Chapter Twelve

A Review of Complexity

Kinds Of Complexity

- ✓ Worst-case complexity.
- ✓ Average complexity.
- Amortized complexity.

Task Sequence

- Suppose that a sequence of n tasks is performed.
- The worst-case cost of a task is c_{wc}.
- Let c_i be the (actual) cost of the i^{th} task in this sequence.
- So, $c_i <= c_{wc}$, $1 <= i <= n$.
- $n * c_{wc}$ is an upper bound on the cost of the sequence.
- $j * c_{wc}$ is an upper bound on the cost of the first j tasks.

Task Sequence

- Let c_{avg} be the average cost of a task in this sequence.
- So, $c_{avg} = \Sigma c_i / n$.
- $n * c_{avg}$ is the cost of the sequence.
- $j * c_{avg}$ is not an upper bound on the cost of the first j tasks.
- Usually, determining c_{avg} is quite hard.

Task Sequence

- At times, a better upper bound than $j * c_{wc}$ or $n * c_{wc}$ on sequence cost is obtained using amortized complexity.

Amortized Complexity

- The amortized complexity of a task is the amount you charge the task.
- The conventional way to bound the cost of doing a task n times is to use one of the expressions
  - n*(worst-case cost of task)
  - Σ(worst-case cost of task i)
- The amortized complexity way to bound the cost of doing a task n times is to use one of the expressions
  - n*(amortized cost of task)
  - Σ(amortized cost of task i)

Amortized Complexity

- The amortized complexity/cost of individual tasks in any task sequence must satisfy:

  Σ(actual cost of task i)

  <= Σ(amortized cost of task i)

- So, we can use

  Σ(amortized cost of task i)

  as a bound on the actual complexity of the task sequence.

Amortized Complexity

- The amortized complexity of a task *may bear no direct relationship* to the actual complexity of the task.

Amortized Complexity

- In conventional complexity analysis, each task is charged an amount that is >= its cost.

  Σ(actual cost of task i)

  <= Σ(worst-case cost of task i)

- In amortized analysis, some tasks may be charged an amount that is < their cost.

  Σ(actual cost of task i)

  <= Σ(amortized cost of task i)

Arithmetic Statements

- Rewrite an arithmetic statement as a sequence of statements without using parentheses.
- a = x+((a+b)*c+d)+y;

  is equivalent to the sequence:

  z1 = a+b;

  z2 = z1*c+d;

  a = x+z2+y;

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- The rewriting is done using a stack and a method processNextSymbol.
- create an empty stack;

  for (int i = 1; i <= n; i++)

  // n is number of symbols in statement

  processNextSymbol();

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- processNextSymbol extracts the next symbol from the input statement.
- Symbols other than) and ; are simply pushed on to the stack.

b
+
a
(
(
+
x
=
a

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.

z1 = a+b;

~~b~~
~~+~~
~~a~~
~~(~~
(
+
x
=
a

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.

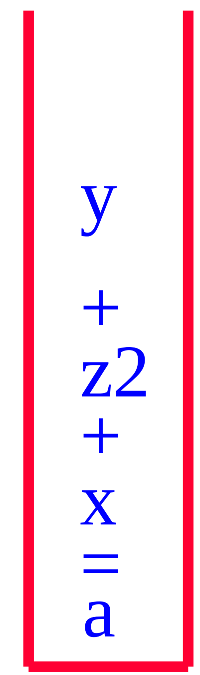
z1 = a+b;
z2 = z1*c+d;

~~d~~
~~+~~
~~c~~
~~*~~
~~z1~~
~~(~~
+
x
=
a

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.



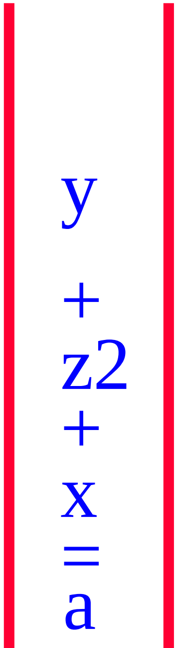
z1 = a+b;
z2 = z1*c+d;

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is ;, symbols are popped from the stack until the stack becomes empty. The final assignment statement
  a = x+z2+y;
  is generated.

z1 = a+b;
z2 = z1*c+d;



Complexity Of processNextSymbol

a = x+((a+b)*c+d)+y;

- O(number of symbols that get popped from stack)
- O(i), where i is for loop index.

Overall Complexity (Conventional Analysis)

```
create an empty stack;
for (int i = 1; i <= n; i++)
    // n is number of symbols in statement
    processNextSymbol();
```

- So, overall complexity is $O(\Sigma i) = O(n^2)$.
- Alternatively, $O(n*n) = O(n^2)$.
- Although correct, a more careful analysis permits us to conclude that the complexity is $O(n)$.

Ways To Determine Amortized Complexity

- Aggregate method.
- Accounting method.
- Potential function method.

Aggregate Method

- Somehow obtain a good upper bound on the actual cost of the n invocations of processNextSymbol()
- Divide this bound by n to get the amortized cost of one invocation of processNextSymbol()
- Easy to see that

$$\Sigma(\text{actual cost}) <= \Sigma(\text{amortized cost})$$

Aggregate Method

- The actual cost of the n invocations of processNextSymbol()

  equals number of stack pop and push operations.
- The n invocations cause at most n symbols to be pushed on to the stack.
- This count includes the symbols for new variables, because each new variable is the result of a) being processed. Note that no)s get pushed on to the stack.

Aggregate Method

- The actual cost of the n invocations of processNextSymbol() is at most $2n$.
- So, using $2n/n = 2$ as the amortized cost of processNextSymbol() is OK, because this cost results in $\Sigma(\text{actual cost}) <= \Sigma(\text{amortized cost})$
- Since the amortized cost of processNextSymbol() is 2, the actual cost of all n invocations is at most $2n$.

Aggregate Method

- The aggregate method isn't very useful, because to figure out the amortized cost we must first obtain a good bound on the aggregate cost of a sequence of invocations.
- Since our objective was to use amortized complexity to get a better bound on the cost of a sequence of invocations, if we can obtain this better bound through other techniques, we can omit dividing the bound by n to obtain the amortized cost.

(**An example, not covered**)

- $P(i) = amortizedCost(i) – actualCost(i) + P(i – 1)$
- $\Sigma(P(i) – P(i–1)) =$

  $\Sigma(amortizedCost(i) –actualCost(i))$
- $P(n) – P(0) = \Sigma(amortizedCost(i) –actualCost(i))$
- $P(n) – P(0) >= 0$
- When $P(0) = 0$, $P(i)$ is the amount by which the first i tasks/operations have been over charged.

Potential Function Example

a = x + ((a + b) * c + d) + y ;

actual cost 1 1 1 1 1 11 1 1 5 1 1 1 1 7 1 1 7

amortized cost 2 2 2 2 2 22 2 2 2 2 2 2 2 2 2 2 2

potential 1 2 3 4 5 67 8 9 6 7 8 9 10 5 6 7 2

Potential = stack size except at end.

Accounting Method

- Guess the amortized cost.
- Show that $P(n) - P(0) >= 0$.

Accounting Method Example

```
create an empty stack;
 for (int i = 1; i <= n; i++)
     // n is number of symbols in statement
     processNextSymbol();
```

- Guess that amortized complexity of processNextSymbol is 2.
- Start with P(0) = 0.
- Can show that P(i) >= number of elements on stack after ith symbol is processed.

Accounting Method Example

	a = x + ((a + b) * c + d) + y ;
actual cost	1 1 1 1 1 11 1 1 5 1 1 1 1 7 1 1 7
amortized cost	2 2 2 2 2 22 2 2 2 2 2 2 2 2 2 2 2
potential	1 2 3 4 5 67 8 9 6 7 8 9 10 5 6 7 2

- Potential >= number of symbols on stack.
- Therefore, P(i) >= 0 for all i.
- In particular, P(n) >= 0.

Potential Method

- Guess a suitable potential function for which $P(n) - P(0) >= 0$ for all n.
- Derive amortized cost of ith operation using $\Delta P = P(i) - P(i-1)$

  $= \text{amortized cost} - \text{actual cost}$
- amortized cost = actual cost + ΔP

Potential Method Example

```
create an empty stack;
 for (int i = 1; i <= n; i++)
     // n is number of symbols in statement
     processNextSymbol();
```

- Guess that the potential function is $P(i) =$ number of elements on stack after i^{th} symbol is processed (exception is $P(n) = 2$).
- $P(0) = 0$ and $P(i) - P(0) >= 0$ for all i.

i^{th} Symbol Is Not) or ;

- Actual cost of processNextSymbol is 1.
- Number of elements on stack increases by 1.
- $\Delta P = P(i) - P(i-1) = 1$.
- amortized cost = actual cost + ΔP

$$= 1 + 1 = 2$$

i^{th} Symbol Is)

- Actual cost of processNextSymbol is #unstacked + 1.
- Number of elements on stack decreases by #unstacked –1.
- $\Delta P = P(i) - P(i-1) = 1 - \#unstacked$.
- amortized cost = actual cost + ΔP

$$= \#unstacked + 1 + (1 - \#unstacked)$$

$$= 2$$

i^{th} Symbol Is ;

- Actual cost of processNextSymbol is #unstacked = P(n–1).
- Number of elements on stack decreases by P(n–1).
- $\Delta P = P(n) - P(n-1) = 2 - P(n-1)$.
- amortized cost = actual cost + ΔP

$$= P(n-1) + (2 - P(n-1))$$

$$= 2$$