CS2223, HW3:
Divide and Conquer; Greedy Algorithms*

Course website: `http://web.cs.wpi.edu/~cs2223/b12/`. Hand in your answer by midnight, 19 Nov., so that we can discuss some problems in class Tuesday. Use Turnin at `https://turnin.cs.wpi.edu/`.

Some *CLRS* problems and exercises are included in this week's list. They are an important part of the homework.

Working in groups and talking about the problems is strongly encouraged. More enjoyable and more educational. You can also discuss them with Fei, Linglong, Xianjing, and with me. Our office hours are listed at `http://web.cs.wpi.edu/~cs2223/b12/#personnel`.

A. Divide and Conquer. You are consulting with the Centers for Diseased Computing (CDC). They have a large number of data-sets—1,025 to be precise—related to drug tests for a particular disease. We will call these data-sets $d_0, \ldots, d_{1024}$.

The CDC would like to perform a processing-intensive operation to combine them. We will call this operation $\oplus$ (pronounced "oh-plus"). A particular $\oplus$ operation on a pair of data-sets of this size takes about a day; the resulting data-set $d \oplus d'$ is of the same size as the inputs.

Luckily, the CDC has a lot of available processors. Although an individual $\oplus$ operation is hard to parallelize, different $\oplus$ operations can be performed at the same time on different processors. Your job is to parallelize the sequence of $\oplus$ computations.

A.1. About how many years will it take to compute step-by-step

$$(((d_0 \oplus d_1) \oplus d_2) \oplus \ldots d_{1024})?$$

*Due: Monday night, 19 Nov.

Answer. *Since there are 1,024 $\oplus$ signs in that expression, and no one of them can be computed until all the $\oplus$s to its left are finished, it will take $1024/365 = 2.8$ years.*

A.2. What algebraic property of $\oplus$ justifies computing

$$((d_0 \oplus d_1) \oplus (d_2 \oplus d_3))$$

instead of

$$(((d_0 \oplus d_1) \oplus d_2) \oplus d_3)?$$

That is, what algebraic property would imply that both computations give us the same result?

Answer. *If $\oplus$ is **associative**, the two expressions give the same result.*

A.3.

(a) What is the total elapsed time required to execute $(((d_0 \oplus d_1) \oplus d_2) \oplus d_3)$ start to finish?

(b) How many processors could be used simultaneously in computing $((d_0 \oplus d_1) \oplus (d_2 \oplus d_3))$?

(c) How much elapsed time (start-to-finish) would the computation take, using those processors?

Answer. *The former requires **3** days. The latter can use **two processors**, one to compute $(d_0 \oplus d_1)$, and one to compute $(d_2 \oplus d_3)$; one of those processors can then combine its result with the result of the other processor's work. Thus, the total elapsed time will be **2** days.*

This suggests using a divide-and-conquer strategy to parallelize the computation. Assume that the procedure

```
do_oplus(i,j)
```

will retrieve the data-sets d_i and d_j, select a currently unused data-set number k, compute $d_i \oplus d_j$, and store the resulting data set as d_k. We always use a new, not-yet-used index k. You will also use the procedure

```
run_parallel(u, proc_call_1,
             v, proc_call_2)
```

It executes `proc_call_1` on the current processor, assigning its result to the variable `u`. Meanwhile, it will find a currently unused processor. It will execute `proc_call_2` on that new processor, assigning its result to the variable `v`. It then frees the new processor. Execution of this procedure is complete as soon as both `proc_call_2` and `proc_call_1` have finished. For instance,

```
run_parallel(u, do_oplus(i,j),
             v, do_oplus(k,l))
```

runs $d_i \oplus d_j$ on the current processor and $d_k \oplus d_\ell$ on a new processor. The variables u, v store the data-set numbers for the two results. The total elapsed time to run this is the maximum of the time to run the two procedure calls.

A.4. *Write pseudocode* for a recursive procedure

```
int oplus_range(bot,top)
```

that will return the data-set number for a data-set that is the result of combining *all* the data-sets with numbers from `bot` to `top` (inclusive).

If `bot == top`, it can just return the data-set number `bot` with no computation. If `bot == top+1`, it should call `do_oplus`.

Otherwise, it should divide the range from `bot` to `top`, using `run_parallel` to call itself on each piece. Finally, a call to `do_oplus` combines the results, returning the data-set number for the result.

Answer.

```
int oplus_range(bot,top) {
    if bot == top {
      return bot
    }
    else if bot == top+1 {
      return do_oplus (bot,top)
    }
    else {
      int u, v, mid;
      mid = floor((bot+top)/2);
```

```
    run_parallel(v, oplus_range(bot,mid),
                 u, oplus_range(mid+1,top));
    return do_oplus(v,u)
  }
}
```

A.5. A Recurrence for Elapsed Time. *Write* and *solve* a recurrence that describes the amount of *elapsed time* $T(n)$ that `oplus_range(bot,top)` requires, when $n =$ `top - bot`.

Each call to `do_oplus` requires a constant time c, where c is about one day. Addition, subtraction, division, assignment, and allocating a new processor take (relative to this) 0 time. Multiple processors working simultaneously do not contribute to elapsed time.

Answer. $T(0) = 0$; $T(1) = c$; otherwise,

$$T(n) = T(\lceil n/2 \rceil) + c.$$

By the recursion tree method, we have a tree of height $\log_2 n$ with an elapsed time of c at each level, adding up to a total elapsed time of $c \log_2 n$

A.6. A Recurrence for processor-days. *Write* and *solve* a recurrence that describes the number of *processor-days* $P(n)$ required for `oplus_range(bot,top)`, when $n =$ `top - bot`. Multiple processors working simultaneously all contribute to the processor-days.

Answer. $P(0) = 0$; $P(1) = 1$ processor-day; otherwise,

$$P(n) = 2 \cdot P(\lceil n/2 \rceil) + 1.$$

By the recursion tree method, we have a tree of height $\lceil \log_2 n \rceil$, in which one processor-day used on the top level and twice as many processor-days used at each successive level. Thus, this uses $\sum_{i \leq \log_2 n} 2^i \in \Theta(n)$ processor-days.

B. A Greedy Algorithm. In your job for the investment bank Golden Smacks, you devise algorithms to help the finance people make decisions.

A small bureau within your bank has an amount of money that they want to loan to a sequence of short-term business projects. The projects are all about the same size, which is about equal to the amount the bureau has available, so they can fund one project at a time. Also, the fee earned by the bank seems to be about equal for each project.

Thus, your job is to maximize the number of projects funded.

Each project has a *start-date* and an *end-date*, with the end-date always after the start-date. If two projects a, b have start dates s_a, s_b and end-dates e_a, e_b, they are *compatible* if $e_a \leq s_b$ or $e_b \leq s_a$. That is, one should have ended no later than the other starts.

The procedures `start(p)` and `end(p)` return the start and end dates of project `p`. Procedure `lessEq(date1,date2)` returns `true` if `date1` is less than or equal to `date2`; that is, `date1` is not after `date2`.

B.1. Sort the entries. What order will you consider the projects in? (You do not need any pseudocode here.)

Answer. Either by *earliest end date*, or else by *latest start date*.

B.2. Build the maximal compatible sets. Suppose that the result of your sorting yields an array of projects `projects[]` of length `k`. So the entries in the array are `projects[0]`, ..., `projects[k-1]`.

Write pseudocode to initialize a set `a` to contain `projects[0]`, and to maintain some information so that you will lbe able to determine which new projects are compatible. Then iterate through the rest of `projects[]`, greedily adding each p to `a` if it is compatible with the projects already in `a`.

Answer.

```
a = { projects[0] };
last_end = end(projects[0]);
for i from 1 to k-1 {
  if lessEq(last_end, start(projects[i])) {
     a = a ∪ { projects[i] };
     last_end = end(projects[i])
     }
  }
```

```
return a
```

B.3. Estimate runtime. *Estimate the runtime* for your code from B.3 by choosing a f such that it will complete in time $\Theta(f(k))$.

Answer. The runtime is linear in k, i.e. $\in \Theta(k)$.

Some Problems from *CLRS*.

Sec. 4.1 Do exercises 4.1-1, 4.1-4, p. 74.

Sec. 4.4 Do exercises 4.4-1, 4.4-2, 4.4-3, 4.4-4, p. 92.

Sec. 6.1 Do exercises 6.1-1 up to 6.1-6, p. 153.

Sec. 6.5 Do exercises 6.5-1 up to 6.5-2, 6.5-7, p. 166.

Sec. 7.1–2 Do exercises 7.1-1, 7.1-2, and 7.1-3, p. 173, and 7.2-1 and 7.2-2, p. 178.

Sec. 16.1 Do exercises 16.1-2, 16.1-3, and 16.1-4.