Topic 12.0: Analysis of Algorithms

Learning Goals:

- Describe why Computer Scientists care about the major “steps” in an algorithm over raw measurements like CPU time.
- Express the complexity of a basic algorithm using big-O notation.
- Compare and contrast the running times of linear versus binary search using big-O notation.
- Compare and contrast the running times of insertion, selection, merge, and quick sort using big-O notation

Searching Algorithms

- Previously we had seen two different searching algorithms: linear search and binary search.
- We know that the two algorithms have different performance (speed). How could we compare them?
    - We could implement them and time them, but this is dependent on the implementation details, computer system we are using, and test data.
    - Or we could **analyze the algorithms.**

Comparing Relative Speed

- The difference between the two algorithms is **huge**!
- If you double the list size:
    - The linear search **doubles** the iterations
    - The binary search **adds only 1** iteration!

List Size	Linear iterations	Binary iterations
10	10	4
20	20	5
1000	1000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Analyzing Speed

- To analyse the speed of an algorithm
    - Count the number of steps or operations needed
    - Look at it as a function of the size of the problem

    - You can look at either the **average** number of steps, or the **maximum** number of steps

- For any kind of search, the size of the problem (size of the input) is the size of the list/array – **call it n**

Analyzing Speed: Linear

- For a linear search:
- The loop will be executed an average of $\frac{n}{2}$ times, or a maximum of n times
- The number of operations (time) needed for each iteration will be some small constant amount – call it c
- So the total average number of operations (time) is $t(n) = c \times \frac{n}{2}$ and the maximum time is $t(n) = c \times n$
- $t(n)$ means the total average number of operations (total time) it takes to complete the given algorithm over n elements

Analyzing Speed: Binary

- For a binary search:
  - each iteration will take some small constant amount of time c
  - Each iteration will cut the size of the list in half
  - The **maximum number of iterations is related to the maximum number of times you can cut n in half**
  - $ceil\left(log_2 n\right)$
- For n **= 15** it will search 15, 7, 3, and 1 elements
  - $ceil\left(log_2 15\right)$ = 3.90689... ~maximum 4 iterations

Analyzing Speed: Binary

- For a binary search:
  - The average number of operations (the average case) is exactly 1 less than this (it takes some mathematics to prove that - out of the scope of this course)
  - So the average time is:
    - $t\left(n\right) = c \times \left(\left(log_2 n\right) - 1\right)$
  - and the maximum time is
    - $t\left(n\right) = c \times log_2 n$

Comparing Algorithms

- In summary:
    - Linear search: $t\left(n\right) = c \times \frac{n}{2}$ or n
    - Binary search: $t\left(n\right) = c \times log_2 n - 1$ or $log_2 n$

- **Constants don't really matter, so you can ignore c's, or ½ or 2 or -1**

Comparing Algorithms: Takeaways

- The really important thing about an algorithm is: as **n** grows, how does the number of operations (time) grow?

- That's determined only by how **n** (size of input) affects the number of operations

Comparing Algorithms: Takeaways

- The really important thing about an algorithm is: as **n** grows, how does the number of operations (time) grow?

- That's determined only by how **n** (size of input) affects the number of operations
- The linear search is "**O(n)**"
  - If **n** doubles, the **time** doubles

- The binary search is "**O(log n)**"
  - If **n** doubles, the time goes up by a small increment (**logn**)

Big “O” Notation?

- Big “O” Notation means **Order of the Function**
  - i.e. How the function grows with respect to **n**
  - (also known as Landau's Symbol)
  - i.e.i.e When **n** gets bigger, what happens to our running time/algorithm?
- Often called “worst-case” complexity
  - As the needs of our algorithm increase, how does the performance of said algorithm hold?

Big “O” Notation?

- Big “O” Notation means **Order of the Function**
  - i.e. How the function grows with respect to **n**
  - (also known as Landau's Symbol)
  - i.e.i.e When **n** gets bigger, what happens to our running time/algorithm?
- Often called “worst-case” complexity
  - As the needs of our algorithm increase, how does the performance of said algorithm hold?

- Let **n** be the size of the input
- Let **T(n)** be the function that defines the number of operations (or the amount of space) required by an algorithm on the input **n**

- Then $T(n) = O(f(n))$ if for positive constants c and n_0 ,

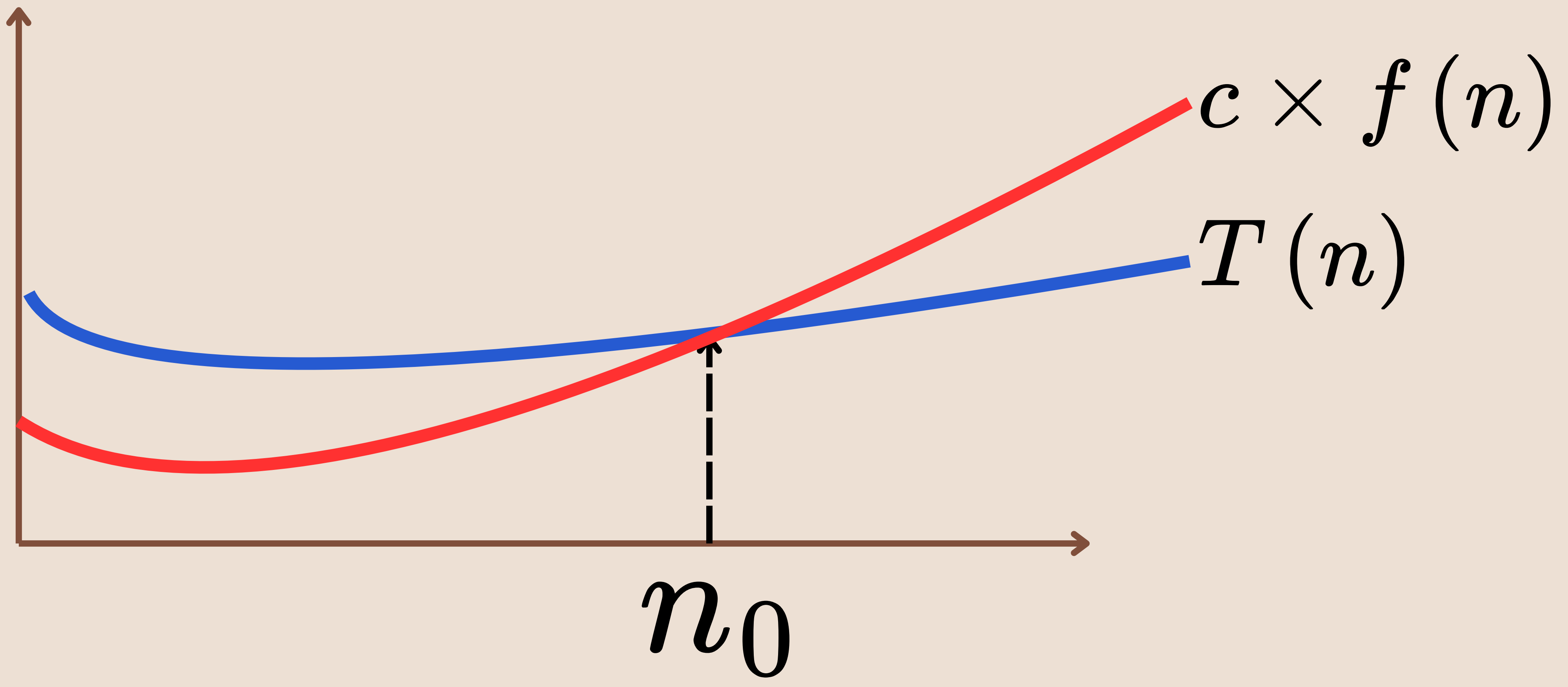
$T(n) <= c \times f(n)$ when $n >= n_0$

Big "O" Notation?

$T(n) = O(f(n))$ if for positive constants c and n_0 ,

$T(n) <= c \times f(n)$ when $n >= n_0$

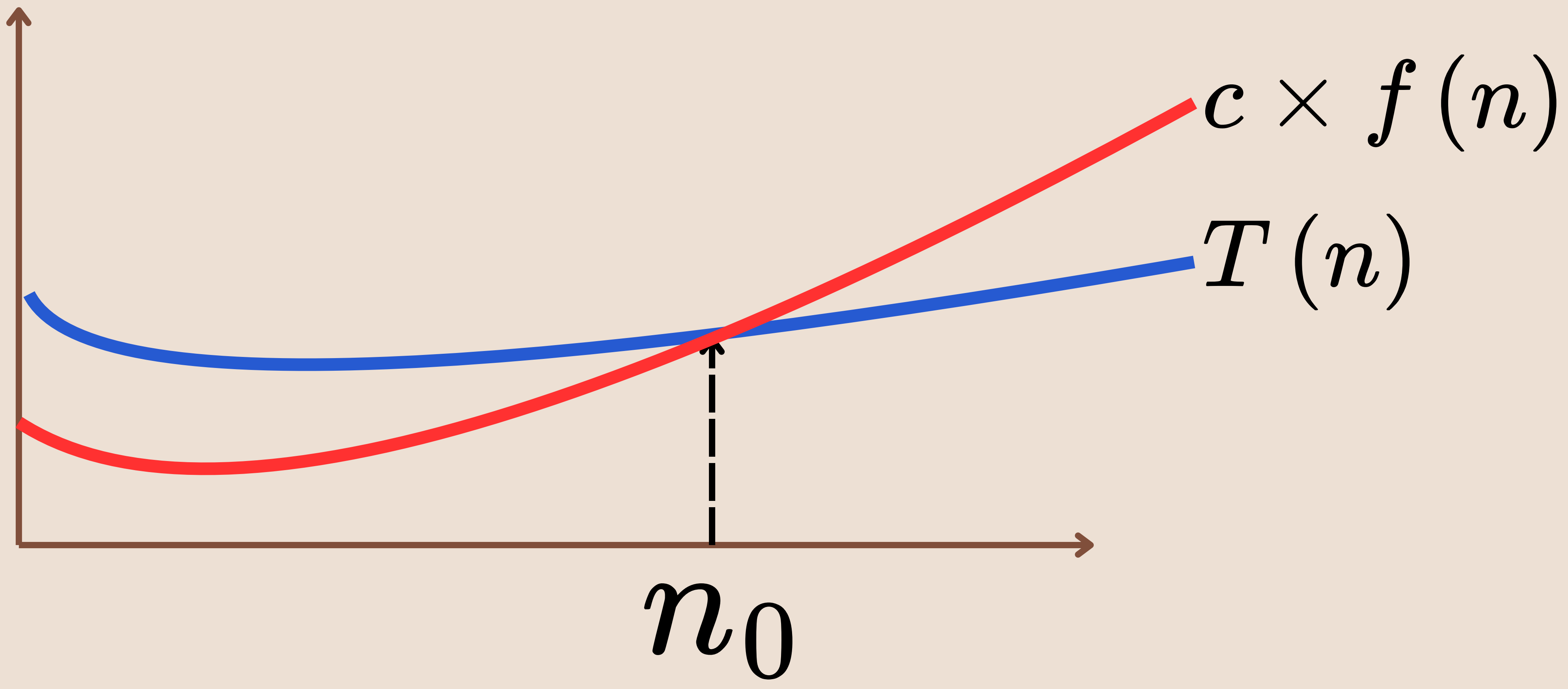
In English: When we say an algorithm is O(f(n)), we are saying that there is a constant **c** and starting **n (n0)** such that, for all larger inputs, the function will not exceed c * the f(n) function



Big “O” Notation?

The goal is to find a **function of n** that will be an upper bound on the number of operations that our algorithm uses **(f(n))**
Then we can say that the number of operations required by our algorithm **is in O(f(n))**

More on this in 2140 next year!



Typical Bounds

F(n)	# operations
c	constant
$log_2 n$	logarithmic
n	linear
$nlog_2 n$	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Quick Estimations

You can get a quick estimate of the order by asking one question

- If I double n, what will happen?

Quick Estimations

You can get a quick estimate of the order by asking one question

- If I double n, what will happen?

F(n)	double n?
$log_2 n$	very fast, n adds c to time
n	double n, double time
$nlog_2 n$	not much slower than n
n^2	double n, 4 times the time
n^3	double n, 8 times the time

These are considered **polynomial**

- **Computable and depends on the input size**

Quick Estimations

But what about these two?

F(n)	# operations
2^n	exponential
$n!$	factorial

Quick Estimations

But what about these two?

F(n)	# operations
2^n	exponential
$n!$	factorial

- Grows more quickly than any polynomial
- Considered **“not computable”** except for teeny-tiny inputs
- Have an algorithm O(n!)?
    - Just throw it out.

Running Times: Who cares?

- We do!
- How else can we make sure we are picking the right algorithm to suit our needs?
- Let’s consider:
    - Binary Search: When should we use it?

Running Times: Who cares?

- We do!
- How else can we make sure we are picking the right algorithm to suit our needs?
- Let’s consider:
  - Binary Search: When should we use it?
  - You **need** a **sorted list** so:
    - keep it sorted as you build it
    - take an existing list and sort it …
  - But remember:

Running Times: Who cares?

- We do!
- How else can we make sure we are picking the right algorithm to suit our needs?
- Let’s consider:
  - Binary Search: When should we use it?
  - You **need** a **sorted list** so:
    - keep it sorted as you build it
    - take an existing list and sort it ...
  - But remember:
    - **Sorting a list (or keeping a list sorted) is even slower than a linear search**
    - **Why bother?!**

Binary Search Times

- You happen to already have a sorted list
- You plan to do **a lot** of searching but not a lot of data manipulation
- You have lots of time to sort and then see above point.

- We really have to consider the different cases and what we are going to be doing in our program. We don’t just get to pick our favorite sorting algorithm and go.

Binary Search Times

- You happen to already have a sorted list
- You plan to do **a lot** of searching but not a lot of data manipulation
- You have lots of time to sort and then see above point.

- We really have to consider the different cases and what we are going to be doing in our program. We don't just get to pick our favorite sorting algorithm and go.

Speaking of which… What are the sorting algorithm running times?

Ordered Insert

- Data Structure Independent Analysis
- How fast is this?
  - There is only one loop that goes through the array one element at a time
  - It might go through all the elements and put our new piece of data at the end
  - It might insert at the front
  - On average, it goes through half the elements
- **O(n)** - same as a linear search
- it is a linear search where we move elements as we go (or don’t need to move elements in the case of a linked list but still possibly **n** steps to find the position)

Insertion Sort

- Let’s consider insertion sort (the loop around the ordered insert):

```
for(int k=1; k < n; k++)
      ordInsert(k, a, a[k]);
```

- It contains one simple loop that always runs **n** times
- Inside that loop it does an ordered insertion, which is **O(n)** – it does **n** steps
- Actually, it does **1, 2, 3, 4, ..., n** steps
  - But that's, on **average, n/2,** which is **O(n)** anyway
- So we do **n** steps, **n** times: this is $O\left(n^2\right)$

Selection Sort

- Next up, let’s analyze the Selection Sort algorithm

```
for(int k=0; k<=a.length-2; k++) {
   ...small bit of work (initializing min)...
   for(int i=k+1; i<=a.length-1; i++)
       ...small bit of work (updating min if necessary)...
   ...small bit of work (swapping)...
}
```

Selection Sort: Rough Analysis

- Next up, let's analyze the Selection Sort algorithm

```
for(int k=0; k<=a.length-2; k++) {
   ...small bit of work (initializing min)...
   for(int i=k+1; i<=a.length-1; i++)
       ...small bit of work (updating min if necessary)...
   ...small bit of work (swapping)...
}
```

- There are two nested loops, both of which are **O(n)** (a.length is n here)
  - Again, the inner one is only ½ of n, on average, but that's still **O(n).** Ignore constants like ½ .
- n steps, n times:
  - So this is $O\left(n^2\right)$

Simple Sorting Algorithms

- All of our symple sorting algorithms: insertion, selection, bubble, are $O\left(n^2\right)$

- This means that even though some may be faster than others when used on the same data, they are all ***theoretically*** equally slow
- When the lists get **very** long (millions, billions, etc) **all** these sorting algorithms become impractical or impossible to use.

The Better Way to Sort: Analysis

- Improving on these simple sorts requires us removing the nested loop situation
  - This is what is making them $O\left(n^2\right)$
- We can replace the loops with the same kind of “divide and conquer” we see to make binary search faster than linear search
  - **This is how we get merge sort**