# Algorithms I – supervision 3

James Wood

February 9, 2015

## 1 Median element

(a) Choose one element of the vector as the pivot element, and compare every other element to it. Put the smaller items in one list, and larger items in another. Let $l$ denote the length of the list of smaller items. If $l = k$, the pivot is the $k$th smallest value. If $l > k$, select its $k$th element by recursion. Otherwise, select the $k - l - 1$th item of the list of greater items.

```
def partition(p, xs):
    (ys, zs) = partition(p, xs[1:])
    if xs[0] ≤ p:
        (xs[0] :: ys, zs)
    else:
        (ys, xs[0] :: zs)

def select(k, xs):
    assert(0 ≤ k)
    assert(k < len(xs))

    (ys, zs) = partition(xs[0], xs[1:])
    l = len(ys)
    if l == k:
        xs[0]
    else if l > k:
        select(k, ys)
    else:
        select(k − l − 1, zs)
```

Just as with quicksort, average time complexity is achieved when the pivot is the median. Starting with a list of $n$ elements, the time taken in the average case is proportional to $T\,n$, where $T\,n = n + T\,(n/2)$

($n$ comparisons and recursive application on half of the list). Approximately, $T\,n = 2 \cdot n$, so the algorithm runs in linear time. However, the worst case occurs when $T_w\,n = n + T_w\,(n-1)$ (e.g: every element is smaller than the pivot and $k = 0$), in which case, the algorithm runs in quadratic time.

An alternative approach that always gives better-than-quadratic time complexity is to sort the whole list (using an $n \cdot \log n$ time algorithm) and then choosing the $k$th item. However, this will usually be worse than linear, and may have a much larger constant factor.

(b) Again, use a quicksort-like approach. After having partitioned the list, if $l = k$, retrun the list of smaller items and if $l+1 = k$, return the list of smaller items, plus the pivot. If $l > k$, pick the smallest $k$ elements from the list of smaller items. If $l + 1 < k$ return all of the smaller items and the pivot, plus the smallest $k - l - 1$ greater items (picked by resursion). Using `partition` from above:

```
def pick(k, xs):
    assert(0 ≤ k)
    assert(k ≤ len(xs))

    (ys, zs) = partition(xs[0], xs[1:])
    l = len(ys)
    if l == k:
        ys
    else if l + 1 == k:
        xs[0] :: ys
    else if l > k:
        pick(k, ys)
    else:
        xs[0] :: ys ++ pick(k − l − 1, zs)
```

The worst case occurs when the pivot is the largest element at each step. In this case, approximately $n$ comparisons have to be done at each step, with $n - k - 1$ steps. This gives asymptotic complexity $O\,(n \cdot (n - k))$. In comparison, the worst case time performance of running `select` $k$ times is $\Theta\,\left(n^2 \cdot k\right)$. For small $k$, `pick`'s worst case time performance is essentially $\Theta\,\left(n^2\right)$, but for large (close to $n$) $k$, this drops to $\Theta\,n$.

(c)

```
def smallest(xs):
    if len(xs) == 1:
        xs[0]
    else:
        min(xs[0], smallest(xs[1:]))
```

This takes $n - 1$ comparisons.

(d)

```
def compareH((x, hx), (y, hy)):
    if x ≤ y:
        (x, y :: hx)
    else:
        (y, x :: hy)

def pairWith(f, xs):
    if len(xs) == 0:
        []
    else if len(xs) == 1:
        [x[0]]
    else:
        f(x[0], x[1]) :: pairWith(f, x[2:])

def smallestH(xs):
    smallestH(pairWith(compareH, xs))

def smallestTwo(xs):
    (x, h) = smallestH(map(fun e: (e, []), xs))
    [x, smallest(h)]
```

In the worst case, `smallestH` makes $f\,n$ comparisons, where $f\,n = \lfloor n/2 \rfloor + f\,\lceil n/2 \rceil$, and `smallest` makes a further $\lceil \log_2 n \rceil - 1$ comparisons. It works out that $f\,n = n - 1$, giving a total of $n + \lceil \log_2 n \rceil - 2$ comparisons.

## 2 Mergesort and insertion sort

(a) The array is split in half, each section is sorted separately, then the two sorted arrays are merged by taking the smallest of the two head elements and adding it to the result array.

In the example, [9,3,6,2,4,1,5] is split into [[9,3,6],[2,4,1,5]], with the parts split to give [[[9],[3,6]],[[2,4],[1,5]]], and finally [[[9],[[3],[6]]],[[[2],[4]],[[1],[5]]].

All of these lists are singletons, and thus sorted. Pairs of lists are then merged, giving [[[9],[3,6]],[[2,4],[1,5]]], then [[3,6,9],[1,2,4,5]]. I will use this to give an example of merging. The heads of the lists, 3 and 1, are compared, then the smallest is moved from its input list to the output list. Hence, the input is changed to [[3,6,9],[2,4,5]] and the output becomes [1]. Then, 3 and 2 are compared, and 2 is moved to the output, giving [[3,6,9],[4,5]] and [1,2]. Doing this a few more times gets us to [[6,9],[]] and [1,2,3,4,5]. The second list is empty, so the first list is appended to the output as-is. The sorted list is [1,2,3,4,5,6,9].

(b) Take $T\,1$ to be constant $k$. For $n > 1$, $T\,n = T\,1 + T\,(n-1) + l \cdot n$, where $l$ is the constant factor of the merging operation. It can be seen that $T\,n = k \cdot (n-1) + l \cdot \operatorname{tr} n$, where $\operatorname{tr} n = \sum_{i \in [1..n]} i$, which is $\frac{n \cdot (n+1)}{2}$. $k \cdot (n-1) \in O\,n$ and $l \cdot \operatorname{tr} n \in O\left(n^2\right)$, so $T\,n \in O\left(n^2\right)$, QED.

(c) (i) Consider the following algorithm:

```
def merge([],        ys      ): ys
    merge(xs,        []      ): xs
    merge(x :: xs, y :: ys):
        if x ≤ y:
            x :: merge(xs, y :: ys)
        else:
            y :: merge(x :: xs, ys)

def split([]            ): ([], [])
    split([x]           ): ([x], [])
    split(x :: y :: zs):
        (xs, ys) = split(zs)
        (x :: xs, y :: ys)

def mergesort(list):
    case list:
        []      : []
        x :: []: [x]
        else    :
            (xs, ys) = split(list)
            merge(mergesort(xs), mergesort(ys))
```

`merge` consumes one item from either of its input lists every time it is called, so runs in time $\Theta\,(m+n)$. Similarly, `split` consumes two elements of its input list every time it is called, so runs in $\Theta\,n$ time. Hence, the time complexity of `mergesort` is proportional to $T\,n$, where $T\,n = k \cdot n + l \cdot n + 2 \cdot T\,(n/2)$. It can be seen that a function in $O\,(n \cdot \log n)$ satisfies this definition.

Given that `merge` and `split` work by consuming the first element of lists and consing it straight onto the output, they only require the modification of pointers, and no copying needs to take place. Since `mergesort` relies on these functions and little else, it also uses constant space.

(ii) The conversion to a linked list would use $\Theta\,n$ time and $\Theta\,n$ space, since each datum has to be copied. This makes no difference to the $O\,(n \cdot \log n)$ time complexity, but puts the space complexity up to $O\,n$, providing no significant benefit (in terms of performance) upon sorting the array directly.

## 3 Matrices

(a) For computing each of the $n \cdot n$ entries, the results of $n$ mutliplications must be summed. Multiplication and addition are constant with respect to the size of the matrix, and summing takes $\Theta\,n$ time, with $n$ being the number of terms. So, computing a single entry takes $\Theta\,n$ time, and thus computing $n \cdot n$ entries takes $\Theta\,\left(n^3\right)$ time.

(b) (i) Divide and conquer.

(ii) Matrix addition takes $\Theta\,\left(n^2\right)$ time, given that it requires $n^2$ scalar additions. The 8 matrix multiplications take $8 \cdot t_{\lceil n/2 \rceil}$ time in total. Adding these together, $t_n = 8 \cdot t_{\lceil n/2 \rceil} + O\,\left(n^2\right)$.

(iii) For large $n$, $t_n \approx 8 \cdot t_{n/2}$. This relation is satisfied by $t_n = 8^{\log_2 n}$, which is $2^{3 \cdot \log_2 n}$, which is $2^{\log_2 n^3}$, which reduces to $n^3$.

(iv) $t_n = 7 \cdot t_{\lceil n/2 \rceil} + O\,\left(n^2\right)$. For large $n$, $t_n \approx 7 \cdot t_{n/2}$. This relation is satisfided by $t_n = 7^{\log_2 n}$, which is $2^{\log_2 n^{\log_2 7}}$, which reduces to $n^{\log_2 7}$.

## 4 Heaps

(a) (i) First, the key is changed to the current value minus the given change. Then, the key is compared to the the key of the parent element. If the changed key is smaller than the parent's key, the two elements are swapped. Then, the same test is done recursively to the modified element in its new position.

(ii) If the argument is negative, do a similar process to the above. Otherwise, compare the current element's key to the keys of its children. If it is larger than exactly one of them, swap it with the one it is larger than and recursively check its new position. If it is larger than both, swap it with the smaller one, and recursively check its new position.

(b) (i) Each node is either a valueless leaf or a node containing a key, a value and $k$ children. The whole heap (and hence all subheaps) must form a complete tree. Any node's key is smaller than the keys of its children. To implement this as an array, the root of the tree is put at position 0, and any child nodes are put in positions $k \cdot p + [1..k]$, where $p$ is the index of the parent.

(ii) Interpret the array as a malformed min-heap. Starting from the end of the array, compare each element to its children. If it is greater than any of its children, swap it with the smallest of its children. Then, recursively do the same check on the current element with its new position.

(iii) In the worst case, an element at level $i$ is moved $h - i$ times, where $h$ is the height of the tree. This gives a worst case time complexity proportional to $\sum_{i \in [0..h)} 2^i \cdot (h - i)$, given that there are $2^i$ elements on level $i$.

$$\begin{aligned}
\sum_{i \in [0..h)} k^i \cdot (h-i) &= k^h \cdot \sum_{i \in [0..h)} k^{i-h} \cdot (h-i) \\
&= k^h \cdot \sum_{i \in [0..h)} (h-i) \cdot \left(\frac{1}{k}\right)^{h-i} \\
&= k^h \cdot \sum_{m \in [1..h]} m \cdot \left(\frac{1}{k}\right)^{m} \\
&\to k^h \cdot \frac{\frac{1}{k}}{\left(1 - \frac{1}{k}\right)^2} \text{ as } h \to \infty \\
&\to k^{h-1} \text{ as } h \to \infty \\
&\to l \cdot n \text{ as } n \to \infty \text{ for some constant } l
\end{aligned}$$

Therefore, the operation runs in linear time.

## 5 String matching

(a) Let $h$ be the hash function. $h\,P = 4$.

$$h\,31 = 9$$
$$h\,14 = 3$$
$$h\,41 = 8$$
$$h\,15 = 4$$
$$h\,59 = 4$$
$$h\,92 = 4$$
$$h\,26 = 4$$

So, there are 3 spurious hash hits.

(b)

```
def match2d(xss, yss):
    lastHashes = Queue()
    hashx = hash(xss)
    wx = width(xss)
    hx = height(xss)
    # Note: `enqueue` and `dequeue` modify their argument

    lastHash = hash(submatrix([:wx], [:hx], yss))
    if hashx == lashHash:
        if xss == submatrix([:wx], [:hx], yss):
            return Just (0, 0)

    for j in [1:width(yss) - wx + 1]:
        hashy = rollRight(lastHash, submatrix([j + wx - 1], [:hx], yss))
        if hashx == hashy:
            if xss == submatrix([j:j + wx], [:hx], yss):
                return Just (j, 0)
        lastHash = hashy
        enqueue(hashy, lastHashes)

    for i in [1:height(yss) - hx + 1]:
        for j in [0:width(yss) - wx + 1]:
            hashy = rollDown(dequeue(lastHashes),
                             submatrix([j:j + wx], [i + hx - 1], yss))
            if hashx == hashy:
                if xss == submatrix([j:j + wx], [i:i + hx], yss):
                    return Just (j, i)
            enqueue(hashy, lastHashes)

    return Nothing
```

## 6 Greedy algorithms

Take the smallest number $x$ and add to the result the interval $[x, 1+x]$. Then, cover the items not covered by this interval recursively. This produces the correct result because any other unit closed interval covering the smallest number of the set will not cover more of the numbers. Hence, it is always among the optimum intervals to choose.