CSCM12 – Software concepts and efficiency February 24th 2023
Tonicha Crook & Cécilia Pradic

Lab 3: divide-and-conquer, dynamic programming, greedy algorithms and sorting

- Once you are done with the firs two question, it might be helpful to go back to the last question of last week's lab.

1. **Assessing time complexity** For each of the recurrence equations below, give an asymptotic estimate(you may use the master theorem for *most* cases)
   (a) $T(n) = 2T(n/2) + 3n$
   (b) $T(n) = 2T(n/2) + 2n^{\log(n)} + \log(n)$
   (c) $T(n) = 2T(n/4) + \sqrt[3]{n}$
   (d) $T(n) = 2T(n/4) + \sqrt{n}$
   (e) $T(n) = 2T(n/4) + n^2$
   (f) **Challenge:** $T(n) = T(n/3) + 2$

2. **The change problem** Recall the following algorithmic problem:
   - **Input:** A sequence of integers $c_0 = 1 < c_1 < \ldots < c_k$ representing *coin* values and a number a
   - **Output:** An repartition of coins $r_0, \ldots, r_k$ such that giving giving back r_i coins of values c_i for all i yields the desired amount a (i.e. $\sum_i r_i c_i = a$)

   (a) Implement in java the greedy algorithm that we have seen in class: if we try to give back amount a, pick the largest i such that $c_i \leq a$; give back one coin of value c_i and proceed to produce the change of value $a - c_i$.
   (b) What is the complexity of that algorithm? Can you improve it?
   (c) Call an answer to an answer *optimal* if it has the minimal amount of coins ($\sum_i r_i$) amongst all answers.
   Check that if we use the coin system $c_0 = 1, c_1 = 4, c_2 = 5$, there is an amount such that the greedy algorithm above does not return an optimal answer on the corresponding instance.
   (d) **Challenge:** prove that if $2c_i \leq c_{i+1}$ for all $i < k$, then the greedy algorithm returns the optimal answer.
   (e) Using dynamic programming, write a solution that returns an optimal solution for all possible coin systems. What is its complexity? (Hint: you may use a `ArrayList<ArrayList<Integer>>` to compute all of the optimal change allocation for all amonts $\leq n$)

3. **Quicksort** Consider the quicksort algorithm, whose code is recalled below (taken from the file `Sorts.java` you have access to on canvas; `PivotFun` is an interface allowing to pass a pivot-picking function as argument)

```
public static void quickSortInner(PivotFun getPivot,
                                  int[] arr,
                                  int min, int max)
```

```
{
  // If there is at most one element, return immediately
  if(max - min <= 1)
   return;

  // get the position of the pivot according to the pivot policy
  // getPivot, and put that pivot in the middle of the array
  int pivotPos = getPivot.apply(arr,min,max);
  swap(arr, min, pivotPos);

  // reshuffle the array so that we have only elements <= than the pivot
  // before the new pivotPos and only >= elements after
  pivotPos = pivotAround(arr, min, max);

  int postPivot = pivotPos+1;

  // recursively sort above and below the pivot
  quickSortInner(getPivot, arr, min, pivot);
  quickSortInner(getPivot, arr, postPivot, max);
}

public static int pivotAround(int[] arr, int pivotPos, int max)
{
  final int pivot = arr[pivotPos];
  for(int i = pivotPos + 1; i < max; i++)
  {
    if(arr[i] < pivot)
    {
      arr[pivotPos] = arr[i];
      pivotPos++;
      arr[i] = arr[pivotPos];
      arr[pivotPos] = pivot;
    }
  }
  return pivotPos;
}
```

(a) Argue that if `getPivot(arr,min,max)` always returns `min`, then the worst running time on an input of size n is $\Theta(n^2)$.

(b) Argue that if `getPivot(arr,min,max)` returns the median of the collection $\{$`arr[min]`$, \ldots,$ `arr[max-1]`$\}$, then the running time is $\Theta(n \log(n))$ (assuming that all of the numbers in the array are pairwise distinct)

(c) **Challenge:** Assume a distribution of inputs on arrays of all size which is invariant under permutations. Show that quick sort runs on average in time $\mathcal{O}(n \log(n))$

(d) Now assume that the input distribution is no longer invariant under permutations. Do you see a way to get an average running time of $\mathcal{O}(n \log(n))$ using `Random`?

4. **Median selection** The goal of this question is to introduce notions for the

algorithm that picks the median of an array in linear time. **Let's assume for simplicity that all elements of the input array are pairwise distinct**

(a) Write a naive algorithm to compute the median of an array. What is its asymptotic complexity? (don't try to optimize it)

(b) Write a function

```
static int[][] chunk(int[] arr, int chunkSize)
```

that splits an array `arr` into chunks of size `k` and a remaining chunk of size $\leq k$ if there are leftovers. For instance, `chunk({1,2,3,4,5,6,7}, 3)` should return `{{1,2,3},{4,5,6},{7}}`.

(c) Now, for the sake of the discussion, let us fix an odd constant k. Consider the following procedure $\textsc{Select}(A, i)$ (assuming that A has size n)

- If A contains a single element, just return that element.
- Otherwise, first split the array A into chunks of size k.
- Then sort all of the chunks individually.
- Then form an array A' of size $\left\lceil \frac{n}{k} \right\rceil$ consisting of the median of each chunk.
- Call $\textsc{Select}\left(A', \left\lceil \frac{n}{2k} \right\rceil\right)$ and get the median m' of A'
- If $i \leq \frac{n}{2}$, build an array $A_<$ containing:
  - All elements from chunks whose middle element is $< m'$.
  - The first $\frac{k-1}{2}$ elements of the other chunks

  and return the result of $\textsc{Select}(A_<, i)$
- otherwise build an array $A_\geq$ containing
  - All elements from chunks whose middle element is $\geq m'$.
  - The last $\frac{k-1}{2}$ elements of the other chunks

  and return $\textsc{Select}(A_\geq, i - \left\lfloor \frac{n}{2} \right\rfloor)$.

What is the asymptotic running time of the operations of this algorithm if we omit the recursive calls?

(d) To simplify matter, assume from now on that all elements of A are pairwise distinct. Give an upper bound on the number of elements of $A_<$ and $A_\geq$. Deduce that a function $T : \mathbb{N} \to \mathbb{N}$ satisfying the following asymptotically bounds the time complexity of the algorithm.

$$T(n) = T\left(\left\lceil \frac{n}{k} \right\rceil\right) + T\left(\left\lceil \frac{n}{4} \right\rceil\right) + Kn$$

(e) Deduce that for $k = 5$ we have a linear running time, but not $n = 3$ (hint: over/under-approximate the equation and use the master theorem)

(f) **Challenge:** Implement this in Java and interface it with the quicksort implementation given on canvas.