Lecture 2

Recap & Interval Scheduling

Supplemental reading in CLRS: Section 16.1; Section 4.4

2.1 Recap of Median Finding

Like MERGE-SORT, the median-of-medians algorithm SELECT calls itself recursively, with the argument to each recursive call being smaller than the original array A. However, while MERGE-SORT runs in $O(n \lg n)$ time, SELECT is able to run in linear time. Why is this? There are two key observations to make:

Observation 1. Given an element x of A, we can partition around x in linear time. Thus, all the steps in SELECT other than the recursive calls take a total of only $O(n)$ time.

Observation 2. Finding the median of n elements can be reduced to finding the median of $n/5$ elements and then finding the median of at most $\frac{7}{10}n$ elements. (The true figures in our case are $\lceil n/5 \rceil$ and $\frac{7}{10}n+6$, but that's not important for the analysis that follows. For a refresher on where this observation comes from, see the proof of running time and Figure 1.1.)

To see why the second observation is important, we'll do a branching analysis. This will also show how we might have figured out the running time of our algorithm without guessing. Observation 2 is just a translation into English of the recurrence relation

$$T(n) \le T\left(\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}n\right) + O(n). \tag{2.1}$$

Pick a constant c such that $T(n) \le T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + cn$ for all sufficiently large n. (Actually, for simplicity, let's assume the relation holds for all n.[1]) Then we can apply (2.1) to each of the terms $T\left(\frac{1}{5}n\right)$ and $T\left(\frac{7}{10}n\right)$, obtaining

$$\begin{aligned}
T(n) &\le T\left(\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}n\right) + cn \\
&\le \underbrace{\left[T\left(\tfrac{1}{5}\cdot\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}\cdot\tfrac{1}{5}n\right) + c\left(\tfrac{1}{5}n\right)\right]} + \underbrace{\left[T\left(\tfrac{1}{5}\cdot\tfrac{7}{10}n\right) + T\left(\tfrac{7}{10}\cdot\tfrac{7}{10}n\right) + c\left(\tfrac{7}{10}n\right)\right]} + cn \\
&\le \cdots \\
&\le cn + c\left(\tfrac{1}{5}+\tfrac{7}{10}\right)n + c\left(\tfrac{1}{5}\cdot\tfrac{1}{5} + \tfrac{1}{5}\cdot\tfrac{7}{10} + \tfrac{7}{10}\cdot\tfrac{1}{5} + \tfrac{7}{10}\cdot\tfrac{7}{10}\right)n + \cdots
\end{aligned}$$

[1] Make sure you see why we are allowed to do this.

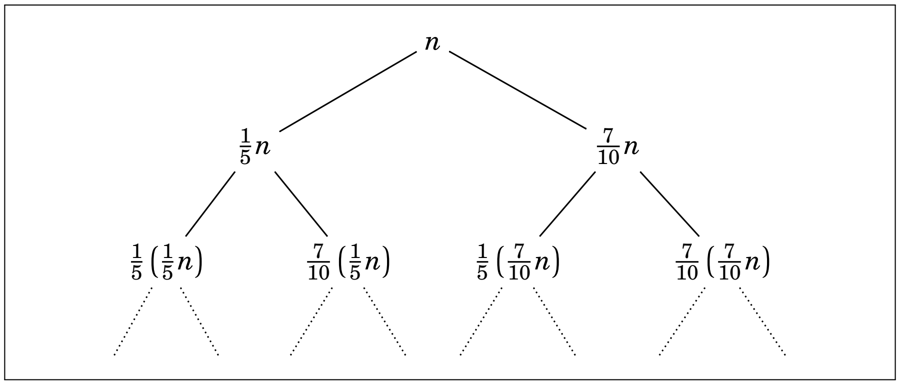

n
$\frac{1}{5}n$
$\frac{7}{10}n$
$\frac{1}{5}\left(\frac{1}{5}n\right)$
$\frac{7}{10}\left(\frac{1}{5}n\right)$
$\frac{1}{5}\left(\frac{7}{10}n\right)$
$\frac{7}{10}\left(\frac{7}{10}n\right)$

Figure 2.1. A recursion tree for SELECT. Running SELECT on an array of n elements gives rise to two daughter processes, with arguments of sizes $\frac{1}{5}n$ and $\frac{7}{10}n$. Each of these daughters produces two of its own daughters, with arguments of sizes $\frac{1}{5}\left(\frac{1}{5}n\right)$, $\frac{7}{10}\left(\frac{1}{5}n\right)$, etc.

$$\begin{aligned}
&= cn + c\left(\tfrac{1}{5} + \tfrac{7}{10}\right)n + c\left(\tfrac{1}{5} + \tfrac{7}{10}\right)^2 n + c\left(\tfrac{1}{5} + \tfrac{7}{10}\right)^3 n + \cdots \\
&= c\left[\sum_{j=0}^{\infty}\left(\tfrac{1}{5} + \tfrac{7}{10}\right)^j\right] n \qquad \text{(a geometric sum)} \\
&= 10cn.
\end{aligned}$$

This geometric sum makes it clear that it is very important for $\frac{1}{5} + \frac{7}{10}$ to be less than 1—otherwise, the geometric sum would diverge. If the number $\frac{1}{5}$ in our recurrence had been replaced by $\frac{2}{5}$, the resulting recurrence would have non-linear solutions and the running time would be $T(n) = \Theta(n^2)$. On an intuitive level, this makes sense: reducing an input of n entries to an input of $\left(\frac{1}{5} + \frac{7}{10}\right)n = \frac{9}{10}n$ entries is a good time-saver, but "reducing" an input of n entries to an input of $\left(\frac{2}{5} + \frac{7}{10}\right)n = \frac{11}{10}n$ entries is not.

Finally, let's make an observation that one is not usually able to make after writing an algorithm:

> "SELECT achieves the best possible asymptotic running time for an algorithm that solves the median finding problem."

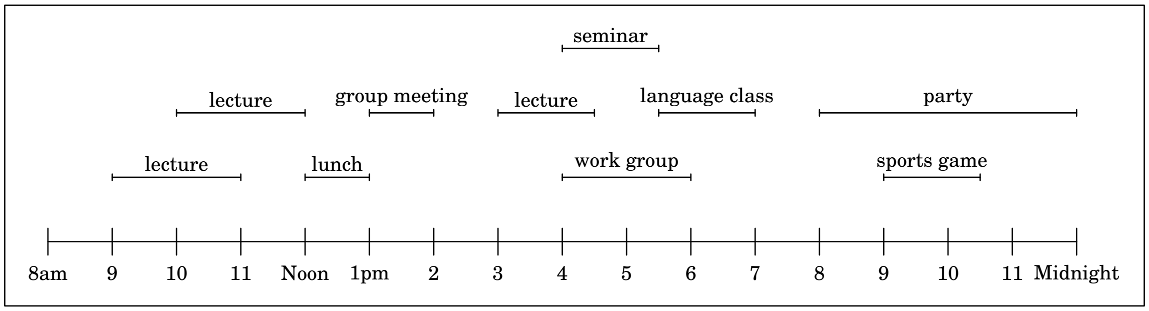
This statement must be interpreted carefully. What we mean is that any correct algorithm which solves the median finding problem must be $\Omega(n)$, since such an algorithm must at least look at each of the n entries of A. However, there are several other considerations that prevent us from being able to conclude that SELECT is the "best" median finding algorithm:

- It is only the *asymptotic* running time of SELECT that is optimal. The overhead could be very large. There may exist algorithms with a larger asymptotic running time that are still much faster than SELECT for all practical purposes because the asymptotic behavior only becomes significant when n is inhumanly large.
- There may also exist algorithms with the same asymptotic running time as SELECT but whose implementations run faster than those of SELECT because they have smaller overhead (i.e., the constants implicit in the big-O notation are smaller).
- There may exist randomized algorithms which run in sublinear time and return the correct answer with high probability, or deterministic algorithms which run in sublinear time and

return approximately the correct answer.[2]

2.2 Interval Scheduling

Does your planner look like this?



The first step to getting the most out of your day is getting enough sleep and eating right. After that, you'll want to make a schedule that is in some sense "maximal." There are many possible precise formulations of this problem; one of the simpler ones is this:

Input: A list of pairs $L = \langle (s_1, f_1), \ldots, (s_n, f_n) \rangle$, representing n events with start time s_i and end time f_i ($s_i < f_i$).

Output: A list of pairs $S = \langle (s_{i_1}, f_{i_1}), \ldots, (s_{i_k}, f_{i_k}) \rangle$ representing a schedule, such that

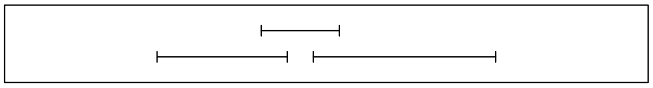
$$s_{i_1} < f_{i_1} \le s_{i_2} < f_{i_2} \le \cdots \le s_{i_k} < f_{i_k}$$

(i.e., no scheduled events overlap), where k is maximal.

This formulation corresponds to wanting to attend as many events as possible (not allowing late arrivals or early departures, without any regard for how long the events are, without any preference of one event over another, and assuming it takes no time to travel between events).

While all of us face problems like this in everyday life, probably very few of us have tried to mathematically rigorize the heuristics we use to decide what to do each day. Often we employ greedy strategies. A **greedy strategy** can be used in situations where it is easy to tell what is "locally optimal." The hope is that by making locally optimal decisions at each point, we will arrive at a globally optimal solution.[3] A couple of reasonable-sounding greedy strategies are:

- Pick the activities that take the least time (minimize $f_i - s_i$).

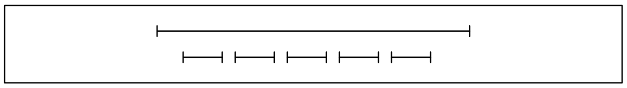


[2] While such a sublinear algorithm is probably out of the question in our case (at least if we have no prior information about A), there are lots of problems in which this does occur. For example, if you believe $P \neq NP$, we will see later in the course that there exist polynomial-time approximations to some NP-hard problems.

[3] An analogous situation is this: Imagine that you live in the Himalayan village of Nagarkot and are trying to find the highest point in the Himalayan mountain range. A greedy strategy would be to hike straight up. Since this strategy is locally optimal, you will eventually arrive at a peak from which it is impossible to go up without first going down. However, your peak will probably not be the global optimum, since Nagarkot is in central Nepal and Mt. Everest is on the Chinese border.

This strategy fails because attending one short activity may prevent you from attending two other activities.

- Each time you find yourself doing nothing, go to the activity that starts the soonest (locally minimize s_i).



This strategy fails because it is possible for the earliest activity to preclude lots of other activities.

Somewhat surprisingly, there is a greedy strategy that is optimal:

- Each time you find yourself doing nothing, go to the activity that *ends* the soonest (locally minimize f_i). In pseudocode,

```
1 while there are still activities to consider do
2     Pick $(s_i, f_i)$ with the smallest $f_i$
3     Remove all activities that intersect $(s_i, f_i)$
```

Claim. The above algorithm outputs a list of pairs $\langle (s_{i_1}, f_{i_1}), \ldots, (s_{i_k}, f_{i_k}) \rangle$ such that

$$s_{i_1} < f_{i_1} \le s_{i_2} < f_{i_2} \le \cdots \le s_{i_k} < f_{i_k}.$$

Proof. By definition, $s_i < f_i$. Now, if $s_{i_2} < f_{i_1}$, then (s_{i_2}, f_{i_2}) overlaps (s_{i_1}, f_{i_1}). Thus (s_{i_2}, f_{i_2}) must have been removed from consideration after we chose (s_{i_1}, f_{i_1}), a contradiction. The same reasoning with 1 and 2 replaced by j and $j+1$ for arbitrary j completes the proof. □

Claim. Suppose given a list of activities L. If an optimal schedule has k^* activities, then the above algorithm outputs a schedule with k^* activities.

Proof. By induction on k^*. If the optimal schedule has only one activity, then obviously the claim holds. Next, suppose the claim holds for k^* and we are given a set of events whose optimal schedule has k^*+1 activities. Let $S^* = S^*[1, \ldots, k^*+1] = \langle (s_{\ell_1}, f_{\ell_1}), \ldots, (s_{\ell_{k^*+1}}, f_{\ell_{k^*+1}}) \rangle$ be an optimal schedule, and let $S = S[1, \ldots, k] = \langle (s_{i_1}, f_{i_1}), \ldots, (s_{i_k}, f_{i_k}) \rangle$ be the schedule that our algorithm gives. By construction, $f_{i_1} \le f_{\ell_1}$. Thus the schedule $S^{**} = \langle (s_{i_1}, f_{i_1}), (s_{\ell_2}, f_{\ell_2}), \ldots, (s_{\ell_{k^*+1}}, f_{\ell_{k^*+1}}) \rangle$ has no overlap, and is also optimal since it has k^*+1 activities. Let L' be the set of activities with $s_i \ge f_{i_1}$. The fact that S^{**} is optimal for L implies that $S^{**}[2, \ldots, k^*+1]$ is optimal for L'. Thus an optimal schedule for L' has k^* activities. So by the inductive hypothesis, running our algorithm on L' produces an optimal schedule. But by construction, running our algorithm on L' just gives $S[2, \ldots, k]$. Since any two optimal schedules have the same length, we have $k = k^*+1$. So S is an optimal schedule, completing the induction. □

This algorithm can be implemented with $\Theta(n \lg n)$ running time as follows:

```
Algorithm: UNWEIGHTED-SCHEDULE($L$)
Sort $L$ according to finish time using MERGE-SORT
$S \leftarrow \langle\rangle$
$curf \leftarrow -\infty$   ▷ the finish time of our current schedule
for $(s,f)$ in $L$ do
    if $s \geq curf$ then
        $S.append[(s,f)]$
        $curf \leftarrow f$
return $S$
```

This implementation makes one pass through the list of activities, ignoring those which overlap with the current schedule. Notice that the only part of this algorithm that requires $\Theta(n \lg n)$ time is the sorting—once L is sorted, the remainder of the algorithm takes $\Theta(n)$ time.

2.2.1 Weighted Interval Scheduling

Let's try a more sophisticated formulation of the scheduling problem. Suppose that rather than maximize the total number of events we attend, we want to maximize the total amount of time we are busy. More generally still, suppose each event in L has a weight $w > 0$ representing how much we prioritize that event.

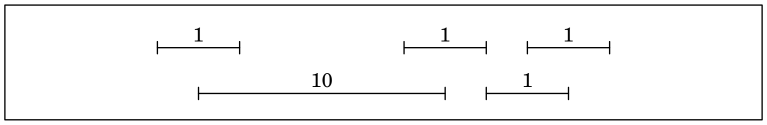
Input: A list of triples $L = \langle (s_1, f_1, w_1), \ldots, (s_n, f_n, w_n) \rangle$, representing n events with start time s_i, end time f_i ($s_i < f_i$), and weight $w_i > 0$.

Output: A list of triples $S = \langle (s_{i_1}, f_{i_1}, w_{i_1}), \ldots, (s_{i_k}, f_{i_k}, w_{i_k}) \rangle$ representing a schedule, such that

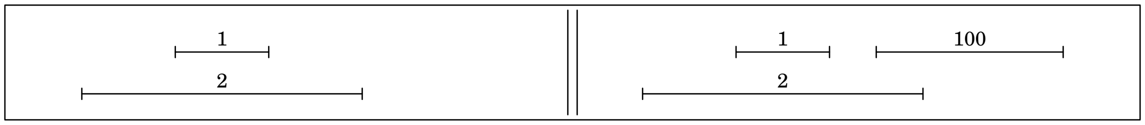
$$s_{i_1} < f_{i_1} \leq s_{i_2} < f_{i_2} \leq \cdots \leq s_{i_k} < f_{i_k}$$

(i.e., no scheduled events overlap), where $\sum_{j=1}^{k} w_{i_j}$ is maximal.

A greedy algorithm of the sort above will not work for this problem.



Which is better: attending a short event with low weight (so that you have more free time for the heavy events), or attending a slightly longer event with slightly higher weight? It depends on whether there are exciting opportunities in the future.



Here is a sketch of an efficient solution to the weighted scheduling problem:

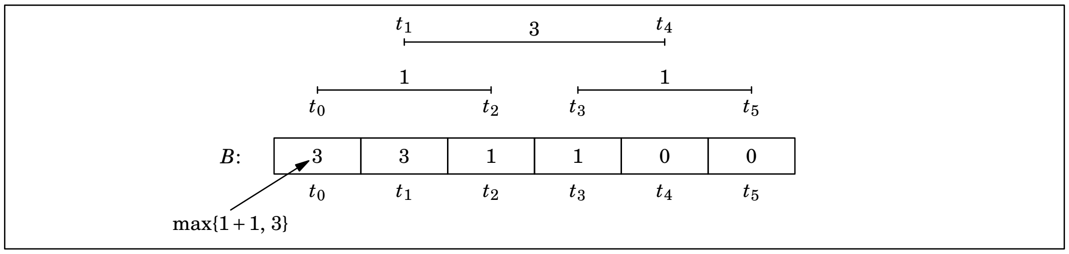


Figure 2.2. An illustration of WEIGHTED-SCHEDULE.

Algorithm: WEIGHTED-SCHEDULE(L)
Put all the start and finish times into an array $A = \langle s_1, f_1, \ldots, s_n, f_n \rangle$. Sort A, keeping track of which activity each number belongs to (that is, the index i). Make a new array B with one entry for each element of A. The idea is that the value of $B[t]$ will be the maximal weight of a schedule that starts at point t, along with a list of event indices that gives such a schedule. Thus $B[t_0]$ (assuming your day starts at $t = t_0$) will give you an optimal schedule for your day.

To construct B, first initialize all entries to zero. Then traverse backwards through A. The first time you hit a start point $s_{i_{\text{first}}}$, set $B[s_{i_{\text{first}}}] \leftarrow w_{i_{\text{first}}}$. From then on, keep track of the most recent start point you have seen in a new variable i_{cur}. Each time you hit an endpoint f_i, set $B[f_i] \leftarrow B[s_{i_{\text{cur}}}]$. Each time you hit a new start point s_i, set

$$B[s_i] \leftarrow \max\left\{B[s_{i_{\text{cur}}}],\ w_i + B[f_i]\right\}.$$

Once you have finished, $B[t_0]$ (assuming your day starts at $t = t_0$) will be the weight of an optimal schedule.

For ease of exposition, I didn't say explicitly how to keep track of the actual schedule you are constructing (which you will usually want to do—what use is the maximal weight if you don't know what schedule gives that weight?). That is an easy implementation detail that you can figure out for yourself. A more subtle issue that I didn't address is the possibility of duplicate elements of A. One way to deal with this is to, for each new start point s, keep a cache of possible weights of schedules starting at s (using a new variable to keep track of the heaviest schedule in cache). The cache should only be cleared when you hit a start point whose value is strictly less than s (or run out of elements of A), at which point you will know what value to assign to $B[s]$.

Again, it is easy to see that this algorithm runs in linear time after sorting, for a total running time of $O(n \lg n)$.

This example introduces the algorithmic paradigm of **dynamic programming**. "Dynamic programming" refers to a strategy in which the solution to a problem is computed by combining the solutions to subproblems which are essentially smaller versions of the same problem. In our example, we started with an optimal schedule for the empty set of events and then added events one at a time (in a strategic order), arriving at an optimal schedule for the full set L of events. Each time we wanted to solve a subproblem, we used the answers to the previous subproblems.

2.2.2 Conclusions

This lecture showed that the way in which a problem is formulated is important. A given real-life problem can be formulated as a precise algorithmic problem in many different, non-equivalent ways, with different solutions.

There are several standard approaches that you should have in your toolbox. So far we have seen greedy algorithms and dynamic programming; more will come up later in the course.