Algorithm Analysis

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

Input: x; $e = (d_{\ell-1}d_{\ell-2} \cdots d_1 d_0)_{\text{NAF}}$
Returns: x^e

begin
  $S \leftarrow x$; $R_1 \leftarrow 1$; $R_{\overline{1}} \leftarrow 1$;
  for each digit d_i (i from 0 up to $\ell - 1$) **do**
    if $d_i \neq 0$ **then**
      $R_{d_i} \leftarrow R_{d_i} \times S$;
    end
    $S \leftarrow S^2$;
  end
  return $R_1 \times (R_{\overline{1}})^{-1}$;
end

$\Theta(n)$

$\Omega(\log n)$

$\Theta(1)$

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Algorithm Analysis

Standard reminder to set phones to silent/vibrate mode, please!

Algorithm Analysis

- Previously, on ECE-250...
  - We looked at:
    - Asymptotic Analysis
    - Asymptotic notation
    - Tried to figure out run times (in asymptotic notation) for some simple algorithms from their description

Algorithm Analysis

- In today's class:
  - We'll look in more detail at the analysis of algorithms.
  - In particular, given an algorithm, analyze it to determine its run time and space requirements
    - (In today's examples, we'll put some more emphasis on runtimes — when dealing with data structures and operations on them, we'll probably put the emphasis on space requirements)
  - We'll look into some of the common constructs in C++ (should be useful for the general case)

Algorithm Analysis

- Analyzing algorithms typically involves:
  - Counting instances of execution for the various sections:
    - A program may be three or four lines, yet those could translate into several thousand operations (e.g., loops)
    - (BTW, what else, other than loops?)
  - Figuring out the run time for each of the simpler parts.
  - Compute totals and figure out the appropriate class (as in, which Landau symbol and what function)

Algorithm Analysis

- We already mentioned the issue of counting operations when the number of times something executes depends on the data (and thus, may vary from execution to execution)
- We also mentioned that if we're restricted to one figure to describe the algorithm, then we use the worst-case analysis.
  - In some (perhaps most) cases, it is also a good idea to include average-case analysis as well.

Algorithm Analysis

- We'll look at:
  - Operators and expressions
  - Control statements
    - Conditional statements and loops
  - Functions
  - Recursive functions (if enough time)

Algorithm Analysis

- Operators, as a general rule, map to one or a few (as in, a fixed number of) assembly-level instructions.
- For example, the statement `a += b;` could map to something like:

```
addl  %eax, -4(%rbp)
```

Algorithm Analysis

- This automatically tells us that these operators' run time is Θ(1)
- This includes:
  - Reading variables / assigning variables (`=`)
  - Arithmetic operations (`+`, `-`, `*`, `/`, `%`, `++`, `--`)
  - Logical and bitwise operations (and/or/not/xor)
  - Comparisons (`==`, `!=`, `<`, `<=`, `>`, `>=`)
  - Subscripting (`[ ]`)
  - Memory allocation and deallocation (`new`, `delete`)

Algorithm Analysis

- However, there may be exceptions: with strings, comparison operators can take linear time (if measured with respect to the length of the string), and so does operator + (for strings, this means concatenation).
- And for that matter, with operators used on user-defined types, operators translate into calling a user-provided function, so all bets are off! (the function could execute in linear time, or in any case, in $\omega(1)$)

Algorithm Analysis

- Notice that, even though memory allocation and deallocation (with new/delete) can be slower than the other operations by a factor of possibly several hundred, they are still Θ(1) — not really a constant amount of time, but upper-bounded by a constant, and thus *constant time* in the sense of being Θ(1).

Algorithm Analysis

- Expressions (e.g., arithmetic or logical expressions) are nothing more than an arrangement of a fixed number of operators; thus, they are also Θ(1).
  - Careful — if the expression involves calling functions, then we have to first figure out (or find out — presumably through its documentation) the runtime of the function: it may not be Θ(1) !

Algorithm Analysis

- We could take it a little further and think of "higher-level" operations such as swapping two variables, and consider them (when possible) a single operation running in Θ(1):

```
int tmp = a;
a = b;
b = tmp;
```

- Three operations, Θ(1) each — thus, as a whole, swapping two integers takes Θ(1).

Algorithm Analysis

- Another example: inserting an element (say, a given **`value`**) to a linked list:

```
Node * new_elem = new Node (value);
new_elem->d_next = cur->next();
cur->d_next = new_elem;
```

Algorithm Analysis

- Another example: inserting an element (say, a given **`value`**) to a linked list:

```
Node * new_elem = new Node (value);
new_elem->d_next = cur->next();
cur->d_next = new_elem;
```

- Shall we work on the board a graphical representation to make sure the above code is correct?

Algorithm Analysis

- Even if it is a doubly-linked list:

```
Node * new_elem = new Node (value);
new_elem->d_next = cur->next();
new_elem->d_prev = cur;
cur->d_next = new_elem;
cur->next()->d_prev = new_elem;
```

Algorithm Analysis

- Even if it is a doubly-linked list:

```
Node * new_elem = new Node (value);
new_elem->d_next = cur->next();
new_elem->d_prev = cur;
cur->d_next = new_elem;
cur->next()->d_prev = new_elem;
```

- Ok, so there is this severe bug, as we just saw on the board, and that hopefully you will fix; but that does not change the example — it is a fixed number of operations, and thus Θ(1).

Algorithm Analysis

- Next we'll look at these flow-control statements:
  - Conditionals (if, if – else, switch)
  - Loops (for, while, do while)

Algorithm Analysis

- In both cases, there are two aspects to consider:
  - The time that it takes to determine what to do (e.g., whether or not to execute a block, whether or not to continue to the next pass of the loop, etc.).
  - How does the outcome of the decision affects the operations count.

Algorithm Analysis

- In both cases, there are two aspects to consider:
  - The time that it takes to determine what to do (e.g., whether or not to execute a block, whether or not to continue to the next pass of the loop, etc.).
  - How does the outcome of the decision affects the operations count.
- Notice that in nested control statements, we must work starting from the inner-most and working our way out.

Algorithm Analysis

- To determine what to do, a condition has to be evaluated — but a condition is nothing more than an expression (one that evaluates to a boolean value), and thus, it takes constant time
  - Again, careful — if, as part of the condition, we call a function, then further analysis is necessary.

Algorithm Analysis

- Once we determine this, then we count operations accordingly — for a conditional instruction (e.g., an if statement), we just add or don't add the runtime of the block associated to the if.

Algorithm Analysis

- Possibly trick question — what is the run time of the following "conditionally swap"?

```
if (a > b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Algorithm Analysis

- Switch statements are actually a very efficient form of of a bunch of if – else if in a row. Thus, not surprisingly, the decision part of a switch statement runs in Θ(1).
- The requirement is that the cases be compile-time integral constants — thus, the compiler implements a so-called *jump table*, translating the whole thing into possibly a single assembly-level instruction! (some times a few of them)

Algorithm Analysis

- Of course, that's for the decision part only; if the instructions in the selected case statement runs in ω(1), then we would take that into account as required.

Algorithm Analysis

- Example (perhaps trick question?):

  What is the run time of the following?

```
switch (user_selection)
{
    case 'a':
        // something Θ(1), then break;
    case 'b':
        // something Θ(log n), then break;
    case 'c':
        // something Θ(n), then break;
    default:
        // Print error message, presumably Θ(1)
}
```

Algorithm Analysis

- Perhaps an even trickier question — what is the average-case run time of that fragment? In particular, how do the probabilities of each selection affect the average-case run time?

Algorithm Analysis

- With loops (both for and while loops), there is inititlization, per-pass condition check and update of the control variable.
- These usually take Θ(1) — thus, the important part to figure out is how many times it will run, and multiply the run time of the body of the loop times the number of times it runs.

Algorithm Analysis

- For example, the following loop:

```
for (int i = 0; i < n; i++)
{
    // code that executes in Θ(f(n))
}
```

- Runs in $\Theta(n\ f(n))$ — $\Theta(1) + n \times (\Theta(1) + \Theta(f(n)))$
- Why each of the terms above?
- Why is $\Theta(1) + \Theta(f(n)) = \Theta(f(n))$??

Algorithm Analysis

- Hmm... Problem: how do we know how many times something like this will run?

```
int total = 0;
for (int i = 0;
          i < size && values[i] >= 0;
          i++)
{
    total += values[i];
}
```

Algorithm Analysis

- We keep in mind the distinction between worst-case analysis and average-case analysis.
- So, what would be the worst-case runtime of the previous fragment that adds all values in an array of *n* elements, stopping at the first negative value?

Algorithm Analysis

- We keep in mind the distinction between worst-case analysis and average-case analysis.
- So, what would be the worst-case runtime of the previous fragment that adds all values in an array of *n* elements, stopping at the first negative value?
- How about a tough one — what would be the average-case run time?

Algorithm Analysis

- Another example: what is the run time of the loop below?

```
for (int i = 1; i < n; i *= 2)
{
    // Something that is Θ(1)
}
```

Algorithm Analysis

- Now for nested loops: what is the run time of the fragment below?

```
int sum = 0;
for (int i = 1; i < n; i++)
{
    for (int k = 0; k < i; k++)
    {
        sum += i+k;
    }
}
```

Algorithm Analysis

- What about functions?

Algorithm Analysis

- What about functions?
- For the most part, functions are quite easy to handle: calling a function involves setting up a few things, adjusting the CPU stack, a few registers, saving the current values of the registers, etc.

Algorithm Analysis

- What about functions?
- For the most part, functions are quite easy to handle: calling a function involves setting up a few things, adjusting the CPU stack, a few registers, saving the current values of the registers, etc.
- Not only these are a fixed number of instructions, and thus $\Theta(1)$ — on most modern CPUs, this is done with a single assembly-level instruction! (because the task is so common!)

Algorithm Analysis

- Notice that inline functions may change the rules, reducing this time to 0 (and for that matter, the optimizing stage of the compiler could radically change the situation).
- (if you're unfamiliar with this notion, look up — not now; after class, of course! — “C++ inline functions” to find out)
- However, since we're often doing worst-case analysis, it's ok to assume no inlining and no aggressive optimizations.

Algorithm Analysis

- Bottom line:

  With functions, the analysis typically boils down to either figuring out, or finding out through the function's documentation, what the run time of the function is, and add it as you would add the run time of any fragment of code in its place.

Algorithm Analysis

- Figuring out the run time of a function is not unlike figuring out the run time of any other block of code — after all, a function is a block of code (properly identified and “packaged” so that it can be called from multiple places).

Algorithm Analysis

- Figuring out the run time of a function is not unlike figuring out the run time of any other block of code — after all, a function is a block of code (properly identified and “packaged” so that it can be called from multiple places).
- BTW, all of this applies as well to member functions in a class!

Algorithm Analysis

- The only caveat is when a function calls itself (i.e., a *recursive* function)

- Not too bad, though — the situation usually boils down to a recurrence relation; the run time of the function that we're trying to figure out includes the recursive call to itself, typically for a problem with lower size (otherwise the recursion would never end, right?)

Algorithm Analysis

- An example: let's do a (somewhat silly) recursive version of finding the highest value in an array — we split the array in two halves, recursively call the function to find the highest in each of the two halves, and return the higher of the two returned values.
- The base case being an array of 1 element, in which case the highest value is *the* (only) value.

Algorithm Analysis

- Something like this (for simplicity, assume n is a power of 2):

```
int find_max (const int * array, int n)
{
    if (n == 1)
    {
        return array[0];
    }

    return std::max (find_max (array, n/2),
                     find_max (array + n/2, n/2));
}
```

Algorithm Analysis

- So, the run time is the sum of three parts:
  - The “setup” to call the function and pass the parameters (we know this is Θ(1)).
  - The if block (which we now know that is Θ(1))
  - Two times the run time of the function for half the size (well, plus a Θ(1) for std::max to determine the higher of the two values)

Algorithm Analysis

- So, the run time is the sum of three parts:
  - The “setup” to call the function and pass the parameters (we know this is Θ(1)).
  - The if block (which we now know that is Θ(1))
  - Two times the run time of the function for half the size (well, plus a Θ(1) for std::max to determine the higher of the two values)
- And what do we think the above is? (you should see by now where this is going — feel like writing it up and solving it?)