CMSC 27200 Assignment 3

Rachel Hwang

January 29, 2014

- **Exercise 4.17**
  While we already have an algorithm for interval scheduling detailed in the testbook, this problem poses some difficulty because jobs can continue through midnight, "wrapping around" if we consider the day from 12:00 to 23:00. In short, there is no fixed place to start the day from. In order to resolve this, we can simply try starting the day from the beginning of a single job, removing all jobs that overlap with it. Given that first job, the textbook interval scheduling alogrithm can be run as usual with the reamining jobs. However, since we don't know which job is the optimal first pick, we must try this process beginning with very job, storing results (how many jobs scheduled) for each case, and returning the best one. The algorithm is as follows:

  1. Pick one job j. Use its start time as the beginning of the 24-hour day.
  2. Remove all jobs which overlap with j from consideration.
  3. Run the interval scheduling algorithm as usual. Record the number of accomodated jobs for j as the starting job.
  4. Repeat steps 1-3 for all jobs.
  5. Return the schedule derived from the starting job that performed the best (had the most jobs accomodated).

  Proof:
  When we choose a starting job j, we assume that j is in the optimal solution set of jobs. The optimal solution must have at least one job in it. We know that the interval scheduling algorithm will produce an optimal solution given a set of jobs and day period. Since at least one of the jobs must be in the optimal solution, and we create an optimal (for standard intervals) solution for each job, the optimal solution must be among the schedules that the algorithm produces. Since we choose the final result based on number of jobs scheduled, which is our criteria for optimality, the returned schedule must be optimal.

  The complexity of the interval scheduling problem is O(n log n) when using a heap. The complexity of this algorithm is thus O(n^2 log n), since the interval scheduling algorithm is performed once per each of n jobs.

- **Exercise 4.19**
  Any *maximum* spanning tree (a spanning tree with the sum of all weights maximized rather than minimized) over G is also an optimal bottleneck spanning tree.

  Proof:
  By way of contradiction, assume that a maximum spanning tree T over G is not an optimal bottleneck spanning tree for G. (In order to reason using minimum spanning tree properties, let all edge weights in G be negated, making T a minimum spanning tree.)

  This means that for some vertices v, w connected by path s in T, there exists a path p between them which is not included in T, and which contains an edge e which has a greater (when negated) cost than any edge in s. We can concatenate paths s and p to create a cycle. By property (4.41), an edge does not belong in a minimum spanning tree if and only if v and w can be joined by a path consisting entirely of cheaper edges. Since e (negated) has been established to be costlier than all edges in that cycle, this gives us a contradiction because we started with e in path s contained in T. Thus, T with edge weights un-negated must be an optimal bottleneck spanning tree.

  Since any maximum spanning tree is an optimal bottleneck tree, we can use use Prim's Algorithm on a graph G with the weights negated in order to find a bottleneck tree.

- **Exercise 5.1**
  The algorithm is as follows given databases A and B each of size n.

```
get_median(A, B, length) {
    a = A[n/2];
    b = B[n/2];
    n = length;
    if (n = 1) {
     return min(a, b);
    }
    if (a > b) {
       get_median(A[0], B[n/2], n/2);
    }
    else {
       get_median(A[n/2], B[0], n/2);
    }
}
```

In each call, this algorithm first finds the medians of A and B. Because the overall median must be between $median(A)$ and $median(B)$, this reduces the search space by a factor of two. The algorithm is then called again with the smaller search space, the halved databases. When the length of each of the databases subsets we are checking reaches 1, we know that the search space is only two elements. Since we want the (n/2)th element, we return the smaller of the two.

Claim:
Let c be the overall median of the two databases. Either $median(A) < c < median(B)$ or $median(B) < c < median(A)$.
Proof:
By way of contradiction, assume that $median(A) < c$ and $median(B) < c$. If $median(A) < c$, then because we know that $median(A)$ is larger than n/2-1 elements in A, c must be greater than n/2 elements in A. By the same logic, if $median(B) < c$, c must be greater than n/2 elements in B. So c is greater than a total of n elements. However, since we define the median in a set of size $2n$ to be greater than exactly n-1 elements, this gives us a contradiction, so c cannot be greater than both $median(A)$ and $median(B)$.

By an analogous argument, c cannot be less than both $median(B)$ and $median(A)$. Therefore c must be between those two values.

Claim:
This algorithm returns the median in O(log n).
Proof:
At each step, this algorithm divides the size of the search space by 2 using the median property proved above. Thus, the search space must be narrowed to 2 elements after $log_2(n)$ iterations. The last two elements must be the (n/2)th and the (n/2+1)th element, thus the less of the two must the desired median. This means that the algorithm returns in $log_2(n) + 1$ iterations, which runs in O(log n) time.

- **Exercise 6.1**

  a. Consider the case: (4)-(5)-(3)-(1)
  The "heaviest first" algorithm chooses the set $\{(5), (1)\}$ with a value of 6. However, a more optimal set is $\{(4), (3)\}$ with a value of 7. This algorithm hence does not always return the optimal independent set.

  b. Consider the case: (4)-(1)-(2)-(3)
  This "even-odd" algorithm chooses the set $\{(4), (2)\}$ with a value of 6. However, a more optimal set is $\{(4), (3)\}$ with a value of 7. This algorithm hence does not always return the optimal independent set.

  c. The algorithm will iterate through nodes in order from v_1 to v_n. First, we can define the function returning the optimum value at the ith node as follows:

$$get\_opt(i) = max \begin{cases} v_i + get\_opt(i-2) \\ get\_opt(i-1) \end{cases}$$

$$get\_opt(2) = max \begin{cases} v_2 \\ get\_opt(1) \end{cases}$$

$$get\_opt(1) = v_1$$

The algorithm returning the optimal set is as follows:

```
set = []
for (i = 1; i <= n; i++) {
    if (i == 1) {
        if (opt(1) = opt(2)) {
         add(list, v_i);
        }
    }
    elif (opt(i) != opt(i-1)) {
        add(list, v_i);
    }
}
return set;
```

Proof:
Let $opt(i)$ return the value of the optimal plan at week i. By definition $opt(0) = 0$. By way of induction, suppose $get\_opt(i)$ finds $opt(i)$ for all $i < j$ where $j > 0$. By induction hypothesis, we know that $get\_opt(j-1) = opt(j-1)$ and $get\_opt(j-2) = opt(j-2)$. Since value is calculated by taking the maximum of two values (1) the result of adding the current node value to the optimal sum at the node before last, or (2) simply using the optimal sum from the previous node, it follows that

$$opt(j) = max(v_i + get\_opt(j-2),\ get\_opt(j-1) = get\_opt(j)$$

In order to run in polynomial time, we must use memoization, recording each $opt(x)$ value once calculated. With memoization, the complexity of this algorithm is linear, since there is a set ammount of work done per element.

- **Exercise 6.2**

a. Consider the following case:

	w1	w2	w3
l	5	5	5
h	5	15	50

The proposed algorithm would return $\{h_2, l_3\}$ for a total value of 20. The correct answer is $\{l_1, h_3\}$, a total value of 55. Therefore, the proposed algorithm does not always return the optimal solution.

b. The algorithm returning the value of the optimal plan up to week i is as follows:

$$get\_opt(i) = max \begin{cases} h_i + get\_opt(i-2) \\ l_i + get\_opt(i-1) \end{cases}$$

$$get\_opt(2) = max \begin{cases} h_2 \\ l_2 + get\_opt(1) \end{cases}$$

$$get\_opt(1) = max \begin{cases} h_1 \\ l_1 \end{cases}$$

Proof:
Let $opt(i)$ return the value of the optimal plan at week i. By definition $opt(0) = 0$. By way of induction, suppose $get\_opt(i)$ finds $opt(i)$ for all $i < j$ where $j > 0$. By induction hypothesis, we know that $get\_opt(j-1) = opt(j-1)$ and $get\_opt(j-2) = opt(j-2)$. Since value is calculated by adding the current week with the sum at the previous week if a low-stress job is chosen, or instead adding the sum from the week before last is a high-stress job is chosen, it follows that

$$opt(j) = max(h_i + get\_opt(j-2),\ l_i + get\_opt(j-1) = get\_opt(j)$$

- **Extra Credit.**

Consider the following strategy to sort an array X of n elements. If $n \leq 2$, sort by comparisons if needed.
Step 1: Sort the first 2/3n elements (the subarray X[1] ... X[2n/3])
Step 2: Sort the last 2/3n elements of the resulting array
Step 3: Sort the first 2/3n elements of the resulting array
(After each sort, the result is in the original array.) Do not worry about floor and ceiling in a first analysis: assume 2n/3 is an integer throughout the recursion.

1. *Show that the algorithm will correctly sort.*

   Proof: By induction. The base case, where $n \leq 2$, is obviously true. Induction hypothesis: assuming that Sort sorts correctly for all input sizes less than n, Sort must correctly sort n elements.

   Assume by way of contradiction that an array E of size n is incorrectly sorted by this algorithm. Then there must be some consecutive elements i and j such that $j < i$ but i comes before j. One of several cases must be true.

   Case 1: i and j are both in the first $2/3n$ elements.
   If this is the case, then we have a contradiction, because the Sort at Step 3 cannot have sorted correctly.

   Case 2: i and j are both in the last $1/3n$ elements.
   If this is the case, then we have a contradiction, because the Sort at Step 2 cannot have sorted correctly.

   Case 3: i is in the first $2/3n$, but j is in the last $1/3n$.
   Let A refer to the first $1/3$ slots in the array, B refer to the middle $1/3$ and C refer to the last $1/3$. In order for this configuration to have occured, i cannot have been the in either B or C during Step 2 (when j was last touched), otherwise the Sort at that step must have failed. This means that i must be in A after Step 1. For that to be true, since we assume Sort worked at that step, all the elements in B must be greater than i after step 1. Assuming that the Sort at Step 2 works, subsequently, since all the elements in B are greater than i which is greater than j, during Step 2, j must be placed in section B (coming after all the $1/3n$ elements that just occupied that section). This gives us a contradiction, since if j is placed in B by Step 2, there is no way for j to be placed back in C during Step 3.

   Since each of these cases create a contradiction, it must be the case that E of size n is correctly sorted. Therefore, by induction, the algorithm correctly sorts lists of any size.

2. *Derive a recurrence relation for the (asymptotic) number of comparisons and solve it.*