Oliver Tonnesen
V00885732

Assignment 2

CSC 425
October 16, 2020

1

Given two sorted arrays, *left* and *right*, we can count the number of significant inversions between the two as follows:

```
function Count-Sig-Invs(left, right)
    i, j, s ← 0, 0, 0
    while i < |left| and j < |right| do
        if left[i] > 2 · right[j] then
            s ← s + |left| − i
            j ← j + 1
        else
            i ← i + 1
    return s
```

Since the lists are sorted, if a_i in the left list and a_j in the right list form a significant inversion (that is, $a_i > 2a_j$), then since each of $a_{i+1}, \ldots, a_{|left|}$ are larger than a_i, each of them also forms a significant inversion with a_j. If (a_i, a_j) is a significant inversion, then we count all of $(a_i, a_j), \ldots, (a_{|left|}, a_j)$ at once, the nchoose a larger j, and thus a larger a_j to see if it is still forms a significant inversion with a_i. If (a_i, a_j) is not a significant inversion, then we choose a larger i, and thus a larger a_i to see if it forms a significant inversion with a_j.

We can plug the above function into a the merge step of a regular implementation of merge sort, roughly as follows:

```
function Merge-Sort-Count-Invs(list)
    if |list| = 0 then
        return 0
    Split list into left and right.
    invs_left  ← Merge-Sort-Count-Invs(left)
    invs_right ← Merge-Sort-Count-Invs(right)
    invs_cross ← Merge(left, right, list)
    return invs_left + invs_right + invs_cross
function Merge(left, right, list)
    invs ← Count-Sig-Invs(left, right)
    Continue merge sort as normal, merging left and right into list in place.
    return invs
```

2

The only vertices connected to a given vertex v are its parent and its two children. Our goal is simply to find a vertex whose value is lower than those of its parent and

its children. Consider the following algorithm:

```
function FIND-LOCAL-MIN(root)
    v ← root
    while v not a leaf do
        l, r ← v_left, v_right
        if x_l < x_v then
            v ← l
        else if x_r < x_v then
            v ← r
        else
            return v
```

We claim the above algorithm returns a local minimum, and give the following proof:

Let $v_1, \ldots, v_k$ be the sequence of vertices taken on by the variable v throughout the algorithm. Note that $x_{v_1}, \ldots, x_{v_k}$ is necessarily a decreasing sequence by the nature of how the algorithm selects v_{i+1} given v_i. The loop in the algorithm only exits when v is either a local minimum or a leaf. If v is a local minimum, then we're already done, so suppose v_k is a leaf. The only vertex joined to a leaf is that vertex's parent, in this case v_{k-1}. Recall that $x_{v_1}, \ldots, x_{v_k}$ is a strictly decreasing sequence, so $x_{v_{k-1}} > x_{v_k}$. In other words, v_k is a local minimum, as desired.

3

Our goal is to find $i < j$ such that $p(j) - p(i)$ is maximized.

We can solve this problem using divide and conquer. If the list has length 2, then we either return $i = 0$ and $j = 1$ if the price is lower on day i than it is on day j, and $i = -1$ and $j = -1$ otherwise. If the list has length greater than 2, then we begin by dividing the list into a left and right sublist. We recursively find the two days giving the largest profit in either list, then we find the two days giving the largest profit when buying on a day in the left list and selling on a day in the right list. For the latter step, we can simply perform a linear search on the left and right sublists to find the lowest and highest prices, respectively. The difference in price between these two days is the largest profit between the two lists.

The following pseudocode corresponds to the above:

---

function FIND-DELTA($list[, \ offset = 0]$)
  if $j - i = 1$ **then**
    if $list[i] < list[j]$ **then**
      return i, j
    else
      return $-1, -1$
  $left, right \leftarrow list$
  $mid \leftarrow |list|/2$
  $\delta_{left}, i_{left}, j_{left} \leftarrow$ FIND-DELTA($left$, $offset$)
  $\delta_{right}, i_{right}, j_{right} \leftarrow$ FIND-DELTA($right$, $offset + mid$)
  $\delta_{cross}, i_{cross}, j_{cross} \leftarrow$ MERGE($left$, $right$, $offset$, mid)
  return δ_x, i_x, j_x corresponding to $\max\{\delta_{left}, \delta_{right}, \delta_{cross}\}$
function MERGE($left$, $right$, $offset$, mid)
  $i \ \leftarrow$ index of smallest element in $left +$ $offset$
  $j \ \leftarrow$ index of largest element in $right +$ $offset +$ mid
  return $list[j] - list[i], i, j$

The MERGE step runs in $O(n)$ time, and is called $O(\log n)$ times, so the above algorithm runs in $O(n \log n)$ time, as desired.

4

We give a solution using dynamic programming. Given an index i, our sub-problem is to find the maximum weight independent set only using vertices from $v_1, \ldots, v_i$.

At index i, we have two options for $OPT(i)$, either:

1. v_i is part of the independent set, in which case $OPT(i) = w_i + OPT(i-2)$, or
2. v_i is not part of the independent set, in which case $OPT(i) = OPT(i-1)$.

More concisely,

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} w_i + OPT(i-2) \\ OPT(i-1) \end{cases} & \text{otherwise} \end{cases}$$

Our goal is to find $OPT(n)$, and use it to reconstruct the set of vertices used to get it. We give an algorithm to demonstrate this:

```
function MAX-WT-INDEP-SET(v_1, ..., v_n, w_1, ..., w_n)
    M[0] = 0
    OPT(n, M)
    return RECONSTRUCT(M)
function OPT(n, M)
    if M[i] uninitialized then
        M[i] ← max{OPT(i − 1, M), w_i + OPT(i − 2, M)}
    return M[i]
function RECONSTRUCT(M)
    U ← ∅
    i ← n
    while i > 0 do
        if M[i] = M[i − 1] then
            i ← i − 1
        else
            U ← U ∪ {i}
            i ← i − 2
    return U
```

5

We view the sub-problems as follows:

Find the maximum possible profit when starting on day i with the machine rebooted j days ago.

The base case is when $i = n$, since we never want to reboot on the last day. When $i = n$, the maximum possible profit is simply the minimum of x_i and s_j.

In the case where $i \neq n$, the decision we must make when finding $OPT(i, j)$ is whether or not to reboot the machine. If we do reboot it, then our profit will be $OPT(i+1, 1)$. If we do not reboot it, then our profit will be $\min\{x_i, s_j\} + OPT(i+1, j+1)$

This gives us the following Bellman equation:

$$OPT(i,j) = \begin{cases} \min\{x_i, s_j\} & \text{if } i = n \\ \max \begin{cases} OPT(i+1,1) \\ \min\{x_i, s_j\} + OPT(i+1, j+1) \end{cases} & \text{otherwise} \end{cases}$$

Given this, we can easily construct a dynamic programming algorithm solving the problem:

---

function MAX-PROFIT($x_1, \ldots, x_n, s_1, \ldots, s_n$)
  for $j = 1, \ldots, n$ **do**
    $M[n][j] \leftarrow \min\{x_n, s_j\}$
  OPT$(1, 1, M)$
  return RECONSTRUCT(M)

function OPT(i, j, M)
  if $M[i][j]$ uninitialized **then**
    $M[i][j] \leftarrow \max\{\text{OPT}(i+1, 1, M), \text{OPT}(i+1, j+1, M)\}$
  return $M[i][j]$

function RECONSTRUCT(M)
  $U \leftarrow \emptyset$
  $i \leftarrow 1$
  $j \leftarrow 1$
  while $i \neq n$ **do**
    if $M[i][j] = M[i+1][1]$ **then**
      $i \leftarrow i + 1$
      $j \leftarrow 1$
      $U \leftarrow U \cup \{i\}$
    else
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
  return U
