DDA6050 - Assignment 1

Yohandi [SID: 120040025]

1. (a). The idea is simply to run the quick sort algorithm and check the index of it if it is being the median. We are looking for the $\frac{n+1}{2}$-th smallest number if n is odd (or the average of the $\frac{n}{2}$-th and $(\frac{n}{2}+1)$-th if n is even). Suppose the current pivot is the p-th smallest in the search space and k is the index that we are looking for:

   - If $p = k$, then we may return the value as we are done.
   - If $p < k$, then we should trace the element to the left side of array.
   - If $p > k$, then we should trace the element to the right side of array.

   C++ code is as follows:

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

using namespace std;

int n;
vector<double> v;

mt19937 rng(time(NULL));

int partition(int l, int r) {
  int pivotIndex = l + rng() % (r - l + 1);
  int pivot = v[pivotIndex];

  swap(v[pivotIndex], v[r]);

  int newIndex = l;
  for (int i = l; i < r; ++i) {
    if (v[i] <= pivot) {
      swap(v[i], v[newIndex]);
      newIndex++;
    }
  }

  swap(v[newIndex], v[r]);
  return newIndex;
}

double kthSmallest(int l, int r, int k) {
  if (l == r) {
    return v[l];
  }

  int p = partition(l, r);

  if (p == k) {
    return v[k];
  } else if (p < k) {
    return kthSmallest(p + 1, r, k);
  } else {
    return kthSmallest(l, p - 1, k);
  }
}

double kthSmallest(int k) { return kthSmallest(0, n - 1, k - 1); } // 0-based

int main() {
  n = 7;
  v = {1, 2, 3, 4, 5, 6, 7};
  cout << kthSmallest((n + 1) / 2) << endl; // 4

  n = 8;
```

```
    v = {1, 2, 3, 4, 5, 6, 7, 8};
    cout << (kthSmallest(n / 2) + kthSmallest(n / 2 + 1)) / 2 << endl; // 4.5
}
```

Console output

```
4
4.5
```

(b). Of course, the best case scenario is encountered when we have the pivot index equals to the index that we are looking for exactly in the first recursion. This requires $\Theta(N)$ (from the partition function) with N as the size of the array. In a case where N is even, the execution time is approximately twice than the case for $N+1$ (where it is odd) due to calling `kthSmallest()` twice; however, the time complexity will still be the same as multiplying by constant will not change it.

In each recursion, the number of removed elements is ranging from 1 to $\frac{n}{2}$, where n is the current size of array. Making a scenario where each recursion removes only one element; then, the worst case scenario is encountered when we find the index that we are looking for in the very last recursion. The time complexity is $\Theta(N+(N-1)+\ldots+1)=\Theta(N^2)$. In a case where N is even, the execution time is approximately twice than the case for $N+1$ (where it is odd) due to calling `kthSmallest()` twice; however, the time complexity will still be the same as multiplying by constant will not change it.

The average case scenario is then computed as follows. As mentioned, the number of removed elements is ranging from 1 to $\frac{n}{2}$ (because if the pivot index equals to 1 or $n-1$, it removes one element in the array; and so on, for 2 or $n-2$, $\ldots$). The average number of removed elements is about $\frac{n}{4}$ (it is $\geq \frac{n}{4}$ if both the left side and the right side of the index that we are looking for are not equal) with n as the current size of array. Then, the time complexity is $\Theta(N+\frac{3}{4}N+\frac{3}{4}^2N+\ldots+1)=\Theta(N)$. In a case where N is even, the execution time is approximately twice than the case for $N+1$ (where it is odd) due to calling `kthSmallest()` twice; however, the time complexity will still be the same as multiplying by constant will not change it.

Auxiliary space takes about $\mathcal{O}(N)$ space complexity. The reason is: in the worst case scenario, only one element is removed in each recursion, making the space stack requires for N recursions. However, as the array is passed by reference, we don't need to worry about the linear space. Only few variables which are considered as $O(1)$ per recursion. Same goes for a case where N is even.

2. $T(n)=aT(\frac{n}{b})+f(n)$
   $T(1)=c$,

where $a \geq 1, b \geq 2, c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then:

- $T(n) = \Theta(n^d)$ if $a < b^d$
- $T(n) = \Theta(n^d \log n)$ if $a = b^d$
- $T(n) = \Theta(n^{\log_b(a)})$ if $a > b^d$

(a). $T(n) = 3T(\frac{n}{2}) + n^2$

We have $a = 3$, $b = 2$, $d = 2$. Since $a < b^d$, $T(n) = \Theta(n^d) = \Theta(n^2)$.

(b). $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$

Let $n = 2^m$, expansion of $T(n)$ is given by:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \frac{n}{\log n} \\
&= 4T(\frac{n}{4}) + \frac{n}{\log \frac{n}{2}} + \frac{n}{\log n} \\
&= 8T(\frac{n}{8}) + \frac{n}{\log \frac{n}{4}} + \frac{n}{\log \frac{n}{2}} + \frac{n}{\log n} \\
&\vdots \\
&= nT(1) + \sum_{k=0}^{m-1} \frac{n}{\log \frac{n}{2^k}} \\
&= n(c + \sum_{k=0}^{m-1} \frac{1}{m-k}) \\
&= n(c + \sum_{k=1}^{m} \frac{1}{k}) \\
&= n(c + \sum_{k=1}^{\log(n)} \frac{1}{k})
\end{aligned}$$

It is clear that:

$$T(n) < n(c + \int_{k=1}^{\log(n)} \frac{2}{k} dk) < 2cn + 2n \ln(\log(n)) = \mathcal{O}(n \log(\log(n)))$$

and

$$T(n) > n(c + \int_{k=1}^{\log(n)} \frac{1}{k} dk) > n \ln(\log(n)) = \Omega(n \log(\log(n)))$$

These imply that $T(n) = \Theta(n \log(\log(n)))$.

(c). $T(n) = 5T(\frac{n}{4}) + n(\log n)^2$

It is clear that when n is big enough:

$$T(n) < S_1(n) = 5T(\frac{n}{4}) + n^{1.1}$$

We have $a = 5$, $b = 4$, $d = 1.1$. Since $a > b^d$,
$S_1(n) = \Theta(n^{\log_4(5)}) \Rightarrow T(n) = \mathcal{O}(n^{\log_4(5)})$.

$$T(n) > S_2(n) = 5T(\frac{n}{4}) + n$$

We have $a = 5$, $b = 4$, $d = 1$. Since $a > b^d$,
$S_2(n) = \Theta(n^{\log_4(5)}) \Rightarrow T(n) = \Omega(n^{\log_4(5)})$.

These imply that $T(n) = \Theta(n^{\log_4(5)})$.

(d). $T(n) = 3T(\frac{n}{3} - 3) + \frac{n}{3}$

Let $U(n) = T(n - \frac{9}{2})$. Then,
$U(n) = 3T(\frac{n-\frac{9}{2}}{3} - 3) + \frac{n-\frac{9}{2}}{3} = 3T(\frac{n}{3} - \frac{9}{2}) + \frac{n-\frac{9}{2}}{3} = 3U(\frac{n}{3}) + \frac{n-\frac{9}{2}}{3}$. We have $a = 3, b = 3, d = 1$; then, $U(n) = \Theta(n \log n)$.

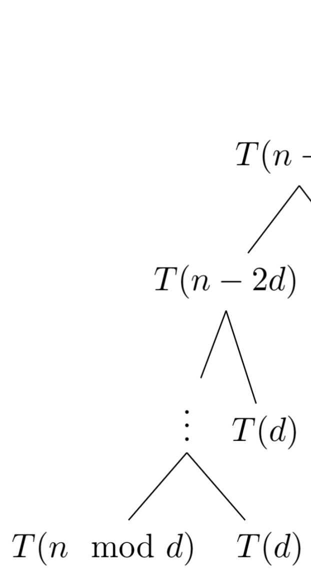
This implies that $T(n) = U(n + \frac{9}{2}) = \Theta((n + \frac{9}{2}) \log(n + \frac{9}{2})) = \Theta(n \log n)$.

(e). $T(n) = 2T(\sqrt{n}) + \log n$

Let $m = \log_2(n) \Rightarrow n = 2^m$ and $S(m) = T(2^m)$. Then,
$T(n) = S(m) = 2S(\frac{m}{2}) + m$.

We have $a = 2$, $b = 2$, $d = 1$. Since $a = b^d$,
$S(m) = \Theta(m \log m) \Rightarrow T(n) = \Theta(\log(n) \log(\log(n)))$.

3. The recursion tree is plotted as follows:



Note that both $T(d)$ and $T(n \mod d)$ are constants; let's denote them as c_1 and c_2, respectively. Then,

$$\begin{aligned}
T(n) &= T(n-d) + kn + c_1 \\
&= T(n-2d) + kn + k(n-d) + 2c_1 \\
&= T(n-3d) + kn + k(n-d) + k(n-2d) + 3c_1 \\
&\vdots \\
&= T(n \mod d) + kn + k(n-d) + k(n-2d) + \ldots + k(n-(\lfloor \frac{n}{d} \rfloor - 1)d) + \lfloor \frac{n}{d} \rfloor c_1 \\
&= kn + k(n-d) + k(n-2d) + \ldots + k(n-(\lfloor \frac{n}{d} \rfloor - 1)d) + \lfloor \frac{n}{d} \rfloor c_1 + c_2 \\
&= \lfloor \frac{n}{d} \rfloor (kn + c_1) - kd(1 + 2 + \ldots + (\lfloor \frac{n}{d} \rfloor - 1)) + c_2 \\
&= \lfloor \frac{n}{d} \rfloor (kn + c_1) - kd\frac{(\lfloor \frac{n}{d} \rfloor - 1) \cdot \lfloor \frac{n}{d} \rfloor}{2} + c_2 \\
&= \Theta(n^2)
\end{aligned}$$

The last equation is obtained accordingly as k, d, c_1, c_2 are all constants.

4. For $T(n) = 4T(\frac{n}{2} + 2) + n$, we guess $T(n) = \mathcal{O}(n^2)$ and want to verify it by substitution method:

   Assume that $T(k) \leq ck^2 - 9ck$ for $k < n$:

$$\begin{aligned}
T(n) &= 4T(\frac{n}{2} + 2) + n \\
&\leq 4c(\frac{n}{2} + 2)^2 - 36c(\frac{n}{2} + 2) + n \\
&\leq cn^2 + 8cn + 16c - 18cn + n - 72c \\
&\leq cn^2 - 10cn + n - 56c
\end{aligned}$$

   When $c \geq 1$, $T(n) \leq cn^2 - 9cn$. This verifies that $T(n) = \mathcal{O}(n^2)$.

5. Suppose we have valid sequences of parentheses: `s` and `t`. Then, `(s)t` is also a valid sequence. Then, the number of ways to add n pairs of parentheses can be found by trying all possible configurations of `s` and `t` such that `s` uses k pairs of parentheses and `t` uses $n-k-1$ pairs of parentheses.

   Let $r_0 = 1$ as there is only one way to make an empty parentheses. Then, $r_n = \sum_{k=0}^{n-1} r_k r_{n-k-1}$. Denote p_n as a set containing all possibilities of parentheses with n pairs. Note that:

   - This solution will not double count because if `s` in p_k is unique and `t` in p_{n-k-1} is unique, so is `(s)` in p_{k+1}, so is `(s)t` in p_n. And since $p_0 = \{ \quad \}$ is unique, so is p_n by induction.
   - This solution will not miss any configuration. For solutions p_k and p_{n-k-1}, we claim that they consist of all possible configurations for their respective sets. Then, let's analyze case by case where `)` is located at:
     - Position 1, which is impossible to be handled
     - Position $2n$, which is guaranteed to be `)`
     - Position 2, which is handled when `s` $\in p_0$

- Position 3, which is handled when `s` $\in p_1$
- Position 4, which is handled when `s` $\in p_1$
- Position 5, which is handled when `s` $\in p_2$
- ...
- Position $2n-1$, which is handled when `s` $\in p_{n-1}$
  Claim for positions of `(` can be analyzed using similar idea. Since $p_0 = \{\}$ does not miss any configuration, so does p_n by induction.

6.

$$\begin{tikzcd}
7 & 5 \arrow[d, red] & 2 \arrow[r, green] \arrow[d, blue] & 3 \arrow[d, green] & 1 \\
3 \arrow[r, blue] & 4 \arrow[d, red] & 1 & 4 & 4 \\
1 & 5 \arrow[r, red] \arrow[d, blue] & 6 \arrow[r, red] \arrow[d, blue] & 7 \arrow[r, green] \arrow[d, red] & 8 \arrow[d, green] \\
3 \arrow[r, blue] & 4 & 5 & 8 \arrow[r, blue] \arrow[d, red] & 9 \\
3 \arrow[r, blue] & 2 & 2 & 7 \arrow[r, red] & 6
\end{tikzcd}$$

Denote $f(r,c)$ as the number of edges that form the maximum length of stairs that can be obtained from submatrix (r,c) to (n,n), we are interested in $f(1,1)$ + 1. The recursion is given by:

$$f(r,c) = \begin{cases} \max(f(r+1,c)+1, f(r,c+1)+1) & r<n, c<n, |a_{r+1,c}-a_{r,c}|=1 \\ & , |a_{r,c}-a_{r,c+1}|=1 \\ \max(f(r+1,c)+1, f(r,c+1)) & r<n, c<n, |a_{r+1,c}-a_{r,c}|=1 \\ \max(f(r+1,c), f(r,c+1)+1) & r<n, c<n, |a_{r,c+1}-a_{r,c}|=1 \\ \max(f(r+1,c), f(r,c+1)) & r<n, c<n \\ f(r+1,c)+1 & r<n, c=n, |a_{r+1,c}-a_{r,c}|=1 \\ f(r+1,c) & r<n, c=n \\ f(r,c+1)+1 & r=n, c<n, |a_{r,c+1}-a_{r,c}|=1 \\ f(r,c+1) & r=n, c<n \\ 0 & r=n, c=n \end{cases}$$

To further simplify the recursion, let's use the Iverson bracket: $[P]$ is 1 if the condition of P is `true`; otherwise, 0. The recursion is as follows:

$$f(r,c) = \begin{cases} 0 & r=n, c=n \\ \max(& \\ [r<n](f(r+1,c)+[|a_{r+1,c}-a_{r,c}|=1]) & \\ , [c<n](f(r,c+1)+[|a_{r,c+1}-a_{r,c}|=1]) & \\ ) & \texttt{otherwise} \end{cases}$$

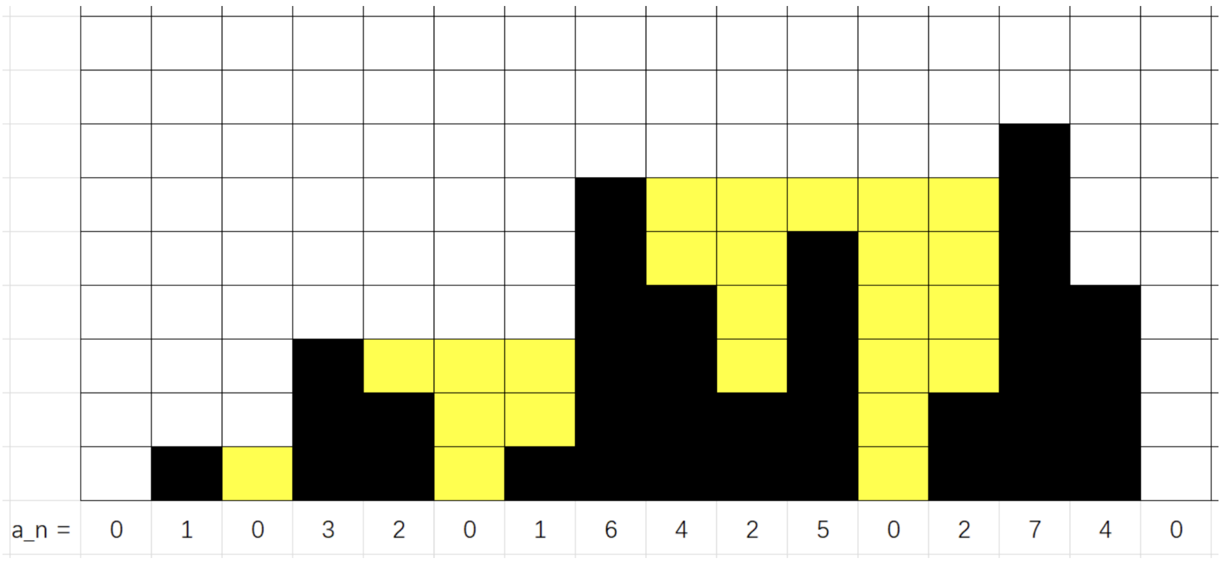
C++ code is as follows:

```
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<vector<int>> grid;

int f(int r, int c) {
  if (r == n && c == n)
    return 0;
  if (r > n || c > n) // outside grid case
    return 0;
  return max((r < n) * (f(r + 1, c) + (abs(grid[r + 1][c] - grid[r][c]) == 1)),
             (c < n) * (f(r, c + 1) + (abs(grid[r][c + 1] - grid[r][c]) == 1))
}

int main() {
  // Sample test case
  n = 5;
  grid = {{-1, -1, -1, -1, -1, -1, -1}, {-1, 7, 5, 2, 3, 1, -1},
          {-1, 3, 4, 1, 4, 4, -1},      {-1, 1, 5, 6, 7, 8, -1},
          {-1, 3, 4, 5, 8, 9, -1},      {-1, 3, 2, 2, 7, 6, -1},
          {-1, -1, -1, -1, -1, -1, -1}};

  cout << f(1, 1) + 1 << endl;
}
```

Console output:

```
8
```

Overall time complexity is $\mathcal{O}(n^2)$.

7. 

For each position i, we can obtain

$$\min(\max_{j=0}^{i} a_j, \max_{j=i}^{n} a_j) - a_i$$

rices. Then, we can maintain two pointers from left side and right side, and two variables denoting most highest cubics on the left side and right side for comparison. We always take the less side as it acts as the capacity for the obtainable rices (minus a_i).

Denote $f(l, r, l_m, r_m)$ as the number of rice that can be obtained from the position l to r with:

- l_m denote the highest stacked cubics from the first $l+1$ stacked cubics (i.e. $a_0, \ldots, a_l$), and
- r_m denote the highest stacked cubics from the last $n-r+1$ stacked cubics (i.e. $a_r, \ldots, a_n$).

We are interested in $f(0, n, 0, 0)$. The recursion is given by:

$$f(l, r, l_m, r_m) = \begin{cases} 0 & l > r \\ (l_m - a_l) + f(l+1, r, \max(l_m, a_{l+1}), r_m) & l_m \leq r_m \\ (r_m - a_r) + f(l, r-1, l_m, \max(r_m, a_{r-1})) & l_m > r_m \end{cases}$$

C++ code is as follows:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<int> a;

int f(int l, int r, int l_m, int r_m) {
  if (l > r)
    return 0;

  if (l_m <= r_m) {
    return (l_m - a[l]) + f(l + 1, r, max(l_m, a[l + 1]), r_m);
  } else {
    return (r_m - a[r]) + f(l, r - 1, l_m, max(r_m, a[r - 1]));
  }
}

int main() {
  // Sample test case
  n = 15;
  a = {0, 1, 0, 3, 2, 0, 1, 6, 4, 2, 5, 0, 2, 7, 4, 0};
  cout << f(0, n, 0, 0) << endl; // 24

  // Asked test case
  n = 15;
  a = {0, 1, 0, 3, 2, 0, 1, 6, 4, 2, 5, 0, 2, 5, 4, 0};
  cout << f(0, n, 0, 0) << endl; // 19
}
```

Console output:

```
24
19
```

Overall time complexity is $\mathcal{O}(n)$.