Algorithms and Data Structures Lecture slides: Analyzing loops, Brassard, sections 4.1 to 4.5

Lecturer: Michel Toulouse

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

Current semester map

Introduction : basic operations, worst case analysis

Asymptotic notations

Analyzing iterative algorithms

Analyzing recursive algorithms

Basic data structures : arrays, linked lists, etc.

Topics cover today

1. Introduction
2. Analyzing loops
   - Analyzing single `for`-loops
   - Analysis of nested loops
3. Worst-case, best-case and average-case analysis
   - Review of worst-case and best-case analysis
   - Average-case analysis
4. Review on summations

Iteration versus recursion

From an algorithm analysis standpoint there are 3 types of algorithms :

- Iterative algorithms where repetitions are driven by loops
- Recursive algorithms which use recursive functions to control repetitions
- Algorithms with a constant number of operations (no loops, no recursions)

The way to count the number of basic operations differ between iterative and recursive algorithms

In this section we study iterative algorithms, where running time is function of the execution of loops.

Summations

The analysis of iterative algorithms makes use of summations which model how basic operations in each iteration of a loop add up.

You might think that summations are simple, not so much when dealing with nested loops, which is where things get more interesting and possibly more complicated...

So we introduce some basic notions related to summations.

Iterative Algorithms

How many basic operations in the following `whatsit` algorithm ?

```
whatsit(input of size n)
for (i = 5; i ≤ n; i++) do
    for (j = 1; j ≤ i; j++) do
        k = 3;
        while k ≤ n do
              - perform one elementary operation -
              k = k * 3;
        end while
    end for
end for
```

Counting basic operations of iterative algorithms

1. Work on one loop at a time.
2. Due to the dependence of inner loops on the loop index of outer loop, we always work from the innermost loop to the outermost loop.
3. After calculating how many elementary operations the innermost loop performs, rewrite the algorithm with the number of elementary operations substituted for the innermost loop.
4. Repeat the process on the new innermost loop.

Single for-loops

Assume that we have the following single for-loop :

for $(i = a; i \leq b; i++)$ **do**
    - execute c elementary operations -
end for

In this loop for each i between a and b, c elementary operations are executed

Let $a = 1; b = 3$

for $(i = 1; i \leq 3; i++)$ **do**
    - execute c elementary operations -
end for

then the # of basic operations is : $c + c + c = \sum_{i=1}^{3} c = 3c$

Single for-loops : loop increment = 1

Assume that we have the following single for-loop :

for $(i = a; i \leq b; i++)$ **do**
    - execute $f(i)$ elementary operations -
end for

In this loop for each i between a and b, $f(i)$ elementary operations are executed

The sum-up of the number of elementary operations executed in this loop is

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

Single for-loops : loop increment $= 1$

Let $a = 1$, $b = n$, and $f(i) = i$ (dependence on loop index) then :

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n \times (n+1)}{2},$$

which gives the result 6 in case $b = n = 3$.

Single for-loops : loop increment $\neq 1$

In the following single for loop, the index increment is x and for each iteration step it executes $f(i)$ elementary operations

for $(i = a; i \leq b; i + x)$
    - execute $f(i)$ elementary operations -
end for

When the index increment differs from 1, the number of loop iterations is no longer $b - a$ where a and b are respectively the first and last index of the loop

for $(i = a; i \leq b; i + x)$
`- execute` $f(i)$ `elementary operations -`
end for

Therefore to compute the number of basic operations $\sum_{i=a}^{b} f(i)$, first the number of loop iterations needs to be calculated

In the case of the above **for** loop, it has $m + 1$ iterations where $m = \lfloor \frac{b-a}{x} \rfloor$ and number of elementary operations the loop executes is

$$\sum_{i=0}^{m} f(a + ix)$$

Example

How many iterations does the following algorithm have ?

```
j = 4; while j ≤ n do -- j = j + 3;
```

We can construct the following table :

Value of loop index	4	4+3	4+3+3	4+3+3+3	···	?
# of elementary operations					···	

and found that j takes on values of the form $4 + 3i$, where i is some integer.

Solving for m

Suppose the last value of $j = 4 + 3m$, so that the next value $4 + 3(m + 1)$ is past the upper bound :

$$4 + 3m \leq n < 4 + 3(m + 1).$$

Solve for m by subtracting 4 from all sides

$$3m \leq n - 4 < 3(m + 1)$$

and then dividing all sides by 3

$$m \leq \frac{n - 4}{3} < m + 1.$$

Since j takes only integer values, m must be an integer. Therefore, m is the largest integer $\leq (n - 4)/3$; that is, $m = \left\lfloor \frac{n-4}{3} \right\rfloor$.

Case 1

How many elementary operations does the following algorithm have ?

for $(i = 3; i \leq n - 3; i++)$
  - perform $i - 1$ elementary operations

end for

- Analyze : This is a single for loop where loop index runs from 3 to $n - 3$ and index incremental 1. Hence,

$$\#\text{ elementary ops } = \sum_{i=3}^{n-3} (i - 1).$$

- To find a closed form for the sum, expand it :

$$\#\text{ elementary ops } = 2 + 3 + 4 + \ldots + (n - 4)$$

This sum nearly matches the known sum of consecutive integers, except the first term is missing :

$$\#\text{ elementary ops } = (\sum_{p=1}^{n-4} p) - 1$$

Substitute the closed form for the known sum :

$$\#\text{ elementary ops } = \frac{(n-4)(n-3)}{2} - 1$$

$$\begin{aligned} \#\text{ elementary ops } &= \frac{(n-4)(n-3)}{2} - \frac{2}{2} \\ &= \frac{n^2-7n+12}{2} - \frac{2}{2} \\ &= \frac{n^2-7n+10}{2} \in \Theta(n^2) \end{aligned}$$

Case 2

How many elementary operations does the following algorithm execute ?

$k = 2$;
while $k \leq n$ do
    - perform k elementary operations -
    $k = k + 3$;
end while

- Analyze : It is a single while-loop with index increment 3 and the loop index from 2 to n.
- A table of the values that the loop index k goes through :

Value of loop index	2	$2+3$	$2+3+3$	$\cdots$	?
# of elementary operations	2	$2+3$	$2+3+3$	$\cdots$	?

The pattern : $k = 2 + 3i$, where i is a non-negative integer.
If $2 + 3m$ is the last value of k

$$2 + 3m \leq n < 2 + 3(m + 1).$$

Solving for m gives

$$m = \left\lfloor \frac{n-2}{3} \right\rfloor < m + 1$$

so the last value of k is

$$k = 2 + 3 \left\lfloor \frac{n-2}{3} \right\rfloor.$$

A table of the values that the loop index goes through :

Value of loop index	$2+3\times 0$	$2+3\times 1$	$2+3\times 2$	$\cdots$
# of elementary operations	$2+3\times 0$	$2+3\times 1$	$2+3\times 2$	$\cdots$

Value of loop index	$\cdots$	$2+3\lfloor (n-2)/3 \rfloor$
# of elementary operations	$\cdots$	$2+3\lfloor (n-2)/3 \rfloor$

In conclusion, the `while`-loop performs
$(2+3\times 0)+(2+3\times 1)+(2+3\times 2)+\cdots+(2+3\lfloor (n-2)/3 \rfloor)$
elementary operations.

Find a closed form for

$$(2 + 3 \times 0) + (2 + 3 \times 1) + (2 + 3 \times 2) + \cdots + (2 + 3\lfloor (n-2)/3 \rfloor).$$

Separate into two sums, a sum of 2's and a sum of multiples of 3 :

$$(2 + 2 + 2 + \cdots + 2) + (3 \times 0 + 3 \times 1 + 3 \times 2 + \cdots + 3 \times \lfloor (n-2)/3 \rfloor)$$

How many 2's are in the first sum ? The number of 2's is equal to the number of terms in the other sum, so there are $\lfloor (n-2)/3 \rfloor + 1$ two's altogether. The value of the first sum is $2 \times (\lfloor (n-2)/3 \rfloor + 1)$.

Each term in the second sum has a common factor of 3, which can be factored out of the sum.

We now have

$$2 \times \left(\left\lfloor \frac{n-2}{3} \right\rfloor + 1 \right) + 3 \times \left(0 + 1 + 2 + \cdots + \left\lfloor \frac{n-2}{3} \right\rfloor \right)$$

The rightmost sum is one of the known sums (plus a 0 term which doesn't change the value of the sum).

$$2 \times \left(\left\lfloor \frac{n-2}{3} \right\rfloor + 1 \right) + 3 \times \sum_{i=1}^{\lfloor (n-2)/3 \rfloor} i.$$

Multiplying out the leftmost term and substituting the closed form for the rightmost sum :

$$2\left\lfloor\frac{n-2}{3}\right\rfloor+2+3\times\frac{(\lfloor(n-2)/3\rfloor)(\lfloor(n-2)/3\rfloor+1)}{2}$$

The first term and the last term have a common factor of $\lfloor(n-2)/3\rfloor$:

$$\left\lfloor\frac{n-2}{3}\right\rfloor\times\left(2+\frac{3}{2}\left(\left\lfloor\frac{n-2}{3}\right\rfloor+1\right)\right)+2$$

The final answer can be written as a polynomial in $\lfloor(n-2)/3\rfloor$: the number of elementary operations performed by the `while`-loop is

$$\frac{3}{2}\left(\left\lfloor\frac{n-2}{3}\right\rfloor\right)^2+\frac{7}{2}\left\lfloor\frac{n-2}{3}\right\rfloor+2\in\Theta(n^2).$$

Analysis of nested loops

for $(i = 1; i \leq \lfloor n/3 \rfloor; i++)$ **do**
    $j = 0$
    while $j \leq 3i$ **do**
        - execute $n - i + 1$ elementary operations
        $j = j + 3$
    end while
end for

Requires the used of nested sums

The terms of a sum can be sums in sigma notation :

$$\sum_{i=a}^{b} \sum_{j=c(i)}^{d(i)} f(i,j)$$

Each term of $\sum_{i=a}^{b}$ is a sum :

$$\sum_{i=a}^{b} \sum_{j=c(i)}^{d(i)} f(i,j) = \sum_{i=a}^{b} \left(\sum_{j=c(i)}^{d(i)} f(i,j) \right)$$

$$\sum_{i=a}^{b}\sum_{j=c(i)}^{d(i)} f(i,j) = \sum_{i=a}^{b}\left(\sum_{j=c(i)}^{d(i)} f(i,j)\right)$$

$$= \underbrace{\left(\sum_{j=c(a)}^{d(a)} f(a,j)\right)}_{i=a \text{ term}} + \underbrace{\left(\sum_{j=c(a+1)}^{d(a+1)} f(a+1,j)\right)}_{i=a+1 \text{ term}}$$

$$+ \underbrace{\left(\sum_{j=c(a+2)}^{d(a+2)} f(a+2,j)\right)}_{i=a+2 \text{ term}} + \cdots$$

$$+ \underbrace{\left(\sum_{j=c(b)}^{d(b)} f(b,j)\right)}_{i=b \text{ term}}$$

Example :

$$\sum_{i=2}^{n}\sum_{j=n-i}^{n} 2j = \underbrace{\sum_{j=n-2}^{n} 2j}_{i=2 \text{ term}} + \underbrace{\sum_{j=n-3}^{n} 2j}_{i=3 \text{ term}} + \underbrace{\sum_{j=n-4}^{n} 2j}_{i=4 \text{ term}} + \cdots + \underbrace{\sum_{j=n-n}^{n} 2j}_{i=n \text{ term}}$$

We can expand out each of the interior sums, too.

$$\begin{aligned}\sum_{i=2}^{n}\sum_{j=n-i}^{n} 2j \quad = \quad & \underbrace{(2(n-2)+2(n-1)+2n)}_{i=2 \text{ term}} \\ & + \underbrace{(2(n-3)+2(n-2)+2(n-1)+2n)}_{i=3 \text{ term}} \\ & + \underbrace{(2(n-4)+2(n-3)+2(n-2)+2(n-1)+2n)}_{i=4 \text{ term}} \\ & + \cdots + \underbrace{(2\times 0+2\times 1+2\times 2+\cdots+2n)}_{i=n \text{ term}}\end{aligned}$$

How to analyze nested loops

for $(i = 1; i \leq \lfloor n/3 \rfloor; i++)$ **do**
    $j = 0$
    while $j \leq 3i$ **do**
        - execute $n - i + 1$ elementary operations
        $j = j + 3$
    end while
end for

First step :

Analyze the innermost loop, treating the loop index of the outermost loop as a constant.

$j = 0$;
while $j \leq 3i$ **do**
    - do $n - i + 1$ elementary operations
    $j = j + 3$;
end while

A table of the values that the loop index goes through :

Value of loop index	0	3	6	$\cdots$	$3i$
# of elementary operations	c	c	c	$\cdots$	c

where $c = n - i + 1$.

Analyzing

- The number of elementary operations in each iteration of while-loop is $n - i + 1$ and does not depend on loop index j.
- The while-loop has index increment 3.
- The loop index is from 0 to $3i$. The number of iterations is

$$1 + \lfloor \frac{3i - 0}{3} \rfloor = i + 1.$$

Hence, the total number of elementary operations in this while-loop is

$$(i + 1)(n - i + 1).$$

How to analyze nested loops : continue

Next step : substitute the number of elementary operations for the innermost loop in the nested loop :

for $(i = 1; i \leq \lfloor n/3 \rfloor; i++)$ **do**
    - `perform` $(i+1)(n-i+1)$ `elementary operations`
end for

The number of elementary operations performed by this `for`-loop is

$$\sum_{i=1}^{\lfloor n/3 \rfloor} (i+1)(n-i+1).$$

Find a closed form for $\sum_{i=1}^{\lfloor n/3 \rfloor}(i+1)(n-i+1)$.
First, multiply out the term :

$$\begin{aligned}(i+1)(n-i+1) &= (i+1)(-i+(n+1)) \\ &= -i^2+(n+1)i-i+(n+1) \\ &= -i^2+ni+(n+1)\end{aligned}$$

Separate the sum into three sums :

$$\begin{aligned}\sum_{i=1}^{\lfloor n/3 \rfloor}(i+1)(n-i+1) &= \sum_{i=1}^{\lfloor n/3 \rfloor}\left(-i^2+ni+(n+1)\right) \\ &= -\sum_{i=1}^{\lfloor n/3 \rfloor} i^2 + n\sum_{i=1}^{\lfloor n/3 \rfloor} i + \sum_{i=1}^{\lfloor n/3 \rfloor}(n+1)\end{aligned}$$

Substitute the closed form for each of the sums (closed form of $\sum_{i=1}^{u} i^2 = \frac{u(u+1)(2u+1)}{6}$) :

$$
\begin{aligned}
& -\sum_{i=1}^{\lfloor n/3 \rfloor} i^2 + n \sum_{i=1}^{\lfloor n/3 \rfloor} i + \sum_{i=1}^{\lfloor n/3 \rfloor} (n+1) \\
&= -\frac{\lfloor n/3 \rfloor (\lfloor n/3 \rfloor + 1)(2\lfloor n/3 \rfloor + 1)}{6} + n\frac{\lfloor n/3 \rfloor (\lfloor n/3 \rfloor + 1)}{2} \\
&\quad + \lfloor n/3 \rfloor (n+1)
\end{aligned}
$$

Take out the common factor of $\lfloor n/3 \rfloor$ and simplify the rest :

$$
\begin{aligned}
& -\frac{\lfloor n/3 \rfloor(\lfloor n/3 \rfloor + 1)(2\lfloor n/3 \rfloor + 1)}{6} + n\frac{\lfloor n/3 \rfloor(\lfloor n/3 \rfloor + 1)}{2} + \lfloor n/3 \rfloor(n+1) \\
&= \quad \lfloor n/3 \rfloor \left(\frac{-(\lfloor n/3 \rfloor + 1)(2\lfloor n/3 \rfloor + 1)}{6} + \frac{n(\lfloor n/3 \rfloor + 1)}{2} + (n+1) \right) \\
&= \quad \lfloor n/3 \rfloor \left(\frac{-(2\lfloor n/3 \rfloor^2 + 3\lfloor n/3 \rfloor + 1)}{6} + \frac{n\lfloor n/3 \rfloor + n}{2} + (n+1) \right)
\end{aligned}
$$

Put over a common denominator (6), and combine like terms :

$$\begin{aligned} & \lfloor n/3 \rfloor \left(\frac{-(2\lfloor n/3 \rfloor^2 + 3\lfloor n/3 \rfloor + 1)}{6} + \frac{n\lfloor n/3 \rfloor + n}{2} + (n+1) \right) \\ = \quad & \lfloor n/3 \rfloor \frac{-(2\lfloor n/3 \rfloor^2 + 3\lfloor n/3 \rfloor + 1) + 3n\lfloor n/3 \rfloor + 3n + 6n + 6}{6} \\ = \quad & \frac{\lfloor n/3 \rfloor}{6} (\lfloor n/3 \rfloor (3n - 2\lfloor n/3 \rfloor - 3) + 9n + 5) \end{aligned}$$

The original nested loops performed

$$\frac{\lfloor n/3 \rfloor}{6} (\lfloor n/3 \rfloor (3n - 2\lfloor n/3 \rfloor - 3) + 9n + 5)$$

elementary operations.

Sort by Insertion I

- Problem statement : Sort in increasing order an array of n integers, output the sorted array.

  Input : Sequence of integers $a_1, a_2, \ldots, a_n$
  Output : The sequence in the increasing order

- Idea of Insertion Sort Algorithm :
  1. Sort the first two elements of the list
  2. Insert the third element on its temporary "correct" position (right position among the first three elements of the array) by shifting on the right elements greater than this third element.

  i. Insert the ith element on its temporary "correct" position (correct position among the first i elements of the array) by shifting on the right elements greater than this i element.

  n. Insert the nth element on its correct position by shifting on the right elements greater than this nth element.

Sort by Insertion II

- Algorithm :

```
Insertion(A[1..n])
for (i = 2; i <= n; i++)
    v=A[i]; j=i-1;
    while(j > 0 & A[j] > v)
        A[j+1]=A[j]; j-;
    A[j+1] = v;
```

- How many elementary operations this sorting algorithm execute on each of the two following inputs : $[4, 3, 2, 1], [1, 2, 3, 4], [3, 1, 4, 2]$?

"Worst-case" and "Best-case" analysis

- Assume we use the comparisons in the "while" loop as our abstract operation.
- **Worst-case :** When the array A is sorted in descending order, $A[j] > v$ for 1 to $i-1$ for every iteration of the "for" loop. The total number of comparisons is $\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$.
- **Best-case :** When the array A is already sorted in ascending order, the algorithm only executes n comparisons !
- Remark : Depending on each instance of problem, algorithms may halt faster or slower. It is necessary to analyze the algorithms in the best-case, the worst-case, as well as the average-case.

Worst-case Analysis

Compute the worst-case running time of the algorithm.

Worst-case running time : The maximum number of elementary operations performed by the algorithm on any input of size n.

$$T(n) = \max_{I \text{ input of size } n} \{\#\text{ of elementary operations executed for input } I\}.$$

An example for the worst-Case analysis I

- Problem statement : Search an integer x in an array of size n. Return 0 if x not found, else return index of the first occurence of x.

  Input : A given array of n distinct integers $L[1..n]$, and an integer x
  Output : The location of x in the sequence (is 0 if x is not in the sequence)

- Idea of Linear Algorithm : Compare x successively to each term of the sequence until a match is found.

An example for the worst-Case analysis II

- Algorithm :

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

An example for the worst-Case analysis III

- Analysis :
  - The elementary operation is : "**if** $L[i] = x$"
  - We need to find the worst-case inputs for this algo.
  - **Worst-case inputs :** Inputs of size n that cause the algorithm to perform the most work :
    - last entry is x and no other entry is x
    - no entry is x

    In each case, linear search performs n elementary operations. If $T(n)$ is the worst case running time of linear search on inputs of size n, then $T(n) = n$.

Exercise

- **Problem statement :** Test if an arrray of size n is sorted in increasing order. If so return true.
- Algorithm :

TestForAscendingOrder($L[1..n]$)
$i = 1$; ascending = true ;
while ($i \leq n - 1$ & ascending) **do**
    ascending = $(L[i] \leq L[i + 1])$;
    $i = i + 1$;
return ascending ;

- Choose elementary operation(s) and decide what the size of the input is for the algorithm.
- Give a worst-case input and compute how many elementary operations are performed for it.
- Give a "best-case" input (uses fewest elementary ops) and compute how many elementary operations are performed for it.

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

Another way to ignore the differences among inputs of the same size is to perform an average-case analysis.

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 with which the inputs of size n will occur ?

Solution : Make some (simplifying) assumption. 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.)

Inputs of size n :

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

Probabilities and assumptions :

- 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 $1-p$.

Rewrite the table :

Category	description	Probability	# of elementary 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

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

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

$$= \left(\sum_{i=1}^{n} \left(\frac{p}{n} \times i\right)\right) + (1-p)n$$

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

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

- 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.

Consider the following search algorithm :

```
FiveSearch(L[1..n], x)
j = 5;
NotFound = true;
while (j ≤ n & NotFound) do
    if x ≤ L[j] then
        NotFound = false;
    else
        j = j + 5;
end while
if NotFound then
    x is not in L
else
    LinearSearch(L[j − 4, j], x)
```

Exercise

- Define the elementary operation to be a comparison between x and an element of L.
- Perform a worst-case analysis of `FiveSearch` :
  - describe the worst-case input(s),
  - give the exact number of elementary operations performed in the worst-case (Assume that n is a multiple of 5 to make the analysis easier).

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 :

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