CS 211	Computers and Programming Fall 1999	Handout 6

Sample Solution to Assignment 4

October 28, 1999

The complete program code is available for download on the course home page.

Problem 1: Perfect Packing

If we have no items at all (this is the base case $n = 0$ of our recursion), then there exists a perfect packing iff the remaining cargo space T is zero (otherwise, we would waste T tons).

In the recursive case for $n > 0$, we pick some element, say $v[n-1]$. If there exists a perfect packing for items $v[0], \ldots, v[n-1]$ and cargo space T, then either

- item $v[n-1]$ is *not* part of this packing, which means that there must be a packing of the items $v[0], \ldots, v[n-2]$ into cargo space T, or
- item $v[n-1]$ *is* part of this packing, which means that some of the remaining items $v[0], \ldots, v[n-2]$ must fit exactly into the remaining cargo space $T - v[n-1]$.

This line of reasoning is translated into the `return` statement below. The initial function call `perfectPack(T, V, V.length)` returns true iff there exists a perfect packing.

```
public static boolean perfectPack(int T, int[] V, int itemsLeft)
{
    if (itemsLeft == 0)  // base case
        return (T == 0);  // no unused space left?
    else  // recursive case
        return (perfectPack(T-V[itemsLeft-1], V, itemsLeft-1)
                || perfectPack(T, V, itemsLeft-1));
        // either item 'itemsLeft-1' goes in, or it doesn't
}
```

The running time of above algorithm is exponential in the number of items. If you can come up with a polynomial (e.g. quadratic, cubic, ...) algorithm, you will become very, very famous.

Problem 2: Three-way Merge Sort

```
public static int[] merge3Sort(int[] A, int first, int end)
{
    if (end - first <= 1) {  // base case: 0- or 1-element array
        int[] R = new int[end-first];
        if (first != end) R[0] = A[first];
        return R;
    }

    int left = first + (end-first) / 3;
    int right = first + (end-first) * 2 / 3;
    int[] leftPart = merge3Sort(A, first, left);
    int[] middlePart = merge3Sort(A, left, right);
    int[] rightPart = merge3Sort(A, right, end);

    return merge3(leftPart, middlePart, rightPart);
}

public static int[] merge3(int[] A, int[] B, int[] C)
{
    int a=0, b=0, c=0, r=0;
    int[] R = new int[A.length+B.length+C.length];

    // merging all three arrays
    while (a<A.length && b<B.length && c<C.length)
        R[r++] = A[a] < B[b] ? (A[a] < C[c] ? A[a++] : C[c++])
                             : (B[b] < C[c] ? B[b++] : C[c++]);

    // now one array is depleted, merge the other two
    // only one of these loops will be executed
    while (a < A.length && b < B.length)
        R[r++] = A[a] < B[b] ? A[a++] : B[b++];
    while (a < A.length && c < C.length)
        R[r++] = A[a] < C[c] ? A[a++] : C[c++];
    while (b < B.length && c < C.length)
        R[r++] = B[b] < C[c] ? B[b++] : C[c++];

    // now two arrays are depleted, copy rest
    // only one of these loops will be executed
    while (a < A.length) R[r++] = A[a++];
    while (b < B.length) R[r++] = B[b++];
    while (c < C.length) R[r++] = C[c++];

    return R;
}
```

First we observe that the merge routine requires $O(a + b + c)$ time to merge three arrays of length a, b, and c, respectively.

Given an array of length n, the merge sort routine takes

$$T(n) = 3 \cdot T(n/3) + O(n)$$

steps, with $T(0) = T(1) = O(1)$. This *recurrence relation* represents a generalized divide-and-conquer algorithm and has the solution

$$T(n) = O(n \log n),$$

as you can easily verify by substituting $T(n)$ in above relation (see the textbook on page 205 for a derivation).

We may conclude that 3-way merge sort does not improve upon 2-way merge sort at all.

Problem 3: Recursive Descent Parsing

Valid Boolean expressions can be described by the *grammar*

$$E \rightarrow \texttt{T} \mid \texttt{F} \mid \texttt{!}E \mid \texttt{(}E * E\texttt{)}$$

where $*$ is one of `&`, `|`, `>`, or `=`. By identifying the first character of the subexpression to be parsed, we know immediately which characters are to follow.

Here is the transcript of a session with a slightly modified version of `Parser` that takes its input from the command line rather than a file:

```
> java Parser "!(T = F)"
!(T=F) ==> true
> java Parser "(!F)"
(!F
Syntax error in expression: expected &, |, >, or =
> java Parser "((F > !F) > F)"
((F>!F)>F) ==> false
> java Parser "(T > F & T)"
(T>F
Syntax error in expression: expected )
> java Parser "(F | !(T&F))"
(F|!(T&F)) ==> true
> java Parser "(T = F) > (F = T)"
(T=F) ==> false
```

Note that in the last example, our parser did not reject the input but evaluated the beginning of the expression that constitutes a valid BE. This is permissible, but not required behavior.

```
private static CS211In boolExp;

// This method will complain about incorrect syntax, but in some cases
// it might simply throw an exception and abort.
public static boolean evalBoolExp()
{
    switch (boolExp.peekAtKind()) {
    case boolExp.WORD: {
        String cons = boolExp.getWord();
        if (cons.equals("T"))
            return true;
        else if (cons.equals("F"))
            return false;
        else
            syntaxError("T or F");
    }
    case boolExp.OPERATOR: {
        char op = boolExp.getOp();
        if (op == '!')
            return !evalBoolExp();
        else if (op == '(') {
            boolean arg1 = evalBoolExp();  // get first arg
            op = boolExp.getOp();
            boolean arg2 = evalBoolExp();  // get second arg
            if (boolExp.getOp() == ')')
                switch (op) {
                case '&': return arg1 && arg2;
                case '|': return arg1 || arg2;
                case '>': return !arg1 || arg2;
                case '=': return arg1 == arg2;
                default:
                    return syntaxError("&, |, >, or =");
                }
            else
                return syntaxError(")");
        }
        else
            return syntaxError("! or (");
    }
    default:
        return syntaxError("T, F, !, or (");
    }
}
```

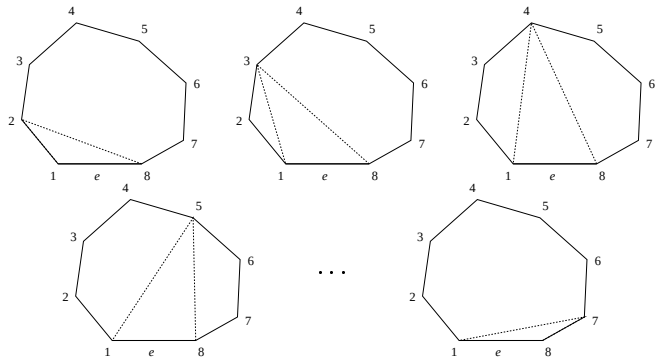

Figure 1: Triangulations of an octagon

Problem 4: Triangulation

We derive a recursive formula $t(n)$ for the number of distinct triangulations of a convex polygon with n vertices.

The base case is obviously a triangle. Since there is exactly one triangulation, namely the triangle itself, we have

$$t(3) = 1.$$

For the recursive case $n > 3$, suppose we are given some polygon with n vertices. Pick an arbitrary edge e. No matter how we triangulate the polygon, this edge e will be the edge of exactly one triangle of the triangulation, and there are $n-2$ possible ways to select this triangle (see Figure 1). Each of these triangles cuts the original polygon into one or two smaller polygons. Depending on the middle triangle, these smaller polygons have

- $n-1$,
- 3 and $n-2$,
- 4 and $n-3$,
- ...
- $n-2$ and 3, and
- $n-1$

vertices, respectively.

For example, let's look at the third octagon in Figure 1. The middle triangle P_{148} cuts the octagon into two smaller polygons P_{1234} and P_{45678}. By our recursive assumption, there are $t(4)$ distinct triangulations of the quadrangle and $t(5)$ triangulations of the pentangle. Both triangulations are independent of each other and can be combined arbitrarily, so that the total number of triangulations of the octagon with given middle triangle P_{145} is $t(4) \cdot t(5)$.

There are $n-2$ different ways to pick the middle triangle over edge e, and summing up all these independent cases yields

$$t(n) = t(n-1) + \left(\sum_{i=3}^{n-2} t(i)t(n+1-i) \right) + t(n-1),$$

which we can simplify to

$$t(n) = \sum_{i=2}^{n-1} t(i)t(n+1-i),$$

by defining $t(2) = 1$ as an additional base case.

It is important to observe that for two different middle triangles over edge e, we cannot arrive at the same triangulation. This guarantees that we are not double-counting some triangulations in above sum.

The following table shows the number of distinct triangulations for some values of n, where the *secs* columns show the time required to compute $t(n)$ on a Sun Ultra workstation using our recursive formula (compare our solution to the `fibo` method in problem 7). An iterative version of the same formula using a `for` loop would run in linear time.

n	$t(n)$	secs	n	$t(n)$	secs
2	1	0	10	1430	1
3	1	0	11	4862	1
4	2	1	12	16796	3
5	5	1	13	58786	5
6	14	1	14	208012	12
7	42	1	15	742900	37
8	132	1	16	2674440	108
9	429	1	17	9694845	346

There is also a *closed* formula that does not use recursion:

$$t(n) = \frac{1}{4n-6} \binom{2n-2}{n-1}$$

Derivation of this expression is non-trivial (which is why we used *Mathematica* for this job), but you can easily verify that it is correct by substituting $t(n)$ into our recursive formula.

Problem 5: $O(\cdot)$-Notation

First observe that the logarithm is a monotonically increasing function, i.e. $x < y$ iff $\log x < \log y$ for $x, y > 0$.

To prove $f(n) = O(g(n))$, we have to show the existence of two constants c, N such that $f(n) < c \cdot g(n)$ for all $n > N$ holds.

(a) $\log n^2 = O((\log 2n)^2)$

Proof: For $n > 2$ we have

$$\begin{aligned} \log n^2 &= 2 \log n \\ &< 2 \log 2n \\ &= \log(2 \cdot 2) \cdot \log 2n \\ &< \log 2n \cdot \log 2n. \end{aligned}$$

So $c = 1$ and $N = 2$, qed.

(b) $(\log 2n)^2 = O(n \log \log n)$

Proof: Set $n = 2^{m-1}$. For $n > 64$, or $m > 7$, we have $m^2 < 2^{m-1}$. Then

$$\begin{aligned} (\log 2n)^2 &= (\log 2^m)^2 \\ &= m^2 \\ &< 2^{m-1} \\ &= n \\ &= n \cdot \log 2 \\ &= n \cdot \log \log 4 \\ &< n \cdot \log \log n. \end{aligned}$$

So $c = 1$ and $N = 64$, qed.

Problem 6: Computational Complexity

(a) Matrix-Matrix Multiplication

```
 1  static int[][] matrixMatrixMult(int[][] A, int[][] B) {
 2     int[][] r = new int[A.length][B[0].length];
 3     for (int i=0; i<A.length; i++)
 4        for (int j=0; j<B[0].length; j++) {
 5            r[i][j] = 0;
 6            for (int k=0; k<B.length; k++)
 7                r[i][j] += A[i][k] * B[k][j];
 8        }
 9     return r;
10  }
```

We assume multiplying an $m \times n$ matrix and an $n \times p$ matrix. Allocation of the return matrix in line 2 takes constant time c_1 (assuming that it takes $c_1 mp$ time won't change the result). The outermost `i` loop (lines 3–8) is iterated m times, and contains only the `j` loop (lines 4–8), which is iterated n times. The `j`

loop consists of an assignment statement, which requires constant time c_2, and the `k` loop (lines 6–7), which adds and multiplies $2p$ elements in constant time c_3 each. Summing this up, we arrive at

$$c_1 + m \cdot n \cdot (c_2 + 2pc_3) = c_1 + c_2mn + 2c_3mnp$$

or $O(mnp)$ as our running time.

The best currently known algorithm for $n \times n$ matrices runs in $O(n^{2.378\ldots})$ time.

(b) Matrix-Vector Multiplication

Multiplying an $m \times n$ matrix and an n vector using given algorithm takes $O(mn)$ time.

(c) Dot-Product

Multiplying two n vectors using given algorithm takes $O(n)$ time.

Problem 7: Running Time

The following table lists some running times as reported by the `RunTime` program. You may have noticed that the value of `fib(50)` computed is incorrect due to an integer overflow.

n ($\times 10^4$)	`lin`	`bin`	n ($\times 10^3$)	`merge`	`sel`	n	`fib`
25	38	0	20	223	11364	25	57
50	63	0	40	436	44617	30	492
75	89	0	60	642	104349	35	5660
100	107	0	80	878	190181	40	58185
125	133	0	100	1132	344479	45	646263
150	164	0	120	1400	472279	50	7122271
175	177	0	140	1597	742062		
200	211	0	160	2102	915238		
225	230	0	180	2069	1219380		
250	254	0	200	2526	1551285		
275	279	0	220	2579	1967834		
300	302	0	240	2936	2351511		

The column for binary search looks peculiar but is perfectly reasonable for a logarithmic algorithm. Binary search is so fast that even a million repetitions are computed within the blink of an eye. Remember that in order to double the running time, you have to square the input size! This requires more data than our small computer can handle.

Benchmarks should generally be taken with a grain of salt. The Java method `currentTimeMillis()` used for timing returns the current system time in milliseconds, but the granularity of this timer is actually much coarser. Further distortions might be introduced by the Java virtual machine and the garbage collector or other processes running on the computer during the benchmark.

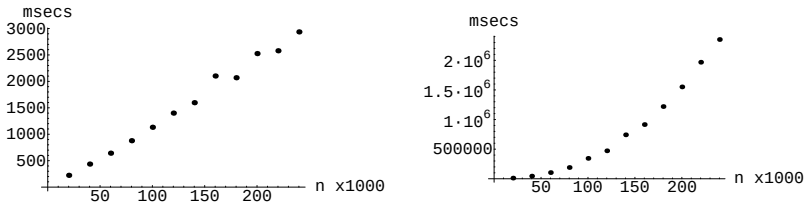

Figure 2: (a) Merge sort and (b) Selection sort

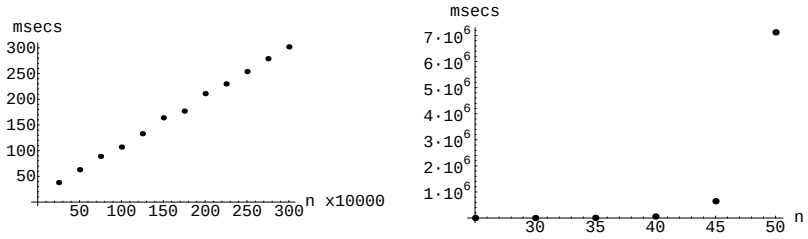

Figure 3: (a) Linear search and (b) Fibonacci numbers

Figure 2 and Figure 3(a) show the running time of merge sort, selection sort, and linear search, respectively. Figure 3(b) shows the exponential running time of the Fibonacci number algorithm. If you had started our algorithm right after the Big Bang (approx. 10 billion years ago), you would have computed no further than `fib(65)` by now.