Analysis of Algorithms
Introduction to Sorting

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, September 25, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Unit Overview
Recursion & Searching

Major Topics

- ✓ Introduction to recursion
- ✓ Search algorithms I
- ✓ Recursion vs. iteration
- ✓ Search algorithms II
- ✓ Eliminating recursion
- ✓ Search in the C++ STL
- ✓ Recursive backtracking

DONE

Review
Recursive Backtracking

We will review the *Recursive Backtracking* topic in a few days, when we discuss *Thoughts on Project 4*.

Unit Overview
Algorithmic Efficiency & Sorting

Major Topics

- Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- Asymptotic Notation
- Divide and Conquer
- Comparison Sorts II
- The Limits of Sorting
- Comparison Sorts III
- Non-Comparison Sorts
- Sorting in the C++ STL

Analysis of Algorithms

Analysis of Algorithms
Efficiency [1/3]

What do we mean by an **efficient** algorithm?

We mean an algorithm that **uses few resources**.

- A **resource** is something limited: something we would prefer to avoid using or doing a lot of.
- Typically, the most important resource is **time**.

So when we say an algorithm is **efficient**, *assuming we do not qualify this further*, we mean that it executes quickly.

How do we determine whether an algorithm is efficient?

- Implement it, and run the result on some computer?
- But the speed of computers is not fixed.
- And there are differences in compilers, etc.

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

Analysis of Algorithms
Efficiency [2/3]

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

Yes!

Idea

- View the tasks an algorithm performs as a series of steps.
- The steps we count are called **basic operations**.
- Determine the maximum number of basic operations required for input of a given size. Write this as a formula, based on the size of the input.
- Look at the most important part of the formula.

The most important part of $n \log_{10} n + 72n + 3n^2 + 945$ is $3n^2$.

See `important.py`*.*

We often do not care about the 3. The really important part is n^2.

Analysis of Algorithms
Efficiency [3/3]

When we talk about **efficiency** of an algorithm, without further qualification of what *efficiency* means, we are interested in:

- **Time** Used by the Algorithm
  - Expressed in terms of number of **basic operations**.
- How the **Size of the Input** Affects Running Time
  - Larger input typically means slower running time. How much slower?
- **Worst-Case** Behavior
  - What is the maximum number of basic operations the algorithm ever performs for a given input size?

In this context "real" time is called **wall-clock time**.

To make the above ideas precise, we need to say:

- What **operations** the algorithm is *allowed* to perform.
- Which operations we *count*: the **basic operations**.
- How we measure the **size** of the input.

These three form our **model of computation**.

Analysis of Algorithms
Model of Computation

Our *usual* model of computation will be as follows.

- Allowed **operations**:
  - Any operation that does *not* access the given data.
  - Client-provided functions for accessing the given data.
    - So data items that are objects are accessed only via their member functions and associated global functions.
- **Basic operations**:
  - Built-in operations on fundamental types (arithmetic, assignment, comparison, logical, bitwise, pointer, array look-up, etc.).
  - Calls to client-provided functions (including operators). In particular, in a template, operations (i.e., function calls) on template-parameter types.
- The functions we analyze will generally be given a list of items as input. The **size** of the input will be the number of items in the list.
  - The list could be an array, Linked List, range specified with iterators, etc.
  - We will generally denote the size of the input by *n*.

This is basically a fancy way of saying, "Do things as usual in C++."

We will also use other models of computation.

Analysis of Algorithms
Big-*O* Notation — Definition

Algorithm *A* is **order** $f(n)$ [written $O(f(n))$] if
there exist constants k and n_0 such that
algorithm *A* performs no more than $k \times f(n)$ basic operations
when given input of size $n \geq n_0$.

Constants k and n_0 are only there to make the definition work. We usually do not care about their values.

- So we do not worry much about whether some algorithm is (say) five times faster than another.
- We ignore small problem sizes.

Big-*O* is important!

A competent programmer is expected to understand it.

We will use it every single day for the rest of the semester.

Analysis of Algorithms
Big-*O* Notation — Worst Case & Average Case

When we use big-*O*, unless we say otherwise, we are always referring to the **worst-case** behavior of an algorithm.

- For input of a given size, what is the *maximum* number of steps the algorithm requires?

We can also do **average-case** analysis. However, if that is what we are doing, then we need to say so.

- Typically this involves finding the average number of basic operations over all inputs of a given size.
- Other kinds of averages are used—for example, the average number of basic operations over repeated calls to an algorithm. Again, if this is what we are doing, then we need to say so.

Analysis of Algorithms
Big-*O* Notation — Example 1, Problem

Determine the order of the following, and express it using big-*O*:

```
int func1(int arr[], int n)  // n = size of array arr
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return sum;
}
```

Continued …

Analysis of Algorithms
Big-*O* Notation — Example 1, Work

```
int func1(int arr[], int n)  // n = size of array arr
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return sum;
}
```

There are 9 different basic operations used in `func1`.

Total number of basic operations executed: $4n+6$.

Operation	Times Executed
`int arr[]`	1
`int n`	1
`int sum = 0`	1
`int i = 0`	1
`i < n`	$n + 1$
`++i`	n
`arr[i]`	n
`sum += …`	n
`return sum`	1
TOTAL	$4n + 6$

Continued …

Analysis of Algorithms
Big-O Notation — Example 1, Solution

Total number of basic operations executed by `func1`: $4n+6$.

Strictly speaking, it is correct to say that `func1` is $O(4n+6)$.

But in practice, we always place a function into one of a few commonly used categories.

Answer. Function `func1` is $O(n)$.

- This works with (for example) $k = 5$ and $n_0 = 100$.
- That is, $4n + 6 \leq 5n$, when $n \geq 100$.
- Intuitively, the most important part of $4n + 6$ is n.

Analysis of Algorithms
Big-*O* Notation — Example 1, Notes

```
int func1(int arr[], int n)  // n = size of array arr
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return sum;
}
```

Function `func1` is $O(n)$.

`sum += arr[i]` was 2 operations. What if we count it as just 1?

What if the loop as a whole is counted as 1 operation?

Conclusions

- Collapsing a *fixed* number of basic operations into 1 does not affect order.
- We can afford to be *somewhat* imprecise about what a basic operation is.

Analysis of Algorithms
Big-*O* Notation — Scalability

An algorithm (or function or technique or design …) that works well when used with increasingly large problems & large systems is said to be **scalable**. Or, it **scales well**.

This definition applies in general, not only in computing.

This course is all about things that scale well.

Information about efficiency, expressed using big-*O*, can tell us a lot about whether an algorithm is scalable.

Next we look at terminology we use to describe the efficiency & scalability of an algorithm.

Analysis of Algorithms
Big-O Notation — Efficiency Categories [1/2]

Know these!

An $O(1)$ algorithm is **constant time**.

- Its running time is essentially independent of its input.
- Such algorithms are rare, since they cannot read all of their input.

An $O(\log_b n)$ [for some $b > 1$] algorithm is **logarithmic time**.

- Again, such algorithms cannot read all of their input.
- As we will see, we do not care what b is.

An $O(n)$ algorithm is **linear time**.

- Such algorithms are not rare.
- This is as fast as an algorithm can be and still read all of its input.

An $O(n \log_b n)$ [for some $b > 1$] algorithm is **log-linear time**.

- This is about as slow as an algorithm can be and still be scalable.

An $O(n^2)$ algorithm is **quadratic time**.

- These are usually too slow for anything but very small data sets.

An $O(c^n)$ [for some $c > 1$] algorithm is **exponential time**.

- These algorithms are much too slow to be useful.

$\uparrow$ Faster

Slower $\downarrow$

Analysis of Algorithms
Big-*O* Notation — Efficiency Categories [2/2]

Faster ↑

Using Big-*O*	**In Words**
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(c^n)$, for some $c > 1$	Exponential time

↓ Slower

Cannot read all of input ↑ (above the line between $O(\log n)$ and $O(n)$)

Probably not scalable ↓ (below the line between $O(n \log n)$ and $O(n^2)$)

We are interested in the *fastest category* that an algorithm fits in.

- Every $O(1)$ algorithm is also $O(n^2)$ and $O(237^n)$; but $O(1)$ interests us most.

I will also allow $O(n^3)$, $O(n^4)$, etc., but we will not see these much.

Analysis of Algorithms
Big-*O* Notation — Example 2, Problem

Determine the order of the following, and express it with big-*O*:

```
int func2(int arr[], int n)  // n = size of array arr
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            sum += arr[j];
    return sum;
}
```

Continued …

Analysis of Algorithms
Big-O Notation — Example 2, Solution

In Example 2:

- There is a loop within a loop. The body of the inside (j) loop:

```
for (int j = 0; j < n; ++j)
    sum += arr[j];
```

- A single execution of this inside loop does $4n+2$ basic operations.
- However, the loop itself is executed n times by the outside (i) loop.
- Thus, a total of $n \times (4n+2) = 4n^2+2n$ basic operations are required.
- The rest of the function does $2n+6$ basic operations.
- Total: $(4n^2+2n) + (2n+6) = 4n^2+4n+6$ basic operations.
- Again, strictly speaking, it would be correct to say that `func2` is $O(4n^2+4n+6)$, but that is not how we do things.
- Instead, we note that, for large n, $4n^2+4n+6 \le 5n^2$. Thus, `func2` is $O(n^2)$: quadratic time.

Analysis of Algorithms
Big-*O* Notation — Example 3, Problem

Determine the order of the following, and express it using big-*O*:

```
int func3(int arr[], int n)  // n = size of array arr
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            sum += arr[j];
    return sum;
}
```

Notice!

Continued …

Analysis of Algorithms
Big-O Notation — Example 3, Solution

In Example 3:

- The number of basic operations done by the j loop is $4i+2$.
- So the total number of basic operations used by the j loop as i goes from 0 to $n-1$ is
  $2 + 6 + 10 + \ldots + [4(n-1)+2]$.
- Computing the sum, we obtain $[2 + 4(n-1)+2] \times n \div 2 = 2n^2$.
- As before, the rest of the function does $2n + 6$ basic operations.
- Total number of basic operations: $2n^2 + 2n + 6$.
- Thus the function is $O(n^2)$: quadratic time.

It is good to be *able* to perform analyses like these. But we would prefer easier methods.

Analysis of Algorithms
Big-*O* Notation — Rule of Thumb & Example 4

When computing the number of steps used by nested loops:

- For nested loops, each of which is either
  - executed *n* times, or
  - executed *i* times, where *i* goes up to *n*.
    - Or up to *n* plus some constant, or minus some constant.
- The order is $O(n^t)$ where *t* is the number of loops.

Example 4 (*n* is the size of the input)

```
for (int i = 0; i < n-4; ++i)
    for (int j = 0; j < i; ++j)
        for (int k = j; k < i+4; ++k)
            ++arr[k];
```

By the above rule of thumb, this has order $O(n^3)$.

- If we had to give this a name, we would call it "cubic time".

Analysis of Algorithms
Big-*O* Notation — Example 5

Find the order of the following code, where *n* is the size of the input.

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i-5; ++j)
        for (int k = 0; k < 5; ++k)
            ++arr[j+k];
```

Notice!

The *k* loop uses a **constant** number of operations.

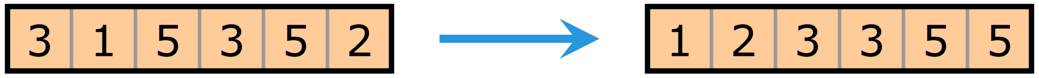
By the Rule of Thumb, this code has order $O(n^2)$.

In words: quadratic time.

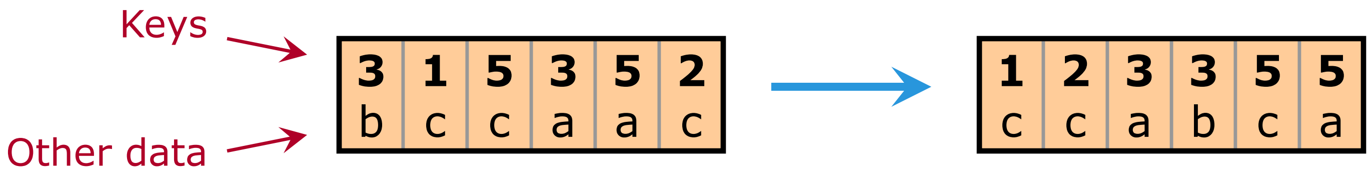
Introduction to Sorting

Introduction to Sorting
Basics [1/2]

To **sort** a collection of data means to rearrange its items in order.



Often the items we sort are themselves collections of data. The part we sort by is the **key**.



Efficient sorting is of great interest.

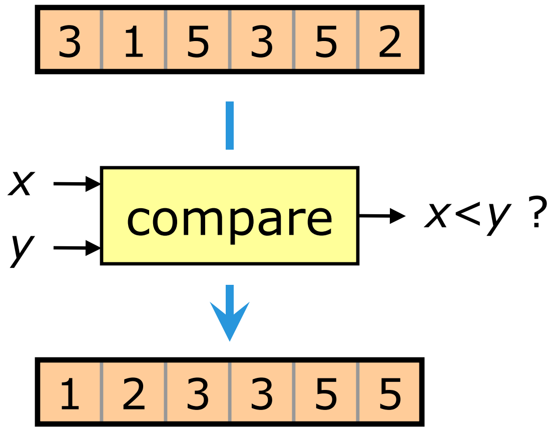
- Sorting is a very common operation.
- Sorting code that is written with little thought/knowledge is often much less efficient than code using a good algorithm.
- Some algorithms (like Binary Search) require sorted data. The efficiency of sorting affects the desirability of such algorithms.

Introduction to Sorting
Basics [2/2]

A **sort** (it is a noun here) is an algorithm that does sorting.

A **comparison sort** is a sort that only gets information about the data items in its input using a *comparison function*.

- A **comparison function** is a function that takes two data items and indicates which comes first. (Think "<".)
- When we study comparison sorts, we modify our model of computation: there are fewer legal operations.



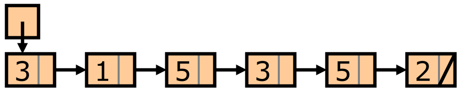
In the next few class meetings, we analyze various **general-purpose comparison sorts**.

- By **general purpose** I mean that we place no restrictions on the size of the list to be sorted, or the values in it. *This is my own definition; it is not standard terminology.*

Introduction to Sorting
Analyzing a Sort [1/2]

We analyze a general-purpose comparison sort using five factors.

- (Time) Efficiency
  - What is the (worst-case!) order of the algorithm?
  - What about **average case**—over all possible inputs of a given size?
- Requirements on Data
  - Does the algorithm require random-access data?
  - Does it work with Linked Lists?



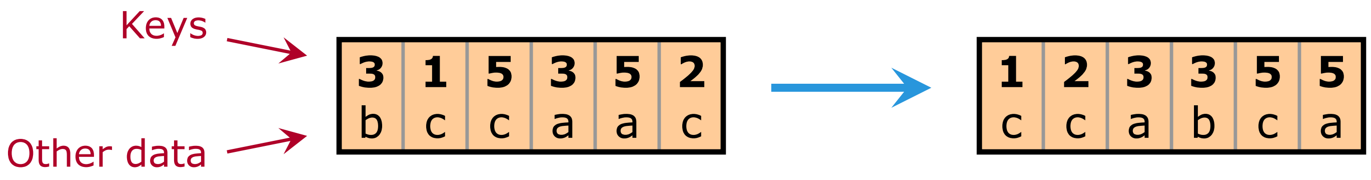
- Space Efficiency
  - Is the algorithm **in-place** (no large additional space required).
  - How much additional space (variables, buffers, etc.) is required?
- Stability
  - Is the algorithm **stable** (never reverses the relative order of equivalent items)?
- Performance on Nearly Sorted Data
  - **Nearly sorted**. Type 1: all items close to proper spots. Type 2: few items out of order.

"No large", "close", "few": at most a fixed constant.

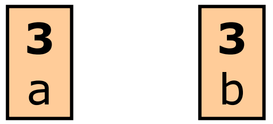
Introduction to Sorting
Analyzing a Sort [2/2]

Again, we say a sort is **stable** if it never reverses the relative order of equivalent items.

Recall the example used to illustrate what **keys** are.



The algorithm used to sort the above list is *not* stable, since the items at right, which are equivalent, had their relative order reversed.



Remember that we only compare *keys* when determining whether two items are equivalent.

Introduction to Sorting
Overview of Algorithms

There is no *known* sort that has all the properties we would like one to have.

We will examine a number of sorting algorithms. Most of these fall into two categories: $O(n^2)$ and $O(n \log n)$.

- Quadratic-Time [$O(n^2)$] Comparison Sorts
  - Bubble Sort
  - Insertion Sort
  - Quicksort
- Log-Linear-Time [$O(n \log n)$] Comparison Sorts
  - Merge Sort
  - Heap Sort (mostly later in semester)
  - Introsort
- Special Purpose—Not Comparison Sorts
  - Pigeonhole Sort
  - Radix Sort

It may seem odd that an algorithm called “Quicksort” is in the *slow* category. But this is not a mistake! *More about this in a few days.*