Algorithm Design Manual Notes

Zachary William Grimm
Notes for ADM by Skiena
zwgrimm@gmail.com

February 20, 2019

Contents

1 Introduction To Algorithm Design

the algorithmic *problem* known as *sorting* is defined as follows:
Problem: Sorting
Input: A sequence of n keys $a_1, ..., a_n$.
Output: The permutation (reordering) of the input sequence such that $a_1^{'} \leq a_2^{'} \leq ... \leq a_{n-1}^{'} \leq a_n^{'}$

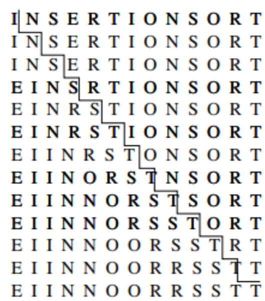

Figure 1: Animation of insertion sort in action (time flows down)

```
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;
        }
    }
}
```

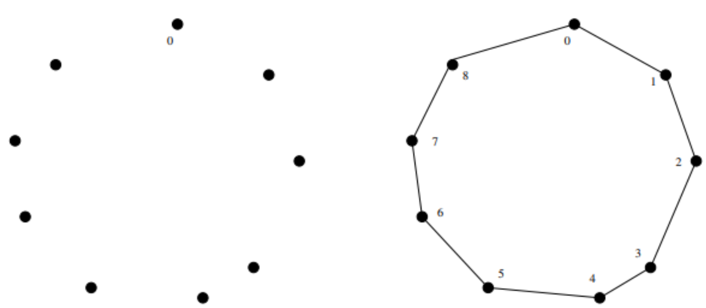

Figure 2: A good instance for the nearest neighbor heuristic

1.1 Robot Tour Optimization

Problem: Robot Tour Optimization
Input: A set S of N points in the plane.
Output: What is the shortest cycle tour that visits each point in the set S?

1.2 Selecting the Right Jobs

Problem: Movie Scheduling Problem
Input: A set I of n intervals on the line.
Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I?

Take-Home Lesson: Reasonable-looking algorithms can easil beincorrect. Algorithm correctnes is a property that must be carefully demonstrated.

1.3 Reasoning about Correctness

1.3.1 Expressing Algorithms

Take-Home Lesson: The heart of any algorithm is an *idea*. If your idea is not clearly revealed when you express an algorithm, then you are using too low-level a notation to describe it.

1.3.2 Problems and Properties

Take-Home Lesson: An important and honorable tecnique in algorithm design is to narrow the set of llowable instances until there *is* a correct and efficient algorithm. For example, we can restrit a graph problem from general graphs down to trees, or a geometric problem from two dimensions down to one.

1.3.3 Demonstrating Incorrectness

- *Verifiability*
- *Simplicity*
- *Think small*
- *Think exhaustively*
- *Hunt for the weakness*
- *Seek extremes*

Take-Home Lesson: Searching for conterexamples is the best way to disprove the correctness of a heuristic.

1.3.4 Induction and Recursion

Take-Home Lesson: Mathematical induction is usually the right wy to verify the correctness of a recursive or incremental insertion algorithm.

1.3.5 Summations

- *Arithmetic progressions*
- *Geometric series*

1.4 Modeling The Problem

1.4.1 Combinatorial Objects

- *Permutations*
- *Subsets*
- *Trees*
- *Graphs*
- *Points*
- *Polygons*
- *Strings*

Take-Home Lesson: Modeling your application in terms of well-defined structures and algorithms is the most important single step towards a solution.

1.4.2 Recursive Objects

- *Permutations*
- *Subsets*
- *Trees*
- *Graphs*
- *Points*
- *Polygons*
- *Strings*

1.5 About the War Stories

1.6 War Story: Psychic Modeling

2 Algorithm Analysis

Our two most important tools are

1. *The RAM model of computation*
2. *The asymptotic analysis of worst-case complexity*

2.1 The RAM Model of Computation

Take-Home Lesson: Algorithms can be understood and studied in a language and machine-independant manner..

2.1.1 Best, Worst, and Average-Case Complexity

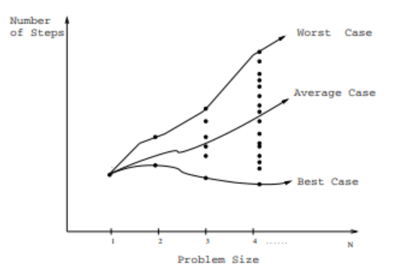

Figure 3: Best, Worst and average case complexity

2.2 The Big Oh Notation

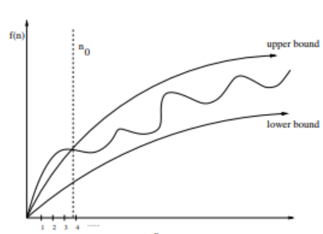

Figure 4: Upper and lower bounds valid for n>n0 smooth out the behavior of complex functions

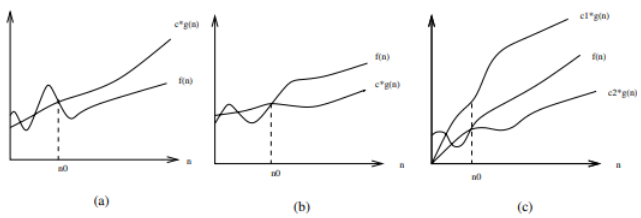

Figure 5: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

Stop and Think: Hip to the Squares

Problem: Is $(x + y)^2 = O(x^2 + y^2)$

Stop and Think: Back to the Definition

Problem: Is $2^{n+1} = \Theta(2^n)$?

2.3 Growth Rates and Dominance Relations

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 6: Growth rates of common functions measured in nanoseconds

2.3.1 Dominance Relations

Take-Home Lesson: Although esoteric functions arise in advanced algorithm analysis, a small variety of time complexities suffice and account for most algorithms that are widely used in practice.

2.4 Working with the Big Oh

2.4.1 Adding Functions

$$O(f(n)) + O(g(n)) \longrightarrow O(max(f(n), g(n))$$

$$\Omega(f(n)) + \Omega(g(n)) \longrightarrow \Omega(max(f(n), g(n))$$

$$\Theta(f(n)) + \Theta(g(n)) \longrightarrow \Theta(max(f(n), g(n))$$

2.4.2 Multiplying Functions

$$O(c * f(n))) \longrightarrow O(f(n))$$

$$\Omega(c * f(n))) \longrightarrow \Omega(f(n))$$

$$\Theta(c * f(n))) \longrightarrow \Theta(f(n))$$

$$O(f(n)) * O(g(n)) \longrightarrow O(f(n) * g(n))$$

$$\Omega(f(n)) * \Omega(g(n)) \longrightarrow \Omega(f(n) * g(n))$$

$$\Theta(f(n)) * \Theta(g(n)) \longrightarrow \Theta(f(n) * g(n))$$

Stop and Think: Hip to the SquaresTransitive Experience

Show that Big Oh relationships are transitive. That is, if $f(n) = O(g(n))$ *and* $g(n) = O(h(n))$*, then* $f(n) = O(h(n))$

2.5 Reasoning About Efficiency

2.5.1 Selection Sort

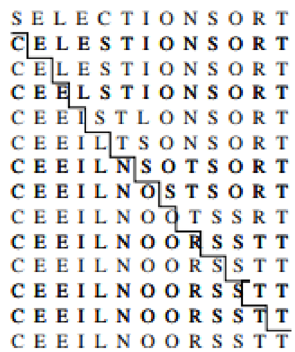

Figure 7: Animation of selection sort in action.

```
selection_sort(int s[], int n)
{
    int i, j:          /* counters */
    int min;           /* index of minimum */

    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]);
    }
}
```

2.5.2 Insertion Sort

```
    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;
        }
    }
```

2.5.3 String Pattern Matching

Problem: Substring Pattern Matching
Input: A text string t and a pattern string p
Output: Does t contain the pattern p as a substring, and if so where?

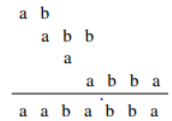

Figure 8: Searching for the substring abba *in the text* aababba

```
int findmatch(char *p, char*t)
{
    int i, j;          /* counters */
    int m, n;          /* string lengths */

    m = strlen(p);
    n = strlen(t);

    for (i=0; i<(n-m); i=i+1) {
          j = 0;
          while((j<m) && (t[i+j] == p[j]))
              j = j + 1;
          if(j ==m) return (i);
    }

    return(-1)
}
```

2.5.4 Matrix Multiplication

Problem: Matrix Multiplication
Input: Two matrices, A (of dimesion $x \times y$) and B (dimension $y \times z$).
Output: An $x \times z$ matrix C where C[i][j] is the dot product of the ith row of A and the jth column of B.

2.6 Logarithms and Their Applications

$$b^x = y \leftrightarrow x = log_b y$$
$$b^{log_b y} = y$$

2.6.1 Logarithms and Binary Search

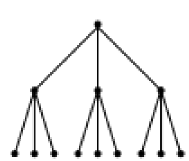

Figure 9: A height h *tree with* d *children per node as* d^h *leaves. Here* $h = 2$ *and* $d = 3$

2.6.2 Logarithms Trees

2.6.3 Logarithms and Bits

2.6.4 Logarithms and Multiplication

$$log_a(xy) = log_a(x) + log_a(y)$$
$$log_a n^b = b \cdot log_a n$$
$$a^b = e^{(ln(a^b))} = e^{(b(ln(a)))}$$

2.6.5 Fast Exponentiation

2.6.6 Logarithms and Summations

Harmonic Numbers:

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} \sim ln(n)$$

2.6.7 Logarithms and Criminal Justice

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

Figure 10: The Federal Sentencing Guidelines for fraud

Take-Home Lesson: Logarithms arise whenever things are repeatedly halved or doubled

2.7 Properties of Logarithms

$$log_a b = \frac{log_c b}{log_c a}$$

Stop and Think: Importnce of an Even Split

How many queries does binary search take on the million-name Manhattan phone boo if each split was 1/3 to 2/3 instead of 1/2 to 1/2?

2.8 War Story: Mystery of the Pyramids

2.9 Advanced Aanalysis (*)

2.10 Esoteric Functions

2.11 Limits and Dominance Relations

Take-Home Lesson: By interleaving the functions here with those of Section 2.3.1 we see where everything fits into the dominnce pecking order:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \cdot log(n) \gg n \gg \sqrt{n}$$

$$\gg log^2 n \gg log(n) \gg \frac{log(n)}{log(log(n))} \gg log(log(n)) \gg \alpha(n) \gg 1$$

3 Data Structures

3 Fundamental abstract data types

1. *Containers*
2. *Dictionaries*
3. *Lists*

3.1 Contigous vs. Linked Data Structures

1. *Continuously allocated structures (arrays)*
2. *Linked data structures (pointers, lists, trees...)*

3.1.1 Arrays

$$M = \sum_{i=1}^{lg(n)} i * \frac{n}{2^i} = n * \sum_{i=1}^{lg(n)} \frac{i}{2^i} \leq n * \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

3.1.2 Pointers and Linked Structures

figure here*** (Linked Lst example showing data and pointer fields)

```
typedef struct list {
  item_type item;          /*data item*/
  struct list *next;       /*point to successor*/
} list;
```

Searching a List

```
list *search_list(list *1, item_type x)
{
  if(1 == NULL) return(NULL);

  if(1->item == x)
    return(1);
  else
    return(search_list(1->next, x) );
}
```

Insertion into a List

```
void insert_list(list **1, item_type x)
{
    list *p                  /* temporary pointer*/

    p = malloc(sizeof(list));
    p->item = x;
    p->next = *1;
    *1 = p;
}
```

Deletion From a List

```
list *predecessor_list(list *1, item_type x)
```

```
{
    if((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought on null list.\n");
        return(NULL);
    }

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

delete_list(list **l, item_type x)
{
    list *p;                      /*item pointer*/
    list *pred                    /*predecessor pointer*/
    list *search_list(), *predecessor_list();

    p = search_list(*l,x);
    if(p != NULL) {
        pred = predecessor_list(*l, x);
        if(pred == NULL)          /*splice out list*/
            *l = p->next;
        else
            pred->next = p->next;
        free(p);                  /*free memory used by node*/
    }
}
```

3.1.3 Comparison

Take-Home Lesson: Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

3.2 Stacks and Queues

1. ***Stacks:*** *LIFO*
2. ***Queues:*** *FIFO*

3.3 Dictionaries

Primary Operations:

1. Search

2. Insert
3. Delete

Stop and Think: Comparing Dictionary Implementations (I)

Problem: What are the asymptotic worst-case running times for each of the seven fundamental dictionary operations (search, delete, successor, predecessor, minimum, maximum) when the data structure is implemented as:

1. *An unsorted array*
2. *A sorted Array*

Take-Home Lesson: Data structure design must balance all the different operations it supports. The fastest data structure to support both operations A and B may well not be the fastest structure to supprt either operation A or B.

Stop and Think: Comparing Dictionary Implementations (II)

Problem: What are the asymptotic worst-case running times for each of the seven fundamental dictionary operations (search, delete, successor, predecessor, minimum, maximum) when the data structure is implemented as:

1. *A singly-linked unsorted list*
2. *A doubly-linked unsorted list*
3. *A singly-linked sorted list*
4. *A doubly-linked sorted list*

3.4 Binary Search Trees

For any binary tree on n nodes and any set of n keys, there is exactly one labeling that makes it a binary search tree

3.4.1 Implementing Binary Search Trees

```
typedef struct tree {
    item_type item;          /*data item*/
    struct tree *parent      /*pointer to parent*/
    struct tree *left        /*pointer to left child*/
```

```
    struct tree *right     /*pointer to right child*/
} tree;
```

Basic binary search tree operations:

1. *search*
2. *traversal*
3. *insertion*
4. *deletion*

Searching in a Tree

```
tree *search_tree(tree *1, item_type x)
{
    if(1 == NULL) return(NULL);

    if(1->item == x) return(1);

    if(x < 1->item)
        return(search_tree(1->left, x));
    else
        return(search_tree(1->right, x));
}
```

Finding Minimum and Maximum Elements in a Tree

```
tree *find_minimum(tree *t)
{
    tree *min;         /*pointer to minimum*/

    if(t == NULL) return(NULL);

    min = t;
    while(min->left !=NULL)
        min = min->left;
    return(min)
}
```

Traversal in a Tree

```
void traverse_tree(tree *1)
```

```
{
    if(l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

Insertion in a Tree

```
insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                          /*temporary pointer*/

    if(*l == NULL) {
        p = malloc(sizeof(tree));     /*allocate new node*/
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;                       /*link into parents record*/
        return;
    }

    if(x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```