Fundamental Algorithms, Sample Final, Solution Set, Fall 2022

1. (**5 pts.**) State whether the following assertions are True (T) or False (F).

a. $100n^4 = O(n^5)$. True.

b. $2^n + 1000n^2 = \Theta(2^n)$. True.

c. $3^n = \Theta(4^n)$. False.

d. Which of the following two functions is larger (or answer "both" if they are equal). Note: this question is not asking about asymptotics but about actual values.

$$2^{\log n} \qquad\qquad n+1$$

Answer: $n+1$.

e. For the following two functions, which one grows to be the asymptotically larger as n tends to infinity. If they remain within constant factors of each other, answer "both".

$$(\log n)^2 \qquad\qquad n$$

Answer: n.

2. (**5 pts.**) Use the recursion tree method to solve the following recurrence equations. Full credit will be given for a solution that is written as a sum, whether as a closed form or as a sequence of terms (but make sure to show what are the first and last terms and what is the form of the sequence).

$$\begin{aligned} S(0) &= 0 \\ S(n) &= 2^n + 4S(n-1) \qquad n > 0 \end{aligned}$$

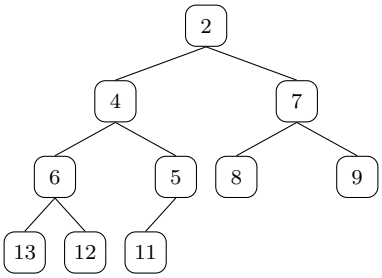
Solution: The non-recursive costs of the recursive calls are as shown in the table below.

problem size	no. of problems	non-rec. cost of one problem	total non-rec. cost of these problems
n	1	2^n	2^n
$n-1$	4	2^{n-1}	$4 \cdot 2^{n-1} = 2^{n+1}$
		...	
$n-i$	4^i	2^{n-i}	$4^i \cdot 2^{n-i} = 2^{n+i}$
		...	
$n-(n-1)$	4^{n-1}	$2^{n-(n-1)}$	$4^{n-1} \cdot 2^{n-(n-1)} = 2^{n+(n-1)}$
$n-n=0$	4^n	0	0

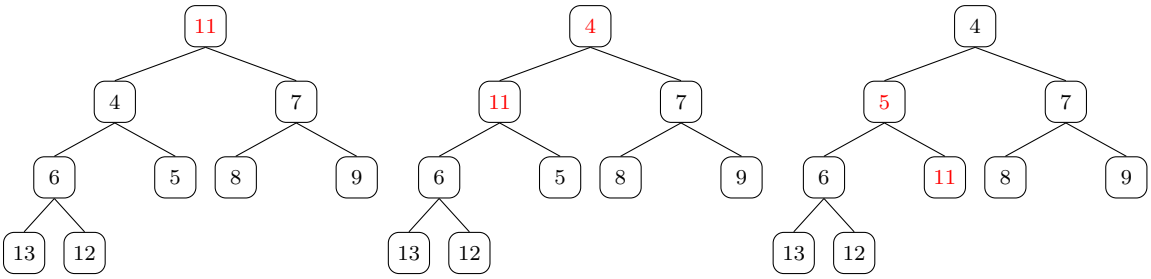
Thus the total cost is given by: $2^n + 2^{n+1} + \ldots + 2^{2n-1} = 2^{2n} - 2^n$.

Check: At $n = 0$ the solution evaluates to 0, which is correct; at $n = 1$ the solution evaluates to 2, which is also correct.

3.(**5 pts.**) Show the effect of the following operations on the given binary heaps.
a. DeleteMin:

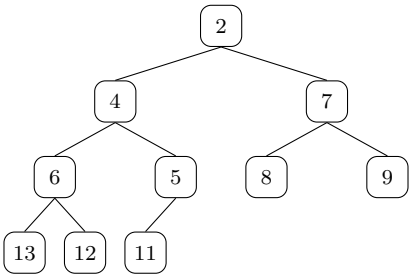


Solution:

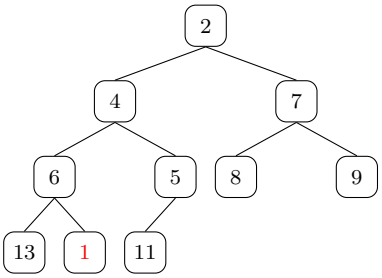


Step 1 Step 2 Step 3

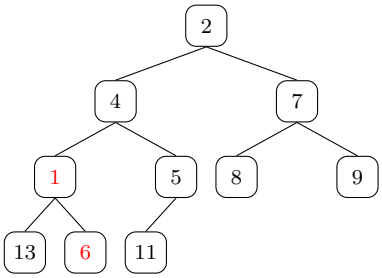
b. ReduceKey(12, 1) (meaning to take the item with key 11 and reduce the value of its key to 1).



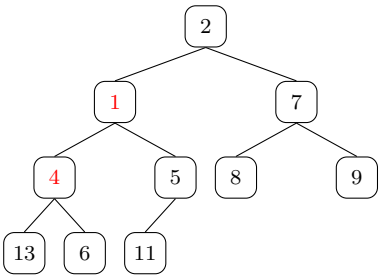
Solution:



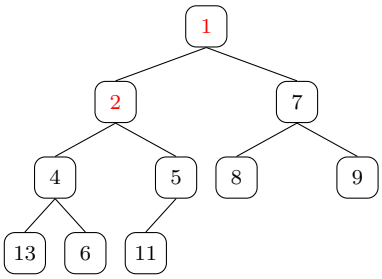
Step 1



Step 2



Step 3



Step 4

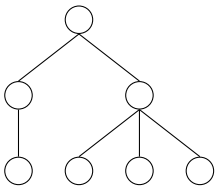
4. (**5 pts.**) What does the statement that the dictionary operations (search, insert, delete) run in expected $O(1)$ time when using a hash table mean?

Solution: It means that the cost of each operation, averaged over all the possible choices of hash function, is $O(1)$.
Comment (this is not part of the answer): However, it does not imply that most of the time, every operation runs in $O(1)$ time.

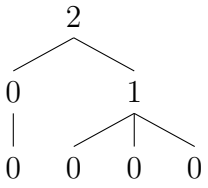
5. (**5 pts.**) Let T be an arbitrary tree so each vertex can have any number of children.

You are to write an algorithm which, for each vertex v, will compute the number of descendants of v including v itself that have two or more children, storing the result in $v.\,\mathrm{mdsc}$.

The following example tree shows the values to be computed.



Input tree



output tree showing
. mdsc field

Please complete the following procedure. Remember to have a driver procedure or to make an initial call.

Solution:

procedure Dsc(v)
  $v.\,\mathrm{mdsc} \leftarrow 0$; $v.\,\mathrm{chdrn} \leftarrow 0$
  for each child w of v **do**
    Dsc(w);
    $v.\,\mathrm{chdrn} \leftarrow v.\,\mathrm{chdrn} + 1$;
    $v.\,\mathrm{mdsc} \leftarrow v.\,\mathrm{mdsc} + w.\,\mathrm{mdsc}$
  end for
  if $v.\,\mathrm{chdrn} \geq 2$ **then** $v.\,\mathrm{mdsc} \leftarrow v.\,\mathrm{mdsc} + 1$
  end if
end procedure

Driver procedure/Initial call (**answer here too**): Dsc(T)

6.(**5 pts.**) Let $G = (V, E)$ be a dag, and let s and t be two vertices in V. Suppose G is given in adjacency list format. Give a linear time algorithm to determine for every vertex $v \in V$ whether it is on a path from s to t. Justify the running time of your algorithm.

Solution: We use an array OnPth$[1:n]$, which is initially all False, to record which vertices are on some s to t path. We then perform a DFS from s, and we update OnPth$[t]$ to True if it is reached, and for other vertices v, we update OnPth$[v]$ to True if OnPth$[w]$ is True for any of v's neighbors w. Pseudo-code is given below.

We use the array Expl$[1:n]$ to identify which vertices the DFS has reached; it is initially all False.

procedure PthDFS(G, v)
  Expl$[v] \leftarrow$ True;
  if $v = t$ **then** OnPth$[t] \leftarrow$ True
  else
    for each edge (v, w) **do**
      if Expl$[w]$ = False **then** Expl$[w] \leftarrow$ True; PthDFS(w)
      end if
      OnPth$[v]$ = OnPth$[v]$ $\vee$ OnPth$[w]$
    end for
  end if
end procedure

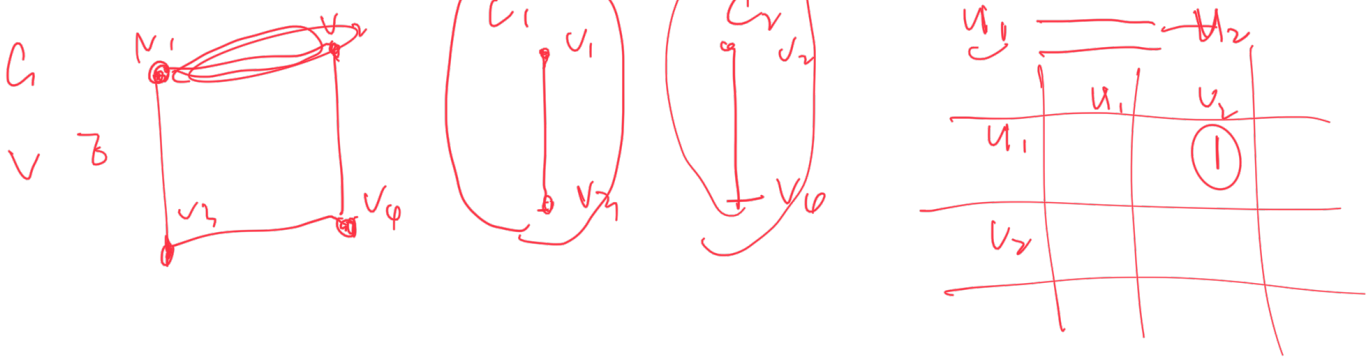
As this modification of DFS still does $O(1)$ work per edge it runs in $O(V + E)$ time.

7. (**5 pts.**) Let $G = (V, E)$ be an undirected graph, and let $F \subset E$ be a subset of the edges in E. Let $H = (V, F)$. Suppose H has connected components $C_1, C_2, \ldots, C_k$. Define the connected components graph $K = (U, D)$ of G w.r.t. F as follows. It has vertices $U = \{u_1, u_2, \ldots, u_k\}$. For each i and j, $1 \leq i < j \leq k$, if there is an edge between a vertex in C_i and a vertex in C_j, then there is an edge $(u_i, u_j) \in D$.

Give an algorithm that takes G and F as input and outputs the graph K. Your algorithm should run in linear time. Briefly explain why it is correct and justify your running time.

Solution: We begin by computing the connected components of the graph (V, F) using DFS (or BFS), and for each component C_i we create a vertex u_i. This takes linear time.

Next, for an edge $(x, y) \in E$, if x and y are in distinct components C_i and C_j then we create an edge from u_i to u_j. The remaining issue is to eliminate duplicate edges. To this end, we sort the edges using radix sort (remember the vertices $u_1, u_2, \ldots, u_k$ can be represented as numbers in the range 1 to k, and $k \leq n$), so this takes $O(m + n)$ time; after the sort duplicates will be adjacent, so the edge set can be reduced to distinct edges with a simple scan over the edges. Now each remaining edge is added to the adjacency lists for its two enpoints in the graph D; this takes linear time in the number of remaining edges, which is at most $O(m)$ time.



8. (**10 pts.**) Let $C = \{a_1, a_2, \ldots, a_n\}$ be a collection of n not necessarily distinct integers. Let $t = \sum_{i=1}^{n} a_i$.

Your task is to give an algorithm to determine whether C can be partitioned into three disjoint collections C_1, C_2, and C_3, such that the items in C_i sum up to t_i, for $i = 1, 2, 3$. The output of your algorithm is one of True or False.

Your algorithm should run in time $O(t^2 n)$. Less efficient solutions will receive partial credit.

You should express your solution as a recursive formula and then explain how to implement it so as to achieve the desired runtime. Remember to justify the runtime.

Solution: We use the following recursive expression $\text{Part}(m, s_1, s_2, s_3)$, for $0 \leq m \leq n$, $0 \leq s_i \leq t_i$, $i = 1, 2, 3$, to record whether the problem with collection $\{a_1, a_2, \ldots, a_m\}$ and targets s_1, s_2, s_3 is solvable.

$$\text{Part}(m, s_1, s_2, s_3) = \begin{cases} \text{True} & m = 0, s_1 = s_2 = s_3 = 0 \\ \text{False} & m = 0,\ s_1 \neq 0 \text{ or } s_2 \neq 0 \text{ or } s_3 \neq 0 \\ \text{Part}(m-1, s_1 - a_m, s_2, s_3) & \\ \vee\, \text{Part}(m-1, s_1, s_2 - a_m, s_3) & m \geq 1 \\ \vee\, \text{Part}(m-1, s_1, s_2, s_3 - a_m) & \end{cases}$$

We implement this recursive formulation using dynamic programming. We perform the following checks to eliminate some cases. If $t_1 + t_2 + t_3 \neq t$ then we return False; this can be checked in $O(n)$ time (the time needed to compute t). Knowing $t_1 + t_2 + t_3 = t$ ensures that $s_1 + s_2 + s_3 = \sum_{i=1}^{m} a_i$ in each recursive call. We do not perform any recursive call with $s_1 < 0$, or $s_2 < 0$, or $s_3 < 0$: again, we simply return the value False. Each remaining recursive subproblem performs $O(1)$ non-recursive work. We now bound the number of these subproblems. There are $n + 1$ possible values for m. There are $t + 1$ possible values for each of s_1, s_2, and s_3. However, we observe that given m, s_1 and s_2, we know that $s_3 = t - s_1 - s_2 - \sum_{i=1}^{m} a_i$, and therefore there are $(m + 1)(t + 1)^2 = O(mt^2)$ distinct subproblems. Thus the overall runtime is $O(mt^2)$.

9. (**10 pts.**) Let $G = (V, E)$ be a directed weighted graph, with two distinct designated vertices s_a and s_b. Suppose there are racing teams A and B located at the distinct vertices s_a and s_b. They are racing to be the first to reach each of the vertices $v \in V$. However, team A is faster: it travels along an edge (u, v) in time $w(u, v)$, while team B takes time $2 \cdot w(u, v)$. To even things up, team B starts at time 0 and team A starts at time $\bar{t} > 0$. In addition, we guarantee that there will be no ties and that team B will not reach s_b by time $\bar{t}$. There is one additional rule: only the first team to reach a vertex can use the outedges from that vertex.

By modifying Dijkstra's algorithm, determine which team reaches each vertex $v \in V$ first (so the answer for each vertex is one of A and B).

The runtime of your algorithm should be at most a constant multiple of the runtime of Dijkstra's algorithm on graphs with n vertices and m edges. Justify your runtime. Also, briefly explain why your algorithm is correct.

Hint: It may be helpful to keep an array Rslt$[1 : n]$, where the entry for vertex v has one of the values A, B, N, according to which team has the current shorter path to v, or neither (if the current distance is infinite).

Solution: We make the following modifications to Dijkstra's algorithm. Rather than start with the pair $(s, 0)$ on the queue as is standard, we have the pairs $(s_b, 0)$ and $(s_a, \bar{t})$ on the queue. Then, whenever we perform a DeleteMin of vertex u, we update the distances (the earliest times at which a vertex can be reached) based on whether team A or team B had been the first to reach u.

More precisely, at all times, Time$[v]$ is the earliest time at which v can be reached from either s_a or s_b using as intermediate vertices only those vertices which have been removed from the priority queue Q, where each team is restricted to using those vertices it reached first. Our modified Dijkstra's algorithm maintains this property, just as in the standard algorithm, thereby justifying correctness.

The resulting pseudo-code is below.

```
procedure DijkstraTm(G, s_a, s_b)
    for each v ∈ V do
        Time[v] ← ∞; Rslt[v] ← N
    end for
    Time[s_a] ← 0; Time[s_b] ← t̄;
    EnQueue(Q, V, Time);
    while Q not empty do
        u ← DeleteMin(Q);
        if Rslt[u] = A then
            for each edge (u, v) do
                new_time ← Time[u] + w(u, v)
                if new_time < Time[v] then
                    Dist[v] ← new_time; Rslt[v] ← A; ReduceKey(Q, v, Time[v])
                end if
            end for
```

```
        else (* Rslt[v] = B *)
            for each edge (u, v) do
                new_time ← Time[u] + 2w(u, v)
                if new_time < Time[v] then
                    Dist[v] ← new_time; Rslt[v] ← B; ReduceKey(Q, v, Time[v])
                end if
            end for
        end if
    end while
end procedure
```

Compared to the standard Dijkstra's algorithm, we have added $O(1)$ work per vertex, and we have added $O(1)$ work per edge. Thus the runtime is at most a constant multiple of the runtime of Dijkstra's algorithm on graphs with n vertices and m edges.