Data Structures and Algorithms — CSCI 230
Algorithm Analysis

Motivating Examples

Here are three standard algorithms — two searches and one sort — which should be analyzed to determine their computational efficiency.

Sequential Search:

```
//  Sequentially search an array of n elements to
//  determine if a given value is there.  If so, set loc to
//  be the first array location containing it and return true.
//  Otherwise, return false.

bool
SeqSearch(float arr[], int n, float value, int & loc)
{
  loc=0;
  while (loc<n && arr[loc] != value) {
    ++ loc;
  }
  return loc<n;
}
```

Insertion Sort:

```
//  Sort an array of n elements using insertion sort.

void
InsertSort(float arr[], int n)
{
  for (int i=1; i<n; i++) {
    float temp = arr[i];
    int j = i-1;
    while (j>=0 && arr[j] > temp) {
      arr[j+1] = arr[j];
      j -- ;
    }
    arr[j+1] = temp;
  }
}
```

Binary Search:

```
// Use binary search to determine if a given value is
// somewhere in an ordered array of n elements.  If so,
// set loc to be the first array location containing it
// and return true.  Otherwise, set loc to be the array
// location where it should be inserted and return false.
// The array ordering is assumed to be what's called
// "non-decreasing" order, which means that
//     arr[0] <= arr[1] <= ... <= arr[n-1]
//  or, more precisely,
//     for 0 <= i < n-1, arr[i] <= arr[i+1]

bool
BinSearch(float arr[], int n, float value, int & loc)
{
  int low = 0, high = n-1, mid;

  //  Before each iteration of the loop, the following
  //  conditions hold:
  //     0 <= low < high < n,
  //     for each j, 0 <= j < low, arr[j] < value
  //     for each j, high <= j < n, value <= arr[j]
  //
  while (low < high) {
    mid = (low + high) / 2;
    if (value <= arr[mid])
      high = mid;
    else
      low = mid+1;
  }

  loc = low;
  if (arr[loc] == value)
    return true;
  else {
    if (loc == n-1 && arr[n-1] < value) loc = n;
    return false;
  }
}
```

Exercise

How many operations, as a function of the array size n, are required by `SeqSearch`. If you finish this, try to answer the same question for `InsertSort`. What issues arose in your discussion?

Algorithm Analysis Rules

- The goal is to determine the worst-case or average-case time required by an algorithm, generally as a function of the "size" of the data. (Sometimes even the "best-case" is considered!)
- Assumptions: sequential execution, simple statements cost 1 unit of time, infinite memory, integers and reals represented in a fixed amount of memory.
- Generally, statements are counted to form a function $f(n)$, where n is the size of the data. Sometimes only special operations such as comparisons or exchanges are counted.
- We will discuss in class rules for counting when algorithms include:
  - Consecutive statements.
  - If-then-else.
  - Loops and nested loops.

Order Notation

Order notation is a mathematical formalism used to summarize the computation time required by an algorithm, simplifying the function derived to count the number of operations. On the positive side, this avoids quibbling over the number of operations (and cost) involved in simple algorithmic steps. On the negative side, this does result in some loss of precision.

- $T(n) = O(f(n))$ if there are constants c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$.
- $T(n) = \theta(f(n))$ if and only if $T(n) = O(f(n))$ and $f(n) = O(T(n))$.
- Limits may be used to simplify this. Suppose

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L.$$

  Then

  - If $L = 0$, $f(n) = O(g(n))$.
  - If $0 < L < \infty$, $f(n) = \theta(g(n))$.

  If L doesn't exist, nothing can be concluded. L'Hopital's rule may be used to analyze the limit. This requires converting f and g from functions of integers to functions of real numbers, which is usually trivial.

Order Notation — Rules for Manipulation

- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then
  - $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n))$, and
  - $T_1(n)T_2(n) = O(f(n) \cdot g(n))$

  The same rules hold when θ is used throughout.
- If $T(n)$ is a polynomial of degree n then $T(n) = O(n^k)$ (actually, $T(n) = \theta(n^k)$).
- $(\log n)^k = O(n)$ for any constant $k > 0$, but $(\log n)^k \neq \theta(n)$. Also, if $a > 0$ is any *fixed* constant, then $a = O((\log n)^k)$, but $a \neq \theta((\log n)^k)$.
- "O" estimates for summations are done in two ways.
  - Evaluate the summation using techniques from Chapter 1 and then determine an "O" (or "θ") estimate from the resulting function.
  - Place upper bounds on terms in the summation to simplify it and eliminate the summation.

Order Notation Exerices

1. Show that $5n^2 + 6n = O(n^2)$ using the original definition of "O" and then using limits.
2. For each pair of functions, $T(n)$ and $f(n)$, determine which of the following hold:
$$T(n) = O(f(n)) \qquad T(n) = \theta(f(n))$$
   Justify your answer. (Assume k, a and b are unspecified constants greater than 1 and $a > b$.)
   (a) $T(n) = n^2 \log n + 5n,\ f(n) = n^3$
   (b) $T(n) = \log(n^k),\ f(n) = (\log n)^k$
   (c) $T(n) = \log_a n,\ f(n) = \log_b n$
   (d) $T(n) = 2^n,\ f(n) = 2^{(2n)}$.
3. Give the best possible O estimate for $T(n)$,
   (a) $T(n) = (n^3 + 10n^2) \cdot (n^3 \log n + 20n^4)$
   Answer:
$$\begin{aligned} T(n) &= (n^3 + 10n^2) \cdot (n^3 \log n + 20n^4) \\ &= (n^6 \log n + 10n^5 \log n + 20n^7 + 200n^6) \\ &= O(n^7) \end{aligned}$$

(b) $T(n) = n3^n + n^{10} + 1500n^3 \log n$.
Answer:

$$\begin{aligned} T(n) &= n3^n + n^{10} + 1500n^3 \log n \\ &= O(n3^n) \end{aligned}$$

(c) $T(n) = \sum_{i=1}^{n} 5i(i-1)$
Answer:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} 5i(i-1) \\ &= 5\sum_{i=1}^{n} i^2 - 5\sum_{i=1}^{n} i \\ &= \frac{5}{6}n(n+1)(2n+1) - \frac{5}{2}n(n+1) \\ &= \frac{5}{3}n^3 + \text{lower order terms} \\ &= O(n^3) \end{aligned}$$

Or, even more simply:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} 5i(i-1) \\ &= \sum_{i=1}^{n} O(i^2) \\ &= O(n^3) \end{aligned}$$

4. Derive an "O" estimate for the worst-case of `InsertSort` based on the function we derived in class.

Answer: We actually got a variety of answers for the number of operations, depending on what we counted. If we count the two simple statements in the body of the outer loop and the two simple statements in the body of the inner loop, and assume the inner loop makes the maximum number of iterations, we get

$$T(n) = \sum_{i=1}^{n-1}(2 + \sum_{j=0}^{i-1} 2)$$

Try evaluating this yourself.

First evaluating the inner sum and then the outer one,

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1}(2 + 2i) \\ &= 2(n-1) + (n-1)n \\ &= n^2 + n - 2 \end{aligned}$$

What's this in O notation?

$$T(N) = O(n^2)$$

We could count other operations, such as incrementing i and j, and we would get different constants in the exact formula for $T(n)$, but constants don't matter in the O notation.

Algorithm Analysis Exercises

1. Count the number of operations in each of the following two code fragments as a function of n, the length of the array. Each should yield a summation. Then, analyze each summation to give the best possible "O" estimate for each fragment.

(a)
```
// assume arr is an array containing n integers
int k = 5;
for (int i=0; i<=n-k; i++) {
  sum = 0;
  for (int j=i; j<i+k; j++) {
    sum += arr[j];
  }
  cout << "Sum of elements " << i << " through "
       << i+k-1 << " is " << sum << "\n";
}
```

Answer: Counting each assignment and output statement as one operation, we have

$$T(n) = 1 + \sum_{i=1}^{n-5} (2 + \sum_{j=i}^{i+5} 1)$$

Try evaluating this yourself.

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-5} (2 + \sum_{j=i}^{i+5} 1) \\ &= 1 + \sum_{i=1}^{n-5} (2 + 5) \\ &= 1 + 7 \sum_{i=1}^{n-5} 1 \\ &= 1 + 7(n-6) \\ &= 7n - 5 \\ &= O(n) \end{aligned}$$

Note: Often when there are nested loops, the number of operations is quadratic. In this case, though, the inner loop only iterates a constant number of times (5), so the total time is linear, not quadratic.

(b)

```
//  assume  arr  is an array containing n integers
int k = n/2;
for (int i=0; i<=n-k; i++) {
  sum = 0;
  for (int j=i; j<i+k; j++) {
    sum += arr[j];
  }
  cout << "Sum of elements " << i << " through "
       << i+k-1 << " is " << sum << "\n";
}
```

Answer: Again counting each assignment and output statement as one operation, we have

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=1}^{n/2} (2 + \sum_{j=i}^{i+n/2} 1) \\
&= 1 + \sum_{i=1}^{n/2} (2 + n/2) \\
&= 1 + 2\sum_{i=1}^{n/2} 1 + (n/2)\sum_{i=1}^{n/2} 1 \\
&= 1 + n + n^2/4 \\
&= O(n^2)
\end{aligned}
$$

2. Rewrite the second code fragment to make it as efficient as possible. Start by thinking carefully about what it actually does! What is the complexity of your new code fragment?

   Hint: It computes a series of sums, but they are closely related to each other, so it's not necessary to compute each sum from scratch.

3. In the analysis of `InsertSort` we assumed that the worst-case would occur at all times. What must be the state of the array for the absolute maximum number of operations to occur? Repeat the analysis of `InsertSort` to derive average case and best case estimates? What state of the array causes the best-case to occur?

More advanced analysis:

- In analyzing recursive algorithms, usually a recursive equation (also called a "recurrence relation") is derived modeling the number of steps required, which is then solved to yield a non-recursive formula. We will examine this using the factorial function and then, later, a solution to the max subsequence sum problem.
- Logarithmic times in analysis usually arise from algorithms, such as Binary Search and Merge Sort, that break an array or data set in half and then consider one or both halves separately.

Max Subsequence Sum

- The simplest solution: algorithm 1 and its analysis.
- An easy refinement: algorithm 2 and its analysis.
- Divide-and-conquer: algorithm 3 and its analysis.
- A simple, fast solution: algorithm 4 and its analysis.

- We will confirm the analysis results experimentally.

Exercises:

1. Derive a recursive equation to analyze MergeSortand then solve this equation. Assume the array is of size $n = 2^k$ for integer $k \geq 0$. For completeness, here is the algorithm (combining material from Ch 1 and the Ch 1 review):

```
template <class T>
void MergeSort(T * pts, int n)
{
  MergeSort(pts, 0, n-1);
}

templage <class T>
void MergeSort(T * pts, int low, int high)
{
  if (low == high) return;

  int mid = (low + high) / 2;
  MergeSort(T, low, mid);
  MergeSort(T, mid+1, high);

  //  At this point the lower and upper halves
  //  of "pts" are sorted. All that remains is
  //  to merge them into a single sorted list.
  T* temp = new T[high-low+1];
     // scratch array for merging
  int i=low, j=mid+1, loc=0;

  //  while neither the left nor the right half is exhausted,
  //  take the next smallest value into the temp array
  while (i<=mid && j<=high) {
    if (pts[i] < pts[j]) temp[loc++] = pts[i++];
    else                 temp[loc++] = pts[j++];
  }

  //  copy the remaining values --- only one of
  //  these will iterate
  for (; i<=mid; i++, loc++) temp[loc] = pts[i];
  for (; j<=high; j++, loc++) temp[loc] = pts[j];

  //  copy back from the temp array
  for (loc=0, i=low; i<=high; loc++, i++) pts[i]=temp[loc];
  delete [] temp;
```

```
}
```

2. Find an efficient algorithm (along with a running time analysis) to find the *minimum* subsequence sum.

Review Problems

Here are a few review problems which have appeared on homeworks or tests in previous semesters. Practice writing solutions carefully and then compare to solutions provided on-line. If you can solve these problems and the problems we worked on in class then you are ready for the chapter quiz!

1. Show that $\sum_{i=1}^{n} 2i^3 = O(n^4)$.

2. For each of the following, find $f(n)$ such that $t(n) = O(f(n))$. Make $f(n)$ as small and simple as possible, i.e. don't write $t(n) = O(n^4)$ when $t(n) = O(n^3)$. Justify your answers.

   (a) $t(n) = 13n^2 + 2^n$

   (b) $t(n) = 5(n + 3\log n)(n \log n + 13)\log n + 13n^2$

   (c) $t(n) = \sum_{i=3}^{n}\sum_{j=i}^{n} i(n-j)$

3. Exercise 2.6a from the text. Try to derive summations first. Note program fragment (6) is quite difficult.

4. Derive a summation to count, as a function of n, the number of times `Hello` is output by each of the following code fragments. Obtain an accurate "O" estimate from the summation.

   (a)
   ```
   for (i=1; i<=n; i++)
     for (j=1; j<=i; j++)
       for (k=j+1; k<=n; k++)
         cout << "Hello\n";
   ```

   (b) For this part, assume $n = 2^k$ and assume the notation `2^i` means 2^i.
   ```
   for (i=0; i<=k; i++)
     for (j=2^i+1; j<=n; j++)
       cout << "Hello\n";
   ```

5. Exercise 2.11 of the text.

6. Write an algorithm that takes an unsorted list (provided as an array) of n floating point values and returns the smallest difference between any two values in the list. For example, for the list

```
2.9, 3.5, 1.1, 6.1, 2.3, 1.8, 8.7, 3.0, 2.4,
```

the algorithm should return 0.1, which is the difference between 3.0 and 2.9. Make your algorithm as efficient as you can and give the worst-case running time of your algorithm as a function of n, briefly justifying your answer. Hint: you may change or reorganize the contents of the array.