Homework 1

CS 600: Data Structures and Algorithms

Fall 2016

John Berlin

Question 2

We want to devise a function $NEXTPER(n, \sigma)$ that given an integer n and a permutation σ of $\{1, 2, ..., n\}$ outputs the ''next permutation'' of $\{1, 2, ..., n\}$ after σ in the lexicographical order. For example, on an input $(3, \langle 1, 3, 2 \rangle)$, $NEXTPER$ should output $\langle 2, 1, 3 \rangle$ and on an input $(5, \langle 2, 3, 5, 4, 1 \rangle)$, $NEXTPER$ should output $\langle 2, 4, 1, 3, 5 \rangle$.
(a) Give pseudocode for the function $NEXTPER$.
(b) Determine the worst case running time of $NEXTPER$.
(c) Implement your pseudocode for $NEXTPER$ using C/C++/Java.

Answer

a) *Pseudocode* for $NEXTPER(n, \sigma)$

```
1:  function NEXTPER(n, σ)
2:      i, j ← n − 1
3:      σ′ ← σ
4:      while i > 0 ∧ σ′[i − 1] ≥ σ′[i] do
5:          i ← i − 1
6:      if i ≤ 0 then
7:          go to 16
8:      while σ′[j] ≤ σ′[i − 1] do
9:          j ← j − 1
10:     σ′ ← swap(σ′, i − 1, j)
11:     j ← n − 1
12:     while i < j do
13:         σ′ ← swap(σ′, i, j)
14:         i ← i + 1
15:         j ← j − 1
16:     print σ′
```

b) $O(N)$
To borrow the definition from An introduction to The Design and Analysis of Algorithms edition 2 [1] In general, we sean a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$ (and, hence, $a_i > \ldots > a_{i+1}$). Then we find the smallest element in the tail that is larger than a_i, i.e., $min a_j | a_j > a_i, j > i$, and put it in position i; the positions from $i+1$ through n are filled with the elements $a_1, a_{1+1}, \ldots, a_n$ from which the element put in the ith position has been eliminated, in increasing order.

But from the code it is clear to see we go through the permutation at most 3 times and from the definition we are $O(N)$

```java
package jberlin.cs600.odu.edu;

import java.util.Arrays;


public class Homework1 {

    public static void nextPermutation(int n, int[] array) {
        int[] permutation = Arrays.copyOf(array, n);
        int i = n - 1;
        while (i > 0 && permutation[i - 1] >= permutation[i]) {
```

```
                --i;
            }

            if (i <= 0) {
                System.out.println(Arrays.toString(array));
                return;
            }

            int j = n - 1;


            while (permutation[j] <= permutation[i - 1]) {
                j--;
            }

            int temp = array[i - 1];
            array[i - 1] = array[j];
            array[j] = temp;

            j = n - 1;
            while (i < j) {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                i++;
                j--;
            }
            System.out.println(Arrays.toString(array));
        }

        public static void main(String[] args) {
            int[] array = {1,2,3,4,5};
            nextPermutation(array.length, array);

        }
}
```

Listing 1: $NEXTPER(n, \sigma)$

Question 3

Give pseudocode for a function $COUNTPERMS(n, \sigma_1, \sigma_2)$ that given an integer n and two permutations σ_1, σ_2 of $\{1, 2, \ldots, n\}$ outputs the number of permutations which come after σ_1, but before σ_2 in the lexicographical enumeration. Your function should run in time $O(n^k)$ for some k. Find a function $f(n)$ such that the worst-case running time of your function $COUNTPERMS(n, \sigma_1, \sigma_2)$ is $\Theta(f(n))$.
Hint: Note that we only want to count the number of permutations between σ_1 and σ_2 and not necessarily enumerate them.

Answer

Question 4

Prove that for all k, n^k is $O(2^n)$

Answer

First let me restate the problem, please note I denote "such that" as $|$.
Is $f(n) = O(g(n)),\ \forall k$ where $f(n) = n^k,\ g(n) = 2^n$.
To prove this the following properties must hold $\{\exists c > 0,\ \exists n_0 \mid \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$.
For this equation $\lim\limits_{n\to\infty} \dfrac{n^k}{2^n} = 0$ thus $0 < \dfrac{n^k}{2^n} \leq 1$ for some large n.
Leaving us at $\forall n$ and $n \geq n_0$, $0 < n^k \leq c2^n$ holds and $n^k = O(2^n)$ where $c = 1$.
Plugging and chugging with $n = 16, k = 2, c = 1$ we have $0 \leq 16^2 \leq 1 * 2^{16} \to 0 \leq 256 \leq 65536$.
Exactly where we wanted to be, n^k is indeed $O(2^n)$.

Question 5

Consider the following idea for sorting a list of size n: Split the list into $\sqrt{n}$ lists of size $\sqrt{n}$ each, sort each of the smaller list individually and then merge the sorted lists to get a single sorted list.
(a) Let $T(n)$ denote the worst case running time of an algorithm based on this idea. Derive a recurrence relation for $T(n)$. Provide a clear explanation of how you arrived at this recurrence.
(b) Solve the recurrence relation to determine a function $g(n)$ such that $T(n)$ is $\Theta(g(n))$

Answer

Part a:
I define the recurrence relation as $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
n or $f(n)$ is the work needed to combined the lists once sorted.
Split the list of size n into $\sqrt{n}$ lists is $T(n) = \sqrt{n}\ldots$
The $\sqrt{n}$ lists get split into $\sqrt{\sqrt{n}}$ sized lists at $T(n) = \sqrt{n}T(\sqrt{n})$ as $T(n) = \sqrt{n}\ldots$
To better explain consider figure 1 below.

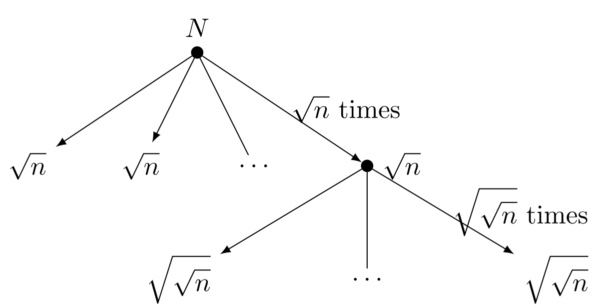

Figure 1: $T(n)$ expansion

Part b:
Since this is a variation of merge sort where instead of $^1/_2$ sized lists which is $2T(^1/_2) + n$ and is $\Theta(n \log n)$ We are using $T(n) = \sqrt{n}T(\sqrt{n}) + n$ so

$$\begin{aligned} T(n) = \quad & \sqrt{n}T(\sqrt{n}) + n \\ & \sqrt{n}\left(\sqrt{n}T(\sqrt{\sqrt{n}}) + \sqrt{n}\right) + n \\ & \dots \end{aligned} \tag{1}$$

But ours is similar too merge sort meaning we gotta be somewhere around $\Theta(n \log n)$ so we gotta find our $\sqrt{n}T(\sqrt{n}) + n \leq cg(n)$. Now I will quickly solve (using class notes from merge sort) merge sort so I can use it to solve $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

$$\begin{aligned} T(n) = \quad & 2T(\frac{n}{2}) + n = 2^1T(\frac{n^1}{2}) \\ & 2^{k-1}T(\frac{n^{k-1}}{2}) + (k-1)n \\ & 2^{k-1}\left(2T(\frac{n^k}{2}) + \frac{n^{k-1}}{2}\right) + (k-1)n \\ & 2^kT(\frac{n^k}{2}) + kn \end{aligned} \tag{2}$$

Last step $T(n) = nT(1) + n \log n \rightarrow n \log n + n$ thus Merge sort is $T(n) = \Theta(n \log n)$. Few...
For our $T(n) = \sqrt{n}T(\sqrt{n}) + n$ it is clear to see $n \log n$ is way too big for our value of Θ. Because of the $\sqrt{n}$ step we a definitely somewhere roughly between $n < us < n \log n$ perhaps $n \log \log n$ or $n\sqrt{\log n}$. I like $n \log \log n$ its a great one plus math. So by playing algebra king we have:

$$\begin{aligned} T(n) = \ & \sqrt{n}T(\sqrt{n}) + n \leq \sqrt{n} * c\sqrt{n}\log(\log(\sqrt{n})) + n \\ & \leq c * n\log(\log(n)) - a * n + n \\ & \leq c * n\log(\log(n)) \end{aligned} \tag{3}$$

By that $T(n) = \Theta(n\log(\log(n)))$.
Revisiting our recursion tree we can see our depth L satisfies $n^{2-L} = 2$
($\sqrt{2}$ does not allow for us to get too 1 thus 2) so by that its clear to see $L = \log(\log(n))$ further solidifying $T(n) = \Theta(n\log(\log(n))$

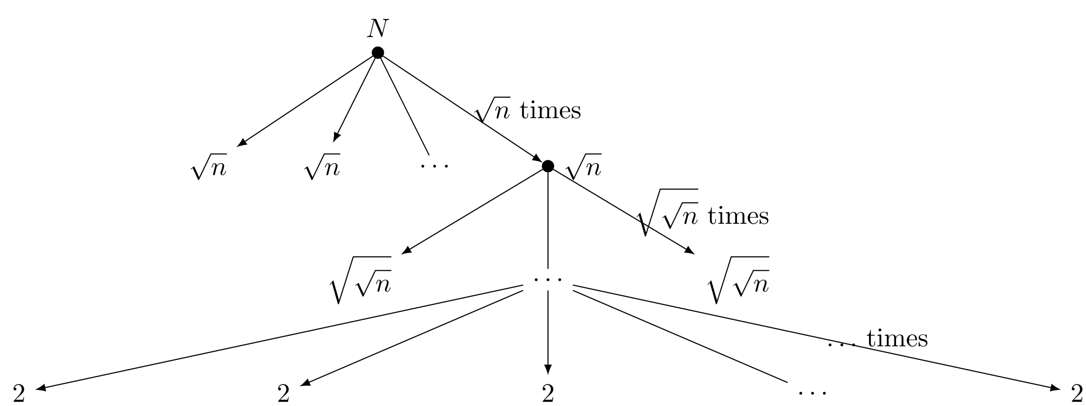

Figure 2: Fuller $T(n)$ expansion

References

[1] Anany V. Levitin, Introduction to the Design and Analysis of Algorithms (2nd Edition), 2006, Addison-Wesley, Reading, MA