Principles of Algorithm Design

When you are trying to design an algorithm or a data structure, it's often hard to see how to accomplish the task. The following techniques can often be useful:

1. **Experiment with examples.** One of the most important things you can do to get a feel for how a problem works is to try generating some random input and seeing what output you should be returning. This helps you understand the problem, and can often given you a feel for how the solution can be constructed. If you experiment with enough small examples, you can often begin to see a pattern in the way that the solution looks in relation to the input. Once you understand that pattern, you are one step closer to being able to solve the problem.

2. **Simplify the problem.** Sometimes, when a problem is difficult to solve, it can be worth it to solve a related, simpler problem instead. Once you develop ideas for the simpler case, you can often apply them to handle the more complex case.

   Instead of trying to solve the problem in all generality, see if you can solve it for just a subset of the inputs. For instance, if the algorithm you are trying to design takes two inputs (such as two numbers n and k), you might consider trying to solve the problem if you set k equal to a small constant, such as 1 or 2. If the algorithm takes a list as an argument, does assuming that the list is sorted help you figure out the problem? What about assuming that all items in the list are distinct? Or assuming that the number of items in the list is a power of 2? Failing that, are there other assumptions you could make that would make the problem simpler?

   Once you have simplified the problem and solved it, you need to figure out how to make the algorithm more general again. But at this point, you have solved a similar (if simpler) problem, so you can often take advantage of …

3. **Look for similar problems.** For many of the questions in this course, the solution involves an algorithm or data structure that you have seen before. Even when considering other questions, it's often possible to draw inspiration from problems that you know how to solve. So if the problem that you are considering seems similar to a problem that you know how to solve, a good first step is to think about how the problems compare to each other. What properties do the problems share? What causes them to be different? Are the differences significant? Are the commonalities easy to see, or is it more of a stretch? Is one problem a more restricted version of the other? A more general version of the other? Or does it seem more general in some ways and more restricted in others?

   Once you understand how the problems relate to each other, think about the techniques that you used to solve the problem you understand. Can you reuse the same algorithm to solve the problem? If not, can you tweak some of the details a little to get a working solution? A lot of algorithms are based around particular techniques — for instance, you've seen several divide-and-conquer solutions, wherein you try to divide up the input into smaller pieces, then put the pieces back together again once you use a recursive call to solve the individual pieces. Try figuring out what techniques were used to solve the known problem, and then see whether you can apply them to the new problem as well.

4. **Delegate the work.** One very powerful concept in Computer Science is the idea of recursion. At its heart, recursion lets us solve problems more easily by delegating the difficult work to a recursive call. For instance, say that we want to compute $n!$. Well, that's pretty hard to compute. But if we knew $(n-1)!$, it would magically become a lot easier to compute $n!$. So we use a recursive call to compute $(n-1)!$, and then use the result of that recursive call to compute $n!$.

   If you can't figure out how to solve a problem, see if you can figure out how to solve only the last bit of it. Then see if you can use recursion to solve the rest of it. Pretty much all of the algorithms you've seen are based around this principal, so being able to apply it is a very useful skill. For ideas of how to break the problem into pieces, and ideas on how to define "the last part" of the problem, you can often look at the algorithms that you've already seen. For instance, a common thing to do involving lists is to split them in two and solve the problem recursively on both halves. If you can figure out a good way to break the list apart and then put it back together again when you're done, that's all you need.

5. **Design according to the runtime.** Sometimes the runtime that we give you provides a lot of information about what you should be doing to construct the algorithm. For instance, say that you are designing an algorithm whose runtime should be $O(\log n)$. The algorithms and data structures we know of that have that runtime are binary search, heaps, and AVL trees. (Note, however, that the cost of constructing a heap or an AVL tree is high enough that it most likely cannot be used for any algorithms with runtime $O(\log n)$. But they might be used in data structure design to implement functions that should take time $O(\log n)$.)

   If none of those seem useful, consider some simple recurrence relations that resolve to the value you're searching for. For instance, say that you know that the algorithm takes time $O(n \log n)$, but that the common $O(n \log n)$ algorithms you know don't seem to work. There are a couple of simple recurrence relations that resolve to this, such as:

   - $T(n) = O(\log n) + T(n-1)$. This could be n binary searches, or n operations on a heap or an AVL tree.
   - $T(n) = O(n) + 2T(n/2)$. This might be an $O(n)$ pass to separate the data into two groups, and then an $O(n)$ pass to put the data back together again at the end.

Maintaining Medians

Say now that we wish to solve the following problem, taken from the Spring 2008 Final Exam:

> Your latest and foolproof (really this time) gambling strategy is to bet on the median option amount your choices. That is, if you have n distinct choices whose sorted order is $c[1] < c[2] < \cdots < c[n]$, then you bet on choice $c[\lfloor (n+1)/2 \rfloor]$. As the day goes by, new choices appear and old choices disappear; each time, you sort your current choices and bet on the median. Quickly you grow tired of sorting. You decide to build a data structure that keeps track of the median as your choices come and go.

Specifically, your data structure stores the number n of choices, the current median m, and two AVL trees S and T, where S stores all choices less than m and T stores all choices greater than m.

(a) Explain how to add a new choice c_{new} to the data structure, and restore the invariants that (1) m is the median of all current choices; (2) S stores all choices less than m; and (3) T stores all choices greater than m. Analyze the running time of your algorithm.

(b) Explain how to remove an existing choice c_{old} from the data structure, and restore invariants (1-3) above. Analyze the running time of your algorithm.

Because the data structure we are using is already given to us, most of the techniques above do not apply. We do not have complete freedom in handling the problem at hand. But there's one technique that can always be used: experimenting with examples. So let's do that now. Say that we start with the list $[1, 5, 7, 10, 16, 31, 42]$. The median of this list is 10; the numbers less than 10 are $1, 5, 7$ and the numbers greater than 10 are $16, 31, 42$. Let (m_1, S_1, T_1) be the initial state of the data structure. Then $m_1 = 10$, S_1 contains $[1, 5, 7]$, and T_1 contains $[16, 31, 42]$.

What happens if we insert the number 12? Now the new sorted list is $[1, 5, 7, 10, 12, 16, 31, 42]$. Let (m_2, S_2, T_2) be the new state of the data structure. Then $m_2 = 10$, S_2 contains $[1, 5, 7]$, and T_2 contains $[12, 16, 31, 42]$. This means that $m_1 = m_2$ and $S_1 = S_2$, and we handled the new number by adding it to T. Does this work as a general technique? Obviously not — T should consist of all number greater than the median, so if we insert a number less than the median, adding it to T would be a mistake.

But maybe it works to just add all numbers greater than the median to T. To verify this, let's try adding another number greater than $m = 10$ to the list — say $c_{\text{new}} = 37$. Then the new sorted list is $[1, 5, 7, 10, 12, 16, 31, 37, 42]$. If (m_3, S_3, T_3) is the new state of the data structure, then $m_3 = 12$, S_3 contains $[1, 5, 7, 10]$, and T_3 contains $[16, 31, 37, 42]$. This is completely different. We did add $c_{\text{new}} = 37$ to T, but we also removed the number 12 from T. We also changed the median from 10 to 12, and added the number 10 to S. There's a bit of a pattern here, actually — it looks like we set m_3 to the value we removed from T, and added m_2 to S.

Some more experimentation with small examples would reveal a couple of patterns in this. When a new item is added, it usually gets added to T if it's greater than the current median and usually gets added to S if it's less than the current median. When the median changes, it usually becomes either the minimum item in T, or the maximum item in S. Once everything has been updated correctly, the number of items in S is never more than $|T|$, and the number of items in T is never more than $1 + |S|$.

These patterns suggest that the algorithm should probably contain the following steps, in some order:

- If $c_{\text{new}} < m$, add c_{new} to S. Otherwise, add c_{new} to T. (Unless c_{new} becomes the new median, which is kind of a special case.)

- If the median needs to change, do one of the following: (1) Add m to T, set $m = \max(S)$, and remove the new m from S. (2) Add m to S, set $m = \min(T)$, and remove the new m from T. (Or set $m = c_{\text{new}}$, which is kind of a special case.)
- If $|S| > |T|$ or $|T| > 1 + |S|$, do something to fix this.

The third instruction is partial — it's missing information on what should be done to fix the sizes of S and T. The second instruction is also missing a bit of information — how exactly do we tell when the median has changed? So why not try putting the two steps together? Now we have the following steps:

- If $c_{\text{new}} < m$, add c_{new} to S. Otherwise, add c_{new} to T.
- If $|S| > |T|$, add m to T, set $m = \max(S)$, and remove the new m from S. If $|T| > 1 + |S|$, add m to S, set $m = \min(T)$, and remove the new m from T.

Those two steps, in the order given above, happen to be exactly what we want.

We can verify this by checking it on a number of small examples, or by thinking about the definition of median. Both steps preserve invariants (2) and (3), given how we select the new median m. To see that the median is always stored in m, it's sufficient to think about the sizes of S and T over time. If $|T| = |S|$ or $|T| = |S| + 1$, then we have $n = 1 + |S| + |T|$ and so either $n = 1 + 2|S|$ or $n = 2 + 2|S|$. Either way, we have $\lfloor (n+1)/2 \rfloor = 1 + |S|$, and therefore m is the median. Hence, in either of those cases, there is no need to update the median.

Otherwise, we know that $|T| < |S|$ or $|T| > |S| + 1$. However, in the previous step the median was computed correctly, so we had $|T_{\text{old}}| = |S_{\text{old}}|$ or $|T_{\text{old}}| = |S_{\text{old}}| + 1$. The addition of c_{new} only changed the size of one of the two sets by 1. Now say that $|T| < |S|$. Then because $|T_{\text{old}}| \geq |S_{\text{old}}|$ and only one of those sizes could have increased, we know that $|T| = |S| - 1$. So when the second step is performed, removing a number from S and adding a number to T, we get $|S_{\text{new}}| = |S| - 1 = |T|$, and $|T_{\text{new}}| = |T| + 1 = |S_{\text{new}}| + 1$, which makes the median correct. A similar argument holds when $|T| > |S| + 1$. So the algorithm does maintain invariant (1).

What is the runtime of this? Each step performs a constant number of arithmetic operations and a constant number of insertions and deletions in an AVL tree, for a total of $\Theta(\log n)$ time in the worst case.

Once we understand how to do insertions, the solution for deletions also becomes apparent. The second step of the algorithm for insertions acts as a way to rebalance the structure, restoring the median when the sizes of S and T changed by at most 1. Therefore, the second step of the algorithm works just as well when deleting a single element of the data structure.

Rotated Array

Say that we are trying to solve the following problem, taken from the Spring 2011 Final Exam:

> Consider an array $A[1 \cdots n]$ constructed by the following process: we start with n distinct elements, sort them, and then rotate the array k steps to the right. For example,

> we might start with the sorted array $[1, 4, 5, 9, 10]$, and rotate it right by $k = 3$ steps to get $[5, 9, 10, 1, 4]$. Give an $O(\log n)$-time algorithm that finds and returns the position of a given element x in array A, or returns None if x is not in A. Your algorithm is given the array $A[1 \cdots n]$ but does *not* know k.

Several of the techniques listed above apply in this case, and most of them suggest the same approach. We are trying to find a number in an array that is somehow "close" to being sorted. We know how to do that: we use binary search. And binary search just so happens to have a runtime of $O(\log n)$, which exactly what we want. Indeed, if $k = 0$, then the array is completely sorted, and binary search works fine.

Ideally, we would like to do binary search even in cases where $k \neq 0$. But binary search requires a sorted array. The array we have is close to being sorted, but every single one of the items in the array is shifted over by some unknown amount k. This seems rather difficult to cope with, so let's try a simplification: what happens if we know k? Then we know that the array consists of two pieces: $A[1 \cdots k]$ is the last part of the sorted sequence, and $A[(k+1) \cdots n]$ is the first part of the sorted sequence. As a result, $A[(k+1) \cdots n]$ concatenated with $A[1 \cdots k]$ is a sorted array. This means that the two individual pieces are each sorted as well. And we know how to find numbers in a sorted array: binary search. So we can solve the problem by performing two binary searches: one on $A[1 \cdots k]$, and one on $A[(k+1) \cdots n]$. (We can even improve the constant by comparing x to $A[1]$ to determine which of the two subarrays to look at, so that we only perform one binary search.)

Therefore, we've solved a simplified problem. If we only knew k, we'd be set. So if we want to solve this problem, we need only develop an algorithm to find k. This algorithm needs to run in $O(\log n)$ time. What do we know that runs in $O(\log n)$ time? Binary search does, but since we don't have a sorted array, we need to do something different. Still, the structure of the algorithm might very well mimic that of binary search: examine a constant number of places in the array (binary search examines one, but the runtime bound would let us look at more), and then use the information gleaned to pick one half of the array to recurse on.

Which items in the array should we examine? Given that we want to mimic binary search, we should probably look at the middle element $m = \lfloor (n+1)/2 \rfloor$, and compare it to some value. We want to use this to find k, so we should probably consider the difference between the case where $k < m$ and the case where $k > m$. More specifically, we would like to find some other item in the array that, when compared to the middle element, tells us which side k lies on. So let's consider both cases.

First, say that $k < m$. Then the items in the array in sorted order are:

$$A[k+1] < \ldots < A[m] < \ldots < A[n] < A[1] < \ldots < A[k]$$

If instead $k > m$, then the items in sorted order are:

$$A[k+1] < \ldots < A[n] < A[1] < \ldots < A[m] < \ldots < A[k]$$

The difference in the rankings gives us a clear way to figure out which side of the array k lies in: $k < m$ if and only if $A[m] < A[1]$. This requires the examination of a constant number of items in the array. Therefore, we may find k in $O(\log n)$ time.

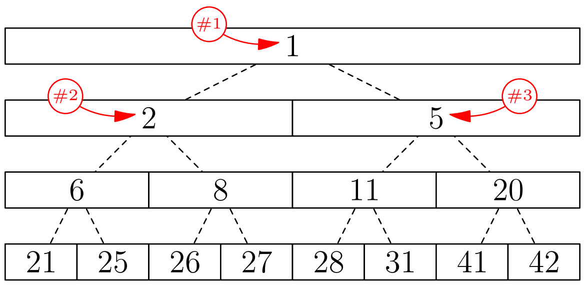
k^{th} minimum in min-heap

Now say that we wish to solve the following problem, taken from the Fall 2009 Final Exam:

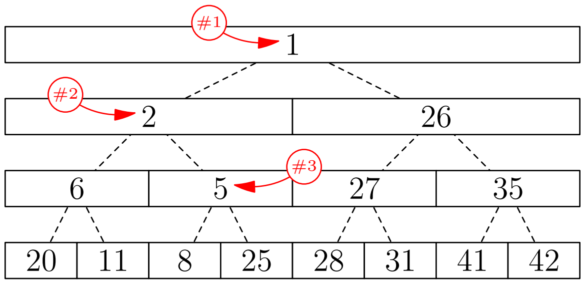
> Present an $O(k \log k)$ time algorithm to return the k^{th} minimum element in a min-heap H of size n, where $1 \leq k \leq n$. Partial credit will be given to less efficient solutions provided your complexity analysis is accurate.

This problem doesn't seem to share any commonalities with any problems we've seen previously. The runtime gives some hints — it looks like we might be sorting k items or performing k operations on a heap or an AVL tree of size k. But k is potentially far smaller than the size n of the data, so how to do we figure out which k items we want to sort or insert in our data structure?

If we simplify the problem by setting $k = 1$, it becomes easy: we're looking for the minimum element, and we can find it in $O(1)$ time by looking at the root of H. But that seems like an oversimplification, because the location of the k^{th} smallest element is not going to be so easy to find. Or is it? Let's look at some small examples, and see where the smallest numbers are within those examples. That could give us a clue as to how to find the k^{th} smallest number.



The heap above looks promising. The smallest number, of course, is at the root. The second smallest number is its left child, the third smallest number is its right child, the fourth smallest number is the left child of the left child, and so on. This seems convenient — if this held for all heaps, we would be able to find the k^{th} largest element easily. So does this hold for all heaps? Consider the following example:



Now the smallest three are the root, its left child, and the right child of the left child. So we cannot count on the top three items to be in the top two rows of the array. But further experimentation with heaps of this size does reveal some information: the smallest three items always seem to lie in the top three levels of the heap.

Why might this be true in general? An item x on the i^{th} level of the heap has $i - 1$ ancestors in the heap. The heap property ensures that each of those $i - 1$ ancestors is strictly smaller than x. Therefore, any x at a level $i \geq k + 1$ has k ancestors that are strictly smaller, and so x cannot be in the k smallest items. This means that we need only examine the top k levels of the heap to find the items we want. Is that sufficient? Well, consider how many items there are in the top k levels. Each level has twice the number of items as the level before, so the k^{th} level will have 2^{k-1} elements. In total, the first k levels will have $2^k - 1$ elements. So examining all of those elements will not let us achieve the desired runtime.

We have to be a bit more clever than that. Let's think again about the ancestors of some node x. All of x's ancestors must be strictly less than x, because of the invariants maintained by the heap. Say that x is the k^{th} smallest. Then because x's parent is strictly smaller than x, x's parent must be one of the $k - 1$ smallest items in the heap. In other words, an item can be the k^{th} smallest if and only if its parent is among the smallest $k - 1$ elements.

This feels a lot like recursion — if we could only find the $k-1$ smallest elements, we'd be able to look at all of their children to find the k^{th} smallest element. But how do we do this efficiently? If we have to search through all $2(k - 1)$ children of the $k - 1$ smallest nodes to find the k^{th} smallest node, then we're spending $\Theta(k^2)$ time in total. How can we make this process faster?

At every step, we're considering a set of "candidate" k^{th} smallest values, and picking the smallest of them. When we find the k^{th} smallest value, we know that it is no longer a candidate for the $(k + 1)^{th}$ smallest value, but that its children are now candidates. Other than that, the set of candidates doesn't actually change. This sounds like a good place to use a data structure that we've seen before: a min-heap. So this means that the overall algorithm we have is:

```
KTH-SMALLEST($H, k$)
1  $I$ = MIN-HEAP()
2  HEAP-INSERT($I, (H[0], 0)$)
3  for $j = 1$ to $k$
4      $(v, i)$ = HEAP-EXTRACT-MIN($I$)
5      if $j = k$
6          return $v$
7      $\ell$ = HEAP-LEFT-CHILD($H, i$)
8      HEAP-INSERT($I, (H[\ell], \ell)$)
9      $r$ = HEAP-RIGHT-CHILD($H, i$)
10     HEAP-INSERT($I, (H[r], r)$)
```

What is the runtime of this algorithm? Each iteration of the loop performs several heap operations on the heap I. The size of I is always $O(k)$, so the runtime of each iteration of the loop is $O(\log k)$. Hence, with k iterations, the total runtime is $O(k \log k)$.

Storing Partial Maxima

Say that we are trying to solve the following problem, taken from the Fall 2010 Final Exam:

> 6.006 student, Mike Velli, wants to build a website where the user can input a time interval in history, and the website will return the most exciting sports event that occurred during this interval. Formally, suppose that Mike has a chronologically sorted list of n sports events with associated integer "excitement factors" $e_1, \ldots, e_n$. You can assume for simplicity that n is a power of 2. A user's query will consist of a pair (i, j) with $1 \leq i < j \leq n$, and the site is supposed to return $\max(e_i, e_{i+1}, \ldots, e_j)$.
>
> Mike wishes to minimize the amount of computation per query, since there will be a lot of traffic to the website. If he precomputes and stores $\max(e_i, \ldots, e_j)$ for every possible input (i, j), he can respond to user queries quickly, but he needs storage $\Omega(n^2)$ which is too much.
>
> In order to reduce storage requirements, Mike is willing to allow a small amount of computation per query. He wants to store a cleverer selection of precomputed values than just $\max(e_i, \ldots, e_j)$ for every (i, j), so that for any user query, the server can retrive two precomputed values and take the maximum of the two to return the final answer. Show that now only $O(n \log n)$ values need to be precomputed.

We may glean our first clue about how to solve this problem by examining not the runtime, but the space requirements. Consider the recurrence relations that might be associated with total space $\Theta(n \log n)$, If we have $T(n) = O(\log n) + T(n-1)$, that means that we must be storing structures of size $O(\log n)$. We don't know any such structures. So it's far more likely to be the other common recurrence relation: $T(n) = O(n) + 2T(n/2)$.

This suggests an approach wherein we build recursive data structures for each half of the list, then supplement it with $\Theta(n)$ extra information to handle the cases tha occur when we merge. Barring any other information about how to divide the list in two, we might just split the list so that $e_1, \ldots, e_{n/2}$ goes in one half, and $e_{n/2+1}, \ldots, e_n$ goes in the other half.

At this point, we take advantage of the power of recurrences. Let's say that we have two data structures S and T, where S can answer partial maxima queries on $e_1, \ldots, e_{n/2}$ by taking the maximum of at most two values, and T can answer partial maxima queries on $e_{n/2+1}, \ldots, e_n$ by taking the maximum of at most two values. How can we use S and T to be able to answer all partial maxima queries on $e_1, \ldots, e_n$?

Say that we are given some (i, j) and want to calculate $\max(e_i, \ldots, e_j)$. Thanks to the power of recursion, there are some very easy cases to handle. If $1 \leq i < j \leq n/2$, then we can simply pass the query along to S and call it a job well done. Similarly, if $n/2 + 1 \leq i < j \leq n$, then we can simply pass the query along to T and return whatever result. So the troublesome case occurs when $i \leq n/2 < n/2 + 1 \leq j$.

Can we store $\Theta(n)$ information to let us handle this troublesome case? There are $n/2$ distinct values for i and $n/2$ distinct values for j, so we can't simply store the answer for all choices of (i, j). In fact, because we're limited to $\Theta(n)$ space, we know that for most values of $1 \leq i \leq n/2$, we can only store $\max(e_i, \ldots, e_j)$ for a constant number of indices j. Similarly, for most values of $n/2 + 1 \leq j \leq n$, we can only store $\max(e_i, \ldots, e_j)$ for a constant number of indices i.

This means that we need some clever way of choosing what to store so that we can always answer a query $\max(e_i, \ldots, e_j)$ by taking the max of two items, as long as $i \leq n/2 < n/2+1 \leq j$. (For other values of i and j, we assume that the recursive data structures will work properly.) Because of the way the max function works, if we have two sets A and B, then $\max(A \cup B) = \max(\max(A), \max(B))$. So we wish to split $e_i, \ldots, e_j$ into two pieces.

To make sure that we don't have to store too much information, the values in the piece A_i should depend only on i, and the values in B_j should depend only on j. If A_i contains any e_k where $k > n/2 + 1$, then depending on the value of j, e_k might not belong in the set $\{e_i, \ldots, e_j\}$. So all items $e_k \in A_i$ should have $k \leq n/2 + 1$. Similarly, all items $e_k \in B_j$ should have $k \geq n/2$. Therefore, to make sure that $\{e_i, \ldots, e_j\} = A_i \cup B_j$, we must have $\{e_i, \ldots, e_{n/2-1}\} \subseteq A_i$ and $\{e_{n/2+2}, \ldots, e_j\} \subseteq B_j$. For simplicity, say that $A_i = \{e_i, \ldots, e_{n/2}\}$ and $B_j = \{e_{n/2+1}, \ldots, e_j\}$. Then for any choice of i and j, we have $A_i \cup B_j = \{e_i, \ldots, e_j\}$.

What does this mean for our algorithm? It means that the $\Theta(n)$ values we want to store to answer those queries that we can't pass along to the recursive structures consist of $\max(A_i)$ for all $1 \leq i \leq n/2$ and $\max(B_j)$ for all $n/2+1 \leq j \leq n$. Then we can take the maximum of two values to compute the maximum of the interval $\{e_i, \ldots, e_j\} = A_i \cup B_j$.