Concept: Types of algorithms

The exposition is based on the following sources, which are all required reading:

1. Corman, Leiserson, Rivest: Chapter 1 and 2.
2. Motwani, Raghavan: Chapter 1.

Concept: Types of algorithms (2)

In this lecture we will discuss different ways to *categorize* classes of algorithms. There is no one ”correct” classification. One should regard the task of categorizing algorithms more as giving them certain attributes.

After discussing how a given algorithm can be labeled (e.g. as a randomized, divide-and-conquer algorithm) we will discuss different techniques to analyze algorithms. Usually the labels with which we categorized an algorithm are quite helpful in choosing the appropriate type of analysis.

Deterministic vs. Randomized

One important (and exclusive) distinction one can make is, whether the algorithm is *deterministic* or *randomized*.

Deterministic algorithms produce on a given input the same results following the same computation steps. Randomized algorithms throw coins during execution. Hence either the order of execution or the result of the algorithm might be different for each run on the same input.

There are subclasses for randomized algorithms. *Monte Carlo* type algorithms and *Las Vegas* type algorithms. A Las Vegas algorithm will always produce the same result on a given input. Randomization will only affect the order of the internal executions.

In the case of Monte Carlo algorithms, the result may might change, even be wrong. However, a Monte Carlo algorithm will produce the correct result with a certain probability.

Deterministic vs. Randomized (2)

So of course the question arises: What are randomized algorithms good for? The computation might change depending on coin throws. Monte Carlo algorithms do not even have to produce the correct result.

Why would that be desirable?

Deterministic vs. Randomized (3)

The answer is twofold:

- Randomized algorithms usually have the effect of perturbing the input. Or put it differently, the input looks random, which makes bad cases very seldom.
- Randomized algorithms are often conceptually very easy to implement. At the same time they are in run time often superior to their deterministic counterparts.

Can you think of an obvious example? We will come to the example later on in more detail.

Offline vs. Online

Another important (and exclusive) distinction one can make is, whether the algorithm is *offline* or *online*.

Online algorithms are algorithms that do not know their input at the beginning. It is given to them *online*, whereas normally algorithms know their input beforehand.

What seems like a minor detail has profound effects on the design of algorithms and on their analysis. Online algorithms are usually analyzed by using the concept of *competitiveness*, that is the worst case factor they take longer compared to the best algorithm with complete information.

Offline vs. Online (2)

One example for an online problem is the *ski problem*.

A skier must decide every day she goes skiing, whether to rent or to buy skis, unless or until she decides to buy them. The skier does not know how many days she can ski, because the whether is unpredictable. Call the number of days she will ski T. The cost to rent skis is 1 unit, while the cost of buying skis is B.

What is the optimal offline algorithm minimizing the worst case cost?

And what would be the *optimal* strategy in the online case?

Exact vs approximate vs. heuristic vs. operational

Usually algorithms have an optimization goal in mind, e.g. compute the shortest path or the alignment or minimal edit distance. *Exact* algorithms aim at computing the optimal solution given such a goal. Often this is quite expensive in terms of run time or memory and hence not possible for large input.

In such cases one tries other strategies. *Approximation* algorithms aim at computing a solution which is for example only a certain, guaranteed factor worse than the optimal solution, that means an algorithm yields a $c-$ *approximation*, if it can guarantee that its solution is never worse than a factor c compared to the optimal solution.

Exact vs approximate vs. heuristic vs. operational (2)

Alternatively, *heuristic* algorithms try to reach the optimal solution without giving a guarantee that they always do. Often it is easy to construct a counter example. A good heuristics is almost always near or at the optimal value.

Finally there are algorithms which do not aim at optimizing an objective function. I call them *operational* since they chain a series of computational operations guided by expert knowledge but not in conjunction with a specific objective function (e.g. ClustalW).

Example: Approximation algorithm

As an example think of the Traveling Saleman Problem with triangle inequality for n cities. This is an NP-hard problem (no polynomial-time algorithm is known).

The following *greedy, deterministic* algorithm yields a $2-$ *approximation* for the TSP with triangle inequality in time $O(n^2)$.

1. Compute a minimum spanning tree T for the complete graph implied by the n cities.
2. Duplicate all edges of T yielding a Eulerian graph T' and then find an Eulerian path in T'.
3. Convert the Eulerian cycle into a Hamiltonian cycle by taking shortcuts.

Can you now argue why this is a $2-$approximation?

Categorization according to main concept

Another way which you have often heard until now is to use the main algorithmic paradigm to categorize an algorithm, such as:

- Simple recursive algorithms
- Backtracking algorithms
- Divide-and-conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Branch-and-bound algorithms
- Brute force algorithms
- and others....

Simple recursive algorithms

A *simple recursive algorithm*

- Solves the base cases directly
- Recurs with a simpler subproblem
- Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

Simple recursive algorithms (2)

Examples are:

- To count the number of elements in a list:
  - If the list is empty, return zero; otherwise,
  - Step past the first element, and count the remaining elements in the list
  - Add one to the result
- To test if a value occurs in a list:
  - If the list is empty, return false; otherwise,
  - If the first thing in the list is the given value, return true; otherwise
  - Step past the first element, and test whether the value occurs in the remainder of the list

Backtracking algorithms

A *backtracking algorithm* is based on a depth-first recursive search. It

- Tests to see if a solution has been found, and if so, returns it; otherwise
- For each choice that can be made at this point,

  - Make that choice
  - Recur
  - If the recursion returns a solution, return it
- If no choices remain, return failure

Backtracking algorithms (2)

For example color a map with no more than four colors:

- color(Country *n*)
  - If all countries have been colored ($n >$ number of countries) return success; otherwise,
  - For each color *c* of four colors,
    * If country *n* is not adjacent to a country that has been colored *c*
      · Color country *n* with color *c*
      · recursivly color country $n+1$
      · If successful, return success

    Return failure (if loop exits)

Divide-and-conquer algorithms

A *divide-and-conquer* algorithm consists of two parts.

- Divide the problem into smaller subproblems of the same type and solve these subproblems recursively
- Combine the solutions to the subproblems into a solution to the original problem

Traditionally, an algorithm is only called divide-and-conquer if it contains two or more recursive calls.

Divide-and-conquer algorithms (2)

Two examples:

- Quicksort:
  - Partition the array into two parts, and quicksort each of the parts
  - No additional work is required to combine the two sorted parts
- Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them

Dynamic programming algorithms

A *dynamic programming algorithm* remembers past results and uses them to find new results. Dynamic programming is generally used for optimization problems in which:

- Multiple solutions exist, need to find the *best* one
- Requires *optimal substructure* and *overlapping subproblem*

- Optimal substructure: Optimal solution contains optimal solutions to subproblems
- Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion

This differs from Divide-and-Conquer, where subproblems generally need not overlap.

Dynamic programming algorithms (2)

There are many examples in bioinformatics. For example:

- Compute an optimal pairwise alignment
  - Optimal substructure: the alignment of two prefixes contains solutions for the optimal alignments of smaller prefixes.
  - Overlapping subproblems: The solution for the optimal alignment of two prefixes can be constructed using the stored solutions of the alignment of three subproblems (in the linear gap model).
- Compute a Viterbi path in an HMM
  - Optimal substructure: the Viterbi path for an input prefix ending in a state of an HMM contains shorter Viterbi paths for smaller parts of the input and other HMM states.
  - Overlapping subproblems: The solution for the Viterbi path for an input prefix ending in a state of an HMM can be constructed using the stored solutions of Viterbi paths for a shorter input prefix and all HMM states.

Greedy algorithms

A *greedy algorithm* sometimes works well for optimization problems. A greedy algorithm works in phases. At each phase:

- You take the best you can get right now, without regard for future consequences
- You hope that by choosing a local optimum at each step, you will end up at a global optimum

This strategy actually often works quite well and for some class of problems it always yields an optimal solution.

Do you know a simple graph problem which is solved greedily to optimality?

Greedy algorithms (2)

Another example would be the following. Suppose you want to count out a certain amount of money, using the fewest possible bills and coins. A greedy algorithm would do this would be to take the largest possible bill or coin that does not overshoot. For example: To make $6.39, you can choose:

- a $5 bill
- a $1 bill, to make $6
- a 25c coin, to make $6.25
- a 10c coin, to make $6.35
- four 1c coins, to make $6.39

For US money, the greedy algorithm always gives the optimum solution (caution: for other money systems not (imagine a currency with units of 1, 7, and 10 and try the algorithm for 15 units).

Branch-and-bound algorithms

Branch-and-bound algorithms are generally used for optimization problems. As the algorithm progresses, a tree of subproblems is formed. The original problem is considered the *root problem*. A method is used to construct an upper and lower bound for a given problem.

At each node, apply the bounding methods.

- If the bounds match, it is deemed a feasible solution to that particular subproblem.
- If bounds do not match, partition the problem represented by that node, and make the two subproblems into children nodes.

Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed.

Branch-and-bound algorithms (2)

An example of a branch-and-bound algorithms would be the folloing for the *Travelling salesman problem (TSP)*. A salesman has to visit each of *n* cities (at least) once each, and wants to minimize total distance travelled.

- Consider the root problem to be the problem of finding the shortest route through a set of cities visiting each city once
- Split the node into two child problems:
  - Shortest route visiting city *A* first
  - Shortest route not visiting city *A* first
- Continue subdividing similarly as the tree grows

Brute force algorithms

A *brute force algorithm* simply tries all possibilities until a satisfactory solution is found. Such an algorithm can be:

- Optimizing: Find the best solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found (Example: Finding the best path for a travelling salesman)
- Satisficing: Stop as soon as a solution is found that is good enough (Example: Finding a travelling salesman path that is within 10% of optimal)

Conclusion

I presented to you many categories with which you can classify or label algorihms. Such a classification gives you a clear understanding about how an algorithm works and an indication how to *analyze* it.

What algorithms do you know? and what labels would they get?

In the following we will talk about how to analyze the different kind of algorithms with appropriate techniques.

Concept: Run time analysis

The exposition is based on the following sources, which are all required reading:

1. Corman, Leiserson, Rivest: Introduction to algorithms, Chapter 18

Concept: Run time analysis (2)

In this section we recall the basic notations for run time analyses and then describe the different concepts of *worst-case run time*, *average case run time*, *expected run time*, *amortized run time*, and the *analysis of competitiveness*.

Lets start by recalling the definitions of the *Landau* symbols ($O, \Omega, \Omega_\infty, \Theta, o, \omega$).

Concept: Run time analysis (3)

$$O(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : g(n) \leq c \cdot f(n)\} \tag{1.1}$$

$$\Omega(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : g(n) \geq c \cdot f(n)\} \tag{1.2}$$

$$\Omega_\infty(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0 : \forall m \in \mathbb{N} : \exists n \in \mathbb{N}, n > m : g(n) \geq c \cdot f(n)\} \tag{1.3}$$

$$\Theta(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : g \in O(f) \text{ and } g \in \Omega(f)\} \tag{1.4}$$

$$o(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\} \tag{1.5}$$

$$\omega(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\} \tag{1.6}$$

In the following we list some commonly used adjectives describing classes of functions. Mind that we use the more common = sign instead of the (correct) $\in$ sign.

Concept: Run time analysis (4)

We say af function f:

- is *constant*, if $f(n) = \Theta(1)$
- grows *logarithmically*, if $f(n) = O(\log n)$
- grows *polylogarithmically*, if $f(n) = O(\log^k(n))$ for a $k \in \mathbb{N}$.
- grows *linearly*, if $f(n) = O(n)$
- grows *quadratically*, if $f(n) = O(n^2)$

Concept: Run time analysis (5)

We say af function f:

- grows *polynomially*, if $f(n) = O(n^k)$, for a $k \in \mathbb{N}$.
- grows *superpolynomially*, if $f(n) = \omega(n^k), \forall k \in \mathbb{N}$.
- grows *subexponentially*, if $f(n) = o(2^{cn}), \forall 0 < c \in \mathbb{R}$.
- grows *exponentially*, if $f(n) = O(2^{cn})$ for a $0 < c \in \mathbb{R}$.

Concept: Run time analysis (6)

After that reminder lets introduce the different run time definitions and explain them using an example.

- worst case analysis: We assume for both, *the input* and *the execution of the algorithm* the worst case. The latter is of course only applicable for non-deterministic algorithms.
- best case analysis: We assume for both, *the input* and *the execution of the algorithm* the best case. The latter is of course only applicable for non-deterministic algorithms.
- average case analysis: We average over all possible *input* the run time of our (deterministic) algorithm.

Concept: Run time analysis (7)

- expected run time analysis: Our algorithm runs depending on the value of some random variables for which we know their distributions. Hence we try to estimate the *expected* run time of the algorithm.
- amortized analysis: Sometimes, an algorithm (usually an operation on a data structure) needs a long time to run, but changes the data structure such that subsequent operations are not costly. A worst case run time analysis would be inappropriate. An *amortized* analysis averages over a series of operations (not over the input).
- competitiveness analysis: For online algorithms we need a new concept of run time analysis. The main concept is to compare the run time an algorithm needs in the worst case (i.e. forall possible inputs) *not* knowing the input, with the runtime of an optimal offline algorithm (which knows the input).

Example: quicksort (8)

The well-known (deterministic) quicksort algorithm for sorting an array chooses a fixed element as its pivot element, lets say w.l.o.g. the first one. It arranges all smaller elements on the left of the pivot, all larger ones on the right and recurses on the two halfs.

- worst case analysis: In the worst case, the left (or the right) half are always empty. Hence the worst case run time is the solution to the recurrence
  $f(n) = (n-1) + f(n-1), f(0) = 0$. Obviously $f = O(n^2)$.
- best case analysis: In the best case, the left and the right half differ in size by at most one. Hence the best case run time is the solution to the recurrence
  $f(n) = (n-1) + 2 * f(n/2), f(0) = 0$. Obviously $f = O(n \log n)$.
- average case analysis: We average over all possible *inputs* the run time of the deterministic quicksort. The result is that quicksort needs $O(n \log n)$ comparisons on average.

Example: quicksort (9)

The dependence on the input and the bad worst case run time of quicksort are worrysome. Quicksort (like many other algorithms) can be made considerably more robust by *randomizing* the algorithm. In randomized quicksort we choose the pivot element randomly using the value of a random variable uniformly distributed over $[1, n]$.

- expected run time analysis: randomized quicksort can be shown to run in *expected* time $O(n \log n)$ with *high probability*. We will discuss such an analysis now in detail (and a similar one using skiplists later in the lecture).

Example: randomized quicksort

Quicksort is a perfect example to demonstrate the power of randomization. First we create a randomized version of quicksort called *RandQS* by simply not choosing the first element as pivot element but a random element of

the sublist we have to sort in each recursive call. Hence RandQS is a Las-Vegas style, randomized, divide-and-conquer algorithm.

By doing so, we created an algorithm whose outcome depends on random choices, or put it differently, on the values of some random variables. Hence we have to analyze its *expected* run time or more particularly the *expected* number of comparisons in an execution of RandQS.

Lets do that.

Example: randomized quicksort (2)

Let $S_{(i)}$ denote the element of *rank i* in the set S which we want to sort.

Now we define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution of RandQS. The variable is 0 otherwise. So obviously the run time of RandQS is proportional to $X = \sum_{i=1}^{n} \sum_{j>i} X_{ij}$. The sum of random variables is itself a random variable.

In the analysis of the expected run time we are hence interested in

$$E(X) = \sum_{i=1}^{n} \sum_{j>i} E(X_{ij}).$$

Let p_{ij} be the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution of RandQS. Then:

$$E(X_{ij}) = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}$$

So we have to concentrate on the question how large p_{ij} is.

Example: randomized quicksort (3)

To analyze this, we view the execution of *RandQS* as a binary tree T, in which each node is labeled with a distinct element $y \in S$. The elements in the left subtree are then all $\leq$ to y, the elements in the right subtree are all $>$ than y.

Observe that the root of the tree is compared to *all* elements in the tree, but there is no comparison between an element of the left subtree with an element of the right subtree. Hence two elements $S_{(i)}$ and $S_{(j)}$ are only compared if one is an ancestor of the other.

An in-order traversal of the tree output the elements of S in sorted order. For the analysis we focus on the level-order traversal. This traversal goes level-by-level and left-to-right and yields a permutation π of the elements of S.

Example: randomized quicksort (4)

Now we make two key observations:

1. $\exists$ a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in π than *any* element of rank between $S_{(i)}$ and $S_{(j)}$. (Why?)
2. Any of the elements $S_{(i)}, S_{i+1}, \ldots, S_{(j)}$ is equally likely to be the first one in the execution of RandQS. Hence the probability of either $S_{(i)}$ or $S_{(j)}$ being the first one is exactly $\frac{2}{j-i+1}$.

From these two observations follows that $p_{ij} = \frac{2}{j-i+1}$.

Example: randomized quicksort (5)

Using this value in our computation of $E(X)$ yields:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j>i} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1} && (1.7)\\
&\leq \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k} && (1.8)\\
&\leq 2\cdot\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k} && (1.9)\\
&= 2\cdot n\cdot H_n. && (1.10)
\end{aligned}$$

where H_n is the $n-th$ harmonic number. given that $H_n \approx \ln n + \Theta(1)$ it follows that the expected run time of RandQS is $O(n \log n)$.

Comparing this to the worst case time of the deterministic Quicksort which is $O(n^2)$ shows how powerful a coin throw can be.

Concept: Run time analysis (6)

As a second concept in run time analysis we have a look at *amortized* analysis. Often (no matter whether for randomized or deterministic algorithms) the worst case analysis is not appropriate, because the worst case *cannot* happen often. No matter what the input is.

If this is the case, then the run time *averaged* over a series of n operations cannot be equal to n times the worst case run time. Indeed, the averaged run time will often be much better.

It is important to note that this is different form the *average run time* analysis. There, one averages over the distribution of inputs. So it could still be that an algorithm, presented with a ”bad” input runs very slowly.

Here, we average over all possible excutions of the algorithms for any given input. To make this distinction clear, this type of analysis is called *amortized analysis*.

Concept: Amortized analysis

Imagine for example a stack. We have the following operations on the stack

- `Pop(S)` pops the top element of the stack and returns it.
- `Push(S,x)` pushed element x on the stack.
- `MultiPop(S,k)` returns at most the top k elements from the stack (it calls `Pop` k times).

Obviously the operations `Pop` and `Push` have worst case time $O(1)$. However the operation `MultiPop` can be linear in the stack size. So if we assume that at most n objects are on the stack, a multipop operation can have worst case cost of $O(n)$. Hence in the worst case a series of n stack operations is bounded by $O(n^2)$.

We will now use this example to illustrate three different techniques to find a more realistic amortized bound for n operations.

Concept: Amortized analysis (2)

The three methods to conduct the analysis which are:

- The aggregate method
- The accounting method
- The potential method

The aggregate method

Here we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. Hence, for this worst case, the average cost, or amortized cost is $T(n)/n$. Note that this method charges the same amortized cost to each operation in the sequence of operations, even if this sequence contains different types of operations.

The other two methods can assign individual amortized costs for each type of operation.

The aggregate method (2)

We argue as follows. In any sequence of n operations on an initially empty stack, each object can be popped at most once for each time it is pushed. Therefore, the number of times `Pop` can be called on a nonempty stack (including calls within `MultiPop`), is at most the number of `Push` operations, which is at most n.

Hence, for any value of n, any sequence of n `Push`, `Pop`, and `MultiPop` operations takes a total of $O(n)$ time. Hence the amortized cost of any operation is $O(n)/n = O(1)$.

The accounting method

In the accounting method we assign differing charges to different operations, with some operations charge more or less than they actually cost. The amount we charge an operation is called its *amortized* cost.

When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*. Credit can then be used to pay later for operations whose amortized cost is less than their actual cost.

The accounting method (2)

One must choose the amortized costs carefully. If we want the analysis with amortized costs to show that in the worst case the average cost per operation is small, then the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. Moreover, this relationship must hold for all sequences of operations, thus the total credit must be nonnegative at all times.

Let us return to our stack example.

The accounting method (3)

The actual costs are:

- `Pop(S)` 1,
- `Push(S,x)` 1,
- `MultiPop(S,k)` $\min(k, s)$, where s is the size of the stack.

Let us assign the following amortized costs.

- `Pop(S)` 0,
- `Push(S,x)` 2,
- `MultiPop(S,k)` 0.

The accounting method (4)

Note that the amortized costs of *all* operations is $O(1)$ and hence the amortized cost of n operations is $O(n)$. Note also that the actual cost of `MultiPop` is variable whereas the amortized cost is constant.

Using the same argument as in the aggregate method it is easy to see that our account is always charged starting with an empty stack. Each `Push` operation pays 2 credits. One for its own cost and one for the cost of popping the element off the stack, either through a normal `Pop` or through a `MultiPop` operation.

Please note that this method can assign individual, different amortized costs to each operation.

The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the *potential method* represents the prepaid work as ”potential energy” or simply ”potential” that can be released to pay for future operations.

The potential is associated with the data structure as a whole rather than with specific objects within the data structure. It works as follows. We start with an initial data structure D_0, on which n operations are performed. For each $i = 1, 2, \ldots, n$ we let c_i be the actual cost of the i-th operation and D_i be the data structure that results after applying the i-th operation.

The potential method (2)

A *potential function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with the data structure D_i. The amortized cost $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

that is, the amortized cost is the actual cost plus the increase in potential due to its operation. Hence the total amortized cost of the n operations is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

The potential method (3)

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost. In practice we do not know how many operations might be performed and therefore guarantee the $\Phi(D_i) \geq \Phi(D_0)$, for all i. It is often convenient to define $\Phi(D_0) = 0$ and then show that all other potentials are non negative.

Lets illustrate the method using our stack example.

The potential method (4)

We define the potential function on the stack as the number of elements it contains. Hence the empty stack D_0 has $\Phi(D_0) = 0$. Since the number of objects on the stack is never negative we have $\Phi(D_i) \geq 0$ for all stacks D_i resulting after the i-th operation.

Let us now compute the amortized costs of the various stack operations. If the i-th operation on a stack containing s objects is a `Push` operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \end{aligned} \tag{1.11}$$

Hence the amortized cost is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

The potential method (5)

If the i-th operation on a stack containing s objects is a `MultiPop(S,k)` then $k' = \min(k, s)$ objects are popped off the stack. The actual cost of the operation is k' and the potential difference is:

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s - k') - s \\ &= -k' \end{aligned} \tag{1.12}$$

Hence the amortized cost is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

A similar result is obtained for `Pop`. This shows that the amortized cost of each operation is $O(1)$ and hence the total amortized cost of a sequence of n operations is $O(n)$.