Analyzing Algorithms

How do we know if our code is good at it's job?

Tim Jackman

BU Summer Challenge

July 15th, 2025

Abundant Algorithms

- So far we've talked about how to implement algorithms into computer programs in order to solve problems

Abundant Algorithms

- So far we've talked about how to implement algorithms into computer programs in order to solve problems
- But there are many algorithms to solve the same problem

Abundant Algorithms

- So far we've talked about how to implement algorithms into computer programs in order to solve problems
- But there are many algorithms to solve the same problem
- How do we assess what algorithms are "better" than others?

Abundant Algorithms

- So far we've talked about how to implement algorithms into computer programs in order to solve problems
- But there are many algorithms to solve the same problem
- How do we assess what algorithms are "better" than others?
- There's not always one "best" algorithm for a problem. It often depends on our situation

Time and Space

- The most straightforward way to judge is on "running time": How long does it take the algorithm to run?

Time and Space

- The most straightforward way to judge is on "running time": How long does it take the algorithm to run?
- We can also judge them on "space": How many bits of memory does the algorithm take to run?

Time and Space

- The most straightforward way to judge is on “running time”: How long does it take the algorithm to run?
- We can also judge them on “space”: How many bits of memory does the algorithm take to run?
- We might prioritize one or the other or try to find a healthy balance.

Time and Space

- The most straightforward way to judge is on “running time”: How long does it take the algorithm to run?
- We can also judge them on “space”: How many bits of memory does the algorithm take to run?
- We might prioritize one or the other or try to find a healthy balance.
- Today we will focus on “running time”.

Analyzing Running Time

- How can we analyze the running time of an algorithm?

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
    - Different computers can take different amounts of time to run the same program

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
  - Different computers can take different amounts of time to run the same program
- Analyze the algorithm *mathematically* and try to count how many steps the algorithm takes

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
  - Different computers can take different amounts of time to run the same program
- Analyze the algorithm *mathematically* and try to count how many steps the algorithm takes
- The same algorithm can take a different number of steps depending on the input (e.g. how large the input array is)

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
  - Different computers can take different amounts of time to run the same program
- Analyze the algorithm *mathematically* and try to count how many steps the algorithm takes
- The same algorithm can take a different number of steps depending on the input (e.g. how large the input array is)
  - Write our analysis as a function of how large our input is (e.g. for an array of size n, it takes $n^2 + n - 5$ steps)

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
  - Different computers can take different amounts of time to run the same program
- Analyze the algorithm *mathematically* and try to count how many steps the algorithm takes
- The same algorithm can take a different number of steps depending on the input (e.g. how large the input array is)
  - Write our analysis as a function of how large our input is (e.g. for an array of size n, it takes $n^2 + n - 5$ steps)
- Algorithms can behave differently for two arrays of the same size

Analyzing Running Time

- How can we analyze the running time of an algorithm?
- We could implement the algorithm and run it and time the computation
  - Different computers can take different amounts of time to run the same program
- Analyze the algorithm *mathematically* and try to count how many steps the algorithm takes
- The same algorithm can take a different number of steps depending on the input (e.g. how large the input array is)
  - Write our analysis as a function of how large our input is (e.g. for an array of size n, it takes $n^2 + n - 5$ steps)
- Algorithms can behave differently for two arrays of the same size
  - *Worst case analysis:* What's the maximum number of steps an algorithm could take?

Analyzing `search`

- On your homework, you implemented `search` that finds an element in the array.

Analyzing search

- On your homework, you implemented search that finds an element in the array.

```
public static int search(String[] arr, String s) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i].equals(s)) {
            return i;
        }
    }

    return -1;
}
```

Analyzing `search`

- On your homework, you implemented `search` that finds an element in the array.

```
public static int search(String[] arr, String s) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i].equals(s)) {
            return i;
        }
    }

    return -1;
}
```

- Worst case is s is not there

Analyzing `search`

- On your homework, you implemented `search` that finds an element in the array.

```
public static int search(String[] arr, String s) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i].equals(s)) {
            return i;
        }
    }

    return -1;
}
```

- Worst case is s is not there
- We'll count "basic" actions as 1 step (i.e. accessing the array, checking equality)

Analyzing `search`

- On your homework, you implemented `search` that finds an element in the array.

```
public static int search(String[] arr, String s) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i].equals(s)) {
            return i;
        }
    }

    return -1;
}
```

- Worst case is s is not there
- We'll count "basic" actions as 1 step (i.e. accessing the array, checking equality)
- We have to loop through the entire array, and each iteration we do 2 actions so roughly $2n$ steps

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*
- We check the middle element m and compare it to s.

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*
- We check the middle element m and compare it to s.
  - If $m = s$, we are done.

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*
- We check the middle element m and compare it to s.
  - If $m = s$, we are done.
  - If $s < m$, then s is to the left of m (or not in the array)

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*
- We check the middle element m and compare it to s.
  - If $m = s$, we are done.
  - If $s < m$, then s is to the left of m (or not in the array)
  - If $s > m$ then s is to the right of m (or not in the array)

Binary Search

- If `arr` is sorted in alphabetical order, we can use *binary search*
- We check the middle element m and compare it to s.
    - If $m = s$, we are done.
    - If $s < m$, then s is to the left of m (or not in the array)
    - If $s > m$ then s is to the right of m (or not in the array)
- We repeat the above step with the half of the array that s must be in if it is in the array.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
  - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
  - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.
  - We compare the middle element of the second quadrant, f to e and see $e < f$ so e must be to the right.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
  - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.
  - We compare the middle element of the second quadrant, f to e and see $e < f$ so e must be to the right.
  - We get down to a single element, e and compare it and see that we found it!

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
    - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
    - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.
    - We compare the middle element of the second quadrant, f to e and see $e < f$ so e must be to the right.
    - We get down to a single element, e and compare it and see that we found it!
- This 15 element array took 3 iterations.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
  - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.
  - We compare the middle element of the second quadrant, f to e and see $e < f$ so e must be to the right.
  - We get down to a single element, e and compare it and see that we found it!
- This 15 element array took 3 iterations.
- In general, if there are $\approx 2^k$ elements, it takes k iterations.

Binary Search Example

- `search({"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o"}, "e")`
  - We compare the middle element, h, to e and see $e < h$ so e must be to the left.
  - We compare the middle element of the left half, d to e and see $e > d$ so e must be to the right.
  - We compare the middle element of the second quadrant, f to e and see $e < f$ so e must be to the right.
  - We get down to a single element, e and compare it and see that we found it!
- This 15 element array took 3 iterations.
- In general, if there are $\approx 2^k$ elements, it takes k iterations.
- So an array of size n takes roughly $2 \cdot \log_2 n$ steps.

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.
- If we wrote this without a helper function, we would use a double for-loop:

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.
- If we wrote this without a helper function, we would use a double for-loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; i++) {
```

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.
- If we wrote this without a helper function, we would use a double for-loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; i++) {
```

- When $i = 0$, the inner for loop runs n times. When $i = 1$, the inner loop runs $n - 1$ times. In general the inner loop runs $n - i$ times.

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.
- If we wrote this without a helper function, we would use a double for-loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; i++) {
```

- When $i = 0$, the inner for loop runs n times. When $i = 1$, the inner loop runs $n - 1$ times. In general the inner loop runs $n - i$ times.
- Total number of iterations is $1 + 2 + 3 + \ldots + n$

Sorting

- On your homework we implemented `selection sort`, where we slowly sort the array by searching the unsorted portion of the array for the minimum and then moving it to the front.
- If we wrote this without a helper function, we would use a double for-loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; i++) {
```

- When $i = 0$, the inner for loop runs n times. When $i = 1$, the inner loop runs $n - 1$ times. In general the inner loop runs $n - i$ times.
- Total number of iterations is $1 + 2 + 3 + \ldots + n$
- This sum actually equals $\frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$

Sorting

- There are other sorting algorithms, like *bubble sort*

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.
  - We compare 5 and 3 and they aren't in the correct order, so we swap them.

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.
  - We compare 5 and 3 and they aren't in the correct order, so we swap them.
  - We compare 5 and 2 and they are not in the correct order, so we swap them.

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.
  - We compare 5 and 3 and they aren't in the correct order, so we swap them.
  - We compare 5 and 2 and they are not in the correct order, so we swap them.
  - After one iteration, we get {`1, 3, 2, 5`}

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.
  - We compare 5 and 3 and they aren't in the correct order, so we swap them.
  - We compare 5 and 2 and they are not in the correct order, so we swap them.
  - After one iteration, we get {`1, 3, 2, 5`}
- In the worst case, it takes an element n iterations for an element to reach its correct position

Sorting

- There are other sorting algorithms, like *bubble sort*
- In bubble sort, we loop over the array until it is sorted. During each iteration, we compare each adjacent element. If they are not in increasing order we swap the two positions.
- For example, if the array is {`1, 5, 3, 2`} we loop through the array as such:
  - We compare 1 and 5 and they are both in the correct order, so we do nothing.
  - We compare 5 and 3 and they aren't in the correct order, so we swap them.
  - We compare 5 and 2 and they are not in the correct order, so we swap them.
  - After one iteration, we get {`1, 3, 2, 5`}
- In the worst case, it takes an element n iterations for an element to reach its correct position
- Each iteration makes $n - 1$ comparisons, for a total of $n(n - 1)$ iterations.

Exponential Brute Force

- If we a Boolean formula with 2 variables x and y, there are 4 combinations of truth-values we can assign x and y:

Exponential Brute Force

- If we a Boolean formula with 2 variables x and y, there are 4 combinations of truth-values we can assign x and y:
- $x = T, y = T$, or $x = T, y = F$, or $x = F, y = T$, or $x = F, y = F$.

Exponential Brute Force

- If we a Boolean formula with 2 variables x and y, there are 4 combinations of truth-values we can assign x and y:
- $x = T, y = T$, or $x = T, y = F$, or $x = F, y = T$, or $x = F, y = F$.
- In general, there are 2^n combinations

Exponential Brute Force

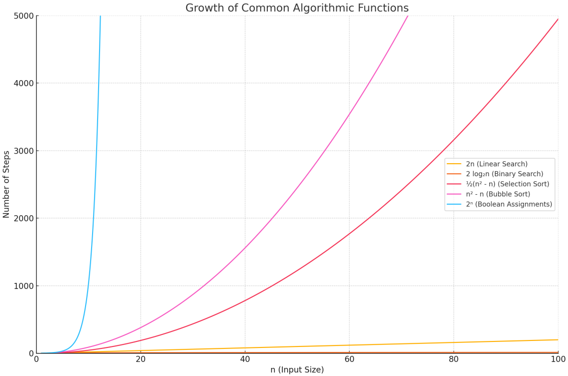
- If we a Boolean formula with 2 variables x and y, there are 4 combinations of truth-values we can assign x and y:
- $x = T, y = T$, or $x = T, y = F$, or $x = F, y = T$, or $x = F, y = F$.
- In general, there are 2^n combinations
- If we wanted to check whether a formula was ever true for any combination, we have to check 2^n combinations.

Comparing These Running Times

	Search	Binary Search	Selection Sort	Bubble Sort	Booleans
Steps	$2n$	$2\log_2(n)$	$\frac{1}{2}(n^2+n)$	n^2-n	2^n

Comparing These Running Times

	Search	Binary Search	Selection Sort	Bubble Sort	Booleans
Steps	$2n$	$2\log_2(n)$	$\frac{1}{2}(n^2+n)$	n^2-n	2^n



Big O Notation

- As n gets bigger, coefficients matter less (e.g. $2n$ vs $3n$), smaller powers matter less (e.g. n^3 vs $n^3 - n^2 + n$)

Big O Notation

- As n gets bigger, coefficients matter less (e.g. $2n$ vs $3n$), smaller powers matter less (e.g. n^3 vs $n^3 - n^2 + n$)
- What matters is the *leading* term (e.g. $\log n$ vs n vs n^2 vs 2^n)

Big O Notation

- As n gets bigger, coefficients matter less (e.g. $2n$ vs $3n$), smaller powers matter less (e.g. n^3 vs $n^3 - n^2 + n$)
- What matters is the *leading* term (e.g. $\log n$ vs n vs n^2 vs 2^n)
- We use *Big O Notation* to hide these coefficients and lower order terms:

Big O Notation

- As n gets bigger, coefficients matter less (e.g. $2n$ vs $3n$), smaller powers matter less (e.g. n^3 vs $n^3 - n^2 + n$)
- What matters is the *leading* term (e.g. $\log n$ vs n vs n^2 vs 2^n)
- We use *Big O Notation* to hide these coefficients and lower order terms:
- $n^5 - n^2 + 1000 = \mathcal{O}(n^5)$

Big O Notation

- As n gets bigger, coefficients matter less (e.g. $2n$ vs $3n$), smaller powers matter less (e.g. n^3 vs $n^3 - n^2 + n$)
- What matters is the *leading* term (e.g. $\log n$ vs n vs n^2 vs 2^n)
- We use *Big O Notation* to hide these coefficients and lower order terms:
- $n^5 - n^2 + 1000 = \mathcal{O}(n^5)$
- If an algorithm doesn't take longer as the input gets longer, we say it is $O(1)$.

Practice vs Theory

- In *theoretical CS*, algorithms that are $\mathcal{O}(n^k)$ are often called *efficient*

Practice vs Theory

- In *theoretical CS*, algorithms that are $\mathcal{O}(n^k)$ are often called *efficient*
- A $\mathcal{O}(2^n)$ algorithm would be inefficient

Practice vs Theory

- In *theoretical CS*, algorithms that are $\mathcal{O}(n^k)$ are often called *efficient*
- A $\mathcal{O}(2^n)$ algorithm would be inefficient
- In real life, $O(n)$ and $O(n \log_2(n))$ algorithms are often called efficient

Practice vs Theory

- In *theoretical CS*, algorithms that are $\mathcal{O}(n^k)$ are often called *efficient*
- A $\mathcal{O}(2^n)$ algorithm would be inefficient
- In real life, $O(n)$ and $O(n \log_2(n))$ algorithms are often called efficient
- Even $O(n^2)$ might be too much in practice