Data Structures and Algorithms
Lecture slides: Analyzing loops

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

Topics cover today

Introduction

Average-case analysis

Amortized/aggregate analysis

Disjoint sets

Review on summations

Motivation

In the introduction section we have seen a few examples of iterative algorithms

We also analyzed the running time of these algorithms for the worst and best cases

In the present section we show how to derive an average analysis of iterative algorithms

We also introduce “amortized analysis” which is an approach that provides more accurate asymptotic running time in some special algorithm cases

Finally we introduce a last data structure, “disjoint set”, which describes how sets can be represented on computers

Possible problems with a worst-case analysis

- The worst-case inputs may not be representative of all inputs ; for example, inputs seen in real-world situations may all be "easy" cases.
- A worst-case analysis is a very useful tool UNLESS you have special knowledge about the distribution of inputs in the real world that tells you that worst-case inputs never (or very rarely) occur.
- A worst-case analysis may be too difficult (although usually, a worst-case analysis is fairly easy).

Average-Case Analysis

Worst-case analysis seeks to find that input of size n that has the largest running time. In best case, we need to identify the input which has the lowest running time.

Unfortunately, in average-case analysis, we need to find the running time of each input of size n !

Then a probability is assigned to each input representing the likelihood that input will occur while running the algo

Average-case running time ($T_{avg}(n)$) of an algorithm :

$$T_{avg}(n) = \sum_{\text{i input of size } n} (p_i \times \{\# \text{ of elementary ops performed for input } i\}),$$

where p_i is the probability that input i will occur out of all inputs of size n.

Problem :

What if you don't know the probabilities of each input of size n?

Solution : Make assumptions. Most common assumption : all inputs of size n are equally likely to occur.

If all inputs of size n are equally likely to occur, then each input i of size n occurs with probability $1/(\text{number of inputs of size } n)$ and

$$T_{avg}(n) = \frac{\sum_{\text{i input of size } n}\{\#\text{ of elementary ops performed for input } i\}}{\#\text{ of inputs of size } n}.$$

Example Average-Case Analysis

```
LinearSearch ($L[1..n]$, $x$)
Foundx = false ; $i = 1$ ;
while ($i \leq n$ & Foundx = false) do
        if $L[i] = x$ then
           Foundx = true ;
        else
           $i = i + 1$ ;
if not Foundx then $i = 0$ ;
return $i$ ;
```

What is the average number of elementary operations performed by LinearSearch on input arrays of size n ?
(elementary operation $=$ a comparison between an array element and the item we are searching for.)

What are all the possible inputs of size n ?

```
**LinearSearch** ($L[1..n]$, $x$)
Foundx = false ; $i = 1$ ;
while ($i \leq n$ & Foundx = false) do
        if $L[i] = x$ then
            Foundx = true ;
        else
            $i = i + 1$ ;
if not Foundx then $i = 0$ ;
return $i$ ;
```

Each input of size n falls into one of the following $n + 1$ categories :

Category	description
1	x will be found in $L[1]$
2	x will be found in $L[2]$
3	x will be found in $L[3]$
$\vdots$	$\vdots$
n	x will be found in $L[n]$
$n + 1$	x will not be found in L

What is the running time of each input ?

For that we need to identify a basic operation in the pseudo-code and count how many time this instruction is executed for each different input of size n

One acceptable candidate for the basic operation is the **if** condition : “**if** $L[i] = x$ **then** ”

```
LinearSearch ($L[1..n]$, $x$)
Foundx = false ; $i = 1$ ;
while ($i \leq n$ & Foundx = false) do
        if $L[i] = x$ then
           Foundx = true ;
        else
           $i = i + 1$ ;
if not Foundx then $i = 0$ ;
return $i$ ;
```

What is the running time of each input ?

```
**LinearSearch** ($L[1..n]$, $x$)
Foundx = false ; $i = 1$ ;
**while** ($i \leq n$ & Foundx = false) **do**
        **if** $L[i] = x$ **then**
            Foundx = true ;
        **else**
            $i = i + 1$ ;
**if** not Foundx **then** $i = 0$ ;
**return** $i$ ;
```

Category	description	# of basic Ops
1	x will be found in $L[1]$	1
2	x will be found in $L[2]$	2
3	x will be found in $L[3]$	3
$\vdots$	$\vdots$	$\vdots$
n	x will be found in $L[n]$	n
$n+1$	x will not be found in L	n

Assumptions about the probability distribution :

- For every array L, there is an input in each of Categories $1..n$ (assuming the entries in L are distinct), and lots in Category $(n+1)$.
- Therefore, the probability that an input falls in Category i is the same as the probability that an input falls in Category j, where i and j are $\leq n$.
- The probability that x will be found versus the probability that x will not be found : unknown.
- Assume that x will be found with probability p.
- Therefore, x will be found in $L[i]$ with probability p/n, and x will not be found with probability $1-p$.

The average-case formulation for linear search

Category	description	Probability	$\#$ of basic Ops
1	x will be found in $L[1]$	p/n	1
2	x will be found in $L[2]$	p/n	2
3	x will be found in $L[3]$	p/n	3
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	x will be found in $L[n]$	p/n	n
$n+1$	x will not be found in L	$1-p$	n

Computing the average running time of linear search

Average-case running time $T_{avg}(n)$ of LinearSearch on inputs of size n :

$$\begin{aligned} T_{avg}(n) &= \sum_{\text{i input of size } n} (p_i \times \{\# \text{ of elementary ops done for input } I\}) \\ &= \left(\sum_{i=1}^{n} \left(\frac{p}{n} \times i \right) \right) + (1-p)n \end{aligned}$$

The summation is $(p/n)(1+2+\cdots+n)$, which is $(p/n) \times n(n+1)/2 = p(n+1)/2$.

$T_{avg}(n) = p\frac{n+1}{2} + (1-p)n$ basic operations.

- If x is always found ($p = 1$), then approximately half the entries of L are compared to x.
- If x is never found ($p = 0$), then x is compared to all entries of L.

Amortized analysis

Often, iterative algorithms have one particularly costly operation, but it doesn't get performed very often

That algorithm shouldn't be labeled a costly just because that one operation, that is seldom performed, is costly

”Amortized analysis is used to average out the costly operations in the worst case.”

Amortized analysis

Assume you have the following algorithm :

```
int a = 0;
for ($i = 0; i < n; i++$)
  for ($j = 0; j < n; i++$)
    if ($i \neq 0$) then
      $a = a - 1$;
      $j = n$;
    else $a = a + 1$;
```

Based on iterative algorithm analysis methods, since there is an inner **for** loop which runs in $O(n)$ iterations in worst case embedded into an outer loop that also runs for n iterations we conclude that the time complexity of this algorithm is $O(n^2)$

While $O(n^2)$ is correct, it is not the tightest bound we can get given the worst case of the inner **for** loop occurs for only one iteration of the outer loop

Amortized analysis

For the previous example, in an amortized analysis, all the iterations of inner loop are added up, then this sum is averaged over the iterations of the outer loop

There are n iterations executed by the outer loop, $n-1$ of these iterations cost $O(1)$ and one cost $O(n)$

The total number of inner loop iterations is $(n-1) \times 1 + 1 \times n$, the total number of inner loop iterations is about $2n - 1 \approx 2n$

The inner loop iterations is averaged over all the iterations of the outer loop, $\frac{2n}{n} = 2$, therefore the cost per outer loop iteration is 2

The cost to run n outer **for** loop iterations according to amortized analysis is $2n \in O(n)$

Amortized analysis : stacks

Stacks have two constant-time operations, $push(s, x)$ puts an element x on the top of the stack s, and $pop(s)$ takes the top element off of the stack s. These operations cost both $O(1)$, so a total of n operations (in any order) will result in $O(n)$ total time.

Now, a new operation $multipop(s, k)$ is added to the stack which either pop the top k elements in the stack, or if it runs out of elements before that, it will pop all of the elements in the stack and stop.

```
$multipop(s, k)$ /* pop $k$ elements from the stack $s$*/
  while $s$ is not empty and $k > 0$
    $pop(s)$
    $k = k - 1$
```

In $multipop$, the number of iterations of the while loop is $min(s', k)$ where $s' =$ number of objects on the stack. However the worst case of $multipop$ is bounded by $s' = k = n$, so $multipop \in O(n)$

Amortized analysis : stacks

According to our classical time complexity analysis of a sequence of n *push*, *pop* and *multipop* operations :

- Worst-case cost of *multipop* is $O(n)$
- We have a sequence of n operations, one of them can cost $O(n)$
- Therefore, worst-case cost of sequence is $O(n^2)$

Amortized analysis : a second example

We can obtain a tighter upper bound than $O(n^2)$

We notice that each element can be popped only once per time that it's pushed. Therefore, in a sequence of *n push*, *pop* and *multipop* operations :

- # pushes $\leq n$ which implies # pops $\leq n$ as well, including those in multipop
- The total number of *push* and *pop* operations is $\in O(n)$
- Therefore the average cost per operation (including *multipop*) is $\frac{O(n)}{n} = O(1)$.

The total cost for executing *n push*, *pop*, *multipop* operations is $O(n)$. This amortized analysis technique is also called aggregate analysis.

Exercise 1 on amortized analysis

A queue can be implemented using two stacks : stack1 (S_1), stack2 (S_2) :

- enqueue(x) : push x onto stack1
- dequeue() : if stack2 is empty then pop the entire contents of stack1 pushing each element in turn onto stack 2. Now pop from stack2

Assume the two stacks have three operations, push, pop and empty, each with cost $O(1)$. You execute a sequence of n enqueue() and dequeue() operations (total number of operations is n)

A conventional worst case analysis would establish that dequeue takes $O(n)$, thus a sequence of n enqueue and dequeue operations is $O(n^2)$

Exercise 1 on amortized analysis (cont.)

$O(n^2)$ is not a very accurate characterization of the time needed for a sequence of n enqueue and dequeue operations, even though in the worst case an individual dequeue can take $O(n)$ time.

1. Write a pseudo code for this algorithm that is detailed enough to make an analysis of the run time
2. Argue about a better bound on the run time of your algorithm using arguments derived from amortized analysis, to simplify the amortized analysis, consider only the cost of the push and pop operations and not of checking whether stack2 is empty.

Exercise 2 on amortized analysis

Assume of queue is implemented using an array of size n.

If the number of items in the queue get larger than n, then the array is re-allocated to an array of size $2n$, and all items are copied from the old array to the new array.

Argue using amortized analysis that each queue operation in a sequence of n operations is $O(1)$.

Disjoint sets

A disjoint set is a data structure for disjoint sets, i.e. a collection S of sets such that $S = \bigcup_{i=1}^{n}\{S_i\}$, $S_i \cap S_j = \emptyset$

The disjoint set data structure supports the following operations :

- MakeSet(x) : S a new set that contains only x (x is the set representative, the key)
    - A linked list can be used to implement a set
- Union(x, y) : the union of the sets containing the objects x (S_x) and y (S_y) into a new set S
- FindSet(x) : returns a pointer to the representative of the set that contains the object x

There is no operation to ”add” an element to a set

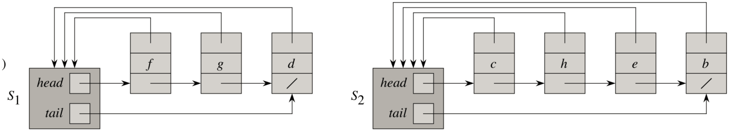
Disjoint Set implementation

Each set is represented by its own linked list.

Each set has a "head" pointer pointing to the first element in the list, a "tail" pointer pointing to the last element.

Each element in the list contains a pointer to the next element in the list, and a pointer back to the head of the list.

The representative of the set is the first element in the list.

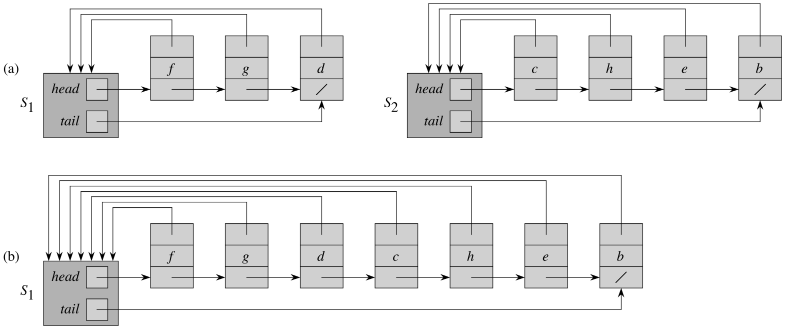


Implementation of disjoint set union operation

$\mathit{Union}(x, y)$ is obtained by appending y's list onto the end of x's list

The representative of x's list becomes the representative of the resulting set

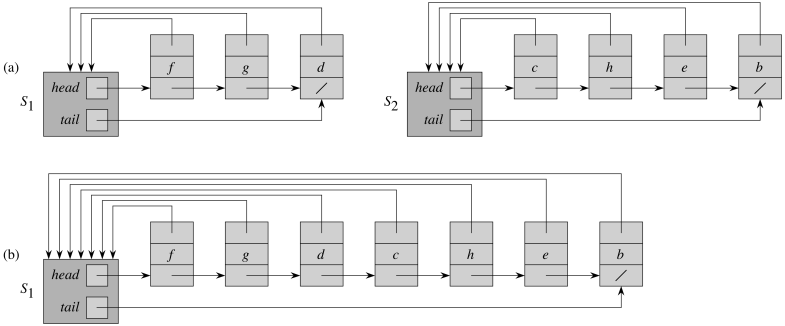
The tail pointer for x's list is used to quickly find where to append y's list.



Implementation of disjoint set union operation

Need to update the tail pointer of the first list and the next pointer of the last element in the first list, cost is constant

Need to update the head pointer of each element in the second list, cost $O(l)$ where l is the length of the second list



Disjoint Set : run time analysis

Assume we create n sets :

Operation	number of elements updated
MakeSet(x_1)	$O(1)$
MakeSet(x_2)	$O(1)$
$\vdots$	$\vdots$
MakeSet(x_n)	$O(1)$
Union(x_2, x_1)	1
Union(x_3, x_2)	2
Union(x_4, x_3)	3
$\vdots$	$\vdots$
Union(x_n, x_{n-1})	$n-1$

A sequence of $2n-1$ MakeSet and Union operations ($n-1$ Unions) cost $O(n^2)$ as the cost of the $n-1$ union operations is $\sum_{i=1}^{n-1} i = O(n^2)$ using the above implementation of the Union operation

Disjoint Set Union : the weight-union heuristic

We can improve the running time of the union operation, always copy smaller list into larger list

Establish an upper bound on the number of times the pointer to the head of an element x can be updated :

- The first time x's pointer is updated the resulting set must have had at least 2 members.
- The second time the resulting set must have had at least 4 members.
- For $k \leq n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members.

Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the union operations.

Performance of disjoint set with weight-union heuristic

Since we have n Unions, each element been copied no more than $O(\lg n)$ times, total running time of n Unions is $O(n \lg n)$.

A sequence of m MakeSet, Union and Findset operations, where n operations are MakeSet operations, runs in $O(m + n \lg n)$

Exercise : weighted-union

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic

```
for $(i = 1; i \leq 16; i++)$
    MAKE-SET($x_i$)
for $(i = 1; i \leq 15; i + 2)$
    UNION($x_i, x_{i+1}$)
for $(i = 1; i \leq 13; i + 4)$
    UNION($x_i, x_{i+2}$)
UNION($x_1, x_5$)
UNION($x_9, x_{13}$)
UNION($x_1, x_9$)
FIND-SET($x_2$)
FIND-SET($x_9$)
```

Solution : exercise on weighted-union

```
1- for $(i = 1; i \leq 16; i++)$
       MAKE-SET($x_i$)
3- for $(i = 1; i \leq 15; i + 2)$
       UNION($x_i$ , $x_{i+1}$)
5- for $(i = 1; i \leq 13; i + 4)$
       UNION($x_i$ , $x_{i+2}$)
7- UNION($x_1$ , $x_5$)
8- UNION($x_9$ , $x_{13}$)
9- UNION($x_1$ , $x_9$)
FIND-SET($x_2$)
FIND-SET($x_9$)
```

Originally we have 16 sets, each containing x_i.

After the for loop in line 3 we have 8 sets of size 2 :

$$\{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}, \{9, 10\}, \{11, 12\}, \{13, 14\}, \{15, 16\}$$

After the for loop on line 5 we have 4 sets of size 4 :

$$\{1, 2, 3, 4\}, \{5, 6, 7, 8\}, \{9, 10, 11, 12\}, \{13, 14, 15, 16\}$$

Line 7 results in :

$$\{1, 2, 3, 4, 5, 6, 7, 8\}, \{9, 10, 11, 12\}, \{13, 14, 15, 16\}$$

Line 8 results in :

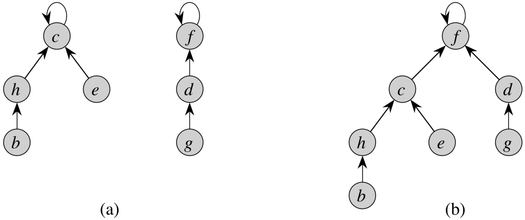
$$\{1, 2, 3, 4, 5, 6, 7, 8\}, \{9, 10, 11, 12, 13, 14, 15, 16\}$$

Line 9 results in :

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$$

FIND-SET(x_2) and FIND-SET(x_9) each return pointers to x_1

Disjoint Set Union : disjoint-set forests implementation



Each element points only to its parent. The root of each tree contains the representative and is its own parent.

The union operation consists of having the root of one tree to point to the root of another tree.

In union by rank, we make the root with smaller rank (smaller height) points to the root with larger rank during a Union operation.

Performance of disjoint-set forests implementation

Union cost $O(1)$, just changing the root pointer of one tree

In the linked list implementation, the operation FindSet(x) is in $O(1)$. In the tree implementation, it is in $O(h)$, the height of the tree

With the union by rank, it is assumed that the height of the trees is $O(\lg n)$

A sequence of m MakeSet, Union and FindSet operations, where n operations are MakeSet operations, runs in $O(m \lg n)$ using union by rank heuristic for the disjoint-set forests implementation

Review on Summations : Patterns in Sums

- **Sum :** $1 + 2 + 3 + \cdots + n$. **Pattern :** each successive term is one plus the previous term
- **Sum :** $2 + 4 + 6 + \cdots + 2n$. **Pattern :** each successive term is two plus the previous term (i.e. the terms are even numbers)
- **Alternate form :** $2 \times (1 + 2 + 3 + \cdots + n)$
- **Sum :** $1 + 2 + 4 + 8 + \cdots + 2^n$. **Pattern :** each successive term is two times the previous term (i.e. terms are powers of two)
- **Alternate form :** $2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^n$

Summation Notation

- General form : “sigma notation” is shorthand for long sums

$$\sum_{i=a}^{b} f(i)$$

- Meaning :

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

assuming a and b are integers and $a \leq b$.

Example of sums

- **Sum :** $1+2+3+\cdots+n$. **Summation notation :**

$$\sum_{i=1}^{n} i$$

- **Sum :** $2+4+6+\cdots+2n$. **Summation notation :**

$$\sum_{i=1}^{n} 2i$$

- **Sum :** $1+2+4+8+\cdots+2^n$. **Summation notation :**

$$\sum_{i=0}^{n} 2^i$$

Operator Priorities

- $\sum$ has the same priority as addition because it represents additions.
- $\sum$ **and** $+$ **:** same priority
  Example :
  $$\sum_{i=a}^{b} 2i + 1 = \left(\sum_{i=a}^{b} 2i\right) + 1$$
- $\sum$ **and multiplication :** multiplication has higher priority

$$\sum_{i=a}^{b} i \times \log_2 10 = \sum_{i=a}^{b} (i \times \log_2 10)$$

Sums and Closed Forms

Definition : A *closed form* for a sum is some function that gives you the value of the sum with only a constant number of operations.

- Example :

```
for($i = 1; i \leq n; i++$) do
    - execute $i$ elementary operations -
end for
```

- $1 + 2 + 3 + \ldots + n = \sum_{i=1}^{n} i = n(n+1)/2$.
- To compute the value from the sum : $n - 1$ operations
- To compute the value from $n(n+1)/2$: 3 operations

Example :

```
for($i = 1; i \leq n; i++$) do
    - execute $n - i$ operations -
end for
```

does

$$\begin{aligned}
\sum_{i=1}^{n}(n-i) &= (n-1)+(n-2)+(n-3)+ \\
&\quad \cdots + 1 + 0 \\
&= \sum_{i=1}^{n-1} i \\
&= \frac{(n-1)n}{2} \text{ operations}
\end{aligned}$$

Sums and Closed Forms : Continue

- **Sum of a constant c :**

$$\begin{aligned}\sum_{i=a}^{b} c &= \overbrace{c + c + \cdots + c}^{b-a+1 \text{ of them}} \\ &= (b-a+1) \times c.\end{aligned}$$

- **Sum of the squares of the first u integers :**

$$\begin{aligned}\sum_{i=1}^{u} i^2 &= 1^2 + 2^2 + 3^2 + \cdots + u^2 \\ &= \frac{u(u+1)(2u+1)}{6}\end{aligned}$$

- **Similar to sum of squares :**

$$\begin{aligned}\sum_{i=1}^{u} i(i+1) &= (1\times 2)+(2\times 3)+(3\times 4)+\\ &\quad \cdots+(u\times(u+1))\\ &= \frac{u(u+1)(u+2)}{3}\end{aligned}$$

- **Sum of powers of a constant c :**

$$\begin{aligned}\sum_{i=0}^{u} c^i &= c^0+c^1+c^2+\cdots+c^u\\ &= \frac{c^{u+1}-1}{c-1}\end{aligned}$$

Example : $\sum_{i=0}^{2} 3^i = (3^3-1)/2 = 13.$

- **Sum of the first u integers times increasing powers of a constant c :**

$$\begin{aligned}\sum_{i=0}^{u} ic^i &= 0\times c^0 + 1\times c^1 + 2\times c^2 + \cdots + u\times c^u \\ &= \frac{((c-1)(u+1)-c)c^{u+1}+c}{(c-1)^2}\end{aligned}$$

Example :

$$\begin{aligned}\sum_{i=0}^{3} i2^i &= 0\times 2^0 + 1\times 2^1 + 2\times 2^2 + 3\times 2^3 \\ &= 34 \\ &= \frac{((2-1)(3+1)-2)2^{3+1}+2}{(2-1)^2}\end{aligned}$$