Topic 11

Computational Complexity

資料結構與程式設計

Data Structure and Programming

11/18/2015

Knowing the language basics, and having the basic idea of software engineering,

the next big thing for writing a good program is to consider the computational complexity

Why should we care?

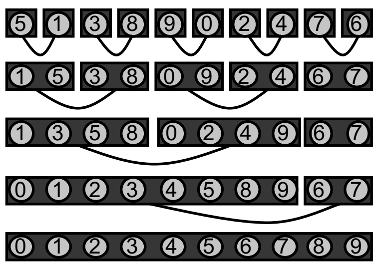
- The most classic example is the "<u>sorting algorithm</u>"
- With straightforward "bubble sort"

```
bubbleSort(arr, n) {
  for (i = 0 to n - 1)
    for (j = i+1 to n - 1)
      if (arr[i] > arr[j])
       swap(arr[i], arr[j]);
}
```

  - Best case: original list is in ascending order
    - ➔ n + n(n – 1) / 2 "for" conditions
    - ➔ (n – 1)(n – 2) / 2 "if" comparison operations
  - Worst case: original list is in descending order
    - ➔ Best case + (n – 1)(n – 2) / 2 "swap" operations
    - ➔ assume (1 swap ~= 3 copies)

➔ How fast can you sort a n-element array?

A Better Sorting Algorithm




```
/*** Merge Sort ***/
// for easier explanation, index = [1, size]
// tmpArr has the same size as arr
mergeSort(arr, n) {
  for (i = 1 to n; i *= 2) {
    mergeSub(arr, tmpArr, n, i);
    i *= 2;
    mergeSub(tmpArr, arr, n, i);
  }
}
mergeSub(arr, resArr, n, i) {
  for (j = 1 to n - 2*i +1; j += 2*i)
    mergeArr(arr, resArr, j, j+i-1, j+2*i-1);
  if ((j+i-1) < n)   // Remaining (< 2*i) or (< i) elements
    mergeArr(arr, resArr, j, j+i-1, n);
  else copyArr(resArr, arr, j, n);
}
mergeArr(arr, resArr, n1, n2, n) {   // merge 2 ordered arrays
  for (i1 = n1 to n2, i2 = n2+1 to n, r = i1; ++r)
    resArr[r] = (arr[i1] <= arr[i2])? arr[i1++] : arr[i2++];
  (i1 > n2)? copyArr(resArr, arr, i2, n)
          : copyArr(resArr, arr, i1, n2);
}
```

Merge Sort Analysis

- Note: the best and worst case complexities are about the same
- Approximately ---
  - n function calls
    $n*\log_2 n$ "for" evaluations
    $n*\log_2 n$ "if" comparisons
    $n*\log_2 n$ copies

Comparison: Bubble vs. Merge Sort

- Assume
  1. "for", "if", "copy" operation: 1 time unit
  2. Function call: 10 time units
- Bubble: $(n^2 - n + 1) \sim (3*n^2 - 5*n + 4) / 2$
  Merge: $n*\log_2 n + 10*n$

n	10	100	1000	10K	1M
Bubble	91 127	10K 15K	1M 1.5M	100M 150M	1T 1.5T
Merge	140	1.7K	20K	240K	30M
B/M	0.91	8.8	75	625	50K

Comparison: Bubble vs. Merge Sort

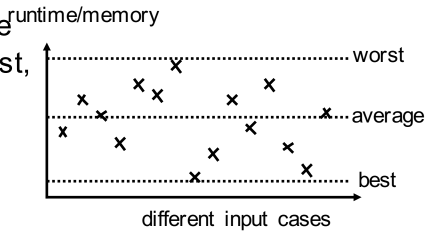
- Time complexity
  - Bubble sort
    - OK for low n
    - Becomes quadric when n gets large
  - Merge sort
    - Much better than bubble sort for large n
- Space tradeoff
  - Bubble sort needs just 1 extra element space
  - Merge sort needs extra n-element space (tmpArr)
    - There are other merge sort algorithms that require just 1 extra space, but the performance is not as well

FYI, there are many interesting videos for "sorting algorithms"

- (e.g.) The folk dance series
  - Quick sort: http://www.youtube.com/watch?v=ywWBy6J5gz8
  - Merge sort: http://www.youtube.com/watch?v=XaqR3G_NVoo&feature=related
  - Bubble sort: http://www.youtube.com/watch?v=lyZQPjUT5B4&feature=related
  - Insertion sort: http://www.youtube.com/watch?v=ROalU379l3U&feature=related
  - Shell sort: http://www.youtube.com/watch?v=CmPA7zE8mx0&feature=related

Measurement of Complexity

- As we can see, the performance for an algorithm may vary on best, worst, and average cases



- Which case is more important?
- Worst case?
  - Yes, *prepare for the raining days*... a robust program should be able to handle such cases
- Average case?
  - Yes, it may be the most commonly happened.
- Best case?
  - Yes, if it happens, we should take the advantage of it.

An Engineering Approach

- Think of the “cache” mechanism in a computer’s memory hierarchy

  ➔ Don’t leave the low-hanging fruits on the tree

  ➔ Try the simple algorithm for the good cases first

  ➔ Turn to complex method only when it gets complicated

- Engineering approach
  1. Try super fast dirty approach
  2. Use heuristic to handle mostly common cases
  3. Turn to a complete algorithm for the remaining difficult cases, if necessary

Example: Pattern Generation Problem

- Given
  - m logic functions $F_1(x_1, x_2,..., x_n)$, F_2,..., F_m, where x_1, x_2,..., x_n are the common input variables
    ➔ n-input / m-output circuit
  - Expected output values on { $F_1, F_2,..., F_m$ }
- Find
  - An assignment to { $x_1, x_2,..., x_n$ } which satisfies the output function values
- Algorithms
  - Complete: may take 2^n operations
  - Random: may find it in a few tries; worst case still 2^n

  ➔ Try "random" for a few patterns first

Overhead??

- In the cache memory case
  - Let hit rate = h,............................(0.0 < h < 1.0)
    memory access time = t,
    cache access time = c*t,.......(0.0 < c < 1.0)
  - Ave access time = h*c*t + (1 – h)*(1 + c)*t
    = t + (c – h) * t

  ➔ Has overhead if "c > h" (any intuitive explanation?)
- Similarly, this can apply to our engineering approach
- Moreover, if the partial result obtained in the quick step can be reused in the later steps
  ➔ Possibly to guarantee "overhead-free"
  ➔ Usually used when there're many repeated problems
  ➔ Best case: t – h*(1 – c) * t (why?)

Importance of Complexity Analysis

- A good "algorithm" should
  1. Be able to finish the task with the fewest operations
  2. Use as little memory as possible

However, the above two objectives are usually mutually conflicting, so ---

- A good "program" should
  1. Be able to strike the balance between runtime and memory complexities
  2. Have multiple strategies to handle best, average, and worst cases

But, how do we "measure" the complexity of a program?

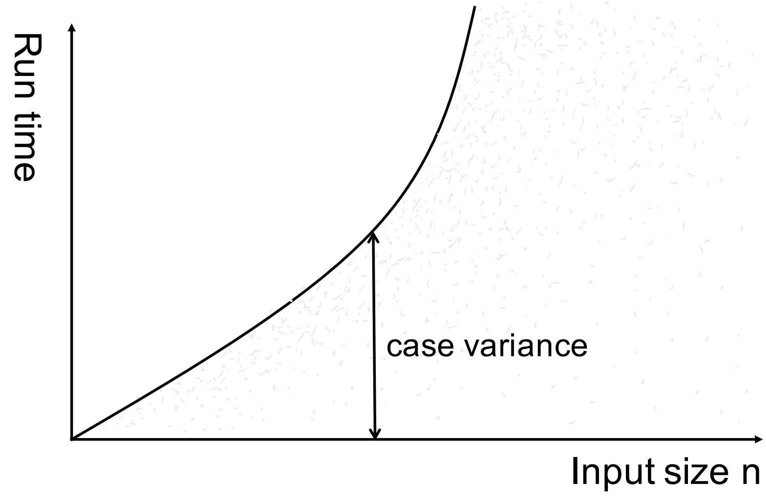
Quantitative Complexity Measurement

- How to describe the complexity of an algorithm/program?
  - Number of (normalized) operations
- Number of operations in terms of what?
  - Input size, number of objects, etc
- But the performance varies case by case, and usually needs infinite sampling to determine the best/average/worst cases

  → Describe the complexity in a range?

  → Use "upper" or "lower" bound !!

What cause the performance variance?



Program Complexity Measurement

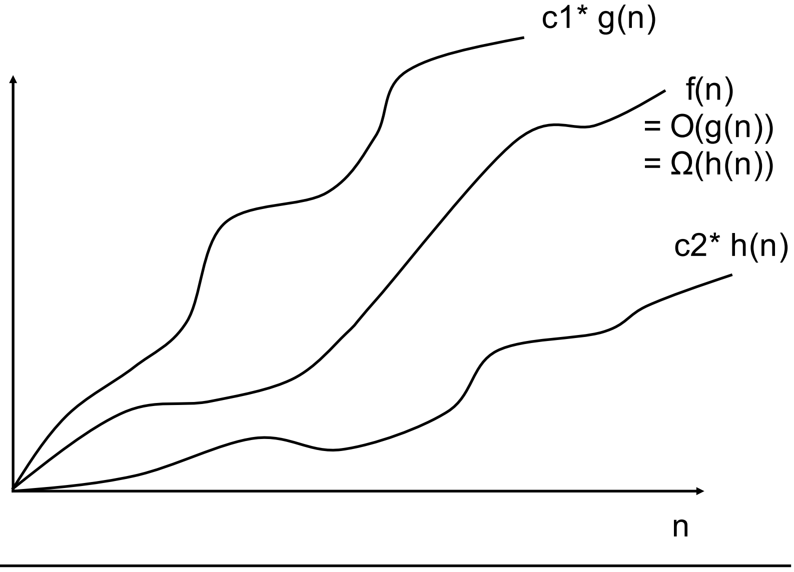
- Measured by number of instructions
  - (For asymptotic measurement) Should we care about the runtime difference between different instructions? (Not really, why??)

1. Analyze the control paths
   - What are the best and worst program flow
2. Focus on the looping statements with non-constant range variables
3. Use rules of sum and product to derive the asymptotic measurements

Asymptotic Notation (O, Ω, Θ)

- Big 'oh' O
  (bounded above by / no worse than / grows as or slower)
  - $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$
  - e.g. $4n^2 + 2n + 3 \rightarrow O(n^2)$, let c = 5
- Omega Ω
  (bounded below by / no better than / grows as or faster)
  - $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \ge cg(n)$ for all n, $n \ge n_0$
  - e.g. $4n^2 + 2n + 3 \rightarrow \Omega(n^2)$, let c = 4
- Theta Θ (bounded above and below by)
  - $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1g(n) \le f(n) \le c_2g(n)$ for all n, $n \ge n_0$
  - e.g. $4n^2 + 2n + 3 \rightarrow \Theta(n^2)$, let $c_1 = 4$, $c_2 = 5$

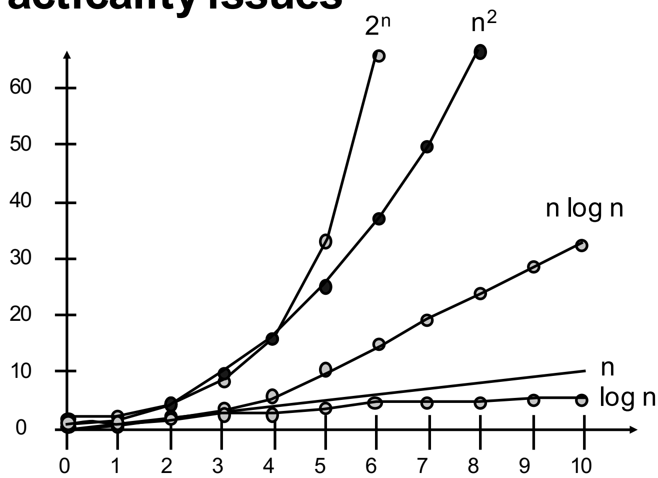
Conceptually....



Properties about (O, Ω, Θ)

1. $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
2. $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
3. Let $p(n)$ be a polynomial function with degree d
   ➔ $p(n) = \Theta(n^d) = O(n^d) = \Omega(n^d)$
4. Let c be any non-negative constant integer
   ➔ $p(n) = O(c^n)$ for $c > 1$
   ➔ e.g. Use a polynomial time heuristic algorithm to solve an exponential complexity problem
5. $\log^k n = O(n)$ for any power k

Practicality issues



◆ How far can you go, if you used "2^n", or "n^2" algorithms??

When we say some program has the complexity $O(...)$ or $\Omega(...)$,
Does $O(...)$ mean the worst case and $\Omega(...)$ mean the best case?

Not really...

➔ Complexity of an algorithm vs.
Performance measurement of a case

1. Input size or number of objects
2. Input values
3. Non-determined reason

Example of Complexity Analysis

```
int binarySearch(int* a, const int x, const int n)
{
   int left = 0, right = n - 1;
   while (left <= right) {
      int middle = (left + right) / 2;
      switch (compare(x, a[middle]) {
         case '>': left = middle + 1; break;
         case '<': right = middle - 1; break;
         case '=': return middle;
      }
   }
   return NOT_FOUND;
}
```

◆ Complexity = $\Theta(\log n)$ (why??)

Example of Complexity Analysis

```
void magicSquare(int** square, int n)
{
   // n must be odd
   int i, j, k, l;
   for (i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         square[i][j] = 0;
   square[0][(n - 1) / 2] = 1;

   key = 2; i = 0; j = (n - 1) / 2;
   while (key <= n * n) {
      if (i - 1 < 0) k = n - 1; else k = i - 1;
      if (j - 1 < 0) k = n - 1; else l = j - 1;
      if (squre[k][l] != 0) i = (i + 1) % n;
      else { i = k; j = l; }
      square[i][j] = key;
      key++;
   }
}
```

- Complexity = $O(n^2)$ (why ??)

Example of Complexity Analysis

```
void permuteGen(char* a, const int k, const int n)
{
   if (k == n - 1) {
      for (int i = 0; i < n; i++)
         cout << a[i] << " ";
      cout << endl;
   }
   else {
      for (int i = k; i < n; i++) {
         swap(a[k], a[i]);
         permuteGen(a, k + 1, n);
         swap(a[k], a[i]);
      }
   }
}
```

- Complexity = $\Theta(n(n!))$ (why??)

Summary

- Important to analyze the complexity of your program
  - If the best, or average cases have much smaller complexity
    ➔ Use some special routines to handle them first
  - If the worst case is equal or greater than $O(n^2)$, and n can be big
    ➔ Provide options to terminate your program gracefully
- For complicated problems, time and space complexities are usually complementary
  - Must take care of both at the same time