

Sorting & Searching

CP:AMA Readings: 9.6

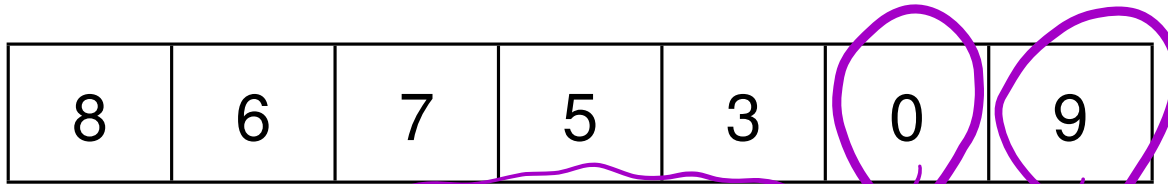
CHTP Readings: 6.8, 6.10, Appendix D

The primary goal of this section is to introduce you to sorting and searching methods and basic algorithm analysis.

Sorting

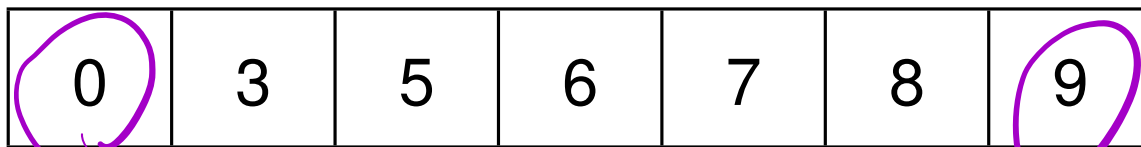
In its most basic form, the **sorting** problem is to take a list of elements and reorder the list so that the numbers appear in a sorted (*e.g.*, ascending) order.

An example list of integers:



8	6	7	5	3	0	9
---	---	---	---	---	---	---

The same list of integers in sorted (ascending) order:



0	3	5	6	7	8	9
---	---	---	---	---	---	---

Selection sort

In *selection sort*, the smallest element is *selected* to be the first element in the new sorted sequence, and then the next smallest element is selected to be the second element, and so on.

First, we find the position of the smallest element...

8	6	7	5	3	0	9
---	---	---	---	---	---	---

and then we *swap* the first element with the smallest.

0	6	7	5	3	8	9
---	---	---	---	---	---	---

Then, we find the next smallest element...

0	6	7	5	3	8	9
---	---	---	---	---	---	---

and then we *swap* that element with the second one, and so forth...

0	3	7	5	6	8	9
---	---	---	---	---	---	---

0 3 | 7 5 6 8 9
LHS is sorted
first pos
min

0 3 5 | 7 6 8 9
min

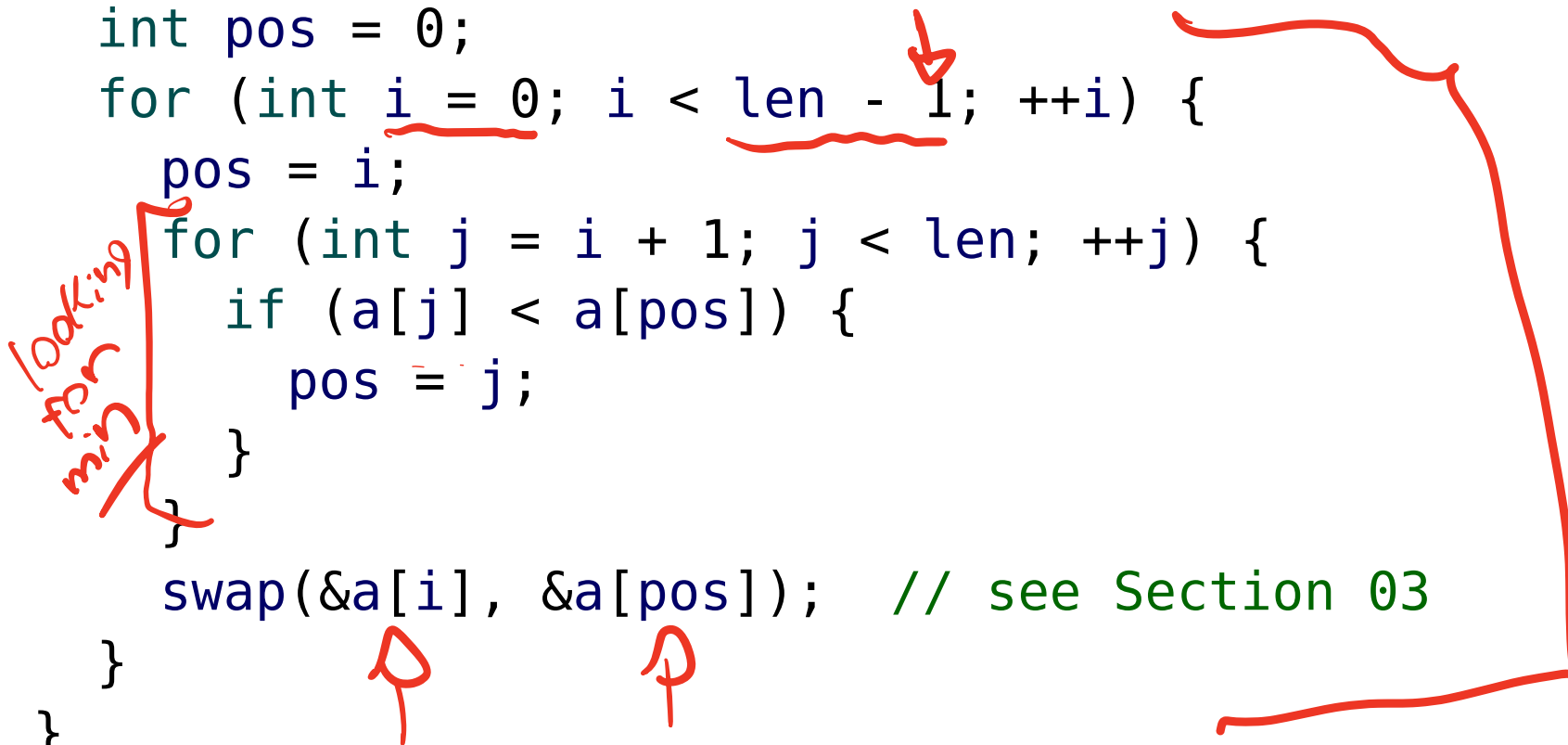
0 3 5 6 | 7 8 9
min

0 3 5 6 7 | 8 9
min

0 3 5 6 7 8 | 9

Done

```
void selection_sort(int a[], int len) {  
    int pos = 0;  
    for (int i = 0; i < len - 1; ++i) {  
        pos = i;  
        for (int j = i + 1; j < len; ++j) {  
            if (a[j] < a[pos]) {  
                pos = j;  
            }  
        }  
        swap(&a[i], &a[pos]); // see Section 03  
    }  
}
```



// Notes:

// i: loops from 0 ... len-2 and represents the
// "next" element to be replaced
// j: loops from i+1 ... len-1 and is "searching"
// for the next smallest element
// pos: position of the "next smallest"

Insertion sort

In *insertion sort*, we consider the first element to be a sorted sequence (of length one).

We then “insert” the second element into the existing sequence into the correct position, and then the third element, and so on.

For each iteration of *insertion sort*, the first i elements are sorted.

We then “insert” the element $a[i]$ into the correct position, moving all of the elements greater than $a[i]$ one to the right to “make room” for $a[i]$.

5 | 1 3 7

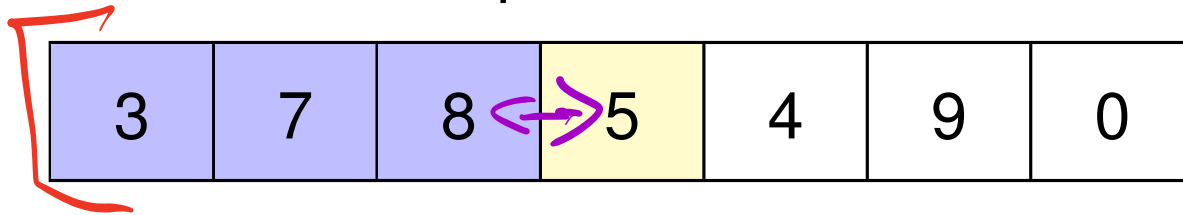


5 1

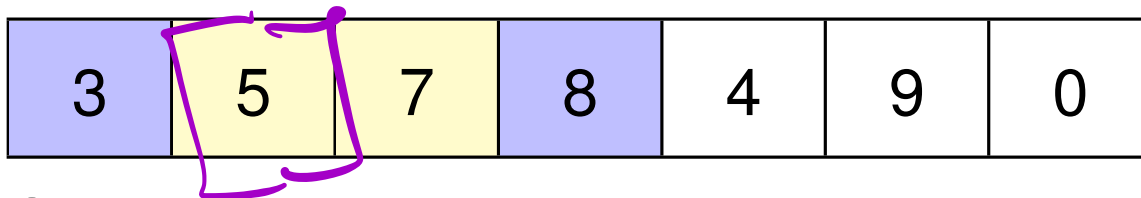
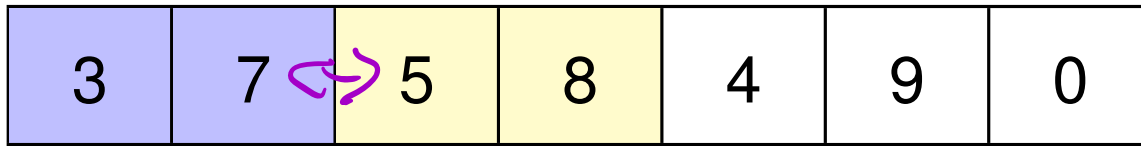
Swap

1 | 5 3 7

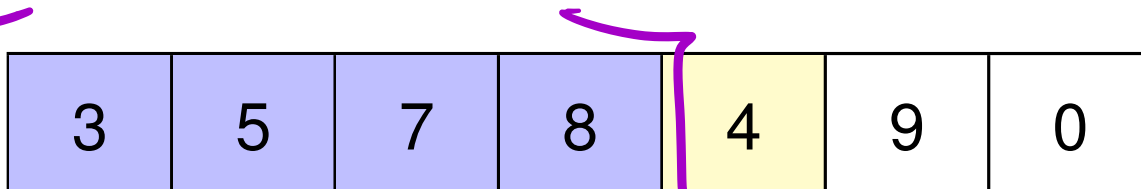
Consider an iteration of insertion sort ($i = 3$), where the first i (3) elements have been sorted. We want to *insert* the element at $a[i]$ into the correct position.



We continue to *swap* the element with the previous element until it reaches the correct position.



Once it is in the correct position, we start on the next element.



3 5 7 8 4 9 0

8 > 4
swap

3 5 7 4 8 9 0

7 > 4
swap

3 5 4 7 8 9 0

5 > 4
swap

3 4 5 7 8 9 0

3 < 4
X swap

3 4 5 7 8 9 0

$8 < 9$
~~X~~ swap

3 4 5 7 8 9 0

↑ ↓
 $9 > 0$
swap

3 4 5 7 8 0 9

↑ ↓
 $8 > 0$
swap

3 4 5 7 0 8 9

↑ ↓
 $7 > 0$
swap

3 4 5 0 7 8 9

↑ ↓
 $5 > 0$
swap

3 4 0 5 7 89

↗
9 > 0
swap

3 0 4 5 7 89

↗
3 > 0
swap

0 3 4 5 7 89

Done.

```

void insertion_sort(int a[], int len) {
    for (int i = 1; i < len; ++i) {
        for (int j = i; j > 0 && a[j - 1] > a[j]; --j) {
            swap(&a[j], &a[j - 1]);
        }
    }
}

```

make swap

// Notes:

```

// i: loops from 1 ... len-1 and represents the
//     "next" element to be replaced
// j: loops from i ... 1 and is "inserting"
//     the element that was at a[i] until it
//     reaches the correct position

```

The condition `a[j - 1] > a[j]` ensures the inner loop only runs while the element that was at `a[i]` is in an incorrect position.

Quicksort

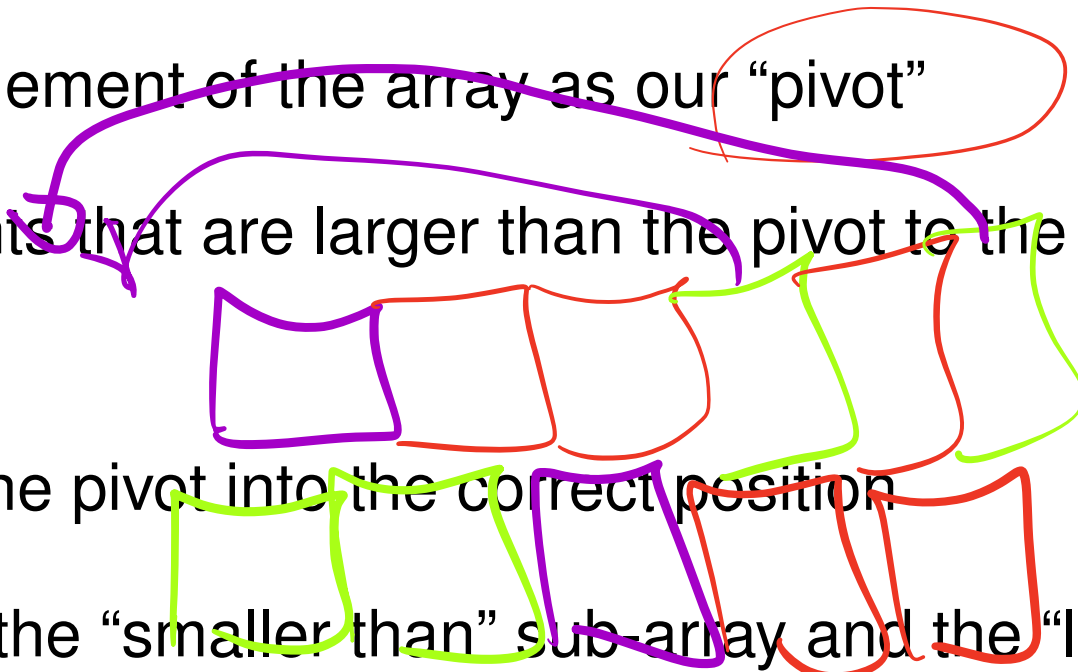
Quicksort is an example of a “divide & conquer” algorithm.

First, an element is selected as a “pivot” element.

The list is then **partitioned** (*divided*) into two sub-groups: elements *less than* (or equal to) the pivot and those *greater than* the pivot.

Finally, each sub-group is then sorted (*conquered*).

In our C implementation of quicksort, we:

- select the first element of the array as our “pivot”
 - move all elements that are larger than the pivot to the back of the array
 - move (“swap”) the pivot into the correct position
 - recursively sort the “smaller than” sub-array and the “larger than” sub-array
- 

To simplify the recursive calls, a helper quicksort function `quicksort_range` has parameters for the range of elements (`first` and `last`) to be sorted. A “wrapper” `quicksort` function just takes the array and its length.

```

void quicksort_range(int a[], int first, int last) {
    if (last <= first) return; // base case: length is <= 1

    int pivot = a[first] // first element is the pivot
    int pos = last;      // where to put next larger element

```

// move items greater than the pivot to the back

```

for (int i = last; i > first; --i) {
    if (a[i] > pivot) {
        swap(&a[pos], &a[i]);
        --pos;
    }
}

```

```

swap(&a[first], &a[pos]); // put pivot in correct place
// recursive calls to quicksort

```

```

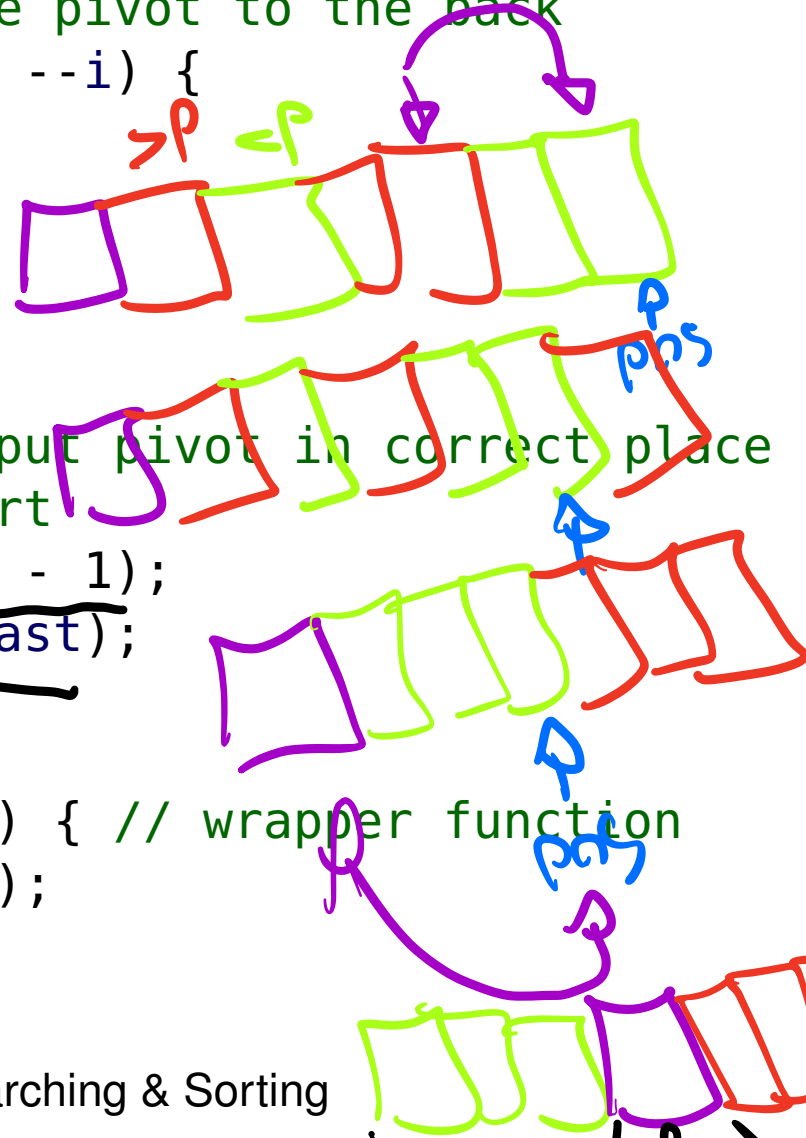
quicksort_range(a, first, pos - 1);
quicksort_range(a, pos + 1, last);
}

```

```

void quicksort(int a[], int len) { // wrapper function
    quicksort_range(a, 0, len - 1);
}

```

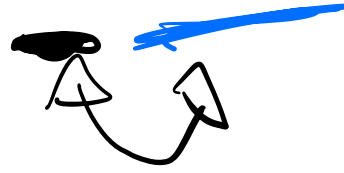


P 5 3 7 8 4 9 0

(9 > 5)? ✓

swap 9 & position (0)

sort sort



P 5 3 7 8 4 0 9

(4 > 5)? X

P 5 3 7 8 4 0 9

(8 > 5)? ✓

swap (8 ↔ 0)

^p [5] 3 7 0 4 8 9

$(7 > 9)? \checkmark$

Swap $(7 \leftrightarrow 4)$

^p [5] 3 4 0 7 8 9

$(3 > 5)? X$

No swap

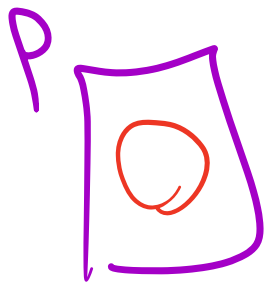
^p [5] 3 4 0 7 8 9

Swap (outside loop)

[0 3 4] [5] [7 8 9]

Recursive
Step LHS

Recursive
Step RHS



3

↕
4

4

<
<
<
<
<



8

9

9

$(4 > 0)? \checkmark$
swap $(4 \leftrightarrow 0)$

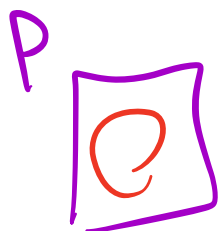


↕
3

4

3

$(3 > 0)? \checkmark$
swap $(3 \leftrightarrow 0)$

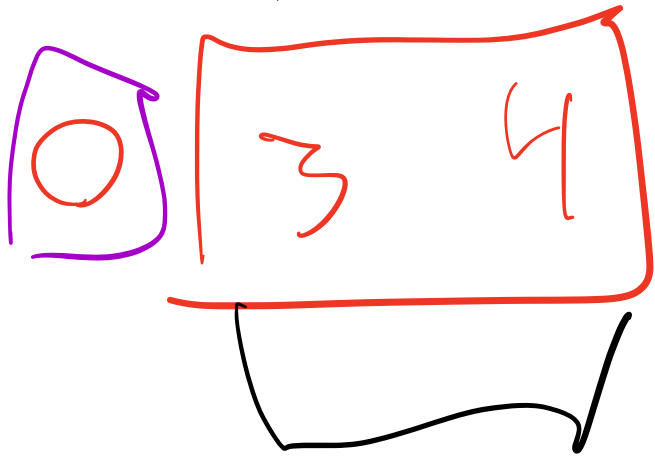


3

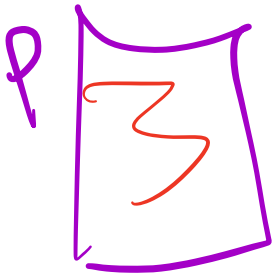
4

↕
swap

$(0 \rightarrow 0)$



call
recursively



4

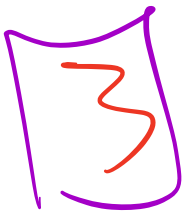
~~4~~

~~4~~

4

$(4 > 3)? \checkmark$

swap (4 → 4)

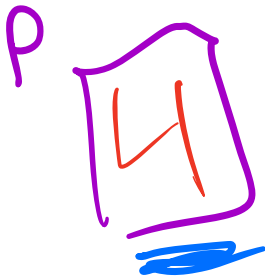


4

~~4~~ swap pivot
and pos



Recursive
call



Base
case

end

Searching

In its most basic form, the *searching* problem is to determine if a certain element appears in a given list.

We will write a C function that accepts an array and an element and determines if the element appears in the array.

If the element appears in the array, the function will return an index that it appears at. If the element does not appear, it will return -1 .

Searching examples

Searching for 3 in

0	1	2	3	4	5	6
8	6	7	5	3	0	9

would

return 4.

Searching for 4 in

8	6	7	5	3	0	9
---	---	---	---	---	---	---

would

return -1.

Not found
return -1

If the element appears multiple times, we will return an arbitrary index that it appears at. For example, searching for 5 in

5	6	7	5	3	0	9
---	---	---	---	---	---	---

could return 0 or 3.

Depends
on alg.

Linear search

```
// find(item, a, len) finds an index of item in array a;  
// returns -1 if item does not exist in a  
// requires: a is of length len
```

```
int find(int item, const int a[], int len) {  
    for (int i = 0; i < len; ++i) {  
        if (a[i] == item) {  
            return i;  
        }  
    }  
    return -1;  
}
```


Find 4

3₀ 5₁

!= 4

↓
!= 4

7₂

↓
!= 4

8₃

↓
!= 4

4₄

↓
== 4
return
index 4

9

4

The clock game

The game show *The Price is Right* has a game where players have to determine the price of a prize by making guesses and only being told if their guesses were too high or low.



The clock game strategy

How can you minimize the number of guesses you make? The prize must be between \$1 and \$999.

One possible strategy: Guess the middle of this range (\$500). If you are too low the prize is between \$501 and \$999; if you are too high the prize is between \$1 and \$499.

Continue guessing the middle of the possible range until you find the correct price.

This strategy is known as **binary search** because each guess splits the remaining possible range exactly in two.

Binary search

If the array is sorted, we can use binary search to find an item:

```
// requires: a is sorted in ascending order
int find_sorted(int item, const int a[], int len) {
    int low = 0;
    int high = len - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == item) {
            return mid;
        } else if (a[mid] < item) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

↳ Must
be sorted

We will see this is more *efficient* than linear search.

3 5 7 8 4 9 0
Find 7

less than 500
500

1000

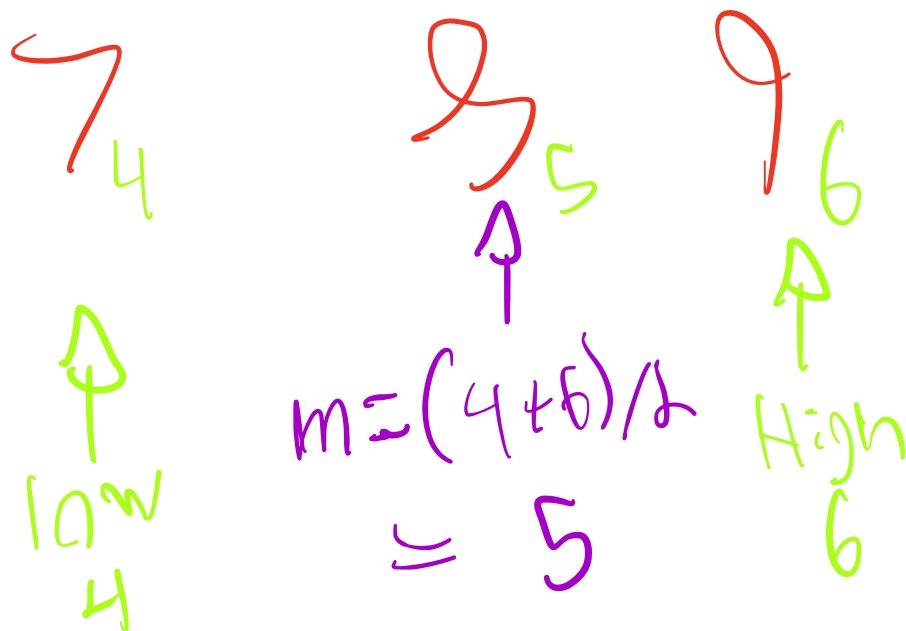
Not Sorted

0 3 4 5 7 8 9
low high
0 6

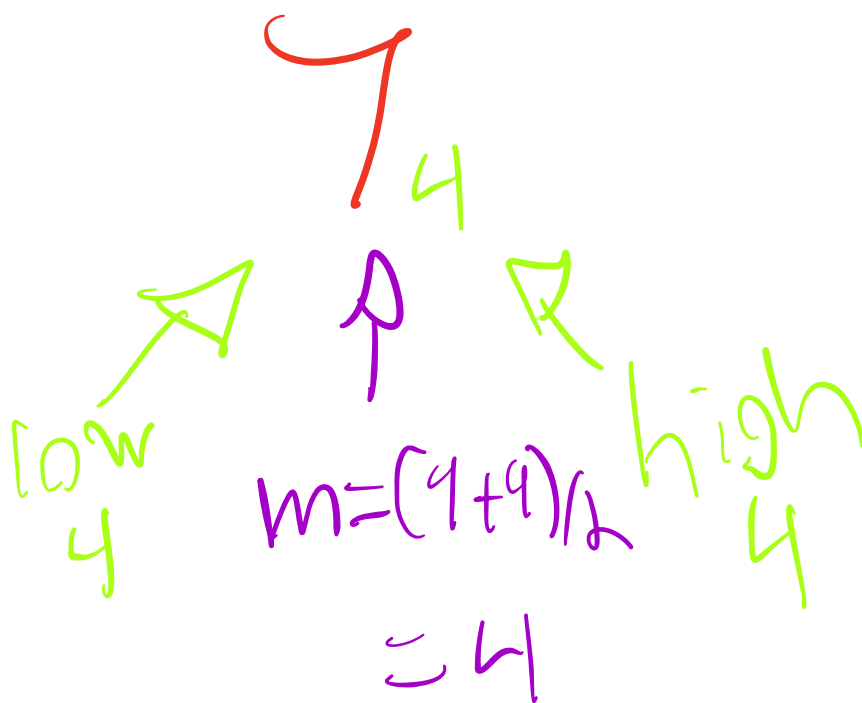
low
(index) 0

$$m = (low + high) / 2 \\ = (0 + 6) / 2 \\ = 3$$

$(5 < 7) \Rightarrow$ Search RHS



$(2 \rightarrow 7)$
 \Rightarrow Search
 LHS



$(7 == 7)$

\Rightarrow return 4

~~Done~~

Algorithms

An *algorithm* is step-by-step description of *how* to solve a “problem”.

Algorithms are not restricted to computing. For example, every day you might use an algorithm to select which clothes to wear.

For most of this course, the “problems” are function descriptions (*interfaces*) and we work with *implementations* of algorithms that solve those problems.

The word *algorithm* is named after Muḥammad ibn Mūsā al-Khwārizmī (\approx 800 A.D.).

There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How accurate is it?
- How robust is it? (Can it handle errors well?)
- How adaptable is it? (Can it be used to solve similar problems?)
- **How fast (efficient) is it?**

In this course, we use *efficiency* to objectively compare algorithms.

Efficiency

The most common measure of efficiency is *time efficiency*, or **how long** it takes an algorithm to solve a problem. Unless we specify otherwise, we **always mean *time efficiency***.

Another efficiency measure is *space efficiency*, or how much space (memory) an algorithm requires to solve a problem. *Power efficiency* (power consumption) is also becoming an important measure.

The *efficiency* of an algorithm may depend on its *implementation*.

Running time

To *quantify* efficiency, we are interested in measuring the **running time** of an algorithm.

What **unit of measure** should we use? Seconds?

“My algorithm can sort one billion integers in 9.037 seconds”.

- What *year* did you make this statement?
- What machine & model did you use? (With how much RAM?)
- What computer language & operating system did you use?
- Was that the actual CPU time, or the total time elapsed?
- How accurate is the time measurement? Is the 0.037 relevant?

Measuring *running times* in seconds can be problematic.

What are the alternatives?

Typically, we measure the number of **elementary operations** required to solve the problem.

In C, we can count the number of operations, or in other words, the number of *operators* executed.

You are not expected to count the exact number of operations.

We only count operations in these notes for illustrative purposes.

We introduce some simplification shortcuts soon.

Data size

What is the number of operations executed for this implementation?

```
int sum_array(const int a[], int len) {  
    int sum = 0;  
    int i = 0;  
    while (i < len) {  
        sum = sum + a[i];  
        i = i + 1;  
    }  
    return sum;  
}
```

Handwritten analysis of the code:

- $(n+1)$ for the `while` loop condition.
- $(n+1)$ for the `sum = sum + a[i];` operation.
- $2n$ for the `i = i + 1;` operation.
- $3n$ for the `sum = sum + a[i];` operation.
- $1 + 1 + 2(n+1) + 3n + 2n$ for the total number of operations.
- $T(n) = 7n + 4$ for the total number of operations.

The running time **depends on the length** of the array.

If there are n items in the array, it requires $7n + 4$ operations.

We are always interested in the running time *with respect to* the **size of the data**.

Traditionally, the variable n is used to represent the **size** (or **length**) of the data. m and k are also popular when there is more than one parameter.

Often, n is obvious from the context, but if there is any ambiguity clearly state what n represents.

For example, with lists of strings (we'll see this in the next section), n may represent the number of strings in the list, or it may represent the length of all of the strings in the list.

The *running* ***Time*** of an implementation is a **function** of n and is often written using functional notation such as $T(n)$.

There may also be another *attribute* of the data that is also important.

For example, with *trees*, we use n to represent the number of nodes in the tree and h to represent the *height* of the tree.

In advanced algorithm analysis, n may represent the number of *bits* required to represent the data, or the length of the *string* necessary to describe the data.

Algorithm comparison

Problem: Write a function to determine if an array of positive integers contains at least **e** even numbers and **o** odd numbers.

```
// check_array(a, len, e, o) determines if array a
//   contains at least e even numbers and
//   at least o odd numbers
// requires: len > 0
//           elements of a > 0
//           e, o >= 0
```

Homer and Bart are debating the best algorithm (strategy) for implementing **check_array**.

Bart just wants to count the total number of odd numbers in the entire array.

```
bool bart(const int a[], int len, int e, int o) {
    int odd_count = 0;
    for (int i = 0; i < len; i = i + 1) {
        odd_count = odd_count + (a[i] % 2);
    }
    return (odd_count >= o) && (len - odd_count >= e);
}
```

If there are n elements in the array, $T(n) = 8n + 7$.

$$T(n) = 1 + 2(n+1) + 1 + 2n + 4n + 4$$

$$= 8n + 8$$

Remember, you are not expected to calculate this precisely.

Homer is lazy, and he doesn't want to check all of the elements in the array if he doesn't have to.

```
bool homer(const int a[], int len, int e, int o) {  
    // only loop while it's still possible  
    while (len > 0 && e + o <= len) {  
        if (a[len - 1] % 2 == 0) { // even case:  
            if (e > 0) {  
                e = e - 1; // only decrement e if e > 0  
            }  
        } else if (o > 0) {  
            o = o - 1;  
        }  
        if (e == 0 && o == 0) {  
            return true;  
        }  
        len = len - 1;  
    }  
    return false;  
}
```

The problem with analyzing Homer's code is that it depends not just on the length of the array, but on the contents of the array and the parameters e and o .

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
// these are fast:  
bool fast1 = homer(a, 10,  $e$ , 11); // false;  
bool fast2 = homer(a, 10, 1,  $o$ ); // true;  
  
// these are slower:  
bool slow1 = homer(a, 10, 5, 5); // true;  
bool slow2 = homer(a, 10, 6, 4); // false;
```

For Homer's code, the **best case** is when it can **return** immediately, and the **worst case** is when *all* of the array elements are visited.

For Bart's code, the best case is the same as the worst case.

homer $T(n) = 4$ (best case)

$T(n) = 17n + 1$ (worst case)

bart $T(n) = 8n + 9$ (all cases)

Which implementation is more efficient?

Is it more “fair” to compare against the best case or the worst case?

Worst case running time

Typically, we want to be conservative (*pessimistic*) and use the *worst case*.

Unless otherwise specified, the running time of an algorithm is the **worst case running time**.

Comparing the worst case, Bart's implementation ($8n + 7$) is more efficient than Homer's ($17n + 1$).

We may also be interested in the *average* case running time, but that analysis is typically much more complicated.

Big O notation

In practice, we are not concerned with the difference between the running times $(8n + 7)$ and $(17n + 1)$.

$O(n)$

We are interested in the order of a running time. The order is the “**dominant**” term in the running time **without any constant coefficients**.

The dominant term in both $(8n + 7)$ and $(17n + 1)$ is n , and so they are both “*order n*”.

To represent *orders*, we use **Big O notation**.

Instead of “*order n*”, we use $O(n)$.

The “dominant” term is the term that *grows* the largest when n is very large ($n \rightarrow \infty$). The *order* is also known as the “*growth rate*”.

In this course, we encounter only a few orders
(arranged from smallest to largest):

$$O(1) \prec O(\log n) \prec O(n) \prec O(n \log n) \prec O(n^2) \prec O(n^3) \prec O(2^n)$$

example: orders

- $2016 = O(1)$

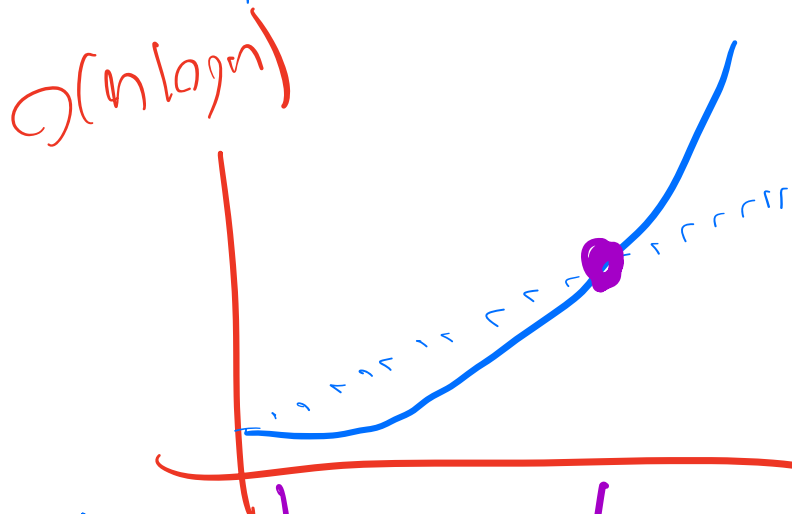
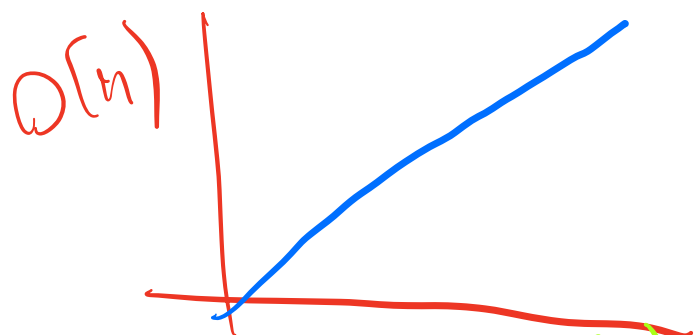
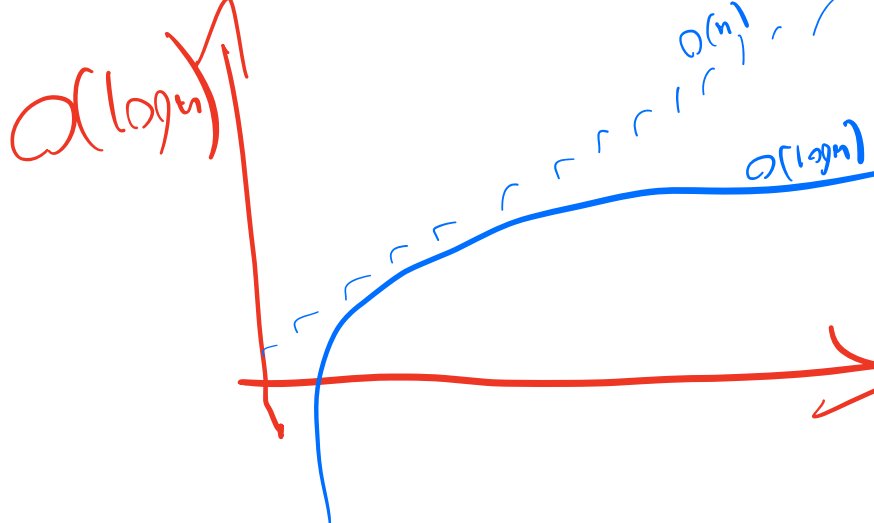
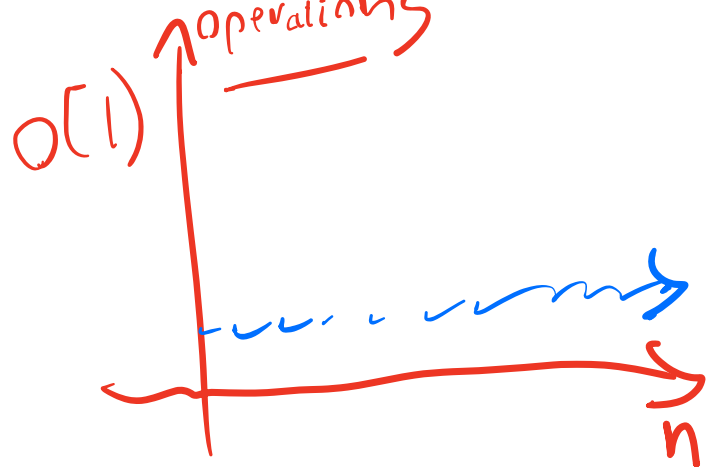
- $100000 + n = O(n)$

- $n + n \log n = O(n \log n)$

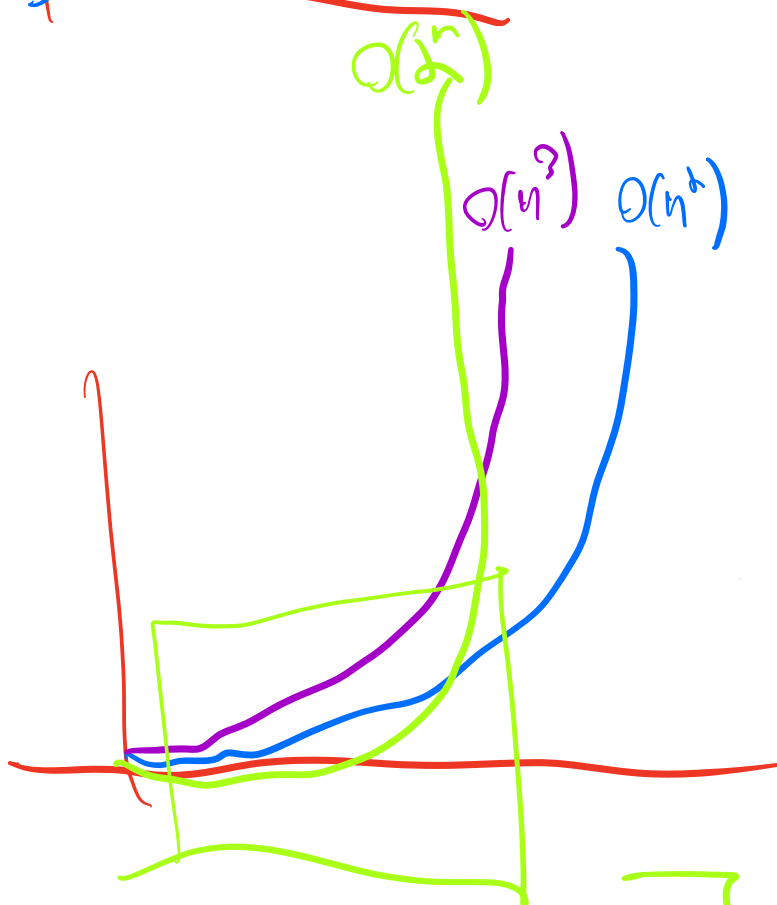
- $999n + 0.01n^2 = O(n^2)$

- $\frac{n(n+1)(2n+1)}{6} = O(n^3)$

- $n^3 + 2^n = O(2^n)$



Over time
 $n \log n > n$



Over time
 $2^n > n^3$

Ex. $n = 9$
 $2^9 = 512$
 $9^3 = 729$

When comparing algorithms, the most efficient algorithm is the one with the lowest *order*.

For example, an $O(n \log n)$ algorithm is more efficient than an $O(n^2)$ algorithm.

If two algorithms have the same *order*, they are considered **equivalent**.

Both Homer's and Bart's implementations are $O(n)$, so they are equivalent.



Big O arithmetic

When *adding* two orders, the result is the largest of the two orders.

- $\underbrace{O(\log n)} + \underbrace{O(n)} = \underbrace{O(n)}$
- $\underbrace{O(1)} + \underbrace{O(1)} = \underbrace{O(1)}$

When *multiplying* two orders, the result is the product of the two orders.

- $O(\log n) \times O(n) = O(n \log n)$
- $O(1) \times O(n) = O(n)$

In many textbooks, **and in this introductory course**, the notation

$$T(n) = 1 + 2n + 3n^2 = O(1) + O(n) + O(n^2) = O(n^2)$$

is acceptable.

In other textbooks, and in other courses, this notation may be too informal.

Algorithm analysis

An important skill in Computer Science is the ability to *analyze* a function and determine the *order* of the running time.

In this course, our goal is to give you experience and work toward building your intuition:

```
int sum_array(const int a[], int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i) {  
        sum += a[i];  
    }  
    return sum;  
}
```

$O(n)$

“Clearly, each element is visited once, so the running time of `sum_array` is $O(n)$ ”.

Analyzing simple functions

First, consider **simple** functions (without recursion or iteration).

```
int max(int a, int b) {  
    if (a > b) return a;  
    return b;  
}
```

$T(n) = 1 = O(1)$

$O(1)$

If no other functions are called, there must be a fixed number of operators. Each operator is $O(1)$, so the running time is:

$$\underbrace{O(1)}_1 + \underbrace{O(1)}_2 + \overset{\text{[fixed \# of times]}}{\dots} + O(1) = O(1)$$

If a simple function calls other functions, its running time depends on those functions.

Iterative analysis

Iterative analysis uses summations.

```
for (i = 1; i <= n; ++i) {  
    printf("*");  
}
```

n time

$$T(n) = \sum_{i=1}^n O(1) = \underbrace{O(1) + \dots + O(1)}_n = \underline{n} \times \underline{O(1)} = \underline{O(n)}$$

log, prob. $O(1) = O(1)$

Because we are primarily interested in *orders*,

$\sum_{i=0}^{n-1} O(n)$, $\sum_{i=1}^n O(n)$, or $\sum_{i=1}^n O(x)$ are equivalent* to $\sum_{i=1}^n O(n)$

* unless x is exponential (e.g., $O(2^i)$).

Procedure for iteration

1. Work from the *innermost* loop to the *outermost*
2. Determine the number of iterations in the loop (in the worst case) in relation to the size of the data (n) or an outer loop counter
3. Determine the running time per iteration
4. Write the summation(s) and simplify the expression

```
sum = 0;  
for (i = 0; i < n; ++i) {  
    sum += i;  
}
```

$$\sum_{i=1}^n O(1) = O(n)$$

Common summations

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

$$\sum_{i=1}^n O(1) = O(n)$$

$$\sum_{i=1}^n O(n) = O(n^2)$$

$$\sum_{i=1}^n O(i) = O(n^2)$$

$$\sum_{i=1}^n O(i^2) = O(n^3)$$

The summation index should reflect the *number of iterations* in relation to the *size of the data* and does not necessarily reflect the actual loop counter values.

```
k = n;           // n is size of the data
while (k > 0) {
    printf("*");
    k -= 10;
}
```

There are $n/10$ iterations. Because we are only interested in the *order*, $n/10$ and n are equivalent.

$$\sum_{i=1}^{n/10} O(1) = O(n)$$

When the loop counter changes *geometrically*, the number of iterations is often logarithmic.

```
k = n;           // n is size of the data
while (k > 0) {
    printf("*");
    k /= 10;
}
```

There are $\log_{10} n$ iterations.

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

When working with *nested* loops, evaluate the *innermost* loop first.

```
for (i = 0; i < n; ++i) {  
    for (j = 0; j < i; ++j) {  
        printf("*");  
    }  
    printf("\n");  
}
```

Inner loop: $\sum_{j=0}^{i-1} O(1) = O(i)$

Outer loop: $\sum_{i=0}^{n-1} (O(1) + O(i)) = O(n^2)$

Recurrence relations

To determine the running time of a recursive function we must determine the **recurrence relation**. For example,

$$T(n) = O(n) + T(n - 1)$$

We can then look up the recurrence relation in a table to determine the *closed-form* (non-recursive) running time.

$$T(n) = O(n) + T(n - 1) = O(n^2)$$

In later courses, you *derive* the closed-form solutions and *prove* their correctness.

The recurrence relations we encounter in this course are:

$$T(n) = O(1) + T(n - c) \qquad = O(n)$$

$$T(n) = O(n) + T(n - c) \qquad = O(n^2)$$

$$T(n) = O(n^2) + T(n - c) \qquad = O(n^3)$$

$$T(n) = O(1) + T\left(\frac{n}{d}\right) \qquad = O(\log n)$$

$$T(n) = O(1) + d \cdot T\left(\frac{n}{d}\right) \qquad = O(n)$$

$$T(n) = O(n) + d \cdot T\left(\frac{n}{d}\right) \qquad = O(n \log n)$$

$$T(n) = O(1) + T(n - c) + T(n - c') \qquad = O(2^n)$$

where $c, c' \geq 1$ and $d > 1$

This table will be provided on exams if necessary.

Procedure for recursive functions

1. Identify the order of the function *excluding* any recursion
2. Determine the size of the data for the next recursive call(s)
3. Write the full *recurrence relation* (combine step 1 & 2)
4. Look up the closed-form solution in a table

```
int sum(int n) {  
    if (n == 0) return 0;  
    return n + sum(n - 1);  
}
```

1. All non-recursive operations: $O(1)$ $+$, $-$, $==$
2. size of the recursion: $n - 1$
3. $T(n) = O(1) + T(n - 1)$ (combine 1 & 2)
4. $T(n) = O(n)$ (table lookup)

Revisiting sorting algorithms

No introduction to efficiency is complete without a discussion of **sorting algorithms**.

For simplicity, we only consider sorting **numbers**.

When sorting strings or large data structures, include the time to compare each element.

When analyzing sorting algorithms, one measure of running time is the number of comparisons.

Selection sort

Recall our C implementation of selection sort:

```
void selection_sort(int a[], int len) {  
    int pos = 0;  
    for (int i = 0; i < len - 1; ++i) {  
        pos = i;  
        for (int j = i + 1; j < len; ++j) {  
            if (a[j] < a[pos]) {  
                pos = j;  
            }  
        }  
        swap(&a[i], &a[pos]); // see Section 03  
    }  
}
```

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n O(1) = O(n^2)$$

Insertion sort

The analysis for the worst case of insertion sort is also $O(n^2)$.

```
void insertion_sort(int a[], int len) {  
    for (int i = 1; i < len; ++i) {  
        for (int j = i; j > 0 && a[j - 1] > a[j]; --j) {  
            swap(&a[j], &a[j - 1]);  
        }  
    }  
}
```

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i O(1) = O(n^2)$$

However, in the *best case*, the array is already sorted, and the inner loop terminates immediately. This best case running time is $O(n)$.

Quicksort

In our C implementation of quicksort, we:

1. select the first element of the array as our “pivot”. $O(1)$
2. move all elements that are larger than the pivot to the back of the array. $O(n)$.
3. move (“swap”) the pivot into the correct position. $O(1)$.
4. recursively sort the “smaller than” sub-array and the “larger than” sub-array. $T(?)$

The analysis of step 4 is a little trickier.

When the pivot is in “the middle” it splits the sublists equally, so

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

But that is the *best case*. In the worst case, the “pivot” is the smallest (or largest element), so one of the sublists is empty and the other is of size $(n - 1)$.

$$T(n) = O(n) + T(n - 1) = O(n^2)$$

Despite its worst case behaviour, quicksort is still popular and in widespread use. The average case behaviour is quite good and there are straightforward methods that can be used to improve the selection of the pivot.

It is part of the C standard library (`qsort` in `stdlib.h`).

Sorting summary

Algorithm	best case	worst case
selection sort	$O(n^2)$	$O(n^2)$
insertion sort	$O(n)$	$O(n^2)$
quicksort	$O(n \log n)$	$O(n^2)$

From this table, it might appear that insertion sort is the best choice.

However, as mentioned with quicksort, the “typical” or “average” case for quicksort is much better than insertion sort.

In Section 8, we will see merge sort, which is $O(n \log n)$ in the worst case.

Binary search

In this section we implemented binary search on a sorted array.

```
int find_sorted(int item, const int a[], int len) {  
    // ...  
    while (low <= high) {  
        mid = (low + high) / 2;  
        // ...  
        if (a[mid] < item) {  
            low = mid + 1;  
        } else {  
            high = mid - 1;  
        }  
    }  
    //...
```

In each iteration, the size of the search range ($n = \text{high} - \text{low}$) was halved, so the running time is:

$$T(n) = \sum_{i=1}^{\log_2 n} O(1) = O(\log n)$$

Algorithm design

In this introductory course, the algorithms we develop are mostly straightforward.

To provide some insight into *algorithm design*, we introduce a problem that is simple to describe, but hard to solve efficiently.

We present four different algorithms to solve this problem, each with a different running time.

example: the maximum subarray problem

Problem: Given an array of integers, find the **maximum sum** of any *contiguous* sequence (subarray) of elements.

For example, for the following array:

31	-41	59	26	-53	58	97	-93	-23	84
----	-----	----	----	-----	----	----	-----	-----	----

the maximum sum is 187:

31	-41	59	26	-53	58	97	-93	-23	84
----	-----	----	----	-----	----	----	-----	-----	----

This problem has many applications, including *pattern recognition* in *artificial intelligence*.

Solution A: $O(n^3)$

```
// for every start position i and ending position j
// loop between them (k) summing elements
int max_subarray(const int a[], int len) {
    int maxsofar = 0;
    int sum = 0;
    for (i = 0; i < len; ++i) {
        for (j = i; j < len; ++j) {
            sum = 0;
            for (k = i; k <= j; ++k) {
                sum += a[k];
            }
            maxsofar = max(maxsofar, sum);
        }
    }
    return maxsofar;
}
```

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j O(1) = O(n^3)$$

Solution B: $O(n^2)$

```
// for every start position i,  
// check if the sum from i...j is the max
```

```
int max_subarray(const int a[], int len) {  
    int maxsofar = 0;  
    int sum = 0;  
    for (i = 0; i < len; ++i) {  
        sum = 0;  
        for (j = i; j < len; ++j) {  
            sum += a[j];  
            maxsofar = max(maxsofar, sum);  
        }  
    }  
    return maxsofar;  
}
```

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n O(1) = O(n^2)$$

Solution C: $O(n \log n)$

We only describe this recursive *divide and conquer* approach.

1. Find the midpoint position m . $O(1)$
2. Find (a) the maximum subarray from $(0 \dots m-1)$, and
(b) the maximum subarray from $(m+1 \dots \text{len}-1)$. $2T(n/2)$
3. Find (c) the maximum subarray that includes m . $O(n)$
4. Find the maximum of (a), (b) and (c). $O(1)$

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

Solution D: $O(n)$

```
// for each position i, keep track of  
// the maximum subarray ending at i
```

```
int max_subarray(const int a[], int len) {  
    int maxsofar = 0;  
    int maxendhere = 0;  
    for (i = 0; i < len; ++i) {  
        maxendhere = max(maxendhere + a[i], 0);  
        maxsofar = max(maxsofar, maxendhere);  
    }  
    return maxsofar;  
}
```

In this introductory course, you are not expected to be able to come up with this solution yourself.

Big O revisited

We now revisit *Big O notation* and define it more formally.

$O(g(n))$ is the **set** of all functions whose “order” is **less than or equal** to $g(n)$.

$$n^2 \in O(n^{100})$$

$$n^3 \in O(2^n)$$

While you can say that n^2 is in the set $O(n^{100})$, it's not very useful information.

In this course, we always want the **most appropriate** order, or in other words, the *smallest* correct order.

Big O describes the *asymptotic* behaviour of a function.

This is **different** than describing the **worst case** behaviour of an algorithm.

Many confuse these two topics but they are completely **separate concepts**. The best case and the worst case can both be described asymptotically.

For example, the best case insertion sort is $O(n)$, while the worst case is $O(n^2)$.

A slightly more formal definition of Big O is

$$f(n) \in O(g(n)) \Leftrightarrow f(n) \leq c \cdot g(n)$$

for large n and some positive number c

This definition makes it clear why we “*ignore*” constant coefficients.

For example,

$$9n \in O(n) \quad \text{for } c = 10, \quad 9n \leq 10n, \text{ and}$$

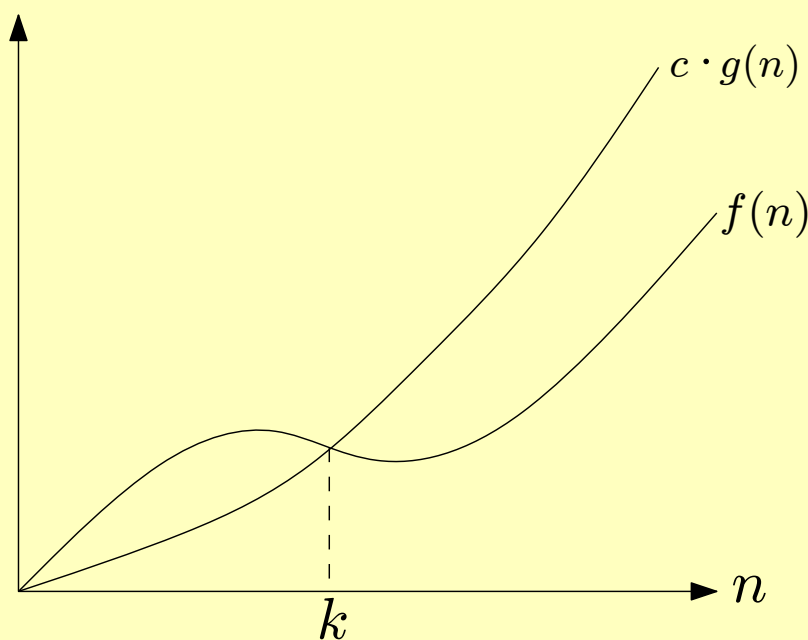
$$0.01n^3 + 1000n^2 \in O(n^3)$$

$$\text{for } c = 1001, \quad 0.01n^3 + 1000n^2 \leq 1001n^3$$

The full definition of Big O is

$$f(n) \in O(g(n)) \Leftrightarrow \exists c, k > 0, \forall n > k, f(n) \leq c \cdot g(n)$$

$f(n)$ is in $O(g(n))$ if there exists a positive c and k such that for any value of $n > k$, $f(n) \leq c \cdot g(n)$.



In later CS courses, you will use the formal definition of Big O to *prove* algorithm behaviour more rigourously.

There are other asymptotic functions in addition to Big O.

(for each of the following, $\exists k > 0, \forall n > k \dots$)

$$f(n) \in \omega(g(n)) \Leftrightarrow \forall c > 0, c \cdot g(n) \leq f(n)$$

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists c > 0, c \cdot g(n) \leq f(n)$$

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 > 0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, f(n) \leq c \cdot g(n)$$

$$f(n) \in o(g(n)) \Leftrightarrow \forall c > 0, f(n) \leq c \cdot g(n)$$

$O(g(n))$ is often used when $\Theta(g(n))$ is more appropriate.

Goals of this Section

At the end of this section, you should be able to:

- describe selection sort, insertion sort, and quicksort
- describe linear search
- describe binary search on a sorted array
- use the new terminology introduced (*e.g.*, algorithm, time efficiency, running time, order)
- compute the order of an expression
- explain and demonstrate the use of Big O notation and how n is used to represent the size of the data

- determine the “worst case” running time for a given implementation
- analyze a recursive function, determine its recurrence relation and look up its closed-form running time in a provided lookup table
- analyze an iterative function and determine its running time