Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 1st, 2011
Professors Erik Demaine, Piotr Indyk, and Manolis Kellis — Problem Set 3

Problem Set 3

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Both Part A and Part B questions are due **Monday, March 7th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF format using LaTeX. Your solution to Part B should be a valid Python file. A template for writing up solutions in LaTeX is available on the course website. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 3-1. [25 points] **Recurrences**

Solve the following recurrences:

(a) [5 points] $T(n) = 2T(\frac{n}{2}) + n + \log n$

Solution: Using the master method, $a = 2, b = 2, f(n) = n + \log n = \Theta(n)$. This is case 2 of the master method since $f(n) = \Theta(n^{log_2 2} \log^0 n)$, so $T(n) = \Theta(n \log n)$

(b) [5 points] $T(n) = 3T(\frac{n}{4}) + \sqrt{n}$

Solution: Using the master method, $a = 3, b = 4, f(n) = \sqrt{n} = \Theta(n^{0.5})$. This is case 1 of the master method since $f(n) = \Theta(n^{log_4 3-\epsilon})$, so $T(n) = \Theta(n^{log_4 3})$

(c) [5 points] $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n^2$ (*Hint:* Calculate upper and lower bounds on $T(n)$ by converting the problem into recurrences you can solve using the Master Theorem.)

Solution: Using the fact that $T(n/2) > T(n/4)$, we can set an upper and lower bound on what $T(n)$ is using the master method. $T(n) < 2T(n/2) + n^2$ and $T(n) > 2T(n/4) + n^2$. In both cases, $f(n) = \Theta(n^{log_b a+\epsilon})$. Using case 3 of the master method, we can see that $T(n)$ is upper bounded and lower bounded by $\Theta(n^2)$. Thus, $T(n) = \Theta(n^2)$.

Merge sort typically divides an array in half, forming two recursive subproblems, recursively sorts the two subproblems, and then merges the two solutions. Consider the ***c-merge-sort*** algorithm, which divides the array into c (roughly) equal-length parts, recursively sorts these c subproblems, and then merges the c solutions. To merge c sorted lists, the algorithm uses a min-heap initialized with the first element of each list, and repeatedly removes the minimum element from the heap, appends that minimum element to the merged list, and then inserts the minimum element's successor (from its source list) into the heap, until all the elements are appended to the merged list.

(d) [5 points] Write the recurrence relation for c-merge-sort on a list of n integers. Solve it to determine the running time for some constant c.

Solution: ***c-merge-sort*** would split the original n size problem into c subproblems of n/c size. Combining the solutions to the subproblems involves making a min-heap of c elements ($O(c)$ time). Then for each of the n elements to merge, we take the heap and extract the minimum ($O(1)$ time), replace the minimum element with its source list's successor ($O(1)$ time), and then heapify the heap ($O(\log c)$ time). Since c is a constant, heapifying's runtime of $O(\log c)$ is equivalent to $O(1)$. Thus, it takes $O(1)$ for each of the n elements to merge and consequently $O(n)$ total for each merge. The total cost of dividing and combining is $O(n)$, giving us the recurrence

$$T(n) = cT(n/c) + O(n)$$

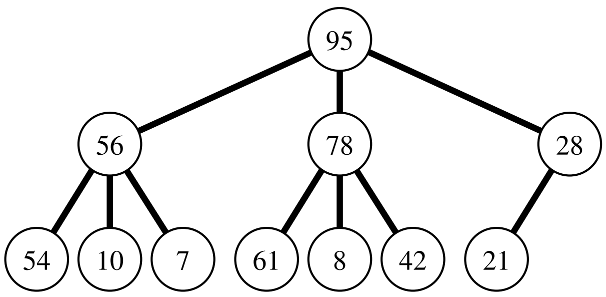
Which by case 2 of the master method will give us a runtime of $O(n \log n)$ regardless of what constant c is.

(e) [5 points] What happens to the running time if c is not a constant, but rather some function of n?

Solution: In this case, we will divide each problem into $c(n)$ subproblems. The runtime of combining the solutions has the bottlenecks of making a min-heap of c elements ($O(c(n))$ time) and heapifying the heap ($O(\log c(n))$ time) n times. Combining the solutions will take $O(c(n) + n \log c(n))$ time, which is asymptotically larger than $O(n)$ when $c(n) = O(1)$. The behavior that each problem gets split into the same number of equally sized subproblems at each recursion is maintained. Since the cost of combining of the solutions is asymptotically larger, the total runtime will be larger when we change c from a constant to a function of n.

Problem 3-2. [25 points] ***d*-ary Heaps**

In class, we've seen binary heaps, where each node has at most two children. A d-ary heap is a heap in which each non-leaf node (except perhaps one) has exactly d children. For example, this is a 3-ary (max) heap:



(a) [2 points] Suppose that we implement a d-ary heap using an array A, similarly to how we implement binary heaps. That is, the root is in $A[0]$, its children are in $A[1 \ldots d]$, and so on. How do we implement the PARENT(i) function, which computes the index of the parent of the ith node, for a d-ary heap?

Solution: Assuming your heap is 0-indexed, $\text{PARENT}(i) = \left\lfloor \frac{i-1}{d} \right\rfloor$. It is equivalent to $\left\lceil \frac{i}{d} \right\rceil - 1$. If your heap is 1-indexed, $\text{PARENT}(i) = \left\lfloor \frac{i-2}{d} \right\rfloor + 1$.

(b) [2 points] Now that there might be more than two children, LEFT and RIGHT are no longer sufficient. How do we implement the CHILD(i, k) function, which computes the index of the kth child of the ith node? ($0 \leq k < d$)

Solution: Assuming your heap is 0-indexed, $\text{CHILD}(i, k) = di + k + 1$. If your heap is 1-indexed, $\text{CHILD}(i, k) = d(i-1) + k + 2$

(c) [5 points] Express, in asymptotic notation, the height of a d-ary heap containing n elements in terms of n and d.

Solution: The height is $\Theta(\log_d n)$.

(d) [5 points] Give the asymptotic running times (in terms of n and d) of the HEAPIFY and INCREASE-KEY operations for a d-ary heap containing n elements.

Solution: HEAPIFY runs in $\Theta(d \log_d n)$ time, since it does d element comparisons at each level of the heap. INCREASE-KEY runs in $\Theta(\log_d n)$, since it does 1 comparison at each level of the heap.

(e) [6 points] Let's suppose that when we build our d-ary heap, we choose d based on the size of the input array, n. What is the height of the resulting heap (in terms of n) if we choose $d = \Theta(1)$? What if $d = \Theta(\log n)$? What about $d = \Theta(n)$?
(*Hint:* remember that $\log_d n = \frac{\log n}{\log d}$. This might simplify your expressions a little.)

Solution: Simply plug in the choice of d into the formula for the height h from part (c):

d	*height*
$\Theta(1)$	$\Theta(\log n)$
$\Theta(\log n)$	$\Theta(\frac{\log n}{\log \log n})$
$\Theta(n)$	$\Theta(1)$

(f) [5 points] What are the running times of HEAPIFY and INCREASE-KEY for the three choices of d above? Do the running times increase or decrease when you increase d? If your program calls HEAPIFY and INCREASE-KEY the same number of times, what would be your choice for d and why?

Solution: After plugging in the three choices of d above into the formulas from part (d), the times are:

d	HEAPIFY	INCREASE-KEY
$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
$\Theta(\log n)$	$\Theta(\frac{(\log n)^2}{\log\log n})$	$\Theta(\frac{\log n}{\log\log n})$
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

In general, a larger choice for d increases the running time of HEAPIFY (it is proportional to $\frac{d}{logd}$), but decreases the running time of INCREASE-KEY (it is proportional to $\frac{1}{logd}$). If HEAPIFY and INCREASE-KEY are called the same number of times, their joint running time is proportional to the sum of their individual running times. This sum is asymptotically smallest ($\Theta(logn)$) when $d = \Theta(1)$. (NOTE: $d = 1$ is a special case in which a tree is a linked list and all formulas above where d is the base of a logarithm are invalid. Instead, all operations take $O(n)$ time. This is not efficient, so we exclude this case.)

Part B

Problem 3-3. [50 points] **Pset Scheduling**

Ben Bitdiddle is behind on his problem sets. In fact, he is already late on N different problem sets ($1 \leq N \leq 100,000$). Fortunately for Ben, all of his classes accept late homework with a grade penalty for each day late (unlike 6.006).

Suppose that problem set i, where $1 \leq i \leq N$, has a penalty of P_i points per day late, and takes one full day to complete. There is no limit to the number of penalty points Ben can accrue. (Ben's penalty adjusted score can become negative.) Ben is required to finish each problem set.

You should implement a function `best_score(penalties)` which takes as input the list of daily point penalties P_i and returns the minimum number of penalties points it is possible for Ben to be assigned.

As part of your program, you will need to do some sorting. You should write your own implementation of heap sort for this problem. Your implementation of heap sort should have the signature `heap_sort(list)` where `list` is the list to sort. Your function should sort `list` in descending order using the default Python ordering defined by < and >. This means, for instance, that `heap_sort([5,1,4,0])` should return `[5,4,1,0]`. Using this function specification will allow us to better test your code if you have a bug and give you more partial credit.

Sample Input:

```
[1,5,2,3]
```

Sample Output:

```
21
```

Sample Explanation:

In the sample above, Ben works on pset #2 on the first day, pset #4 the next day, pset #3 the third day, and pset #1 the last day. In total, he accrues $5 \cdot 1 + 3 \cdot 2 + 2 \cdot 3 + 1 \cdot 4 = 21$ penalty points.