Algorithm Questions

An algorithm is a clearly specified set of instructions to solve a problem. It is defined by its input, its desired output, and its (precise) description, usually in pseudocode.

1. Does my algorithm work? (is it provably correct?)
2. Does my algorithm work efficiently?
   a. How much time does it take? (what is its time complexity?)
   b. How much space does it need? (what is its space complexity?)

In this course, we’re mostly concerned with time complexity. Proving algorithms correct is the subject of a future course (280-algorithms). We’ll only touch on it occasionally in 151.

Time Complexity

Why should we care about the time complexity of an algorithm?

1. To know how fast it is. Duh.
2. To be able to compare it to other algorithms.

Idea #1:

- implement each candidate
- run them
- time the results

Q: Is there anything wrong with this approach?

Idea #2:

- find a function that expresses running time as a function of problem size
- should be able to determine this function without actual implementation
- this function should be independent of hardware and software environments
- should be able to determine this function without actual implementation
- this function should be independent of problem size

So, we want a standardized way of determining the time complexity of an algorithm.

2. To be able to compare it to other algorithms.
1. To know how fast it is. Duh.

Why should we care about the time complexity of an algorithm?
Examples

A More Complicated Example

```python
ex. sum = 0
for i = 1 to n do
    sum = sum + i
```

```python
ex. for i = 1 to n do
    for j = 1 to n do
        sum = sum + i*j
```

```python
ex. for i = 1 to n do
    for j = 1 to m do
        a[i][j] = i*j
```

```python
ex. for i = 1 to n do
    for j = i to n do
        foo.bar()
```

```python
ex. while n > 1 do
    n = n/2
```

Algorithm to get max value in an array

```python
def find_max(A):
    max = A[0]
    for i = 1 to |A|-1 do
        if A[i] > max then
            max = A[i]
    return max
```

For example: if the max value in A is A[0] then we never enter the while loop.

The running time may change for different input arrays A.

The algorithm we have for finding the max value in an array A.

```python
def find_max(A):
    max = A[0]
    for i = 1 to |A|-1 do
        if A[i] > max then
            max = A[i]
    return max
```

A More Complicated Example

Some of the running time functions we've seen are:

- **Constant** $T(n) = c$
- **Logarithmic** $T(n) = c_1 \log_2 n + c_2$
- **Linear** $T(n) = c_1 n + c_2$
- **Logarithmic** $T(n) = c_1 n \log_2 n + c_2$
- **Quadratic** $T(n) = c_1 n^2 + c_2 n + c_3$
- **Cubic** $T(n) = c_1 n^3 + c_2 n^2 + c_3 n + c_4$
- **Exponential** $T(n) = c_1 2^n$
- **Factorial** $T(n) = c_1 n!$

Glass half-full or half-empty?

So, in analyzing time complexity, should we be

a. optimistic?

b. pessimistic?

c. neither?

Computer scientists do all three.

a. optimistic?

We call this best-case analysis.

b. pessimistic?

We call this worst-case analysis.

c. neither?

We can do average-case analysis.

Is known for good measure. We'll save the optimism for other people.

We'll mostly focus on worst-case analysis, with the occasional average case.

A glass half-full or half-empty?
So, if we really only care about efficiency as input size tends to infinity...
• we can ignore multiplicative constants, e.g. $2n$, $3n$, $1000n$ all same as $n$
• we can ignore lower-order additive terms, e.g. $n^2 + n$ same as $n^2$
• under this system, $1000n$ is somehow better than $n^2$
• (which, it is, if $n$ gets large enough!)
• under this system, $1000000n^2 + 1000n$ is equal to $n^2$. (well, they're both better than $n^3$ and worse than $n\log n$)

Obviously, this is not fool-proof...
Recall:

\( T(n) \) is \( O(f(n)) \) if there are positive constants \( c, n_0 \) such that

\[ T(n) \leq c \cdot f(n) \]

for all \( n \geq n_0 \). That is, \( c \cdot f(n) \) is an upper bound on \( T(n) \) for all \( n \geq n_0 \).

Definition:

\( T(n) \) is \( \Omega(f(n)) \) if there are positive constants \( c, n_0 \) such that

\[ T(n) \geq c \cdot f(n) \]

for all \( n \geq n_0 \). That is, \( c \cdot f(n) \) is a lower bound on \( T(n) \) for all \( n \geq n_0 \).

So \( n \) is \( O(n^2) \), but \( n \) is not \( \Omega(n^2) \). And \( n^2 \) is \( \Omega(n^2) \), but \( n^2 \) is not \( O(n^2) \).

That is, \( c \cdot f(n) \) is a matching upper and lower bound on \( T(n) \).

Best case: Item is in the middle, and this algorithm takes time \( O(1) \).

Worst case: Item is not in the list at all, and algorithm is \( \Omega(n) \) because we repeatedly divide our search space into 2 (can only do this log \( n \) times).

\( \text{linearSearch}(\text{list } L, \text{item } x) \)

for \( i = 1 \) to \(|L| - 1\) do

if \( A[i] = x \) then

return true

return false

Input: a list \( L \) of \( n \) items, a desired item \( x \)

Goal: determine whether \( x \) is in the list \( L \)

Best case: \( X \) is at the front, and this algorithm takes time \( O(1) \).

Worst case: \( X \) is at the end, or is not in the list at all, and algorithm is \( O(n) \).

Thus we consider this algorithm to be \( O(n) \).

\( \text{binarySearch}(\text{sorted list } L, \text{item } x) \)

\( \text{found} = \text{false} \), \( \text{low} = 0 \), \( \text{high} = |L| - 1 \)

while \( \text{low} < \text{high} \)

\( \text{mid} = (\text{low} + \text{high}) / 2 \)

if \( x = A[\text{mid}] \) then

return true

if \( x < A[\text{mid}] \) then

\( \text{high} = \text{mid} - 1 \)

else

\( \text{low} = \text{mid} + 1 \)

return false

Input: a sorted list \( L \) of \( n \) items, a desired item \( x \)

Goal: determine whether \( x \) is in the list \( L \)

Best case: \( X \) is in the middle, and this algorithm takes time \( O(1) \).

Worst case: Item is not in the list at all, and algorithm is \( \Omega(n) \) because we repeatedly divide our search space into 2 (can only do this \( \log n \) times).

\( \Theta \) notation:

Best case: \( X \) is at the front, and this algorithm takes time \( O(1) \).

Worst case: \( X \) is at the end, or is not in the list at all, and algorithm is \( \Omega(n) \).

Thus we consider this algorithm to be \( \Theta(n) \).

\( \text{binarySearch}(\text{sorted list } L, \text{item } x) \)

\( \text{found} = \text{false} \), \( \text{low} = 0 \), \( \text{high} = |L| - 1 \)

while \( \text{low} < \text{high} \)

\( \text{mid} = (\text{low} + \text{high}) / 2 \)

if \( x = A[\text{mid}] \) then

return true

if \( x < A[\text{mid}] \) then

\( \text{high} = \text{mid} - 1 \)

else

\( \text{low} = \text{mid} + 1 \)

return false

Input: a sorted list \( L \) of \( n \) items, a desired item \( x \)

Goal: determine whether \( x \) is in the list \( L \)

Best case: \( X \) is in the middle, and this algorithm takes time \( O(1) \).

Worst case: Item is not in the list at all, and algorithm is \( \Omega(n) \).

Thus we consider this algorithm to be \( \Omega(n) \).

\( \text{binarySearch}(\text{sorted list } L, \text{item } x) \)

\( \text{found} = \text{false} \), \( \text{low} = 0 \), \( \text{high} = |L| - 1 \)

while \( \text{low} < \text{high} \)

\( \text{mid} = (\text{low} + \text{high}) / 2 \)

if \( x = A[\text{mid}] \) then

return true

if \( x < A[\text{mid}] \) then

\( \text{high} = \text{mid} - 1 \)

else

\( \text{low} = \text{mid} + 1 \)

return false

Input: a sorted list \( L \) of \( n \) items, a desired item \( x \)

Goal: determine whether \( x \) is in the list \( L \)

Best case: \( X \) is at the front, and this algorithm takes time \( O(1) \).

Worst case: \( X \) is at the end, or is not in the list at all, and algorithm is \( \Omega(n) \).

Thus we consider this algorithm to be \( \Omega(n) \).