

Problem 2

Consider a variant on the problem of Interval Scheduling where instead of wanting to schedule as many jobs as we can on one processor, we now want to schedule ALL of the jobs on as few processors as possible.

The input to the problem is $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$, where $n \geq 1$ and all $s_i < f_i$ are non-negative integers. The integers s_i and f_i represent the start and finish times, respectively, of job i .

A schedule specifies for each job i a positive integer $P(i)$ (the processor number for job i). It must be the case that if $i \neq j$ and $P(i) = P(j)$, then jobs i and j do not overlap. We wish to find a schedule that uses as few processors as possible, i.e., such that $\max\{P(1), P(2), \dots, P(n)\}$ is minimal.

1. Design an algorithm to solve the problem in time $O(n^2)$, i.e. strictly less than $O(n^2)$
 - (a) Let P be an empty array of size n , representing $P(i)$ at index i
 - (b) Sort (s_i, f_i) by start time such that for all $i \leq j$, $s_i \leq s_j$
 - (c) Starting from the start of the sorted job array J and for each job j_i do
 - i. Starting from processor number $k = 1$
 - ii. Define $J_k \subseteq J$ such that for all $j \in J_k$, $P(j) = k$. If j_i is compatible with all $j \in J_k$, then assign j_i processor number k , i.e. let $P(i) \leftarrow k$
 - iii. Otherwise, increment k and try the previous step again until j_i is assigned to either a previously used processor number or a new processor number not used before.
 - (d) Return P
2. Prove that the above algorithm is guaranteed to compute a schedule that uses the minimum number of processors.

We will prove that the greedy choice is always in some optimal solution to the problem. Then we prove that the problem exhibits optimal substructure. Here we define a *compatible* processor number k for job j be an integer such that all jobs previously assigned k are compatible with j . Let J be the input jobs given. Let $J_t := \{j_i \in J : s_i \geq s_t\}$ be subset of J such that all jobs in J_t starts after j_t starts. Let $\max(P)$ be the maximum of processor numbers in P

Proposition. Consider any subproblem J_t , let $j_i \in J_t$ be the job with earliest starting time, and let k be the lowest compatible processor number with j_i . Then assigning k to j_i is in some optimal ($\max\{P(1), \dots, P(n)\}$ minimized) solution to J_t

Proof. Assume P' is an arbitrary optimal solution to J_t . Let $k' = P'(i)$. If $k = k'$, then we are done the proof since k is assigned to j_i by the greedy choice, which is in the optimal solution P' . Otherwise if $k \neq k'$, since k is the lowest processor number possible (i.e. $k \leq k'$), then $k < k'$. Now we can construct a solution $P = P'$ where $P(i) = k$, thus $P(i) < P'(i)$. We arrive at a contradiction on the assumption that P' is optimal. Hence we conclude that the greedy choice is always in some optimal solution to J_t \square

Proposition. *The scheduling problem exhibits optimal substructure.*

Proof. Given arbitrary index i , we separate the problem into a greedy choice and a single subproblem, i.e. $\{j_i\}$ and $J_{after} = J_i$. We make the choice assigning a processor number k to j_i , Assume such assignment is in some optimal solution to the problem P' . Now we are left with assigning processor number to J_{after} with P_{after} . Then the optimal solution follows

$$Max(P') = Max\{k, Max(P_{after})\}$$

We claim that if P' is optimal, then P_{after} is also optimal, in a sense that if $Max(P_{after}) > k$, then $Max(P_{after})$ is minimized. If $Max(P_{after}) \leq k$, then solution is already optimal. Otherwise if $Max(P_{after}) > k$, then suppose we can find a more optimal solution P''_{after} such that $Max(P''_{after}) < Max(P_{after})$ then we can substitute P''_{after} for P_{after} and construct another solution set P'' with

$$Max(P'') = Max\{k, Max(P''_{after})\} < Max\{k, Max(P_{after})\} = Max(P')$$

Hence contradicting the optimality assumption for P' , hence P_{after} must be optimal in itself. \square

We conclude by combining propositions proved earlier. By optimal substructure of the problem, given that at each step the greedy choice is optimum and we are left with finding optimal solution to a smaller subproblem, i.e. J_{after} , the solution to the original solution is optimal, specifically, the algorithm uses minimum number of processors.

3. Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.

We will use a min heap H to store an array of finish time of currently scheduled jobs.

$Q.size$ is size of the heap and assume is updated during insertion and deletion.

```

1 Function Schedule-All ( $s, f$ )
   Input:  $s, f$  are arrays of size  $n$ , representing job  $j_i = (s_i, f_i)$  at index  $i$ 
   Output:  $P$  is an array of size  $n$  storing  $P(i)$  at index  $i$ , where
            $\max\{P(1), \dots, P(n)\}$  is minimized
2    $P \leftarrow$  Array of size  $n$ 
3    $H \leftarrow$  Min-Heap
4   Sort  $s, f$  by start time together such that  $s_i \leq s_j$  for all  $i \leq j$ 
5   for  $i = 1$  to  $n$  do
6       while Heap-Maximum ( $H$ )  $< s_i$  do
7           Heap-Extract-Max ( $H$ )
8       Heap-Insert ( $H, f_i$ )
9        $P(i) \leftarrow H.size$ 
10  return  $P$ 

```

At line 6-7, we remove jobs' finish time from the heap H such that the heap retains previously scheduled jobs that overlaps j_i . The size of the heap represent the smallest compatible processor number, which we record in solution $P(i)$ at each iteration after inserting the finish time of j_i to the heap.

Now we analyze running time

- (a) Sorting takes $O(n \lg n)$
- (b) By the time the procedure terminates, each jobs' finish time is inserted and removed from the heap, since HEAP-EXTRACT-MAX and HEAP-INSERT has worst case running time of $O(\lg n)$. Heap insertion and deletion has worst case running time of $O(2n \lg n) = O(n \lg n)$
- (c) HEAP-MAXIMUM is called at least once per iteration of for loop for a total of n iterations; and it is called at most n number of times for each successful condition evaluation and subsequent deletion operation (since at most deleting a total of n items). HEAP-MAXIMUM has worst case running time of $O(1)$ hence by the time procedure terminates, heap lookup operation has a worst case running time of $O(2n) = O(n)$
- (d) Assigning P at index i takes $O(1)$ each iteration and since there are n iterations, has a worst case running time of $O(n)$
- (e) To conclude, the algorithm has a worst case running time of $O(n \lg n)$

Problem 3

Here is another variant on the problem of Interval Scheduling. Suppose we now have two processors, and we want to schedule as many jobs as we can. As before, the input is

$(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$, where $n \geq 1$ and all $s_i < f_i$ are nonnegative integers. A *schedule* is now defined as a pair of sets (A_1, A_2) , the intuition being that A_i is the set of jobs scheduled on processor i . A schedule must satisfy the obvious constraints: $A_1 \subseteq \{1, 2, \dots, n\}$, $A_2 \subseteq \{1, 2, \dots, n\}$, $A_1 \cap A_2 = \emptyset$, and for all $i \neq j$ such that $i, j \in A_1$ or $i, j \in A_2$, jobs i and j do not overlap.

1. Design an algorithm (write a pseudocode) to solve the above problem in time $O(n^2)$, i.e., strictly less than $O(n^2)$.

Let $J : \{1, 2, \dots, n\}$ be the input set of jobs given. Here we define that a job $j \in J$, having start time s , is *compatible* with $J_s \subseteq J$ if j starts after every job in J_s finishes, in other words,

$$\forall j \in J_s : f_j \leq s$$

Let subproblem J_t be a set of jobs such that

$$\forall j \in J_t : f_t \leq s_j$$

in other words, J_t is the set of jobs that starts after job t ends.

We define *waste time* W_i for a job j , having start time s , with respect to a compatible set of jobs J_s as

$$W_i = s - \max_{j \in J_s} \{f_j\}$$

in other words, the waste time is the time period between the finish time of the last finishing jobs in J_s and the start time of the job j in consideration

- (a) Let $(A_1, A_2) = (\emptyset, \emptyset)$
 - (b) Sort (s_i, f_i) by finish time such that for all $i \leq j$, $f_i \leq f_j$
 - (c) Starting from the start of the sorted job array and for each job $j \in J$ do
 - i. Test if j is compatible with A_1 and A_2 .
 - ii. If j is not compatible with either set, continue to next iteration
 - iii. If j is compatible with only one of A_1 and A_2 , then add j to the compatible set
 - iv. If j is compatible with both A_1 and A_2 , then add j to A_i such that waste time for job j with respect to A_i , W_i , is minimized.
 - (d) Return (A_1, A_2)
2. Prove that the above algorithm is guaranteed to compute an optimal schedule.

Proposition. Consider any subproblem J_t , let $j_e \in J_t$ be the job with earliest finish time with waste time W_1 and W_2 . Then making the choice of assigning j_e described above is in some optimal (i.e. $|A_1| + |A_2|$ maximized) solution to the problem.

Proof. Let (O_1, O_2) be some optimal solution to the original problem J . Let $(A_1 \subseteq O_1, A_2 \subseteq O_2)$ be the optimal solution to the subproblem J_t with respect to (O_1, O_2) . and let $(B_1 \subseteq O_1, B_2 \subseteq O_2)$ be the optimal solution to subproblem $J \setminus J_t$. Let $j_e \in J_t$ be job with earliest finish time. By definition of J_t , j_e is compatible with at least one of B_i .

- (a) Suppose j_e is compatible with exactly one of B_i , without loss of generality, suppose B_1 is the compatible set and B_2 is the in-compatible set. Let $j_a \in A_1$ be the first finishing job in A_1 . Then we have,
 - i. If $j_e = j_a$, then the proposition holds
 - ii. If $j_e \neq j_a$. Since j_e is not compatible with B_2 , $j_e \notin A_2$. Then consider a new solution set $A'_1 = A_1 \cup \{j_e\} \setminus \{j_a\}$. Note A_2 is unchanged. jobs in A'_1 are disjoint because A_1 is disjoint, $j_a \in A_1$ is the first job to finish and $f_{j_e} \leq f_{j_a}$. Since $|A'_1| + |A_2| = |A_1| + |A_2|$, (A'_1, A_2) is an optimal solution to subproblem J_t that contains j_e , hence the proposition holds.

Similar argument holds if B_2 is the compatible set

- (b) Now consider the case where j_e is compatible with both B_1 and B_2 . Without loss of generality, suppose $W_1 < W_2$, hence the greedy algorithm assigns j_e to B_1 . Let $j_a \in A_1$ be the earliest finishing job in A_1 ; let $j_b \in A_2$ be the earliest finishing job in A_2 ,
 - i. If $j_e = j_a$, then the proposition holds
 - ii. If $j_e \neq j_a$, Since j_e is compatible with B_2 as well as B_1 there are two cases as to where j_e might end up
 - A. If $j_e = j_b$, then consider a new solution set where $A'_1 = A_2$ and $A'_2 = A_1$, i.e. switching the set of jobs for processor 1 and 2. Note A'_1 and A'_2 are disjoint sets of jobs since A_1 and A_2 are disjoint sets. and $|A'_1| + |A'_2| = |A_1| + |A_2|$. Since (A_1, A_2) optimal, then (A'_1, A'_2) are optimal solutions and that A'_1 now contains j_e , because $j_e = j_b \in A_2 = A'_1$. The proposition hence holds.
 - B. If j_e is not in either A_1 or A_2 , then consider a new solution $A'_1 = A_1 \cup \{j_e\} \setminus \{j_a\}$. Proposition holds with same argument provided in (a).ii.

Similar argument holds if $W_1 > W_2$.

Proposition. *This scheduling problem exhibits optimal substructure.*

Proof. Given arbitrary index t , we separate the problem into a greedy choice and a single subproblem, i.e. $\{j_e\}$ and J_t . Let (A_1, A_2) be the optimal solution for J_t . We make the greedy choice of adding j_e to A_i or skipping j_e (i.e. $j_e = \emptyset$) to maximize time as a resource for subsequent jobs, and to minimize waste time as a resource if there is a choice to select one assuming both processors are available at the time. Assume such greedy choice is optimal, we are left with processing a smaller subproblem

$J_{after} = J_t \setminus \{j_e\}$. We claim that if (A_1, A_2) is optimal, then solution to subproblem J_{after} , (C_1, C_2) must also be optimal. Consider an alternative solution (C'_1, C'_2) that is even more optimal, i.e. $|C'_1| + |C'_2| \geq |C_1| + |C_2|$. We can construct a new solution by replacing (C_1, C_2) with (C'_1, C'_2) and get an overall more optimal solution $(A'_1 = \{j_e\} \cup C'_1, A'_2 = \{j_e\} \cup C'_2)$ such that

$$|A'_1| + |A'_2| < |A_1| + |A_2|$$

Contradicts assumption that (A_1, A_2) is optimal. Hence solution to subproblem J_{after} must be optimal \square

We conclude by combining propositions proved earlier. By optimal substructure of the problem, given that at each step the greedy choice is optimum and we are left with finding optimal solution to a smaller subproblem, i.e. J_{after} , the solution to the original solution is optimal, specifically, the algorithm schedules most jobs on two processors. \square

3. Briefly describe an efficient implementation of the algorithm, making it clear what data structures you are using. Express the running time of your implementation as a function of n (the number of jobs), using appropriate asymptotic notation.

Let A_1 and A_2 be two linked list. Job j may be appended to the tail of A_1 or A_2 in constant time $O(1)$. Assume that the last job added to A_i can be efficiently looked

up in $O(1)$ time with $A_1.tail$ operation.

```

1 Function Compatible ( $j, A_1, A_2, s, f$ )
   Output: Returns None if  $j$  not compatible with  $A_1$  or  $A_2$ , Return Both if  $j$  is
             compatible with both  $A_1$  and  $A_2$ , and return the processor number,
             either 1 or 2, if  $j$  is compatible with only  $A_1$  or  $A_2$ . Each return
             statement also include the computed waste time  $W_1$  and  $W_2$ 
2   diff-one =  $s[j] - f[A_1.tail]$ 
3   diff-two =  $s[j] - f[A_2.tail]$ 
4   if diff-one > 0 and diff-two > 0 then
5       return (Both, diff-one, diff-two)
6   else if diff-one ≤ 0 and diff-two ≤ 0 then
7       return (None, diff-one, diff-two)
8   else
9       if diff-one ≥ 0 then
10          return (1, diff-one, diff-two)
11       else
12          return (2, diff-one, diff-two)
13 Function Schedule-On-Two-CPU ( $s, f$ )
   Input:  $s, f$  are arrays of size  $n$ , representing job  $j_i = (s_i, f_i)$  at index  $i$ 
   Output:  $(A_1, A_2)$  is a set of solution to the problem given
14    $A_1, A_2 \leftarrow$  Linked-List
15    $A_1.append(1)$  // Add first finishing job arbitrarily to  $A_1$ 
16   for  $j = 2$  to  $n$  do
17        $(T, W_1, W_2) = \text{Compatible}(j, A_1, A_2, s, f)$ 
18       if  $T$  is Both then
19           if  $W_1 < W_2$  then
20                $A_1.append(j)$ 
21           else
22                $A_2.append(j)$ 
23       else if  $T$  is 1 then
24            $A_1.append(j)$ 
25       else if  $T$  is 2 then
26            $A_2.append(j)$ 
27   return  $(A_1, A_2)$ 

```

Now we analyze running time. There are $O(n)$ iterations, and in each iteration, at most one *append* operation and *tail* operation, each $O(1)$, is required for Linked List operation. There is also some $O(1)$ array lookup in s and f . Hence the algorithm has a worst case running time of $O(n)$