Program Analysis and Elementary Data Structures

Lecture 3 & 4

Problem of the Day

For each of the following pairs of functions $f(n)$ and $g(n)$, state whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$, or none of the above.

1. $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$
2. $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$
3. $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$

Big Oh Multiplication by Constant

Multiplication by a constant does not change the asymptotics:

$$O(c \cdot f(n)) \rightarrow O(f(n))$$

$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$

$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$

The "old constant" C from the Big Oh becomes $c \cdot C$.

Big Oh Multiplication by Function

But when both functions in a product are increasing, both are important:

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$

$$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$$

$$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$$

This is why the running time of two nested loops is $O(n^2)$.

Reasoning About Efficiency

Grossly reasoning about the running time of an algorithm is usually easy given a precise-enough written description of the algorithm.

When you *really* understand an algorithm, this analysis can be done in your head. However, recognize there is always implicitly a written algorithm/program we are reasoning about.

Selection Sort

```
selection_sort(int s[], int n)
{
        int i,j;
        int min;

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

Worst Case Analysis

The outer loop goes around n times.
The inner loop goes around at most n times for each iteration of the outer loop
Thus selection sort takes at most $n \times n \rightarrow O(n^2)$ time in the worst case.
In fact, it is $\Theta(n^2)$, because at least $n/2$ times it scans through at least $n/2$ elements, for a total of at least $n^2/4$ steps.

More Careful Analysis

An exact count of the number of times the *if* statement is executed is given by:

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

$S(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = n(n-1)/2$

Thus the worst case running time is $\Theta(n^2)$.

Insertion Sort

```
insertion_sort(item s[], int n)
{
        int i,j; /* counters */

        for (i=1; i < n; i++) {
                j=i;
                while ((j > 0) && (s[j] < s[j-1])) {
                        swap(&s[j],&s[j-1]);
                        j = j-1;
                }
        }
}
```

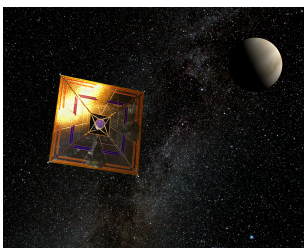
This involves a while loop, instead of just for loops, so the analysis is less mechanical.

But n calls to an inner loop which takes at most n steps on each call is $O(n^2)$.

The reverse-sorted permutation proves that the worst-case complexity for insertion sort is $\Theta(n^2)$.

$$(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$$

Solar Sails vs. Rockets



The bad-ass rocket hits a high speed before it runs out of fuel, then coasts at constant speed v_r.

The solar sail slowly accelerates from the force of radiation/solar wind hitting it, but its speed of $v_s = at$ must eventually exceed the bad-ass rocket.

This is asymptotic dominance in action.

Logarithms

It is important to understand deep in your bones what logarithms are and where they come from.

A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Logarithms reflect how many times we can double something until we get to n, or halve something until we get to 1.

Binary Search

In binary search we throw away half the possible number of keys after each comparison. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!

How many time can we halve n before getting to 1?

Answer: $\lceil \lg n \rceil$.

Logarithms and Trees

How tall a binary tree do we need until we have n leaves?
The number of potential leaves doubles with each level.
How many times can we double 1 until we get to n?
Answer: $\lceil \lg n \rceil$.

Logarithms and Bits

How many bits do you need to represent the numbers from 0 to $2^i - 1$?

Each bit you add doubles the possible number of bit patterns, so the number of bits equals $\lg(2^i) = i$.

Logarithms and Multiplication

Recall that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

This is how people used to multiply before calculators, and remains useful for analysis.
What if $x = a$?

Federal Sentencing Guidelines

2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.
(a) Base offense Level: 6
(b) Specific offense Characteristics

(1) If the loss exceeded $2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

Make the Crime Worth the Time

The increase in punishment level grows *logarithmically* in the amount of money stolen.

Thus it pays to commit one big crime rather than many small crimes totalling the same amount.

Elementary Data Structures

"Mankind's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps are the "off-the-shelf" components we build our algorithm from.

There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementation of these operations.

Data Abstraction

That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.

- *Push(x,s)* – Insert item x at the top of stack s.
- *Pop(s)* – Return (and remove) the top item of stack s.

That there are different implementations of the same abstract operations enables us to optimize performance in difference circumstances.

Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either *contiguous* or *linked* depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

Arrays

An array is a structure of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.
Advantages of contiguously-allocated arrays include:

- Constant-time access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.
- Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays

Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.

Compensating by allocating extremely large arrays can waste a lot of space.

With *dynamic arrays* we start with an array of size 1, and double its size from m to $2m$ each time we run out of space.

How many times will we double for n elements? Only $\lceil \log_2 n \rceil$.

How Much Total Work?

The apparent waste in this procedure involves the recopying of the old contents on each expansion.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is:

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Thus each of the n elements move an average of only twice, and the total work of managing the dynamic array is the same $O(n)$ as a simple array.

Geometric series convergence is the free lunch of algorithm analysis.

Pointers and Linked Structures

Pointers represent the address of a location in memory.
A cell-phone number can be thought of as a pointer to its owner as they move about the planet.
In C, `*p` denotes the item pointed to by `p`, and `&x` denotes the address (i.e. pointer) of a particular variable `x`.
A special `NULL` pointer value is used to denote structure-terminating or unassigned pointers.

Linked List Structures

```
typedef struct list {
        item_type item;
        struct list *next;
} list;
```

Searching a List

Searching in a linked list can be done iteratively or recursively.

```
list *search_list(list *l, item_type x)
{
      if (l == NULL) return(NULL);

      if (l->item == x)
            return(l);
      else
            return( search_list(l->next, x) );
}
```

Insertion into a List

Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head.

```
void insert_list(list **l, item_type x)
{
      list *p;

      p = malloc( sizeof(list) );
      p->item = x;
      p->next = *l;
      *l = p;
}
```

Note the `**l`, since the head element of the list changes.

Deleting from a List

```
delete_list(list **l, item_type x)
{
      list *p; (* item pointer *)
      list *last = NULL; (* predecessor pointer *)

      p = *l;
      while (p->item != x) { (* find item to delete *)
            last = p;
            p = p->next;
      }

      if (last == NULL) (* splice out of the list *)
            *l = p->next;
      else
            last->next = p->next;

      free(p); (* return memory used by the node *)
}
```

Advantages of Linked Lists

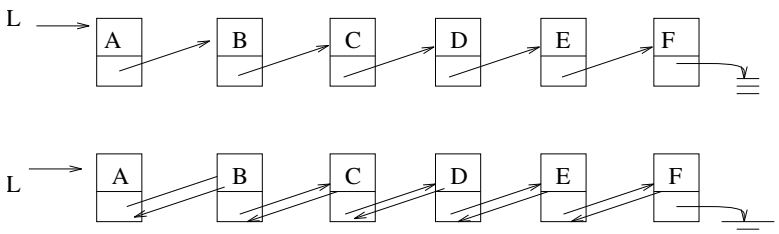
The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

Singly or Doubly Linked Lists

We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.



Since the extra big-Oh costs of doubly-linked lists is zero, we will usually assume they are so maintained, although it might not always be necessary.

Singly linked to doubly-linked list is as a conga line is to a can-can line.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.
A *stack* supports last-in, first-out operations:

- *Push(x,s)* – Insert item x at the top of stack s.
- *Pop(s)* – Return (and remove) the top item of stack s.

A *queue* supports first-in, first-out operations:

- *Enqueue(x,q)* – Insert item x at the back of queue q.
- *Dequeue(q)* – Return (and remove) the front item from queue q.

Stack/Queue Implementations

- Stacks are more easily represented as an array, with push/pop incrementing/decrementing a counter.
- Queues are more easily represented as a linked list, with enqueue/dequeue operating on opposite ends of the list.

All operations can be done in $O(1)$ time for both structures, with both arrays and lists.

Why Stacks and Queues?

Both are appropriate for a container class where order doesn't matter, but sometime it does matter.

Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

The entire difference between depth-first search (DFS) and breadth-first search (BFS) is whether a stack or a queue holds the vertices/items to be processed.

Dictonary / Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- *Search(S, k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x]$ = k, or nil if no such element belongs to S.
- *Insert(S,x)* – A modifying operation that augments the set S with the element x.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- Logical *Predecessor(S,x), Successor(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next smallest (largest) element in S, or NIL if x is the maximum (minimum) element.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

There is an inherent tradeoff between these operations. We will see that no single implementation will achieve the best time bound for all operations.

Array Based Sets: Unsorted Arrays

- *Search(S,k)* – sequential search, $O(n)$
- *Insert(S,x)* – place in first empty spot, $O(1)$
- *Delete(S,x)* – copy nth item to the xth spot, $O(1)$
- *Min(S,x), Max(S,x)* – sequential search, $O(n)$
- *Successor(S,x), Predecessor(S,x)* – sequential search, $O(n)$

Array Based Sets: Sorted Arrays

- *Search(S,k)* – binary search, $O(\lg n)$
- *Insert(S,x)* – search, then move to make space, $O(n)$
- *Delete(S,x)* – move to fill up the hole, $O(n)$
- *Min(S,x), Max(S,x)* – first or last element, $O(1)$
- *Successor(S,x), Predecessor(S,x)* – Add or subtract 1 from pointer, $O(1)$