Analysis of Algorithms

Running Time

An algorithm is a step-by-step procedure for solving a problem in a finite amount of time. Most algorithms map input to output:
- running time typically grows with input size
- average time often difficult to determine
- Focus on worst case running time:
  - easier to analyse
  - crucial to many applications: finance, robotics, games, …

Empirical Analysis

1. Write program that implements an algorithm
2. Run program with inputs of varying size and composition
3. Measure the actual running time
4. Plot the results

Theoretical Analysis

- Uses high-level description of the algorithm instead of implementation ("pseudocode")
- Characterises running time as a function of the input size, $n$
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: Find maximal element in an array

arrayMax(A):

<table>
<thead>
<tr>
<th>Input</th>
<th>array A of n integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>maximum element of A</td>
</tr>
</tbody>
</table>

- `currentMax=A[0]`
- `for all i=1..n-1 do`
- `if A[i]>currentMax then`
- `currentMax=A[i]`
- `end if`
- `end for`
- `return currentMax`

Control flow

- `if … then … |else| … end if`
- `while .. do … end while`
- `repeat … until`
- `for [all][each] .. do … end for`
Exercise #1: Pseudocode

Formulate the following verbal description in pseudocode:

In the first phase, we iteratively pop all the elements from stack \(S\) and enqueue them in queue \(Q\), then dequeue the element from \(Q\) and push them back onto \(S\).

As a result, all the elements are now in reversed order on \(S\).

In the second phase, we again pop all the elements from \(S\), but this time we also look for the element \(x\).

By again passing the elements through \(Q\) and back onto \(S\), we reverse the reversal, thereby restoring the original order of the elements on \(S\).

Sample solution:

```pseudocode
while ¬empty(S) do
    pop e from S, enqueue e into Q
end while
while ¬empty(Q) do
    dequeue e from Q, push e onto S
end while
found=false
while ¬empty(S) do
    pop e from S, enqueue e into Q
    if e=x then
        found=true
    end if
end while
while ¬empty(Q) do
    dequeue e from Q, push e onto S
end while
```

Exercise #2: Pseudocode

Implement the following pseudocode instructions in C

1. \(A\) is an array of ints

   ```c
   int temp = A[i];
   A[i] = A[j];
   A[j] = temp;
   ```

2. head points to beginning of linked list

   ```c
   NodeT *succ = head->next;
   head->next = succ->next;
   succ->next = head;
   head = succ;
   ```

3. \(S\) is a stack

   ```c
   x = StackPop(S);
   y = StackPop(S);
   StackPush(S, x);
   StackPush(S, y);
   ```

The following pseudocode instruction is problematic. Why?

```c
... swap the two elements at the front of queue \(Q\) ...
```

The Abstract RAM Model

RAM = Random Access Machine

- A CPU (central processing unit)
- A potentially unbounded bank of memory cells
  - each of which can hold an arbitrary number, or character
- Memory cells are numbered, and accessing any one of them takes CPU time

Primitive Operations
Basic computations performed by an algorithm
Identifiable in pseudocode
Largely independent of the programming language
Exact definition not important (we will shortly see why)
Assumed to take a constant amount of time in the RAM model

Examples:
- Evaluating an expression
- Indexing into an array
- Calling/returning from a function

Counting Primitive Operations
By inspecting the pseudocode ...

we can determine the maximum number of primitive operations executed by an algorithm
as a function of the input size

Example:

arrayMax(A):

<table>
<thead>
<tr>
<th>Input</th>
<th>array A of n integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>maximum element of A</td>
</tr>
</tbody>
</table>

currentMax = A[0]

for all i = 1 .. n - 1 do

| if A[i] > currentMax then | 2(n-1) |
| currentMax = A[i]         | n-1    |

end if

end for

return currentMax

\[ \text{Total} = 5n-2 \]

Estimating Running Times
Algorithm arrayMax requires \( 5n - 2 \) primitive operations in the worst case

- Best case requires \( 4n - 1 \) operations (why?)

Define:

- \( a \) … time taken by the fastest primitive operation
- \( b \) … time taken by the slowest primitive operation

Let \( T(n) \) be worst-case time of arrayMax. Then

\[ a(5n - 2) \leq T(n) \leq b(5n - 2) \]

Hence, the running time \( T(n) \) is bound by two linear functions

- Examples:
  - \( 10^2n + 10^5 \) is a linear function
  - \( 10^7n^2 + 10^8n \) is a quadratic function

Estimating Running Times
Seven commonly encountered functions for algorithm analysis

- Constant \( \approx 1 \)
- Logarithmic \( \approx \log n \)
- Linear \( \approx n \)
- N-Log-N \( \approx n \log n \)
- Quadratic \( \approx n^2 \)
- Cubic \( \approx n^3 \)
- Exponential \( \approx 2^n \)

In a log-log chart, the slope of the line corresponds to the growth rate of the function

The growth rate is not affected by constant factors or lower-order terms

- Examples:
  - \( 10^2n + 10^5 \) is a linear function
  - \( 10^7n^2 + 10^8n \) is a quadratic function
Changing the hardware/software environment affects $T(n)$ by a constant factor but does not alter the growth rate of $T(n)$.

$\Rightarrow$ Linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm `arrayMax`

### Exercise #3: Estimating running times

Determine the number of primitive operations

```
matrixProduct(A,B):
    Input  n×n matrices A, B
    Output n×n matrix A·B

    for all i=1..n do
        for all j=1..n do
            C[i,j]=0
            for all k=1..n do
                C[i,j]=C[i,j]+A[i,k]·B[k,j]
            end for
        end for
    end for
    return C
```

### Exercise #4: Estimating running times

```
matrixProduct(A,B):
    Input  n×n matrices A, B
    Output n×n matrix A·B

    for all i=1..n do
        for all j=1..n do
            C[i,j]=0
            for all k=1..n do
                C[i,j]=C[i,j]+A[i,k]·B[k,j]
            end for
        end for
    end for
    return C
```

### Big-Oh

**Big-Oh Notation**

Given functions $f(n)$ and $g(n)$, we say that

$$f(n) \text{ is } O(g(n))$$

if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

**Example: function $2n + 10$ is $O(n)$**

- $2n+10 \leq c \cdot n$
  $\Rightarrow (c-2)n \geq 10$
  $\Rightarrow n \geq 10/(c-2)$
- pick $c=3$ and $n_0=10$

**Example: function $n^2$ is not $O(n)$**

- $n^2 \leq c \cdot n$
  $\Rightarrow n \leq c$
- inequality cannot be satisfied since $c$ must be a constant
Exercise #5: Big-Oh

Show that

1. $7n - 2$ is $O(n)$
   
   need $c>0$ and $n_0$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$
   
   true for $c=7$ and $n_0=1$

2. $3n^3 + 20n^2 + 5$ is $O(n^3)$
   
   need $c>0$ and $n_0$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
   
   true for $c=4$ and $n_0=21$

3. $3\log n + 5$ is $O(\log n)$
   
   need $c>0$ and $n_0$ such that $3\log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
   
   true for $c=8$ and $n_0=2$

Big-Oh and Rate of Growth

- Big-Oh notation gives an upper bound on the growth rate of a function
  - "$f(n) \leq g(n)"$ means growth rate of $f(n)$ no more than growth rate of $g(n)$
  - use Big-Oh to rank functions according to their rate of growth

<table>
<thead>
<tr>
<th>f(n) is $O(g(n))$</th>
<th>g(n) is $O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>g(n) grows faster</td>
<td>yes</td>
</tr>
<tr>
<td>f(n) grows faster</td>
<td>no</td>
</tr>
<tr>
<td>same order of growth</td>
<td>yes yes</td>
</tr>
</tbody>
</table>

Big-Oh Rules

- If $f(n)$ is a polynomial of degree $d$ $\Rightarrow f(n) \leq O(n^d)$
  - lower-order terms are ignored
  - constant factors are ignored
- Use the smallest possible class of functions
  - say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"  
  - Use the simplest expression of the class
  - say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Exercise #6: Big-Oh

Show that $\sum_{i=1}^{n} i$ is $O(n^2)$

$\sum_{i=1}^{n} i = \frac{n(n + 1)}{2} = \frac{n^2 + n}{2}$

which is $O(n^2)$

Asymptotic Analysis of Algorithms

Asymptotic analysis of algorithms determines running time in Big-Oh notation:

- find worst-case number of primitive operations as a function of input size
- express this function using Big-Oh notation

Example:
- algorithm `arrayMax` executes at most $5n - 2$ primitive operations
  $\Rightarrow$ algorithm `arrayMax` "runs in $O(n)$ time"

Constant factors and lower-order terms eventually dropped
$\Rightarrow$ can disregard them when counting primitive operations

Example: Computing Prefix Averages

- The $i$-th prefix average of an array $X$ is the average of the first $i$ elements:
  
  $A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1}$

NB. computing the array $A$ of prefix averages of another array $X$ has applications in financial analysis

... Example: Computing Prefix Averages
A quadratic algorithm to compute prefix averages:

prefixAverages1(X):
| Input array X of n integers
| Output array A of prefix averages of X
| for all i=0..n-1 do O(n)
| s=X[0] O(n)
| for all j=1..i do O(n^2)
| s=s+X[j] O(n^2)
| end for O(n)
| A[i]=s/(i+1) O(n)
| end for O(n)
| return A O(1)

2·O(n^2) + 3·O(n) + O(1) = O(n^2)

⇒ Time complexity of algorithm prefixAverages1 is O(n^2)

... Example: Computing Prefix Averages

The following algorithm computes prefix averages by keeping a running sum:

prefixAverages2(X):
| Input array X of n integers
| Output array A of prefix averages of X
| s=0 O(n)
| for all i=0..n-1 do O(n)
| s=s+X[i] O(n)
| A[i]=s/(i+1) O(n)
| end for O(n)
| return A O(1)

Thus, prefixAverages2 is O(n)

Example: Binary Search

The following recursive algorithm searches for a value in a sorted array:

search(v,a,lo,hi):
| Input value v
| array a[lo..hi] of values
| Output true if v in a[lo..hi]
| false otherwise
| mid=(lo+hi)/2
| if lo>hi then return false
| if a[mid]=v then
| return true
| else if a[mid]<v then
| return search(v,a,mid+1,hi)
| else
| return search(v,a,lo,mid-1)
| end if
for best case, \( C_n = 1 \)
for \( a[i..j], j < i \) (length=0)
\( C_0 = 0 \)
for \( a[i..j], i \leq j \) (length=n)
\( C_n = 1 + C_n/2 \Rightarrow C_n = \log_2 n \)

Thus, binary search is \( O(\log_2 n) \) or simply \( O(\log n) \) (why?)

... Example: Binary Search

Why logarithmic complexity is good:

Math Needed for Complexity Analysis

- Summations
- Logarithms
  \( \log_b (xy) = \log_b x + \log_b y \)
  \( \log_b (x/y) = \log_b x - \log_b y \)
  \( \log_b x^a = a \log_b x \)
  \( \log_b x = \log_a x / \log_a b \)
- Exponentials
  \( a^{b+c} = a^b a^c \)
  \( a^{bc} = (a^b)^c \)
  \( a^b / a^c = a^{b-c} \)
  \( b = a^\log_a b \)
  \( b^c = \mu^{\log_\mu b} \)
- Proof techniques
- Summation (addition of sequences of numbers)
- Basic probability (for average case analysis, randomised algorithms)

Exercise #7: Analysis of Algorithms

What is the complexity of the following algorithm?

\[
\text{splitList}(L): \quad \begin{align*}
\text{Input} & : \text{non-empty linked list } L \\
\text{Output} & : L \text{ split into two halves} \\
& \quad \begin{align*}
\text{// use slow and fast pointer to traverse } L \\
& \text{slow=head(L), fast=head(L).next} \\
& \text{while } \text{fast} \neq \text{NULL} \land \text{fast.next} = \text{NULL} \text{ do} \\
& \quad \text{slow=slow.next, fast=fast.next.next} \quad \text{// advance pointers} \\
& \text{end while} \\
& \text{cut } L \text{ between slow and slow.next}
\end{align*}
\]

Answer: \( O(|L|) \)

Exercise #8: Analysis of Algorithms

What is the complexity of the following algorithm?

\[
\text{binaryConversion}(n): \quad \begin{align*}
\text{Input} & : \text{positive integer } n \\
\text{Output} & : \text{binary representation of } n \text{ on a stack} \\
& \quad \begin{align*}
& \text{create empty stack } S \\
& \text{while } n > 0 \text{ do} \\
& \quad \text{push } (n \mod 2) \text{ onto } S \\
& \quad n = n/2 \\
& \text{end while} \\
& \text{return } S
\end{align*}
\]

Assume that creating a stack and pushing an element both are \( O(1) \) operations ("constant")

Answer: \( O(\log n) \)

Relatives of Big-Oh

\textit{big-Omega}

\( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that

\[
f(n) \geq c \cdot g(n) \quad \forall n \geq n_0
\]

\textit{big-Theta}

\( f(n) \) is \( \Theta(g(n)) \) if there are constants \( c' \cdot c'' > 0 \) and an integer constant \( n_0 \geq 1 \) such that

\[
c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \quad \forall n \geq n_0
\]
• $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$
• $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$
• $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

... Relatives of Big-Oh

Examples:

\[ \frac{1}{4}n^2 \text{ is } O(n^2) \]

need $c > 0$ and $n_0$ such that $\frac{1}{4}n^2 \leq cn^2$ for $n \geq n_0$

- let $c = \frac{1}{4}$ and $n_0 = 1$

\[ \frac{1}{4}n^2 \text{ is } \Omega(n) \]

need $c > 0$ and $n_0$ such that $\frac{1}{4}n^2 \geq cn$ for $n \geq n_0$

- let $c = 1$ and $n_0 = 2$

\[ \frac{1}{4}n^2 \text{ is } \Theta(n^2) \]

since $\frac{1}{4}n^2$ is in $\Omega(n^2)$ and $O(n^2)$

Complexity Classes

Problems in Computer Science …

• some have polynomial worst-case performance (e.g. $n^2$)
• some have exponential worst-case performance (e.g. $2^n$)

Classes of problems:

- $P$ = problems for which an algorithm can compute answer in polynomial time
- $NP$ = includes problems for which no $P$ algorithm is known

Beware: NP stands for "nondeterministic, polynomial time (on a theoretical Turing Machine)"

... Complexity Classes

Computer Science jargon for difficulty:

• tractable … have a polynomial-time algorithm (useful in practice)
• intractable … no tractable algorithm is known (feasible only for small $n$)
• non-computable … no algorithm can exist

Computational complexity theory deals with different degrees of intractability

Generate and Test Algorithms

Generate and Test

In scenarios where

• it is simple to test whether a given state is a solution
• it is easy to generate new states (preferably likely solutions)

then a generate and test strategy can be used.

It is necessary that states are generated systematically

• so that we are guaranteed to find a solution, or know that none exists
  • some randomised algorithms do not require this, however
    (more on this later in this course)

... Generate and Test

Simple example: checking whether an integer $n$ is prime

• generate/test all possible factors of $n$
• if none of them pass the test $\Rightarrow n$ is prime

Generation is straightforward:

• produce a sequence of all numbers from 2 to $n-1$

Testing is also straightforward:

• check whether next number divides $n$ exactly

... Generate and Test

Function for primality checking:

\[ \textit{isPrime(n)}: \]

<table>
<thead>
<tr>
<th>Input</th>
<th>natural number $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>true if $n$ prime, false otherwise</td>
</tr>
</tbody>
</table>

for all $i = 2 \ldots n-1$ do // generate
  if $n \mod i = 0$ then // test
    return false // $i$ is a divisor $\Rightarrow n$ is not prime
  end if
end for
return true // no divisor $\Rightarrow n$ is prime

Complexity of $\textit{isPrime}$ is $O(n)$

Can be optimised: check only numbers between $2$ and $\lfloor \sqrt{n} \rfloor \Rightarrow O(\sqrt{n})$

Example: Subset Sum

Problem to solve ...
Is there a subset \( S \) of these numbers with \( \text{sum}(S) = 1000 \)?

\[
\]

General problem:
- given \( n \) integers and a target sum \( k \)
  - is there a subset that adds up to exactly \( k \)?

**Example: Subset Sum**

Generate and test approach:

\[
\text{subsetsum}(A,k):
| \text{Input} \; \text{set} \; A \; \text{of} \; n \; \text{integers}, \; \text{target} \; \text{sum} \; k
| \text{Output} \; \text{true} \; \text{if} \; \Sigma_{b \in B} b = k \; \text{for some} \; B \in A
| \text{false} \; \text{otherwise}
| \text{for each} \; \text{subset} \; S \subseteq A \; \text{do}
| | \text{if} \; \text{sum}(S) = k \; \text{then}
| | | \text{return} \; \text{true}
| | \text{end if}
| \text{end for}
| \text{return} \; \text{false}
\]

- How many subsets are there of \( n \) elements?
- How could we generate them?

**Example: Subset Sum**

Given: a set of \( n \) distinct integers in an array \( A \) …

- produce all subsets of these integers

A method to generate subsets:
- represent sets as \( n \) bits (e.g. \( n=4 \), \( 0000, 0011, 1111 \) etc.)
- \( \text{bit} \; i \) represents the \( i \)-th input number
- if \( \text{bit} \; i \) is set to 1, then \( A[1] \) is in the subset
- if \( \text{bit} \; i \) is set to 0, then \( A[1] \) is not in the subset
- e.g. if \( A[] = \{1, 2, 3, 5\} \) then 0011 represents \( \{1, 2\} \)

**Example: Subset Sum**

Algorithm:

\[
\text{subsetsum1}(A,k):
| \text{Input} \; \text{set} \; A \; \text{of} \; n \; \text{integers}, \; \text{target} \; \text{sum} \; k
| \text{Output} \; \text{true} \; \text{if} \; \Sigma_{b \in B} b = k \; \text{for some} \; B \in A
| \text{false} \; \text{otherwise}
| \text{for} \; s=0..2^{n-1} \; \text{do}
| | \text{if} \; k = \Sigma_{(i\text{-th bit of } s \text{ is } 1)} A[i] \; \text{then}
| | | \text{return} \; \text{true}
| | \text{end if}
| \text{end for}
| \text{return} \; \text{false}
\]

Obviously, \( \text{subsetsum1} \) is \( O(2^n) \)

**Example: Subset Sum**

Alternative approach …

\[
\text{subsetsum2}(A,n,k)
| \text{Input} \; \text{array} \; A, \; \text{index} \; n, \; \text{target} \; \text{sum} \; k
| \text{Output} \; \text{true} \; \text{if some subset of} \; A[0..n-1] \; \text{sums up to} \; k
| \text{false} \; \text{otherwise}
| \text{if} \; k = 0 \; \text{then}
| | \text{return} \; \text{true} \; \// \; \text{empty set solves this}
| \text{else if} \; n = 0 \; \text{then}
| | \text{return} \; \text{false} \; \// \; \text{no elements} \Rightarrow \; \text{no sums}
| \text{else}
| | \text{return} \; \text{subsetsum}(A,n-1,k-A[n-1]) \; \lor \; \text{subsetsum}(A,n-1,k)
| \text{end if}
\]

Cost analysis:
- \( C_i = \# \text{calls to} \; \text{subsetsum2}() \; \text{for array of length} \; i \)
  - for best case, \( C_n = C_{n-1} \) (why?)
  - for worst case, \( C_n = 2C_{n-1} \Rightarrow C_n = 2^n \)

Thus, \( \text{subsetsum2} \) also is \( O(2^n) \)
Example: Subset Sum

Subset Sum is typical member of the class of \textit{NP-complete problems}

- intractable … only algorithms with exponential performance are known
  - increase input size by 1, double the execution time
  - increase input size by 100, it takes $2^{100} = 1,267,650,600,228,229,401,496,703,205,376$ times as long to execute
- but if you can find a polynomial algorithm for Subset Sum, then any other \textit{NP}-complete problem becomes \textit{P}!

Summary

- Big-Oh notation
- Asymptotic analysis of algorithms
- Examples of algorithms with logarithmic, linear, polynomial, exponential complexity

Suggested reading:
- Sedgewick, Ch.2.1-2.4,2.6

Produced: 14 Aug 2017