byte have been xored in 17 bits away and 7 bits away and since those numbers are prime, the next time those will be shifted 17+17 and 17+7 ... so in just two cycles we will have mixed up a lot of bits.

The more complicated this kind of function gets, the better it probably hashes the bits, but also the slower it gets. That's the tradeoff.

```
hash(key)
  sum = 0 // or set an arbitrary constant with 1s and 0s
  for i = 0 to len(key)-1
    sum = ((sum << 17) | (sum >> 15)) ^
          ((sum << 7) | (sum >> 25)) ^ key[i]
  end
  return sum
```

Algorithm 8.1: hash

8.2.3 Hash function for objects and pointers

Objects in most programming languages like C++ and Java are guaranteed to be at a unique location. For mapping objects therefore, we can use that location and hash it. That means a hash algorithm capable of hashing pointers.

The vital fact is that pointers are typically wordaligned. This means that the hash function must use all bins even though the numbers will all be multiples of 4 for a 32bit computer or 8 for a 64bit computer.

Imagine if the hash function leaves the low 2 or 3 bits intact. This means, for a hash map of 10k buckets, that only every 4th one is used. That would be disastrous and could result in way more collisions than planned.

The following table shows what it would look like if values hashed only to even multiples of 4 elements:

A				B				C				D			

The above table, which we think has 16 elements, effectively has only 4, and with 4 elements would have likely had a collision.

8.3 Linear Probing

```
LinearProbing.insert(key, value)
```

```
  h ← hash(key)
  do
     h ← (h + 1) mod size
    p ← table[h mod size]
  while table[h] ≠ null
    h ← h+1
    if h ≥ size
      h ← 0
    end
  end
  table[j] ← (key,value)
end
```

Algorithm 8.2: LinearProbing.insert

8.4 Handling Collisions

Collisions in hashing are inevitable. And in the case of the linear probing algorithm, if a collision happens in one bucket it spills over to the next. Because of this property, busy buckets can ”ruin the neighborhood” for other bucket near them. The solution is to have an excess of buckets so that the probability is that there is always an empty bucket next to each one that is used. This means that linear probing does best with 100 percent extra space, or to put it another way, 50 percent open space.

s = “abc” a=1, b= 2, c=3 a=97 b=98 c=99

```
hash(s)
  sum ← 0
  for i ← 0 to length(s)
     sum ← sum + s[i]
  return sum
end
```

Algorithm 8.3: SimpleHashString

The above algorithm is very poor because for any words with the same letters the hash value is the same. For example, eat, ate, tea all would hash to the same value.

A much more effective approach is to weight each letter by different amounts so that ab does not have the same hash value as ba.

```
hash(s) {
  sum ← 0
  for i ← 0 to length(s)
     sum ← sum * 26 + s[i]
  return sum
}
```

Algorithm 8.4: Base26tashString

This algorithm too has a problem, in that long string will see the initial parts multiplied by large values and not affect the bottom bits as much. In order to make sure that every letter affects as much as possible, between adding in the new letters we can shift/rotate so that bits from early affect high bits and low bits. The exact winning design of a hash function requires vast amounts of testing, but the following is schematic.

```
hash(s) {
  sum← length(s)
  for i ← 0 to length(s)
     sum ← ((sum << 17) | (sum >> 15)) ^ ((sum << 7) | (sum >>
        25)) + s[i]
  return sum
}
```

Algorithm 8.5: Hash with bit mixing

The above code uses some very specific tricks. First, the numbers of the bit shifts are not divisible by a power of 2 so successive shifts will cover more of the word, overlapping in complex ways. Second, because $17 + 15 = 32$, and $7 + 25 = 32$ both of the ored terms are converted by C++ into a single rotate rather than two shifts and an or. This menas the above code only has 4 operations: 2 rotates, an XOR to combine them, and an addition to add in the new letter (which could also be an XOR).

8.5 The Birthday Paradox

What is the probability that two people in a class of 20 have the same birthday?

Given hash function f(x) = x^2 and data 5, 9, 16, 31

show the table after each insert

					5		
	9				5		
16	9				5	31	

8.6 Quadratic Probing

In linear probing, a collision results in moving to the next bucket over. If that is full, then the algorithm must keep scanning forward until it finds an empty bucket. Quadratic probing tries to get away from the neighborhood with the collision faster, by first adding 1, the next time $2^2 = 4$, the next time $3^2 = 9$, and so on. We cover it in the course for

completeness, but Quadratic Probing isn't very useful. Far more important is a good hash function that uniformly spreads the values across the entire table.

To visualize the problem, consider a pathological hash function f(x) = 1 which always puts each object in the same bucket, resulting in 100 percent collision rate. The following tables show what it would look like to put in 5 values into the table both with linear probing and quadratic probing. Both have exactly the same collision resolution, except quadratic probing is more complicated and expensive.

		1					
		1	2				
		1	2	3			

		1					
		1	2				
		1	2				3

```
QuadraticProbing.insert(key, value)
  h ← hash(key)
  i ← 1
  do
     h ← (h + i*i) mod size
    i ← i + 1
    p ← table[h mod size]
  while table[h] ≠ null
    h = h + 1
    if h >= size
      h = 0
    end
  end
  table[j] ← (key,value)
end
```

Algorithm 8.6: QuadraticProbing.insert

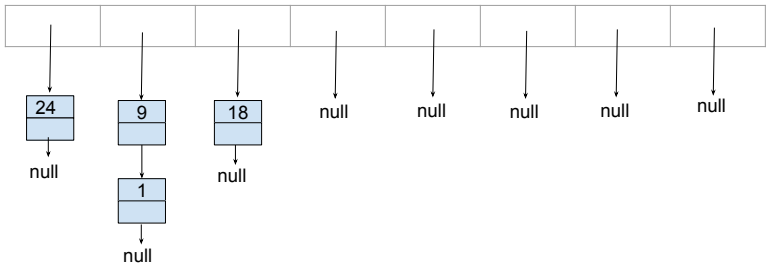
8.7 Linear Chaining

Linear chaining is a different scheme where every bucket in the hash table is a linked list. The head pointer is null if the bucket is empty. Linear chaining creates far fewer collisions because one back bucket does not poison its neighborhood. It is still vulnerable to a bad hash function because if a function puts too many elements in one bucket, the collisions result in a large linked list, and traversal is $O(n)$ rather than $O(1)$.

When inserting items into the bucket, insert first (ie build the linked list in reverse). This is a good strategy because in general, the most recent item is the most likely to be accessed again (the principle of locality).

Linear chaining is better than linear probing for collisions because each bucket does not affect its neighbors, and it also uses less memory for empty buckets which only need a single pointer to null. However, when buckets are filled the nodes must contain not only the data but a pointer to the next node, so there is more memory for these. Another advantage of linear chaining is that an empty bucket is represented by null. With linear probing, there must be a value designating an empty bucket, and that value can therefore not be represented in the hashmap.

The following example shows linear chaining for a hashmap of size 8 with $hash(k) = k \mod 8$ and inserting the values 1, 8, 9, 17, 26.



8.8 Perfect Hashing

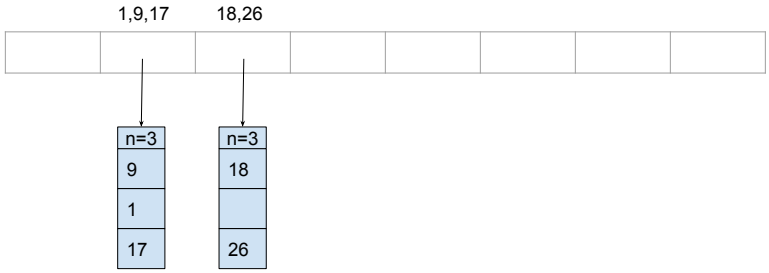
For data that is known in advance, it is possible to create a special hash table that may be faster by completely eliminating collisions. This would normally be completely impossible because of the birthday paradox. Consider an example with 1000 elements stored in a table with 1 million buckets. On the surface, it would seem fairly easy to achieve no collisions: after all, only one bin in 1000 is occupied. But if the hash function for whatever reason gives the same number for two values, it does not matter how many bins there are, and with so many values, that becomes quite probable.

Perfect hashing defeats this problem by breaking a large number of elements using a hash function, so that no bin has more than a few elements. Because collisions don't matter, the hash function can be simple and fast. Then, each bucket uses a second simple but different hash function to resolve any collisions. Because each bucket has very few elements, the birthday paradox works in reverse and even a relatively simple hash function can separate those elements, as long as it is different from the main one.

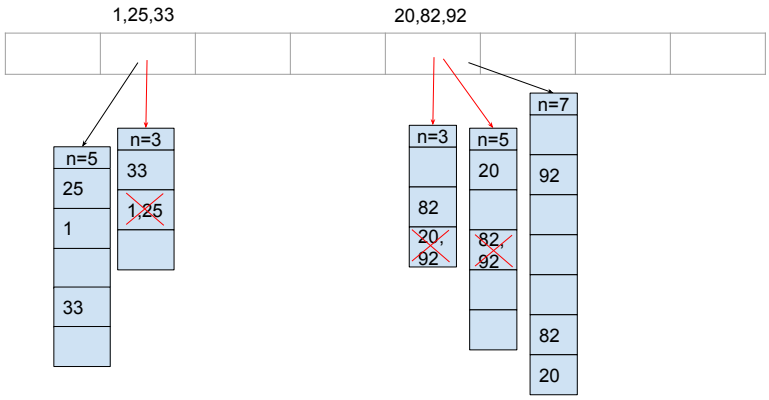
The following example illustrates perfect hashing using the hash function $f(x) = x \mod n$. The secondary hash function is also $f(x) = x \mod m$ but m is the size of the secondary table which is chosen to be different (and always relatively prime) to the larger table. It is possible to write code that will automatically find a hashmap using this scheme that has no collisions for the data chosen, and once found, this can be somewhat faster than hashmaps with more complicated hash functions, even though it requires two hash evaluations per lookup.

Example: insert 1, 9, 18, 17, 26 into a hashmap size $n = 8$ using perfect hashing. The

function is simple and fast, and in any case there are 5 elements in only 8 buckets. Bucket 1 contains 1, 9, and 17. Bucket 2 contains 18 and 26. We then pick a secondary hash table size for each bucket. For bucket 1, because there are 3 elements we should choose at least 3 or we are guaranteed a collision. We can start with 3. Never choose 4, because it evenly divides the original hash table with size 8. When we insert the numbers into the second hash table, we get $1 \mod 3 = 1, 9 \mod 3 = 0, 17 \mod 3 = 2$. So each value which was in the same bin happens to be in different bins in the secondary table, and all collisions are resolved. For bin 2 we select size 3 (because the original size 8 is evenly divisible by 2). $18 \mod 3 = 0$ and $26 \mod 3 = 2$ so again, collisions are resolved.



The following example shows what would happen when values do collide. The algorithm simply picks the next size for the secondary hash table and tries again. Example: insert 1, 25, 33, 20, 84, 92 into a perfect hash table size $n = 8$. Bin 1 gets elements 1, 25, 33 and bin 4 gets 20, 82, and 92. To resolve the collisions, the algorithm selects the first relatively prime size (3) for the secondary table and tries to insert the three numbers but this time $1 \mod 3 = 1$ and $25 \mod 3 = 1$ so there is a collision. The algorithm then tries size 5 (skipping 4 because it evenly divides the main table size). The result is $1 \mod 5 = 1, 25 \mod 5 = 0, 33 \mod 5 = 3$. For bin 4, the algorithm tries a secondary hash table with size $n = 3,5,7$. $20 \mod 7 = 6, 82 \mod 7 = 5, 92 \mod 7 = 1$. The collision is resolved.



8.9 Cryptographic Hashing and Digital Signatures

A cryptographic hash is one that not only evenly mixes the bits, but one that is deliberately designed to be difficult to construct an equivalent hash value with a

different set of bytes. Imagine a document composed of letters. If we add the letters together to get a checksum, which is a primitive hash value, it would be easy to create a new document that is equivalent to the old one, that still has the same hash value. Suppose embedded in the document is the text:

"I hereby bequeath to my children all of my possessions."

A criminal wishes to change the will. In order to change the statement to:

"I hereby bequeath to Fred Wilson all of my possessions."

All the criminal has to do is calculate the change in the checksum and add or subtract letters to achieve the same one. The goal with cryptographic hashing is to make that difficult or impossible.

By definition, if a hash is 256 bits, far smaller than the document that it hashes, then there must be multiple documents for which the value is the same. However, the hope is that all the other documents are nonsensical and will be obviously wrong.

In order to sign a document, we make use of RSA (see Number Theoretic chapter). In asymmetric encryption, there are two operators. Anyone can encrypt E() a message using the public key, and only the possessor of the private key can decrypt D() the message. And because they are inverses, $E(D(m)) = D(E(m)) = m$.

To digitally sign a document, the author computes a secure cryptographic hash, then decrypts it, even though it was never encrypted. The hash is appended to the document.

$doc, D(hash(doc))$

The receiver gets the document and computes the hash, then encrypts the hash that is appended with it:

$hash(doc), E(D(hash(doc))$

Since $E()$ and $D()$ are inverses, the two should match. If they do, the document is supposed to be as originally written. To prevent tampering by a third party along the way, who could at least try to generate a bogus (meaningless) document which hashes to the same value, it is recommended that the entire message be encrypted so that only the recipient can read it.

8.10 Avoiding Overuse of Hashing

There is a natural tendency having learned hashing to use it too much, since it is very powerful. The following situations are special cases where hashing can be simplified into an array. Remember that any time the keys have a pattern, inverting it is potentially the best possible hash function.

- The values are sequential integers n, $n+1$, $n+2$, ... (use an array $x[i-n]$)

- The values are an arithmetic series n, $n+k$, $n+2k$, ... (use an array $x[(i-n)/k]$)
- The values are floating point n, $n+0.1$, $n+0.2$, ... For this one, you have to be careful because of roundoff error. If you can come up with any expression that rounds this off and turns the result into an integer, do it.
- The values are integers in a range $[a,b]$ with many holes. If the range is not too big, just use an array. If memory requirements are too large and there are many holes, then use hashing.
- The values are in a geometric series n, nk, nk^2, ... (use an array, $x[c_1 log(i-n)]$

8.11 References

- Bob Jenkin's Hash: http://burtleburtle.net/bob/index.html
- Paul Hsieh SuperFastHash: http://www.azillionmonkeys.com/qed/hash.html
- Stanford Bit Hacks: https://graphics.stanford.edu/~seander/bithacks.html

9. Sorting

9.1 Introduction

Sorting is one of the oldest topics in computer science. Ordering a list of elements requires defining a desired order, and for the purposes of this chapter, we will always use ascending order since it makes no difference. Minimally, just to determine what is in each element of a list requires $O(n)$ time. Simple methods that compare adjacent elements will require $O(n^2)$ because for an element to move from one end of a list to the other one element at a time takes $O(n-1)$ operations, and that is only for a single element. We will cover six methods. The first two, bubble sort and selection sort are useful only for comparative purposes. They are poor algorithms and are dominated in all cases by better algorithms.

9.1.1 Bubble Sort

The bubble sort is a classic, simple sort that compares adjacent elements to bubble one element to the end (or the start) of a list. having done so with $O(n)$, the remaining elements have size $n-1$ and the process repeats with one fewer elements.

The following table shows pass 1 of a bubble sort. Notice the number 10 bubbling to the correct position on the first pass.

10	9	8	7	6	5	4	3	2	1
9	10	8	7	6	5	4	3	2	1
9	8	10	7	6	5	4	3	2	1
9	8	7	10	6	5	4	3	2	1
9	8	7	6	10	5	4	3	2	1
9	8	7	6	5	10	4	3	2	1
9	8	7	6	5	4	10	3	2	1
9	8	7	6	5	4	3	10	2	1
9	8	7	6	5	4	3	2	10	1
9	8	7	6	5	4	3	2	1	10

9.2 Bubble Sort

```
BubbleSort(a)
  for i = 0 to a.len - 1
    for j = i+1 to a.len-1;
      if a[j] > a[j+1]
        swap(a[j], a[j+1])
      end
    end
  end
end
```

Algorithm 9.1: BubbleSort

This version of the bubblesort will stop early if a complete pass is made without any swapping. Thus the lower bound on complexity is $\Omega(n)$ Unfortunately, adding this check slows the algorithm down (increases the constant) for all cases. Insertion set is both faster and similarly has a lower bound of $O(n)$ so bubble sort is useless and dominated by insertion sort.

```
BubbleSort(a)
  for i = 0 to a.len - 1
    done = true
    for j = i+1 to a.len-1;
      if a[j] > a[j+1]
        swap(a[j], a[j+1])
        done = false
      end
    end
    if done
      return
    end
  end
end
```

Algorithm 9.2: ImprovedBubbleSort

9.3 Swapping

Swapping two elements in the list is the basis of all the algorithms covered in this chapter, so it's worth taking a moment to discuss how this is done. The basic operation of swapping two values can use a third variable to hold one:

```
swap(a,b)
  temp ← a
  a ← b
```

```
  b ← temp
end
```

Algorithm 9.3: Swap

Swapping with integers is also possible using inverse operations. this has occasionally used as a trivia interview question, ie "how can you swap two variables without a third one as a temp? In days when computers had few registers this could actually speed up code, but this is no longer true.

```
swap(a,b)
  a ← a + b
  b ← a − b
  a ← a − b
end
```

Algorithm 9.4: SwapIntegersUsingInverseOps

Finally, since XOR is its own inverse, the following also works (try it)

```
swap(a,b)
  a ← a ⊕ b
  b ← a ⊕ b
  a ← a ⊕ b
end
```

Algorithm 9.5: SwapIntegersUsingXOR

9.4 Selection Sort

The selection sort is a completely different method than bubble sort. Rather than swap elements that are out of order, find the biggest element, and put it at the end. From then on, ignore the last element and work on the first $n-1$ elements again. Since this method involves swapping only n times, and swapping requires more time than comparison, the constant on this algorithm is lower than bubblesort. But it is still an $O(n)$ algorithm, and since it does not swap multiple elements, there is no way to tell if the algorithm is over early. Selection sort is therefore also dominated by insertion sort.

```
SelectionSort(a)
  for i = 0 to a.len − 1
    min = a[i]
    minpos = i
    for j = i+1 to a.len−1;
      if a[j] < min
        min = a[j]
        minpos = j
      end
```

```
    end
    swap(a[i], a[j])
  end
end
```

Algorithm 9.6: SelectionSort

9.5 Insertion Sort

Insertion sort also does not use swapping. Instead, we consider a subset of the list and try to make it sorted. The first element is by defition sorted. The first two elements might not be sorted, so if the second is out of order, put it before the first.

This is the algorithm used to sort a hand of playing cards.

9.6 Quicksort

Quicksort is generally considered the fastest exchange-based sort. With fairly low overhead compared to heapsort or mergesort, it is generally the first choice for sorting (for example in the C and Java library, and formerly in the C++ libray, which is now using spreadsort). The original quicksort is due to C.A.R Hoare in 1959, published in 1961. A modification due to Lomuto is an alternate scheme. Both rely on selecting a value called the pivot, partitioning the array into two parts (all elements less than the pivot, and all elements greater than or equal to the pivot), and then recursively calling the function to continue until the array is sorted.

We will consider three schemes for selecting the pivot:

- $pivot = (x[L] + x[R])/2$
- $pivot = (x[L]] + x[R] + x[(L + R)/2)/3$
- $pivot = x[random]$

Of these, the only one that consistent works is the random pivot. Any algorithmic scheme to select a pivot can run into data for which it is pathological and results in $O(n^2)$. For large arrays this means that the code will never finish, and most industrial algorithms have a fallback to a different method such as heapsort in this event.

The following table shows the original Hoare algorithm working on an ideal case, an array that is exactly opposite the desired ascending order. A pivot is selected, and on the left the first value is found that is greater than the pivot, and on the right the first value that is less than the pivot. These are then exchanged and the algorithm goes on to do this until the two indices cross in the middle.

$pivot = (x[L] + x[R])/2$

10	9	8	7	6	5	4	3	2	1
i	9	8	7	6	5	4	3	2	j
1	i	8	7	6	5	4	3	j	10
10	2	8	7	6	5	4	3	9	1
10	2	i	7	6	5	4	j	9	1
10	2	3	7	6	5	4	8	9	1
10	2	3	i	6	5	j	8	9	1
10	2	3	4	6	5	7	8	9	1
10	2	3	4	i	j	7	8	9	1
10	2	3	4	5	6	7	8	9	1
10	2	3	4	6	ij	7	8	9	1

Partitioning ends when the two indices i and j become equal. In the above example, every element less than or equal to the pivot is to the left. It is equally valid to put every value greater than or equal to the pivot to the right. The decision of which side the pivot goes to is arbitrary, but results in two slightly different results. The following algorithm for partitioning, the original due to Hoare, is complexity $O(n)$ because for each pass of the array, all elements are scanned. The first time, there is a single call to quicksort, and it scans all elements. The second time, there are two calls to quicksort, and each scans roughly half the array, but importantly together, they scan the whole array.

The following shows an extremely bad case using the pivot algorithm $pivot = (x[L] + x[R])/2$. The sort is inefficient because each time the pivot is the smallest in the set, and therefore the array is split into one element, and the rest, equivalent to selection sort but with a higher constant due to increased complexity. In fact on modern operating systems the stack is limited to harden against attacks, so in this case a large enough list will crash due to the stack growing too deep.

1	3	5	7	10	9	8	6	4	2
1	3	5	7	10	9	8	6	4	2

```
partitionOriginal(a, L, R)
  i ← L
  j ← R
  while i < j
    while i<j and x[i] ≤ pivot //alternatively < pivot
      i ←i+1
    end
    while i<j and x[j] > pivot //alternatively ≥ pivot
      j ←j−1
    end
    if i < j
     swap(a[i], a[j])
    end
  end
  return i
end
\endalg
```

```
Once the array is partitioned, quicksort is recursively called on
    the two halves of the array.

\alg{Quicksort}{nlogn}
quicksort(a, L, R)
  i ← partitionOriginal(a, L, R)
  quicksort(a, L, i−1)
  quicksort(a, i, R)
end
\endalg

Note that the choice of where the partition is: i−1 and i or i,
   and i+1 depends on the code in partition and whether the loops
    are specified as x[i] <= pivot and x[j] > pivot or x[i] < pivot and
   x[j] >= pivot.

The Lomuto variant of selecting the pivot is more complicated and
    slower, but it does result in a partition into the left side,
    the right side, and the pivot which is in the correct place
   and can therefore be ignored.

\alg{LomutoPartition}{n}
```

Algorithm 9.7: PartitionOriginal

The algorithm for quicksort is tricky, and most implementations found on the internet are wrong and will not work. One very popular mistake is to select as the pivot the first or last element in the list which is $O(n^2)$.

Algorithm 9.8: Quicksort

9.7 Heapsort

Heapsort is another $O(n \log n)$ algorithm, but radically different than quicksort. Heapsort does not rely on divide and conquer, and it totally rearranges all the elements even if they are sorted. Nonetheless, in terms of complexity it is competitive with quicksort even though the constant factor is higher.

Heapsort relies on first creating a heap, a binary tree in which every parent is bigger than their children. Creation of a heap takes n log n time. Then, by pulling out the root and placing it at the end, which takes log n time, it is possible to create a sorted list with another $O(n \log n)$.

Algorithm MakeHeap $O(n \log n)$

```
makesubheap(a, i, n)
  if i ≥ n or 2i ≥ n
    return
  end
  if 2i+1 ≥ n
    if a[2i] > a[i]
      swap(a[i], a[2i])
      makesubheap(a, 2i, n)
    end
  else
    if a[2i] > a[2i+1]
      if a[2i] > a[2]
        swap(a[i], a[2i])
        makesubheap(a, 2i, n)
      end
    else
      if a[2i+1] > a[i]
        swap(a[i], a[2i+1])
        makesubheap(a, 2i+1, n)
      end
    end
  end
end

MakeHeap(a)
  for i = a.length/2 to 1
    makesubheap(a, i, n)
end
```

Algorithm 9.9: MakeHeap

```
  for i = n−1 to 2
    swap(a[0], a[i])
    makesubheap(a, 0, i−1)
  end
end
```

Algorithm 9.10: rebuildHeap

Algorithm Heapsort $O(n \log n)$

```
HeapSort(a)
  MakeHeap(a)         // O(n log n)
  rebuildHeap(a, i) // O(n log n)
end
```

Algorithm 9.11: HeapSort

9.8 Merge sort

Merge sort has a higher constant than quicksort or heapsort, and it requires $O(n)$ extra storage, so it is not generally used for ordinary sorting. However, when the data being sorted is either larger than available memory, therefore on disk or tape, or when the data is in a linkedlist (in other words, for cases where sequential access is far faster than random access) then mergesort is far faster than the alternatives.

Even 50 years ago, with very small RAM capacity and slow magnetic tapes, it was possible to sort hundreds of millions of records by reading in data on two magnetic tapes and writing it out on one. In this way, the speed of each pass of the sort is limited by the sequential access speed of the medium. Today the same is true for hard drives which are far faster when accessing data sequentially.

Mergesort could be called recursively but this is fairly tricky given the need to keep both a primary array and temporary storage. In this case, bottom up is far simpler. It is also far faster because a great deal of time is taken by all the recursive calls. In a mergesort of 1 million elements, the bottom layer merges 500,000 times, each a function call for a recursive version.

Consider a section of an array of size 1. It obviously cannot be in the wrong order. Now consider merging two of them. The first example shows a pair of numbers already in the right order. It is copied to the output verbatim:

1	2
1	2

The second example shows a pair of numbers in the wrong order. It must be copied into the output in reverse order:

4	3
3	4

Now consider a list with many such units. First we merge into pairs of elements, each sorted. Then we merge the sorted pairs into groups of 4 elements, all sorted. Finally we merge two groups of 4 into 8. The same principle of divide and conquer is used as in quicksort, but in reverse. There is no pivot (good) and there is a need for $O(n)$ temporary storage because reading the data must write it back into different memory. However, we do not have to allocate $O(n)$ memory each time, because once the data has been copied into the temporary copy, the original can be overwritten, and the next pass can copy from the temp back to the original data.

cells of size 1	7	8	6	5	4	3	1	2
cells of size 2	7	8	5	6	3	4	1	2
cells of size 4	5	6	7	8	1	2	3	4
merge 4 into 8	1	2	3	4	5	6	7	8

At any point in time, if two segments are sorted and being merged, if the rightmost element of the left hand group is less than the leftmost element of the righthand group,

merging does not need to be done because the two groups are already sorted. For example:

1 | 4 | 5 | 8 | 12 | 13 | 16 | 19

At the beginning, mergesort is relatively expensive. A way to reduce the cost is to sort the smallest sections using a different method. In the following example, below $n = 8$ insertion sort is used to order each group without requiring temporary storage or the other overhead of mergesort. For an example with a hard drive, the obvious efficient thing to do is read in a number of blocks that fit well into memory, sort them in memory, then write them back and move to the next blocks. In order to avoid the disk seeking back to the start of each group of blocks, it would even be possible to use the temporary storage, read in from one hard drive and write out to another with each group sorted.

The optimal size of the lowest method must be determined by experiment, and is clearly far larger than $n = 8$ shown in the algorithm below. After this initial phase, each group is merged into a group exactly double the size.

```
constexpr int minPartition = 8;
void insertionSortSmall(int a[], uint32_t n, uint32_t
   minPartition) {
  for (uint32_t i = 0; i < n; i += minPartition) {
    for (uint32_t j = i+1; j < i + minPartition; j++) {
       int temp = a[j];
       for (int32_t k = j-1; k > i; k--)
          if (a[k] > temp) // there is a bug here...
           a[k+1] = a[k];
          else {
            a[k+1] = temp;
            break;
          }
    }
  }
}

void bottomUpMergeSort(int* a, uint32_t n) {
  insertionSortSmall(a, n, minPartition); // first sort each
     group
  uint32_t partSize = minPartition;
  uint32_t partSize2 = partSize + partSize;
  uint32_t last = n - partSize2;
  int* temp = new int[n]; // allocate temporary storage O(n)

  while (partSize2 < n) {
    // first check whether this pass is necessary
     uint32_t countSortedPartitions = 0;
     for (uint32_t i = 0; i < last; i += partSize2){
       if (a[i+partSize-1] < a[i+partSize]) {
          countSortedPartitions++;
```

```
      }
    }
    if (countSortedPartitions == n / partSize2)
      continue; // skip this iteration
    for (uint32_t i = 0; i < last; i += partSize2) {
      uint32_t end1 = i+partSize, end2 = i + partSize2;
      uint32_t j = i + partSize;
      uint32_t dest = i;
      while (i < end1 && j < end2) {
        if (a[i] < a[j])
          temp[dest++] = a[i++];
        else
          temp[dest++] = a[j++];
      }
      while (i <= end1)
        temp[dest++] = a[i++];
      while (j <= end2)
        temp[dest++] = a[j++];
    }
    partSize = partSize2;
    partSize2 = 2*partSize2;
    swap(a, temp); // each iteration, swap pointers so temp is
       the other one
  }
  int log2 = (log(n) - log(minPartition)) / log(2);
  if (log2 \% 2 != 0) { // if data ends in temp, swap back one
     more time
    memcpy(temp, a, n * sizeof(int));
    swap(a, temp);
  }
  delete [] temp; // get rid of temporary storage
}
```

9.9 Radix sorting

[TBD]

9.10 Spreadsort

Spreadsort is a relatively new algorithm due to Steven Ross 2002.

9.11 Sorting Large Records, and Sorting on Multiple Keys

[TBD]

9.12 Shuffling

Shuffling is the reverse of searching. Just as in the physical world, entropy is ever-increasing. It is easier to shuffle an array than to sort it. When we sort an array we are finding one correct ordering amid all the possible $n!$ orders. Sorting is $O(n^2)$ for brute force solutions, and best-case $O(n \log n)$ for any sort involving exchanging elements.

Shuffling is also $O(n^2)$ for a poor solution, which is covered here because it has an interesting feature, and is $O(n)$ for the good solution.

9.12.1 Brute Force Shuffling

The brute force shuffling algorithm takes each element from an array and adds it to a new list. At first, the probability is 100 percent that the algorithm finds an element. But the more elements have been removed, the higher the probability is that the element selected is one that has already been chosen. While this is a very poor algorithm, it is a good illustration of probability in the calculation of complexity.

```
Shuffle(a)                                                          1
  count = 0                                                         2
  for i = 0 to a.len - 1                                            3
    do                                                              4
      r ← random(0,a.len - 1)                                       5
    while a[r] < 0                                                  6
    b[count] = a[r] // pull the number out and add to the 2nd       7
       list
    a[r] = -1       // mark no longer there                         8
    count = count + 1                                               9
  end                                                               10
  return b                                                          11
end                                                                 12
```

Algorithm 9.12: BruteForceShuffle

The complexity of the inner do..while loop is the interesting part of this algorithm. The first time, the loop executes deterministically 1 time. However, as the elements are removed from the original array, the probability of the inner loop finding a random element goes down. The last time, the probability is only $1/n$ that the value is found. How then does this translate into the number of times the loop executes?

With a probability of $1/n$ it is not guaranteed that the last number will be found in n tries. However, while the probability is not $n\frac{1}{n} = 1$, it is on the order of n. It might take $2n$ or $3n$, but the expected value is on the order of n. Since the outer loop is also $O(n)$ the complexity of the algorithm is therefore $O(n^2)$

9.12.2 Unfair Shuffling

A shuffle is first and foremost supposed to be fair. That is, every value is supposed to be equally likely to wind up in any location. This second algorithm, while efficient, turns out to be unfair for a very subtle reason.

```
UnfairShuffle(a)
  for i = 0 to a.len - 1
    swap(a[i], a[random(0,a.len-1)]
  end
end
```

Algorithm 9.13: UnfairShuffle

Because each random number is selected from the entire list, a number can be moved to a position, and then can be moved again. Numbers that are near the beginning when sorted (0, 1) are more likely to be moved twice, and this has a noticable effect on the distribution of probabilities of where they will end up. Consider an array of 10 elements with 0 to 9, for example. The number zero is the first element and will be placed first. If it is placed at element 9, it has multiple chances of being moved again if the random selection happens to pick 9 again, and will definitely be moved again when i becomes 9. So the odds that the number 0 ends up in the last slot is noticeably lower. See the table below for a frequency distribution given $n = 10^7$ trials.

	0	1	2	3	4	5	6	7	8	9
0:	0.1	0.13	0.12	0.11	0.1	0.098	0.092	0.087	0.082	0.077
1:	0.1	0.094	0.12	0.12	0.11	0.1	0.096	0.091	0.086	0.082
2:	0.1	0.095	0.09	0.12	0.11	0.11	0.1	0.096	0.091	0.086
3:	0.1	0.095	0.091	0.087	0.12	0.11	0.11	0.1	0.096	0.092
4:	0.1	0.096	0.092	0.089	0.086	0.12	0.11	0.11	0.1	0.098
5:	0.1	0.097	0.093	0.091	0.088	0.086	0.12	0.11	0.11	0.1
6:	0.1	0.097	0.095	0.093	0.091	0.089	0.087	0.12	0.12	0.11
7:	0.1	0.098	0.096	0.095	0.093	0.092	0.091	0.09	0.12	0.12
8:	0.1	0.099	0.098	0.097	0.097	0.096	0.095	0.095	0.094	0.13
9:	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

9.12.3 Fischer-Yates Shuffling

Only a slight modification is needed to make the unfair shuffle fair. Each time we pick a random number, we are selecting one value for the final list. Once selected, we must never pick again. Assuming a fair random number generator, this will work.

The following algorithm counts down rather than up, and each time, it picks a random element and puts it last. Then it never looks that that element again. The critical difference is that the random number is selected from 0 to i, not the length of the array.

```
FischerYates(a)
  for i = a.len - 1 to 1
    swap(a[i], a[random(0,i)]
end
```

Algorithm 9.14: FischerYates

9.13 References

https://en.wikipedia.org/wiki/Insertion_sort
https://en.wikipedia.org/wiki/Quicksort
https://en.wikipedia.org/wiki/Heapsort
https://en.wikipedia.org/wiki/Merge_sort
https://en.wikipedia.org/wiki/Radix_sort
https://en.wikipedia.org/wiki/Spreadsort
see the ref directory for the original papers on spreadsort

10. Searching

10.1 Introduction

Searching is another classic topic in computer science. Sorting data is important, but largely the reason it is important is that we are searching a great deal afterwards, and sorting makes the subsequent searching faster.

In this chapter, we cover linear searching, the obvious brute force approach, then linear searching through a sorted list which isn't significantly better. Binary search shows how to search efficiently in a sorted list. Finally, we use golden mean search to find the largest (or smallest value) if there is only a single maximum or minimum. Golden mean is one strategy used to build heuristics for more complicated, global searches for optimization.

10.2 Linear Search

```
LinearSearch(a, target)
  for i ← 0 to length(a)
    if a[i] = target
      return i
    end
  end
  return -1 // not found
end

end
```

Algorithm 10.1: LinearSearch

10.3 Binary Search

Binary search splits the list in half each time, picks an element and throws out the half of the list that can be excluded.

```
BinarySearch(a, target)
  L ← 0
  R ← a.len − 1
  while R ≥ L
    guess = (L + R) / 2
    if a[guess] > target
      R = guess − 1
    else if a[guess] < target
      L = guess + 1
    else
      return guess
    end
  end
  return −1  // not found
end
```

Algorithm 10.2: BinarySearch

The following algorithm is recursive instead of iterative.

```
BinarySearch(a, L, R, target)
  if L > R
    return −1 // not found
  end
  guess ← (L + R)/2
  if target > a[guess]
    return BinarySearch(a, guess + 1, R, target)
  else if target < a[guess]
    return BinarySearch(a, L, guess − 1, target)
  else
    return guess
  end
end
```

Algorithm 10.3: BinarySearch

One very useful test in binary search is to check whether the value is completely out of range. If many searches are for values out of the range of the array, then this can reduce complexity for those cases from $O(logn)$ to $O(1)$

```
BinarySearch(a, target)
  if target < a[0] || target > a[a.len−1]
    return −1
  end
  L ← 0
  R ← a.len − 1
  while R ≥ L
    guess = (L + R) / 2
    if a[guess] > target
```

```
      R = guess - 1
    else if a[guess] < target
      L = guess + 1
    else
      return guess
    end
  end
  return -1  // not found
end
```

Algorithm 10.4: BinarySearch

For a similar solution in continuous floating point, see the bisection algorithm for finding roots.

10.4 Golden Mean Search

Golden mean is a search algorithm to find the maximum or minimum. Assumptions:

- The function has a single global maximum or minimum extremum (so-called quadratic)
- The function cannot have any other local maxima or minima.

The search does not know what the maximum value is, so this algorithm is a little slower than binary search $O(\log_{phi} n) = O(\log n)$.

```
GoldenMeanSearch(a)
end
```

Algorithm 10.5: GoldenMeanSearch

11. Strings

11.1 Searching

The simple string search problem is to find a string text of length k within a potentially much larger string s of length n.

Brute force string search is obvious but not very good. It looks for the first letter of the target string. When it finds the desired character, it looks for the remainder of the string. On the face of it, this might seem to be $O(n)$. However, because the first character may be found, and then subsequently the match may not happen, there may have to be multiple checks, and this can result in O(kn) time.

The following example shows the string "this is a test" (n=13) being searched for the target string "test" (k=4) In this case, it seems like the complexity is O(n).

t	h	i	s		i	s		a		t	e	s	t																
t	t	t	t	t	t	t	t	t	t	t	t	t	t																

However, if the search string contains many partial matches, the algorithm must keep trying to check whether each one works. In this next case, the string "xoxoxoxoxoxoxo..." is being search for the pattern "xoxoy" The first 4 letters match, and it is only when the algorithm encounters the y that it has to start over, and since it is ignorant of the structure, it must do so on the very next letter, doing it all over again

x	o	x	o	x	o	x	o	x	o	x	o	x	o	x	o															
x	o	x	o	y																										
	x																													
		x	o	x	o	y																								

In the above case, you can see that it will compare the entire target string O(k), fail on the last letter, and do so n/2 times.

```
bfStringSearch(s, target)                                    1
  n ← length(s)                                              2
  m ← length(target)                                         3
  for i = 0 to n − m                                         4
    if s[i] = target[i]                                      5
      for j = 1 to m − 1                                     6
        if s[i+j] ≠ target[j]                                7
```

```
            goto nomatch
          end
        end
        return i
nomatch:
    end
    return -1
end
```

Algorithm 11.1: Brute Force String Search

11.2 Boyer-Moore

Boyer-Moore is a far more clever string search algorithm that uses the length of the target string to advantage. If we are looking for a string t within a much larger one

```
BoyerMoore(s, target)
    n ← length(s)
    m ← length(target)
    t ← new int[256] // create a table of offsets
    for i ← 0 to 255
        t[i] ← m
    end
    for i ← 0 to n - 1
        t[target[i]] ← m-i+1 // build the table of offsets
    end

    for i = m to n-1
        advance ← t[s[i]]
        if advance = 0
            for j = m - 1 to 0
                ...
            end
    end
    return -1
end
```

Algorithm 11.2: Boyer-Moore

```
FSM(s, fsm)
    n ← length(s)
    state ← 0       // select initial state
    for i ← 0 to n - 1
        if fsm.found[s[i]]
            return i
        state ← fsm.next[state][s[i]]
```

```
  end
end
```

Algorithm 11.3: Finite State Machine

11.3 Longest Common Subsequence and EditDistance

One important problem in string manipulation is discovering how similar two strings are when they do not match.

The Longest Common Subsequence (LCS) defines the cost to compare two strings for the maximum characters they share in common (not sequentially)

```
LCS(a, b)
  if isEmpty(a) or isEmpty(b)
    return 0
  end
  if a[0] == b[0]
    return 1 + LCS(a.substr(1), b.substr(1)) //Note: this cannot
       copy the substring or cost gets very high
  end
  return max(LCS(a,b.substr(1)), LCS(a.substr(1), b))
```

Algorithm 11.4: Longest Common Subsequence

```
LCS(a, b, m, n)
  if m = length(a) or n = length(b)
    return 0
  end
  if memo[m][n] ≠ 0
    return memo[m][n]
  end
  if a[m] == b[n]
    return 1 + LCS(a.substr(1), b.substr(1)) //Note: this cannot
       copy the substring or cost gets very high
  end
  memo[m][n] ← max(LCS(a,b.substr(1)), LCS(a.substr(1), b))
  return memo[m][n]
```

Algorithm 11.5: Longest Common Subsequence with Dynamic Programming

```
LCS(a, b)
  m ← length(a)
  n ← length(b)
  t ← new int[m+1][n+1]
  maxVal ← 0
  for i ← 0; i ≤ m; i++)
```

```
    t[i][0] = 0
  for i ← 0; i ≤ n; i++)
    t[0][i] = 0
  for i ← 1; i ≤ m; i++)
    for j ← 1; i ≤ n; j++)
      if (a[i] = b[j]
        t[i][j] ← 1 + t[i−1][j−1]
      else
        t[i][j] ← maxVal ← max(t[i−1][j], t[i][j−1])
      end
```

Algorithm 11.6: Longest Common Subsequence as a single loop

The EditDistance algorithm computes the cost to turn one string into another using three operations: inserting a letter, deleting a letter, or replacing a letter with another. The algorithm is therefore a measure of how different two strings are, contrasted with LCS which measures how much of two strings is in common. Both LCS and EditDistance offer a crude approximation of how similar two documents are to each other, but for people submitting programs as homework (such as in a data structures class) this does not measure whether two homeworks are from the same code, because some relatively simple operations can confuse these two algorithms. For example, students try to change a variable name, rearrange functions, and other trivial changes that still show the code has the same structure. The next algorithm, Winnowing is used to compare documents where someone has deliberately tried to change.

```
EditDistance(a, b, m, n) {
  if (n == s2.length())
    return s1.length() − m;
  if (m == s1.length())
    return s2.length() − n;
  if (s1[m] == s2[n]) return count(s1, s2, m + 1, n + 1);
  if (s1[m] != s2[n]) {
    return 1 + min(EditDistance(s1, s2, m, n+1), EditDistance(s1,
        s2, m+1, n), EditDistance(s1,s2,m+1,n+1));
  }
}
```

Algorithm 11.7: Edit Distance

```
int EditDistance(string s1, string s2) {
  int m = s1.length();
  int n = s2.length();
  for (int i = 0; i ≤ m; i++) {
    v[i][0] = i;
  }
  for (int j = 0; j ≤ n; j++) {
    v[0][j] = j;
  }
```

```
  for (int i = 1; i ≤ m; i++) {
    for (int j = 1; j ≤ n; j++) {
      if (s1[i-1] == s2[j-1]) v[i][j] = v[i-1][j-1];
      else v[i][j] = 1 + min(min(v[i][j-1],v[i-1][j]),v[i-1][j
          -1]);
    }
  }
  return v[m][n];
}
```

Algorithm 11.8: Edit Distance with Dynamic Programming

11.4 Plagiarism and the Winnowing Algorithm

While LCS and EditDistance are useful to compare two documents, they are not useful when a human is trying to actively change the documents to obfuscate their similarity. For example, two students attempting to cheat on a homework might literally submit the same file (it's happened) but if not completely incompetent they might make an attempt to hide the fact that the documents are the same. This can be done by rearranging sections, by adding spaces. More sophisticated modifications can involve changing variable names.

Winnowing is the central algorithm in MOSS (Measure of Software Similarity) written at Stanford. It has been used to compare student assignments at classes all over the world as well as in litigation involving one party claiming that another stole their code.

The winnowing algorithm compares blocks if they have been moved around and gives a comparison of how similar the structure of the code is. It is not perfect, and it is possible for a student to change variables and rewrite an algorithm, but frankly if you cite your friend's help, and with their help rewrite their code and get it working, that is actually legal in the course even though it still does not really show that you are able to develop on your own. At least it shows you put in significant work, and usually people who plagiarize are far too lazy for that.

The following example shows code which would be very different using the diff utility (LCS) but shows almost 100% similarity using winnowing:

```
#include <iostream>

int f(int x) {
    return x*x;
}

int g(int x) {
    return 2*x;
}

int main() {
    cout << f(2) << '\n';
    cout << g(3) << '\n';
}
```

```
#include <iostream>

int  g(int x)
{
    return 2 * x;
}

int  f(int x)
{
    return x * x;
}

int main() {
    cout << f(2) << '\n';

    cout << g(3) << '\n';
}
```

Winnowing(a,b) [TBD]

Algorithm 11.9: Winnowing

11.5 References

http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
https://en.wikipedia.org/wiki/Dynamic_programming
https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm

https://medium.com/factory-mind/
regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285

12. Backtracking

12.1 Introduction

Backtracking is a general algorithm for finding solutions to a computational problem involving permutations that builds candidates for solutions and applies a filter backtracking whenever a potential solution does not work.

Backtracking problems are typically executed on huge search problems, and there is no way to try every permutation. The most important part is to filter out large regions of the search space early, because otherwise the search is intractably large.

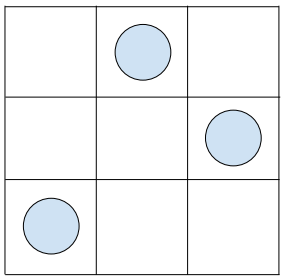
Backtracking is a recursive implementation that typically replaces n nested loops. In other words, not only is it computationally expensive, but without recursion each value of n would require custom-written code to solve.

Problems in this chapter include n-queens, magic squares, sudoku and finding all words on a game board.

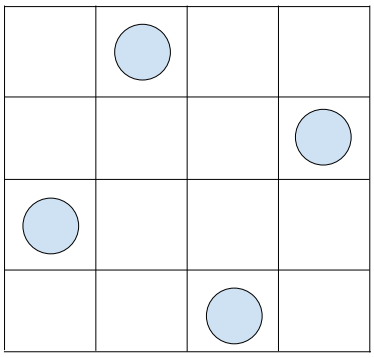
12.2 N Queens

The N queens problem is to lay out n queens on an n x n chess board in such a way that none are in position to capture each other. Queens in chess can move in straight lines horizontally, vertically and diagonally, so this means that the queens are constrained to be one per row, one per column, and then each queen calculates $row + col$ and $row - col$ and those also must not be the same.

There are no solutions for n=2 or 3.



For n=4, here is one sample solution:



A hardcoded solution for $n = 4$ is shown below. It is brute force, $O(n^n)$ with checking that is $O(n^2)$

```
for (int a = 1; a <= 4; a++)
  for (int b = 1; b <= 4; b++)
    if (abs(b-a) <= 1)
      continue;
    for (int c = 1; c <+ 4; c++)
      if (c == a || abs(c-b) <= 1)
        continue;
        for (int d = 1 ; d <= 4; d++) {
          if (d == a || d == b || abs(d-c) <= 1)
            continue;
          print solution
        }
    }
  }
}
```

12.3 Permutation Algorithms

Many backtracking problems involve permuting something, so very often the first step is to write permutations. With n nested loops each ranging from 1 to n, complexity is n^n whereas permutation is $n!$ which is significantly better. Of course, since both are gigantic, it is crucial to be able to filter out a very large percentage of execution or it will never complete.

In this section we will go over two such algorithms from Robert Sedgewick's huge collection. You can find his article from the 70s summarizing at least 30 such methods. However, as far as I know, the one that I want isn't among them. As you will see, for efficiency we want a permutation algorithm that picks a first value and then permutes everything off that.

The simplest algorithm is $2n! = O(n!)$

```
perm(n)
  if (n == 0)
     doit()
     return
  end
  for c ← 0 to n
     swap(c, n);
     perm(n−1);
     swap(c, n);
  end
end
```

Heap's algorithm (named after a person named Heap, no relation to heapsort) uses only one swap per permutation, so it is much less expensive than the first algorithm:

```
heap(n)
  if n == 0
    doit();
    return;
  end
  for i ← 0 to n
    heap(n−1);
    if n mod 2 = 0
      swap(i, n);
    else
      swap(0, n);
    end
  end
end
```

Algorithm 12.2: Heap's

12.4 Magic Squares

Magic squares are n x n grids of numbers populated with integers from 1 to n^2. The sum of each row, column and diagonals must be equal. The sum of the numbers from 1 to n^2 is:

$\Sigma_1^{n^2} i = 1 + 2 + 3 + ... + n^2 = \frac{(n^2)(n^2+1)}{2n} = \frac{n(n^2+1)}{2}$

For $n = 3$ the sum of 1 to 9 is 45. Divided by n is 15. Therefore, any row or column that is not equal to 15 can immediately terminate the solution and skip to the next one without computing the entire square.

12.5 Sudoku

Sudoku is a gigantic constraint problem. There are $9x9 = 81$ boxes on the board. Each box takes a number from 1 to 9. A brute force estimate then for $n = 9$ is 9^{81}. Of course not all these states are possible. Each row, column and box can only have each digit once. This means that at the very least, each row is 9! and since there are 9 rows, the result is $(9!)^9$. As huge as this number is, it is tiny compared to the first one.

Even the second estimate is way over, because once the first row is down, it constrains all subsequent rows, much like magic squares but larger. With the first 3 rows in place, we can check that none of the columns have the same values, and that each box has only one of each digit. The first 3 rows are $(9!)^3$ and many solutions can be rejected. For every row added after the first three, each element can only have 6 values since three are already taken in each column.

1	2	3	4	5	6	7	8	9
9	8	7	1	2	3	6	4	5
4	5	6	7	8	9	1	2	3
			1		2			
3				3				
6								
8								
9								

12.6 Boggle Problem

The game of boggle is played on an n x n board. The board is filled with randomized letters and the player must find as many words as possible. Each word begins at a location, and may continue with any adjacent letter (any of 8 directions horizontal and diagonal). Letters may not be reused in the same word. Thus for a 4x4 board, the maximum length possible would be 16 letters.

The following picture shows a sample 5 x 5 boggle board:

Starting on the top-left you can see the words ”poor”, ”word”, ”boggle”, ”search”. The word ”search” also contains ”sear” and ”ear.” A computer searching exhaustively will find hundreds of words in a typical board.

The boggle homework is to find all words on a boggle board that are in a dictionary. This can be done by backtracking through all sequences of letters on the board, rejecting any that are not in the dictionary.

For the purpose of boggle it is vastly superior to use a dictionary based on a trie than a hashmap. A hashmap dictionary can only determine whether a given word is in the dictionary or not, for example, ”CXQ” is not a word. But a trie-based dictionary can more efficiently stated that there are NO WORDS STARTING WITH ”CX” which is vastly better for terminating the search and moving on.

To compute all words on a boggle board

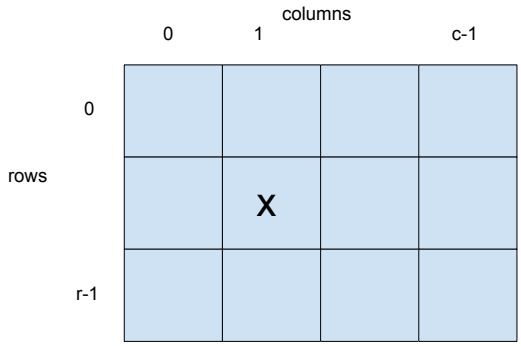
```
for each letter on board
   word = letter

  backtrack(all neighboring letters)
    wordi = word + newleter
    if NOT dict.containsPrefix(word)
      skip
    else
      if dict.contains(word)
        add to list
     end
    end
 end
```

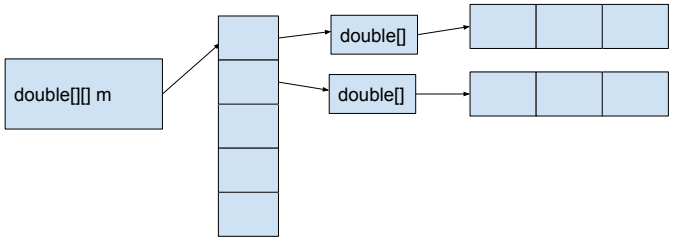
Algorithm 12.3: SolveBoggle

13. Matrices

13.1 Matrix Representations

A matrix is a two-dimensional table of numbers. The dimension of a matrix is rows and columns. A matrix can be viewed as representing a set of equations, one per row, or as a space of column vectors. An ordinary matrix is represented in one of two ways. Either as an array of rows, each containing an array of columns (Fig. 1), or as a single block of memory (Fig. 2).





The identity matrix is a square matrix n x n in which all elements are zero except the main diagonal which is 1. The following example is a 3x3 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

There are a number of matrix types where many of the elements are zero, and in such cases, it is possible to represent it in other ways that are more compact and often much faster for various algorithms.

An upper triangular matrix has all zero elements below the main diagonal (Fig. 3). A lower triangular matrix has zero elements above the main diagonal (Fig. 4). Both of these are not better in complexity than an ordinary matrix because they essentially have $n^2/2 = O(n^2)$ elements.

A diagonal matrix, however, has a big advantage, because only the elements on the main diagonal are non-zero (Fig. 5). Diagonal matrices will be able to do some operations like multiplication in $O(n)$ instead of $O(n^3)$ which is a major win. Last, a tridiagonal matrix has non-zero elements on the main diagonal, and on diagonal on either side. Tridiagonal matrices have approximately 3 times as many non-zero elements as a diagonal, but this is $3n$ so complexity is still $O(n)$.

An orthogonal matrix is one in which each column is perpendicular (normal) to the others. For example: $\begin{bmatrix} 2 & 2 \\ -2 & 2 \end{bmatrix}$

A normal vector is a vector with unit length. A normal matrix has normal vectors in all columns. $\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{\sqrt{2}}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} & 0 \end{bmatrix}$

An orthonormal matrix has columns that are all normal to each other (orthogonal) and

unit length (normal). $\begin{bmatrix} \frac{\sqrt{(2)}}{2} & \frac{\sqrt{(2)}}{2} \\ \frac{\sqrt{(-2)}}{2} - 2 & \frac{\sqrt{(2)}}{2} \end{bmatrix}$

The inverse of a matrix is a matrix such t $AA^{-1} = A^{-1}A = I$

A sparse matrix is one where most of the elements are zero.

13.2 Operations

In this chapter we will learn algorithms to implement matrix algorithms including

identity	Generate an identity matrix
Gauss-Jordan	solve a system $A\vec{x} = B$
GramSchmidt	Turn a matrix into an orthonormal one
partial pivoting	Find the largest row to do row-reduction for numerical stability
full pivoting	Use the largest element in the matrix to do row-reduction for greater numerical sta
inverse	Compute the inverse of A
dot product	$v_1 \cdot v_2 = v_{1x}v_{2x} + v_{1y}v_{2y} + v_{1z}v_{2z}$
LU Factorization	Solving same matrix against multiple constants faster
Least squares	Generalized least squares fit of any degree

13.3 Representations

A regular rectangular matrix can be stored as an array of pointers to rows, or as a single block of continuous memory. The single block is somewhat faster because in order to jump between rows, the indexed must go to the pointer for each row and find where it is, but the overhead is not serious for the actual operations because jumping between rows is also sub-optimal for sequential block memory. Computers are optimized for sequential access. So when we access row 0, col 0, row 0, col1, etc memory can keep up with the CPU. When we access row 0 col 0, then row 1, col 0, we are skipping however many elements are in a row, and this can significantly slow down the algorithm. So much so that for doing matrix multiplication, it is worth computing the transpose of the matrix, then using the transpose as a copy where each column is sequential. This is used in real libraries such as the Intel matrix library for a factor of 2-3 in speed.

An upper triangular matrix has a first row with n elements, a second row with $n-1$, and so on. The rest are zero and therefore may be ignored although this will require an if statement in the index computation.

The total number of non-zero elements is $n(n+1)/2$ including the main diagonal. Given row r and column c, The code to access a given element is:

```
get(r, c)                                                    1
  if c > r                                                   2
```

```
    return 0
  return m[r * (r+1)/2 + c -r]  %todo: one of these is wrong!
end
```

Algorithm 13.1: get

A lower triangular matrix has a first row with 1 elements, a second row with 2, and so on. The rest are zero and therefore may be ignored although this will require an if statement in the index computation.

```
get(r, c)
  if c<r
    return 0
  return m[r * (r+1)/2 + c -r]
end
```

Algorithm 13.2: get

Upper and lower triangular matrices are not that useful from a complexity point of view because they are still n/2 elements and therefore all the interested algorithms are still $O(n^3)$ although it may be lower by a factor of 8.

Diagonal and Tridiagonal Matrices are fundamentally better, and they are used in many algorithms where the problem is constrained. A diagonal matrix has non-zero elements only on the main diagonal and therefore uses storage $O(n)$.

The algorithm to access the elements of a diagonal matrix is

```
get(r, c)
  if c≠r
    return 0
  return m[r]
end
```

Algorithm 13.3: get

A tridiagonal matrix has elements on the main diagonal, and one up and down as well. It looks like this:

The logic for computing the index of a tridiagonal seems confusing since the first and last rows have 2 elements instead of 3. The easy way to think of it is to assume that all rows have 3 elements. In that case, assuming we ask for one of the elmeents that exists, the index is: $3r + c - r$ which can be simplified to $2r + c$. If we then subtract 1 we can eliminate the bogus first element. $2r + c - 1$. The algorithm to access the elements of a tridiagonal matrix is

```
get(r, c)
  if abs(r-c)>1
    return 0
  return m[2*r + c]
```

```
end
```

Algorithm 13.4: get

The total number of non-zero elements is $n(n+1)/2$ including the main diagonal. Given row r and column c, The code to access a given element is:

```
get(r, c)
  return r * (r+1)/2 + c -r
end
```

Algorithm 13.5: get

13.4 Addition

Matrix addition is a simple $O(n^2)$ operation.

$C = A + B$

adds every corresponding element of A and B and stores the result in C. The row and column sizes of A, B, and C must be the same for this to make any sense.

$$\begin{bmatrix} 0 & 1 \\ 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} + \begin{bmatrix} -1 & -2 \\ 1 & -1 \end{bmatrix}$$

13.5 Multiplication

Multiplication for a matrix of size (m,n) by a matrix of size (n,p) is $O(mnp)$. When the matrices are square with all dimensions of size n, the complexity is $O(n^3)$. The complexity of n^3 obviously gets bad fast – a mere $n = 1000$ results in 10^9 operations. Also, when dealing with matrices, the non-ideal nature of computer memory becomes apparent. RAM is designed for sequential access. It is far faster given a request for memory location i to get location $i + 1$. But part of the multiplication algorithm goes down the columns of B. In a practical library, this problem can be solved by transposing the rows and columns of B, and then performing a modified matrix multiply.

```
Mult(a, b)
  assert cols(a) = rows(b)

  for k = 0 to rows(a)-1
    for j = 0 to cols(b)-1
      ans(k,j) = 0
      for i = 0 to cols(a)-1
        ans(k, j) += a(i,j) * b(j,k)
```

```
      end
    end
  end
end
```

Algorithm 13.6: Multiplication

Faster Multiplication by storing temporary result in a scalar

```
Mult(a, b)
  assert cols(a) = rows(b)

  for k = 0 to rows(a)-1
    for j = 0 to cols(b)-1
      dot = 0
      for i = 0 to cols(a)-1
        dot += a(i,j) * b(j,k)
      end
      a(k,j) = dot
    end
  end
end
```

Algorithm 13.7: Multiplication

13.6 Gauss-Jordan Elimination

Solving a system of n equations in n unknowns can be represented as:

$A\vec{x} = B$

Row reduction is commonly taught in high school algebra, but the algorithm used will not work except in very small, very well-conditioned systems. The coefficients from a set of n equations is converted to a matrix. The element (1,1) is used to zero out the elements below it, which must also modify the entire row and the constant vector B as well.

$x + 2y - 3z = -5$
$3x - y + z = 8$
$2x + 3y + 2z = 13$

$$A = \begin{bmatrix} 1 & 2 & -3 \\ 3 & -1 & 1 \\ 2 & 3 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} -5 \\ 8 \\ 13 \end{bmatrix}$$

Row reduction uses the first row to wipe out the second. First we calculate the ratio needed to multiply the 1 to zero out the 3 below it.

$s = -3/1 = -3$

Then the entire augmented first row (including B) is multiplied by this scalar and added to the second row:

1	2	-3	x		-5
0	-7	10	y	=	23
2	3	2	z		13

The complexity of reducing a single row is $O(n)$ The complexity of reducing all rows from the first one is $n(n-1) = O(n^2)$. This example is only a 3 x 3 matrix, but in general for n x n, the next row requires $(n-1)(n-2) = O(n^2)$ also, and the sum of all the n^2 is $O(n^3$. In fact, all interesting fundamental matrix algorithms are $O(n^3)$ unless highly optimized.

```
GaussJordan(A, x, B)
  for k = 0 to rows(a)-1
    partialPivot(A, B, k); // this can be partial or full
       pivoting. See algorithms
    for j = k+1 to cols(b)-1
      s = -A(j,k) / A(k,k)
      ans(j,k) = 0              // zero out the elements below the
         pivot
      for i = j+1 to cols(a)-1
        ans(j, i) += s * A(k,i)  // do a row reduction with the
           ratio computed
      end
      B(j) += s * B(k)           // do the same thing to the
         augmented matrix including the solution
    end
  end
  backSubstitute(A, x, B)
end
```

Algorithm 13.8: Gauss-Jordan

```
partialPivot(A, B, k)
  pivotloc = k
  pivot = A(k,k)
  for i = k+1 to n-1
    if A(i,k) > pivot
      pivot = A(i,k)
      pivotloc = i
    end
  end
 if pivotLoc ≠ k
   for i = 0 to n-1
```

```
      swap(A(k,i), A(pivotLoc,i))
    end
  end
end
```

Algorithm 13.9: PartialPivot

Full pivoting not only switches rows, it selects the biggest element in a column. This means the algorithm effectively does variable substitution so for full pivoting we must store which column corresponds to which original variable

```
FullPivot(A, B, n)
[TBD]
end
```

Algorithm 13.10: FullPivot

```
backSubstitute(A, B, n)
  for k= n-1 downto 0
    B(k) /= A(k,k)
    A(k,k) = 1
    var = B(k)              // hold the current variable in a
       scalar
    for j = k-1 downto 0
      B(j) -= var * A(j,k) // subtract off the variable just
         solved for
    end
  end
end
```

Algorithm 13.11: BackSubstitute

13.7

13.8 Gram-Schmidt Orthogonalization

```
Gram-Schmidt(m)
  for i = 0 to n-1
    invmag ← mag(m[i][])
    for j = 0 to n-1            //  normalize the matrix
      m(i,j) = m(i,j) * invmag
    end
    // next, subtract the previous basis vectors
    for k = 0 to i-1
      subtract(m, i, j)
```

```
    end

  end
end
```

Algorithm 13.12: Gram-Schmidt

13.9 Least Squares

A least squares linear fit (also called linear regression) is the best line through data. In order to compute a best fit linear least squares, calculate:

$y = a + bx + \epsilon$

The variables a and b are the two parameters fit for the linear model, and ϵ is the error. In order to minimize the square of the error, the mean ϵ should be zero.

such that the constants a and b minimize the residual error when the y are compared to the actual values. For example, given the following data:

x	y
3	4
4.1	4.95
4.9	6.03
5.5	6.59
5.7	6.46

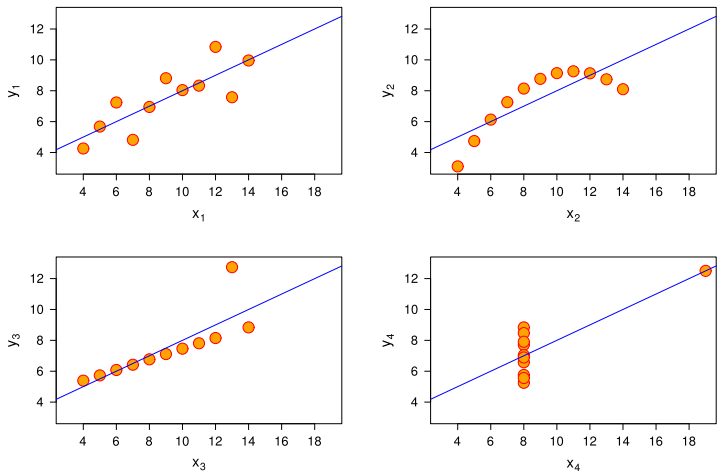
The following algorithms show the specific solution for a linear least squares fit.

Compute $\Sigma x, \Sigma y, \Sigma x^2, \Sigma y^2, \Sigma xy$
$S_x x = \Sigma(x_i - \bar{x})^2$
$S_x y = \Sigma(y_i - \bar{y})(x_i - \bar{x})$
b $\leftarrow \frac{S_{xy}}{S_{xx}}$
a $\leftarrow \bar{y} - bx$
$R^2 \leftarrow (\Sigma y_i - (a + bx))^2$

Algorithm 13.13: LeastSquares

Linear least squares is used when the data is overdetermined, when there are more data than unknowns. However, just because you can compute linear regression does not mean it makes sense. The following graph due to Anscombe (and generated in Wikimedia Commons) shows four different datasets, each with the same mean, variance, and linear regression. Clearly the top-left data set is at least possibly linear with noise. Just as clerly, the top-right data set is not a line at all. If anything, it should be fit to a quadratic or other curve. The bottom-left graph shows that one outlier point can throw the fit off, so either discount one bad point if there is some reason to believe it was an error, or come up with a different theory other than a linear fit. Last, the bottom right

shows that a relationship showing absolutely no correlation between two variables can be thrown off by one spurious data point.



The moral of the above diagram is to always graph data and look at it, not just compute a linear fit without thinking.

For a quadratic fit, we need three parameters plus the residual error. Again, do not simply fit the quadratic without knowing that it is actually sensible.

$y = a + bx + cx^2 + \epsilon$

Computing the quadratic fit requires a few more terms

$$\begin{pmatrix} \Sigma x_i^4 & \Sigma x_i^3 & \Sigma x_i^2 \\ \Sigma x_i^3 & \Sigma x_i^2 & \Sigma x_i \\ \Sigma x_i^2 & \Sigma x_i & \Sigma n \end{pmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \Sigma x_i^2 y_i \\ \Sigma x_i y_i \\ \Sigma y_i \end{bmatrix}$$

Compute $\Sigma x, \Sigma y, \Sigma x^2, \Sigma y^2, \Sigma xy$
Compute $\Sigma x^2 y$
$S_x x = \Sigma(x_i - \bar{x})^2$
$S_x y = \Sigma(y_i - \bar{y})(x_i - \bar{x})$
$c \leftarrow$
$b \leftarrow$
$a \leftarrow$
$R^2 \leftarrow (\Sigma y_i - (a + bx + cx^2))^2$

Algorithm 13.14: LeastSquaresQuadratic

The following equation is the general matrix form for any polynomial:

$A^T Ax = A^T B$

13.10 Nonlinear Least Squares Fit

Any general equation can be fit in the same way as the two examples above. In order to fit an exponential curve of the form:

$y = ae^{bx} + \epsilon$

we can take the log of both sides

$\log y = \log(a) + bx$

In other words, do a linear fit using the logarithm of the y values.

13.11 Principal Component Analysis (PCA)

Principle Component Analysis is a better version of least squares, used to analyze data sets with many variables and discover relationships hidden in the data.

Least squares is simple because it assumes that the independent variable is correct (time, or the x coordinate) and the errors are in the y. Principle Component Analysis attempts to minimize the errors without this assumption. The following diagram shows the error for a least squares fit:

13.12 Further Reading

https://mathworld.wolfram.com/LeastSquaresFittingExponential.html

14. Graph Theory

14.1 Introduction

Leonhard Euler invented the concept of graph theory and wrote what is considered the first paper on the subject in 1736.

In this chapter first the terminology of graphs is defined, then three different representations of graphs are contrasted. Each has advantages. Finally, some graph algorithms are shown. The field is large, and this is just an overview.

Algorithms include

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. Djikstra
4. Floyd-Warshall
5. Prim
6. Kruskal
7. Travelling Salesman Problem (TSP)

14.1.1 Definitions

A graph is a set V vertices and a set E edges, where each edge is a pair of vertices. Note that because it is a set, there can only be a single edge between any pair of vertices.

Two vertices of a graph are adjacent if there is an edge between them. In the following example, vertices 2 and 3 are adjacent.

A graph is connected if, it is possible to reach every vertex from every other. For example, in the following diagram graph A is connected while graph B is not.

A graph is biconnected if any single link can be removed and it is still connected. This means that every vertex is connected by two paths.

14.1.2 Representation

There are three good representations of graphs, and one bad one that is useful to understand. The first, obvious but bad representation is a set of vertices (which can be turned into an ordered list of vertices) and a list of edges which are pairs of vertices.

This representation is bad for two reasons. First, the list of vertices is useless. Given that we know that a graph has V vertices, there is no reason to store the numbers from 1 to V. The second is that by storing a single list of all edges, it is expensive to find the right one. Since the number of edges in a graph is $O(V^2)$ searching for any particular edge becomes $O(V^2)$ and $\Omega(1)$.

14.1.3 Algorithms

Depth First Search (DFS) is an algorithm for finding every connected vertex starting with a vertex V and using a simple rule: always go deeper and explore before returning, and never visit a vertex twice.

In order not to revisit a vertex despite the possibility of cycles, there must be a data structure to remember where the algorithm has visited. This is simply a vector of V booleans.

Algorithm 14.1: DFS-Recursive

Algorithm 14.2: DFS-Iterative

Algorithm 14.3: BFS-Iterative

```
function BellmanFord(list vertices, list edges, vertex source) is
    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and
       edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance ← list of size n
    predecessor ← list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do

        distance[v] ← inf           // Initialize the distance
          to all vertices to infinity
```

```
        predecessor[v] ← null        // And having a null
           predecessor

    distance[source] := 0             // The distance from the
       source to itself is, of course, zero

    // Step 2: relax edges repeatedly

    repeat V−1 times:
         for each edge (u, v) with weight w in edges do
             if distance[u] + w < distance[v] then
                 distance[v] ← distance[u] + w
                 predecessor[v] ← u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
         if distance[u] + w < distance[v] then
             error "Graph contains a negative-weight cycle"

    return distance, predecessor

\alg{Djikstra}{(V+E)logV}
function Dijkstra(Graph, source)
  Q ← \empty
  for each vertex v in Graph
  dist[v] ← ∞
  prev[v] ← null
  add v to Q
  dist[source] ← 0
  while Q is not empty
    u ← vertex in Q with min dist[u]
    Q.remove(u)
    for each neighbor v of u still in Q:
      alt ← dist[u] + length(u, v)
      if alt < dist[v]:
        dist[v] ← alt
        prev[v] ← u
      end
    end
  end
  return dist, prev
end
```

Algorithm 14.4: Bellman-Ford

Algorithm 14.5: Float-Warshall

15. Number Theoretic Algorithms

15.1 Introduction

Number theory is the branch of mathematics concerned with the properties of the integers. Number-theoretic algorithms solve problems involving the integers. In this chapter, we cover some of the key elements of number theory:

1. The classical algorithms of arithmetic, how to do addition, multiplication etc.
2. Examples of how to do arbitrary precision arithmetic in Java and C++
3. Efficient computation of Greatest Common Denominator (GCD) and Lowest Common Multiple (LCM)
4. Primality testing by trial division, and factoring numbers
5. Eratosthenes sieve
6. Prime number wheels to filter out many non-primes
7. The power and powermod algorithms
8. Testing primality without trial division: probabilistic algorithms
9. The Fermat algorithm
10. The problem with Fermat: Carmichael numbers
11. the Miller-Rabin algorithm
12. Agrawal-Kayal-Saxena: The AKS algorithm and the coming challenge to RSA
13. Asymmetric cryptography and how it works
14. The specifics of the RSA cryptosystem, and the attacks that are threatening it
15. The notion that the basis for asymmetric cryptography may not even exist!

The topics in this chapter have been selected first and foremost because they are interesting, but also because number theoretic algorithms are so important to cryptography, which is used in internet security and blockchain. Also, if you wish to enter programming competitions such as google code jam, this is one of the main areas of knowledge you need to know, as problems regularly incorporate elements from number theory.

15.2 Arbitrary Precision Arithmetic

Many of the algorithms in this chapter are commonly performed on huge numbers, far bigger than the 32 or 64-bit values that can be processed in hardware. The first consideration is the time it will take to do arbitrary precision arithmetic. With a single word, the time is $O(1)$. A slow instruction like division or modulo might take 15 clock cycles, while a fast one like addition might take only 1, but these are still constants. With arbitrary precision arithmetic however, the longer the number is, the more time it takes.

To illustrate the complexity of basic operations, we will write out the classic algorithms for some of the arithmetic operations like addition and multiplication.

Here is long addition showing each digit and the carry.

		1	1	1	
		9	8	8	9
+		7	9	9	8
	1	7	8	8	7

The above example shows 4 digit arithmetic. We humans can compute one digit at a time, so this takes us 4 units of time and the number can (as in this example) grow by 1 digit to 5. So the complexity of the operation of adding an n-digit number is $O(n)$. On the computer it is no different. However, since the computer can process 64 bits at a time, each additional step adds 18 digits or so. This is much better, but by a constant factor. To grow a number by a factor of 2 will still take twice as long asymptotically.

The next operation to consider is multiplication. By looking at the following example it should be quite clear that not only is multiplication $O(n^2)$ but the number of digits can grow by a factor of 2. This means that a sequence of multiplication can easily create numbers that are gigantic, which in turn take longer to compute.

						9	8	8	9
*						7	9	9	8
					7	9	1	1	2
				8	9	0	0	1	
			8	9	0	0	1		
		6	9	2	2	3			
	7	9	1	0	1	0	1	0	9

Last, consider exponentiation. Since exponentiation is repeated multiplication, the number of digits can double with every multiply. Consider the following huge number exponentiation operations. One is clearly much, much worse than the other:

555555555555^2

$2^{555555555555}$

The first exponentiation, while large and exceeding what can be stored in a 64-bit integer, is only approximately double the number of digits because it is squared. The second expression is gigantic, larger than the number of particles in the universe, with easily more than 3 trillion digits. Why? Because it only takes 30 multiplies for 2 to reach 1 billion, the next 30 doubles that to 18 digits, and there are more than 5.5 trillion doublings.

The complexity of a^b is $O(ab2^b)$

While we will illustrate most of the algorithms in this section with small numbers, if you wish to try them on real problems, the following examples in Java and C++ will show you how you can write multi-precision arithmetic code. In Java, the library is built into the language with the BigInteger class. In C++, gnu provides the gmp (Gnu Multi Precision) library which is not quite as easy to use but includes far more including large floating point numbers as well as integers.

Here is an example Java program that calculates $n!$ in arbitrary precision.

```
import java.util.Scanner;
import java.math.BigInteger;
public class FactorialMP {
     public static void main(String[] args) {
          BigInteger p = BigInteger.ONE;
          int n;
          Scanner s = new Scanner(System.in);
          n = s.nextInt();
          for (int i = 2; i <= n; i++) {
               p = p.multiply(new BigInteger(i+""));
          }
          System.out.println(p);
     }
}
```

Here is a C/C++ program that does the same using gmp.

```
#include "gmp.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void fact(int n){
  mpz_t p;

  mpz_init_set_ui(p,1); // p = 1
```

```
  for (int i = 1; i <= n ; ++i){
    mpz_mul_ui(p,p,i); // p = p * i
  }
  mpz_out_str(stdout,10,p); / print in base 10
  mpz_clear(p);
}

int main(){
  int n;
  cin >> n; // read in the number from keyboard
  fact(n);
  return 0;
}
```

15.3 Greatest Common Denominator and Lowest Common Multiple

The first algorithm in this chapter is GCD, greatest common denominator. It was invented by Euclid more than 2500 year ago, running on the original silicon computer – sand. GCD takes two integer parameters and computes the greatest number that divides both of them. For example:

$gcd(12,18) = 6$

For more information on Euclid or the algorithm, see:
https://en.wikipedia.org/wiki/Euclidean_algorithm

The brute force algorithm, not by Euclid, is what is taught in typical elementary schools. It is easy conceptually, and it works well for small numbers but does not scale well. We simply count from 2 to the smaller of the two numbers and try division. Whenever a number divides both, we record it. This algorithm requires trying every number up to $min(a,b)$ so for large numbers it is very slow.

Algorithm gcdBruteForce

```
gcdbruteForce(a,b)
  biggestDivisor = 1
  for i = 2 to min(a,b)
    if a mod i == 0 and b mod i == 0
       biggestDivisor = i
    end
  end
  return biggestDivisor
end
```

Slightly better, but still $O(n)$ we can start with the biggest number and count down. At least this way, the algorithm can stop as soon as the first divisor is found. Because of the condition it is $\Omega(1)$ but worst case is still the same. Try to identify under what conditions the worst case would happen.

```
gcdbruteForce(a,b)
  for i = min(a,b) to 2
    if a mod i == 0 and b mod i == 0
       return i
    end
  end
  return 1
end
```

Euclid's algorithm is much more elegant. Define the recursive relationship:

$gcd(a,b) = gcd(b,a \mod b)$

The terminating case is:

$gcd(a,0) = a$

The trick in Euclid's algorithm is that a mod b gets small very fast. Consider the case where a is big and b is small:

$gcd(1000,11)$

Because the second number is 11, the next stage is $1000 \mod 11$ so must be in $[0,10]$.

$gcd(1000,11) = gcd(11,9) = gcd(9,2) = gcd(2,1) = gcd(1,0)$

Thus the greatest common denominator is 1 because 1000 and 11 have no common factors. If the two numbers are close, for example:

$gcd(1000,999)$

Then $1000 \mod 999 = 1$, and the next stage collapses even faster. The worst case for gcd is actually the fibonacci series in reverse.

$gcd(55,34) = 21$
$gcd(34,21) = 13$
$gcd(21,13) = 8$

In this worst case, the number is decreasing by a factor of $\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$

The following shows Euclid's algorithm iteratively:

Algorithm gcd (iterative)

```
int gcd(int a, int b) {
  while (b != 0) {
    temp = a % b;
```

```
    a = b;
    b = temp;
  }
  return a;
}
```

The following is the same Euclid's algorithm but using recursion

Algorithm gcd (recursive)

```
int gcd(int a, int b) {
  if (b == 0)
    return a;
  return gcd(b, a % b);
}
```

The least common multiple algorithm is closely related to GCD. In order to find the lowest number that is a multiple of two numbers, the two numbers must be factored because the smallest number will only use the factors of each once. For example, consider:

$LCM(12,18)$

The prime factors of 12 are 2, 2, 3. The prime factors of 18 are 3, 3, 2. The factors in common are 2, 3, and the extra is 2 from the 12 and 3 from the 18. Together, $2*3*2*3 = 36$ which is the answer. Factoring would be slow, as we will see $O(\sqrt{n})$ so instead we calculate $a*b$ which uses all the factors, and then to eliminate double counting the factors in common, divide by $gcd(a,b)$.

Algorithm LCM

```
int LCM(int a, int b) {
  return a * b / gcd(a,b);
}
```

15.4 Brute Force Primality Testing by Trial Division

A prime number is a positive integer greater than 1 which is evenly divisible only by itself and 1. For example 5 is prime because the only two divisors are 5 and 1, 6 is not prime because $2*3 = 6$, 7 is prime, while 9 is not prime because $3*3 = 9$.

When the prime numbers were discovered, the method of testing for them was in accordance with the definition. If a number p is to be tested, try to divide p by every number smaller than itself, and if any divisor is found then p is not prime.

The complexity of the following algorithms is not given. Work it out for yourself as an exercise in analyzing code.

Algorithm: Primality Test Using trial Division

```
bool isPrime_v1(int n) {
  bool prime = true;
  for (int i = 2; i < n; i++) {
    if (n % i == 0)
      prime = false;
  }
  return prime;
}
```

```
bool isPrime_v2(int n) {
  for (int i = 2; i < n; i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

```
bool isPrime_v3(int n) {
  for (int i = 2; i < n/2; i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

```
bool isPrime_v4(int n) {
  if (n % 2 == 0)
    return false;
  for (int i = 3; i < n;  i += 2) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

```
bool isPrime_v5(int n) {
  for (int i = 2; i <= sqrt(n); i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

15.5 Factoring

There are better methods of primality testing than trial division. But at the moment there are no known methods of factoring a number other than trial division. See the discussion further on about RSA, and the basis behind the cryptography used to keep the internet safe. It is not at all clear that this situation will continue.

The following code prints all the factors of an integer n in pairs. For each number i up to the square root, the factors are i and n/i.

```
void factor(int n) {
  for (int i = 2; i <= sqrt(n); i++) {
    if (n % i == 0)
      cout << i << ' ' << n/i << ' ';
  }
}
```

15.6 Eratosthenes' Sieve

Eratosthenes sieve is another algorithm that is approximately 2500 years old, also implemented in the original silicon computer – sand! The procedure is to write out all the integers up to a desired maximum. Each number is initially assumed to be prime. For every number that is prime, cross off all multiples of that number. For example, starting with 2 (prime), cross off all multiples of 2 which are not prime. Then checking 3 which is prime because it was not a multiple of 2, cross off $3*2=6, 3*3=9, \ldots$

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	18	19	20	21	22	23	24	25
2	3		5	~~6~~	7		~~9~~		11	~~12~~	13		~~15~~		17	~~18~~	19	20	~~21~~	22	23	24	25

Algorithm eratosthenes

```
uint64_t eratosthenes(uint64_t n) {
  uint64_t count = 0;
  bool* isPrime = new bool[n+1]; // allocate giant array of
     boolean (better to do this with bits than bool)
  for (uint64_t i = 2; i <= n; i++)
    isPrime[i] = true;
  for (uint64_t i = 2; i <= n; i ++)
    if (isPrime[i]) {
      count++; // add the new prime and then remove all multiples
          of it starting with $i^2$
      for (uint64_t j = 2*i; j <= n; j += i)
        isPrime[j] = false;
  }
}
```

Eratosthenes' is an algorithm with amazing performance. The complexity is $O(\log\log n)$ which is hard to see from the loop. It comes from the fact that the density of prime numbers is very low ($1/\log n$). This is not apparent at small numbers, where the primes seem fairly dense (2, 3, 5, 7, 11), but at $n = 10^6$ the density means that only 1 in 20 numbers is prime, at $n = 10^9$ only 1 in 30 is prime, and at numbers of 1000 digits (10^{1000}) only one in thousands would be prime. The second log is because as the numbers increase, the increment $j+ = i$ is skipping more and more, so there is less work for each new prime.

As good as it is, the original Eratosthenes can be improved by a constant factor, but a large one. For every prime i, it is removing $2i, 3i, 4i, ...$, but this has already been done by the loop which removed the multiples of 2, and 3. In fact, the first number that is unique is i^2. For example, when considering the prime number 97 there is no reason to try to eliminate $97 * 2, 97 * 3, ...$ the first number that must be crossed off is $97 * 97 = 9409$. In the improved Eratosthenes, we handle 2 as a special case since it is the only even prime. Then all further primes are odd. That means that for any prime i found, i^2 is also odd. So when cancelling multiples, it is not necessary to set the number $i^2 + i$ to be false, because $odd + odd = even$. Instead of adding i each time in the inner loop we can add $2i$ which effectively halves the time. This is only a constant factor reduction, the algorithm is still $O(\log\log n)$.

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	
3	5	7	~~9~~	11	13	~~15~~	17	19	~~21~~	23	25	~~27~~	29	31	~~33~~	35	37	
3	5	7		11	13		17	19		23	~~25~~		29	31		~~35~~	37	

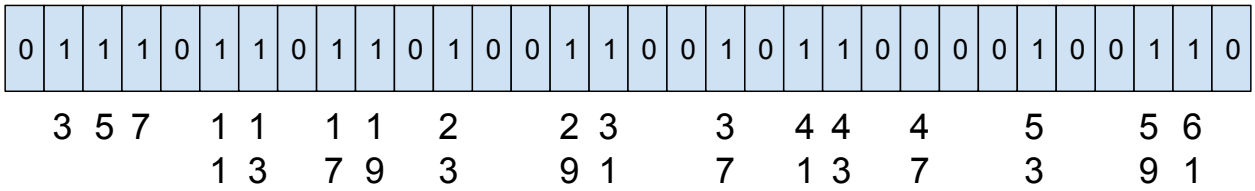
Algorithm improvedEratosthenes

```
uint64_t improvedEratosthenes(uint64_t n) {
  uint64_t count = 1; // special case for 2 which is even
  bool* isPrime = new bool[n+1]; // allocate giant array of
     boolean (better to do this with bits than bool)
  for (uint64_t i = 3; i <= n; i += 2)
    isPrime[i] = true;
  for (uint64_t i = 3; i <= n; i += 2)
    if (isPrime[i]) {
      count++; // add new prime
      // remove all multiples starting with i^2
      for (uint64_t j = i*i; j <= n; j += 2*i)
        isPrime[j] = false;
  }
}
```

One final performance optimization, left to the reader, is to implement the array of booleans more efficiently. The code shown here stores each bool as a byte. But a boolean requires only one bit. This means that for computing the primes up to 1 billion, instead of needing 1Gbyte, only 125Mb is required. Further optimizations are possible. We can

store only the odd numbers since the only even prime is 2, handled as a special case. This brings the storage down to 62.5Mb. The fact that computers process 32 or 64 bits means that once a bit vector is used, it is possible to set multiple bits true at the same time. On a PC with a 64-bit CPU, this means that it is possible to set 64 elements of the bit vector true in a single operations, with vastly improved performance.

The following diagram shows a bit vector on a 32-bit machine which represents the odd primes in the range 1 to 63. The next word would represent primes from 65 to 127.



Going further, even 62.5Mb is a fairly large block of memory, so computers will not have enough cache to hold it all. It is therefore more efficient to write not the whole thing, but a range that efficiently fits into cache. Working on a range at a time is called a segmented Eratosthenes' sieve, and is the fastest implementation in this class. Of course it is also possible to do the computation on a GPU with thousands of processors in parallel, but that is an implementation that is beyond the scope of this course.

15.7 Prime Number Wheel

Prime number wheels are not in themselves a method of finding primes, but they are a method of skipping tests and making searching for multiple primes faster.

The first level wheel considers numbers modulo the first prime, 2. Since all numbers are either odd or even, obviously there is no point in checking even numbers (aside from 2). Therefore, any prime search can handle 2 as a special case, and the loop can process only odd numbers.

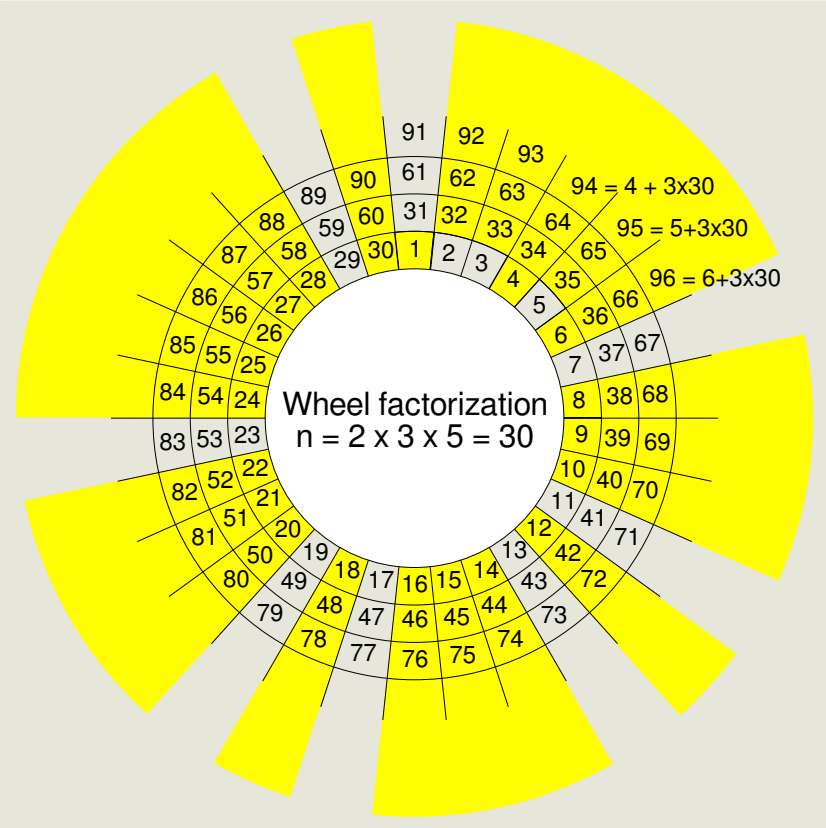
The next level of wheel considers multiples of 2 and 3. Multiplied together, $2 * 3 = 6$ there are 6 possibilities for any number $n \mod 6$. All entries in the following table are marked Composite if they are definitely not prime. Of the 6 possibilities, 3 are even, and 2 are multiples of 3. There are only 2 numbers which may be prime and must be checked.

0	Composite (2,3)
1	possible prime
2	Composite (2)
3	Composite (3)
4	Composite (2)
5	possible prime

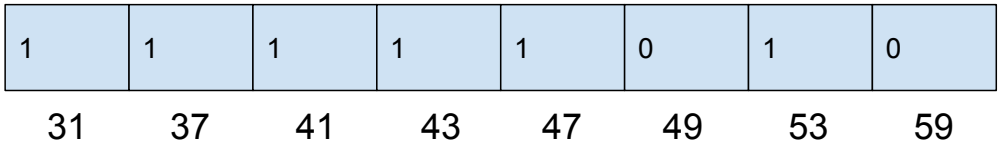
Note that this does not mean that every number $n \mod 6 = 1$ is prime. For example, 7, 13, 19 are prime, but the next one 25 is not because is it 5^2. Similarly, $n \mod 6 = 5$ may be prime, and 5, 11, 17, 23, 29 are prime, but 35 is not.

The following diagram from Wikipedia shows a prime number wheel for $n = 2,3,5 = 30$. Only 8 of every 30 numbers must be checked, which is a little over 1 in 4. For example, from 30 to 59, only the numbers 31, 37, 41, 43, 47, 49, 53 and 59 must be checked.

Taking this further is difficult because the wheels grow exponentially. Including 7 yields a wheel of size 210. Adding 11 increases the size to 2310, and adding 13 increases to 30030. And the gains come slowly. In order to eliminate 99% of the numbers, it would be necessary to build a wheel of all the numbers up to 256 which would be gigantic.



Combining segmented Eratosthenes with a bitwise representation and a wheel is particularly efficient. The following diagram shows that the 8 possible primes in each 30 numbers can be represented as a single byte with each bit true if that number is prime, implementing the wheel shown above:



15.8 Probabilistic Algorithms

Eratosthenes' sieve is the most efficient algorithm for finding large batches of primes. However, while it avoids divisions, it still requires lots of memory and $O(n \log \log n)$ computation. It would not be practical to use for a single large number, because it would require finding all primes up to the square root of that number. And for huge numbers with hundreds of digits, even the square root is more memory than is available.

For large numbers, it is possible to efficiently tell whether they are prime or not without trial division at all, which is startling. Interestingly, this is not some recent development in computer science, but was discovered by Fermat more than 300 years ago.

Fermat is famous for his "last theorem" in which he famously said he had come up with a marvelous proof but it was too large to write down in the margin. Today, we think that he was mistaken. The only proof mathematicians have come up with since was assisted by a computer and involved 6000 cases and a gigantic proof hundreds of pages long. I wonder whether Fermat was just torturing other mathematicians by claiming to know an answer.

In any case, Fermat's last theorem, that $c^n = a^n + b^n$ is false for any $n > 3$ has no practical application that we know of yet. But his little theorem is incredibly useful.

The little theorem states that if p is prime, then for any witness a where $1 < a < p$ that $a^{p-1} mod p = 1$.

On the surface, this seems useless. For a large number like 124182512581210000001000000007 even the smallest witness ($a = 2$) would result in the operation $a^{124182512581210000001000000006}$ which is utterly beyond our power to compute. The number of digits is too large to store, requiring more memory than the estimated number of particles in the universe which is a mere 10^{72} or so. But there is a hidden catch. We are not asked to compute this gigantic number. Rather, we are asked to compute it mod n. Modulo is a way to keep numbers small. Consider factorial. If we ask you to compute:

$5! = 5 * 4 * 3 * 2 * 1$

you can readily do it (120). But if we ask for $n = 10^{12}$ then $n!$ is again utterly impossible, huge! Yet what if we ask not for n factorial, but

$n!$ mod 10

Now we only need the last digit, which is easy – it is zero. How do we know? Well, $5! = 120$ the last digit is zero, and any factorial after that must end in zero because zero times anything is zero. In fact, as the numbers in $n!$ keep multiplying, for every 2 and 5, there will be another zero. Consider:

n	$n!$
2	2
3	6
4	24
5	120
...	...
10	3628800
...	...
15	1307674368000

The 10 obviously adds a zero, and the 2 in the 12 and the 5 in the 15 combine to make another 0 at 15. So for $n = 10^{12}$ not only is the last digit 0, at least the last 200 billion digits are zero.

In any case, this shows that computing something mod n is way easier than computing the full number. Using the fact that:

$(a * b) \mod n = ((a \mod n) * (b \mod n)) \mod n$ we can keep the number limited and tractable at all times. It only remains to devise an efficient algorithm for computing powers.

Obviously the definition of a^b is to multiply a together b times. This is horrendously inefficient. Instead, consider breaking down the exponent by powers of 2. Suppose we have a^{17}. That can be written:

$a^{17} = a^1 * a^{16}$

$a^{16} = a^{8^2}$

$a^8 = a^{4^2}$

$a^4 = a^{2^2}$

$a^2 = a * a$

So starting from a, repeatedly squaring we can obtain a^2, a^4, a^8, a^{16} in just 4 multiplies. We must multiply two of them together. The power algorithm breaks the exponent down by bits, and for every 1 bit multiplies into prod.

Algorithm power

```
uint64_t power(uint64_t x, uint64_t n,uint64_t  m) {
  uint64_t prod = 1;
  while (n > 0) {
    if (n & 1) { // if n is odd, then 1 bit is set
```

```
      prod = prod * x;
    x = x * x;
    n = n / 2; // repeatedly divide by 2
  }
  return prod;
}
```

For example, consider power(2, 13) which constructs $2^{13} = 2^1 * 2^4 * 2^8$

prod	x	n	comment
2	2	13	17 is odd, multiply (prod=2)
1	4	6	6 is even, no multiply
32	16	3	3 is odd, multiply
8192	256	1	1 is odd, multiply

The powermod algorithm is similar, but at each step the number must be limited by computing it modulo m.

Algorithm powerMod

```
uint64_t powerMod(uint64_t x, uint64_t n,uint64_t  m) {
  uint64_t prod = 1;
  while (n > 0) {
    if (n & 1) { // if n is odd, then 1 bit is set
      prod = prod * x % m; // compute $prod^x mod n$
    x = x * x % m;
    n >>= 1; // divide by 2
  }
  return prod;
}
```

Example, compute $powermod(2,16,17)$

prod	x	n	
1	2	16	16 is even, no multiply
1	4	8	8 is even, no multiply
1	16	4	4 is even, no multiply
1	1	2	$256 \mod 17 = 1$
1	1	1	$2^{16} \mod 17 = 1$

Example, compute $powermod(2,40,41)$

prod	x	n	
1	2	40	40 is even, no multiply
1	4	20	20 is even, no multiply
1	16	10	10 is even, no multiply
10	10	5	$16^2 \mod 41 = 10$
10	18	2	$10^2 \mod 41 = 18$
1	37	1	$18^2 \mod 41 = 18$

The final result of 1 is because $(10 * 37) \mod 41 = 1$. Notice that in the above implementation, the largest number that can be handled is $2^{32} - 1$ because when squared, anything bigger will overflow. So in order to process larger numbers, there must be a way to perform arithmetic on entities larger then 64 bits.

Fermat's algorithm is probabilistic. For a random witness a, $1 < a < p$, it may state that p is prime. But if we try a few, one of them will probably work. And we can easily make the probability that the algorithm fails smaller than the probability that the hardware will fail. At that point, the algorithm is no more "probabilistic" than the computer.

If $powermod(a, p-1, p) \neq 1$, then the number is definitely composite. if the result is 1, then it may be false witness, which is why we must try several. But there is a false positive, rare but it exists.

For a rare class of numbers called Carmichael numbers, Fermat will always return true even though the number is not prime. A Carmichael number is the product of at least 3 primes so it is definitely composite. It is square free, so no Carmichael number has two factors which are the same. The first Carmichael number is 561. See Wikipedia: Carmichael Numbers

There are infinitely many Carmichael numbers, and any of them will return true for all witnesses UNLESS you happen to pick one of the factors of the Carmichael number. For example, for 561, powermod(2, 560, 561) will return 1, thus claiming that it is prime. Any witness to powermod will also return 1 except for the factors of 561 which are 3, 11, 17.

So for a Carmichael number, Fermat requires that we know the factors, in other words that we have to try every possible factor which is what we are trying to avoid in the first place. One approach to solve this problem is to ignore it. There are 3 Carmichael numbers under 2000, a bad failure ratio. But we would never use this algorithm for small numbers. The Wikipedia article states that for numbers under 10^{21} there are approximately 20 million Carmichael numbers. So the chances of picking one are 1 in $5x10^{12}$ or so. And as the numbers grow, the probability goes down exponentially. Still, if the small chance means that your web session can be hijacked and your money stolen from the bank, it's a risk we don't wish to take. After Fermat, the next algorithm Miller Rabin solves the problem by splitting off the case of Carmichael numbers and detecting it.

Algorithm Fermat

```
bool fermat(uint64_t p, uint32_t k) {
  for (int i = 0; i < k; i++) {
    uint64_t a = rand(2, p-1);
```

```
    if (powerMod(a, p-1, p) != 1)
      return false; // definitely not  prime
  }
 return true; // if all tests passed, probably prime
```

15.9 Miller-Rabin and Solovay-Strassen

The Miller-Rabin and Solovay-Strassen algorithms are also probabilistic, so they don't give a definite answer, but can give one so close to definite that it is irrelevant. More important, they solve the problem of Fermat, there are no false positives due to Carmichael numbers.

The Miller-Rabin algorithm is based on the observation that the witness a is a number that can be decomposed into leading digits, and a number of trailing zeros. For example, the number 2 in binary is 10, the leading digit is therefore 1 and there is 1 trailing zero. The number 220 is written 11011100, which is 110111 with 2 trailing zeros.

This number is called d in the pseudocode below. There are probably many proofs, but the one shown in Wikipedia is as follows:

$a^{2^s d} \equiv 1 (mod n)$

but if the number is prime, then one of two things are possible. The underlying Fermat states that if $x^a mod n$ is 1, then the number is probably prime. But there are two ways at arriving at that 1. The number can be prime, or it can be a Carmichael number. Given the number d which is the leading digits of $n-1$, if that number is either 1 or -1 then the number is probably prime and we conduct k tests. Note that we will be repeatedly squaring x, and that $-1^2 = 1$. If the number is not 1 or -1, it may yet be prime. The algorithm keeps squaring:

$x \leftarrow x^2 mod m$

The number of times the loop executes is the number of trailing zeros. If the answer is -1, then the number again is probably prime and the algorithm can stop.

However, if at any stage x becomes 1, that is a Carmichael number and Miller-Rabin returns true. At the end of s iterations, if it has never reached -1 then it is definitely composite (not prime).

Algorithm MillerRabin $O(k \log n)$

```
MillerRabin(n, k)
  d ← n-1
  s ← number of trailing zeros of d
  d ← leading non-zero digits
WitnessLoop:
  for i ← 1 to k
```

```
    a ← random(2, n − 2)                                                    7
    x ← a^d mod n                                                           8
    if x = 1 or x = n − 1 then                                              9
      continue WitnessLoop                                                  10
   repeat s − 1 times:                                                      11
      x ← x^2 mod n                                                         12
      if x = n − 1 then                                                     13
        continue WitnessLoop                                                14
      if x = 1 then                                                         15
        return false     // this is the Carmichael case                     16
      end                                                                   17
    return false // not prime if it doesn't find a −1 along the way         18
  end                                                                       19
return true //(probably prime)                                             20
```

Algorithm 15.1: MillerRabin

Another competitive algorithm for primality testing is Solovay-Strassen, not currently added to the course materials, and not in the course yet.

Algorithm Solovay-Strassen [TBD]

```
SolovayStrassen(n, k)                                                       1
```

Algorithm 15.2: SolovayStrassen

15.10 RSA Public Key Cryptography

RSA is the Rivest, Shamir, and Adelman cryptosystem used to secure the internet. It relies on the fact that with a computer using a fast primality test like Miller-Rabin or Solovay-Strassen, it is relatively fast to pick two large prime numbers and multiply them together, but very difficult to take the resulting number and factor it back into the primes. In other words, RSA relies on a function that is relatively inexpensive (multiplication), and its inverse which is extremely slow (factoring).

RSA Cryptosystem

In order to generate an RSA key, two large random primes must be found. It cannot be predictable, so the random number generator must be cryptographically secure. For example, it cannot be based on the time the password was generated because an attacker can potentially find out that information and use it to recreate the password. When generating a random number, what are the odds that it is prime? Obviously, except for 2, any even number is not prime, so there is no point picking a number with the last bit 0. Even with that, the density of prime numbers is $1/log(n)$ so that at 10^9, approximately 1 in every 30 numbers is prime, while at 10^100 approximately 1 in every 300 numbers is prime.

Presumably, the prime number wheel concept can be used to filter out numbers that are obviously not prime. There is no point in checking whether a multiple of 2,3,5 or 7 is prime, since it obviously can't be. I have not learned the practical algorithms for selecting a random prime, but you can find the source code in all open source toolkits. For example Java provides the method nextProbablePrime in BigInteger.

```
RSAGenerateKey(k)
  p ← randomPrime(k) // generate two random keys of k bits
  q ← randomPrime(k)
  n ← p * q
  select e such that 1 < e < λ(n)
  d · e = 1( mod ψ(n))
end
```

Algorithm 15.3: Algorithm RSAgenerateKey

To encrypt a number using RSA, use the encrypt key (e,n). Note that knowing that message m can be used to determine the key, so RSA cannot be used to encrypt messages, but must instead be used to encrypt random numbers which are then used to encrypt the message.

```
RSAencrypt(m, e, n)
  return c ← m^e mod n
```

Algorithm 15.4: RSAencrypt

Algorithm RSAdecrypt

```
RSAdecrypt(c, d, n)
  return ← c^d mod n
```

Algorithm 15.5: RSAencrypt

Notice that encryption and decryption are inverses. $E(D(m)) = D(E(m) = m$.

One critical part of cryptography is determining authenticity of a message. Alice sends a message to Bob. How is Bob to know that the message is right. For example, Alice sends "Attack at Dawn" to Bob, but Mary intercepts and changes the message to "Surrender at Dawn." Public key cryptosystems support the operation digital signature to solve this problem.

Digital signature uses the fact that everyone can know the public key, but only the signer knows the private key. A secure hash algorithm is used to turn the message m into a number. Obviously, a big message cannot be reduced to a number with a small number of bits like 512. When we do this, by definition there will be collisions – multiple messages for which the hash value is the same. A cryptographic hash algorithm has the property that it is very difficult to find a message m_2 such that $hash(m) = hash(m_2)$. It's not that it doesn't exist, it simply is hard to compute. Likewise, it's vital that the attacker is not able to craft a second message that is sensible

Since only the true author can decrypt a message, in this case, the true author cryptographically hashes the message, and then decrypts the number. The resulting number makes no sense, and no one else knows how it is computed. Algorithm digitalSign

```
digitalSign(m, k_{priv})
  return D(hash(m), k_{priv})
```

To verify the signature, the recipient has to take the public key of the sender, which must be known, encrypts it, and computes the hash themselves. if the messages match, then the message is valid. Note that if the attacker could construct a second message for which the hash matches, then a message can be forged, but not only is it computationally expensive, but it would be very hard to change a real message "attack at dawn" to a meaningful message "surrender at dawn." Still, as computation improves and weaknesses in the secure hash algorithms are found there is always a risk that a message can be forged.

Algorithm verifySignature

```
verifySignature(m, h, k_{pub})
  return E(hash(m), k_{pub}) == h
```

The original paper on asymmetric cryptography was by Diffie and Hellman. For a long time, it has been known how to do cryptography with a shared secret. The problem is how to distribute the secret key to all parties where the channel is insecure (like the internet). The Diffie-Hellman paper described how two parties could exchange a secret. The following algorithm shows how this is implemented today in RSA using AES-256 as the algorithm for symmetrically encrypting the message once the keys are found. In order to make it more difficult, a new key is chosen once every 30 minutes or so.

Algorithm DiffieHellman

```
DiffieHellman()
  r = each party picks random number
  encrypt r using other party's public key and send
  sessionKey = r, other key
  encrypt session using symmetric AES256 key, used by both sides
```

15.11 Do one-way Functions Exist?

The question today is whether the entire edifice of web-based security built on RSA is secure. And if it is secure, for how much longer? IBM has warned in 2019 that they believed RSA will be insecure within 5 years. Quantum computers with sufficient qubits can break RSA in polynomial time using Schorr's algorithm (see the reference section for the paper).

Agrawal, Kayal and Saxena (AKS2002, also in the reference section) proved that it is possible to deterministically prove whether a number is prime or not in polynomial time. The fact that their algorithm is slower than Miller-Rabin and is not used in practice is immaterial. The fact that we can know deterministically whether a number is prime or not without trial division opens the question to whether factoring might also be similarly vulnerable. No one has proved it can not be done.

Algorithm AKS

```
AKS(p, k)
  TBD
```

Boaz Barak, a cryptography professor at Harvard states that there is a much more fundamental problem with the basis for all public key cryptography. It has never been proven that one-way functions exist (see his paper in the references). At the moment, there are a number of problems such as discrete logarithm, elliptic curve cryptography (ECC), and NTRU lattice methods that all provide operations that are fast in one direction and slow in another. The cryptography for each of these methods is based on the fact that a key can be generated quickly but it would take essentially forever to crack it. Lattice methods are specifically supposed to be safe against an attack by quantum computers. However, professor Barak states in his article that it has never been proven that one-way functions exist at all. Perhaps we just don't know how to invert these problems efficiently, but we might learn. The stakes are high. Anyone cracking methods that are used for finance could wreak havoc on web transactions, breaking everyone's secrets and stealing money.

15.12 Exercises

Identify the errors in the following algorithms

```
int gcd(int a, int b) {
  while (b != 0) {
    a = temp % b;
    temp = b;
    b = temp;
  }
  return a;
}
```

```
int gcd(int a, int b) {
  if (b > a) {
    swap(a,b);
  }
  return gcd(b, a % b);
}
```

The following brute force prime number solver is wrong. Identify the errors and determine the complexity of what is actually happening. Then fix it.

```
bool is_prime(long p) {
  for (long i = 1; i <= p; i++)
    if (p % i == 0)
      return false;
    else
      return true;
}
```

Implement Eratosthenes' sieve using a bit vector that stores only odd numbers. The input of the program is from standard in, a single number n that gives the upper bound. For example given the input n=1000000000 compute the number of primes up to 1 billion.

As a more difficult variant of Eratosthenes' try a program with two very large integers a and b which may be 64 bit numbers up to 10^16. With an input of:

60000000000 61000000000

your program should be able to display the number of primes between 60 billion and 61 billion. This requires a sieve that is not up to 61 billion, but only up to the square root of 61 billion which is more tractable. Once you have all the booleans identifying the primes, you can build a second table from 60 billion to 61 billion, only dividing by the numbers which are prime in the first. The above example would require less than 1 million elements in the first table since $10^{6^2} = 10^{12}$ bigger than anything we have here. The second table would have to store only odd numbers between 60 and 61 billion, which is 500 million bits or 62.5 million bytes, well within the ability of any laptop.

Implement Miller-Rabin using big arithmetic. The input should consist of a single integer in standard input, the program should print true if it is prime, or false if not.

15.13 References

Solovay-Strassen (wikipedia) https://en.wikipedia.org/wiki/AKS_primality_test

16. Floating Point

16.1 Introduction

Floating point is the data type used to perform calculations with decimal numbers. Floating point is an approximation to the reals, but since it has finite accuracy, there are properties that make it distinctly different. On a computer integers behave much like their mathematical theory basis, but with a limited range. But a floating point number is much stranger, as you will see below.

16.2 Properties of Floating Point

Floating point numbers are approximations to the reals. A floating point number has a sign bit (if 1, the number is negative), an exponent controlling the position of the "binary" point (not the decimal point), and a mantissa which contains the bits of the number. The exponent controls where the binary point is. The size of the exponent defines the range of the numbers possible.

16.2.1 IEEE-754

The IEEE-754 standard defines the different sizes of floating point. The first two were single precision (float in C++ and Java) and double precision (double in C++ and Java). The newer standard supports half precision (16 bits) and quad precision (128 bits). The half precision is used for neural networks where accuracy is not as important, and quad precision is for ultra-accurate computation such as navigation of spacecraft over 20 years. Floating point units in current processors typically support single and double precision. There is a software library supporting quad precision: https://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format.

The following table shows the IEEE-754 standard for single, double and quad precision.

C++ type	bits	exponent	mantissa	digits	minval	maxval
float	32	8	24	7	$1.23e-38$	$1.23e+38$
double	64	11	53	15	$1.23e-308$	$1.23e+308$
???	128	15	113	34	$6.4751751e-4966$	$1.23e+4966$

16.2.2 Roundoff Error

Floating point uses binary to represent numbers which are exact powers of 2. The whole numbers are just like integers. For example:

$101 = 4 + 1 = 5$ $1101 = 8 + 4 + 1 = 13$

For numbers to the right of the decimal (really binary) point, the bits are negative powers of 2: 1/2, 1/4 etc. For example:

$.1 = \frac{1}{2}$
$.01 = \frac{1}{4}$
$1.11 = 1 + \frac{1}{2} + \frac{1}{4}$
Except for multiples of powers of 2, floating point is not exact. In decimal, fractions that are not divisible by 10 are not exactly representable, like $1/3 \approx 0.3333333$. Because the fractions in floating point are expressed in binary, fractions such as $1/5 = 0.2$ are not exactly representable because 5 is not a power of 2.

For example, the output of the following program shows the errors building up as the inexact decimal 0.1 is repeatedly added.

```
#include <iostream>
using namespace std;

int main() {
  for (float x = 0.0f; x < 10; x += 0.1f)
    cout << x;
  }
  return 0;
}
```

16.2.3 Associativity breaks down

Because there is a finite accuracy, and numbers can be both very large ($1.234567e + 10$) and small ($1.234e - 33$), operations such as add do not result in the full accuracy from both numbers. Therefore the order of operations will change the answer. The associative property does not hold in floating point: $a + b + c \neq a + (b + c)$

Here is an example of a hypothetical 3-digit computer adding from large to small

	1.23
+	.899
=	2.12
+	.478
=	2.59

The same numbers in a different order

	.899
+	.478
=	1.377
+	1.23
=	2.60

The second result is more accurate because adding the small numbers keeps more information from the lower digits, and results in a higher sum. From this you can see that the result is off by one in the second digit just after a single operation.

16.2.4 Resolution is not Uniform

Floating point has a constant number of digits. Single precision. for example, has 7 digits accuracy. The distance between 1.0 and 1.0000001 is not the same as $1.0e+10$ and $1.0000001e+10$. This implies that floating point error is always relative to the size of the number.

16.2.5 Subtractive Cancellation

When two numbers are known to a high accuracy and are nearly equal, subtracting them can, in a single step, cancel almost all the accuracy.

For example: $t1 = 74.00001, t2 = 74.00000\ t1 - t0 = 1e-5$

The answer has only a single digit accuracy because there is no information about the digits coming after.

Note that with subtractive calculation, it does not matter whether the hardware has more digits of precision. The problem is that we lack the information of what those digits should be.

16.2.6 Machine epsilon

Given the finite resolution of floating point, there must exist a value epsilon such that:

$x + \epsilon == x$

In floating point when the machine epsilon is mentioned, this is normalized for x = 1. The epsilon for any given floating point type can be found by successively dividing by 2 and adding until the two are no longer distinguishable.

```
float eps;
for (eps = 1; x + eps > x; eps *= 0.5)
  ;
eps *= 2; // smallest value that can be added to 1
cout << eps << '\n';
```

16.2.7 Special Floating Point Values INF and NAN

The set of real numbers includes positive and negative infinity. With integers, a divide by zero causes a hardware trap which kills the program if it is not caught. This is not very good behavior in code which is running a guidance system for an airplane or UAV. Since divide by zero is always a possibility, the IEEE floating point standard defines special values +inf and -inf to represent infinity.

For example:

```
  double x = 1.0 / 0.0;
```

The value of x is positive infinity.

The concept of infinity was studied by Kantor. Countable infinity, the infinity for the set of integers represents a number that is not measurable. It is not possible to compare infinities. The IEEE floating point standard specifies a value representing infinity. Infinity is the result of a computation that is too large to be represented in the floating point type. For example, given that single precision can represent numbers up to 10^38 or so, computing:

```
float x = 1e30;
cout << x * x << '\n';
```

will result in infinity. Once at infinity, all arithmetic results in infinity. There is no difference between infinity, infinity+1, infinity*2. Infinity is uncountable. However, when faced with a fight between competing infinite forces, the result may not be infinity. For example, we can construct positive and negative infinity in the following way:

```
double x = 1.0 / 0.0; // positive infinity
double y = -1.0 / 0.0; // negative infinity
```

If we then add positive and negative infinity, since each is uncountable, the answer is unknown (not zero!) Whenever the answer is unknown the IEEE floating point standard uses another special definition, NaN (Not A Number). Thus for example the following are all NaN because the answer is unknown:

```
double a = 0.0;
double x = 0.0 / a; // NaN
double y = sin(1.0 / a); // NaN: we don't know where between -1
   and 1
double z = sqrt(-2.0 / a); NaN: square root of a negative number
```

16.2.8 Floating Point Coding Patterns

Because of the properties of floating point, certain actions are not possible or extremely unlikely to work. for example, comparing to a specific value is never a good idea:

```
double x = 1;
for (int i = 0; i < 10; i++)
  x /= 3;
x *= 3 * 3 * 3 * 3 * 3  *  3 * 3 * 3 * 3 * 3;
if (x == 1) {
 // this could be true, but it's not likely!
}
```

Instead, whenever comparing floating point numbers, use an approximate measure:

```
if (abs(x - 1) < .0000001) { // approximately equal
}
```

The following code is wrong. Can you identify the reason? Run or step through in the debugger if you cannot.

```
for (float x = 1e8; x < 1.0001e8; x++)
  cout << x;
```

16.3 Further Reading

The IEEE standard is in the ref directory: https://drive.google.com/file/d/1CT_AumUC4iHwdXRIJ9Lx4fwhKDEhrfVx/view?usp=sharing

For high-precision geometric computation, arbitrary-precision arithmetic is sometimes necessary to compute perfect answers. For example, given a set of polygons that divide a plane, a point containment test may compute that a point is not in any polygon even though this is not possible. The point must be on one side of a line or the other, or on the line, but roundoff error can result in points not showing in either polygon.

A paper by Jonathan Schewchuk shows how to use high precision mathematics to resolve problems.

The following calculator converts between floating point and the internal bit representation https://www.h-schmidt.net/FloatConverter/IEEE754.html

https://en.wikipedia.org/wiki/Georg_Cantor

17. Numerical Algorithms

17.1 Root Finding

For some equations there is an analytical equation for finding the root, but not for all. For some equations, writing a program to find the root to a desired precision is the only approach that works. There are three methods discussed here. First, bisection which is simple and guaranteed given an initial condition, but also slow. Then there is NewtonRaphson that is fast and dangerous, and finally the method recommended in numerical recipes which uses an exponential approximation and is quite robust and fast.

For an example, take an easy function such as:

$f(x) = x^2 - 3$

The function is by inspection, obviously zero at $x = \pm\sqrt{3}$

17.1.1 Bisection

If a function is continuous, and on an interval $[a,b]$, $f(a) < 0$ and $f(b) > 0$ then somewhere in that interval the function must cross zero.

This is the principle behind bisection, which works by divide and conquer, to keep dividing the interval in half each time. Given the initial brackets are in the neighborhood of the root, every iteration gives one more bit accuracy, so for double precision, 53 iterations should yield maximum accuracy.

Starting bisection on the interval [0,3] with $eps = .1$

a	b	guess	abs(b-a)
0	3	1.5	3
1.5	3	2.25	1.5
1.5	2.25	1.875	0.75
1.5	1.875	1.6875	0.375
1.6875	1.875	1.78125	0.1875
1.6875	1.78125	1.7344	0.080566

The algorithm for bisection can be expressed recursively, but is quite clear iteratively as shown below.

```
Bisection(f, a, b, ε)
  do
    x ← (a+b)/2
    y ← f(x)
    if y < 0
      a = x
    else y > 0
      b ← x
    else
      return x
    end
  while abs(b−a) < ε
end
```

Algorithm 17.1: Bisection

17.1.2 Newton-Raphson

Newton's method requires that you have the analytical derivative. If you do, this method certainly converges quickly. It is confusingly called quadratic, meaning that each iteration doubles the number of digits accuracy.

Newton requires an initial guess. If this guess is outside the zone of convergence, then the solution will diverge instead of converging. But within the zone, the number of correct bits in the answer will double each time.

Starting with an initial guess x_0 compute each guess as $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

Example: Given $f(x) = x^2 - 2$ and $f'(x) = 2x$ with initial guess $x_0 = 3$ show Newton-Raphson converging on the root.

$x_0 = 3$	
$x_1 = x_0 - \frac{3^2-2}{2*3} = \frac{7}{6} = 3 - 1.16666$	1.833333
$x_2 = x_1 - \frac{x_1^2-2}{2*x_1}$	1.440476
$x_3 = x_2 - \frac{x_2^2-2}{2*x_2}$	1.414529

The third approximation is already accurate to 4 significant digits.

Newton can fail spectacularly. For the above function, given the initial guess $x_0 = 0$ the derivative is zero, leading to the first approximation $x_1 = \infty$.

It is also possible to be outside the radius of convergence. In this case, instead of converging on a solution, Newton iterates away from the solution.

For example, consider the function $f(x) = (x-2)^2 - 1$ with derivative $f'(x) = 2x - 2$

If we start with $x_0 = 3$, then $x_1 = 3 - \frac{f(3}{f'(3)} = 3$ This is the borderline and will result in an infinite loop.

If we start with $x_0 = 4$, then:

$x_0 = 4$	
$x_1 = 4 - \frac{(4-2)^2-1}{2*4-2} = 4 - \frac{3}{6}$	3.5
$x_2 = 3.5 - \frac{(3.5-2)^2-1}{2*3.5-2}$	3.25
$x_3 = 3.25 - \frac{(3.25-2)^2-1}{2*3.25-2}$	0.125

And therefore you can see that all values greater than 3 are being drawn inexorably into the vortex. The same is true for values $2 > v > 3$.

So it is important to select a first guess that is close, and within the zone of convergence. Since Newton is a quadratic method, meaning that it doubles the number of digits each iteration, if an initial guess is approximately correct, only one or two can take the answer to the full accuracy of the machine.

17.2 Ridder's Method

Ridder's method fits an exponential curve to the function to approximate where the root will be. It is quadratic (doubles the number of digits) under many but not all circumstances. It should be a lot more reliable than Newton's method, which can go crazy if the derivative goes to zero (a one-way ride to infinity) or if outside the radius of convergence, Newton's method will work in reverse, getting further from the root each time.

[TBD]

17.3 Numerical Integration

Numerical integration is a rich field, needed because some integrals are not analytically solvable. We cover trapezoidal for understanding the concepts in a simple method. Gaussian integration is better than trapezoidal for many reasons. We can see that higher order can be better, but is not always better. Then it turns out that using a clever cancellation scheme called Romberg is even more efficient than the Gaussian schemes.

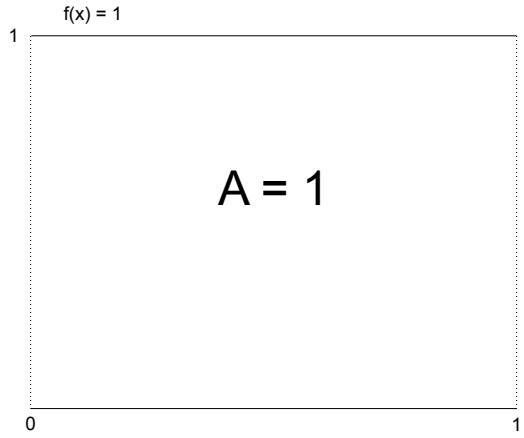
Last, because integration over a range could have a problem because some areas have higher error than others, adaptive quadrature can help focus on the areas that need finer approximation.

17.3.1 Trapezoidal Method

Given a function on an interval, and choosing two points to represent the height of the function, what would be the best way to sample it? Since we don't know the function, you would have to expect the answer to be symmetric. We wish to pick two points which in this case will be the two endpoints of the region. Then we pick two weights w_0 and w_1 and solve for them so they match a known function. We pick polynomials as the function, because polynomials can approximate many functions well (though not all).

All approximations will be on $x = [0,1]$ which can be transformed to any region $[a,b]$ without loss of generality.

We start by approximating the zeroth order polynomial $f(x) = 1$.



Given that the two points to sample the function are $x_0 = 0$ and $x_1 = 1$, and the fact that we need the overall weight to be 1 (if not, the function will be multiplied by a constant factor. For the function we know the area $A = 1$. With these facts, we can solve for the weights:

$h = 1$

$w_0 = w_1$

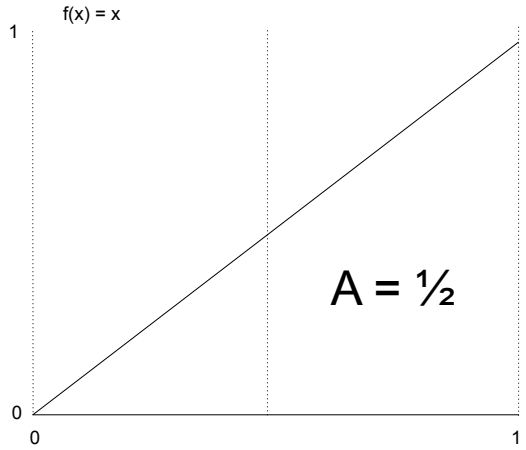
$A = 1 = h(w_0 f(0) + w_1 f(1)) \implies w_0 + w_1 = 1$

Solving, we get $w_0 = w_1 = 0.5$

In order not to require intuition stating that the two weights are equal, we could also go to the next polynomial of degree 1

$f(x) = x$

The area on $[0,1] = 0.5$



$0.5 = w_0 f(0) + w_1 f(1) = w_1$

Therefore $w_1 = 0.5$ and since the two sum to 1, w_0 is also 0.5.

Having used both polynomials, we now have a polynomial fit that works for zeroth and first order. Any polynomial of these degrees will be an exact fit. Any higher degree, or a function that does not look like a polynomial will have error. The size of the error terms is unknown, but the approximation I_n to the true value of the integral has a leading error term proportional to h^2.

$I_n = I + \alpha_1 h^2 + \alpha_2 h^4 + \alpha_3 h^6 + ...$

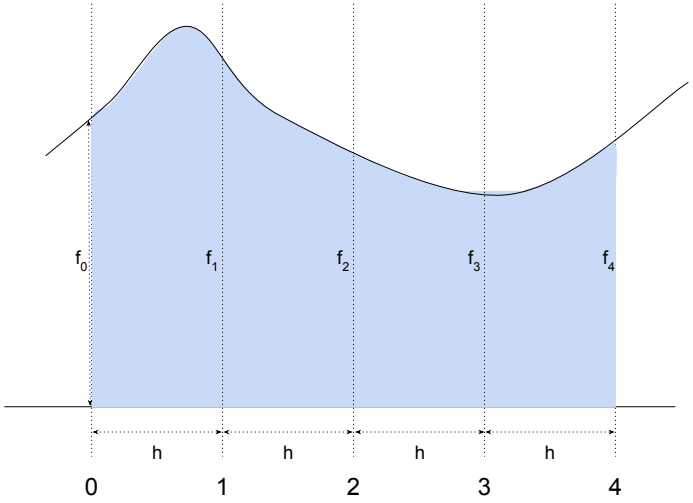
For example, consider the polynomial $f(x) = 3x + 2$ when integrated on $[0,1]$ is $3/2x^2 + 2x = 3.5$. This polynomial is composed of a constant and linear term and therefore should be exactly fit (with zero error to the limit of machine precision) by trapezoidal.

Trapezoidal yields $h(0.5 * f(0) + 0.5 * f(1)) = 1.0(0 + 3.5) = 3.5$

which is exact.

For a higher order polynomial or arbitrary function, the trapezoidal method is an approximation. The error will grow depending on how closely the function approximates the polynomials trapezoidal maps.

The following diagram shows the two points on the ends are used once each, so their weight of 0.5 is global. But all the internal points are used twice, one by the interval to its left and once by the interval to its right. So all internal points have weight 1.

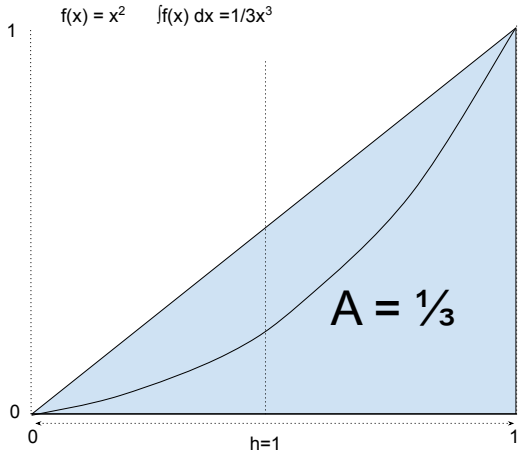


Since trapezoidal maps constant and linear terms, the first error term is $O(h^2)$. In order to get an accurate answer we will keep reducing the width of the sections (h). Every time h is divided by 2, the error term will decrease by $h^2 = 4$.

Example: Integrate the function $f(x) = x^2$ on the interval $[0,1$.

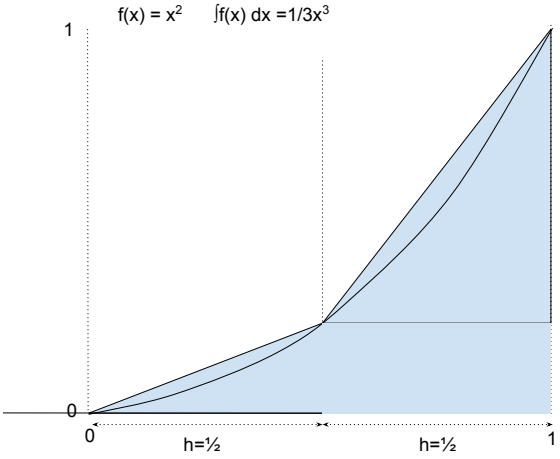
First we apply trapezoidal with a single slice. This will not be very accurate as the first diagram indicates. Keep the summation of the height of the function separate from the final estimate of the integral I_n, because it can be reused in the next step:

$S_1 = 0.5(f_0 + f_1) = 0.5(0 + 1) = 0.5$



The first estimate of the integral, $I_1 = hS_1$. In this case $h = 1$ so $I_1 = 0.5$. The correct answer is $\frac{1}{3}$ so the error for the first step is $\frac{1}{6}$.

Then we split each slice in half, doubling the number of slices. Note that $I_2 = I_1 + f_2$, the height of the function in the middle.

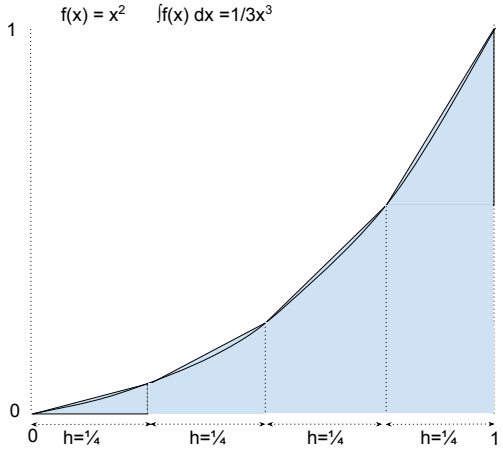


For the second iteration, $h = 0.5$

$S_2 = [0.5(f_0 + f_1) + f2]h = [0.5(0+1) + .25] * h = .75 * h = .375$

The error is thus $-.04$, already much smaller.

The next step is dividing each slice in 2 again:



Example: Consider the integral of $f(x) = 2x^3$ on the interval $[0,3]$. The analytical answer is $\frac{2}{4}x^4$ on $[0,3] = \frac{2(81)}{4} = 40.5$

Starting with trapezoidal:

$S_1 = 0.5(f(0) + f(3)) = 0.5(0+54) = 27$	$I_1 = hS_1 = 3(27) = 81$
$S_2 = S_1 + f(1.5) = 27 + 6.75 = 33.75$	$I_2 = hS_2 = 1.5(33.75) = 50.625$
$S_4 = S_2 + f(0.75) + f(2.25) = 57.375$	$I_4 = hS_4 = 0.75(57.375) = 43.031$
$S_8 = S_4 + ... = 109.69$	$I_8 = hS_8 = 0.375(109.69) = 41.133$
$S_{16} = S_8 + ... = 216.84$	$I_{16} = hS_8 = 0.1875(216.84) = 40.658$

The most efficient way to increase the number of slices in trapezoidal is to exactly double them. Then, the midpoint of each slice then becomes an edge. Given the weights are 0.5 on the edge and 1 on the interior, if the first sum with 2 slices is called S_1 then the second $S_2 = S_1 + P_2$. The new points are added with weight 1.

The value of the first integral approximation

$I_1 = S_1 * h_1$

The value of the second approximation

$I_2 = S_2 * h_2$.

In general, the value of the integral is not known which is why it is being computed. Therefore the way to use any integration method is to compute successive values I_1, I_2, I_4, ... and stop when:

$|I_n - I_{n-1}| < \epsilon$

The following C++ code is a trapezoidal integrator that runs until the difference between successive sums is sufficiently small.

```
double trap(Func f, double a, double b, double eps) {
  double S1 = 0.5*(f(a) + f(b));
  double S2 = S1 + f((a+b)*0.5);
  double h = (b-a) * 0.5;
  while (abs(S2-S1) > eps) {
    S1 = S2;
    double h2 = h * 0.5;
    double internalSum = 0; // sum all internal points
    for (double x = a+h2; x < b; x += h)
      internalSum += f(x);
    S2 = S1 + internalSum;
  }
  return S2 * h;
}
```

17.3.2 Midpoint Method

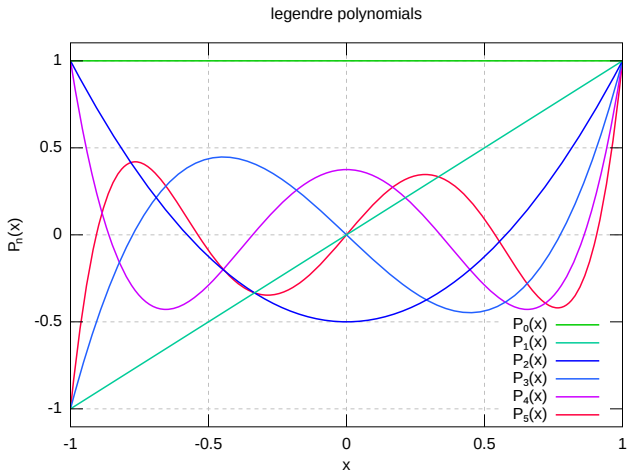
Trapezoidal is simple, has $O(h^2)$ error and if doubled each time, can robustly approach the answer. But it has certain disadvantages. Trapezoidal requires values on the endpoints, and sometimes the endpoints can be poles and therefore cannot be evaluated.

Midpoint as it turns out is exactly the same as trapezoidal $O(h^2)$ error but has only a single point to evaluate, dead center of the interval. It turns out halving the interval works exactly the same way, and that the error terms are $O(h^2 + h^4 + h^6 + ...)$ this result due to Roger Pinkham formerly a professor at Stevens. This is left as an exercise to the reader, and note that midpoint is also the first order Gaussian method. Gauss quadrature 2nd and 3rd order are described in the next section.

17.3.3 Gaussian Quadrature

Gaussian Quadrature is an interesting generalization of midpoint and trapezoidal method. It asks the question, where should the points be on a region to optimal sample the function? For example, with the two points of the trapezoidal method we can solve up to a 2nd order polynomial, that is the constant and linear term, but not the quadratic term because there are two weights, two unknowns which uniquely determine a linear function. However, with the same two points if we allow the x position to vary, there are 4 unknowns, and the first error term is $O(h^4)$.

It turns out that the optimal points for sampling the value of the function is at the roots of the Legendre polynomials. A graph of these polynomials is shown here.



The interval used to define Gauss Quadrature is $[-1,1]$ (see the picture above). Everything is relative to the center of the interval and therefore h is half the interval. For this reason, the sum of the weights should be 2 to avoid scaling the value.

Example: Compute the integral of $f(x) = x^3$ on $[-1,1]$. The analytical answer is

$\frac{1}{4}x^4 = 0$

The Wikipedia article in the references has a table of values and weights for different order solutions. Here we will reproduce just order 2 and 3.

order	points	weights
1	0	2
2	$\pm\sqrt{\frac{1}{3}}$	1
3	0	$\frac{8}{9}$
	$\pm\sqrt{\frac{3}{5}}$	$\frac{5}{9}$

To compute the answer with Gauss 2nd order, compute the function at the midpoint plus or minus the value $sqrt(1/3)$ in the table above. Since $h = 1$, we have to evaluate $f(\sqrt{\frac{-1}{3}})$ and $f(\sqrt{\frac{+1}{3}})$. The answer to 5 digits is $\pm.19245$ and when summed together this is, of course, 0.

Because Gauss quadrature 2nd order has 4 degrees of freedom it will be perfectly analytically acurate up to a 4th order polynomial, and Gauss 3rd order up to 6th order polynomial, so to demonstrate any problem we will have to compute an answer for higher order polynomials, or for an expression that isn't a polynomial. For example, $f(x) = x^6$ or $g(x) = exp(-x)$. Taking the first because it is easier, the analytical answer is $\frac{1}{7}x^7|_-1^1 = \frac{2}{7}$.

Wikipedia: Gauss Quadrature

17.3.4 Romberg

Romberg is a devilishly clever optimization of either trapezoidal or midpoint. The Euler-Maclauren theorem states that the error terms for these second order methods will only be even powers of two. Thus the approximation:

$I_n = I + O(h^2) + O(h^4) + O(h^6) + ...$

has a leading h^2 term, and the next one is not h^3 but h^4.

We can take advantage of this relationship in the following way: First compute 3 successive Integration answers, for example I_1, I_2, I_4. Since each successive approximation has h which is half the previous one, and since the leading error term is $O(h^2)$, for each answer, the error term is related to h but divided by 4. For example, since $h_2 = \frac{1}{2}h_1$, the leading error for $I_2 = O(\frac{h_1}{2}^2) = \frac{h_1}{4}$.

Therefore, we can compute: $R_1 = \frac{4I_2 - I_1}{3}$ and even though we do not know what the terror term is, we cancel the leading term leaving only the next term which is $O(h^4)$. If two Romberg sums are computed, the same thing can be done with these to cancel out the h^4 term:

$R_1 = \frac{4I_2 - I_1}{3}$ $R_2 = \frac{4I_4 - I_2}{3}$

$Q_1 = \frac{16R_2 - R_1}{15}$

The resulting answer thus has a leading error term of $O(h^6)$ with almost no additional computational effort. This amazing trick can be done twice, perhaps with one more layer, but that is the limit because small amounts of roundoff error mean that the errors are not actually identical in practice and by the third time, roundoff error is already at least partially destroying the accuracy.

Nonetheless, because Romberg takes so little computation, it presumably dominates over Gauss Quadrature in terms of computational efficiency.

Example: compute the integral of $f(x) = x^5$ on $[0,2]$. The analytical answer is $\frac{1}{6}x^6 = \frac{64}{6} = \frac{32}{3} = 10.666667$.

We know that this equation is beyond what midpoint or trapezoidal can do. Again, using trapezoidal:

Sum	h	terms	Estimate
S_1	2	$0.5(f(0)+f(2))=16$	$I_1=2(16)=32$
S_2	1	$S_1+f(1)=17$	$I_2=1(17)=17$
S_4	0.5	$S_2+f(0.5)+f(1.5)=24.844$	$I_4=0.5(28.844)=14$

Clearly the error from trapezoidal is getting better as the number of subdivisions increases, but not as fast as is desirable. To get 6 digits accuracy might require $n=512$ slices.

Instead, we compute Romberg and you can see how fast it reaches the correct answer

Doing Romberg again on the first Rombergs requires canceling the 2nd term which is $O(h^4)$. This means that if the second answer has an error of $h/2$ then the error may be cancelled with: $Q_i=\frac{16R_i-R_{i-1}}{15}$

17.3.5 Adaptive Quadrature

In an integral in which there is a pole, or sections where the values are large as well as sections where they are small, it may take a large number of slices to get the answer to the desired accuracy.

In such cases, what is needed is a recursive scheme to break up the integral into smaller pieces and solve each piece to the desired accuracy (ie a divide and conquer strategy). Just because some parts of the integral need a lot of work does not mean that they all do.

An adaptive quadrature routine automatically splits the region up, typically using divide and conquer, and computes the sum for each part to the desired accuracy. Midpoint using Romberg is the obvious choice for the implementation method. The complexity is unknown, because it depends on how hard the integral is, but the adaptive splitting is $O(nlogn)$.

```
AdaptiveQuadrature(f, a, b, ϵ)
  S1 ←
  do
     x ← (a+b)/2
     y ← f(x)
     if y < 0
        a = x
     else y > 0
        b ← x
     else
        return x
     end
  while abs(b−a) < ϵ
end
```

Algorithm 17.2: AdaptiveQuadrature

17.4 Solving Ordinary Differential Equations

17.5 Introduction

A differential equation starts at a location and using local (derivative) rules, computes the next position. The algorithm takes a small step forward, computes the new derivative and determines the next step. There is an error due to the finite step size. The smaller the step size, like integration, the smaller the error. But since roundoff error will kick in after enough small steps, The simplest algorithm is Euler (1st order approximation), effectively just an integration. Given a function $f'(x,t)$, presumably a function of x and time:

$y = x_0 + f'(x,t)dt$

Euler approximates the above equation with a finite Δt. There are two problems with this approach. The first is that the error is proportional to the step size so to get half the error, twice as many steps are needed. This quickly runs into millions of steps at which point roundoff error creates random error that kills any hope of a solution. The second problem is that there is no estimate as to what the error is. What is needed is a method that gives not only an answer but an estimate of the probable error of the answer. One of the most popular ways of achieving this are a class of differential equation solvers called Runge-Kutta which is next.

17.5.1 Runge-Kutta Methods

Runge-Kutta methods work by choosing intermediate values of dt, computing some trial points, and then using the information, jumping forward by a larger dt. Think of this as testing to see what the function is doing. The most famous is RK45, which cleverly computes two separate Runge-Kutta steps using some of the same values for efficiency. Given two separate estimates, a 4th order and a 5th order, by comparing the answers we can get an estimate of the error efficiently.

RKF45(r, c)

$k_1 = hf(t_k, y_k)$

$k_2 = hf(t_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1)$

$k_3 = hf(t_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2)$

$k_4 = hf(t_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 + \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3)$

$k_5 = hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4)$

$k_6 = hf(t_k + \frac{1}{2}h, y_k - \frac{8}{27}k_1 + 2k_2 - \frac{3533}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5)$

$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4101}k_4 - \frac{1}{5}k_5$

//or, use a 5th order step

$y_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$

end

Algorithm 17.3: Runge-Kutta-Fehlberg or RKF45

The optimal step size sh can be determined by multiplying the scalar s times the current step size h

$s = (\frac{tolh}{2|z_{k+1} - y_{k+1}|})^{\frac{1}{4}}$

17.5.2 Predictor-Corrector Methods

Runge Kutta methods are more efficient than low order methods because they permit a larger ΔT for the same accuracy. However, they do require many invocations of the function, and if the function is slow (for example, a gravity simulation in which the accelerations of every body on every body must be computed for each timestep) then the method still has very high computational cost.

One way to reduce the cost is with a predictor-corrector method. The predictor is an equation which uses the last n points of the differential equation to compute an estimate for the next value, and having computed the estimate, the corrector back-computes and attempts to fix any error. The difficulty with predictor-corrector schemes is that they require storage, and if an error is discovered, it may be necessary to backup multiple steps and recompute. Thus very often, predictor-corrector methods must use something like RK45 to get them started and to recover if the error climbs too high.

The Adams-Bashforth Predictor corrector is

Predictor: $y_{n+1} = y_n + \frac{h}{24}(55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3}$

Corrector: $y_{n+1} = y_n + \frac{h}{24}(9y'_{n+1} + 19y'_n - 5y'_{n-1} - y'_{n-2}$

17.6 Fast Fourier Transform (FFT)

A Fourier transform is an analytical transformation to turn a function amplitude as a function of time and decompose it into frequency. The Discrete Fourier Transform (DFT) takes a set of complex numbers $x_0, x_1, ..., x_{N-1}$

$X_k = \Sigma_1^{N-1} x_n e^{-i2\pi kn/N}, k = 0, 1, 2, ...N-1$

A DFT with $n = 4096$ points is $O(n^2)$ including N^2 complex multiplications and $N(N-1)$ additions. An efficient implementation would be about 30 million operations. By comparison, a Cooley-Tukey transform which is $O(nlogn)$ takes approximately 30,000. Fast Fourier transforms can be done on any size n, but is most efficient for powers of 2.

The accuracy of numerical Fourier transform is limited by the resolution of the data

coming in. For example with n points, the maximum resolution is $1/n$.

```
[x_0, ..., N-1] ← FFT(x, N, s)
  if N = 1 then
    X_0 ← x_0
  else
    [X_0,...,X_{N/2-1}] ← FFT(x,N/2,2s)
    [X_{N/2},...,N-1] ← FFT(x+s,N/2,2s)
    for k = 0 to N/2-1 do
      t ← X_k
      X_k ← t + e^{-2πik/N} X_k + N/2
      X_k + N/2 ← t - e^{-2πik/N} X_k + N/2
    end
  end
```

Algorithm 17.4: FFT

The following is a slightly modified version of the FFT found in Numerical Recipes. It is fairly efficient, and can be used both as the forward FFT, and the inverse operation. It uses an array of double, and each pair is a complex, so the variable n is the number of complex numbers, and nn is the number of individual values.

In order to compute $cos(2\pi m/n)$ and $sin(2\pi m/n)$ recursively as m doubles, they use a recursion relation for speed. First, $wr = 1$ and $wi = 0$ which are the first cos and sin values. Then, at the end of each loop a recursion relationship computes the values of the next cos and sin using a double angle formula for speed (although with a modern computer it is less certain how fast this is).

```
/*
   ******************************************************************

Replace data[1..2*nn] by its discrete Fourier transform, if isign
   =1
or replaces data[1..2*nn] by
nn times its inverse discrete Fourier transform, if isign=-1

data is a complex array of length nn or, equivalently,
a real array of length 2*nn.
nn MUST be an integer power of 2 (this is not checked).
******************************************************************
   */
void Fourier(double data[], uint32_t nn, int isign) {
  uint32_t m;
  uint32_t n = nn << 1;
  uint32_t j=1;

  // Reverse the bits of the positions
  for (uint32_t i = 1; i < n; i += 2) {
    if (j > i) {
```

```
      swap(data[j], data[i]); /* Exchange the two complex numbers
         . */
      swap(data[j+1],data[i+1]);
    }
     m = nn;
     while (m >= 2 && j > m) {
       j -= m;
       m >>= 1;
     }
     j += m;
  }

  uint32_t mmax = 2;
  uint32_t istep;
  while (n > mmax) { /* Outer loop executed log2 nn times. */
    istep=mmax << 1;
    double theta=isign*(6.28318530717959/mmax); // Initialize the
        trigonometric recurrence
    double wtemp = sin(0.5*theta);
    double wpr = -2.0*wtemp*wtemp;
    double wpi = sin(theta);
    double wr = 1.0; //cos(2pi*m/n)
    double wi = 0.0; //-sin(2pi*m/n)
    for (m = 1; m < mmax; m += 2) {
      for (int i = m; i <= n; i += istep) {
        j = i + mmax; // Danielson-Lanczos formula.
        double tempr = wr*data[j] - wi*data[j+1];
        double tempi = wr*data[j+1] + wi*data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] += tempr;
        data[i+1] += tempi;
       }
      wr=(wtemp=wr)*wpr-wi*wpi+wr; // Trigonometric recurrence.
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
}
```

To see what the code does, I printed out the code of the inner loop with the values of i and j for $n = 8$. Here is the result. The swaps seem wrong to me, but the code is straight out of their book, so the error is probably mine, this algorithm is still new to me. Note all the inefficiencies because of reading from memory, multiplying by 1 and zero (or worse, nearly zero giving roundoff errors). One big improvement is to write a function that does all the manipulation up to $n = 8$ and continue FFT from that point. By hand-coding the bottom 8 passes, the special cases with 1 and 0 can remove many multiplications,

and with a vectorized implementation using AVX2 instructions for example, all the calculations can be done in registers, probably a staggering improvement in performance.

One very nice group project would be to try to take advantage of AVX2 and use the 32 registers to do this and more, to try to keep the calculation in registers up to n= 16 or 32, whatever can be managed, while calculating the fourier in single precision floating point doing multiple operations at one time using the vectorized instructions.

```
swap 9, 3
swap 10, 4
swap 13, 7
swap 14, 8
double tempr = 1*data[3] - 0*data[4];
double tempi = 1*data[4] + 0*data[3];
data[3] = data[1] - tempr;
data[4] = data[2] - tempi;
data[1] += tempr;
data[2] += tempi;
double tempr = 1*data[7] - 0*data[8];
double tempi = 1*data[8] + 0*data[7];
data[7] = data[5] - tempr;
data[8] = data[6] - tempi;
data[5] += tempr;
data[6] += tempi;
double tempr = 1*data[11] - 0*data[12];
double tempi = 1*data[12] + 0*data[11];
data[11] = data[9] - tempr;
data[12] = data[10] - tempi;
data[9] += tempr;
data[10] += tempi;
double tempr = 1*data[15] - 0*data[16];
double tempi = 1*data[16] + 0*data[15];
data[15] = data[13] - tempr;
data[16] = data[14] - tempi;
data[13] += tempr;
data[14] += tempi;
double tempr = 1*data[5] - 0*data[6];
double tempi = 1*data[6] + 0*data[5];
data[5] = data[1] - tempr;
data[6] = data[2] - tempi;
data[1] += tempr;
data[2] += tempi;
double tempr = 1*data[13] - 0*data[14];
double tempi = 1*data[14] + 0*data[13];
data[13] = data[9] - tempr;
data[14] = data[10] - tempi;
data[9] += tempr;
```

```
data[10] += tempi;
double tempr = -6.66134e-16*data[7] - 1*data[8]; // roundoff!
   this should be 0*data[7]!
double tempi = -6.66134e-16*data[8] + 1*data[7]; // roundoff!
   this should be 0*data[8]!
data[7] = data[3] - tempr;
data[8] = data[4] - tempi;
data[3] += tempr;
data[4] += tempi;
double tempr = -6.66134e-16*data[15] - 1*data[16]; //roundoff
double tempi = -6.66134e-16*data[16] + 1*data[15]; //roundoff
data[15] = data[11] - tempr;
data[16] = data[12] - tempi;
data[11] += tempr;
data[12] += tempi;
double tempr = 1*data[9] - 0*data[10];
double tempi = 1*data[10] + 0*data[9];
data[9] = data[1] - tempr;
data[10] = data[2] - tempi;
data[1] += tempr;
data[2] += tempi;
double tempr = 0.707107*data[11] - 0.707107*data[12]; // could be
    wr*(data[11]-data[12])
double tempi = 0.707107*data[12] + 0.707107*data[11];
data[11] = data[3] - tempr;
data[12] = data[4] - tempi;
data[3] += tempr;
data[4] += tempi;
double tempr = -7.77156e-16*data[13] - 1*data[14]; // more
   roundoff
double tempi = -7.77156e-16*data[14] + 1*data[13]; // more
   roundoff
data[13] = data[5] - tempr;
data[14] = data[6] - tempi;
data[5] += tempr;
data[6] += tempi;
double tempr = -0.707107*data[15] - 0.707107*data[16];
double tempi = -0.707107*data[16] + 0.707107*data[15];
data[15] = data[7] - tempr;
data[16] = data[8] - tempi;
data[7] += tempr;
data[8] += tempi;
```

In all, it should be possible to make the implementation at least 30% more efficient by implementing non-recursively and computing special cases below n=8. It might even be higher than that given that modern computers are typically limited by memory latency. Those first passes can be implemented as 16 sequential reads followed by everything in

register, followed by 8 sequential writes. Not only is this faster for memory access, but since there is so much less memory accessed, 4 cores could easily be used, each one taking a group of 16 numbers, working in parallel. For parallel access, since memory is the limiting factor, designing the algorithm to minimize memory accesses is critical to performance.

17.7 References

Numerical Recipes

18. Data Compression

18.1 Information Theory

Data compression is built on a theory of information originated by Shannon in the 1950s.

Data compression is the act of writing data in a more compact form that uses fewer bits in such a way that the original data may be retrieved.

By definition, n bits have 2^n possible combinations. If all n-bit groups appear at random equally likely, there is no opportunity to compress. However, if patterns are not distributed uniformly, it is possible to dramatically reduce the length.

Bit entropy is the predictability of bits. If bits are truly random, there is no way of compressing them. However, there are many seemingly random sequences of bits that in fact, have an underlying structure. For example, analyzed statistically, the digits of pi are random, and the only way of encoding them is to store all the digits. However, there are programs that can not only compute the digits of pi, but can do so random access (ie, calculate the digits between 10000 and 10032). Given this ability, the most compact way of storing the digits of pi is to store a program capable of computing it.

Similarly, the Mandelbrot set is a complicated mathematical shape, and storing a high resolution image of a region can take Megabytes. Yet a small program can generate that image. The notion of using a program to construct the data, and computing the lower bound of size as the program is called Kolmogorov Complexity after the mathematician who defined it.

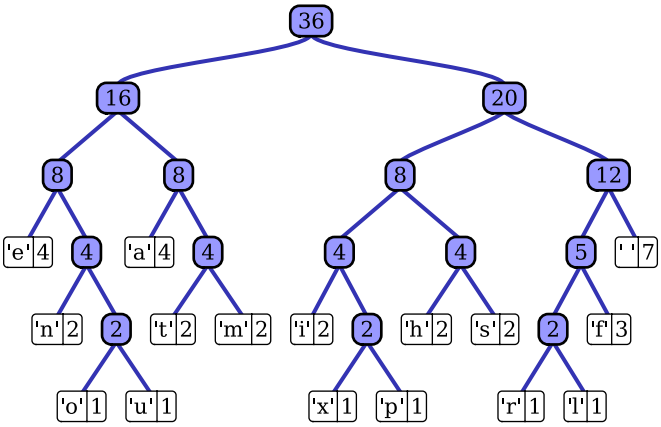
18.2 Run-Length Encoding (RLE)

Run-length encoding stores the number of each symbol along with the symbol. This can be very effective when there are many repeated values. For example, the string: aaaaaaabbbbbbbaaaaa could be encoded a7b7a5. The size of the lengths must be decided of course. The longer the allowable number of repeats, the more bits must be allocated to the counts (the log of the maximum number). For example, suppose the length is encoded as a single byte 1-256 since there is no point in encoding 0. Then for a sequence with 500 a, we would need: a256a244 for a total of 4 bytes.

18.3 Huffman Coding

Huffman coding allocates variablesized bit sequences to different symbols. The most common symbols get the shortest bit length. This reduces the average bit length for the input, unless the input is random. Unfortunately, assigning an integer number of bits does not perfectly match the distribution found, so Huffman is not optimal. For example, if 40 percent of the characters found are the letter 'e' then we can assign the bit 0 to e. Any other letter will require a 1 followed by a code to identify. By doing so, The letter e is as short as possible, but the remaining 60 percent of the input will be lengthened by 1 bit.

The following diagram shows the tree that the algorith must generate in order to encode as Huffman:



The algorithm for Huffman encoding is as follows:

Algorithm HuffmanEncode

```
HuffmanEncode(C)
  n = C.size
  Q = priorityQueue()
  for i = 1 to n
    n = node(C[i])
    Q.push(n)
  end
  while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
  end
  return Q
end
```

Algorithm 18.1: HuffmanEncode

Algorithm HuffmanDecode

```
HuffmanDecode(root, S):      // root represents the root of Huffman Tree
  n ← S.length               // S refers to bit-stream to be decompressed
  for i ← 1 to n
  current ← root
    while current.left ≠ null and current.right ≠ null
      if S[i] = 0
        current ← current.left
      else
        current ← current.right
      end
      i ← i + 1
    end
    print current.symbol
  end
end
```

Algorithm 18.2: HuffmanDecode

18.4 Lempel-Ziv-Welch

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It builds a dictionary of words as the source file is read. Any word that appears again can be encoded as the index of the word. In this way patterns, even very long ones that appear frequently can be reduced in size to the number of bits necessary to encode them in the dictionary. The more words there are, the more bits are needed to store them, and this can limit the savings. For example, in English the words ”and”, ”or”, ”but” appear very frequently, but given 1000 words in a dictionary, each with a code would be 10 bits, so the ratio of compression from two bytes (”or”) to 10 bits is not that large. On the other hand, in a repetitive text using formulaic language, like a math text, if the text ”without loss of generality” keeps occurring, it can be encoded as a single ”word” in the dictionary. Note that the match is exact, so if sometimes the phrase is at the start of a sentence ”Without loss of generality” then that would be an entirely different word in the dictionary, and there would therefore be two codes reserved for it. Also, note that the dictionary starts empty for each file. The first time a dictionary word is found the dictionary is augmented, so LZW works best when there are long sequences that repeat many times. For a sequence that only repeats twice, compression should be less than a factor of 2.

```
  s = ""; // initialize with empty string
  while (there is still data to be read)
    ch ← character input
    if (dictionary contains s+ch)
```

```
    s = s+ch;
  else
    encode s to output file;
     add s+ch to dictionary;
     s ← ch
  end
end
encode s to output file;
```

Algorithm 18.3: LZWEncode

```
prevcode = read in a code;
decode/output prevcode;
while (there is still data to read)
  currcode ← read in a code;
  entry ← dictionary[currcode]
  output entry;
  ch ← first char of entry;
  add ((translation of prevcode)+ch) to dictionary;
  prevcode ← currcode
end
```

Algorithm 18.4: LZWDecode

18.5 Burroughs-Wheeler

BurroughsWheeler is the underlying algorithm used by the Unix utility bzip2. It is a block compression algorithm that yield substantially higher compression than Huffman or LZW. It increases compression over LZW with three distinct phases.

- The algorithm attempts to transform the input into something more compressible by rotating bytes so that series of the same byte occur together, making them more amenable to compression.
- Movetofront encoding. Given sequences of the letters close to each other, they encode them as integer deltas and if the numbers are small, these are highly repetitive (many 0,1,2)
- Finally Huffman code the resulting data, and since short sequences are assigned to common values, the average length goes down substantially.

18.5.1 Burrows-Wheeler transform

The following excellent explanation is borrowed from an assignment in a Princeton CS class:

The goal of the Burrows-Wheeler transform is not to compress a message, but rather to transform it into a form that is more amenable to compression. The transform rearranges the characters in the input so that that there are lots of clusters with repeated characters, but in such a way that it is still possible to recover the original input. It relies on the following intuition: if you see the letters hen in English text, then most of the time the letter preceding it is t or w. If you could somehow group all such preceding letters together (mostly t's and some w's), then you would have an easy opportunity for data compression.

First treat the input string as a cyclic string and sort the N suffixes of length N. The following example shows how it works for the text message "abracadabra". The 11 original suffixes are abracadabra, bracadabraa, ..., aabracadabra, and appear in rows 0 through 10 of the table below. Sorting these 11 strings yields the sorted suffixes. Ignore the next array for now - it is only needed for decoding.

i	Original Suffixes	Sorted Suffixes	t	next
0	a b r a c a d a b r a	a a b r a c a d a b	r	2
1	b r a c a d a b r a a	a b r a a b r a c a	d	5
*2	r a c a d a b r a a b	a b r a c a d a b r	a	6
3	a c a d a b r a a b r	a c a d a b r a a b	r	7
4	c a d a b r a a b r a	a d a b r a a b r a	c	8
5	a d a b r a a b r a c	b r a a b r a c a d	a	9
6	d a b r a a b r a c a	b r a c a d a b r a	a	10
7	a b r a a b r a c a d	c a d a b r a a b r	a	4
8	b r a a b r a c a d a	d a b r a a b r a c	a	1
9	r a a b r a c a d a b	r a a b r a c a d a	b	0
10	a a b r a c a d a b r	r a c a d a b r a a	b	3

18.5.2 Move to front encoding

The concept behind move-to-front encoding is to maintain an ordered list of legal symbols, and repeatedly read in symbols from the input message, print out the position in which that symbol appears, and move that symbol to the front of the list. As a simple example, if the initial ordering over a 6 symbol alphabet is a b c d e f, and we want to encode the input caaabcccaccf, then we would update the movetofront lists as follows

movetofront	in	out
a b c d e f	c	2
c a b d e f	a	1
a c b d e f	a	0
a c b d e f	a	0
a c b d e f	b	2
b a c d e f	c	2
c b a d e f	c	0
c b a d e f	c	0
c b a d e f	a	2
a c b d e f	c	1
c a b d e f	f	5
f c a b d e		

After the first two passes, the result is Huffman encoded, and since there are many small numbers, the resulting average compression ratio is highly efficient.

18.5.3 Decompression of Burrows-Wheeler

18.6 Arithmetic Encoding

For encodings like Huffman or dictionaries like LZW, the problem is that there are not always an even power of 2 of symbols. For example, with 64 symbols in a dictionary we need 6 bits to represent all of them. With 65 symbols, every symbol grows to 7 bits even though most of the last half of the dictionary is empty. Arithmetic encoding uses multiplication and division which is slower but allows tighter packing of values that do not fit evenly into k bits. For example, suppose we have the following sequence of numbers:

4	0	3	1	4	3	3	0	2	2	3

Each number requires 3 bits, but since no value above 4 exists, really there are just 5 values not 7 (0 to 4). In order to encode in a 32 bit number, we can hold just 10 values if they are each 3 bits with 2 bits left over. Instead, we multiply by 5 each time and store using base 5:

$(((((((((4*5+0)*5+3)*5+1)*5+4)*5+3)*5+3)*5+0)*5+2)*5+2)*5+3 =$ 40386313

To unpack take the number mod 5 and remove the last number, then divide by 5 and repeat. Division of course is the slowest operation so this is less efficient than working with bits, but with the data shown we can fit 13 numbers using base 5, and still have a maximum of value of 1 billion. There is not enough to fit a 14th number but at more computational cost, that capacity could be used as well. There are effectively 2 bits remaining in the number.

18.7 Prediction by Partial Match (PPM)

TBD

18.8 ZPaq and the Hutter Prize

In the early 2000s, Matthew Mahoney, a professor at Florida Institute of Technology, wrote a paper on data compression using a neural network. Essentially, the neural network determines the kind of data, and once it recognizes it, it selects the most appropriate engine from a set of standard ones, and begins to predict what each bit will be. With an accurate prediction of bits, the only encoding needed is when the prediction is wrong. Prediction methods are extremely powerful because in theory, as the predictors become more and more intelligent, the limiting factor is very fundamental – how much knowledge is really encoded in the data?

The current generation of predictors based on Dr Mahoney's concept are very, very slow. It takes 8 hours or even days to compress 1Gbyte of data, but these engines can compress more than any other algorithm. The Hutter prize was founded to encourage research in this area and to explore the ultimate limits of how compactly information can be encoded. It consists of a one gigabyte section of Wikipedia. A bounty is paid any time a new program can successfully compress the data further. The size of the data includes the program. If it did not, we could write a "program" that included the entire gigabyte, write out 1 byte, and bring back the data from the code. The challenge of course is to not only encode it, but to do so with the simplest possible program and model. The Hutter prize is clearly related to Kolmogorov complexity. They are in effect writing a program, although most of it are neural network models, to squeeze out redundant bits and reconstitute information.

Huffman Coding	https://en.wikipedia.org/wiki/Huffman_coding
LZW Compression	https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
Prediction by Partial Match	https://en.wikipedia.org/wiki/Prediction_by_partial_matching
Hutter Prize	https://analyticsindiamag.com/hutter-prize-data-compression
Matthew Mahoney Data Compression	http://mattmahoney.net/dc/

19. Localization

19.1 Introduction

This section is incomplete: TBD

Localization is the term for determining a robot's location. Effectively it is the same thing as navigation, except for an autonomous computer.

Localization requires data, fed by sensors which measure some external signals to try to determine location.

19.1.1 Weaknesses of all Localization Algorithms

In this chapter we will cover two major methods of Localization: Kalman filtering and Particle filters. Kalman filters use an initial estimate of the accuracy of each sensor to decide how to compute a best-estimate of location based on all sensors. Particle filters do multiple randomized experiments, typically $n = 100$ to $n = 1000$, and by averaging all the estimates, compute at any moment the best estimate of location.

Because particle filters is significantly more computationally complex than Kalman filters, as part of my PhD thesis I came up with an alternative way of achieving the same thing called MADF.

Robots have a variety of sensors for determining position. First of all, if the robot is at a known location, ie sitting on top of a surveyer's mark, the position may be known to the measured accuracy of that station. We will consider this perfectly accurate. The earth has a magnetic field, so the robot may know the direction it is heading, and it may have some way of estimating its speed, such as how fast the propeller is spinning for a boat, how fast the wheels are turning for a car. Note that since wind can blow boats slightly in any direction, and wheels can slip, both of these methods are subject to steady erosion of accuracy which keeps increasing the further you go.

There is GPS, a complex beacon system where satellites in orbit relay the exact time and using 3 or more and measuring delays the receiver can compute its position. There are radio beacons, effectively the same thing using the delay from a radio source at a known location. Some systems may also give a relatively primitive angle toward the beacon as well. There are acoustic systems based on the same thing. Optical scene recognition can

allow the robot to recognize a landmark, and by noting the direction and distance to that landmark, estimate a position.

Most of these methods share a common theme. A sensor often has a means of determining the distance to a known location. This has an error which is probably normally distributed and a percentage of the signal. There may also be a fixed error, also normally distributed. Worst of all, sometimes this sensor will utterly fail due to some kind of environmental disruption (possibly manmade, ie jamming). In such cases, weighing in the wrong reading is catastrophically bad. All the methods of localization covered in this chapter share this problem.

What is needed, beyond these methods, is an intelligence to weight the input and make hypotheses and decisions based on those. This kind of decision-making is called Bayesian logic and is not covered here. Essentially, the robot must decide which sensors to believe at any moment. When a failure happens, it must discount the data from the bad sensor, because otherwise it will poison the results of the good sensors.

19.2 Kalman Filtering

Kalman filtering is a general information filter that takes n inputs about a variable, which may disagree, and constructs a single "best estimate" of the value of the variable.

19.2.1 1D Kalman

To simplify the initial discussion, consider a 1-dimensional situation. In a room, measuring a position in only the x direction. The following diagram shows a rectangular room where the robot is trying to determine its position from left to right:

[TODO: Diagram of 1D Kalman]

If there is a single SONAR sensor measuring the distance from one wall, then the best estimate of position is whatever that sensor says. Suppose this sensor has error that is normally distributed with a standard deviation of $sigma = 0.1m$. This means that the following table shows the probability of error:

probability	36%	86%	95%	98.1%	99.3%
distance $\pm$	σ	2σ	3σ	4σ	5σ

Notice that the probability if being outside 5σ or 0.5m is not zero, even though it is frequently neglected. Thus if SONAR informs the robot that its position is 1.5m, there is only a 36% chance that it is actually between 1.4 and 1.6m, but there is an 86% chance that is is between 1.3 and 1.7 and a 95% chance that it is between 1.2 and 1.8.

Next consider two identical SONARs, each pointing to opposite walls. The room is a known size, exactly 10m wide. SONAR A reports $x = 1.5m$, while SONAR B reports $x = 8.3m$, which is 8.5 meters from the wall at $x = 10$, in other words $x = 10 - 8.3 = 1.7m$.

There are two SONARS with conflicting information. Both cannot possibly be right (assuming that the dimension of the room is definitely known).

With no way to choose between them, the best guess is to take the average

$\hat{x} = \frac{1.5+1.7}{2} = 1.6m$

The fact that the two estimates differed by 0.2m seems believable since each answer is only accurate to ± 0.3 with 95% probability. But that means that sometimes the answer should be closer (for example SONAR A=1.65, SONAR B = 8.55). So if we consistently see the two disagree in the same sine we might start to conclude that both sonars are under-estimating distance and change our estimate of the variance. In practice, while there may be bias, the error is usually worse than the theoretical and random.

Moreover, even if the SONARs are identical the situation is probably not identical. The error may be proportional to the distance from the wall. If σ is a relative error of 1% of the measured distance then the error of the left SONAR is $1.5(0.01) = .015$ or 1.5cm while the error to the right side is $8.3(0.01) = 0.083$ or 8.3cm. Therefore, the best idea is to weight the left SONAR more heavily than the right because it is more accurate.

Kalman filtering weighs the inputs by the inverses of their variance, which is the square of the standard deviation. So in this case:

$w_A = (1.5(0.01))^2 = 0.000225$ $w_B = (8.3(0.01))^2 = 0.006889$ $\hat{x} = \frac{\frac{x_A}{w_A} + \frac{x_B}{w_B}}{w_A + w_B} = 1.5063m$

In other words, the bigger the variance of SONAR B, the more we trust SONAR A. We therefore assume the answer is closer to SONAR A's estimate than SONAR B.

The problem with Kalman filtering is that the definition of the variance is arbitrary, and very difficult to pin down. After all, without knowledge of the truth we cannot begin to measure the real error and begin to characterize it.

19.2.2 2D and 3D Kalman

To estimate a position on earth requires two coordinates, latitute and longitude. To estimate a position in 3d requires an elevation as well. For the purposes of the discussion here, consider the earth to be a perfect sphere. The references contain links to the real story which is quite a bit more complicated.

Given a number of conflicting reports, the best guess as defined by Kalman is based on an intrinsic accuracy of each reading. Each sensor is assigned a variance, and the smaller that variance, the more we rely on that sensor. For example, given that we have GPS and a wheel rotation counter, we could say that since the GPS is accurate to 2m 95% of the time, and the wheel loses a little bit every rotation, that measuring distance in a straight line, the GPS has a variance of $\sqrt{2}$ while the wheel sensor has a variance of ϵR where epsilon is a small error each time such as 0.001 and R is the number of rotations. This is highly misleading however. If someone jams the GPS, or if there is interference with multiple paths in a city, the GPS could change and lose accuracy. And given that we

know our current location, the relative error of a few wheel turns gives us a very accurate estimate of our location for a short time. So if the GPS were to fail, the best estimate of our location is not to assume the GPS is much more accurate, but to create a reasonable hypothesis, like GPS jamming began at time = t, and all subsequent estimates from GPS satellite x were wrong after that time.

TODO: Variance estimates of sources

19.3 Particle Filtering

19.4 Multiple Analytical Distribution Filter (MADF)

This last one was my idea based on a hatred of Monte-Carlo methods.

20. Graphical Algorithms

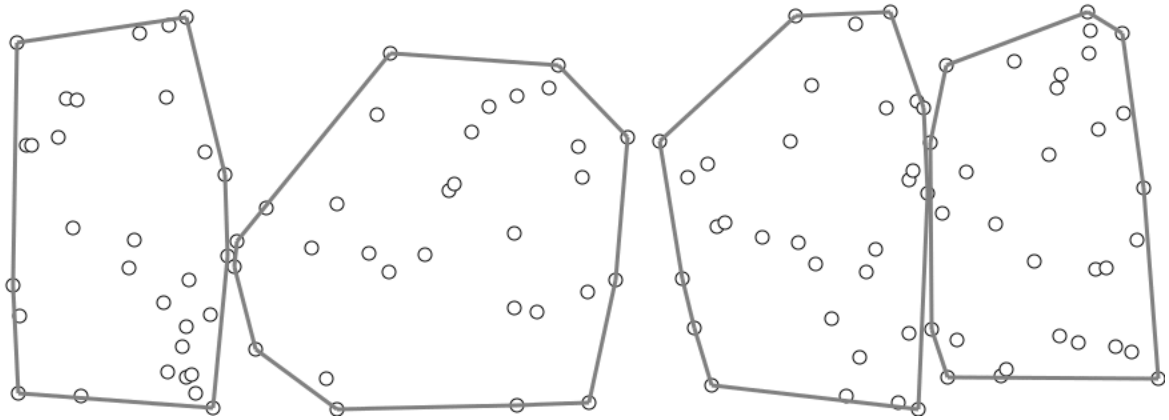
20.1 Convex Hull

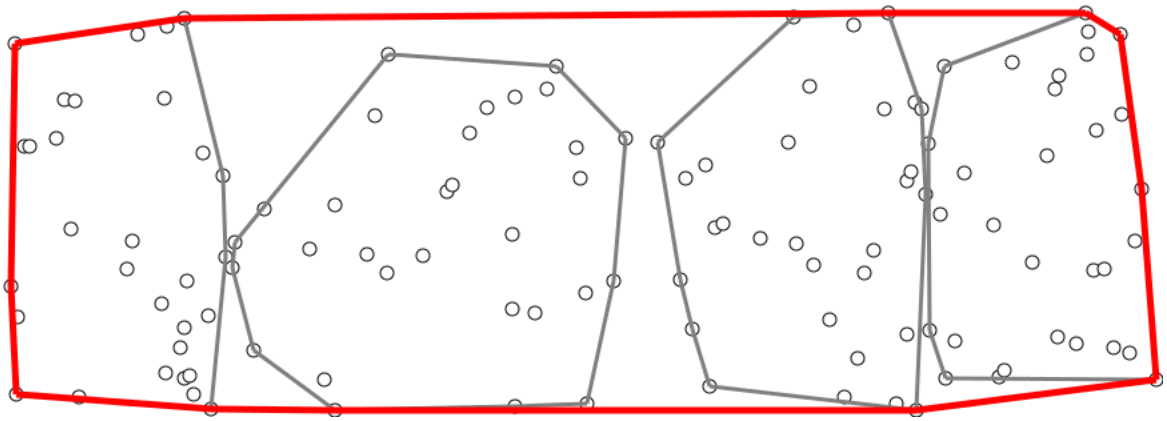
The convex hull problem is simply stated. Given a set of points find the smallest convex polygon and encloses them all. In three dimensions, the analogous problem would be to find a polyhedron.

There are many algorithms as usual. The brute force are $O(n^2)$. For each point, we need to find the next one clockwise and reject all points that are within. Since most points are on the inside, this approach is horribly inefficient.

The best currently is due to Chan which is $O(n \log h)$ where n is the number of points, and h is the size of the output polygon, which is usually much smaller than n.

The following diagram shows how the Chan algorithm splits the points into groups, solves the hull of each groups, then combines them into a single hull.





```
Chan algorithm goes here
```

Algorithm 20.1: Chan Algorithm

20.2 Delaunay Triangulation

20.3 Grid Assignment

20.4 Box Packing

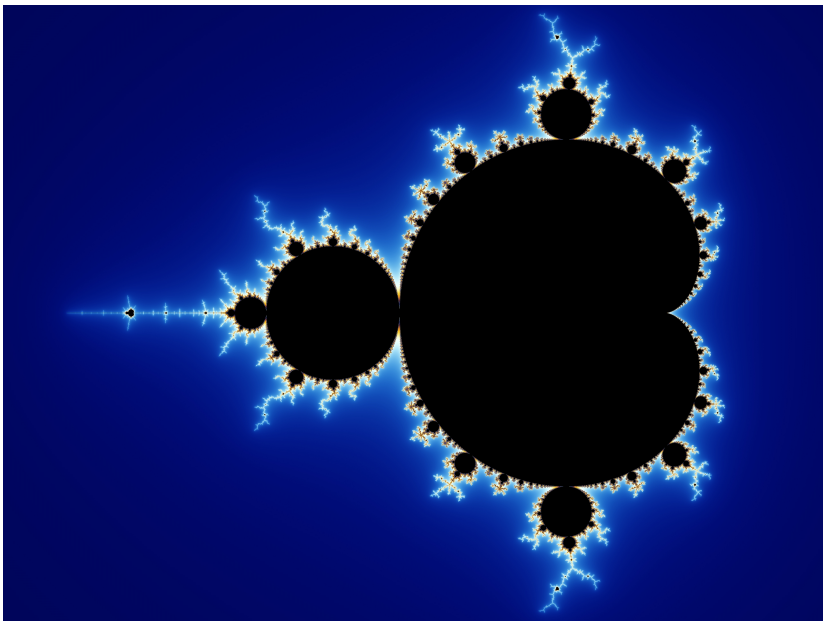
20.5 Mandelbrot Set

This algorithm does not quite fit into this chapter, but it is graphical, and doesn't fit anywhere else. The Mandelbrot set was invented or discovered by Bernard Mandelbrot. For any point C in the complex plane, repeatedly compute:

$Z = C$

$Z = Z^2 + C$

The magnitude of some points will increase fast, others will stay small forever. After any point has a magnitude of 2, it will quickly go to infinity. By counting how many times it takes to get to a magnitude of 2.0, and then colorizing a picture according to the counts, we can create the extraordinary images of the Mandelbrot set. One of the properties of the set is that it is self-similar at all levels. If you blow up a region you can see that it contains features looking exactly like the main one. In this picture, black is set to those points that never escape to infinity.



This image courtesy of Dr. Wolfgang Beyer from Wikimedia.

The Mandelbrot set can be computed brute force by a triple-nested loop. The complexity is the resolution in the x and y, (m,n) and the number of iterations k computed in order to calculate the colors. At first k can be just 64, but as the algorithm zooms into an area, it may grow to millions and be the dominating cost.

Algorithm Mandelbrot $O(kmn)$

```
  for i = 1 to m                                    1
    for j = 1 to n                                  2
      z = c_ij                                      3
      count = 0                                     4
      while abs(z) < c_ij                           5
        count ← count + 1                           6
      end                                           7
      pixel(i,j) = colorlookup[count]               8
    end                                             9
  end                                               10
end                                                 11
```

Algorithm 20.2: Mandelbrot

Mandelbrot is an ideal algorithm for parallelization. After all, every pixel involves a separate computation that is completely independent, and memory only needs to be written once for each pixel. It should be possible to compute in parallel on a modern multi-core CPU, but also to use vectorized instructions like AVX2. With vectorization though, there is a problem that if one pixel ends earlier than others, they must all go in lockstep until the last one is done. Still, it may be possible to achieve a great deal of speedup with vectorization. GPU programming is also a method with huge potential, as the GPU has thousands of execution units and is ideal for parallel computation of this type.

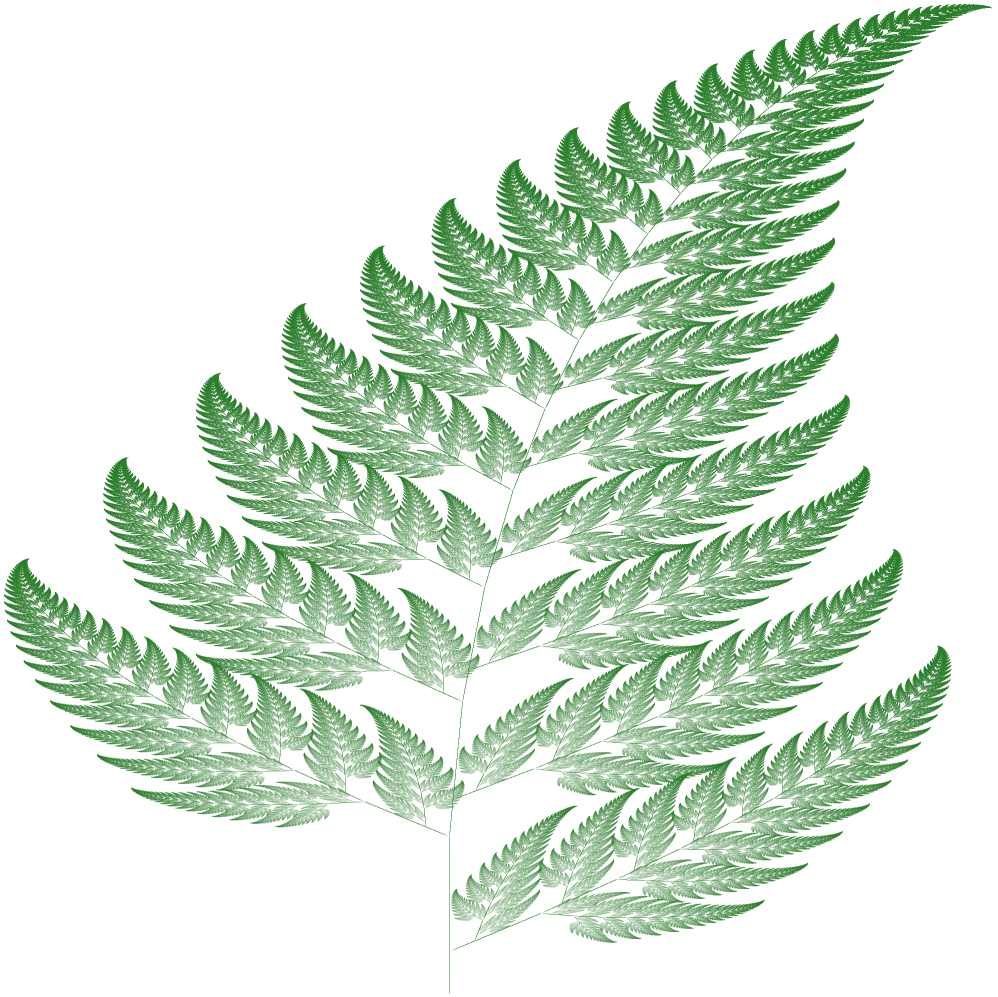
Potentially a huge win are algorithmic improvements. For example, at any resolution, if a rectangle can be computed, and every element on the edge of the rectangle is black (maximum iterations) then it should be provable that every interior element must be black as well, and there is no reason to compute the interior. Figuring out an optimal strategy to calculate edges to eliminate the maximum amount of interior is an interesting problem for an arbitrary region of the Mandelbrot set.

https://en.wikipedia.org/wiki/Mandelbrot_set

20.6 Barnsley Ferns

Another fascinating graphical algorithm that doesn't fit anywhere else in this course is Barnsley ferns. Using only a simple iterative system with a small matrix, it is possible to generate a set of points that look like ferns, and with slight tweaks, many different kinds of plants. One might think based on the pictures that these plants are recursively constructed by big leaves which branch to smaller leaves. But the amazing thing is that the iterative system doesn't work that way. If you watch the pixels added, they jump around in seemingly random order, yet as more dots are added, the plant emerges. The question of course is whether any plant has DNA programming somehow encodes a mechanism like this (which seems utterly impossible), or whether a plant has equally simple instructions that branch off at random intervals, getting smaller for each branch.

The following image shows a Barnsley fern.



The algorithm is to start at coordinate (0,0) and repeatedly apply an affine transformation which can translate rotate and scale the fern. Each affine transformation is a 2x2 matrix:

$$f_1(x,y) = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.16 \end{bmatrix}$$
$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix}$$
$$f_3(x,y) = \begin{bmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{bmatrix}$$
$$f_4(x,y) = \begin{bmatrix} 0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix}$$

The odd thing is that each of these transforms is computed with a probability, and the result draws the fern. The dots are not drawn sequentially from biggest leaf to smallest. They fill in at random, since the transformations are being chosen randomly, yet the result will look like a fern every time. And by picking different numbers for the matrices and different probabilities, different plants can be generated.

Drawing a fern like this takes tens of thousands of points. It is not fast, but it allows realistic drawing of plants which can be used in movies, for example.

The following example draws a Barnsley fern in processing:

```
/*
  Barnsley Fern
*/

// creating canvas
void setup() {
  size(800, 800);
  background(255);
  stroke(35, 140, 35);
  strokeWeight(1);
}

void draw() {
  float x = 0, y = 0;
  float temp;
  for (int j = 0; j < 100; j++) // because draw is slow, do 100
     of these each time
    for (int i = 0; i < 100; i++) { // iterate each point 100
       times
      float px = map(x, -2.1820, 2.6558, 0, width); // pick
         initial point
      float py = map(y, 0, 9.9983, height, 0);
      point(px, py);
      float r = random(1);// pick a random transform

      //now compute the next x,y value using transform
      if (r < 0.01) {
        x = 0;
        y = 0.16 * y;
      } else if (r < 0.86) {
        temp =  0.85 * x + 0.04 * y;
        y = -0.04 * x + 0.85 * y + 1.6;
        x = temp;
      } else if (r < 0.93) {
        temp =  0.20 * x - 0.26 * y;
        y =  0.23 * x + 0.22 * y + 1.6;
        x = temp;
      } else {
        temp = -0.15 * x + 0.28 * y;
        y =  0.26 * x + 0.24 * y + 0.44;
        x = temp;
      }
    }
  }
}
```

For more details, see: https://en.wikipedia.org/wiki/Barnsley_fern

21. Serialization

21.1 Introduction

Many times, a data structure must be loaded from permanent storage on disk into RAM, or perhaps sent across the internet. Pointers are typically local to a computer and not repeatable, so any data structure involving pointers must be reconstructed every time it is loaded. In certain cases, such as dictionaries, the act of loading can be the single largest computational cost of using the data structure, larger than the cost of doing the spell check since the document is often smaller than the dictionary.

This chapter examines approaches that can be used to efficiently serialize a data structure.

There are two main technical problems in order to serialize data. The first is cycles. Objects may be in a graph where each object refers to others, and the serialization mechanism must not write anything more than once. Second, each object is at a unique location in memory, but that location will not be the same when the data structure is loaded in later. This means that serialization requires a mapping of pointers to some representation that can be written to disk, and then loaded back again.

For the storable representation, we choose sequential integers, but each type of object can have its own list.

TBD: topics include

1. reconstituting pointers
2. relative integers vs. pointers
3. maps