Linear Search

Input: Sequence $A = a_1, a_2, \ldots, a_n$; element x
Output: YES if x occurs in A; NO otherwise

```
for i=1 to n do
  if a_i = x then
    return YES
  fi
od
return NO
```

Binary search

Input: Sorted sequence A; element x
Output: YES if x occurs in A, NO o.w.

```
low = 1 ; high = n
while low < high do
  mid = ⌊(low + high)/2⌋
  if x > a_mid then
    low = mid+1
  else
    high = mid
  fi
od
if x = a_low then
  return YES
else
  return NO
fi
```

Which is better?

What do we mean by 'better'?

- correctness
- ease of understanding / implementation
- running time
- memory usage

~~We will look at~~
We are going to compare running times. This typically only makes sense if we have multiple correct solutions to the same problem.

Things that affect running time:

- CPU speed
- Operating system
- Cache size
- Branch prediction
- Programming language
- Cache misses
- Background processes
- CPU Temperature
- Number of operations

Complexity Analysis

Counting the number of elementary operations an algorithm performs for a certain input size.
We are typically interested in the worst-case.

An elementary operation is:

- $+, -, \times, \div, <, >, =$
- read/write memory
- logical bit operations

This is a model of computation, called the unit cost random access machine.

Analysis of Linear Search

```
for i:=1 to n do              Σ_{i=1}^{n} (1+2)    => 3n+1
  if a_i=x then               1 )
    return YES ....           0 ) 1
  fi
od
return NO ....                1
```

(increment i; test i≤n)

- Count inside-out: start with loop body.
- Assume worst-case ~~beha~~ input.
- If inside of a for-loop performs $f(i,n)$ operations,

```
for i:=a to b do
  ...
od
```

performs $\sum_{i=a}^{b} f(i,n)$ operations.

~~Suppose we have another other algorithms that solve the problem in~~

Analysis of Binary Search

```
initialize             a  )
while low < high do    ?? )
  stuff                b  )   a + f(n)·b + c
od                        )        k
finalize               c  )
```

How many times does the loop body execute?
- Every time, our seach interval is <u>halved</u>.
- We can halve n $\log_2 n$ times.

Thus, the running time is $b \cdot \log_2 n + a + c$, for $a,b,c \in \mathbb{N}$.

Suppose we have other algorithms that solve the problem in $\sqrt{n}$ ~~operations~~, $\frac{1}{100}n^2$ and 2^n operations. Our computer executes 10^{11} ~~instructions~~ operations per second. How long do they take?

n	100	1 million	1000 million	10^{12} (1 TB)	10^{80} (#atoms in universe)
$\log_2 n$	<1 ns	<1 ms	<1 ms	<1 ms	<1 ms
$\sqrt{n}$	<1 ns	<1 ms	<1 ms	<1 ms	> age of the universe
$3n+1$	<1 ns	<1 ms	30 ms	30 s	"
$\frac{1}{100}n^2$	<1 ns	100 ms	> 1 day	> 3 millennia	"
2^n	~29 × age of the universe				

For big inputs, we care about the ~~growth~~ asymptotic growth of the running time.

$3n+1$ "grows like" n
$\frac{1}{100}n^2$ "grows like" n^2

$\Rightarrow$ Constant factors and lower order terms don't matter.

Formally, we write this using big-O notation:
Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say that $f(n)$ is $O(g(n))$ if ~~$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$~~ there exist constants $c, n_0 \in \mathbb{R}^+$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example: $f(n) = \frac{1}{100}n^2$ is $O(n^2)$

Proof: ~~$\frac{1}{100}n^2 \leq c \cdot n^2$ for $c = \frac{1}{100}$ and $n_0 = 0$.~~
$f(n) \leq \underbrace{\frac{1}{100}}_{c>0} n^2$ for all $n \geq \underbrace{1}_{n_0>0}$

Therefore $f(n)$ is $O(n^2)$ with $c = \frac{1}{100}$ and $n_0 = 1$.

Example: $f(n) = 3n+1$ is $O(n)$

$f(n) = 3n+1$

$\leq 3n+n$ (for $n \geq 1$)

$= 4n$

Thus, $f(n) \leq 4n$ for all $n \geq 1$. Therefore $f(n)$ is $O(n)$ with $c = 4$ and $n_0 = 1$.

Note: think of big-O as an upper bound ($\leq$).

Example: $f(n) = 3n-5$ is $O(n^5)$

$f(n) = 3n-5$

$\leq 3n$

$\leq 3n^5$ (for $n \geq 1$)

Therefore $f(n)$ is $O(n^5)$ with $c = 3$ and $n_0 = 1$.

Strategy: (for proving big-O)

1. Eliminate negative terms
2. Multiply to match highest order term

Exercise: $f(n) = 2n^2 - 4n + 3$ is $O(n^2)$

$f(n) = 2n^2 - 4n + 3$

$\leq 2n^2 + 3$

$\leq 2n^2 + 3n^2$ (for $n \geq 1$)

$= 5n^2$

Therefore $f(n)$ is $O(n^2)$ with $c = 5$ and $n_0 = 1$.

Sometimes it is also useful to have a lower bound. We write this with big-Omega:

$f(n)$ is $\Omega(g(n))$ if $\exists c, n_0 \in \mathbb{R}^+ \left(\forall n \geq n_0 \left(f(n) \geq c \cdot g(n) \right) \right)$

Example: $f(n) = 3n+1$ is $\Omega(n)$

Proof: $f(n) = 3n+1$
$\geq 3n$.

Therefore $f(n)$ is $\Omega(n)$ with $c=3$ and $n_0=1$.

Example: $f(n) = 3n^2 - 13n + 18$ is $\Omega(n^2)$

Proof: $f(n) = 3n^2 - 13n + 18$
$\geq 3n^2 - 13n$
$= 2n^2 + (n^2 - 13n)$
$\geq 2n^2$ when $n^2 - 13n \geq 0$

So $f(n) \geq 2n^2$ when $n^2 - 13n \geq 0$. Does there exist an n_0 such that this holds for all $n \geq n_0$?

$n^2 - 13n \geq 0$
$n^2 \geq 13n$
$n \geq 13$ (for $n \geq 1$)

Yes! $n^2 - 13n \geq 0$ holds for $n \geq 13$. Therefore $f(n)$ is $\Omega(n^2)$ with $c=2$ and $n_0 = 13$.

Strategy (for proving big-Omega)

1. Get rid of positive lower-order terms
2. Multiply negative terms with small factors to match the highest order term
3. Split a bit off the highest order term and solve the remaining sub-problem

Example: $4n^3 - 6n^2 + 3n - 3$ is $\Omega(n^3)$

Proof:

$$\begin{aligned} 4n^3 - 6n^2 + 3n - 3 &\geq 4n^3 - 6n^2 - 3 \\ &\geq 4n^3 - 6n^2 - 3n^3 \quad (\text{for } n \geq 1) \\ &= n^3 - 6n^2 \\ &= \tfrac{1}{2}n^3 + (\tfrac{1}{2}n^3 - 6n^2) \\ &\geq \tfrac{1}{2}n^3 \quad (\text{when } \tfrac{1}{2}n^3 - 6n^2 \geq 0) \end{aligned}$$

$$\begin{aligned} \tfrac{1}{2}n^3 - 6n^2 &\geq 0 \\ \tfrac{1}{2}n^3 &\geq 6n^2 \\ \tfrac{1}{2}n &\geq 6 \\ n &\geq 12 \end{aligned}$$

Therefore $4n^3 - 6n^2 + 3n - 3$ is $\Omega(n^3)$ with $c = \frac{1}{2}$ and $n_0 = 12$.

Big-Theta

If a function $f(n)$ is <u>both</u> $O(g(n))$ and $\Omega(g(n))$, we say $f(n)$ is $\Theta(g(n))$.

End of L8