COMP2521 19T0
Week 2, Tuesday: Algorithms!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

algorithm analysis
complexity
recursion

Complexity

Problems, Algorithms, Programs, Processes

problem	something that needs to be solved
algorithm	well-defined instructions to solve the problem
program	implementation of the algorithm in a particular programming language
process	an instance of a program being executed

Analysis of Software

What makes software ‘good’?

correctness	returns expected result for all valid inputs
robustness	behaves ‘sensibly’ for non-valid inputs
efficiency	returns results reasonably quickly (even for large inputs)
clarity	clear code, easy to maintain/modify
consistency	interface is clear and consistent (API or GUI)

lecture 2: correctness.
today: efficiency.

Algorithm Efficiency

- algorithm runtime tends to be a function of input size
- often difficult to determine the average run time
- we tend to focus on asymptotic worst-case execution time
  ... easier to analyse!
  ... crucial to many applications: finance, robotics, games, ...

Algorithm Efficiency

By far, the *most important* determinant of a program's efficiency.

Small, often constant-factor speedups from

- operating systems,
- compilers,
- hardware,
- implementation details

More important: an efficient algorithm.

Algorithm Efficiency

Determining Theoretically and Empirically

Design

- complexity theory!

Implementation and Testing

- measure its properties!
  ...run-time using *time(1)*
  ...profiling tools like *gprof(1)*
  ...performance counters like *pmc(3)*, *hwpmc(4)*

Algorithm Efficiency, Empirically

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

Algorithm Efficiency, Empirically

1. Write a program that implements an algorithm.
   ... which may not always be possible!
2. Run the program with inputs of varying size and composition.
3. Measure the actual runtime.
4. Plot the results.

Algorithm Efficiency, Empirically

1. Write a program that implements an algorithm.
   ... which may not always be possible!
2. Run the program with inputs of varying size and composition.
   ... which may not always be possible!
   ... choosing good inputs is *extremely* important
3. Measure the actual runtime.
4. Plot the results.

Algorithm Efficiency, Empirically

1. Write a program that implements an algorithm.
   ... which may not always be possible!
2. Run the program with inputs of varying size and composition.
   ... which may not always be possible!
   ... choosing good inputs is *extremely* important
3. Measure the actual runtime.
   ... which may not always be possible (or easy)!
   ... similar runtime environments required
4. Plot the results.

Algorithm Efficiency, Empirically

1. Write a program that implements an algorithm.
   ... which may not always be possible!
2. Run the program with inputs of varying size and composition.
   ... which may not always be possible!
   ... choosing good inputs is *extremely* important
3. Measure the actual runtime.
   ... which may not always be possible (or easy)!
   ... similar runtime environments required
4. Plot the results.
   (Optionally, be confused about the results.)

Algorithm Efficiency, Theoretically

- Don’t necessarily use an implementation!
  … Use pseudocode or something close to it.
- Characterise efficiency as a function of inputs.
- Take into account *all possible* inputs
- Generally produces a value that is environment-agnostic
  … allowing us to evaluate comparative efficiency of algorithms

Timing Execution

Absolute times will differ
between machines, between languages
...so we're not interested in absolute time.

We are interested in the *relative* change
as the problem size increases

Timing Execution

time(1)

We can use the *time(1)* command to measure execution time (and several other interesting properties).

There are two common implementations:
one built-into the shell,
and one at `/usr/bin/time`
both are OK for our purposes.

Timing Execution

time(1) output

```
$ time ./prog

./prog 0.01s user 0.02s system 97% cpu 0.028 total
0k shared 0k local 11k max 0+3280 faults
13+0 in 0+0+0 out 4 vcsw 4 ivcsw
```

Most of this information isn’t interesting to us.

Timing Execution

time(1) output

```
$ time ./prog

./prog 0.01s user 0.02s system 97% cpu 0.028 total
0k shared 0k local 11k max 0+3280 faults
13+0 in 0+0+0 out 4 vcsw 4 ivcsw
```

Most of this information isn't interesting to us.
The *user* time is!

Timing Execution

time(1) output

```
$ time ./prog

./prog 0.01s user 0.02s system 97% cpu 0.028 total
0k shared 0k local 11k max 0+3280 faults
13+0 in 0+0+0 out 4 vcsw 4 ivcsw
```

Most of this information isn't interesting to us.
The *user* time is!

Redirect input into your program:

```
$ time ./prog < input > /dev/null
$ ./mkinput | time ./prog > /dev/null
```

Empirical Analysis

Analysis of a Linear Search

Time a linear search with different-sized inputs —

```
$ ./gen 100  A | time ./linear > /dev/null
$ ./gen 1000 A | time ./linear > /dev/null
```

(repeat a number of times and average)
What is the relation between *input size* and *user time*?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run …

- how long for 2000?
- how long for 10,000?
- how long for 100,000?
- how long for 1,000,000?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run ...

- how long for 2000? 4.8 seconds
- how long for 10,000?
- how long for 100,000?
- how long for 1,000,000?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run …

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000?
- how long for 1,000,000?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run …

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000? 12000 seconds (3.3 hours)
- how long for 1,000,000?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run …

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000? 12000 seconds (3.3 hours)
- how long for 1,000,000? 1200000 seconds (13.9 days)

Searching in a Sorted Array

Given an array a of n elements,
where for any pair of indices i, j,
$i \leq j < n$ implies $a[i] \leq a[j]$
search for an element e in the array.

```
int a[N];        // array with N items
bool found = 0;

int i = 0;
while ((i < N) && (! found)) {
    found = (a[i] == e);

    i++;
}
```

Searching in a Sorted Array

Given an array a of n elements,
where for any pair of indices i, j,
$i \leq j < n$ implies $a[i] \leq a[j]$
search for an element e in the array.

```
int a[N];        // array with N items
bool found = 0;
bool finished = false;
int i = 0;
while ((i < N) && (! found)   && (! finished)) {
    found = (a[i] == e);
    finished = (e < a[i]);
    i++;
}
```

Searching in a Sorted Array

Complexity Analysis

How many comparisons do we need
for an array of size N?

Best case: $t(N) \sim O(1)$

Worst case: $t(N) \sim O(N)$

Average case: $t(N) \sim O(N/2)\ O(N)$

Still a *linear* algorithm!
Can we do better?

Searching in a Sorted Array

Exploiting a Binary Search

Let's start in the middle.

- If e==$a[N/2]$, we found e; we're done!
- Otherwise, we split the array:
  ... if $e < a[N/2]$, we search the left half ($a[0]$ to $a[(N/2) - 1)]$)
  ... if $e > a[N/2]$, we search the right half ($a[(N/2) + 1)]$ to $a[N - 1]$)

Why Binary Search?

How many comparisons do we need
for an array of size N?

Best case: $t(n) \sim O(1)$

Why Binary Search?

How many comparisons do we need for an array of size N?

Best case: $t(n) \sim O(1)$

Worst case:

$$t(N) = 1 + t(\frac{N}{2})$$
$$t(N) = \log_2 N + 1$$
$$t(N) \sim O\left(\log N\right)$$

Algorithm Efficiency, Theoretically

Cost Modelling of Primitive Operations

In C, a line of code can do *lots* of things!

We're interested in 'primitive operations', though:
operations that can execute in one step,
which we can think of as hardware instructions.

(In COMP1521, we use the MIPS instruction set;
we get a feel for the primitive nature of instructions.)

Our cost-modelling will roughly follow the same lines,
but strictly we don't need to consider how long a primop takes.
We'll see why in a moment.

Expressing Complexity Classes

We express complexity using a range of
complexity models and *complexity classes*.

Most commonly, time complexity,
for which we use Big-O notation,
representing asymptotic worst-case time complexity.
I'll sometimes call this WCET.

Sometimes, space complexity too.
(Not so much in this course, but useful!)

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

- When does the worst case occur?
- How many data comparisons were made?
- What is the worst-case cost?

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

- When does the worst case occur? ... key $\notin$ a
- How many data comparisons were made?
- What is the worst-case cost?

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

- When does the worst case occur? ... key $\notin$ a
- How many data comparisons were made? ... n
- What is the worst-case cost?

Worst-Case Execption Time

```c
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

$1 + (n + 1) + n(1+$
$\quad 2 + ($
$\qquad 0)$
$) + 1$

- When does the worst case occur? ... key $\notin$ a
- How many data comparisons were made? ... n
- What is the worst-case cost?

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

$1 + (n + 1) + n(1+$
$2 + ($
$0)$
$) + 1$

- When does the worst case occur? ... key $\notin$ a
- How many data comparisons were made? ... n
- What is the worst-case cost? ... $3 + 4n$

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

$$1 + (n+1) + n(1+ \\ 2 + (\\ 0) \\ ) + 1$$

- When does the worst case occur? ... key $\notin$ a
- How many data comparisons were made? ... n
- What is the worst-case cost? ... $3 + 4n$ ... $O(n)$

Complexity Theory

Big-O Notation

Growth rate is not affected (much, usually)
by constant factors or lower-order terms ...
so we discard them.

$3 + 4n$ becomes $O(n)$ — a *linear* function
$3 + 4n + 3n^2$ becomes $O(n^2)$ — a *quadratic* function

These are an *intrinsic property* of the algorithm.

Complexity Theory

Discarding Less Obvious Terms

If a is time taken by the fastest primitive operation,
and b is time taken by the slowest primitive operation,
and $t(n)$ is the WCET of our algorithm ...

$$a \cdot (3 + 4n) \leq t(n) \leq b \cdot (3 + 4n)$$

Complexity Theory

Discarding Less Obvious Terms

If a is time taken by the fastest primitive operation,
and b is time taken by the slowest primitive operation,
and $t(n)$ is the WCET of our algorithm ...

$$a \cdot (3 + 4n) \leq t(n) \leq b \cdot (3 + 4n)$$

Where does the log-base go?

$O(\log_2 n) \equiv O(\log_3 n) \equiv \cdots$

(since $\log_b(a) \times \log_a(n) = \log_b(n)$)

Complexity Classes

All The Mathematics!

$f(n)$ 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 $g(n)$

Given $f(n)$ and $g(n)$, we say $f(n)$ is $O(g(n))$
if we have positive constants c and n_0 such that

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

Complexity Theory

Some Common Big-O Functions

constant $O(1)$ ...constant-time execution, independent of the input size.

logarithmic $O(\log n)$ ...some divide-and-conquer algorithms with trivial split/recombine operations

linear $O(n)$ ...every element of the input has to be processed (in a straightforward way)

n-log-n $O(n \log n)$ ...divide-and-conquer algorithms, where split/recombine is proportional to input

quadratic $O(n^2)$ ...compute every input with every other input ...problematic for large inputs!

cubic $O(n^3)$ ... misery

factorial $O(n!)$ ... real misery

exponential $O(2^n)$ ... running forever is fine, right?

Complexity Theory

Complexity Classes

tractable have a polynomial-time (‘P’) algorithm
... polynomial worst-case performance (e.g., $O(n^2)$)
... (useful and usable in practical applications)

intractable no tractable algorithm exists (usually ‘NP’[1])
... worse than polynomial performance (e.g., $O(2^n)$)
... (feasible only for small n)

non-computable no algorithm exists (or can exist)

[1] nondeterministic polynomial time, on a theoretical Turing Machine

Thinking about Complexity

What would be the time complexity of
inserting an element at the beginning of

... a linked list?
... an array?

Thinking about Complexity

What would be the time complexity of inserting an element at the beginning of

... a linked list?
... an array?

What about the end?

Thinking about Complexity

What would be the time complexity of
inserting an element at the beginning of

... a linked list?
... an array?

What about the end?

What if it's ordered?

Recursion

Recursive Functions

Sometimes, problems can be expressed in terms of a simpler instance of the same problem.

- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$
- $\vdots$
- $(n-1)! = (n-1) \times \cdots \times 3 \times 2 \times 1$
- $(n)! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$

Recursive Functions

Sometimes, problems can be expressed in terms of a simpler instance of the same problem.

- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$
- $\vdots$
- $(n-1)! = (n-1) \times \cdots \times 3 \times 2 \times 1$
- $(n)! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$

$$n! = (n-1)! \times n$$

Recursive Functions

Solving problems recursively in a program involves developing a program that calls itself.

base case
: (or *stopping case*)
no recursive call is needed

recursive case
: calls the function on a smaller version of the problem

Iteration vs Recursion

```
int factorial (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

```
int factorial (int n) {
    if (n == 1) return 1;
    else return n * factorial (n - 1);
}
```

Too Many Damn Rabbits

Recursive code can be horribly inefficient!
2^n calls is $O(k^n)$ time — exponential!

```
switch (n) {
case 0:  return 0;
case 1:  return 1;
default: return fib (n - 1) + fib (n - 2);
}
```