Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Piotr Indyk and David Karger

March 2nd, 2010
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.

Part A questions are due **Tuesday, March 16th** at **11:59PM**.

Part B questions are due **Thursday, March 18th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file which runs from the command line.

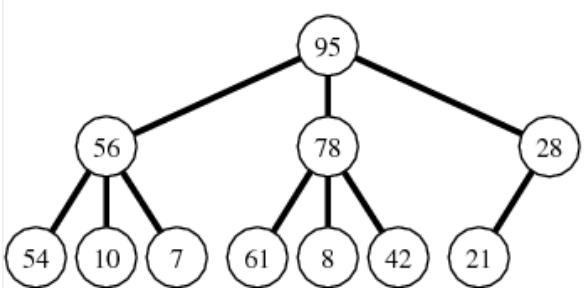
A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the 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 Solutions

1. **(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 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 contained in $A[0]$, its children are contained 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, PARENT(i) $= \left\lfloor \frac{i-1}{d} \right\rfloor$. It is equivalent to $\left\lceil \frac{i}{d} \right\rceil - 1$. (Note that heaps in CLRS are 1-indexed, but the formula is more complex for a 1-indexed heap.)

(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, CHILD(i, k) $= di + k + 1$. (Note that heaps in CLRS are 1-indexed, but the formula is more complex for a 1-indexed heap.)

(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.)

2. **(25 points)** Monotone Priority Queues

   A "monotone priority queue" (MPQ) is a (max) priority queue that only allows monotonically decreasing elements to be extracted. It supports the following operations:

   - `Max`(Q): Returns the key of the most recently extracted node. If no nodes have been extracted, returns ∞. This does not modify the MPQ.
   - `Extract_Max`(Q): Removes and returns the maximum node currently in Q, and updates `Max(`Q`)`. If Q is empty, returns `Max(`Q`)`.
   - `Insert`(Q, x): Inserts x into Q given that $x \leq$ `Max`(Q). If $x >$ `Max`(Q), then the MPQ is not modified.

   When asked to "describe an implementation", you may start with something already proven in class or in the book, and simply describe modifications to that.

   (a) **(10 points)** Describe an implementation of a monotone priority queue that takes $O(m \log m)$ time to perform m operations starting with an empty data structure.

   Solution:

   Simply start with an ordinary max_heap priority queue, and modify it for the new functionality. Keep track of `Max`(Q) separately. On insert, do an extra check to see if the new element is bigger than `Max`(Q). On `Extract_Max`(Q), update the `Max`(Q). All operations still take the same $O(\log m)$ time, because the heap has at most m elements, so in total, at most $O(m \log m)$ work is done.

   (b) **(15 points)** Now suppose that every inserted key x is an integer in the range $[0, k]$ for some fixed integer value k. Describe an implementation of such a monotone priority queue that takes $O(m + k)$ time to perform m total operations.

   Hint: Use an idea from `Counting_Sort`.

   Warning: Be careful about the case when the queue becomes empty.

   Solution:

   Use an array A, where $A[i]$ stores the number of elements with $key = i$ currently in the queue. Again, the max is stored separately. Inserting takes $O(1)$ time (if the key is less than the max, increment $A[key]$). Extracting the max is done by starting at the previous max in the array, and trying every lower slot until one is found that isn't empty. Over the lifetime of the queue, we will only walk down each slot once, and the rest of the work done is constant time.

We need to be careful that when the queue is empty, if we try to extract max then we don't waste time walking down slots in the array, so instead we separately keep track of the size of the array. This way we can figure out in $O(1)$ time whether the queue is empty.

Part B: Due Thursday, March 18th

(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.

Suppose that problem set i, where i is in $\{1 \ldots N\}$, takes D_i days to complete and has a penalty of P_i points per day late. 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.

Help Ben by writing a program to determine the order in which Ben should do his problem sets in order to minimize the total number of penalty points Ben receives on all of his assignments. Your program will be run from the command line. (Note that when a Python program is run from the command line, the test `__name__ == '__main__'` will return `True`.) Your program should read the input from the file `ps3b.in` and write the correct output to `ps3b.out`. After writing the output, your program should exit. The input and output formats are given below. Failure to correctly implement this specification will lose you points.

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:

```
def heap_sort(list):
```

where `list` is the list to sort. Your function should sort `list` in ascending order using the default Python ordering defined by < and >. This means, for instance, that `heap_sort([5,1,4,0])` should return `[0,1,4,5]`. (If you wish to sort more complicated objects than numbers using this function, one approach is to put them in a tuple. Python will then sort the tuples by their first elements, breaking ties by second element etc.) Using this function specification will allow us to better test your code if you have a bug and give you more partial credit.

Input Format (file `ps3b.in`)

Line 1: The single integer N.

Lines $2 \ldots (N+1)$**:** The $(i+1)$-st line describes the Ben's i-th problem set and contains two integers, D_i and P_i, separated with a single space.

Sample Input

```
4
4 1
2 5
1 2
2 3
```

Output Format (file `ps3b.out`)

Line 1: A single integer which is the minimum possible number of penalty points Ben can receive. (Note that a floating point answer, such as `40.0`, will receive fewer points.)

Sample Output

```
40
```

Sample Explanation

In the sample above, Ben first spends 2 days finishing pset #2, accruing 10 penalty points. He then spends 1 day finishing pset #3, accruing 6 penalty points. He then 2 days finishing pset #4, accruing 15 penalty points. He lastly spends 4 days finishing pset #1, accruing 9 penalty points.

Hint: Think about how you might approach this in real life. You should somehow prioritize Ben's problem sets and then sort them according to this priority.