CS/IS C363 Data Structures & Algorithms

Review: Top Down Design

Technique: Divide-and-Conquer
Examples: Sorting, Matching Parentheses

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.

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;
}
```

CS/IS C363 Data Structures & Algorithms

Review: Efficiency & Complexity

Resources and Measurements
Time and Space Complexity
- Order Complexity and Notation
- Examples

Cost Models

Resources

- Resources and Usage

Resource	Resource Usage
CPU	CPU Time
Space – Main Memory and Secondary Memory	Memory Used (during computation)
I/O Devices (including networking devices)	I/O Time (for input/output and swapping) Communication Time (for message exchanges)
Power	Power consumed (for the entire process)

- Measurement
  - Absolute (exact) measurement
  - Design Time measurement (estimate)

Algorithmic Complexity

- Design Time Measurement of Resource Usage
  - Measured and expressed in proportion to problem size (i.e. input size)
- Factors:
  - Time Complexity
  - Space Complexity
  - I/O Complexity
  - [Will not be covered in this course.].
  - Energy Complexity
  - [Models are complex and not completely understood today. Not covered in this course.]

Complexity - Example [1]

- Example 1 (Y and Z are input)

```
X = 3 * Y + Z;
    // operations: addition, multiplication, assignment
X = 2 + X;
      // operations: addition, assignment
```

We count it in the abstract:

each statement takes 1 unit of time

under the assumption / knowledge

- that the difference across instruction sets and hardware organization is a “small constant factor” and
- that the typical statement is made of a constant number of operations

Space Complexity : 1 unit

Complexity - Example [2]

```
// a and N are input
j = 0;
while (j < N) {
    a[j] = a[j] * a[j];
    b[j] = a[j] + j;
    j = j + 1;
}
```

// 3 statements and 1 comparison inside the loop
// N iterations, so time taken is 4*N + 2 units
// because comparison happens one extra time

// N+1 units of storage – array b and variable j

Complexity - Example [3]

```
// a and N are input
j = 0;
while (j < N) do {
    k = 0;
    while (k < N) do {    a[k] = a[j] + a[k];   k
= k + 1;  }
    b[j] = a[j] + j;
    j = j + 1;
 }
```

// Inner loop: 3 units per iteration * N iterations = 3 * N + 1
// Outer loop : (3 * N + 5) units per iteration * N iterations
// Total time: 2 + 5*N + 3*N*N
// comparisons happen 1 extra time

// N+2 units of storage – array b and variables j and k

Complexity - examples

- (Order of) Complexity:

Example	Time Units	Time Complexity Order	Space Units	Space Complexity Order
1	2	Constant	1	Constant
2	4*N+2	Linear	N+1	Linear
3	1 + 5*N + 3*N*N	Quadratic	N+2	Linear

- W
  - Capturing proportionality (i.e. growth rate)
  - Machine independent measurement
  - Asymptotic values (of input sizes)

Motivation for Order Notation – Growth Rates

log2N	N	N2	N3	2N
1	2	4	8	4
3.3	10	102	103	>103
6.6	100	104	106	>1025
9.9	1000	106	109	>10250
13.2	10000	108	1012	>102500

Motivation for Order Notation - Big O

- Examples
  - 100 * log2N < N for N > 1000
  - 70 * N + 3000 < N2 for N > 100
  - 105 * N2 + 106 * N < 2N for N > 26

- Abstraction – Upper Bound
  - 100*log2 N is O(log N)
  - 70 * N + 3000 is O(N)
  - 105 * N2 + 106 * N is O(N*N)
  - 2 * 2N + 106 * N17 + 1789 is O(2N)

Motivation for Order Notation

N	N2/10	N3/10	2N/10
2	0.4	0.8	0.4
10	10	102	>102
100	103	105	>1024
1000	105	108	>10249
10000	107	1011	>102499

Compare , for example,

an O(N) Algorithm A running on a machine M1 with speed x, with an O(N*N) Algorithm B running on a machine with speed 10x

Motivation for Order Notation

N	N2/104	N3/104	2N/104
2	0.0004	0.0008	0.0004
10	0.01	0.1	>0.1
1000	100	105	>10246
104	104	108	>102496
106	108	1020	?!*@

Compare , for example,

an O(N) Algorithm A running on a machine M1 with speed x, with an O(N*N) Algorithm B running on a machine with speed 10000x

Order Notation and Conventions

- Asymptotic Complexity – Upper Bound

  g(n) is O(f(n))

  if there is a constant c such that g(n) <= c(f(n))

i.e. if limn□□ (g(n) / f(n)) = c and c<>0

- We are usually not interested in the best case.

- Typical measures are for the worst case and the average (or expected) case.

Binary Search Algorithm

```
// A indexed from 1 to N
low = 1;  high = N;
while (low <= high) {
     mid = (low + high) /2;
    if (A[mid] = = x)  return x;
    else if (A[mid] < x) low = mid +1;
     else high = mid – 1;
}
return Not_Found;
```

Worst case:

- Loop executes until low > high i.e. until size of list becomes 0
- Size halved in each iteration N, N/2, N/4, … 1
- Number of steps is K such that 2K = N i.e. logN steps where N is input size

Time complexity O(logN)

Time Complexity

- Polynomial Time Complexity
  - Time Complexity is **O(Nk)** for some constant k, where N is input size.
- Exponential Time Complexity
  - Time Complexity is **O(2N)**, where N is the input size.

Time Complexity

- Consider the following algorithm:

```
int fact(int N)   {
      j=1;  prod=1;
      while (j<=N) {
           j=j+1; prod=prod*j;
       }
      return prod;
}
```

What is the time complexity?

Is this polynomial time? Why or why not?

- Uniform Cost vs. Logarithmic Cost
  - Uniform Cost – All basic operations cost same (constant) amount of time (irrespective of the data size)
  - Logarithmic Cost – Each operation has a cost that is proportional to the size of the data
  - Hint:
    - Refer to the RAM model slide for assumptions;
    - consider the call *fact(100).*