COEN279 - Design and Analysis of Algorithms

Algorithm and Its Computational Complexity

Dr. Tunghwa Wang

Fall 2016

Announcement

- E-mail: twang1@cse.scu.edu
- Web: http://www.cse.scu.edu/~twang1
- All class handouts for a week shall be available by the Saturday of the week before:
  - Please click the active entries to find out: not ready if result is file not found.
- You need to check into the followings immediately: The same scheme adopted by Dr. Minghwa Wang
  - Syllabus ←will go through in the class
  - Prerequisite
    - Submit answers using e-mail immediately.
  - Programming requirements
  - Programming assignment #0 (bonus assignment #1)
    - Do the practice ASAP because it must be followed for all programming assignment.
    - You always get bonus points – you simply do it yourself.
  - Programming assignment #1
  - Term project requirements

Algorithms

- An algorithm is designed and implemented to solve a computational problem for a target application.
- Other than the correctness of an algorithm, a computer engineering expert should create:
  - Back-end: most efficient algorithm
  - Front-end: most user-friendly API
- In this class, we always assume the correctness. Thus, we only focus on efficiency.
- Efficiency is a complicated issue, but we address complexity instead of efficiency:
  - What is complexity?
    - Same complexity
    - Different complexity

Computers

- An algorithm is also tied with underlying computer designed to run on.
- Considerations:
  - Main memory size
  - Cache
  - Hardware accelerations
    - Floating-point instructions/processors
    - DSP instructions/processors
    - Special purpose co-processors
  - Multi-core/multi-processor

Data Structures

- Abstract data types: a mathematical model with various operations defined on the model:
  - Container: put and get
  - Dictionary: search, insert, and delete
- Modeling is the art of abstracting a real-world application into a clean problem suitable for algorithmic attack.
- Thus an algorithm is tightly associated with underlying data structures designed for a target application. examples

Data Structures

- Data structure and algorithm classes can only teach basic and common techniques used. Once mastered the skills, a target problem may demand:
  - Using derivatives of basic data structures
  - Using compounded data structures
  - Inventing new data structures

Algorithmic Techniques

- Divide-and-conquer
- Greedy algorithm
- Dynamic programming
- Brand-and-bound
- Backtracking
- Probabilistic or randomized algorithm
- Parallel algorithm
- more

Complexity

- Complexity is how fast the performance metric function grows when the input data size grows.
- Complexity is understood using common algebraic functions:
  - Constant function: f(n) = c
  - Polynomial function: $f(n) = a_n*n^m + a_{n-1}*n^{m-1} + \ldots + a_1*n + a_0$
    - Linear function (m=1): f(t) = a*n + b
  - Log function: f(n) = c*log(n)
  - Exponential function: $f(n) = c*a^n$
- Big-O: asymptotic tight worst-case or upper bond ($\leq$)
  - f(n) = O(g(n)) if there exists constant c and n_0 such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$
  - Little-O: worst-case or upper bond (<)

Complexity

- Big-Ω: asymptotic tight best-case or lower bond (≥)
  - f(n) = Ω(g(n)) if there exists constant c and n_0 such that for all $n \geq n_0$, $0 \leq cg(n) \leq f(n)$
  - Little-Ω: best-case or lower bond (>)
- Big-Θ: asymptotically tight bounded
  - f(n) = Θ(g(n)) if there exists constant c_1, c_2, and n_0 such that for all $n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- Order of complexity:
  - constant $O(1) < O(\lg^* N) < O(\lg\lg N) < O(\lg N) < O(\lg^2 N) <$ linear $O(N) < O(N \lg N) < O(N \lg^2 N) <$ quadratic $O(N^2) < O(N^3) <$ exponential $O(2^N) \leq O(N!)$

Complexity

- Analysis: example
  - Worse-case
  - Average-case
  - Best-case
- Kind of complexity:
  - Time complexity
  - Space complexity
  - Cost complexity
  - Other resource complexity: e.g. CPU

Complexity

- Deterministic Polynomial Time (P) is the set of decision problems solvable in polynomial time by a deterministic Turing machine.
- Nondeterministic Polynomial Time (NP) is the set of decision problems solvable in polynomial time by a theoretical non-deterministic Turing machine.
- EXPSPACE $\supseteq$ NEXPTIME $\supseteq$ EXPTIME $\supseteq$ PSPACE $\supseteq$ NP $\supseteq$ P
  - Unsolved problem in computer science: NP = P?
- co-NP is the set of decision problems where the "no" instances can be solved in polynomial time by a theoretical non-deterministic Turing machine.
  - A decision problem is a member of co-NP if and only of its complement is a member of NP.
  - Instances of decision problems in co-NP are sometimes called "counterexamples". In simple terms, co-NP is the class of problems for which efficiently verifiable proofs of "no" instances exist.
  - Unsolved problem in computer science: NP = co-NP?

Complexity

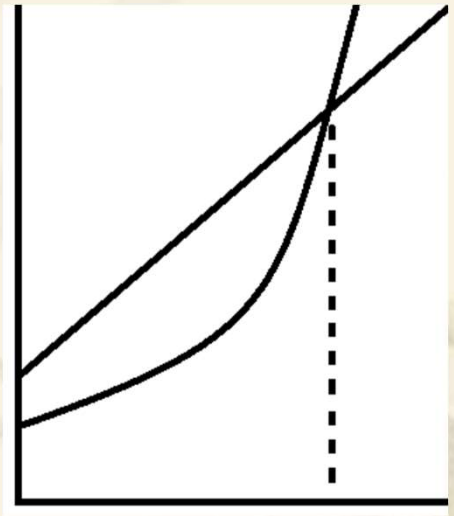
- Rule of complexity:
  - Only the highest complexity term is significant.
  - Rule of sum
  - Rule of product
  - Rule of if-then-else
- Simple analysis:
  - Go through the code following the rules.
- Complicated analysis: recursion
  - Assume $N = 2^k$, the result is the same even $N \neq 2^k$,
  - telescoping a sum: divide the recurrence relation through by N, add up all the equations, cancel all the terms appear on both sides.
  - Substitute the recurrence relation continually on the right-hand side.

Recursion

- Usually, mathematical induction is implied:
  - Start from base case and make progress toward base case
  - Simplify the structure of programs, easy to proof correctness.
  - Suffer run-time penalty, avoid redundant computations.
  - Use recursion in early design, translate into iteration.
  - Optimize with tail recursion elimination by compiler.
- More complications:
  - Indirect recursion

Complexity

- How is it related to efficiency?
  - Trick is on the coefficient c and threshold value n_0.
- Normally, lower complexity implies higher efficiency.
  - We want to solve problem expecting unbounded n: n is in [0, ∞).
- However, we may face problems with n bounded and small.
  - We may want to design algorithm with higher complexity because it is more efficient in the range of [0, n].
- If space is not an issue, we may even want to implement two algorithms such that:
  - Choose one more efficient in the range of [0, n_{max}].
  - Choose one more efficient for n > n_{max}.



Performance

- Real running time:
  - 90% of the run time of a program is spent in 10% of the code.
  - A faster algorithm running on a slower computer will always win for sufficiently large instances.
  - Use profiling and compiler optimization options.
  - Do benchmarking.
- However, in this class, we focus on analytical analysis.
  - Different complexity: we are interested in analyzing it.
  - Same complexity: your boss is interested in getting higher efficiency.
    - If $f_1(n)$ and $f_2(n)$ are both O(n) in such way that $f_1(n) = a_1*n + b_1$ $f_2(n) = a_2*n + b_2$, then usually better algorithm is the one with smaller a value.

Trade-offs

- An algorithm is designed to achieve certain goal which may get achieved by:
  - Space-time trade-off
  - Approximation trade-off
    - Accuracy
    - Optimal solution
- May also subject to strange requirements:
  - Consistent execution time
  - Consistent power consumption

Verification

- If solution already existed in library or on another platform:
  - Use it to generate golden result for validating the results from designed algorithm.
- If an algorithm has a simple equation:
  - Brute-force implementation is always implemented to generate golden result for validating the results from designed algorithm.
- Otherwise:
  - Test cases should cover boundary and invalid input conditions.

Backup Slides

Compiler

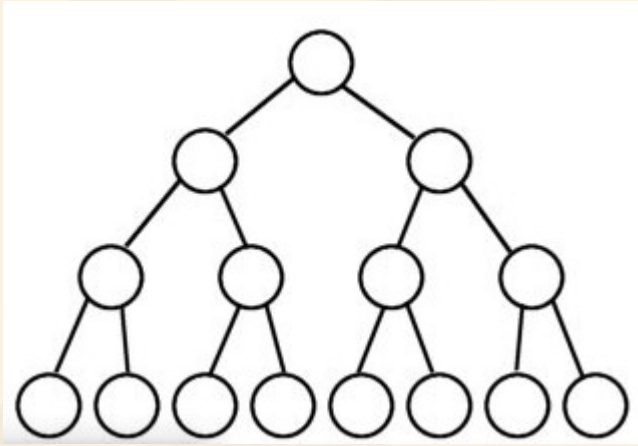
- Parser parses a program file into intermediate representation (IR).
- Abstractions:
  - Keyword: token
  - Variable: symbol table entry
  - Program code: Linked list of abstract syntax trees (AST)
  - Control flow graph: Execution paths
  - more

Automated Drug Synthesis

- Expert system to design organic synthesis of a given target molecule from available simple molecules. return
- Abstractions:
  - Molecule: 3-D non-planar graph
  - Chemical reaction: 3-D graph transformation
  - Problem space:
    - And/Or tree (graph) of graphs
      - Root node: chemical compound to be synthesized
      - Non-terminal nodes: chemical compound not found in catalog
      - Terminal nodes: chemical compound available from catalog
    - A* algorithm

Full Binary Tree Search

- A full binary tree is a tree in which every node other than the leaves has two children.
  - K layers: $n = 2^K - 1$, thus $K = \lg(n+1)$
- Time complexity of search operation:
  - Best case: O(1)
  - Worse case: O(lg(n))
  - Average case: O(lg(n))
    - Sum = $1*1+2*2+3*4+\ldots+K*2^{K-1}$
    - $2*Sum = 1*2+2*4+\ldots+(K-1)*2^{K-1}+K*2^K$
    - $Sum = K*2^K-2^{K-1}-\ldots-4-2-1 = K*2^K-(2^K-1) = (K-1)*2^K+1$
    - Average = Sum/n = ((lg(n+1)-1)*(n+1)+1) / n



Fibonacci Number

- $F_n = F_{n-1} + F_{n-2}$ for $n > 2$
  - $F_2 = F_1 = 1$
- $F_n = (((1+\sqrt{5})/2)^n - ((1-\sqrt{5})/2)^n) / \sqrt{5}$:
  - n=1: $F_1 = (((1+\sqrt{5})/2) - ((1-\sqrt{5})/2)) / \sqrt{5} = \sqrt{5} / \sqrt{5} = 1$
  - n=2: $F_2 = (((1+\sqrt{5})/2)^2 - ((1-\sqrt{5})/2)^2) / \sqrt{5} = \sqrt{5} / \sqrt{5} = 1$
  - If n up to k is true, then by induction,

$$
\begin{aligned}
&F_{k+1} = F_k + F_{k-1} \\
&= ((((1+\sqrt{5})/2)^k - ((1-\sqrt{5})/2)^k) / \sqrt{5}) + ((((1+\sqrt{5})/2)^{k-1} - ((1-\sqrt{5})/2)^{k-1}) / \sqrt{5}) \\
&= (((1+\sqrt{5})/2)^k / \sqrt{5}) - (((1-\sqrt{5})/2)^k / \sqrt{5}) \\
&+ (((1+\sqrt{5})/2)^{k-1} / \sqrt{5}) - (((1-\sqrt{5})/2)^{k-1}) / \sqrt{5}) \\
&= (((1+\sqrt{5})/2)^{k+1} / (\sqrt{5}(1+\sqrt{5})/2)) - (((1-\sqrt{5})/2)^{k+1} / (\sqrt{5}(1-\sqrt{5})/2)) \\
&+ (((1+\sqrt{5})/2)^{k+1} / (\sqrt{5}((1+\sqrt{5})/2)^2)) - (((1-\sqrt{5})/2)^{k+1}) / (\sqrt{5}((1-\sqrt{5})/2)^2)) \\
&= (((1+\sqrt{5})/2)^{k+1} / \sqrt{5})((2/(1+\sqrt{5}))+(2/(1+\sqrt{5}))^2) \\
&- (((1-\sqrt{5})/2)^{k+1} / \sqrt{5})((2/(1-\sqrt{5}))+(2/(1-\sqrt{5}))^2) \\
&= (((1+\sqrt{5})/2)^{k+1} / \sqrt{5})(2/(1+\sqrt{5}))(1+(2/(1+\sqrt{5}))) \\
&- (((1-\sqrt{5})/2)^{k+1} / \sqrt{5})(2/(1-\sqrt{5}))(1+(2/(1-\sqrt{5}))) \\
&= (((1+\sqrt{5})/2)^{k+1} / \sqrt{5})(2(1-\sqrt{5})/(1-5))(1+(2(1-\sqrt{5})/(1-5))) \\
&- ((1-\sqrt{5})/2)^{k+1} / \sqrt{5})(2(1+\sqrt{5})/(1-5))(1+(2(1+\sqrt{5})/(1-5))) \\
&= (((1+\sqrt{5})/2)^{k+1} / \sqrt{5})(((-1+\sqrt{5})/2)(1+(-1+\sqrt{5})/2))) \\
&- (((1-\sqrt{5})/2)^{k+1} / \sqrt{5})(((-1-\sqrt{5})/2)(1+(-1-\sqrt{5})/2))) \\
&= (((1+\sqrt{5})/2)^{k+1} / \sqrt{5}) - (((1-\sqrt{5})/2)^{k+1} / \sqrt{5})
\end{aligned}
$$

Fibonacci Number

- $F_n = F_{n-1} + F_{n-2}$ for n > 2
  - $F_2 = F_1 = 1$
- Recursion:
  - Time complexity: $O(2^n)$
  - Space complexity: O(1)
    - Actually, it is O(n). Why?
- Dynamic programming:
  - Time complexity: O(n)
  - Space complexity: O(n)

return

```
int FN(int n)
{
  if (n > 2)
    return FN(n-1) + FN(n-2);
  else
    return 1;
}
```

```
int FN(int n)
{
  if (n > 2)
  { int val[n], k;

    val[1] = val[2] = 1;
    for (k = 3; k < n; k++)
      val[k] = val[k-1} + val[k-2];
    return val[n-1] + val[n-2];
  } else
    return 1;
}
```