

CSE373 Homework 1 Solutions & Hints

1 Program Execution Time (11pts)

Program	Theoretical Running Time
A prog1 = 2-to-n.c	$\Theta(2^n)$
B prog2 = binary-search.c	$O(\log n)$
C prog3 = fac_time.c	$\Theta(1)$
D prog4 = insert-sort.c	$\Theta(n^2)$
E prog5 = merge-sort.c	$\Theta(\log n)$
F prog6 = seq-search.c	$\Theta(n)$
G prog7 = triple.c	$\Theta(n^3)$

Several things:

a) binary-search VS. seq-search

The latter one is faster in our experiment because there's a 'sleep(1);' statement in the former one, so the constant factor of the former one is larger.

b) insert-sort VS. merge-sort

When n is small, the former one's faster; while when n is large, the latter one's faster. This is because merge-sort uses recursive call. (By the way, it's not good to use 'malloc' without 'free' ^_^)

When calculating the running time, I also consider the time for comparisons.

c) fac_time

It's actually $\Theta(1)$ because $n \leq 16, n! \leq 16$ after we have executed the 24th line. (Maybe for preventing overflow) But, the running time is very long, as you can easily see.

d) In these problems, we can assume random() is $O(1)$, and that calculating the multiplication of two integers is also $O(1)$. But please be careful in other situations. And when the problem concerns large space, you may need to consider the transfer of data between hard disk and memory.

You can get full credits if you have actually run each program and done some analyses.

2 Problem 1-17, 1-19 (11pts)

1-17:

We assume that there are about 40 lines per page and about 10 words per line. Multiply by 500 pages and we get about 200,000 words.

1-19:

The population of US is approximately 300 million and there are approximately 5,0000 people per city or town. Therefore the answer is $300 * 10^6 / 5,0000 = 6,000$

You can get full credits if your calculation is reasonable.

3 Problem 2-7, 2-8 (11pts)

2-7:

- (a) True
- (b) False

Reasons omitted.

2-8:

Let A: $f(n) = O(g(n))$; B: $f(n) = \Omega(g(n))$; C: $f(n) = \Theta(g(n))$

(a)~(h): CBBBBBCBA

If two functions satisfy C, they also satisfy A and B. So when you say that the answer is B, you should give reasons why they do not satisfy A.

Common mistakes: h.

4 Problem 2-19 (11pts)

$$\left(\frac{1}{3}\right)^n \ll 6 \ll \log \log n \ll \log n \approx \ln n \ll (\log n)^2 \ll n^{\frac{1}{3}} + \log n \ll \sqrt{n} \ll \frac{n}{\log n} \ll n \ll n \log n \\ \ll n^2 \approx n^2 + \log n \ll n^3 \ll n - n^3 + 7n^5 \ll \left(\frac{3}{2}\right)^n \ll 2^n \ll n!$$

Common mistakes: $\left(\frac{1}{3}\right)^n \ll 6$

5 Problem 2-21, 2-22, 2-23, 2-24 (11pts)

2-21:

(a)~(g): TFTFTTF

2-22:

Let A: $f(n) = O(g(n))$; B: $f(n) = \Omega(g(n))$; C: $f(n) = \Theta(g(n))$; D: none of the above.

(a)~(c): BAB

2-23:

(a)~(e): YYYNY

(b) If an algorithm is $O(n)$, it is also $O(n^2)$

(c) If the worst-case time of an algorithm is $f(n)$, then $T \leq f(n)$ for all inputs, assuming T to be the running time of the algorithm.

2-24:

(a)~(d): NYYY

Common mistakes: b.

6 Problem 3-2 (11pts)

I'll write in a C-style code.

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

1 void reverse_list(list **l){
2     list *p = NULL;
3     list *q = *l;
4     list *r;
5     while(q!=NULL){
6         r = q -> next;
7         q -> next = p;
8         p = q;
9         q = r;
10    }
11    *l = q;
12 }
```

Criteria:

Wrong when the list is empty: -1

Fail in updating the head of the list: -1

Here's a sample program for you to find the bugs in your code (although its style is bad, you can actually run it in Microsoft Visual C++ 2005 ^_^):

```
#include <iostream>
using namespace std;
typedef struct list{
    int item;
    struct list *next;
}list;
void print_list(list *l){
    if (l){cout << l -> item; print_list(l -> next);}
}
void free_list(list **l){
    list *p = *l; list *q;
    while(p){q = p; p = p -> next; delete(q);}
    *l = 0;
}
void reverse_list(list **l){ //test your own code here
```

```

list *p = 0; list *q = *l; list *r;
while(q){
    r = q -> next; q -> next = p; p = q; q = r;
}
*l = p;
}
int main()
{
    list *temp; list *temp1;
    list *p1 = 0;
    list *p2 = new list(); p2 -> item = 1; p2 -> next = 0;
    list *p3 = new list(); p3 -> item = 1; temp = new list();
    p3 -> next = temp; temp -> item = 2; temp -> next = 0;
    list *p4 = new list(); p4 -> item = 1; temp = new list(); p4 -> next = temp; temp -> item = 2;
    temp1 = new list(); temp -> next = temp1; temp1 -> item = 3; temp1 -> next = 0;
    reverse_list(&p1); print_list(p1); cout << endl; //change reverse_list(&p1) if necessary
    reverse_list(&p2); print_list(p2); cout << endl;
    reverse_list(&p3); print_list(p3); cout << endl;
    reverse_list(&p4); print_list(p4); cout << endl;
    free_list(&p1); free_list(&p2); free_list(&p3); free_list(&p4); return 0;
}

```

Output:

1
21
321

7 Problem 3-4 (11pts)

Use an arraylist directly: $D[1..n]$

$D[i] == 0$ iff i is not in the dictionary.

$D[i] == 1$ iff i is in the dictionary.

Initialization: $D[1] \sim D[n] = 0$

Insertion: $D[i] = 1$, while i is the number we want to insert

Deletion: $D[i] = 0$, while i is the number we want to delete

Search: return $D[i]$

8 Problem 3-10 (11pts)

Using any kind of balanced binary search tree (e.g. AVL or RBtree) would satisfy the requirement, since the data structure satisfies:

Insertion, Deletion, Search are all $O(\log n)$.

We use a balanced binary search tree to store the bins which have objects in them. The keys of the bins are their free spaces.

- (a) We try to put the objects one by one. We search in the tree for the bins which has the smallest amount of extra room and the extra room is sufficient to hold the object. If such bin exists, we put objects in it and update our tree; if not, we use a new bin and insert it to the tree.
- (b) We try to put the objects one by one. We search in the tree for the bins which has the largest amount of extra room. If such bin exists and is sufficient to hold the object, we put objects in it and update our tree; otherwise, we use a new bin and insert it to the tree.

Notes:

- a) A Binary Heap does not operate well in searching. So we cannot use a heap in (a); but we can use it in (b) because we only need EXTRACT_MIN operation in (b).
- b) This is a NP-Complete problem. So neither best-fit nor worst-fit can always give you the optimal solution.

Criteria:

Do not use 'balanced' tree: -1 (a simple binary tree operates $\Theta(n^2)$ in the worst case)

9 Problem 3-11 (12pts)

In this problem, we ignore the time for insertion, deletion or initialization (online algorithm).

(a) Use a n -by- n matrix directly. The $i \times j$ cell denotes the smallest value in x_i, \dots, x_j

(b) Let $\min(i, j)$ denote the smallest value in x_i, \dots, x_j . We build a binary tree like this:

The root of the tree is $\min(1, n)$.

The left child of the root is $\min\left(1, \left\lfloor \frac{n}{2} \right\rfloor\right)$. The right child of the root is $\min\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, n\right)$

We do this recursively, that if a node in the tree denotes $\min(i, j)$, then its left child is

$\min\left(i, \left\lfloor \frac{i+j}{2} \right\rfloor\right)$; its right child is $\min\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1, j\right)$

The leaves are x_1, \dots, x_n .

Then we can write an algorithm:

```
int Search(root, i, j){
    if (root -> down == i && root -> up == j) return root -> value;
    boundary = (root -> down + root -> up) / 2;
    if (i > boundary) return search(root -> rightchild, i, j);
    if (j <= boundary) return search(root -> leftchild, i, j);
    return min(search(root -> leftchild, i, boundary),
              search(root -> rightchild, boundary + 1, j));
}
```

But when we try to analyze the running time of the algorithm, we have:

$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$, hence $T(n) = O(n)$. How to fix this problem? This question is left to you.

Criteria:

If your algorithm runs $O(\log n)$ in average cases, you get full points.

If your algorithm runs $O(\log n)$ in the worse case, you get +2 bonus. (But your score of Hw1 is bounded by 100)

Notes:

The data structure we used is a segment tree.

This problem is called RMQ (Range Minimum Query). Considering online algorithms, we have:

	Space/Init Time	Query Time
Brute Force	$O(n^2)$	$O(1)$
Segment Tree	$O(n)$	$O(\log n)$
Sparse Table	$O(n \log n)$	$O(1)$

For more information, you can visit:

<http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor>