COMP110: Principles of Computing

5: Computational Complexity

Learning outcomes

- **Explain** the notion of computability
- **Use** "big O" notation to express computational complexity
- **Apply** appropriate algorithms to achieve efficiency

Computation time

Resources

Resources

- All programs use **resources**

Resources

- All programs use **resources**
  - Time

Resources

- All programs use **resources**
  - Time
  - Memory

Resources

- All programs use **resources**
  - Time
  - Memory
  - Network bandwidth

Resources

- All programs use **resources**
  - Time
  - Memory
  - Network bandwidth
  - Power

Resources

- All programs use **resources**
  - Time
  - Memory
  - Network bandwidth
  - Power
  - ...

Resources

- All programs use **resources**
  - Time
  - Memory
  - Network bandwidth
  - Power
  - ...
- Often **time** is the resource we care about the most

Resources

- All programs use **resources**
  - Time
  - Memory
  - Network bandwidth
  - Power
  - ...
- Often **time** is the resource we care about the most
  - Particularly in games: want to maintain a good **frame rate** free of **lag** or **stuttering**

Basic time measurement in Python

```
import time

start_time = time.clock()

... do something here ...

end_time = time.clock()
print "Computation took", end_time - start_time, " ↩
    seconds"
```

Repeating for better accuracy

```
import time

start_time = time.clock()

repetition_count = 1000

for repetition in xrange(repetition_count):
    ... do something here ...

end_time = time.clock()
time_per = (end_time - start_time) / repetition_count
print "Computation took", time_per, "seconds"
```

Scaling

Scaling

- Timing is dependent on hardware and software issues

Scaling

- Timing is dependent on hardware and software issues
- We are often less interested in how many milliseconds a particular computation takes on today's hardware, and more interested in how the execution time **scales** with the problem size

Search

Search

- We have a list of names, each with some data associated

Search

- We have a list of names, each with some data associated
- We want to find one of them

Linear search

procedure FIND(name, list)

Linear search

procedure FIND(name, list)
    for each item in list **do**

Linear search

procedure FIND(name, list)
    for each item in list **do**
        if item.name = name **then**

Linear search

```
procedure FIND(name, list)
    for each item in list do
        if item.name = name then
            return item
```

Linear search

```
procedure FIND(name, list)
    for each item in list do
        if item.name = name then
            return item
        end if
    end for
    throw "Not found"
```

Linear search

```
procedure FIND(name, list)
    for each item in list do
        if item.name = name then
            return item
        end if
    end for
    throw "Not found"
end procedure
```

How long does it take?

Socrative room code: `FALCOMPED`

- Suppose there are 25 items in the list

How long does it take?

Socrative room code: `FALCOMPED`

- Suppose there are 25 items in the list
- In the **best case**, how many items do we need to visit before finding the one we want?

How long does it take?

Socrative room code: `FALCOMPED`

- Suppose there are 25 items in the list
- In the **best case**, how many items do we need to visit before finding the one we want?
- How about in the **worst case**?

How long does it take?

Socrative room code: `FALCOMPED`

- If there are 25 items in the list, the **worst case** number of items visited is 25

How long does it take?

Socrative room code: `FALCOMPED`

- If there are 25 items in the list, the **worst case** number of items visited is 25
- How about if there are 50 items?

How long does it take?

Socrative room code: `FALCOMPED`

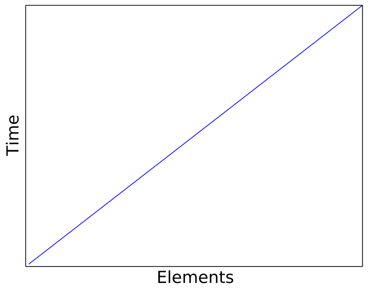
- If there are 25 items in the list, the **worst case** number of items visited is 25
- How about if there are 50 items?
- How about 100 items?

How long does it take?

Socrative room code: `FALCOMPED`

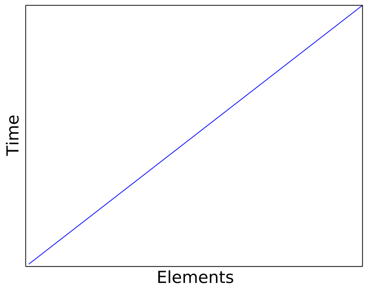
- If there are 25 items in the list, the **worst case** number of items visited is 25
- How about if there are 50 items?
- How about 100 items?
- If the number of items **doubles**, what happens to the amount of time the search takes?

Linear time



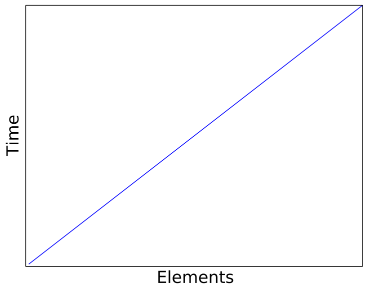
- The running time of linear search is **proportional** to the size *n* of the list

Linear time



- The running time of linear search is **proportional** to the size *n* of the list
- Linear search is said to have **linear time complexity**

Linear time



- The running time of linear search is **proportional** to the size n of the list
- Linear search is said to have **linear time complexity**
- Also written as $O(n)$ **time complexity**

Searching a sorted list

- If the list is **sorted** in alphabetical order, we can do better than linear...

Binary search

procedure FIND(name, list)

Binary search

```
procedure FIND(name, list)
    if list is empty then
        throw "Not found"
    end if
```

Binary search

procedure FIND(name, list)
    if list is empty **then**
        throw "Not found"
    end if
    mid $\leftarrow$ the "middle" item of the list

Binary search

procedure FIND(name, list)
    if list is empty **then**
        throw "Not found"
    end if
    mid $\leftarrow$ the "middle" item of the list
    if name $=$ mid.name **then**
        return mid

Binary search

procedure FIND(name, list)
    if list is empty **then**
        throw "Not found"
    end if
    mid $\leftarrow$ the "middle" item of the list
    if name = mid.name **then**
        return mid
    else if name $<$ mid.name **then**
        return FIND(name, first half of list)

Binary search

procedure FIND(name, list)
    if list is empty **then**
        throw "Not found"
    end if
    mid $\leftarrow$ the "middle" item of the list
    if name $=$ mid.name **then**
        return mid
    else if name $<$ mid.name **then**
        return FIND(name, first half of list)
    else if name $>$ mid.name **then**
        return FIND(name, second half of list)

Binary search

```
procedure FIND(name, list)
    if list is empty then
        throw "Not found"
    end if
    mid ← the "middle" item of the list
    if name = mid.name then
        return mid
    else if name < mid.name then
        return FIND(name, first half of list)
    else if name > mid.name then
        return FIND(name, second half of list)
    end if
end procedure
```

How long does it take?

Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**

How long does it take?

Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**
- Worst case: we have to keep halving until we get down to a single element

How long does it take?

Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**
- Worst case: we have to keep halving until we get down to a single element
- If the size of the list is **doubled**, what happens to the worst-case **number of iterations** required?

How long does it take?

Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**
- Worst case: we have to keep halving until we get down to a single element
- If the size of the list is **doubled**, what happens to the worst-case **number of iterations** required?
- **Answer:** it increases by 1

How long does it take?

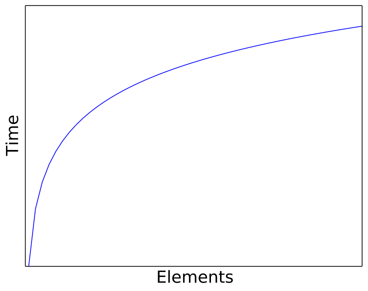
Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**
- Worst case: we have to keep halving until we get down to a single element
- If the size of the list is **doubled**, what happens to the worst-case **number of iterations** required?
- **Answer:** it increases by 1
- The running time is **logarithmic** or $O(\log n)$

How long does it take?

Socrative room code: `FALCOMPED`

- Each iteration cuts the list in **half**
- Worst case: we have to keep halving until we get down to a single element
- If the size of the list is **doubled**, what happens to the worst-case **number of iterations** required?
- **Answer:** it increases by 1
- The running time is **logarithmic** or $O(\log n)$



Hidden complexity

if name $<$ mid.name **then**
    return FIND(name, first half of list)
else if name $>$ mid.name **then**
    return FIND(name, second half of list)
end if

Hidden complexity

if name $<$ mid.name **then**
    return FIND(name, first half of list)
else if name $>$ mid.name **then**
    return FIND(name, second half of list)
end if

- Careful how you implement this!

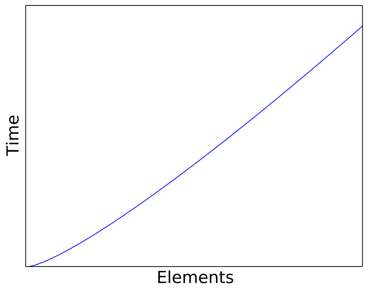
Hidden complexity

if name $<$ mid.name **then**
    return FIND(name, first half of list)
else if name $>$ mid.name **then**
    return FIND(name, second half of list)
end if

- Careful how you implement this!
- **Copying** (half of) a list is **linear** $O(n)$

Hidden complexity

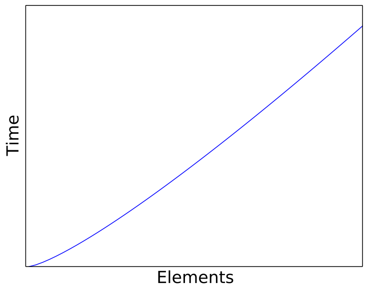
if name < mid.name **then**
    return FIND(name, first half of list)
else if name > mid.name **then**
    return FIND(name, second half of list)
end if



- Careful how you implement this!
- **Copying** (half of) a list is **linear** $O(n)$
- The actual running time would be $O(n \log n)$

Hidden complexity

if name $<$ mid.name **then**
    return FIND(name, first half of list)
else if name $>$ mid.name **then**
    return FIND(name, second half of list)
end if



- Careful how you implement this!
- **Copying** (half of) a list is **linear** $O(n)$
- The actual running time would be $O(n \log n)$
- Use **pointers** into the list instead of copying

Binary search done wrong

```
def binary_search(name, mylist):
    if mylist == []:
        raise ValueError("Not found")

    mid = len(mylist) / 2
    mid_name = mylist[mid_index].name

    if name == mid_name:
        return mid
    elif name < mid_name:
        return binary_search(name, mylist[:mid])
    else:
        return binary_search(name, mylist[mid+1:])
```

Binary search done right

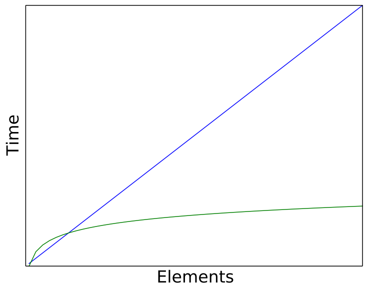
```
def binary_search(name, mylist, start, end):
    if end <= start:
        raise ValueError("Not found")

    mid = (start + end) / 2
    mid_name = mylist[mid].name

    if name == mid_name:
        return mylist[mid]
    elif name < mid_name:
        return binary_search(name, mylist, start, mid)
    else:
        return binary_search(name, mylist, mid+1, end)
```

Binary search vs linear search

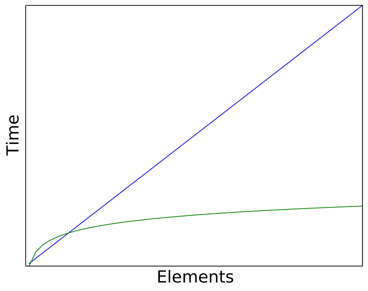
Socrative room code: `FALCOMPED`



- So binary search is better than linear search... right?

Binary search vs linear search

Socrative room code: `FALCOMPED`



- So binary search is better than linear search... right?
- Discuss in **pairs**
- On Socrative, post **one reason** why, or **one situation** where, linear search may be a better choice than binary search

Hashing

- Come up with a **hashing function** which maps elements to numbers

Hashing

- Come up with a **hashing function** which maps elements to numbers
- Example: assign $A = 1, B = 2, C = 3$ etc, and add them together

Hashing

- Come up with a **hashing function** which maps elements to numbers
- Example: assign $A = 1, B = 2, C = 3$ etc, and add them together
- Use these numbers to assign each element to a "bin" where it can be found

Hashing

- Come up with a **hashing function** which maps elements to numbers
- Example: assign $A = 1, B = 2, C = 3$ etc, and add them together
- Use these numbers to assign each element to a "bin" where it can be found

⋮	⋮
112	Ward, Jessica
113	Baker, Theresa
114	Collins, Jane
115	—
116	—
117	Hughes, Aaron
118	—
119	—
120	—
121	—
122	Brown, Janet
123	—
124	—
125	Gonzalez, Adam Lewis, Rose
126	—
127	—
128	—
129	—
130	—
131	—
132	Young, Frank
⋮	⋮

Hash look-up

98	Diaz, Harold
99	Parker, Debra Perez, Diana White, Amanda
112	Ward, Jessica
113	Baker, Theresa
114	Collins, Jane
117	Hughes, Aaron
122	Brown, Janet
125	Gonzalez, Adam Lewis, Rose
132	Young, Frank
135	Kelly, Philip
138	Cox, Shirley
142	Clark, Stephanie
144	Scott, Michelle
145	Miller, Jeremy
147	Davis, Marilyn
149	Lopez, Jeffrey
151	Anderson, Martha
158	Williams, Billy
162	Sanders, Phillip
171	Russell, Mildred
175	Stewart, Howard
183	Henderson, Lawrence

"Lopez, Jeffrey"

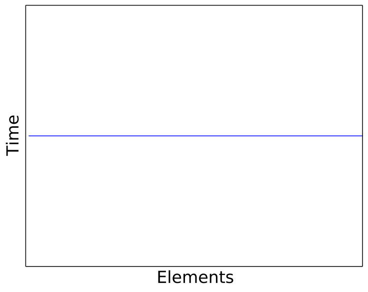
Hash look-up

98	Diaz, Harold
99	Parker, Debra Perez, Diana White, Amanda
112	Ward, Jessica
113	Baker, Theresa
114	Collins, Jane
117	Hughes, Aaron
122	Brown, Janet
125	Gonzalez, Adam Lewis, Rose
132	Young, Frank
135	Kelly, Philip
138	Cox, Shirley
142	Clark, Stephanie
144	Scott, Michelle
145	Miller, Jeremy
147	Davis, Marilyn
149	Lopez, Jeffrey
151	Anderson, Martha
158	Williams, Billy
162	Sanders, Phillip
171	Russell, Mildred
175	Stewart, Howard
183	Henderson, Lawrence

"Lopez, Jeffrey"

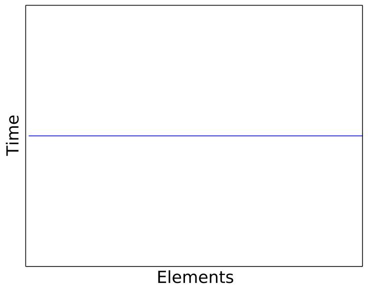
$12 + 15 + 16 + 5 + 26 + 10 + 5 + 6 + 6 + 18 + 5 + 25 = 149$

How long does it take?



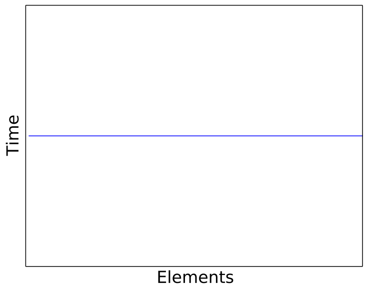
- If there are no "collisions", look-up time is **constant** or $O(1)$

How long does it take?



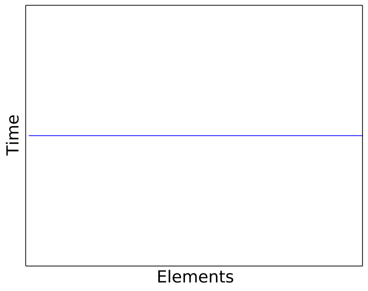
- If there are no "collisions", look-up time is **constant** or $O(1)$
  - (NB: constant **with respect to** n)

How long does it take?



- If there are no “collisions”, look-up time is **constant** or $O(1)$
  - (NB: constant **with respect to** n)
- I.e. doubling the size of the list **does not change** the look-up time

How long does it take?



- If there are no "collisions", look-up time is **constant** or $O(1)$
  - (NB: constant **with respect to** n)
- I.e. doubling the size of the list **does not change** the look-up time
- When there are collisions, need to fall back on something like linear or binary search within each bin

Don't reinvent the wheel!

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method
  - **List comprehension**, e.g.

```
[person for person in people if person.name
    == "Lopez, Jeffrey"]
```

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method
  - **List comprehension**, e.g.

```
[person for person in people if person.name
    == "Lopez, Jeffrey"]
```

- Binary search in Python:

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method
  - **List comprehension**, e.g.

```
[person for person in people if person.name
    == "Lopez, Jeffrey"]
```

- Binary search in Python:
  - The `bisect` module

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method
  - **List comprehension**, e.g.

```
[person for person in people if person.name
    == "Lopez, Jeffrey"]
```

- Binary search in Python:
  - The `bisect` module
- Hash tables in Python:

Don't reinvent the wheel!

- We are using search as an **example**, to learn the **principles** — in practice you should hardly ever implement your own search
- Linear search in Python:
  - `list.index()` method
  - **List comprehension**, e.g.

```
[person for person in people if person.name
    == "Lopez, Jeffrey"]
```

- Binary search in Python:
  - The `bisect` module
- Hash tables in Python:
  - The `dict` (dictionary) data structure

More on complexity

Common complexity classes

Common complexity classes

"Faster"	Constant	$O(1)$

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\vert$	Fractional power	$O(n^k)$, $k < 1$

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\vert$	Fractional power	$O(n^k), k < 1$
$\vert$	Linear	$O(n)$

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\vert$	Fractional power	$O(n^k)$, $k < 1$
$\vert$	Linear	$O(n)$
$\vert$	Quadratic	$O(n^2)$

FALMOUTH UNIVERSITY

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\mid$	Fractional power	$O(n^k)$, $k < 1$
$\mid$	Linear	$O(n)$
$\mid$	Quadratic	$O(n^2)$
$\mid$	Polynomial	$O(n^k)$, $k > 1$

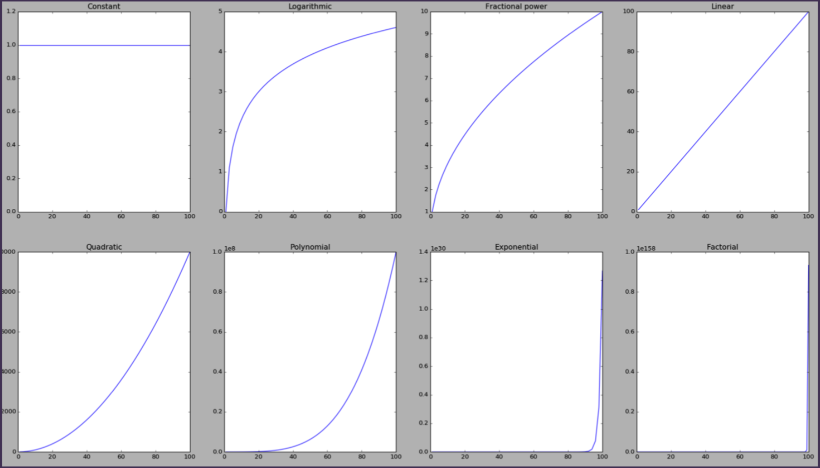
Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\vert$	Fractional power	$O(n^k), k < 1$
$\vert$	Linear	$O(n)$
$\vert$	Quadratic	$O(n^2)$
$\vert$	Polynomial	$O(n^k), k > 1$
$\downarrow$	Exponential	$O(e^n)$

Common complexity classes

"Faster"	Constant	$O(1)$
$\uparrow$	Logarithmic	$O(\log n)$
$\vert$	Fractional power	$O(n^k)$, $k < 1$
$\vert$	Linear	$O(n)$
$\vert$	Quadratic	$O(n^2)$
$\vert$	Polynomial	$O(n^k)$, $k > 1$
$\downarrow$	Exponential	$O(e^n)$
"Slower"	Factorial	$O(n!)$

Common complexity classes



Working with big O notation

Working with big O notation

- Can ignore **leading constants**

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$
- Take only the **dominant term**

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$
- Take only the **dominant term**
  - The term that is largest when n is large

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$
- Take only the **dominant term**
  - The term that is largest when n is large
  - If an algorithm takes $0.1n^3 + 300n^2 + 7000$ operations, it is $O(n^3)$

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$
- Take only the **dominant term**
  - The term that is largest when n is large
  - If an algorithm takes $0.1n^3 + 300n^2 + 7000$ operations, it is $O(n^3)$
- Multiply **compound** algorithms

Working with big O notation

- Can ignore **leading constants**
  - If one algorithm takes n^2 operations, another takes $500n^2$ and a third takes $0.00000001n^2$, all three are $O(n^2)$
- Take only the **dominant term**
  - The term that is largest when n is large
  - If an algorithm takes $0.1n^3 + 300n^2 + 7000$ operations, it is $O(n^3)$
- Multiply **compound** algorithms
  - If an algorithm does n "things" and each "thing" is $O(n)$, then the overall algorithm is $O(n^2)$

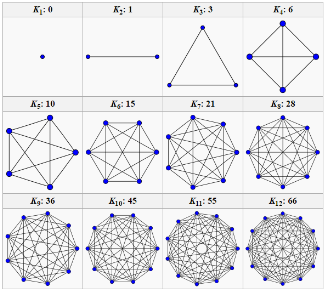
Quadratic complexity

- Collision detection between *n* objects

Quadratic complexity

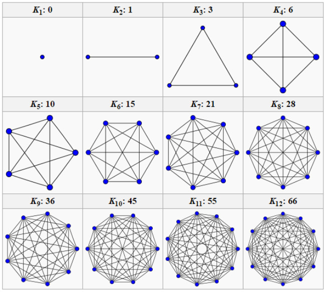
- Collision detection between n objects
- The naïve way: check **each pair** of objects to see whether they have collided

Quadratic complexity



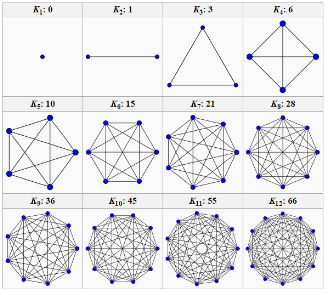
- Collision detection between n objects
- The naïve way: check **each pair** of objects to see whether they have collided
- This is **quadratic** or $O(n^2)$

Quadratic complexity



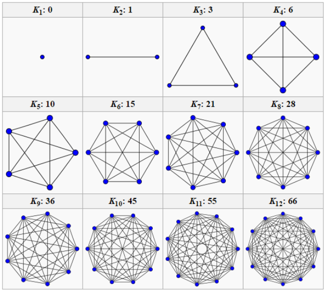
- Collision detection between n objects
- The naïve way: check **each pair** of objects to see whether they have collided
- This is **quadratic** or $O(n^2)$
- Doubling the number of objects would **quadruple** the time required!

Quadratic complexity



- Collision detection between n objects
- The naïve way: check **each pair** of objects to see whether they have collided
- This is **quadratic** or $O(n^2)$
- Doubling the number of objects would **quadruple** the time required!
- Cleverer methods exist that are more scalable

Quadratic complexity



- Collision detection between n objects
- The naïve way: check **each pair** of objects to see whether they have collided
- This is **quadratic** or $O(n^2)$
- Doubling the number of objects would **quadruple** the time required!
- Cleverer methods exist that are more scalable
  - Further reading: spatial hashing, quadtrees, octrees, Verlet lists

Exponential complexity

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself
- Given an n-bit number $m = pq$ that is a product of two primes p and q, find p and q.

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself
- Given an n-bit number $m = pq$ that is a product of two primes p and q, find p and q.

for $p = 2, 3, \ldots, m$ **do**
  $q \leftarrow m/p$
  if q is an integer **then**
    return p, q
  end if
end for

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself
- Given an n-bit number $m = pq$ that is a product of two primes p and q, find p and q.

for $p = 2, 3, \ldots, m$ **do**
  $q \leftarrow m/p$
  if q is an integer **then**
    return p, q
  end if
end for

- Since $m \leq 2^n - 1$, in the worst case this is $O(2^n)$

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself
- Given an n-bit number $m = pq$ that is a product of two primes p and q, find p and q.

for $p = 2, 3, \ldots, m$ **do**
    $q \leftarrow m/p$
    if q is an integer **then**
        return p, q
    end if
end for

- Since $m \leq 2^n - 1$, in the worst case this is $O(2^n)$
  - Actually even slower because division is not $O(1)$

Exponential complexity

- A prime number is a number that is divisible only by 1 and itself
- Given an n-bit number $m = pq$ that is a product of two primes p and q, find p and q.

for $p = 2, 3, \ldots, m$ **do**
  $q \leftarrow m/p$
  if q is an integer **then**
    return p, q
  end if
end for

- Since $m \leq 2^n - 1$, in the worst case this is $O(2^n)$
  - Actually even slower because division is not $O(1)$
- Adding 1 to n potentially **doubles** the running time!

Aside: a famous unanswered question in computing

Aside: a famous unanswered question in computing

- A problem is “in P” if it can be solved with an algorithm running in $O(n^k)$ time

Aside: a famous unanswered question in computing

- A problem is "in P" if it can be solved with an algorithm running in $O(n^k)$ time
- A problem is in NP if a potential solution can be checked in $O(n^k)$ time

Aside: a famous unanswered question in computing

- A problem is "in *P*" if it can be solved with an algorithm running in $O(n^k)$ time
- A problem is in *NP* if a potential solution can be checked in $O(n^k)$ time
  - Equivalently, it can be solved with an algorithm running in $O(n^k)$ time on an infinitely parallel machine

Aside: a famous unanswered question in computing

- A problem is "in *P*" if it can be solved with an algorithm running in $O(n^k)$ time
- A problem is in *NP* if a potential solution can be checked in $O(n^k)$ time
  - Equivalently, it can be solved with an algorithm running in $O(n^k)$ time on an infinitely parallel machine
- Are there any problems in *NP* but not in *P*?

P versus NP

P versus NP

- If you can find a **mathematical proof** that either $P = NP$ or $P \neq NP$, there's a \$1 million prize...

P versus NP

- If you can find a **mathematical proof** that either $P = NP$ or $P \neq NP$, there's a \$1 million prize...
- It is believed that $P \neq NP$, so large instances of NP-hard problems are not solvable in a feasible amount of time

P versus NP

- If you can find a **mathematical proof** that either $P = NP$ or $P \neq NP$, there's a $1 million prize...
- It is believed that $P \neq NP$, so large instances of NP-hard problems are not solvable in a feasible amount of time
  - Many types of cryptography are based on this assumption

P versus NP

- If you can find a **mathematical proof** that either $P = NP$ or $P \neq NP$, there's a $1 million prize...
- It is believed that $P \neq NP$, so large instances of *NP*-hard problems are not solvable in a feasible amount of time
  - Many types of cryptography are based on this assumption
  - Quantum computers are "infinitely parallel" in a sense so *can* solve some large *NP*-hard problems

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements
  - Naïve collision detection is fine if your game only ever has 4 objects on screen

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements
  - Naïve collision detection is fine if your game only ever has 4 objects on screen
  - Sometimes complexity in terms of other resources (e.g. space, bandwidth) are more important than time

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements
  - Naïve collision detection is fine if your game only ever has 4 objects on screen
  - Sometimes complexity in terms of other resources (e.g. space, bandwidth) are more important than time
- Software development is all about choosing **the right tool for the job**

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements
  - Naïve collision detection is fine if your game only ever has 4 objects on screen
  - Sometimes complexity in terms of other resources (e.g. space, bandwidth) are more important than time
- Software development is all about choosing **the right tool for the job**
  - If you need scalability, choose a scalable algorithm

Caveats

- Time complexity only tells us how an algorithm **scales** with the size of the input
  - If we know the input will always be **small**, time complexity is not so important
  - Linear search is quicker than binary search if you only ever have 3 elements
  - Naïve collision detection is fine if your game only ever has 4 objects on screen
  - Sometimes complexity in terms of other resources (e.g. space, bandwidth) are more important than time
- Software development is all about choosing **the right tool for the job**
  - If you need scalability, choose a scalable algorithm
  - Otherwise, choose simplicity

Summary

- Time complexity tells us how the running time of an algorithm **scales** with the size of the data it is given

Summary

- Time complexity tells us how the running time of an algorithm **scales** with the size of the data it is given
- Choice of data structures and algorithms can have a large impact on the efficiency of your software

Summary

- Time complexity tells us how the running time of an algorithm **scales** with the size of the data it is given
- Choice of data structures and algorithms can have a large impact on the efficiency of your software
- ... but only if scalability is actually a factor

Computability

Computability theory

Computability theory

- Let A and B be **sets** of elements

Computability theory

- Let A and B be **sets** of elements
  - NB: A may be **infinite**

Computability theory

- Let A and B be **sets** of elements
  - NB: A may be **infinite**
- A function $f : A \to B$ is **computable** if there exists a Turing machine which computes f

Computability theory

- Let A and B be **sets** of elements
  - NB: A may be **infinite**
- A function $f : A \to B$ is **computable** if there exists a Turing machine which computes f
  - I.e. given an encoding of $a \in A$ as input, the Turing machine outputs an encoding of $f(a)$

An uncomputable function

The **halting problem**

An uncomputable function

The **halting problem**

- A = the set of all Turing machines

An uncomputable function

The **halting problem**

- A = the set of all Turing machines
- $B = \{\text{true}, \text{false}\}$

An uncomputable function

The **halting problem**

- A = the set of all Turing machines
- $B = \{\text{true}, \text{false}\}$
- $f(a) = \begin{cases} \text{true} & \text{if } a \text{ halts in finite time on all inputs} \\ \text{false} & \text{otherwise} \end{cases}$

An uncomputable function

The **halting problem**

- A = the set of all Turing machines
- $B = \{\text{true}, \text{false}\}$
- $f(a) = \begin{cases} \text{true} & \text{if } a \text{ halts in finite time on all inputs} \\ \text{false} & \text{otherwise} \end{cases}$
- There is **no** Turing machine that computes f

An uncomputable function

The **halting problem**

- A = the set of all Turing machines
- $B = \{\text{true}, \text{false}\}$
- $f(a) = \begin{cases} \text{true} & \text{if } a \text{ halts in finite time on all inputs} \\ \text{false} & \text{otherwise} \end{cases}$
- There is **no** Turing machine that computes f
- f is **uncomputable**

Turing completeness

Turing completeness

- A system (e.g. a computer or programming language) is **Turing complete** if it can implement any given Turing machine

Church-Turing Thesis

Church-Turing Thesis

- If a function is **effectively calculable**, then it is **computable** by a Turing machine

Church-Turing Thesis

- If a function is **effectively calculable**, then it is **computable** by a Turing machine
- Effectively calculable = there is a method or algorithm for computing it

Church-Turing Thesis

- If a function is **effectively calculable**, then it is **computable** by a Turing machine
- Effectively calculable = there is a method or algorithm for computing it
- So in terms of computability, Turing machines are as powerful as computers can be

Halting revisited

Halting revisited

- Write a software tool that, given a Python program, predicts whether that program can go into an infinite loop

Halting revisited

- Write a software tool that, given a Python program, predicts whether that program can go into an infinite loop
- Your tool must work for **all** Python programs

Halting revisited

- Write a software tool that, given a Python program, predicts whether that program can go into an infinite loop
- Your tool must work for **all** Python programs
- Is this possible?