CS/IS C363 Data Structures & Algorithms

Review: Top Down Design

Data Types
Algorithm Design
Strategy: Top-Down Design
Technique: Divide-and-Conquer
Examples: Sorting, Matching Parentheses

Data representation

- Choice of representation is important.
- Representation should be chosen based on the desired set of operations :
  - Are the operations feasible with a given representation?
  - Can they be easily implemented?
  - Can they be efficiently implemented?
- E.g. Natural numbers:
  - Representation: English, Roman numerals, Arabic numerals

Types

- Types classify values:
  - E.g. Taxonomies in Biology
  - Useful for abstract understanding and reasoning
  - E.g. Given Platypus is a mammal
    - Valid reasoning: a platypus does not lay eggs
- Data types classify data values:
  - ***int***, ***char***, ***bool*** …
  - Useful for reasoning as well as for implementing such reasoning
  - e.g. ***int x; int y;*** …. ***x + y*** ….
  - ***x + y*** is an ***int*** value can be inferred.
  - e.g. ***float x; int y;*** …. ***y = x*** …
  - Compiler can identify/prevent (by type checking) such assignments

Data Types

- A (data) type is a set of values
  - grouped on the basis of a common set of operations and hence, typically,
  - implemented using a common representation
  - E.g.
    - int =def { -2k-1,…,-1, 0, 1,…2k-1-1 }
  - operations: { +,-,/,*,% }
  - representation: k bit 2’s complement

Structured Data Types

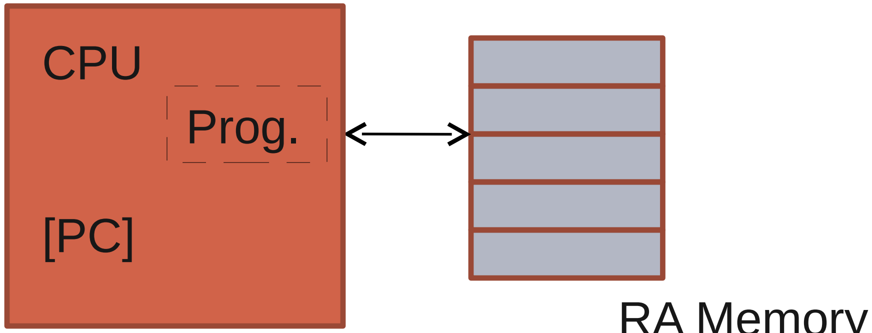
- Programming languages allow programmers to create structured data types:
    - e.g. struct in C: sets of tuples (i.e. cartesian products)
    - The common set of operations (e.g. get or set a field) and the common representation (e.g. contiguous locations) are decided by the language designer and/or compiler implementor.

Course Motivation

- Solving Problems
  - Requires writing Programs ("Concrete solutions")
  - Solve one specific problem i.e. for a class of inputs
  - That can run on one specific language/platform
- Writing Programs
  - Requires designing Algorithms ("abstract solutions")
  - May solve a class of problems
  - Solution not dependent on specific language/platform

Algorithm Design

- High level Specification
  - i.e. independent of specific machines/machine architectures and/or specific language constructs
- Generic Machine Model
- Random Access Machine Model –



Typical Instruction Set

Instructions for

- arithmetic/logic operations,
- load / store, and
- control (jmp/br)

Instructions operate on single memory words (or registers of same size)

Q: Why is this relevant?
Hint: How many operations for

10^20 + 10^15 ?

Algorithm Design

- Top-Down Design (Top Down Decomposition)
  1. Divide the problem into sub problems.
  2. Find solutions for sub problems
  3. Combine the sub solutions.
- How do we find solutions for sub problems?
  - Apply top-down design recursively
  - Q: When do we stop dividing?
  - A: When we reach “atomic” problems.
    - Atomic problems have known solutions

Top Down Design – Example I

- Problem (FindWord):
  - Find the number of occurrences of a word in a body of text.
- Data Model:
  - A word is a *sequence of alphabetic characters*.
  - The given body of text is a string – *any sequence of characters.*
  - We are required to count the occurrences –
  - we will count only if the string occurs as a word
  - i.e. separated by whitespaces or punctuation marks on both sides.

Top Down Design – Example I

- Sub-problems:
  1. *Getting the next word from a long text*.
  2. *Comparing a word with another*.
- Combination :
  - Repeat the two steps in sequence
- Termination:
  - Stop when there is no more text
- Solution:

repeat

get the next word *nw;*

if *nw* **equals** the given word *w* increment *count*

until (no more text)

Top Down Design

- Does any decomposition work?
  - Divide (the problem) only if you know how to combine (the solutions)
  - Combination should be “fairly” easy / obvious.
  - Do not divide into “too many” sub problems.
  - Q: Why?
  - E.g. Find an element E in a list [L0, L1, … Ln-1]
  - Consider this decomposition:
    - n sub-problems each requiring comparison of E with a single list element, say Lj.
  - Q: What is the right decomposition for this example?

Divide-And-Conquer

- Special case of Top-Down-Design
  - Structure of sub problem(s) is same as the (original) problem
  - i.e. once a decomposition and combination have been worked out, the process can be repeated i.e. “recursed”
  - Size of the problem should reduce progressively (as we recur)
  - i.e. size of the input (to the problem/sub-problem)

Divide-And-Conquer: Example i

- Sort, in-place, a list of N elements.
  - Assume list is stored as an array (i.e. logically contiguous memory locations): A[0], A[1], … A[n-1]
- Design
  - Sub-problem: Sort a list of N-1 numbers (A[0], A[1],…A[n-2])
  - Combination: Insert A[n-1] in order (i.e. in the right position)
  - Termination: Stop when size is <= 0.
  - Why?
- Algorithm
  - // Precondition: A is an array indexed from 0 to n-1
  - // Postcondition: A is ordered in place

```
insertSort(A, n) {
 // sort A in-place
}
```

Divide-And-Conquer: Example i

- Algorithm
  - // Precondition: A is an array of size n
  - // Postcondition: A is ordered in place

```
insertSort(A, n) {
    if (n>1) { insertSort(A,n-1);
               insertInOrder(A[n-1], A, n-1); }
}
```

- Note: Of course, insertInOrder has to be designed. End of Note
- Exercise: Apply Divide-and-Conquer to design insertInOrder.

Divide-And-Conquer: Example ii

- Sort a list of N elements.
  - Assume list is stored as an array (i.e. logically contiguous memory locations): (A[0], A[1], … A[n-1])
- Design
  - Sub-problems: Sort sub-lists of (approx.) n/2 numbers
  - (A[0], A[1] … A[mid]) and (A[mid+1], A[mid+2], …, A[n-1])
  - where mid = floor(n/2)
  - Combination: Merge two sorted lists to get a single sorted list.
  - Termination: When list size is <= 1

Divide-And-Conquer: Example ii

- Algorithm
  - // Precondition: A is an array indexed from st to en
  - // Postcondition: A is ordered in place

```
mergeSort(A, st, en)  {
        if (en-st < 1) return;
        mid=floor((st+en)/2);
        mergeSort(A, st, mid);
        mergeSort(A, mid+1,en);
        merge(A, st, mid, A, mid+1, en, A, st, en);
}
```

- Note: merge has to be designed. End of Note
- Exercise: Apply Divide-and-Conquer to design merge.

Divide-and-Conquer – Example III

- Count the number of strings of matched parentheses of length N. (Assume N=2K for some K)
  - Data Model (for strings of matched parentheses):
  - An empty string has matching parentheses (trivially)
  - If a string S has matching parentheses
    then (S) has matching parentheses
  - If non-empty strings S1 and S2 each have matching parentheses then the concatenation S1 S2 has matching parentheses
  - This is an inductive data model:
  - Strings with 0 pairs;
  - Strings with K+1 pairs given strings with K pairs;
  - Strings with K1+K2 pairs given strings with K1 pairs and strings with K2 pairs

Divide-and-Conquer – Example III

- Data Model (for strings of matched parentheses):
- An empty string has matching parentheses
- If a string S has matching parentheses then (S) has matching parentheses
- If non-empty strings S1 and S2 each have matching parentheses then the concatenation S1 S2 has matching parentheses.
- Data Model – Rewritten (combining 2 & 3):
- An empty string has matching parentheses
- If strings S1 and S2 each have matched parentheses
  - then the concatenation (S1) S2 has matching parentheses

  - [Exercise: Argue that these two models are equivalent
  - Argue that this (either one) model is complete.]

Divide-and-Conquer – Example III

- Counting strings of matched parentheses (k pairs):
  - Count matched pairs of the form
    - *(matched_pairs_1)   matched_pairs_2*
- Sub-problems:
  - The sub strings of matched pairs could be of any length:
  - But if *matched_pairs_1* has j-1 pairs, then *matched_pairs_2* must have k-j pairs.
  - so there will be a pair of sub-problems for each j from 1 to k
  - count strings of matched parentheses (j-1 pairs)
  - count strings of matched parentheses (k-j pairs)
- Combination
  - Sum from j = 1 to k
  - Product of the two counts (see sub-problems above)

Divide-and-Conquer – Example III

- Input: K (number of pairs)
- Algorithm:
- // Precondition: K >= 0
- countMatchedPars(K)

```
if K==0 return 1;
else  {
     count = 0;
     for j = 1 to K {
      count  +=  countMatchedPars(j-1) * countMatchedPars(K-j)
     }
     return count;
}
```