Lecture Notes

Analysis of Algorithms

Overview

"An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output." It is a sequence of computational steps. It is correct if it terminates with the correct output. - Cormen

We will use algorithms to:

- Sort data
- Perform operations on graphs, trees, hash tables

Analysis of algorithm aims to predict the resources that will been needed by an algorithm:

- CPU / run time / time complexity
- Memory / space complexity
- Network, disk usage?

Random Access Machine

- Single processor
  - Ignore multi-threading
- Access to memory locations are uniform
  - Ignore memory hierarchy (caching, swapping, etc.)
- Basic set of operations / instructions, each taking constant, uniform time.
  - No "magic" operations: no "sort" op.
  - + - / * % ceil floor
  - store, load, copy
  - if, jump, call, return

$T(N)$

We want to identify a time function $T(N)$ that

- Takes as input one or more variables describing the problem *size*
- Returns the number of operations that will run (or something proportional)

Counting Examples

Need to find a function that describes the number of operations as a function of N. Will start with counting # each line is run, since each line alone is constant time.

Algorithm 1

```
Alg1(A,N)
   FOR I IN 0 .. (N-1) DO           N
      FOR J IN 0 .. (N-1) DO        N^2
         X := A[I][J]               N^2
         X := X * 2                 N^2
         A[I][J] := X               N^2
      OD                            N^2
   OD                               N
END
```

Total number of steps is $2N + 5N^2$.

Algorithm 2

```
Alg2(A,N)
   FOR I IN 0 .. (N-1) DO          N
      FOR J IN I .. (N-1) DO       1 + 2 + 3 + ... N
         X := A[I][J]              "
         X := X * 2                "
         A[I][J] := X              "
      OD                           "
   OD                              N
END
```

Total number of steps is

$$2N + 5\sum_{k=1} Nk = 2N + 5\frac{n(n+1)}{2} = \frac{9}{2}N + \frac{5}{2}N^2$$

Algorithm 3

```
Alg3(A,N)
   FOR I IN 0 .. (N-1) DO        N
     FOR J IN 0 .. 99 DO         100*N
        X := A[I][1]             "
        X := X * 2               "
        A[I][1] := X             "
     OD                          "
   OD                            N
END
```

Total number of steps is $2N + 5 * 100 * N = 502 * N$

Algorithm 4

```
Alg4(A,N)
   I := 1                          1
   WHILE I < N DO                  ceil(lgN)+1
      X := A[1][1]                 ceil(lgN)
      X := X * 2                   ceil(lgN)
      A[I][J] := X                 ceil(lgN)
      I := I * 2                   ceil(lgN)
   OD                              ceil(lgN)
END
```

Total number of steps is $2 + 6\lceil \lg N \rceil < 2 + 6(1 + \lg N) = 8 + 6 \lg N$

ABCD:

- For small N, which takes longest to run?
- For large N, which takes longest to run?
- For large N, which takes shortest to run?
- For large N, which one is #2 closest to?

PPQ: Write the time function for the following algorithm:

```
I := 1
WHILE I < N DO
   FOR J in 0 .. (N-1) DO
      X := A[I][J]
      X := X * 2
      A[I][J] := X
   OD
   I := I * 2
OD
```

Introduce Big-O Classes with Live Demonstration

Each define sets of functions whose complexity is "equivilant" as N grows larger and larger.

Class	Name	Example(s)
$O(1)$	Constant	3
$O(\lg N)$	Logarithmic	$8 + 6 \lg N$
$O(N^c), 0 < c < 1$	Fractional Power	$\sqrt{N}$
$O(N)$	Linear	$502N$
$O(N \lg N)$	"N Log N" or Linearithmic	$3N \lg N$
$O(N^2)$	Quadratic	$2N + 5N^2$
		$\frac{9}{2}N + \frac{5}{2}N^2$
$O(N^3)$	Cubic	$N + 5N^3$
		$10N^2 + 5N^3$
$O(2^N)$	Exponential	$(5)2^N$
$O(3^N)$	Exponential	3^N
$\vdots$		
$O(N!)$	Factorial	$3N!$

Why does this matter?

http://www.ccs.neu.edu/home/jaa/CS7800.12F/Information/Handouts/order.html

- Other data points: N^2 at 1M takes 3.17 years. NlgN at 1M takes 33 min.

What does big-O notation mean?

$$O(g(n)) = \{f(n) : \exists_{c>0,n_0>0} S.T. 0 \leq f(n) \leq cg(n), \forall_{n \geq n_0}$$

For algorithm 2: $\frac{9}{2}N + \frac{5}{2}N^2$. Can we prove $O(n)$? Can we prove $O(n^2)$? Pick $c = 3$, solve for N.

$$\begin{aligned} 3N^2 &\geq \frac{9}{2}N + \frac{5}{2}N^2 \\ 6N^2 &\geq 9N + 5N^2 \\ N^2 &\geq 9N \\ N &\geq 9 \end{aligned}$$

So $n_0 = 9$ works.

For algorithm 1: Pick $N_0 = 1$, solve for c:

Proof.

$$\begin{aligned} 5N^2 + 2N &= 2N^2 + 2N(1) \\ &\leq 5N^2 + 2N(N) \\ &= 5N^2 + 2N^2 \\ &= 7N^2 \end{aligned}$$

So $c = 7$ works.

$$O(g(n)) = \{f(n) : \exists c, n_o S.T. 0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

$2N + 5N^2$ vs $\frac{9}{2}N + \frac{5}{2}N^2$ vs $502N$

PPQ: Pick an N_0 and c and show that $10 + 502N$ is $O(N)$.

Rules of Thumb For Function to Big-O

What is important and what isn't in determining the big-O?

- Can ignore low-order terms, in the long run higher order always wins out. N^2 vs $1000N$.
- Can drop the constant on the high order term, because we can always choose a c that is sufficiently higher.

Look for the highest-order term, and ignore the constant.

Examples:

- $4N^2 + 10N + 3 \in O(N^2)$
- $15N \log N + 35N \in O(N \log N)$

ABCD

- A means $O(1)$
- B means $O(\lg N)$
- C means $O(N)$
- D means $O(N^2)$
- WHITE means OTHER

Then:

- $N + 7 \log N \in O(N)$
- $6422 \in O(1)$
- $12N^3 + 20N^2 \in O(N^3)$
- $9 \log N + 10 \in O(\log N)$
- $N + 7 \log N \in O(N)$

Beware of the hidden constant: $200N$ vs $N \lg N$

Rules of Thumb For Algorithm to Big-O

- Loops that increment or decrement by constant amount (e.g. ++, –) contribute $O(N)$
- Loops that increment or decrement by a multiplicative factor (e.g. double, half) contribute $O(\lg N)$
- Nested loops multiply
- Loops in sequence just add

PPQ:

- Write a simple algorithm whose time complexity is $O(N^2 \lg N)$

ABCD: What big-O class is the following:

- A: $O(N)$
- B: $O(N \lg N)$
- C: $O(N^2)$
- D: $O(N^2 \lg N)$

```
FOR I IN 0 .. (N-1) DO
   FOR J IN I .. (N-1) DO
      A[I][J] *= 2
   OD
   K := 1
   WHILE K < N DO
      A[I][K] *= 2
   OD
OD
```

Analyzing Recursive Algorithms: Binary search example

1 Sorts

A sort algorithm is one which takes as input a series of values (usually array, sometimes list) in arbitrary order and produces one in sorted order, either *ascending* or *descending*.
Note that the items in the series need not be simple, e.g. people. Choose a key to sort by: height, age, SSN, name. Keys must have an ordering.

- Ascending Order: the first value is the "smallest", each of the remaining values is equal to or larger than the value before it.
- Descending Order: the first value is the "largest", each of the remaining values is equal to or smaller than the value before it.

Some more definitions:

- In Situ Sort: $O(1)$ additional memory needed.
- Stable Sort: If two entries have the same key, after the sort their relative location is preserved **Question: does anybody know why this is important?**

- Internal Sort: one where all of the values to be sorted fit in main memory at once.

- External Sort: one where all of the values to be sorted DO NOT fit in main memory at once. **Question: why is this distinction important? What differs between the two situations?**

1.1 $O(N^2)$ Sorts

1.1.1 Insertion Sort

Deck of cards. Insert new elements into the proper place of a growing, sorted list.

```
InsertionSort(A,N)
  FOR I in 1 .. N-1 do                    O(N) N
    J := I                                 N   N
    WHILE J > 0 && A[J-1] > A[J] do        N   (N^2 + N)/2
      swap(A[J],A[J-1])                    0   (N^2 + N)/2
      J--                                  0   (N^2 + N)/2
    OD                                     0   (N^2 + N)/2
  OD                                       N    N
END
```

EXAMPLE:

```
8 2 9 6 3
```

ABCD: Is insertion sort stable? **ABCD**: Is insertion in situ?

Now lets return to analyze, but this has a conditional - uncertainty. Most algorithms have different time complexities for different inputs. Best, average, worst-case analyses. Advantages of worst-case:

- Upper bound
- May be common (searching for missing item)
- Often same as average
- Can be easier to compute (avoid probability)

Partway through the insertion sort, since we know that the left portion of the list is sorted, can we not use binary search to locate the spot and then insert it there.

Answer: yes, but this does not affect the runtime. Locating the location will take O(logN), but moving the adjacent cells will still take O(N) time. (Similar to delayed selection sort, except we need to move O(N) data instead of O(1). This assumes random access (array). If we have a list, then finding the spot takes O(N) time, but insertion takes O(1). Either way, we save several swaps, but cannot avoid the O(N) term. Okay so what if we have a binary tree, then finding the spot would be O(logN) and insertion would be O(1). Aha, this is binary tree sort! O(N logN). But this requires O(N) memory. Summary:

Sort	Locating Spot	Inserting	Constant Factor
Vanilla	O(N)	O(1)	Large Constant Factor in Locating
Binary Search	O(logN)	O(N)	Low Constant Factor in Inserting
List	O(N)	O(1)	Not bad, but now you have a list
BST	O(logN)	O(1)	Overall O(N logN), but extra space

1.2 $O(N \log N)$ Sorts

1.2.1 Merge Sort

Divide and conquer approach. Three parts:

1. Divide (into subproblems)
2. Conquer (the subproblems)
3. Combine (the solutions to subproblems)

The concept of merge sort first requires the concept of merging:

```
4 8 11 14
1 5 7 9
MERGE
1 4 5 7 8 9 11 14
```

First, develop a list-based **recursive** merge sort.

```
MergeSort(L)
  IF length(L) < 2 THEN
    return L
  FI

  (Left,Right) := Split(L);
  Left  := MergeSort(Left)
  Right := MergeSort(Right)

  return Merge(Left,Right)
END

Merge(Left,Right)
    Result := empty list
    WHILE !Empty(Left) && !Empty(Right) DO
        IF First(Left) <= First(Right) THEN
            # Pull from Left
            Result := Result + First(Left) # concatenate
            Left := Rest(Left)
        ELSE
            # Pull from Right
            Result := Result + First(Right) # concatenate
            Right := Rest(Right)
        FI
    OD

    # One, or both, of lists are empty (Q: when could both be empty)
    IF Empty(Left) THEN
```

```
        Result := Result + Right
    ELSE
        Result := Result + Left
    FI

return Result
END
```

ABCD: Will the best, average and worst cases differ?

ABCD: In which divide-and-conquer phase does merge sort do most of its work?

Tree example:

`8, 14, 4, 11, 7, 1, 5, 9`

- Show recursion tree, derivation of $O(N \lg N)$ time complexity bound.
- **ABCD**: Is this in-situ?
  - What is the space complexity? Avg/worst = $O(\lg N)$.
  - Each call to MergeSort is $O(1)$ memory, but $O(\lg N)$ active frames on the call stack
- **ABCD**: Is merge sort stable?
- Use of insertion sort for lists smaller than k improves the hidden constant

Assumed List data type with the following operations:

- Empty(L) : Boolean – is the list empty / O(1)
- First(L) : Element – return the first element in the list / O(1)
- Rest(L) : List – return the list minus the first element / O(1)
- Length(L) : Natural – return the length of the list / O(N)
- L + E : List – Concatenate element E to the end of List / O(1) (pointer to tail)
- L + L : List – return the two lists concatenated / O(1)

PPQ: Write psuedo-code for an array-based Merge function. Has access to $O(N)$ extra memory.

Log Facts

Recall that

- $\log_b(x) = \log_a(x)/\log_a(b)$
- $\lg(x^a) = a \lg(x)$

PPQ

- Is $\log_{10}(N) \in O(\lg N)$? Why or why not?
- What big-O class is $\lg(N^3)$?
- What big-O class is $\lg(2^N)$?
- Challenge: what big-O class is $\lg(N!)$?

Question: Which is $O(log(N^2))$ closest to: $O(\log N), O(N), O(N^2)$?
Answer: $O(2 \log N) = O(\log N)$

1.2.2 QuickSort

Basic Idea: Chooses a pivot that is hopefully somewhere around median, and swaps values until it has all values less than on left, and values greater than on right. Then recursively sorts each side. **Does its work "going in."** Selecting the pivot: choose first value, choose last value, choose center (if array), or choose median of those three values. Choose random value.

Basic QuickSort Algorithm:

```
QuickSort(A,P,R)
   IF P < R
      Q := Partition(A,P,R)
      QuickSort(A,P,Q-1)
      QuickSort(A,Q+1,R)
   FI
END

Partition(A,P,R)
   I := P-1
   FOR J IN P .. R-1 DO
      IF A[J] <= A[R]
         I++
         Swap(A[I],A[J])
      FI
   OD
   return I+1
END
Swap(A[I+1],A[R])
```

The loop invariant

- For all k in $P \leq k \leq I$, $A[k] \leq A[R]$
- For all k in $I + 1 \leq k \leq J - 1$, $A[K] > A[R]$
- Elements between index J and $R - 1$, inclusive, are unsorted

Example from page 172 of Algorithms book:

```
2 8 7 1 3 5 6 4
```

- **ABCD**: Will the best case for QuickSort differ from the worst case?
- How much work is partitioning? $O(N)$
- *Draw trees for good and bad*
- How many partitions are neccessary in best case? $O(\log n)$ Worst case? $O(N)$
- Average case much closer to $O(N \log N)$: even 99-1 split has cost of $O(Nlog_{100/99}N) = O(N \lg N)$
- **ABCD**: Quicksort stable? No. Why?
- **ABCD**: In situ? No. Why? Call stack.

1.3 Other Sorts

1.3.1 LSD Radix Sort

Algorithm Description:

- Sorting N values whose keys have D base-K digits
- Proceed down list, dump into K FIFOs/queues (of size N) by least significant digit. Inserting each element takes $O(1)$ time.
- Pop the queues, starting from the first FIFO to the last, placing them back into the original array.
- Repeat with increasingly signficant digits until all digits have been used. Then it is sorted.

```
RadixSort(A)
  for I in 0 .. K-1 do
    B[I] := empty queue
  od

  for J in 1 .. D do
    for I in 0 .. N-1 do
      B[lsd(A[I],J)].enqueue(A[I])
    od
    M := 0
    for I in 0 .. K-1 do
      while !B[I].isEmpty() do
        A[M] := B[I].dequeue()
        M++
      od
    od
  od
end
```

Example: Phone extensions. List of extensions:

```
856, 800, 334, 206, 333, 896, 894
```

- Properties
  - **ABCD** Is Radix Sort stable? Yes.
- Space
  - **ABCD**: Is radix sort in-situ?. How much space? $O(K * N) = O(N)$
  - Can do in-place with binary trick, but no longer stable.
- Time
  - **ABCD**: Does Radix sort's best case differ from its worst case?
  - How many iterations? $O(D) = O(1)$
  - Cost per iteration: $O(N)$ (have to copy each element to a bin)
  - Overall runtime: $O(N)$!
  - But high hidden constant (especially for large keys)
  - With Base-2 (not in place) keys for SSN, memory requirement is approximately the same as merge sort, need roughly 30 iterations. When inserting 1,000,000 elements, the $logN$ term is only ≈ 20 - pretty comparable! And quicksort can do it using only $O(\log N)$ memory.