Time & Space Complexities

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Simple arithmetic function

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$S_{abc}(I) = 0$

Iterative function for summing a list of numbers

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array & pass by value

Space Complexity: Example 2

```
1.  Algorithm Sum(a[], n)
2.  {
3.      s:= 0.0;
4.      for i = 1 to n do
5.          s := s + a[i];
6.      return s;
7.  }
```

Space Complexity: Example 2.

- Every instance needs to store array `a[]` & `n`.
  - Space needed to store `n` = 1 word.
  - Space needed to store `a[ ]` = n floating point words (or at least n words)
  - Space needed to store `i` and `s` = 2 words
- $S_p(n) = (n + 3)$. Hence $S(P) = (n + 3)$.

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition

  $$T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

Regard as the same unit
machine independent

Program running time – Why?

When is the running time (waiting time for user) noticeable/important?

- web search
- database search
- real-time systems with time constraints

Factors that determine running time of a program

- problem size: n
- basic algorithm / actual processing
- memory access speed
- CPU/processor speed
- # of processors?
- compiler/linker optimization?

Running time of a program or transaction processing time

- amount of input: n → min. linear increase
- basic algorithm / actual processing → depends on algorithm!
- memory access speed → by a factor
- CPU/processor speed → by a factor
- # of processors? → yes, if multi-threading or multiple processes are used.
- compiler/linker optimization? → ~20%

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement

    steps per execution × frequency
  - add up the contribution of all statements

Methods to compute the step count

- What is a program step?
  - a+b+b*c+(a+b)/(a-b) → one step;
  - comments → zero steps;
  - `while (<expr>) do` → step count equal to the number of times <expr> is executed.
  - `for i=<expr> to <expr1> do` → step count equal to number of times <expr1> is checked.

Iterative summing of a list of numbers

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;              /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;         /* last execution of for */
   return tempsum;
   count++;         /* for return */
}
```

2n + 3 steps

Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
        count++;        /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

2n+2

Tabular Method

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Time Complexity: Example 3

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[],n,m)	0	–	0
2	{	0	–	0
3	for i=1 to n do;	1	n+1	n+1
4	for j=1 to m do	1	n(m+1)	n(m+1)
5	s = s+a[i][j];	1	nm	nm
6	return s;	1	1	1
7	}	0	–	0
				2nm+2n+2

Exercises

Coding example #1

```
for ( i=0 ; i<n ; i++ )
    m += i;
```

O(n)

Exercises

Coding example #2

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<n ; j++ )
             sum[i] += entry[i][j];
```

O(n^2)

Exercises

Coding example #3

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<i ; j++ )
            m += j;
```

Total iterations=0+1+2+...+(n−1)

O(n^2)

Exercises

Coding example #4

```
i = 1;
while (i < n) {
      tot += i;
      i = i * 2;
}
```

O(log n)

Exercises

Example #4: equivalent number of steps?

```
i = n;
while (i > 0) {
    tot += i;
    i = i / 2;
}
```

O(log n)

Exercises

Coding example #5

```
for( i=0 ; i<n ; i++ )
  for( j=0 ; j<n ; j++ )
    for( k=0 ; k<n ; k++ )
      sum[i][j] += entry[i][j][k];
```

O(n^3)

Exercises

Coding example #6

```
for( i=0 ; i<n ; i++ )
  for( j=0 ; j<n ; j++ )
     sum[i] += entry[i][j][0];
```

O(n^2)

```
for( i=0 ; i<n ; i++ )
  for( k=0 ; k<n ; k++ )
     sum[i] += entry[i][0][k];
```

O(n^2)