Assignment 1

1. Problem 3-1 on p. 57
2. Problem 3-2 on p. 58
3. Problem 4-5 on p. 86
4. Problem 6-1 on p. 142
5. Problem 7-4 on p. 162
6. Prove correctness (including halting) of SelectionSort (use loop invariants)
7. Provide worst and average case runtime analysis of SelectionSort
8. Provide runspace analysis of SelectionSort

Solutions

Problem 1: Suppose a polynomial in n of degree d has the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

with leading coefficient $a_d > 0$, and let $k \geq 1$ be a constant (not necessarily an integer).

Prove the following:

(a) If $k \geq d$ then $p(n) \leq O(n^k)$.
(b) If $k \leq d$ then $p(n) \geq \Omega(n^k)$.
(c) If $k = d$ then $p(n) = \Theta(n^k)$.

Proof of (c). First we do some algebra:

$$p(n) = \sum_{i=0}^{d} a_i n^i = n^d \sum_{i=0}^{d} a_i n^{i-d} = n^d(\sum_{i=0}^{d-1} a_i n^{i-d} + a_d) = n^d(a_d + q(n))$$

where

$$q(n) = \sum_{i=0}^{d-1} a_i n^{i-d}.$$

Note that $q(n)$ is a sum of terms each of which is a constant multiplied by a power of n, the power being less than or equal to -1. By calculus, the limit of $q(n)$ is zero as $n \to \infty$. So there is an integer n_0 such that

$$|q(n)| < 0.5a_d$$

for all $n \geq n_0$. Using more elementary algebra, and assuming $n \geq n_0$, we can see that

$$n^d(a_d - 0.5a_d) \leq n^d(a_d + q(n)) \leq n^d(a_d + 0.5a_d).$$

Letting $c_1 = 0.5a_d$ and $c_2 = 1.5a_d$, we have:

$$c_1 n^d \leq p(n) \leq c_2 n^d$$

for all $n \geq n_0$. From this last inequality we see that $p(n) = \Theta(n^d)$. QED

We have shown part (c) first, which makes (a) and (b) very straightforward:

Proof of (a). Since $k - d \geq 0$, we have $n^d \leq n^d n^{k-d} = n^k$, so that $O(n^d) \leq O(n^k)$. Using part (c), we have

$$p(n) \leq O(n^d) \leq O(n^k)$$

as required. QED

Proof of (b). Since $d - k \geq 0$, we have $n^d = n^k n^{d-k} \geq n^k$, so that $\Omega(n^d) \geq \Omega(n^k)$. Using part (c), we have

$$p(n) \geq \Omega(n^d) \geq \Omega(n^k)$$

as required. QED

Problem 2: Indicate in the table below whether A is O, Ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants.

	A	B	O	Ω	Θ
$a.$	$\log_k n$	n^k	yes	—	—
$b.$	n^k	c^n	yes	—	—
$c.$	$\sqrt{n}$	$n^{\sin n}$	—	—	—
$d.$	2^n	$2^{n/2}$	—	yes	—
$e.$	$n^{\log c}$	$c^{\log n}$	yes	yes	yes
$f.$	$\log(n!)$	$log(n^n)$	yes	yes	yes

(No entry = `no`.)

c: The essential point is that $\sin n$ takes on values between -1 and 1, getting arbitrarily close to any particular value as n traverses the positive integers. In particular, $\sin n$ gets "within epsilon" of 1 and also "within epsilon" of -1 for sufficiently large n. (This is because n gets arbitrarily near multiples of π for large n.) Therefore $n^{\sin n}$ gets near $n = n^1$ and $1/n = n^{-1}$, thus wandering all over the place between zero and n.

e: Note that A and B are actually equal: Show that $\log A = \log B$ and hence conclude $A = B$.

f: Note that $\log n! = \Theta(n \log n)$ [formula (6)] and that $n \log n = \log n^n$ [property of logarithms] whence $\log n! = \Theta(\log n^n)$.

Problem 3: This problem develops properties of the Fibonacci numbers, which are given by the recurrence $f_n = f_{n-1} + f_{n-2}$ with initial conditions $f_0 = 0$, $f_1 = 1$. Define the *generating function* for the Fibonacci sequence as the formal power series

$$F(z) = \sum_{i=0}^{\infty} f_i z^i$$

.

(a). Show that $F(z) = z + zF(z) + z^2F(z)$.

Proof: Expand the right hand side RHS:

$$z + zF(z) + z^2F(z) = z + z(\sum_{i=0}^{\infty} f_i z^i) + z^2(\sum_{i=0}^{\infty} f_i z^i) = (1 + f_0)z + \sum_{i=2}^{\infty}(f_{i-1} + f_{i-2})z^i$$

Now plug in the initial conditions and recurrence relation, to obtain

$$RHS = z + \sum_{i=2}^{\infty} f_i z^i = \sum_{i=0}^{\infty} f_i z^i$$

which is identical to the left hand side. QED

(b). Show that

$$F(z) = \frac{z}{1 - z - z^2} = \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} = \frac{1}{\sqrt{5}}(\frac{1}{(1 - \phi z)} - \frac{1}{(1 - \hat{\phi} z)})$$

where

$$\phi = \frac{1 + \sqrt{5}}{2} = +1.61803...$$

and

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803...$$

Proof: The first equality is proved by solving the equation proved in (a) for $F(z)$. The other two equalities are proved by calculation, for example:

$$(1 - \phi z)(1 - \hat{\phi} z) = 1 - (\phi + \hat{\phi})z + (\phi\hat{\phi})z^2 = 1 - (1/2 + 1/2)z + ((1 - 5)/4)z^2 = 1 - z - z^2$$

(c). Show that

$$F(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i$$

Proof: Recall the fact:

$$\frac{1}{1-az} = \sum_{i=0}^{\infty} a^i z^i$$

(verify by multiplying both sides by $1-az$). Thus applying part (b) we obtain:

$$F(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{(1-\phi z)} - \frac{1}{(1-\hat{\phi} z)}\right) = \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty} \phi^i z^i - \sum_{i=0}^{\infty} \hat{\phi}^i z^i\right) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) z^i$$

which proves the result.

(d). Prove that $f_i = \phi^i/\sqrt{5}$, rounded to the nearest integer.

Proof: Note that the absolute value of the second base is $|\hat{\phi}| = |(1-\sqrt{5})/2| < 1$ and hence $|\hat{\phi}^i| < 1$ for all i. Applying part (c), we have $f_i = \phi^i/\sqrt{5} - \hat{\phi}^i/\sqrt{5}$, where the second term is smaller than $1/2$ in absolute value. QED

(e). Prove that $f_{i+2} \geq \phi^i$ for $i \geq 0$.

Proof: First note that ϕ and $\hat{\phi}$ are the roots of the quadratic $P(z) = z^2 - z - 1$. In particular, $\phi^2 = \phi + 1$, so that

$$\phi^{i-1} + \phi^{i-2} = \phi^{i-2}(\phi + 1) = \phi^{i-2}\phi^2 = \phi^i,$$

which shows that the sequence ϕ^i satisfies the Fibonacci recursion (but not the initial conditions defining f_i). We prove the assertion by induction on $i \geq 2$.

For the base cases, note that $f_2 = 1 = \phi^0$ and $f_3 = 2 = (1+\sqrt{9})/2 > (1+\sqrt{5})/2 = \phi^1$. The inductive step uses the calculation:

$$f_{i+2} = f_{i+1} + f_i \geq \phi^{i-1} + \phi^{i-2} = \phi^i$$

where the inequality follows from the inductive hypothesis and the equality follows from the fact that ϕ satisfies the recursion. QED

Problem 4: The procedure Build-Max-Heap in section 6.3 can be implemented by repeatedly using Max-Heap-Insert to insert the elements into a heap. Consider the following implementation:

```
template <class I, class P>
void g_build_max_heap (I beg, I end, const P& LessThan)
// pre:  I is a random access iterator class
//       T is the value_type of I
//       P is a predicate class for type T
// post: the specified range of values is a max-heap using LessThan,
{
  if (end - beg <= 1)
    return;
  size_t size = end - beg;
  for (size_t i = 1; i < size; ++i)
    g_push_heap(beg, beg + (i + 1), LessThan);
 }
```

(a) Do `g_build_max_heap` and Build-Max-Heap in the text always create the same heap when run on the same input array? (Prove or disprove.)

Answer: *No.* Run the two algorithms on the array `A = [1,2,3,4,5]` results in `[6,4,5,1,3,2]` and `[6,5,3,4,2,1]`, respectively.

(b) Show that in the worst case `g_build_max_heap` requires $\Theta(n \log n)$ time to build an n-element heap.

The algorithm body consists of a single loop of length n that executes the loop body $n-1$ times. The iteration i of the loop body consists of one call to `g_push_heap` on a range of length $i+1$. We know that `g_push_heap` has worst-case runtime $\Theta(\log i)$. Therefore the algorithm runtime is

$$\Theta(\sum_{i=1}^{n} \log i) = \Theta(n \log n),$$

by application of Equation (4) from the *Formulas* handout.

Problem 5: This problem is about the stack depth (which adds to the runspace requirements of the algorithm) of QuickSort. Here are two versions of QuickSort, for an array `A` with range `[p,r)`. (Note that the notation here is the standard C interpretation of range, that *includes* the begin index and *excludes* the end index. This differs from the text.)

```
void QuickSort(A,p,r)
{
  if (r - p > 1)
  {
    q = partition(A,p,r);
    QuickSort(A,p,q);
    QuickSort(A,q+1,r);
  }
}
```

```
void QuickSort2(A,p,r)
{
  while (r - p > 1)
  {
    q = partition(A,p,r);
    QuickSort2(A,p,q);
    p = q + 1;
  }
}
```

These each call the same version of `Partition` (below). (`QuickSort2` is obtained from `QuickSort` by eliminating tail recursion, a process that can be formalized and accomplished by optimizing compilers.)

```
size_t partition(A,p,r)
{
  i = p;
  for (j = p; j < r-1; ++j)
  {
    if (A[j] <= A[r-1])
    {
      swap(A[i],A[j]);
      ++i;
    }
  }
  swap(A[i],A[r-1]);
  return i;
}
```

(a) Give an informal argument that `QuickSort2` is a sort.

Note that by setting `p = q+1` at the end of the loop body of QuickSort2, the effect is that the next execution of the loop body results in the same process as a call to QuickSort(q+1,r). Thus the two algorithm bodies perform the same sequence of tasks. Since we have already shown that QuickSort is a sort, so also must QuickSort2 be a sort.

(b) Describe a scenario in which the stack depth of `QuickSort2` is $\Theta(n)$ on an n-element array ($n = \mathtt{r} - \mathtt{p}$).

If the input array is sorted, the partition index will always be the largest index in the range, resulting in recursive calls to `QuickSort2(A,p,i)` for `i = r ... p`, a total on n recursive calls.

(c) Modify the code for `QuickSort2` so that the worst-case stack depth is $\Theta(\log n)$, while maintaining $O(n \log n)$ expected runtime of the algoriithm.

Using a randomized version of Partition will at least make the expected stack usage $O(\log n)$, but we would still have the worst case $\Omega(n)$. To ensure that stack space does not grow worse than $\log n$ we can modify the algorithm so that the recursive call is made on the smaller of the two ranges:

```
void QuickSort3(A,p,r)
{
  while (r - p > 1)
  {
    q = partition(A,p,r);
    if (q - p < r - q)
    {
      QuickSort3(A,p,q);
      p = q + 1;
    }
    else
    {
      QuickSort3(A, q+1, r)
      r = q;
    }
  }
}
```

This modification ensures that the recursive call is made on a range that is no larger than 1/2 the size of the previous call and terminates when the range is 1, so there are at most $\log n$ recursive calls.

Problem 6: Prove correctness (including halting) of SelectionSort (use loop invariants)

Here is selection sort (from the lecture notes, with some added loop invariants):

```
SelectionSort ( array A[0..n) )
// pre:
// post: A[0..n) is sorted
{
  for (i = 0; i < n; ++i)
  {
    // Loop Invariant 1: A[0..i) is sorted
    k = i;
    for (j = i; j != n; ++j)
      if (A[j] < A[k])
        k = j;
    // Loop Invariant 2: A[k] is a smallest element of A[i..n)
    Swap (A[i], A[k]);
    // Loop Invariant 3: A[i] is a smallest element of A[i..n)
    // Loop Invariant 4: A[i] is a largest element of A[0..i]
  }
  return;
}
```

Proof of halting: There are two loops, nested, each with length bounded by n.

Proof of correctness: Let $P_1(i)$, $P_2(i)$, $P_3(i)$, $P_4(i)$ denote the four loop invariants shown in the listing. Clearly if we prove $P_1(n)$ we have proved that SelectionSort is a sort.

First consider $P_2(i)$: suppose that s is the index of a smallest element of the range `A[i .. n)`. Then the condition will ensure that k is assigned the value s after the comparison. Therefore $P_2(i)$ is true for all i. Then it is straightforward to deduce $P_3(i)$, just observing the effect of the call to Swap.

Now we can prove $P_1(i)$ and $P_4(i)$ by "double" induction.

Base case: $P_1(0)$ and $P_4(0)$

These are trivially true: an empty range or a range of one element is automatically sorted.

Inductive step part 1: $P_1(i)$ implies $P_4(i)$

By $P_3(i-1)$, `A[i-1]` is a smallest element of `A[i-1..n)`, which implies that `A[i-1] <= A[i]`. $P_4(i)$ follows because `A[0..i)` is sorted.

Inductive step part 2: $P_1(i)$ and $P_4(i)$ imply $P_1(i+1)$

We are given that `A[0..i)` is sorted and must prove that `A[0..i+1)` is sorted after the next iteration of the loop. Invoking $P_1(i)$ and $P_4(i)$ we see that `A[0..i)` is sorted and that `A[i]` is at least as large as any element in `A[0..i)`. Therefore `A[0..i] = A[0..i+1)` is sorted.

Problem 7: Provide worst and average case runtime analysis of `SelectionSort`.

The algorithm body runs exactly the same independent of data, because the lengths of the outer and inner loops are not data dependent. This runtime is

$$\Theta(\sum_{i=0}^{n}\sum_{j=i}^{n}1) = \Theta(\sum_{i=0}^{n}(n-i)) = \Theta(\sum_{i=0}^{n}i) = \Theta(n^2)$$

by Equation (1) of *Formulas*.

Problem 8: Provide runspace analysis of `SelectionSort`.

There are four local variables used in the algorithm body, independent of n. Therefore the runspace is $+\Theta(1)$.