CMPT 830 - Bioinformatics and Computational Biology

Chapter 1: Review of Basic Mathematics, Algorithms and Complexity

Ian McQuillan and Tony Kusalik
mcquillan@cs.usask.ca

University of Saskatchewan

September 5, 2019

What is Computation?

- Your laptop (electronic computer) is carrying out computations,
- Is nature computing?
- I decided to look it up on Wikipedia to see what it gave me:

Definition (by wikipedia)

Computation is a general term for any kind of information processing. This includes phenomena ranging from simple calculations to human thinking.

What is Computation?

That definition used the words “information processing”.

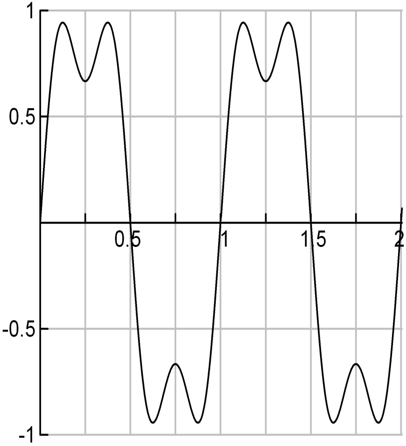
Definition (by wikipedia)

In general, *information processing* is the changing (processing) of information in any manner detectable by an observer. As such, it is a process which describes everything which happens (changes) in the universe, from the falling of a rock (a change in position) to the printing of a text file from a digital computer system.

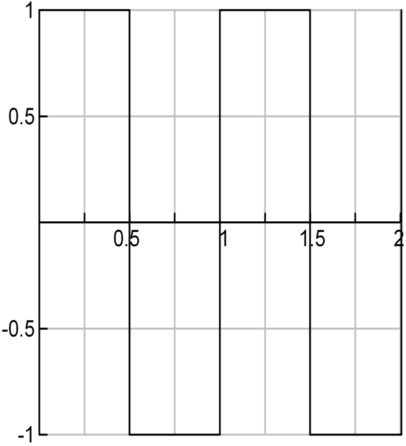
What About Digital vs. Analog?

- Voltages exist continuously, not just in two distinct states.
- However, if the voltage is "high", a computer interprets this as "1". If the voltage is "low", then a computer interprets this as "0".
- This is how an electronic computer stores a single bit of information, in binary.
- It also allows us to deal with stored information on a computer in an *abstract* way, using 0's and 1's, using discrete time points, instead of voltages changing continuously.

Abstraction



$\Rightarrow$



Pictures from "Data and Computer Communications", fourth edition, by William Stallings, 1994.

What about Digital vs. Analog?

- Anything built on top of this abstraction is “digital”.
- Are there any “digital-like” abstractions in biology, or physics?

What is Computation?

- It is natural to study biological systems in terms of information processing, and hence, computation!
- It gives us a formal, mathematical framework in order to understand biology.

Computation $\rightarrow$ Bioinformatics

- Most work in bioinformatics involves
  - the development of applications (and algorithms) to analyze biological information,
  - and the analysis (making conclusions) about the biological information.
- The closer the tools conform to the biological process, the better it'll be. We want the algorithms to *help inform the biology*.
- Computer science can provide more than just tools for biology, but a way to improve our understanding of biology.

Original Notion of Computation

The original formulation of computation was defined in the early 1900's, in the form of a *Turing Machine*.

At each step, the machine can

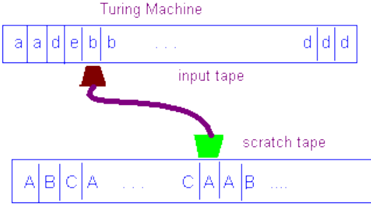
based on:

1) current state, 2) current input symbol,
3) current scratch symbol,

do the following:

1) switch states, 2) move input tape head right one symbol or leave,
3) change symbol on the scratch tape, 4) move the scratch tape head one symbol to the left or right or leave.

The machine has a finite number of “states”, and at each time step, the machine is in one state.



Original Notion of Computation

- No physically realizable method of computing that we can come up with is more powerful than this.
- Any physically realizable method of computing that humans can develop can be "simulated" by a Turing Machine.
- This does not mean that there are not quicker methods of computing, however.
- More on this in a minute.

What is an Algorithm?

Definition (informal)

An *algorithm* is an ordered sequence of effective instructions.

- *Effective* means that each instruction is unambiguous and understandable. We know how to execute each instruction.
- We construct algorithms in order to solve a well-formulated problem.
- Typically, we specify algorithms in terms of a method of translating inputs into outputs.

What is an Algorithm?

- The definition of algorithm is vague and informal, because a Turing Machine is the closest we can do to defining it formally.
- This is known as the *Church-Turing Thesis*

Church-Turing Thesis (slight variation)

Every function that can be physically computed can be computed by a Turing Machine.

- If this is actually true, then one can see why Computer Science is important for studying biology (and the universe).

Oh No!

- If we want to study algorithms, do we have to be experts in "programming" Turing Machines?
- Fortunately no.
- We will formally specify algorithms, using *Pseudocode*, which is how computer scientists usually specify algorithms.
- It's still not easy though.

Pseudocode

- There is no way to execute programs written in pseudocode, however it is convenient for computer scientists.
- It can be easily translated into high-level computer languages such as `C` or `Java` or `Python`.
- Each has the same "power" as Turing machines.

Pseudocode Constructs

Variables

- In pseudocode, we need "containers" to hold data so the computation can work with it.
- This is the concept of *variables*.
- Variables mean subtly different things in Computer Science and Mathematics.

in Math.

- A variable can take on any value from a universe of alternatives (a set).
- It makes sense to say, "Let x be an integer".
- Time is not associated with the variables. Saying $x = 4$ followed by $x = 5$ is a contradiction.
- Otherwise, one could conclude that $4 = 5$ which is never true.

Variables

in Comp. Sci.

- At each time, a variable will contain *at most* a single value.
- In Computer Science pseudocode, if you say,

```
x←3
x←4
```

  this means that at one step, x contained the variable 3, and the next step, it contained 4.
- So, there is an *order* to pseudocode, which is implicitly there.

For the math savvy, you could describe a computer science variable x as a partial function from the positive integers into whatever domain the variable type is in, in this case, the integers. So for the example above: $x(1) = 3$, $x(2) = 4$ and $x(i)$ is undefined if $i > 2$.

Variables

in Comp. Sci.

- At each time, a variable will contain *at most* a single value.
- In Computer Science pseudocode, if you say,

```
x←3
x←4
```

  this means that at one step, x contained the variable 3, and the next step, it contained 4.
- So, there is an *order* to pseudocode, which is implicitly there.

For the math savvy, you could describe a computer science variable x as a partial function from the positive integers into whatever domain the variable type is in, in this case, the integers. So for the example above: $x(1) = 3$, $x(2) = 4$ and $x(i)$ is undefined if $i > 2$.

Atomic Instructions

- What are the *effective* instructions in our pseudocode?
- That is, what are the elementary or "*atomic*" instructions that makes up our pseudocode?
- Here is the first type of atomic instruction: assignment instructions.

assignment

Syntax: $a \leftarrow b$
Effect: copies the contents of the variable or constant b into the variable a.

Atomic Instructions

- What are the *effective* instructions in our pseudocode?
- That is, what are the elementary or "*atomic*" instructions that makes up our pseudocode?
- Here is the first type of atomic instruction: assignment instructions.

assignment

Syntax: $a \leftarrow b$
Effect: copies the contents of the variable or constant b into the variable a.

Assignment

example

```
b ← 2
a ← 3
b ← a
```

terminates with *a* and *b* both containing 3.

Atomic Instructions

arithmetic

Syntax: $a + b, a - b, a \cdot b$ or $a * b, a/b$
Effect: performs real addition, subtraction, multiplication and division respectively on real numbers a and b.

example

```
a ← 2
b ← 3
b ← a + b
```

terminates with a containing 2 and b containing 5.

Atomic Instructions

arithmetic

Syntax: $a + b$, $a - b$, $a \cdot b$ or $a * b$, a/b
Effect: performs real addition, subtraction, multiplication and division respectively on real numbers a and b.

example

```
a ← 2
b ← 3
b ← a + b
```

terminates with a containing 2 and b containing 5.

Atomic Instructions

Is the following pseudocode defined?

```
a ← 2
b ← 3
a+b ← b
```

Atomic Tests and Instructions

A test decides whether a proposition is true or false.

Tests

Syntax: a equals b, $a = b$
where a and b are arbitrary variables.
Effect: evaluates to true if the contents of a is equal to the contents of b, and false otherwise.

Tests

Syntax: $c > d$, $c < d$, $c \geq d$, $c \leq d$
where c and d are real numbers.
Effect: evaluates to true if the contents of a is, greater than, less than, greater than or equal to, less than or equal to b, respectively. Evaluates to false otherwise.

The Conditional Instruction

Conditional

Syntax: if A is true
    B
  else
    C
where A is a test and B and C are sequences of instructions.
Effect: if the test A is true, then execute B, otherwise execute C.
We will sometimes omit "else C".

```
x ← 2
if x > 0 is true
  y ← x
else
  y ← 0
```

The Conditional Instruction

Conditional

Syntax: if A is true
B
else
C

where A is a test and B and C are sequences of instructions.
Effect: if the test A is true, then execute B, otherwise execute C.
We will sometimes omit "else C".

```
x ← 2
if x > 0 is true
  y ← x
else
  y ← 0
```

Non-Atomic Instructions

- We would like to not have to use atomic instructions all the time.
- If we already write a program, we don't want to write it again.
- We develop *subroutines*, which are "mini-algorithms" that we can use inside other algorithms.

Non-Atomic Instructions

A subroutine is denoted by some appropriate name, followed by a number of arguments (or parameters, or inputs) that it requires. It may provide a result of its computation.

```
input: a and b, real numbers
returns: the smaller value of the two variables a and b.

MIN(a,b){
  if a< b
    return a
  else
    return b
}
```

Non-Atomic Instructions

Example

Now, let's say that we have three variable *a*, *b* and *c*, and we would like to find the smallest of the three variables.

```
result ← MIN(a,b)
result ← MIN(result, c)
```

Then, the variable 'result' contains the smallest of *a*, *b* and *c*.

Example

or even quicker with

```
result ← MIN(a, MIN(b, c))
```

Non-Atomic Instructions

Example

Now, let's say that we have three variable a, b and c, and we would like to find the smallest of the three variables.

```
result ← MIN(a,b)
result ← MIN(result, c)
```

Then, the variable 'result' contains the smallest of a, b and c.

Example

or even quicker with

```
result ← MIN(a, MIN(b, c))
```

Non-Atomic Instructions

- Hence, subroutines allow us to define our own instructions, much like the atomic ones.
- We can break our pseudocode into logically separate parts, so that we can
  1. reuse existing pseudocode
  2. extend existing pseudocode
  3. organize our pseudocode into logically separate parts

Non-Atomic Instructions

Now that we've got an algorithm for MIN, it becomes really easy to write an algorithm for MAX!

```
input: a and b, real numbers
returns: the larger value of the two variables a and b.

MAX(a,b){
  return -1 · MIN(-1 · a, -1 · b)
}
```

Back to Atomic Instructions - For Loops

for loops

Syntax: for $i \leftarrow a$ to b
B
Effect: sets i to a, executes a sequence of instructions B, increases i by one, executes B, and so on for $i = a, a+1, \ldots, b$, where i, a, b are integers.

For example, let's write a subroutine for exponentiation.

```
input: a, a real number and e, a non-negative number
returns: returns a to the exponent of e

EXP(a,e){
  j ← 1
  for i ← 1 to e
    j ← j · a
  return j
}
```

While Loops

while loops

Syntax: while A is true
B

Effect: tests if condition A is true, and if it is then execute a sequence of instructions B, and repeat until A is false.

```
input: n, a natural number
returns: returns the first square greater than or equal to n

FIRST_SQUARE(n){
  j ← 1
  while EXP(j,2) < n is true
    j ← j + 1
  return EXP(j,2)
}
```

Arrays

An array is a sequence of variables, each of which can be accessed by an index corresponding to its position in the sequence.

Syntax: $A(i)$
where $\mathbf{A} = (A(1), \ldots, A(i), \ldots, A(n))$ and $1 \leq i \leq n$.
Effect: Accesses the i^{th} element of the array.

```
input: an array A of integers of size n
returns: returns the sum of the elements in the array,

SUMMAND(A,n){
  j ← 0
  for i ← 1 to n
    j ← j + A(i)
  return j
}
```

Example Algorithm

- One of the basic problems in Computer Science is that of sorting an array of integers into increasing order.
- That is, if the input is the array $(4, 2, 6, 11, 8)$, the output would be $(2, 4, 6, 8, 11)$.
- We state the problem as follows:

Sorting Problem:

sort an array of integers into increasing order
input: An array of n distinct integers $\mathbf{A} = (a_1, \ldots, a_n)$,
output: An array of n distinct integers $\mathbf{B} = (b_1, \ldots, b_n)$, such that $b_1 < b_2 < \cdots < b_n$ and $\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$.

A Subroutine

- We break our problem into subproblems, each accomplishing one "logical" task.
- Before we tackle the sorting problem, let's look at devising an algorithm to solve an easier problem:

```
input: an array A of size n, two positive integers a,b
  such that 1 ≤ a ≤ b ≤ n
returns: the index of the smallest number in
  {A(a), A(a+1), ..., A(b)}

INDEX_SMALLEST(A,n,a,b){
  index ← a
  for k ← a+1 to b
    if A(k) < A(index)
      index ← k  //remember this index
  return index
}
```

A Subroutine

```
input: an array A of size n, two positive integers a,b
  such that 1 ≤ a ≤ b ≤ n
returns: the index of the smallest number in
  {A(a), A(a+1), ..., A(b)}

INDEX_SMALLEST(A,n,a,b){
  index ← a
  for k ← a+1 to b
    if A(k) < A(index)
      index ← k   //remember this index
  return index
}
```

- `index` can be thought of as the "index of the smallest element encountered so far".
- We initially set `index` to `a` as a "guess".

A Subroutine

```
input: an array A of size n, two positive integers a,b
  such that 1 ≤ a ≤ b ≤ n
returns: the index of the smallest number in
  {A(a), A(a+1), ..., A(b)}

INDEX_SMALLEST(A,n,a,b){
  index ← a
  for k ← a+1 to b
    if A(k) < A(index)
      index ← k   //remember this index
  return index
}
```

- if $a = b$, the loop is not performed

Sorting

Now that we've solved that simplified problem, let's return to the problem of sorting.

```
input: an array A with n distinct elements
returns: An array of n distinct integers B = (b1, ..., bn),
such that b1 < b2 < ··· < bn and
{a1, ..., an} = {b1, ..., bn}

SORT(A,n){
  for i ← 1 to n
    j ← INDEX_SMALLEST(A, n, i, n))
    temp ← A(i)      //The next three lines swap
    A(i) ← A(j)      //A(i) with A(j)
    A(j) ← temp
  return A
}
```

Sorting

The last time through the loop, nothing is done. Therefore we can optimize the algorithm a bit

input: an array A with n distinct elements
returns: An array of n distinct integers $B = (b_1, \ldots, b_n)$, such that $b_1 < b_2 < \cdots < b_n$ and $\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$

```
SORT(A,n){
  for i ← 1 to n-1
    j ← INDEX_SMALLEST(A, n, i, n))
    temp ← A(i)     //The next three lines swap
    A(i) ← A(j)     //A(i) with A(j)
    A(j) ← temp
  return A
}
```

Speed

- The “speed” of an algorithm is surprisingly important.
- Many (most) algorithms take so long that we never get an answer back.
- More on this later.

Speed of an Algorithm

- We would like to talk about the inherent *speed of an algorithm*, rather than the speed on a particular computer architecture, since architectures constantly change.

- We need some architecture independent metric.

- We can weight each atomic instruction and test as being 1 *unit* or step and then count the total number.

Still More Complicated...

- The total number of atomic instructions which are executed depends on the particular input to the algorithm.
- Usually, the longer the input, the longer it will take.
- Most often, computer scientists measure the number of instructions *as a function of the length of the input*.
- In the case of sorting an array **A** of size n, we would measure the time as a function of n, the size of the array.

Still More Complicated...

- We would like to be able to say one algorithm is "faster" than another.

example

Suppose algorithm A executes $100n^3$ steps (on an input of size n) and algorithm B executes $100n$. Here, we can safely say that algorithm B will require fewer steps to finish.

example

Suppose algorithm A executes $100n^3$ steps (on an input of size n) and algorithm B executes $n + 10000$. Then algorithm A will perform better when the input is small, but as the input grows, algorithm B will become better, since the function $100n^3$ is a faster growing function than $n + 10000$.

Still More Complicated...

- We would like to be able to say one algorithm is "faster" than another.

example

Suppose algorithm A executes $100n^3$ steps (on an input of size n) and algorithm B executes $100n$. Here, we can safely say that algorithm B will require fewer steps to finish.

example

Suppose algorithm A executes $100n^3$ steps (on an input of size n) and algorithm B executes $n + 10000$. Then algorithm A will perform better when the input is small, but as the input grows, algorithm B will become better, since the function $100n^3$ is a faster growing function than $n + 10000$.

Time Complexity

- Computer Scientists like to use *Big-O* notation to describe the running time of an algorithm.
- The formal definition is to follow, but the concept is more easily described with an example.

Example

- If the running time is $100n^3$, then the running time of the algorithm is $O(n^3)$.
- If the running time is $n + 10000$, then the running time is $O(n)$.
- If the running time is $100n^3 + 50n^2$, then the running time is $O(n^3)$.

- It represents the fastest growing portion of the time complexity, ignoring any constant factors.

Time Complexity - Upper Bounds

- We say that the Big-O relationship establishes an *upper-bound* on the growth of a function.
- If an algorithm runs in $O(n^3)$, then it will *never* perform more than cn^3 steps, where c is some constant, on an input of size n.
- It is possible that it could perform less however. This is where the *upper-bound* notion comes into play.
- An algorithm which is $O(n^3)$ is also $O(n^4), O(n^5)$, etc. We usually take the "smallest" (least rapidly increasing) function which is still an upper bound.

Time Complexity - Lower Bounds

- We would like to be able to speak of lower bounds also.
- Say we have an algorithm that takes $50n^2$ steps. Then we know it runs in time $O(n^2)$, but it also runs in time $O(n^3)$ since the running time is *no worse* than $O(n^3)$.
- But we say that it runs in $\Omega(n^2)$ time if it requires *at least* n^2 steps (excluding any constant factor).
- We could also say that it runs in $\Omega(n)$ time because it requires at least n steps.
- In some sense, we have figured out exactly how fast it is.

Time Complexity - Lower Bounds

- We would like to be able to speak of lower bounds also.
- Say we have an algorithm that takes $50n^2$ steps. Then we know it runs in time $O(n^2)$, but it also runs in time $O(n^3)$ since the running time is *no worse* than $O(n^3)$.
- But we say that it runs in $\Omega(n^2)$ time if it requires *at least* n^2 steps (excluding any constant factor).
- We could also say that it runs in $\Omega(n)$ time because it requires at least n steps.
- In some sense, we have figured out exactly how fast it is.

Time Complexity - Tight Bounds

- Thus, as the algorithm requires $50n^2$ steps, it runs in time $O(n^2)$ and in time $\Omega(n^2)$.

definition

If an algorithm is $O(f(n))$ and is $\Omega(f(n))$, then the algorithm is $\Theta(f(n))$ and we say that $f(n)$ represents a *tight bound* on the algorithm.

- So, the algorithm above is $\Theta(n^2)$ since it requires no more than n^2 steps, but it also requires at least n^2 steps (excluding constant factors).

Calculating the Worst Case Time Complexity

- We would like to actually try this on a real problem.
- We will try to calculate the worst case time complexity on our sorting algorithm, from start to finish.

Worst Case Time Complexity of Sort

Recall the method INDEX_SMALLEST.

```
INDEX_SMALLEST(A,n,a,b){
1  index ← a
2  for k ← a+1 to b
3    if A(k) < A(index)
4      index ← k
5  return index
}
```

- We want to calculate the time complexity in terms of the parameters of the method, that is, in terms of some of $\mathbf{A}$, n, a and b.

- Step 1 takes 1 step and step 5 takes 1 step, as they are atomic instructions.

Worst Case Time Complexity of Sort

```
INDEX_SMALLEST(A,n,a,b){
1  index ← a
2  for k ← a+1 to b
3    if A(k) < A(index)
4      index ← k
5  return index
}
```

- Each time through the loop takes 1 step for line 2, 1 step for line 3 and at most one step for line 4. Thus, each time through the loop, it takes at most 3 steps.

- The loop is executed $b - a$ times. Thus, steps 2 through 4 takes at most $3(b - a)$ steps. Hence, the whole method takes at most $3(b - a) + 2$ steps.

Worst Case Time Complexity of Sort

```
INDEX_SMALLEST(A,n,a,b){
1  index ← a
2  for k ← a+1 to b
3    if A(k) < A(index)
4      index ← k
5  return index
}
```

- Each time through the loop takes 1 step for line 2, 1 step for line 3 and at most one step for line 4. Thus, each time through the loop, it takes at most 3 steps.

- The loop is executed $b - a$ times. Thus, steps 2 through 4 takes at most $3(b - a)$ steps. Hence, the whole method takes at most $3(b - a) + 2$ steps.

Worst Case Time Complexity of Sort

Recall the sorting algorithm

```
SORT(A,n){
1  for i ← 1 to n-1
2    j ← INDEX_SMALLEST(A, n, i, n))
3    temp ← A(i)     //The next three lines swaps
4    A(i) ← A(j)     //A(i) with A(j)
5    A(j) ← temp
6  return A
}
```

- Line 6 takes 1 step.
- Each time through the loop, lines 1,3,4 and 5 each take 1 step.
- For line 2, this is no longer an atomic instruction, so we must be careful!

Worst Case Time Complexity of Sort

```
SORT(A,n){
1  for i ← 1 to n-1
2    j ← INDEX_SMALLEST(A, n, i, n))
3    temp ← A(i)      //The next three lines swaps
4    A(i) ← A(j)      //A(i) with A(j)
5    A(j) ← temp
6  return A
}
```

- For line 2, the first time through the loop takes $\leq 3(n-1)+2$ steps, the second time takes $\leq 3(n-2)+2$ steps, …, the $n-1^{\text{st}}$ time takes $\leq 3(n-(n-1))+2 = 3(1)+2$ steps.

- Thus, line 2 takes $\leq (3(1)+2)+(3(2)+2)+\cdots+(3(n-1)+2)$ steps.

Worst Case Time Complexity of Sort

```
SORT(A,n){
1  for i ← 1 to n-1
2    j ← INDEX_SMALLEST(A, n, i, n))
3    temp ← A(i)      //The next three lines swaps
4    A(i) ← A(j)      //A(i) with A(j)
5    A(j) ← temp
6  return A
}
```

- For line 2, the first time through the loop takes $\leq 3(n-1)+2$ steps, the second time takes $\leq 3(n-2)+2$ steps, ..., the $n-1^{\text{st}}$ time takes $\leq 3(n-(n-1))+2 = 3(1)+2$ steps.
- Thus, line 2 takes $\leq (3(1)+2)+(3(2)+2)+\cdots+(3(n-1)+2)$ steps.

Worst Case Time Complexity of Sort

- Reorganizing, line 2 takes

$$\begin{aligned} &\leq \quad 3(1) + 3(2) + \cdots + 3(n-1) + 2(n-1) \\ &= \quad 3(1 + 2 + \cdots + (n-1)) + 2(n-1) \end{aligned}$$

  steps.

- That's a horribly long description of the time complexity, but here's a trick!

trick

$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$

- So, line 2 takes $3(n(n-1)/2) + 2(n-1) =$ $(3/2)n(n-1) + 2(n-1)$ steps.

Worst Case Time Complexity of Sort

```
SORT(A,n){
1  for i ← 1 to n-1
2    j ← INDEX_SMALLEST(A, i, n))
3    temp ← A(i)      //The next three lines swaps
4    A(i) ← A(j)      //A(i) with A(j)
5    A(j) ← temp
6  return A
}
```

- Recall that each time through the loop, we also performed 4 steps in lines 1, 3, 4 and 5.
- How many times is the loop preformed? $n-1$ times.
- Hence lines 1 through 5 account for $\leq (3/2)n(n-1) + 2(n-1) + 4(n-1)$ steps.
- Recall that there is 1 step performed in line 6.

Worst Case Time Complexity of Sort

```
SORT(A,n){
  for i ← 1 to n-1
    j ← INDEX_SMALLEST(A, i, n))
    temp ← A(i)      //The next three lines swaps
    A(i) ← A(j)      //A(i) with A(j)
    A(j) ← temp
  return A
}
```

- Putting all this together, the method takes

$$
\begin{aligned}
&\leq (3/2)n(n-1) + 2(n-1) + 4(n-1) + 1 \\
&= (3/2)(n^2 - n) + 6n - 5 = (3/2)n^2 - (3/2)n + 6n - 5 \\
&= (3/2)n^2 + (9/2)n - 5 \text{ steps.}
\end{aligned}
$$

Worst Case Time Complexity of Sort

Hence, the function $(3/2)n^2 + (9/2)n - 5$ is $O(n^2)$ since there exists a positive real constant 2 and a number 0 such that $(3/2)n^2 + (9/2)n - 5 \leq 2n^2$ for all values of $n \geq 0$.

Therefore, our SORT function has $O(n^2)$ complexity.

Worst Case Time Complexity of Sort

Definition

- An algorithm that can be solved in time $O(n)$ is called a *linear* algorithm.
- An algorithm that can be solved in time $O(n^2)$ is called *quadratic*
- $O(n^3)$ is called *cubic*
- $O(n^k)$ for some constant k is called *polynomial*
- $O(k^n)$ for some constant k is called *exponential*.

Why Do We Even Bother?

- The time complexity plays a huge part in determining whether it is actually feasible to use a particular algorithm to solve a problem.
- Say we have a whole bunch of algorithms which we are going to use to scan the whole human genome, which is approximately $3,200,000,000$ base pairs in length.
- That number represents our n, the input.
- Say one algorithm runs in $\Omega(2^n)$ time.
- There are not $2^{3,200,000,000}$ atoms in the universe, so that may cause a problem!

Why Do We Even Bother?

- The time complexity plays a huge part in determining whether it is actually feasible to use a particular algorithm to solve a problem.
- Say we have a whole bunch of algorithms which we are going to use to scan the whole human genome, which is approximately $3,200,000,000$ base pairs in length.
- That number represents our n, the input.
- Say one algorithm runs in $\Omega(2^n)$ time.
- There are not $2^{3,200,000,000}$ atoms in the universe, so that may cause a problem!

Why Do We Even Bother?

- The time complexity plays a huge part in determining whether it is actually feasible to use a particular algorithm to solve a problem.

- Say we have a whole bunch of algorithms which we are going to use to scan the whole human genome, which is approximately $3,200,000,000$ base pairs in length.

- That number represents our n, the input.

- Say one algorithm runs in $\Omega(2^n)$ time.

- There are not $2^{3,200,000,000}$ atoms in the universe, so that may cause a problem!

Why Do We Even Bother?

- Say one algorithm runs in $\Omega(n^2)$ time. Then it would require some number proportional to $3,200,000,000^2 \approx 1.0 \times 10^{20}$ is not very feasible.
- A linear time algorithm would "likely" be practical.
- For smaller inputs, for example, an individual gene, then "likely" any polynomial time algorithm would be feasible.
- Typically, exponential time algorithms (or even worse) will only work for very small inputs.

NP-Completeness

- Although we will not define it formally in this course, there is a whole class of problems, which are collectively called *NP-Complete* which satisfy several interesting properties:
    1. The best known algorithms which solves any of these problems runs in exponential time.
    2. It has not been proven mathematically that there is, or is not, an algorithm which runs in polynomial time which solves one of these problems.
    3. If someone finds an algorithm which solves *any* one of these problems, then that will automatically provide a polynomial algorithm that solves *every* one of these problems.

NP-Completeness

- This is known as the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question.
- So, if you see a problem which is NP-complete, then it "likely" has no polynomial algorithm which solves it.

Math

Much of mathematics is build on so-called *set theory*. It is also very important to computer science and bioinformatics. We will only introduce set theory informally.

definition

A *set* is an unordered collection of objects or elements.

Skim of Set Theory

If a set is finite and small, we can describe it by listing its elements inside curly braces.

example

$A = \{1, 2, 3, 4\}$.

Abstract symbols are used to denote the objects in the set.

Skim of Set Theory

We can also describe a set by listing a property necessary for membership.

example

$C = \{x \mid x \text{ is a positive, even integer}\}$.

This set contains the elements $2, 4, 6, 8, \ldots$

Skim of Set Theory

definition

- If X is a set and n is an element, we write $n \in X$ if n is an element of X.

- If n is not an element of X, we write $n \notin X$.

So, $2 \in C$, but $5 \notin C$.

Skim of Set Theory

definition

- Given sets X and Y, we say that X is a *subset* of Y, written $X \subseteq Y$, if every element of X is an element of Y.
- We say that X is a *strict subset* of Y, written $X \subset Y$, if $X \subseteq Y$ and there is an element of Y which is not in X.
- Two sets X and Y are equal, written $X = Y$ if they have the same elements.

example

$\{1, 3, 4\} = \{4, 1, 3\}$.

- Notice that $X = Y$, if and only if, $X \subseteq Y$ and $Y \subseteq X$.

Set Theory

- The *empty set* is denoted by $\emptyset$.
- We will introduce any other math at the time of use.