EECS 281

Data Structures and Algorithms

David Paoletti - Winter 2015

Contributors: Steven Schmatz, Max Smith

Latest revision: August 23, 2015

Contents

Abstract

EECS 281 is an introductory course in data structures and algorithms at the undergraduate level. The objective of the course is to present a number of fundamental techniques to solve common programming problems. For each of these problems, we will determine an abstract specification for a solution and examine one or more potential representations to implement the abstract specification, focusing on those with significant advantages in time/space required to solve large problem instances. When appropriate, we will consider special cases of a general problem that admit particularly elegant solutions.

1 Introduction and Workflow

2 Complexity Analysis

What affects runtime?

- **The algorithm**: By relative measure, this is always on top. If there is an algorithm to do something in linear time when the competing implementation is in polynomial time, there's a clear winner.
- **Implementation details**: Without knowledge of good programming style, there can be very subtle and tricky errors that can slow the algorithm down. For example, when iterating through a 2D array, you should always go row-major style to access sequential elements in memory.
- **CPU speed and memory**: Usually, by investing a lot of money, you can only get a 2x-5x improvement in computing power. Not much.
- **Compiler options**: The -g flag adds annotations to your files and should only be used for debugging. The -O3 flag turns on the highest level of optimization.
- **Parallel programs**: What is running at the same time as your program in the processor. Virtually makes no difference.
- **Input size**: This is the hardest thing to change, anyway.

The key question we seek to answer is: does a fast algorithm remain fast at large values of n?

Notation:

- n: input size
- $f(n)$: max number of steps taken by an algorithm when input has length n
- $\mathcal{O}(f(n))$: upper bounds up to a constant

However it some cases defining n remains ambigious: given a graph $V = 5, E = 6$ what would be n? It depends.

Steps in a program are considered to be a single expression:

- Variable assignment
- Arithmetic operations (not all)
- Comparisons
- Array indexing
- Function calls

– etc.

An example of counting follows:

```
int sum = 0;                        // init: 1
for (int i = 0; i < n; ++i){        // init: 1, test: n, update: n, test fail: 1
    sum += 1;                       // 1*n
}
return sum;                         // 1
```

In this example the sum of steps is $4 + 3n$. **Note**: loops generally take $2n + 2$ steps.

Definition 2.1 (Big O)**.** $f(n) = \mathcal{O}(g(n))$ iff there exists $c > 0, n_0 > 0$ such that $f(n) \leq c \times g(n)$.

Definition 2.2 (Limit Definition of Big O)**.** If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = d < \infty$ then $f(n)$ is $\mathcal{O}(g(n))$.

Common Orders of Functions:

Notation	Name
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(N)$	Linear
$\mathcal{O}(n \log n)$	Loglinear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(c^n)$	Exponential
$\mathcal{O}(n!)$	Factorial
$\mathcal{O}(2^{2^n})$	Doubly Exponential

3 Measuring Runtime, and Pseudocode

3.1 Logarithmic Runtime

Logarithmic behavior is typically in a program that divides the input size every step. This could be something like binary search, or an update condition in a `for` loop:

```
for (int i = 100; i > 0; i /= 2){/* loop contents */}
```

3.2 Complexity Analysis

– Each step takes $\mathcal{O}(1)$ time

– **Goal**: Find time based on n

– Tools:

  – Find rate of growth of $f(n)$

  – Use Big-O notation to compare growth rates

3.3 Complexity Cases

- **Best case**: The input case of size n which will take the shortest amount of time. For example, in linear search, the element could be the first element in the array.
- **Worst case**: The input case of size n which would take the most amount of steps. In linear search, this would mean checking every element until the end.
- **Average case**: The average input case. This would be $\frac{n}{2}$ in linear search.

3.4 Amortized Complexity

A special kind of worst-case complexity that considers how an algorithm performs over a large series of inputs. It is just the total cost of the operations divided by n. "If you use it a lot, it behaves as if it were ..."

3.5 Measuring Runtime

On Unix, use usr/bin/time. call it like so:

`/usr/bin/time` [**command**]

Or, if you want to measure the amount of time a block of code takes in C++, use `<sys/time.h>`:

```
#include <resource.h>
#include <sys/resource.h>
#include <sys/time.h>

int main() {
    struct rusage startu;
    struct rusage endu;

    getrusage(RUSAGE_SELF, &startu);
    // Computation
    getrusaage(RUSAGE_SELF, &endu);
}
```

3.6 Pseudocode

Pseudocode is a special way to describe algorithms. It is a mixture of English and programming to communicate whan an algorithm does in the simplest way.

function FizzBuzz
    $print_number \leftarrow true$
    for i := 1 to 100 **do**
        if i is divisable by 3 **then**
            print "Fizz";
            $print_number \leftarrow false$
        end if
        if i is divisble by 5 **then**
            print "Buzz";
            $print_number \leftarrow false$
        end if

```
        if print_number then
            print i + newline;
        end if
    end for
end function
```

It is meant to be readable, and there is no single standard.

4 Recursion and the Master Theorem

Consider the factorial function:

```
int factorial(int n){
    if (n == 0){
        return 1;
    return n * factorial(n - 1);
}
```

The time complexity of this function is:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ T(n-1) + c_1 & \text{if } n > 0 \end{cases}$$

Solving this function we obtain the growth function:

$$\begin{aligned} T(n) &= T(n-1) + c_1 \\ T(n-1) &= (T(n-2) + c_1) + c_1 \\ T(n-2) &= ((T(n-3) + c_1) + c_1) + c_1 \\ &\vdots \\ T(n-k) &= T(n-k) + kc_1 \\ T(n) &= nc_1 + c_0 \end{aligned}$$

This growth function is on the order $\mathcal{O}(n)$.

4.1 Tail Recursion

When a function gets called, it gets a stack frame, which stores the local variables. Also, each recursive call generates another stack frame for each recursive call!

A function is **tail recursive** if there is no pending computation at the end of the recursive step.

Example of a normal recursive function:

```
int factorial(int n){
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Contrasted with the tail recursive version (notice there's no further computation in the return statement):

```
int factorial(int n, int result = 1) {
    if (n == 0)
        return result;
    else
        return factorial(n - 1, result * n);
}
```

4.2 Counting Steps

```
int myFunc(          // 1 step
            char *p, // 1 step
            int aN,  // 1 step
            int ar[] // 1 step) {

    return   // 1 step
        zzz; // 1 step
}
```

```
char *course = "EECS281";           // 7 steps (copy chars at runtime)
int HWKs[4] = {100, 110, 120, 140}; // 4 steps (copy at run time)
int retCode =                       // 1 step
    myFunc(                         // 2 steps (call and return);
        course, 4, HWKs);           // 4 steps (each var and return var)
```

4.3 The Program Stack

Function Call Internal Operations

When a function call is **made**, all local variables are saved in a special storage called the stack. Then, argument values are passed onto the stack.

When a function call is **received**, the function's arguments are popped off the stack.

When a function issues a `return`, the return value is pushed onto the stack.

When `return` is received, the return value is popped off the stack, and saved local variables are restored.

4.4 Stack Properties

The stack supports **nested function calls**, and each has its own set of *local variables* and *arguments*.

There is only **one program stack** (per thread). This is different from the program heap, where dynamic memory is allocated.

Program stack size is **limited** in practice, so the number of nested function calls is limited based on the size of that stack. This means that "plain recursion" is a bad idea over every element. Use tail recursion or iterative algorithms instead. However, for programs solvable with $\mathcal{O}(1)$ additional memory, they do not favor "plain" recursive algorithms.

4.5 Exercise: Step Counting

```
int factorial(int n) {          // 2 steps
    if (n == 0)                 // 1 step
```

```
3         return 1;                 // 2 steps
4     return n * factorial(n - 1); // 6 step
5 }
```

The last line is 6 steps because there is subtraction, function call, argument passing, internal return, external return, and multiplication.

4.6 Exercise: Tail Recursive Power Function

This is an okay implementation. Could be better. Runtime: $\mathcal{O}(n)$

```
1 int power_recursive(int x, unsigned int y) {
2     if (y == 0)
3         return 1;
4     return x * power_recursive(x, y - 1);
5 }
```

This is much better, because it doesn't create additional stack frames per recursive call. Runtime: $\mathcal{O}(n)$

```
1 int power_tail(int x, unsigned int y, int result = 1) {
2     if (y == 0)
3         return result;
4     return power_tail(x, y - 1, result * x);
5 }
```

This is the best implementation here, because it has $\mathcal{O}(\log n)$ time.

```
1  int power_iterative(int x, unsigned int y) {
2      int result = 1;
3
4      while (y > 0) {
5          if (y % 2)
6              result *= x;
7          x *= x;
8          y /= 2;
9      }
10
11     return result;
12 }
```

This is a recursive version of this better logarithmic power function helps us write the runtime:

```
1 int power(int x, unsigned int y, int result = 1) {
2     if (y == 0)
3         return result;
4     else if (y % 2)
5         return power(x * x, y / 2, result * x);
6     else
7         return power(x * x, y / 2, result);
8 }
```

The runtime is:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ T(n/2) + c_1 & \text{if } n > 0 \end{cases}$$

So:

$$\begin{aligned} T(n) &= T(n/2) + c_1 \\ T(n/2) &= (T(n/4) + c_1) + c_1 \\ &\vdots \\ T(1) &= T(\frac{n}{2^k}) + kc_1 = c_0 + kc_1 \\ n &= 2^k \\ k &= \log n \end{aligned}$$

Resulting in the growth function: $T(n) = c_0 + \log n = \mathcal{O}(\log n)$

4.7 Common Recurrence Relations

- **Binary search**: $T(n) = T(n/2) + c$
- **Sequential search**: $T(n) = T(n-1) + c$
- **Tree traversal**: $T(n) = 2T(n/2) + c$
- **Insertion sort**: $T(n) = T(n-1) + c_1 n + c_2$

4.8 Solving Recurrences with the Master Theorem

One way to do it is through the telescoping method, which can be difficult at times. This is where the **Master Theorem** helps.

Let $T(n)$ be a monotonically increasing function that satisfies:

$$T(n) = aT(\frac{n}{b}) + f(n) T(1) = 1$$

Where $a \geq 1, b \geq 2$. If $f(n) \in \Theta n^2$, then:

$$T(n) = \begin{cases} \Theta(n^{n \log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^2 \\ \Theta(n^c) & \text{if } a < b^2 \end{cases}$$

Note that this doesn't work in the following circumstances:

- $T(n)$ is not monotonic, such as $T(n) = \sin(n)$
- $f(n)$ is not polynomial, such as $f(n) = 2^n$
- b is not a constant, such as when it's a function
- Not dividing n by anything in each step
- $f(n)$ has to be a polynomial

There is another case which allows polylogarithmic functions for $f(n)$.

4.9 Job Interview Question

Write an efficient algorithm that searches for a value in an $n \times m$ array. THis is sorted along rows and columns. That is,

```
table[i][j] <= table[i][j+1]
table[i][j] <= table[i+1][j]
```

Obvious way: linear or binary search in every row. Better way: start in the middle.

4.10 Solution 1: Quad Partition

Split the region into four quadrants one can be eliminated. If the number you see is too small, it can't be in the up left quadrant. If it is too big, it can't be in the bottom right quadrant.

$$T(n) = 3T(n/2) + c$$

By the master theorem,

$$T(n) = \Theta(2^{\log_2(3)})$$

4.11 Solution 2: Binary Partition

Split the region into four quadrants.

- Scan a middle row/column/diagonal for the target element. Look for where the 13 should be.
- If not found, split where it would have been
- Eliminate 2 or 4 sub-regions

$$T(n) = 2T(n/2) + cn \text{ or } T(n) = 2T(n/2)$$

By the master theorem:

$$T(n) = \Theta(n \log n) \text{ or } T(n) = \Theta(n)$$

4.12 Solution 3: Stepwise Linear Search

Start from the top right corner, and only move down or right depending on whether the number at the index is less than or greater than the target.

```
bool stepwise(int mat[][N_Max], int N, int target, int &row, int &col) {
    if (target < mat[0][0] || target > mat[N-1][N-1]) {
        return false;
    }
    row = 0;
    col = N - 1;

    while (row <= N && col >= 0) {
        if (mat[row][col] < target) {
            row++;
        } else if (mat[row][col] > target) {
            col--;
        } else {
```

```
14                 return true;
15             }
16         }
17
18         return false;
19     }
```

4.13 Linear Recurrences

Linear recurrences are sequences of the form:

$$F_n = c_1 F_{n-1} + c_2 F_{n-2}$$

These appear frequently in many contexts:

- Stock-trading strategies
- Nice-looking architectural proportions
- Nature
- Interview questions

These can be calculated recursively (which ends up being terrible), linearly, and with some nifty tricks.

5 Arrays and Container Classes

Let's start with an interview question.

A majority element is an element in array constitutes more than 50% of elements. For example, consider the numbers:

$$11, 13, 99, 12, 99, 10, 99, 99, 99$$

Here, the majority element is 99. **Challenge**: Find the majority element in linear time using $\mathcal{O}(1)$ memory.

Use Moore's voting algorithm.

5.1 C Arrays and Strings

C arrays and strings are useful to know for legacy codebases, but you should always use `array<T>` and `vector<T>` instead for projects.

5.2 Range-based for loops

In C++11, there are for-in loops. They are used like so:

```
for (auto x: array){/* use x */}
```

If you want to change values in `array`, use:

```
for (auto &x: array){/* change x values */}
```

5.3 Container Classes

Objects that contain multiple data items, such as ints, doubles, or objects. They allow for control and protection over editing of objects, and can copy/edit/sort/order many objects at once.

They're also very combinable - you can have a vector of stacks.

6 Linked Lists and Iterators

6.1 Recap - Bounds Checking for Arrays Example

Basic Layout

```
class Array {
 public:
    Array(unsigned len = 0) : length(len) {
        data = (len ? new double[len] : nullptr);
    }
 private:
    double *data;         // array data
    unsigned int length; // array size
}
```

You could use a struct, but users of your class would be able to access all member variables of your struct by default.

Copy Constructor

To copy the array, you need to perform a *deep copy*:

```
Array(const Array & a) {
    length = a.getLength(); // getter for length // 1 step
    data = new double[length];                   // 1 step
    for (unsigned i = 0; i < length; i++) {      // n times
        data[i] = a[i];                          // c steps
    }
}
```

The complexity of this is: $\mathcal{O}(! + 1 + (nc) + 1) = \mathcal{O}(n)$

Better Copying

```
void copyFrom(const Array & a) {
    if (length != a.length) {
        delete[] data;
        length = a.length;
        data = new double[length];
    }

    for (unsigned int i = 0; i < length; i++) {
        data[i] = a.data[i];
```

```
    }
}

Array(const Array & a) : length(0), data(nullptr) {
    copyFrom(a);
}

Array & operator=(const Array & a) {
    copyFrom(a);
    return *this;
}
```

6.2 The Big 5

- Destructor
- Copy constructor
- Overloaded operator=
- Copy constructor from *r-value*
- Overloaded operator= from *r-value*

Destructor

```
~Array() {
    if (data != nullptr) {
        for (int i = 0; i < length; i++) { // n times
            delete data[i];                // 1 step
        }
        delete[] date;                     // 1 step
        data = nullptr;                    // 1 step
    }
}
```

Complexity: $\mathcal{O}(n)$

Access Operator[]

```
const double & operator[](int idx) const {
    if (idx < length && idx >= 0)
        return data[idx];
    throw runtime_error("bad idx");
}
```

- Declares read-only access/
- Automatically selected by the compiler when an array being access is marked `const`
- Helps compiler optimize code for speed
- Some functions, like `ostream &operator<<(ostream &os, const Array &a)` require `const`.

- Must also make a non-constr version

Access with more than 2 dimensions

```
const double &operator()(int i, int j) const {
    // return by const reference
}
```

- Can't overload `operator[][]`

Inserting an Element

```
bool insert(int index, double val) {
    if (index >= size || index < 0) {
        return false;
    } else {
        for (int i = size - 1; i > index; --i) {
            data[i] = data[i - 1];
        }
        data[index] = val;
        return true;
    }
}
```

Complexity:

- Best case: $\mathcal{O}(1)$
  - The element is inserted at the end
- Average case: $\mathcal{O}(n)$
  - Go through half of elements (but that is a constant times n)
- Worst case: $\mathcal{O}(n)$
  - Go through all elements

Append

When array is full, resize:

- Double array size from n to $2n$
- Copy n items from the original array to the new array
- Appending n elements
- Total: $1 + n + n = 2n + 1$ steps
- Amortized: $(2n + 1)/n = \mathcal{O}(1)$ steps

6.3 Linked Lists and Iterators

- **Linked list**: Each person points to the next person
- **Doubly linked list**: Each person points to each other
- **Circularly linked list**: A list which has the first and last nodes pointing to each other

Arrays vs Linked Lists

Arrays:

- Access:
  - Random: $\mathcal{O}(1)$ time
  - Sequential: $\mathcal{O}(1)$ time
- Insert:
  - Insert: $\mathcal{O}(n)$ time
  - Append: $\mathcal{O}(n)$ time
- Book-keeping:
  - `ptr` to beginning
  - `current_size` or `ptr`
  - `max_size` or `ptr` to end off allocated space
- Memory:
  - Wastes memory if size is too large
  - Requires reallocation if too small

Linked Lists:

- Access:
  - Random: $\mathcal{O}(n)$ time
  - Sequential: $\mathcal{O}(1)$ time
- Insert:
  - Insert: $\mathcal{O}(n)$ time
  - Append: $\mathcal{O}(1)$ time
    - Append to `tail_ptr`
- Book-keeping:
  - `head_ptr` to first node
  - `size` (optional)
  - `tail_ptr` to last node
  - In each node, `ptr` to next node

  - Wasteful for small data times (overhead on pointers)
- Memory:
  - Allocates memory as needed
  - Requires memory for pointers

6.4 Linked List Implementation

```
class LinkedList {
 public:
    LinkedList();
    ~LinkedList();
 private:
    struct Node {
        double item;
        Node *next;
        Node() {next = nullptr;}
    };

    Node *head_ptr;
};
```

Methods

```
int size() const;

bool append_item(double item);
bool append_node(Node *n);

bool delete_item(double item);
bool delete_node(Node *n);
```

Maintaining Consistency

Arrays:

- Stored size matches number of elements at all times
- Be sure that `start_ptr + size < end_ptr`

Linked Lists

- Stored size matches number of elements
- The last node points to `nullptr`
- If the list is a circular list, last node points to head
- If a doubly-linked list, next/prev pointers are consistent

7 The Standard Template Library

The C++ Standard Template Library, included in C++11, is a high-quality library of implementations of the best algorithms and data structures at your fingertips (with plenty of documentation). The implementations are entirely in .h files, so there is no linking necessary.

Some things contained in the STL:

- Containers and iterators
- Memory allocators
- Utilities and function objects
- Algorithms

7.1 Benefits

- **DIY implementation is difficult**: introsort, red-black trees, hash-tables, mergesort... you don't have to know how they work to use them.
- **Uniformity**: a lot of the algorithms and data structures are templated, so their interfaces are relatively similar.
- **Saves a lot of debugging time**: greater than half of development time is spent testing and debugging, so if you don't have to do something yourself then you can save a lot of time.

7.2 Drawbacks

- Sometimes, more specific implementations may be faster. Often, if you know that you're not going to encounter a test case, then you can make trade-offs that can make your code faster in some way.
- You need to understand the library well to fully utilize it. The library has specific implementations of data structures and algorithms, and you have to know the complexity of operations - how well they perform.

The STL uses a lot of C++ features in its implementation, including:

- Type `bool`
- `const`-correctness and `const`-casts
- Namespaces
- Templates
- Inline functions
- Exception handling
- Keywords `explicit` and `mutable`

And so on. You do not, however, need to know what these features do to use their power.

It's nearly impossible to memorize the entire STL. It's not even necessary. Instead, it's helpful to know what's out there, and how to look things up when you need them.

7.3 Generic Programming

A lot of the STL minimizes use of pointers and dynamic memory allocation, so the debugging time is greatly decreased.

Also, since everything is templated, a lot of the same algorithms can be used with multiple data structures!

7.4 Complexity

Most STL implementations have the best possible big-O complexities, given their interface. There are two notable exceptions:

- `nth_element()`
- Linked lists

7.5 Examples of Container

Miscellaneous:

```
vector<>
deque<>
bit_vector<>  // same as vector<bool>
set<>
multi_set<>
map<>
multi_map<>
list<>
array<>
```

Linked list containers:

```
list<>          // doubly linked, .size() in O(1)
slist<>         // singly linked, .size() in O(n)
forward_list<>  // singly linked, .size() does not exist
```

7.6 Copying and Sorting

Do this:

```
#include <algorithm>

copy(vec.begin(), vec.end(), arr); // copy over
copy(arr, arr + SIZE, vec.begin()); // copy back

sort(arr, arr + SIZE);
sort(v.begin(), v.end());
```

Not this:

```
auto it = vec.begin();
int i = 0;
```

```
while (it != vec.end()) {
    arr[i] = *it;
    i++;
}
```

Or anything of the variety. Using builtins like `sort()` and `copy()` is a lot safer and in a more functional style.

7.7 Memory Allocation

Data structure	Memory overhead
vector	Compact
list	Not very compact
unordered_map	Memory hog

If memory is a worry, don't use an `unordered_map<>`.

7.8 Utilities and Functional Programming

- There are some functions that perform common operations, like `swap<>` and `max<>`.
- C++11 introduced lambdas instead of functors, which are basically anonymous functors.

```
void double_all(std::vector<int> & v) {
    std::for_each(v.begin(), v.end(), [](int in) {in *= 2;});
}
```

Pretty nifty, right? There is also the function `std::transform()`, which copies the lambda's return value to each element.

7.9 Sorting Custom Classes

Let's say, however, that you want to sort a custom class. Instead of overloading the `operator<()`, use a **functional object** as your comparison and then use standard library tools.

```
struct SortByName {
    bool operator()(const Employee & left,
                    const Employee & right) const {
        return left.getName() < right.getName();
    }
};
```

You can then use it like so:

```
vector<Employee> people(100);
// fill them
SortByName nameSort();
sort(people.begin(), people.end(), nameSort);
```

7.10 Generating Random Permutations

This is a little tidbit which is great for testing your program:

```
srand(time(nullptr));
vector<int> perm(N);

// fill it up
for (unsigned int i = 0; i < N; i++) {
    perm[i] = i;
}

random_shuffle(perm.begin(), perm.end());
```

You can also sort arrays by using:

```
random_shuffle(perm, perm + N);
```